arm

Arm[®] Architecture Reference Manual Supplement, The Scalable Matrix Extension (SME), for Armv9-A

Document number	DDI0616
Document quality	EAC
Document version	B.a
Document confidentiality	Non-confidential
Document build information	5084c744 Tuesday, 22 A just 201 0.40

vright © 2022-2023 Arm Limited or its affiliates. All rights reserved.

This document is now RETIRED. The Arm Architecture Reference Manual for Aprofile architecture (DDI0487) is the definitive reference for this architecture specification.

Arm® Architecture Reference Manual Supplement, The Scalable Matrix Extension (SME), for Armv9-A

Release information

Date	Version	Changes
2023/Aug/18	B.a	• Second release, including SME2 and EAC
2022/Feb/07	A.a	• First release.

Non-Confidential Proprietary Notice

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

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

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

This document may include technical inaccuracies or typographical errors.

TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WIL' ARM 5 LIABL FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIREC' SPECIAL, 'NCIF ANTAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND REGAP' LESS OF THE THE SY OF LIABILITY, ARISING OUT OF ANY USE OF THIS DOCUMENT, EVEN IF ARM HAS L SN A VISED OF THE POSSIBILITY OF SUCH DAMAGES.

This document consists solely of commercial items. You shall be responsible for the stand regret items and regret items to assure that this document or any portion thereof is not exported, directly or indirectly, in violation of such expensions to assure that this document or any portion thereof is not exported, directly or indirectly, in violation of such expensions to assure that this document or any portion thereof is not exported, directly or indirectly, in violation of such expensions to assure that this document or any portion thereof is not exported, directly or indirectly, in violation of such expensions to assure that this document or any portion thereof is not intended to create or refer to any partnership relations to any other company. Arm may make changes to this document at any time and without notice.

This document may be translated into other ' ruages for co enience, and you agree that if there is any conflict between the English version of this document and any unstand, the terms the English version of the Agreement shall prevail.

The Arm corporate logo and words more de with (a) or the registered trademarks or trademarks of Arm Limited (or its affiliates) in the US and/or elsewhere. All rights reserved. Other that and names mentioned in this document may be the trademarks of their respective owners. Please flow Arr trademark usage guidelines at http://www.arm.com/company/policies/trademarks

Copyright © 2022-2023 Uinite for its a^x nates). All rights reserved.

Arm Limited. Com ... a in England.

110 Fulbourn Road, 'amh age, L. d CB1 9NJ.

LES-PRE-20349 versio. 1.0

Product Status

The information in this document is final; that is, it 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.
- Information can be used for software and hardware development.

Arm[®] Architecture Reference Manual Supplement, The Scalable Matrix Extension (SME), for Armv9-A

	Arm® Architecture Reference Manual Supplement, The Scalable Matrix Extension (SME),
	for Armv9-A
	Release information
	Non-Confidential Proprietary Notice
	Product Status
Preface	
	About this supplement
	Conventions
	Typographical conventions
	Numbers
	Pseudocode descriptions
	Asterisks in instruction mnemonics
	Assembler syntax descript ⁱ
	Rules-based writing
	Content item identifie
	Content item rendering
	Content item clas
	Additional reading
	Feedback
	Feedbrace this book
	Progressive terminol commitment
Part A Introduc	stic
Chaptor A1	SMb roducti a
	JME JOUUC' II 1 Neuri la Seeleble Metrix Extension 05
Chapter A2	rebitecture Features and Extensions
enapter n=	A2.1 Extensions and features defined by SME
	A2.2 Changes to existing features and extension requirements
Part B SME Ap	plication Level Programmers' Model
Chapter D1	Application processing modes
	Application processing modes
	DI.1 Overview
	B1.2.1 PSTATE 74
	D1.2.2 FSTATE.ZA
	D1.2.4 IPIDH2_ELU
Chapter B2	Architectural state
•	B2.1 Architectural state summary
	B2.2 SME ZA storage
	B2.2.1 ZA array vector access

		B2.2.2	ZA tile access	37
		B2.2.3	Accessing an 8-bit element ZA tile	38
		B2.2.4	Accessing a 16-bit element ZA tile	39
		B2.2.5	Accessing a 32-bit element ZA tile	40
		B2.2.6	Accessing a 64-bit element ZA tile	41
		B2.2.7	Accessing a 128-bit element ZA tile	42
	B2.3	ZA sto	prage layout	43
		B2.3.1	ZA array vector and tile slice mappings	43
		B2.3.2	Tile mappings	43
		B2.3.3	Horizontal tile slice mappings	44
		B2.3.4	Vertical tile slice mappings	45
		B2.3.5	Mixed horizontal and vertical tile slice mappings	46
	B2.4	SME2	2 Multi-vector operands	48
		B2.4.1	Z multi-vector operands	48
		B2.4.2	ZA multi-slice operands	48
		B2.4.3	ZA multi-vector operands	49
	B2.5	SME2	2 Multi-vector predication	56
	B2.6	SME2	2 Lookup table	58
Chapter B3	Floa	ting-poi	nt behaviors	
•	B3.1	Overv	view	59
	B3.2	Suppo	prted floating-poin' and pes	60
	B3.3	BFloa	t16 behaviors	61
		B3.3.1	Common E pat16 be aviors	61
		B3.3.2	Standard BFIC 16 Jnaviors	61
		B3.3.3	Extend BFloa behaviors	61
	B3.4	Floati	ng-r int bei viors in *reaming SVE mode	63
	B3.5	Floati	r point behave 's targeting the ZA array	64

Part C SME System L vel Prog. mmers' Model

Chapter C1	Sv a	em m	agement	
•	C1.	√er\	/iew	66
		.1.1	le' utification	67
		C. 2	raps and exceptions	67
		C1.1.	Vector lengths	68
		~1.4	Streaming execution priority	70
	C1.2	Proce	ssor behavior	71
		C1.2.1	Exception priorities	71
		C1.2.2	Synchronous Data Abort	73
		C1.2.3	Validity of SME and SVE state	73
		C1.2.4	Streaming execution priority for shared implementations	74
		C1.2.5	Security considerations	76
	C1.3	Chan	ges to existing System registers	77
		C1.3.1	CPACR_EL1	77
		C1.3.2	CPTR_EL2	77
		C1.3.3	CPTR_EL3	77
		C1.3.4	ESR_EL1, ESR_EL2, and ESR_EL3	77
		C1.3.5	HCR_EL2	78
		C1.3.6	HCRX_EL2	78
		C1.3.7	HFGRTR_EL2	79
		C1.3.8	HFGWTR_EL2	79
		C1.3.9	ID_AA64PFR1_EL1	79
		C1.3.10	ID_AA64ZFR0_EL1	79

C1.4	C1.3.11 SCR_EL3 80 C1.3.12 SCTLR_EL1 80 C1.3.13 SCTLR_EL2 80 C1.3.14 ZCR_EL1, ZCR_EL2, and ZCR_EL3 80 SME-specific System registers 81 C1.4.1 ID_AA64SMFR0_EL1 81 C1.4.2 SMCR_EL1 81 C1.4.3 SMCR_EL2 81 C1.4.4 SMCR_EL2 81 C1.4.5 SMIDR_EL1 81 C1.4.5 SMIDR_EL1 81 C1.4.6 SMPRI_EL1 81 C1.4.7 SMDRMAP_EL2 81	
	C1.4.7 SMPRIMAP_EL2	,
Chapter C2 Inte	raction with other A-profile architectur , feature	
• C2.1	Watchpoints	Ļ
	C2.1.1 Reporting watchpoints	Ļ
C2.2	Self-hosted debug	i
C2.3	External debug	I
C2.4	Memory Tagging Extension (N, ⁻)	I
C2.5	Reliability, Availability, and Service ' .y (RAS)	
C2.6	Memory Partitioning an me 'oring 'PAM' 92	
	C2.6.1 MPAMSM_EI	
	C2.6.2 MPAM2_E! 92	
C2.7	Transactional Memo. 5y ision (mu.)	i
C2.8	Memory consil ncy mil	•
Part D SME Instruc	tion Sr.	

Part D SME Instruction Sr .

Chapter D1	SME in truc	tions
-	D1.1 SME	and SME ₂ hta-processing instructions
	D1.1.1	ADD (to vector)
	D1 _	ADD (array accumulators)
	.1.3	A ^r (array results, multiple and single vector)
	D '4	DD (array results, multiple vectors)
	ר. D1. ז	ADDHA
	□1 1.6	ADDSPL
	ד.1.7	ADDSVL
	D1.1.8	ADDVA
	D1.1.9	BFCVT
	D1.1.10	BFCVTN
	D1.1.11	BFDOT (multiple and indexed vector)
	D1.1.12	BFDOT (multiple and single vector)
	D1.1.13	BFDOT (multiple vectors)
	D1.1.14	BFMLAL (multiple and indexed vector)
	D1.1.15	BFMLAL (multiple and single vector)
	D1.1.16	BFMLAL (multiple vectors)
	D1.1.17	BFMLSL (multiple and indexed vector)
	D1.1.18	BFMLSL (multiple and single vector)
	D1.1.19	BFMLSL (multiple vectors)
	D1.1.20	BFMOPA
	D1.1.21	BFMOPS
	D1.1.22	BFVDOT
	D1.1.23	BMOPA 139

D1.1.24	BMOPS 14	41
D1.1.25	CNTP	43
D1.1.26	FADD	45
D1.1.27	FCLAMP	47
D1.1.28	FCVT	49
D1.1.29	FCVTN	50
D1.1.30	FCVTZS	51
D1.1.31	FCVTZU	53
D1.1.32	FDOT (multiple and indexed vector)	55
D1.1.33	FDOT (multiple and single vector)	57
D1.1.34	FDOT (multiple vectors)	59
D1.1.35	FMAX (multiple and single vector)	61
D1.1.36	FMAX (multiple vectors)	63
D1.1.37	FMAXNM (multiple and single vector)	65
D1.1.38	FMAXNM (multiple vectors)	67
D1.1.39	FMIN (multiple and single vector)	69
D1.1.40	FMIN (multiple vectors)	71
D1.1.41	FMINNM (multiple and sing' vector)	73
D1.1.42	FMINNM (multiple vector	75
D1 1 43	FMLA (multiple and inde divect 1	77
D1 1 44	FMLA (multiple and single un)	80
D1 1 45	FMLA (multiple v multiple v multi	82
D1 1 46	FMLAL (multi-) and in aved ver	84
D1.1.40	FMLAL (multiple and in executive for the second sec	87
D1 1 48	FMLAL (multi) v (org)	80
D1.1.40	FMLS (, tiple : 'indexed vector)	03
D1.1.49	F[N] = S(m) lo and add vector	01
D1.1.50	$P = 25 \text{ (multiple vector)} \dots \dots$	94
D1.1.51	VILS (Inullip) (Vectors)	90
D1.1.2	F. SL (multiple of indexed vector)	90
DT 1.53		01
, 1.54		03
		05
D1.1 J		07
0, .57	FM 'PS (widening)	09
1.58		11
D1. 99	FRINIA	13
D1.1.CJ	FRINTM	15
7.61	FRININ	1/
D1.1.62	FRINTP	19
D1.1.63	FSUB	21
D1.1.64	FVDOI	23
D1.1.65	LD1B (scalar plus immediate, consecutive registers)	25
D1.1.66	LD1B (scalar plus scalar, consecutive registers)	27
D1.1.67	LD1B (scalar plus immediate, strided registers)	29
D1.1.68	LD1B (scalar plus scalar, strided registers)	31
D1.1.69	LD1B (scalar plus scalar, tile slice)	33
D1.1.70	LD1D (scalar plus immediate, consecutive registers)	35
D1.1.71	LD1D (scalar plus scalar, consecutive registers)	37
D1.1.72	LD1D (scalar plus immediate, strided registers)	39
D1.1.73	LD1D (scalar plus scalar, strided registers)	41
D1.1.74	LD1D (scalar plus scalar, tile slice)	43
D1.1.75	LD1H (scalar plus immediate, consecutive registers)	45
D1.1.76	LD1H (scalar plus scalar, consecutive registers)	47
D1.1.77	LD1H (scalar plus immediate, strided registers)	49
D1.1.78	LD1H (scalar plus scalar, strided registers)	51

	I D1H (scalar plus scalar tilo slico)	252
D1.1.79		200
D1.1.80		255
D1.1.81	LD1W (scalar plus immediate, consecutive registers)	257
D1.1.82	LD1W (scalar plus scalar, consecutive registers)	259
D1.1.83	LD1W (scalar plus immediate, strided registers)	261
D1.1.84	LD1W (scalar plus scalar, strided registers)	263
D1.1.85	LD1W (scalar plus scalar, tile slice)	265
D1.1.86	LDNT1B (scalar plus immediate, consecutive registers)	267
D1.1.87	LDNT1B (scalar plus scalar, consecutive registers)	269
D1 1 88	I DNT1B (scalar plus immediate strided registers)	271
D1 1 89	L DNT1B (scalar plus scalar strided registers)	273
D1.1.09	LDNT1D (scalar plus scalar, structure registers)	275
D1.1.90	LDNT1D (scalar plus infinediate, consecutive registers)	273
D1.1.91	LDNTTD (scalar plus scalar, consecutive redisters)	277
D1.1.92		2/9
D1.1.93	LDN I1D (scalar plus scalar, strided gisters)	281
D1.1.94	LDNT1H (scalar plus immediate, cc ecutive regis s)	283
D1.1.95	LDNT1H (scalar plus scalar, cr secut, registers)	285
D1.1.96	LDNT1H (scalar plus immeriate, strided inistersi	287
D1.1.97	LDNT1H (scalar plus scr., strider' egister	289
D1.1.98	LDNT1W (scalar plus indiateonsecutive registers)	291
D1.1.99	LDNT1W (scalar plus scale usecutive egisters)	293
D1 1 10	LDNT1W (scala: us medu stric' registers)	295
D1 1 10	LDNT1W (scr. r plus s alar structedisters)	297
D1.1.10	2 LDR (vecto	200
D1.1.10		200
D1.1.10	$\int LDR(2 0) \dots $	201
D1.1.10	$= L \int Q(f_0) = r r r r r r r r r r r r r r r r r r $	202
D1.1.10		304
D1.1.10		306
D1.1.1	L_{14} (two regis s)	307
D1 1.10	3 LUI. (four registers)	309
F .1.10	∃ LUTI4 , `gle)	311
1.1.11 ق) MOV (tile 🗸 vector, two registers)	312
D1.1 .	MOV (tile to vector, four registers)	315
.11: יס	2 MC [*] (array to vector, two registers)	318
1.11:	3 / JV (array to vector, four registers)	319
D1. 11	MOV (tile to vector, single)	320
D1.1	5 MOV (vector to tile, two registers)	323
2.11	6 MOV (vector to tile, four registers)	326
D1.1.11	7 MOV (vector to array, two registers)	329
D1.1.11	B MOV (vector to array, four registers)	330
D1.1.11	A MOV (vector to tile, single)	331
D1 1 12) MOVA (tile to vector two registers)	334
D1 1 12	MOVA (tile to vector, four registers)	337
D1.1.12	MOVA (array to vector, two registers)	340
D1.1.12	2 MOVA (array to vector, two registers)	240
D1.1.12	MOVA (all all to vector, lour registers)	042
D1.1.12	= MOVA (useter te tile, two resisters)	044
D1.1.12	$\Delta MOVA$ (vector to tile, two registers) $\dots \dots \dots$	347
D1.1.12	r NOVA (vector to the, rour registers) $\dots \dots \dots$	350
D1.1.12		353
D1.1.12	3 MOVA (vector to array, tour registers)	355
D1.1.12	OMOVA (vector to tile, single)	357
D1.1.13) MOVT (ZT0 to scalar)	360
D1.1.13	1 MOVT (scalar to ZT0)	361
D1.1.13	2 PEXT (predicate)	362
D1.1.13	3 PEXT (predicate pair)	364

D1.1.134 P	PTRUE		 	366
D1.1.135 F	RDSVL		 	367
D1.1.136 S	CLAMP		 	368
D1.1.137 S	CVTF		 	370
D1.1.138 S	DOT (2-way, multiple and indexed vector)		 	372
D1.1.139 S	DOT (2-way, multiple and single vector)		 	374
D1.1.140 S	DOT (2-way, multiple vectors)		 	376
D1.1.141 S	DOT (4-way, multiple and indexed vector)		 	378
D1.1.142 S	DOT (4-way, multiple and single vector)		 	381
D1.1.143 S	DOT (4-way, multiple vectors)		 	383
D1.1.144 S	5EL		 	385
D1 1 145 S	MAX (multiple and single vector)		 	387
D1 1 146 S	MAX (multiple vectors)		 	389
	MIN (multiple vectors)		 •••	201
D1.1.147 3	MIN (multiple and single vector)		 •••	202
D1.1.140 S	MIN (multiple vectors)		 	292
D1.1.149 3			 	395
D1.1.150 S	MLAL (multiple and single ver br)		 • • •	398
D1.1.151 S	SMLAL (multiple vectors)		 	400
D1.1.152 S	MLALL (multiple and in _xed vec ⁺)		 	402
D1.1.153 S	$MLALL (multiple and s, ever f) \dots \dots$		 	406
D1.1.154 S	MLALL (multiple vectors)		 	409
D1.1.155 S	MLSL (multiple in a lexed stor)		 	412
D1.1.156 S	MLSL (multille and sigle vector		 	415
D1.1.157 S	MLSL (mu he vect		 	417
D1.1.158 S	MLSL' (mu، `م ،d indexed-vector)		 	419
D1.1.159 S	MLSL⊾ nultipi⊾ nd single vector)		 	423
D1.1.160 S	SLL (n. 'tiple v. 'ors)		 	426
D1.1.161	VIOPA (2-wa		 	429
D1.1.1 / S	$\mathcal{P}PA$ (4-way)		 	431
D1.1.163.S	$M_{\rm L} = S(2-way)$		 	434
F 1 164 S	(MOP. 1-way)		 	436
.11.1.165 S	$CVT (1 \circ registers)$		 •••	130
	$OCVT$ (four registers) $\cdots \cdots \cdots$		 •••	400
D1.1 30 3			 •••	440
1 100 5	C VIN		 •••	441
			 	442
DI. IF S			 	443
D1.1.70 S			 	444
.1/1 S	QDMULH (multiple and single vector)		 	445
D1.1.172 S	QDMULH (multiple vectors)		 	447
D1.1.173 S	QRSHR (two registers)		 	449
D1.1.174 S	QRSHR (four registers)		 	450
D1.1.175 S	GQRSHRN		 	452
D1.1.176 S	QRSHRU (two registers)		 	454
D1.1.177 S	QRSHRU (four registers)		 	455
D1.1.178 S	QRSHRUN		 	457
D1.1.179 S	RSHL (multiple and single vector)		 	459
D1.1.180 S	RSHL (multiple vectors)		 	461
D1.1.181 S	T1B (scalar plus immediate, consecutive registers)		 	463
D1.1.182 S	T1B (scalar plus scalar, consecutive registers) .		 	465
D1.1.183 S	T1B (scalar plus immediate, strided registers)		 	467
D1.1.184 S	T1B (scalar plus scalar, strided registers)		 	469
D1.1.185 S	T1B (scalar plus scalar, tile slice)		 	471
D1 1 186 S	ST1D (scalar plus immediate consecutive registers)		 	473
D1 1 187 S	ST1D (scalar plus scalar consecutive registers)		 	475
D1 1 188 S	ST1D (scalar plus immediate strided registers)		 	477
D1.1.100 G	(social plus initionale, struct registers)	• • • •	 • • •	7//

01.1.189 ST1D (scalar plus scalar, strided registers)		 	479
01.1.190 ST1D (scalar plus scalar, tile slice)		 	481
1 1 101 ST1H (scalar plus immediate, consecutive registers)		 •••	102
1.1.191 STITI (scalar plus infinediale, consecutive registers).	•••	 •••	405
01.1.192 STIH (scalar plus scalar, consecutive registers)	•••	 •••	485
01.1.193 ST1H (scalar plus immediate, strided registers)	• • •	 	487
01.1.194 ST1H (scalar plus scalar, strided registers)		 	489
01.1.195 ST1H (scalar plus scalar, tile slice)		 	491
1 1 196 ST10		 •••	103
	• • •	 • • •	405
JT. 1. 197 STTW (scalar plus infinediale, consecutive registers)	•••	 • • •	495
01.1.198 ST1W (scalar plus scalar, consecutive registers)	• • •	 	497
01.1.199 ST1W (scalar plus immediate, strided registers)		 	499
01.1.200 ST1W (scalar plus scalar, strided registers)		 	501
1 1 201 ST1W (scalar plus scalar tile slice)			503
1 1 200 CTNTID (scalar plus scalar, the shee)		 • • •	505
JT. 1.202 STINT IB (scalar plus infinediale, conservative, sters)	 • • •	505
01.1.203 SINI1B (scalar plus scalar, consective register.	• • •	 •••	507
01.1.204 STNT1B (scalar plus immediate, stored registers)		 	509
01.1.205 STNT1B (scalar plus scalar, st ded reaters)		 	511
1 1 206 STNT1D (scalar plus immeri de consecu de reciders	3		513
11200 STNT1D (scalar plus section consecutive routies)	,	 •••	515
	•••	 •••	515
01.1.208 SINI1D (scalar plus im. cliate unded registers)	•••	 •••	517
01.1.209 STNT1D (scalar plus scala, ded regirers)		 	519
01.1.210 STNT1H (scalar .us, media, conf Jutive registers)	 	521
)1 1 211 STNT1H (scal only solar con vive registers)	,		523
11212 STNT1H (c) ar plug v madiate strided registers)	•••	 •••	525
	•••	 •••	525
01.1.213 SINI1/4 (scal, 'n' 3 scalar, suided registers)	• • • •	 •••	527
01.1.214 STNT1v (scalar, is immediate, consecutive registers	3)	 	529
01.1.215 S7 (1W) a plan plan calar, consecutive registers)		 	531
01.1.216 (NT1W (sc. plus unmediate, strided registers)		 	533
1 1 2 S JT1W (ccala, jus scalar, strided registers)		 •••	535
1 1 / 10 CTL seter)	•••	 • • •	500
	•••	 • • •	537
.1.219 SIR (2)	• • •	 •••	539
01.1.220 SUB (arra, accumulators)		 	540
01.1 _1 SUB (array results, multiple and single vector)		 	542
222 SLV (array results, multiple vectors)			544
1 222 (DOT (multiple and indexed vector)	•••	 •••	546
	•••	 •••	540
1. 22 SUDOT (multiple and single vector)	•••	 •••	548
01.1.225 SUMLALL (multiple and indexed vector)	• • •	 	550
226 SUMLALL (multiple and single vector)		 	553
01.1.227 SUMOPA		 	555
1 1 228 SUMOPS			558
	•••	 • • •	500
71.1.229 SUNFR	•••	 •••	501
$1.1.230 \mathbf{SUVDOT} \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots $	•••	 •••	563
01.1.231 SVDOT (2-way)		 	565
01.1.232 SVDOT (4-way)		 	567
01.1.233 UCLAMP		 	569
1 1 234 LICVTE		 •••	571
1.1.204 UDOT (0. way multiple and indexed vector)	• • •	 •••	571
J1.1.235 UDOT (2-way, multiple and indexed vector)	•••	 •••	5/3
01.1.236 UDOT (2-way, multiple and single vector)	• • •	 	575
01.1.237 UDOT (2-way, multiple vectors)		 	577
01.1.238 UDOT (4-way, multiple and indexed vector)		 	579
1 1 239 LIDOT (4-way multiple and single vector)		 	582
11.240 UDOT (4 way, multiple and single vectors)	•••	 •••	502
	•••	 •••	504
01.1.241 UMAX (multiple and single vector)	• • •	 • • •	586
01.1.242 UMAX (multiple vectors)	• • •	 	588
01.1.243 UMIN (multiple and single vector)		 	590
· · · · · · · · · · · · · · · · · · ·			

D	1.1.244	UMIN (multiple vectors)	592
D	1.1.245	UMLAL (multiple and indexed vector)	594
D	1.1.246	UMLAL (multiple and single vector)	597
D	1 1 247	UMI AL (multiple vectors)	599
	1 1 2/10	LIMEALE (multiple vocior)	601
D	1.1.240		001
D	1.1.249		605
D	1.1.250	UMLALL (multiple vectors)	608
D	1.1.251	UMLSL (multiple and indexed vector)	611
D	1.1.252	UMLSL (multiple and single vector)	614
D	1.1.253	UMLSL (multiple vectors)	616
Л	1 1 254	LIMI SLL (multiple and indexed vector)	618
	1 1 255	LIMESTE (multiple and single vector)	622
D	1.1.200		022
D	1.1.256		625
D	1.1.257	UMOPA (2-way)	628
D	1.1.258	UMOPA (4-way)	630
D	1.1.259	UMOPS (2-way)	633
D	1.1.260	UMOPS (4-way)	635
Л	1 1 261	LIOCVT (two registers)	638
	1 1 262		620
D	1.1.202		039
D	1.1.263		640
D	1.1.264	UQRSHR (two registers)	641
D	1.1.265	UQRSHR (four r gisic)	642
D	1.1.266	UQRSHRN	644
D	1.1.267	URSHL (my ble and polovector)	646
Л	1 1 268	UBSHI (multi, v (ors)	648
ם	1 1 260	USDO) autiple ad indexed vector)	650
D	1.1.209		050
D	1.1.270		652
D	1.1.271	SDOT (mult, vectors)	654
D	1.1.2 _	U 1LALL (multi, and indexed vector)	656
D	1 1.273	USIN. 1LL (multiple and single vector)	659
Г	.1.274	USML, (multiple vectors)	661
0	1 1 275	USMOPA	663
	11 6		666
D	1.1 .0		000
			669
	1.278	/ JNPK	6/1
D	<u>ו. 7</u> 7	JVDOT (2-way)	673
D	0د ۲.1.	UVDOT (4-way)	675
	1.281	UZP (four registers)	677
D	1 1 282	UZP (two registers)	679
	1 1 283		681
	1 1 004		001
	1.1.204		003
D	1.1.285	WHILEHI	685
D	1.1.286	WHILEHS	687
D	1.1.287	WHILELE	689
D	1.1.288	WHILELO	691
D	1.1.289	WHILELS	693
ם	1 1 200	WHILFIT	692
	1 1 004		607
D -	1.1.291		09/
D	1.1.292	∠EKU (∠10)	699
D	1.1.293	ZIP (four registers)	700
D	1.1.294	ZIP (two registers)	702
D1.2	SVE2	data-processing instructions	704
n	1.2.1	BFMLSLB (vectors)	704
ם	122	BEMI SI B (indexed)	706
	100		700
D	۲.۷.۱		108

		D1.2.4	BFMLSLT (indexed)	710
		D1.2.5	FCLAMP	712
		D1.2.6	FDOT (vectors)	714
		D1.2.7	FDOT (indexed)	716
		D1.2.8	PFALSE	718
		D1.2.9	PSEL	719
		D1.2.10	REVD	721
		D1.2.11	SCLAMP	723
		D1.2.12	SDOT (2-way, vectors)	725
		D1.2.13	SDOT (2-way, indexed)	727
		D1.2.14	SQCVTN	729
		D1.2.15	SQCVTUN	730
		D1.2.16	SQRSHRN	731
		D1.2.17	SQRSHRUN	733
		D1.2.18	UCLAMP	735
		D1.2.19	UDOT (2-way, vectors)	737
		D1.2.20	UDOT (2-way, indexed)	739
		D1.2.21	UQCVTN	741
		D1.2.22	UQRSHRN	742
		D1.2.23	WHILEGE (predicate pa	744
		D1.2.24	WHILEGT (predicate pair)	746
		D1.2.25	WHILEHI (predir)	748
		D1.2.26	WHILEHS (p ^r uicate p r)	750
		D1.2.27	WHILELE (, `dicate	752
		D1.2.28	WHILE' O (pr∈ '∽ , pair)	754
		D1.2.29	WHILE⊾ 'oredic	756
		D1.2.30	$W' \perp ELT_{i_1} \text{rdicate} \text{rir}) \dots \dots \dots \dots \dots \dots \dots \dots \dots $	758
	D1.3	Base	4 instruction	760
		D1.3.1	Λ . \neg (immediate,	760
		D1 3.2	RPh. 1	764
		Г.3.3	SMSTA T	767
		1.3.4ل	SMSTOP	769
Devit C Appendix				
Part E Append	^a s			
Chapter E1	le str	uctiv	affected by SME	
	r su		instructions in Stroaming SVE mode	772
				773
		E1.1.1		781
	42	LI.I.Z		785
	F1.3	Redu	ced performance in Streaming SVE mode	786
	L1.0	F1.3.1	Scalar floating-point instructions	786
		E132	SVE instructions	786
		21.0.2		, 00
Chapter E2	SME	Shared	pseudocode	
•	E2.1	Pseud	docode functions	789

E2.1.1

E2.1.2

E2.1.3

E2.1.4

E2.1.5

E2.1.6

E2.1.7

E2.1.8

789

789

E	E2.1.9 E2.1.10	CheckSMEZT0Enabled
F	E2 1 11	CheckStreamingSVEEnabled 792
E	E2 1 12	CounterToPredicate 792
F	E2.1.13	CurrentNSVI 793
F	E2.1.14	CurrentSVI 793
Ē	E2 1 15	CurrentVI 793
F	E2 1 16	EncodePredCount 794
F	E2 1 17	EPAdd 7A 792
F	E2 1 18	EPDot 792
F	E2 1 19	EPDotAdd 796
E	E2 1 20	FPDotAdd 7A 796
E	E2 1 21	FPMulAdd 7A 796
E	E2 1 22	FPMulAddH 70 796
E	E2 1 22	EPProcessDenorms/ 707
E	E2.1.20	EPProcessNaNe4
E	L2.1.24 E0 1 05	FPSub 7Δ 707
L	L2.1.20	HoveEPE16 700
	E2.1.20	
E	EZ.1.2/	
E	EZ.1.20	
E	EZ.1.29	
E	E2.1.30	
E	E2.1.31	
E	E2.1.32	
E	E2.1.33	
E	E2.1.34	
E	E2.1.35	
E	E2.1.36	SMEEnable
E	=2.1.?	Is 'EEnabled
E	E2 1.58	
	.1.39	Maybe∠ [•] oSVEUppers
c	£2.1.40	PredCourt.test
E	E2.1	ResetSMEState
E	₹? .42	ReptSVEState
	. 1.43	CPSTATE_SM
E	E2. 1/	SetPSTATE_SVCR
E	E2.1.50	SetPSTATE_ZA
	2.46	SMEAccessTrap
E	E2.1.47	System
E	E2.1.48	ZAhslice
E	E2.1.49	ZAslice
E	E2.1.50	ZAtile
E	E2.1.51	ZAvector
E	E2.1.52	ZAvslice
E	E2.1.53	ZT0
Chapter E2 Svote	m rogio	stara affacted by CME
Chapter ES System		Siers anecieu by SME
E3.1	-3IVIE-0 ⊏2 1 1	D AAGASMEDO EL 1 SME Eacture ID register 0
E	=0.1.1 =0.1.0	MDAMSM EL 1. MDAM Stragging Made Degister
E	=3.1.2 =2.4.0	WCD EL1 SME Control Devictor (EL1)
E	⊏ວ.I.ວ ⊏ວ + ≠	SINCE_ELI, SIVIE CONTROL Register (ELI)
E	⊏ວ.⊺.4 ⊏ວ 1 ⊑	SINCE_EL2, SIVE CONTROL REGISTER (EL2)
E	=0.1.0 =0.1.0	SWIDE EL1 Streaming Mode Identification Deviator
E	=3.1.0 =2 4 7	SWIDLELI, Streaming Mode Detertine Desister
E	⊑3.1./	SWITH_ELT, Streaming wode Phonty Register

Contents Contents

	E3.1.8	SMPRIMAP_EL2, Streaming Mode Priority Mapping Register 837
	E3.1.9	SVCR, Streaming Vector Control Register
	E3.1.10	TPIDR2_EL0, EL0 Read/Write Software Thread ID Register 2 847
	E3.1.11	EDHSR, External Debug Halting Syndrome Register
E3.2	Chang	ges to existing System registers
	E3.2.1	CPACR_EL1, Architectural Feature Access Control Register 854
	E3.2.2	CPTR_EL2, Architectural Feature Trap Register (EL2)
	E3.2.3	CPTR_EL3, Architectural Feature Trap Register (EL3)
	E3.2.4	FAR_EL1, Fault Address Register (EL1)
	E3.2.5	FAR_EL2, Fault Address Register (EL2)
	E3.2.6	FAR_EL3, Fault Address Register (EL3)
	E3.2.7	FPCR, Floating-point Control Register
	E3.2.8	HCRX_EL2, Extended Hypervisor Configuration Register 903
	E3.2.9	HFGRTR_EL2, Hypervisor Fine-Grain reas Tap Register 913
	E3.2.10	HFGWTR_EL2, Hypervisor Fine-Gr ded Write D Register 941
	E3.2.11	ID_AA64ISAR1_EL1, AArch64 Inst. tion Set Attri. te Register 1 964
	E3.2.12	ID_AA64PFR1_EL1, AArch64 roces. Feature F gister 1 973
	E3.2.13	ID_AA64ZFR0_EL1, SVE F 0 r 0 979
	E3.2.14	MPAM2_EL2, MPAM2 R ^r , ster (El)
	E3.2.15	SCR_EL3, Secure Com, ratior egister
	E3.2.16	SCTLR_EL1, System Con. egister (F.1)
	E3.2.17	SCTLR_EL2, St. Jun ontrol giste cL2)
	E3.2.18	EDDEVID1, F. ernal C. bug Dev. D register 1

Chapter E4

Glossary terms

Preface

About this supplement

IRFSSZThis supplement is the Arm® Architecture Reference Manual Supplement, The Scalable Matrix Extension (SME),
for Armv9-A.ILZCVCThis supplement describes the changes and additions introduced by SME to the Armv9-A architecture.IFMHRZThis supplement also describes the changes and additions introduced by The Scalable Matrix Extension version 2
(SME2) to the Armv9-A architecture.IFMHRZIn this supplement, unless stated otherwise, when SME is used, the behavious poplies to SME2.IcompreFor SME, this supplement is to be read with the following documents

- Arm[®] Architecture Reference Manual for A-profile architec*ure
- Arm[®] Architecture Registers, for A-profile architecture
- Arm[®] A64 Instruction Set Architecture, for A-profile _____nitectur [3]

Together, the supplement and these documents provide a fun. Scrive on of the Armv9-A Scalable Matrix Extension, and the Armv9-A Scalable Matrix Extension version 2

This supplement is organized into parts:

SME Application Level Programmers Indel

Describes how the PE at an applic, on leve altered by the implementation of SME.

• SME System Level Program Lers' Mo.

Describes how the PE 2 ... tem level is a red by the implementation of SME.

• SME Instruction Sot

Describes the censions made for S. E to the A64 instruction set.

• Appendice

Provides refere z informs on relating to the SME. This includes summarized information about the instaction et, im_k ted ared pseudocode and System register data, and a glossary that defines terms used z this doc ment that we a specialized meaning.

Conventions

Typographical conventions

The typographical conventions are:

italic

Introduces special terminology, and denotes citations.

bold

Denotes signal names, and is used for terms in descriptive lists dere appropri-

monospace

Used for assembler syntax descriptions, pseudocode, ? . source code (mp'

Also used in the main text for instruction mnemon. and free ferences to other items appearing in assembler syntax descriptions, pseudocode, and source example

SMALL CAPITALS

Used for some common terms such as 'PLEME' ATION DEFINED.

Used for a few terms that have sp ific tec. al meanings, and are included in the Glossary.

Blue text

Indicates a link. This car

- A cross-reference to another logition within the document
- A URL, for exar .e http://develo, arm.com

Numbers

Number are n mally itten decimal. Binary numbers are preceded by 0b, and hexadecimal numbers by 0x. In bo cases, t prefix a me associated value are written in a monospace font, for example 0xFFFF0000. To imply reread in long numbers can be written with an underscore separator between every four characters, for example 0xFFF_0000_0000_0000. Ignore any underscores when interpreting the value of a number.

Pseudocode descript. Jns

This book uses a form of pseudocode to provide precise descriptions of the specified functionality. This pseudocode is written in a monospace font. The pseudocode language is described in the Arm Architecture Reference Manual.

Asterisks in instruction mnemonics

Some behavior descriptions in this manual apply to a group of similar instructions that start with the same characters. In these situations, an * might be inserted at the end of a series of characters as a wildcard.

Preface Conventions

Assembler syntax descriptions

This book contains numerous syntax descriptions for assembler instructions and for components of assembler instructions. These are shown in a monospace font.

Rules-based writing

This specification consists of a set of individual content items. A content item is classified as one of the following:

- Declaration
- Rule
- Goal
- Information

Declarations and Rules are normative statements. An implementation that is compliant with this specification must conform to all Declarations and Rules in this specification that apply to t^{1} , mp. pentation.

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

Arm strongly recommends that implementers read *all* choicers and ctions as document to ensure that an implementation is compliant.

Content items other than Declarations and Rules inform, ve stater ints. These are provided as an aid to understanding this specification.

Content item identifiers

A content item may have an asso died iden or which is unique among content items in this specification.

After this specification reach or status, a give, ontent item has the same identifier across subsequent versions of the specification.

Content item rendering

In this document, a contribution is condered with a token of the following format in the left margin: L_{iiiii}

- I a labe hat inc. te the content class of the content item.
- *diii* is the dentifier of the content item.

Content item clas. s

Declaration

A Declaration is a statement that does one or more of the following:

- Introduces a concept
- · Introduces a term
- Describes the structure of data
- Describes the encoding of data

A Declaration does not describe behavior.

A Declaration is rendered with the label D.

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*.

Goal

A Goal is a statement about the purpose of a set of rules.

- A Goal explains why a particular feature has been included in the specification
- A Goal is comparable to a "business requirement" or an "emergent provery."
- A Goal is intended to be upheld by the logical conjunction of a set of _____es.
- A Goal is rendered with the label G.

Information

An Information statement provides information ar _____'ance a ____ n aid tranderstanding the specification. An Information statement is rendered with thrabel *I*.

Additional reading

This section lists publications by Arm and by third parties.

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

[1] Arm[®] Architecture Reference Manual for A-profile architecture. (ARM DDI 0487) Arm Ltd.

[2] Arm[®] Architecture Registers, for A-profile architecture. (ARM DDI 0601) Arm Ltd.

[3] Arm[®] A64 Instruction Set Architecture, for A-profile architecture. (ARM DDI 0602) Arm Ltd.

[4] Arm[®] Architecture Reference Manual Supplement, Memory Syster Resource Cartitioning and Monitoring (MPAM), for A-profile architecture. (ARM DDI 0598) Arm Ltd.

[5] Arm[®] Architecture Reference Manual Supplement, The Transctional Vemory Extension (TME), for A-profile architecture. (ARM DDI 0617) Arm Ltd.

Feedback

Arm welcomes feedback on its documentation.

Feedback on this book

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

As part of the ticket, include:

- The title, (Arm® Architecture Reference Manual Supplement, T^{*} Scalable atrix Extension (SME), for Armv9-A).
- The number, (DDI0616 B.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, `applic_te.
- A concise explanation of your comments.

Arm also welcomes general suggestions for add: ons a. improver

Note

Arm tests PDFs only in Adobe crobat a. Acroba Peader, and cannot guarantee the appearance or behavior of any document when view vith any othe. DF reader.

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.

We believe that this document contains no offensive terms. If you find offensive terms in this document, please contact terms@arm.com.

Part A Introduction

Chapter A1 SME Introduction

A1.1 About the Scalable Matrix Extension

I_{YVCPK}

The Scalable M (x Exter n (SME) defines:

- Architectural . capable of .olding two-dimensional matrix tiles.
- A S can, g SVL processing mode, which supports execution of SVE2 instructions with a vector length that intches the tile with
 - instruction at accumulate the outer product of two vectors into a tile.
- L d .ore, and move instructions that transfer a vector to or from a tile row or column.

The extension also defines System registers and fields that identify the presence and capabilities of SME, and enable and coursel its behavior at each Exception level.

I_{PYQMB} The *Scalable Matrix Extension version 2* (SME2) extends the SME architecture to increase the number of applications that can benefit from the computational efficiency of SME, beyond its initial focus on outer products and matrix-matrix multiplication.

SME2 adds data processing instructions with multi-vector operands and a multi-vector predication mechanism. These include:

- Multi-vector multiply-accumulate instructions, that read SVE Z vectors and accumulate into ZA array vectors to permit reuse of the SME outer product hardware for vector operations, including widening multiplies that accumulate into more vectors than they read.
- Multi-vector load, store, move, permute, and convert instructions, that read and write multiple SVE Z vectors to preprocess inputs and post-process outputs of the multi-vector multiply-accumulate instructions.
- An alternative predication mechanism to the SVE predication mechanism, to control operations performed on multiple vector registers.

SME2 also adds:

- A Range Prefetch hint instruction to prepare the memory system to prefetch and retain a set of strided address ranges in the most appropriate cache levels.
- Compressed neural network capability using dedicated lookup table instructions and outer product instructions that support binary neural networks.
- A 512-bit architectural register, ZTO, to support the lookup table feature.
- I_{SQCGB} Unless otherwise specified by this document, the behaviors of instructions and architectural state when the PE is in
Streaming SVE mode are as described in $Arm^{(B)}$ Architecture Reference Manual for A-profile architecture [1].

Chapter A2 Architecture Features and Extension.

A2.1 Extensions and features c fined by SME

SME inherits the alles for chitectural features and extensions from $Arm^{\textcircled{m}}$ Architecture Reference Manual for *A-profile architect.* [1] This specification describes changes to those rules, and defines any features added by SME.

R_{PDXHJ}	SME represe ed by the ure FEAT_SME.
R _{QFSVK}	FEA SMF VAL extension from Armv9.2.
I _{VQCZZ}	The follo ng list summarizes the OPTIONAL SME features:
	 FEAT_S'E_FA64, support the full A64 instruction set in <i>Streaming SVE mode</i>. FEAT_SME_F64F64, Double-precision floating-point outer product instructions. FEAT_SME_I16I64, 16-bit to 64-bit integer widening outer product instructions. FEAT_EBF16, support for Extended BFloat16 mode.
R _{KXCXC}	FEAT_SME_FA64 requires FEAT_SVE2.
R _{kqddr}	SME2 represents a version of the SME architecture that implements FEAT_SME2.
R _{szztv}	FEAT_SME2 is an OPTIONAL extension from Armv9.2.
R _{bccql}	FEAT_SME2 requires FEAT_SME.

Chapter A2. Architecture Features and Extensions A2.2. Changes to existing features and extension requirements

A2.2 Changes to existing features and extension requirements

R_{dshws}

If SME is implemented, the following features are also implemented:

- FEAT_HCX.
- FEAT_FGT.
- FEAT_FCMA.
- FEAT_FP16.
- FEAT_FHM.
- FEAT_BF16.

Part B SM^r. App. 'cat. 'n Level Programmers' Model

Chapter B1 Application processing modes

B1.1 Overview

SME extends the AArch6⁺ application level programmers' model with added processing modes and related instructions, archaetur2⁺ ate, and registers:

- The ... T. SM & trol to .able an execution mode, known as *Streaming SVE mode*.
- T e PSTAL .ZA con. ' enable access to ZA storage, and to the ZTO register when SME2 is implemented. ('he Spec d-purpose register, svcr, which provides read/write access to PSTATE.SM and PSTATE.ZA from any rer on level.
- The MSTART and SMSTOP instructions, aliases of the MSR (immediate) instruction, that can set or clear PSTATL SM, PSTATE.ZA, or both PSTATE.SM and PSTATE.ZA from any Exception level.
- A softwa. thread ID register to manage per-thread SME context, TPIDR2_EL0.

B1.2 Process state

D _{XDPXS}	A PE that implements SME has a Streaming SVE mode.					
D _{JYVLM}	Streaming SVE register state is the vector registers <i>Z0-Z31</i> and predicate registers <i>P0-P15</i> that can be accessed by SME, SVE, Advanced SIMD, and floating-point instructions when the PE is in <i>Streaming SVE mode</i> .					
D _{dmzfr}	Streaming SVE register state includes the SVE <i>FFR</i> predicate register if FEAT_SME_FA64 is implemented and enabled at the current Exception level.					
I _{XXKGV}	If SME is implemented, a PE has the following additional architectural state:					
	 Streaming SVE register state. ZA storage. When SME2 is implemented, the ZTO register. 					
I _{ZTTNW}	A PE enters Streaming SVE mode to access Streaming SVE vector ar predicate regimer state.					
I _{NXQFB}	If SME is implemented, this does not imply that FEAT_SVE and 'EAT 'E2 are implemented by the PE when it is not in <i>Streaming SVE mode</i> .					
I _{rntpx}	When the PE is in <i>Streaming SVE mode</i> , a different set c ['] ector ler ins might vavailable for SVE instructions, as specified in C1.1.3 <i>Vector lengths</i> .					
I _{TDSPN}	When the PE is in <i>Streaming SVE mode</i> , the performance harace some instructions might be significantly reduced, as specified in E1.3 <i>Reduced performance in Streaming Conde</i> .					
I _{NWFQX}	SME extends a PE's <i>Process state</i> or PSTATE 'th the state of estate fields can be modified by the SMSTART and SMSTOP instructions, and c also b ad and written using the SVCR register.					
I _{DHSSW}	The PSTATE. SM field controls the y' of $Streering Sreering determined on the state of the st$					
I _{NVWRT}	The PSTATE.ZA field controls . `the followine					
	 Access to ZA storage. Access to the ZT register, when TE2 is implemented. 					
I _{DVPDL}	The SMSTART instruction doe lither or both of the following:					
	 Enters Stream of E mode. Enability by ZA and then SME2 is implemented enables the ZTO register. 					
I _{QQZTL}	The surger instruction do the or both of the following:					
	 tits S common C mode. Dr. tes the ZA storage, and when SME2 is implemented disables the ZTO register. 					
I _{NKJKL}	After entern. Streaming SVE mode, subsequent SMSTART and SMSTOP instructions might be used to enable and disable the ZA storage, and the ZTO register when SME2 is implemented, for different phases of execution within Streaming SVE mode, before using a final SMSTOP instruction to exit Streaming SVE mode.					
D _{WHXDZ}	SME and SME2 instructions are the instructions defined by the SME architecture in Chapter D1 SME instructions.					
D _{NHNFF}	A <i>legal</i> instruction is an implemented instruction that can be executed by a PE when PSTATE.SM and PSTATE.ZA are in the required state, unless its execution at the current Exception level is prevented by a configurable trap or enable.					
D _{HZFSG}	An <i>illegal</i> instruction is an implemented 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.					
	See also:					
	• MSR (immediate).					

• SMSTART.

Chapter B1. Application processing modes B1.2. Process state

- SMSTOP.
- C1.1.3 Vector lengths.
- C1.2.3 Validity of SME and SVE state.
- C1.3.4 ESR_EL1, ESR_EL2, and ESR_EL3.
- C1.4.8 SVCR.
- Chapter E1 Instructions affected by SME.

B1.2.1 PSTATE.SM

I _{YYQJK}	The value of PSTATE.SM can be changed by executing the MSR instructions that access the SVCR. For more information, see B1.2.3 <i>Changing PSTATE.SM and PSTATE.ZA</i> .
D _{PRGHY}	The PE is in Streaming SVE mode when the Effective value of PSTATE. SM is 1
R _{VJZBC}	When the PE is in <i>Streaming SVE mode</i> :
	 Streaming SVE register state is valid. SME and SME2 instructions that access the Streaming SV register onte are <i>le l.</i> SME and SME2 instructions that do not access the ZA orage or ZTO is sterilized re <i>legal</i>. SME and SME2 instructions that access the ZA storilized or ZTO gister are <i>legal</i>. SME and SME2 instructions that access the ZA storilized or ZTO gister are <i>legal</i>. <i>Legal</i> instructions that access SVE or SIMD&FP reg. rs cless the Streaming SVE register state.
I _{YDRPH}	The SVE <i>FFR</i> predicate register is not archite and visit where he PE is in <i>Streaming SVE mode</i> if FEAT_SME_FA64 is not implemented or not abled at he curre. The ception level.
R _{cksbs}	When the PE is in <i>Streaming SVE mode</i> and . `AT_S' not implemented or not enabled at the current Exception level:
	 Most Advanced SIMD instructions an <i>'legal</i>, described in E1.1.1 <i>Illegal Advanced SIMD instructions</i>. Some SVE and SVE2 instructions are <i>ili</i>, <i>'l</i>, as a scribed in E1.1.2 <i>Illegal SVE instructions</i>. Most other instruction on p. bented by the F, including scalar floating-point instructions, remain <i>legal</i>.
D _{dvdty}	The PE is not in <i>Stream's SVE mode</i> ben the Effective value of PSTATE.SM is 0.
R _{XBBFD}	When the PE is not Streaming SVE moa
	 Streaming C 'E register state is not valid. SME and SM. 's dructions' dat access the Streaming SVE register state are <i>illegal</i>. SM' and 'vector' STR (vector), and ZERO (tile) instructions that access the ZA storage are <i>legal</i> if ZA is vabled, a 1 all othe 'vector' are (ZTO), and ZERO (ZTO) instructions that access the ZTO register are <i>legal</i> if ZA is constructions that access the ZTO instructions that access the ZTO register are <i>legal</i> if ZA is constructions that access the ZTO register are <i>legal</i>. SME2 1 ('TTO), STR (ZTO), and ZERO (ZTO) instructions that access the ZTO register are <i>legal</i> if ZA is constructions that access the ZTO register are <i>legal</i>. The TR and MRS instructions that directly access the SME SVCR register are <i>legal</i>. Instructions which access SVE or SIMD&FP registers access the Non-streaming SVE or SIMD&FP register state. All other instructions implemented by the PE are <i>legal</i>.
R _{rswfq}	When the Effective value of PSTATE.SM is changed by any means from 0 to 1, an entry to <i>Streaming SVE mode</i> is performed, and each implemented bit of SVE registers <i>Z0-Z31</i> , <i>P0-P15</i> , and <i>FFR</i> in the new mode is set to zero.
R _{KFRQZ}	When the Effective value of PSTATE.SM is changed by any means from 1 to 0, an exit from <i>Streaming SVE mode</i> is performed, and each implemented bit of SVE registers <i>Z0-Z31</i> , <i>P0-P15</i> , and <i>FFR</i> in the new mode is set to zero.
R_{MHTLZ}	When the Effective value of PSTATE.SM is changed by any means from 0 to 1, or from 1 to 0, the FPSR is set to the value 0x0000_0000_0800_009f, in which all of the cumulative status bits are set to 1.
I _{YTZVD}	Statements which refer to the value of the SVE vector registers, <i>Z0-Z31</i> , implicitly also refer to the lower bits of those registers accessed by the SIMD&FP register names <i>V0-V31</i> , <i>Q0-Q31</i> , <i>D0-D31</i> , <i>S0-S31</i> , <i>H0-H31</i> , and <i>B0-B31</i> .
	See also:

- C1.1.2 *Traps and exceptions*.
- C1.4.8 SVCR.

B1.2.2 PSTATE.ZA

The value of PSTATE.ZA can be changed by executing the MSR instructions that access the SVCR. For more I_{GJZLD} information, see B1.2.3 Changing PSTATE.SM and PSTATE.ZA. The following are enabled when PSTATE.ZA is 1: D_{HBFWD} • The ZA storage. • When SME2 is implemented, the ZT0 register. When ZA storage is enabled: R_{SFWMY} • The contents of ZA storage, and the ZTO register when SME2 is ; plementea, 'e valid and are retained by hardware irrespective of whether the PE is in Streaming SVE n. • SME and SME2 instructions that access the ZA storage of the ZI register are gal and can be executed, unless execution is prevented by some other trap or excr .ion. The following are disabled when PSTATE.ZA is 0: D_{VLMFC} • The ZA storage. • When SME2 is implemented, the ZT0 regis* When ZA storage is disabled: $R_{\rm JHMYL}$ • The contents of ZA storage, and the ZTO. ist when Sum 2 is implemented, are not valid. • SME and SME2 instructions that a stream 'storage or the ZT0 register are illegal. • There is no effect on other in Luction. mplen. "ed by the PE. When PSTATE. ZA is changed b ... means from 1, all implemented bits of the ZA storage, and the ZTO register Ryrzrm when SME2 is implemented are sub-zero. When PSTATE. ZA is choiced from 1 to there is no architecturally defined effect on the ZA storage, and the ZTO ILRDZR register when SMF^{*} s implemented, beca, e the contents of ZA storage and the ZTO register cannot be observed when PSTATE. Z7 0. ged from (.0 1, or 1 to 0, there is no effect on the SVE vector and predicate registers and When PSTATE.ZA is IQWCJS the FPSP ... E.SM. ot ch ged. See .o: 6 AE2 Loon table. • C1. ? Traps and exceptions. • C1.4. *VCR*.

B1.2.3 Changing PSTATE.SM and PSTATE.ZA

D_{QRSXV} The following MSR (immediate) instructions are provided to independently set or clear PSTATE.SM, PSTATE.ZA, or both PSTATE.SM and PSTATE.ZA respectively:

- MSR SVCRSM, #<imm1>.
- MSR SVCRZA, #<imml>.
- MSR SVCRSMZA, #<imml>.

R_{MPQWY} MSR SVCRSM, MSR SVCRZA, and MSR SVCRSMZA instructions are permitted to be executed from any Exception level.

D_{YGDXX} The SMSTART instruction is the preferred alias of the following instructions:

• MSR SVCRSM, #1.

Chapter B1. Application processing modes B1.2. Process state

- MSR SVCRZA, #1.
- MSR SVCRSMZA, #1.

D_{dztdh}

- MSR SVCRSM, #0.
- MSR SVCRZA, #0.
- MSR SVCRSMZA, #0.
- L_{SZYBC} Access to SVCR through the MRS and MSR (register) instructions might be used where a calling convention or ABI requires saving, restoring, or testing of PSTATE.SM and PSTATE.ZA, and are permitted to be executed from any Exception level. However, the MSR (immediate) instructions might be higher performance than the MSR (register) instruction, so the MSR (immediate) instructions are preferred for explicit changes to PSTATE.SM and PSTATE.ZA.

I_{HNNJR} The PE might consume less power when PSTATE.SM is 0 and PSTATE.ZA is 0.

The SMSTOP instruction is the preferred alias of the following instructions:

See also:

- MSR (immediate).
- SMSTART.
- SMSTOP.
- C1.4.8 SVCR.

B1.2.4 TPIDR2_EL0

- D_{FJMMT} If SME is implemented, the register TPIDR2 0 is add
- ISLNHN
 The Software Thread ID Register #2 provides adding information that can be read and written from all Exception levels.
- I_{QPMJN} This register is reserved for use the ABI to mage per-thread SME context.

See also:

• TPIDR2_EL0.

Chapter B2 Architectural state

B2.1 Architectural state summ.

D_{XJCGQ} The *Effective St* ming SV vector length (SVL) is a power of two in the range 128 to 2048 bits inclusive.

I_{NBPPM} When the PE is in 5. *ing SVE Jde*, the *Effective SVE vector length* (VL) is equal to SVL.

This m^2 is be c ferent i. n^2 value of VL when the PE is not in *Streaming SVE mode*, as described in C1.1.3 *Vect lengths.*

D_{JBVYJ} In a ver SVL bus.

- SVL_{b} the number of 8-bit elements.
- SVL_H is e number of 16-bit elements.
- SVL_s is the number of 32-bit elements.
- SVL_D is the number of 64-bit elements.
- SVL_Q is the number of 128-bit elements.

SVL [bits]	SVL _B	SVL _H	SVL _S	SVL _D	SVLQ
128	16	8	4	2	1
256	32	16	8	4	2
512	64	32	16	8	4
1024	128	64	32	16	8
2048	256	128	64	32	16

Chapter B2. Architectural state B2.1. Architectural state summary

See also:

- Chapter B1 Application processing modes.
- C1.1.3 Vector lengths.


B2.2 SME ZA storage

 D_{SSXPL} The ZA storage is an architectural register state consisting of a two-dimensional ZA array of [SVL_B × SVL_B] bytes.

B2.2.1 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 (vector) and STR (vector) instructions, an untyped ZA array vector is selected by the sum of a 32-bit general-purpose vector select register Wv and an immediate vector select offset offs, modulo SVL_B. D_{YXHFR} The preferred disassembly for an untyped ZA array vector is ZA[Wv, offs] where c_{A} 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, 61-bit, 128-bit eler nts.

D_{WMVZT} An elementwise vector access to the ZA array is indicated by pending a vector $:inc^{+}$::[N]" to the ZA array name and element size qualifier, where N is in the range 0 to S^V $_{-B}$ -1 inc¹ , ve, as for ws:

- An 8-bit element vector access to the ZA array is repreted by ZA.F $\sqrt{2}$.
- A 16-bit element vector access to the ZA arr \dots present ¹ by Z' A[N].
- A 32-bit element vector access to the ZA ray is resented. A.S[N].
- A 64-bit element vector access to the I array is oresented by ZA.D[N].
- A 128-bit element vector access to the Z. "r? is represented by ZA.Q[N].

B2.2.2 ZA tile access

D _{VSVMX}	A ZA tile is a square, two-a. nensio. sub-array of elements within the ZA array.
I _{WLRTV}	Depending on the element size with whether 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 indic. ¹ by pending the tile number to the ZA name.
D _{zgbht}	A ZA tile a one "mensic at set of horizontally or vertically contiguous elements within a ZA tile.
R_{PZNWB}	A ve or access o a tile re. or writes a ZA tile slice.
I _{NFXHH}	A ZA crobe access as vectors of 8-bit, 16-bit, 32-bit, 64-bit, or 128-bit elements.
I _{YZDBS}	A ZA tile 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 "V" 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 <i>offs</i> , modulo the number of slices in the named tile.

B2.2.3 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 m for SVL of 256 bits:



 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[Ws, offs], for a horizontal 8-bit element ZA tile slice selection.

• ZAOV.B[Ws, offs], for a vertical 8-bit element ZA tile slice selection.

Where *offs* is an immediate in the range 0-15 inclusive.

B2.2.4 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.
- 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 c \cdot , ... V is in the range 0 to SVL_H-1 inclusive:
 - ZA*t*H.H[*N*] indicates a horizontal 16-bit element ZA tile slice set tion.
 - ZAtV.H[N] indicates a vertical 16-bit element ZA tile slic election.
- I_{ZSWJW} Horizontal and vertical ZA*t*.H slice accesses, where *t* is 0 c \cdot , are illy 'rated i.' following diagram for SVL of 256 bits:



Chapter B2. Architectural state B2.2. SME ZA storage

R _{btlqc}	An access to the horizontal slice $ZAtH.H[N]$ 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 $ZAtV.H[N]$ reads or writes the 16-bit element [N] within each horizontal slice of $ZAt.H$.
D _{rhqjt}	 The preferred disassembly is as follows: ZAtH.H[Ws, offs], for a horizontal 16-bit element ZA tile slice selection.
	• ZATV.H[Ws, offs], for a vertical 16-bit element ZA file slice selection.

Where *t* is 0 or 1, and *offs* is an immediate in the range 0-7 inclusive.

B2.2.5 Accessing a 32-bit element ZA tile

 D_{HBKZV} A 32-bit element ZA tile is indicated by a ".S" qualifier following the t' name.

- D_{RDRRT} There are four tiles named ZA0.S, ZA1.S, ZA2.S, and ZA3.S. Each tile visits of [SV1 × 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 ince read or write S L_S 32-bit elements.
- D_{JFPSJ} An access to a horizontal or vertical 32-bit element ZA tile fice indicated by appending a slice index "[N]" to the tile name, direction suffix, and qualifier. For example, is 0, 1, 2 or 3, and N is in the range 0 to SVL_S-1 inclusive:
 - ZAtH.S[N] indicates a horizontal 32-t element slice selection.
 - ZAtV.S[N] indicates a vertical 32-bit ele. nt 1 tile sncc election.
- ISERVICEHorizontal and vertical ZAt.S slice cces.when is 0, 1, 2, or 3, are illustrated in the following diagram for
SVL of 256 bits:



 \mathbb{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 ZAtV.S[N] reads or writes the 32-bit element [N] within each horizontal slice of ZAt.S.

D_{LOLJH} The preferred disassembly is:

- ZAtH.S[Ws, offs], for a horizontal 32-bit element ZA tile slice selection.
- ZAtV.S[Ws, 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.

B2.2.6 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.P and ZA7.D. Each tile consists of [SVL_D × SVL_D] 64-bit elements, and occupies an eighth of the ZA rage.
- R_{ZXYBQ} An access to a horizontal or vertical 64-bit element ZA tile slice reads writes SVL_{E} 4-bit elements.
- D_{DCXSX} An access to a horizontal or vertical 64-bit element ZA tile slip is indicate by appending a slice index "[N]" to the tile name, direction suffix, and qualifier. For example where t is in the time to 0-7 inclusive, and N is in the range 0 to SVL_D-1 inclusive:
 - ZAtH.D[N] indicates a horizontal 64-bit elem 74 tile ce select ...
 - ZAtV.D[N] indicates a vertical 64-bit elem int ZA \Rightarrow slice $\uparrow \circ c \uparrow$ in.
- I_{LGJZC} Horizontal and vertical ZAt.D slice accesse. where t j is the range 0-7 inclusive, are illustrated in the following diagram for SVL of 256 bits:



R _{CVVJK}	An access to the horizontal slice ZA <i>t</i> H.D[<i>N</i>] 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 $ZAtV.D[N]$ reads or writes the 64-bit element [N] within each horizontal slice of $ZAt.D$.

- D_{MQQPX} The preferred disassembly is:
 - ZAtH.D[Ws, offs], for a horizontal 64-bit element ZA tile slice selection.
 - ZAtV.D[Ws, *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.

B2.2.7 Accessing a 128-bit element ZA tile

A 128-bit element ZA tile is indicated by a ".Q" qualifier following the tile name. D_{gzdsh}

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, D_{RPMJL} ZA11.Q, ZA12.Q, ZA13.Q, ZA14.Q, and ZA15.Q. Each tile consists of $[SVL_0 \times SVL_0]$ 128-bit elements, and occupies 1/16 of the ZA storage.

An access to a horizontal or vertical 128-bit element ZA tile slice reads or writes SVL₀ 128-bit elements. Roghpf

- ZAtH.Q[N] indicates a horizontal 128-bit element ZA tile slice selection
- ZAtV.Q[N] indicates a vertical 128-bit element ZA tile slice select
- Horizontal and vertical ZAt.Q slice accesses, where t is in the range (5 inclusive, ar llustrated in the following IYOPWS diagram for SVL of 256 bits:



- An access to the hor intal slice ZAtH.QL reads or writes the SVL₀ 128-bit elements in ZA array vector ZA.Q[t R_{PJTOJ} + 16 * *N*].
- slice $ZAt^{\vee} Q[N]$ reads or writes the 128-bit element [N] within each horizontal slice of An access to the ve. R_{TRJFZ} ZAt.Q.
- The *r* _ferred d assembly D_{VCLJP}
 - **A** tH (ws, v), a horizontal 128-bit element ZA tile slice selection.
 - ZA Q[Ws, 0], for a vertical 128-bit element ZA tile slice selection.

Where *t* is in range 0-15 inclusive, and the slice index offset is always zero.

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 SVL₀-1 inclusive:

B2.3 ZA storage layout

B2.3.1 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]	_A1H., 기	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 ¹	`43H.D[0'	ZA3H.Q[0]
ZA[4]	ZA0H.B[4]	ZA0H.H[2]	ZA0H [1]	ZA V J]	ZA4H.Q[0]
ZA[5]	ZA0H.B[5]	ZA1H.H[2]	ZA1h. [1]	ZA 5H.D[0]	ZA5H.Q[0]
ZA[6]	ZA0H.B[6]	ZA0H.H[3]	Ч.S[1,	_A6H.D[0]	ZA6H.Q[0]
ZA[7]	ZA0H.B[7]	ZA1H.H[3]	ZA ⁷ <i>I</i> .S[1]	ZA7H.D[0]	ZA7H.Q[0]
ZA[8]	ZA0H.B[8]	ZA0H.H[4]	۸0H.δ _L ∠,	ZA0H.D[1]	ZA8H.Q[0]
ZA[9]	ZA0H.B[9]	ZA1H H[4]	[•] A1H.S[2]	ZA1H.D[1]	ZA9H.Q[0]
ZA[10]	ZA0H.B[10]	Z´,H.H[5]	ZA	ZA2H.D[1]	ZA10H.Q[0]
ZA[11]	ZA0H.B[11]	ZA1. 4[5]	ZA3H.S[2]	ZA3H.D[1]	ZA11H.Q[0]
ZA[12]	ZA0H.B[12]	ZA0H.HL	ZA0H.S[3]	ZA4H.D[1]	ZA12H.Q[0]
ZA[13]	ZA0H.B[¹	7A1H.H[6]	ZA1H.S[3]	ZA5H.D[1]	ZA13H.Q[0]
ZA[14]	ZA0H.BL	ZA0H H[7]	ZA2H.S[3]	ZA6H.D[1]	ZA14H.Q[0]
ZA[15]	7 [15]	Z .H.H[7]	ZA3H.S[3]	ZA7H.D[1]	ZA15H.Q[0]
if applicable ZA[16] to ZA[SVL _B -1]					

B2.3.2 Tile mappings

IYVYJPThe 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:

Overlaps
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.Q, ZA2.Q, ZA4.Q, ZA6.Q, ZA8.Q, ZA10.Q, ZA12.Q, ZA14.Q
ZA1.Q, ZA3.Q, ZA5.Q, ZA7.Q, ZA9.Q, ZA11.Q, ZA13.Q, ZA15.Q

Chapter B2. Architectural state B2.3. ZA storage layout

Tile	Overlaps
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.

 I_{WGZBT} The architecture permits concurrent use of dif tent eler int size the

B2.3.3 Horizontal tile slice mappings

I_{NJJXW} The following diagram illustrate the ZA store, mapping for SVL of 256 bits, for a 32-bit element and 64-bit element horizontal tile slice

Each small numbered s are represen. ² bits.



Chapter B2. Architectural state B2.3. ZA storage layout

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:

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

 31
 30
 29
 28
 27
 26
 25
 24
 23
 22
 21
 10
 14
 13
 12
 11
 10
 9
 8
 7
 6
 5
 4
 3
 2
 1
 0

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

 32
 24
 23
 22
 21
 20
 19
 18
 17
 16
 15
 14
 13
 12
 11
 10
 9
 8
 7
 6

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 ZA4H.D[2]

B2.3.4 Vertical tile slice mappings

 I_{TNCCV} The following diagram illustrates the ZA storage mapping for SVL of 56 bits, to 32-bit element and 64-bit element vertical tile slice.

Each small numbered square represents 8 bits.



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:

 I_{CGXPJ}

B2.3.5 Mixed horizontal and vertical tile slice mappings

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.



It is possible to signal taneously use non-overlapping ZA array vectors within tiles of differing element sizes. For example, tiles ZA, H, Z, J.S, and ZA2.D have no ZA array vectors in common, as illustrated in the following diagram for SVL of ... oits:

Chapter B2. Architectural state B2.3. ZA storage layout



It is possible to access overlapping Z^A rray v. s within tiles of differing element sizes. For example, tiles ZA0.H, ZA2.S, and ZA6.D have common. A array octors.

B2.4 SME2 Multi-vector operands

R_{KLRKJ} Multi-vector operands allow certain SME2 instructions to access source and destination operands which each consist of one of the following:

- 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.

B2.4.1 Z multi-vector operands

D _{PSTFY}	A multi-vector operand consisting of two or four SVE Z vector registers is colled a Z multi-vector operand.
R _{VCXBQ}	A Z multi-vector operand can occupy:
	 Consecutively numbered Z registers. Z registers with strided numbering.
D _{NYNRZ}	A Z multi-vector operand occupying two consecutively number operand Z ectors x and $Zn+0$ and $Zn+1$, where $n+x$ modulo 32 is a register number in the range 0-31 inclusive.
D _{dzdbm}	A Z multi-vector operand occupying four consecutive number Z vector consists of $Zn+0$ to $Zn+3$, where $n+x$ modulo 32 is a register number in the range 0-31 inclus.
D _{VYKCM}	The preferred disassembly for a Z multi-vec r operand of consecutively numbered Z vectors is a dash-separated register range, for example { Z0.S-Z1.S } or 730 -Z1.B $_{1.2}$ -volchains must also support assembler source code that uses the alternative comma-separated lise obtain, for example { Z0.S, Z1.S } or { Z30.B, Z31.B, Z0.B, Z1.B }. Disassemblers can provide r optic r select retween the dash-separated range and comma-separated list notations.
D _{PCYZS}	A Z multi-vector operand o $apyin_{\xi}$ vo Z vectors with strided register numbering consists of a first register in the range Z0-Z7 or Z16-Z2 ² followed by second register with a number that is 8 higher than the first register.
D _{rzttv}	A Z multi-vector op and occupying four \geq ectors with strided register numbering consists of a first register in the range Z0-Z3 or 7 \rightarrow -Z19, fc \rightarrow wed by three registers each with a number that is 4 higher than the previous register.
D _{DMTSL}	The preferred disa. Sly for a multi-vector operand of Z vectors with strided register numbering is a comma-commune 1 register list, f example { Z0.D, Z8.D } or { Z0.H, Z4.H, Z8.H, Z12.H }.
B2.4.2	ZA mu ¹ -sli - op ds

D _{jmctk}	A multi-vec • operand consisting of two or four ZA tile slices is called a 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 _{JFDSB}	In instructions operating on ZA multi-slice operands, the lowest-numbered slice is:
	A multiple of 2 for a two-slice <i>ZA</i> operand.A multiple of 4 for a four-slice <i>ZA</i> operand.
	The lowest-numbered slice is selected by the sum of a 32-bit general-purpose slice index register Ws and an immediate slice index offset <i>offs</i> .
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:

- ZAtH.T[Ws, offs1:offs2], for horizontal ZA two-slice operands, where offs2 = offs1 + 1.
- ZAtH.T[Ws, offs1:offs4], for horizontal ZA four-slice operands, where offs4 = offs1 + 3.
- ZAtV.T[Ws, offs1:offs2], for vertical ZA two-slice operands, where offs2 = offs1 + 1.
- ZAtV.T[Ws, offs1:offs4], for vertical ZA four-slice operands, where offs4 = offs1 + 3.

B2.4.3 ZA multi-vector operands

A multi-vector operand consisting of two, four, eight, or sixteen ZA array vectors is called a ZA multi-vector D_{RGXBK} operand. Digdre One ZA array vector is called a ZA single-vector group. Two consecutively numbered vectors in the ZA array are called a ZA vble-vector gip. D_{FCYGL} Four consecutively numbered vectors in the ZA array are called ZA qual ector group. D_{GCTYB} The ZA multi-vector operand consists of one, two, or for vector operand constraints are specific vector operand consists of one, two, or for vector operand constraints are specific vector operand co a vector group is one of the IPMORO following: • ZA single-vector group. • ZA double-vector group. • ZA quad-vector group. The SME2 architecture includes multi-vector i. v ons that access a ZA multi-vector operand consisting of the I_{KLBYZ} same number of vector groups as there λ vector, γ each Z multi-vector operand. The preferred disassembly for 2 A multi-v tor op, nd consisting of two or four vector groups, defined in I_{HPKZM} declarations D_{KOZYZ}, D_{JWRSN}, D_{TTNGH}, inc. 'es the symbol VGx2 or VGx4, respectively. The symbol VGx2 or VGx4 can optionally be omi d in a mbler source code if it can be inferred from the other operands. In instructions that acc _ a ZA multi-vex r operand, the lowest-numbered vector is selected by the sum of a 32-bit D_{CLJBX} general-purpose ver c select register Wv. d an immediate vector select offset offs, modulo one of the following values: .rand cor .s of one ZA vector group. • SVL_B when the • SV' ____ 'en the peran' onsists of two ZA vector groups. • $\int L_{\rm B}/4$ w on the option of four ZA vector groups.

B2.4. 1 ZA mum-vector operands of single-vector groups

- D_{QFPHH} In instruction, 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. The halves of the ZA array are as follows, where *n* is in the range 0 to (SVL_B/2 - 1) inclusive:
 - ZA[n+0].
 - $ZA[SVL_B/2 + n+0]$.
- 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. The quarters of the ZA array are as follows, where *n* is in the range 0 to (SVL_B/4 - 1) inclusive:
 - ZA[n+0].
 - $ZA[SVL_B/4 + n+0]$.
 - $ZA[SVL_B/2 + n+0]$.
 - $ZA[SVL_B*3/4 + n+0].$

Chapter B2. Architectural state B2.4. SME2 Multi-vector operands

 D_{KQZYZ} The preferred disassembly for a ZA multi-vector operand of single-vector groups is as follows, where *offs* is an immediate in the range 0-7 inclusive, and T is one of B, H, S, or D:

- ZA.*T*[Wv, offs, VGx2], when the operand consists of two single-vector groups.
- ZA.*T*[Wv, offs, VGx4], when the operand consists of four single-vector groups.
- 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:



 $I_{\rm MLNNG}$

The apping b ween ZA. atti-vector operands of single-vector groups and 64-bit element ZA tile slices when SVL 256 b to the following diagram:



B2.4.3.2 ZA multi-vector corrands of double-vector groups

- D_{RBSQJ} In instructions where the ZA multi-vector c_x erand consists of one double-vector group, the vector group is held in ZA array vectors A[n+0] (ZA[n+1], where *n* is a multiple of 2 in the range 0 to (SVL_B 2) inclusive.
- D_{KKVVG} In instructions where 'ZA multi-ctor operand consists of two double-vector groups, each vector group is held in a separation of the 'arra' the halves of the ZA array are as follows, where *n* is a multiple of 2 in the range 0 to $(L_B/2 -)$ inclusive
 - $A[n+1, \infty, n-1]$. • $Z_{L_B/2} + n+0$ to $ZA[SVL_B/2 + n+1]$.
- D_{VMYGN} In instructio, 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. The quarters of the ZA array are as follows, where *n* is a multiple of 2 in the range 0 to (SVL_B/4 - 2) inclusive:
 - 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 $ZA[SVL_B/2 + n+1]$.
 - $ZA[SVL_B*3/4 + n+0]$ to $ZA[SVL_B*3/4 + n+1]$.

 D_{JWRSN} The preferred disassembly for a ZA multi-vector operand of double-vector groups is as follows, where offs2 = offs1 + 1, and T is one of B, H, S, or D:

- ZA.*T*[Wv, *offs1:offs2*], where *offs1* is a multiple of 2 in the range 0-14 inclusive, when the operand consists of one double-vector group.
- ZA.*T*[Wv, *offs1:offs2*, VGx2], where *offs1* is a multiple of 2 in the range 0-6 inclusive, when the operand consists of two double-vector groups.

Chapter B2. Architectural state B2.4. SME2 Multi-vector operands

- ZA.*T*[Wv, offs1:offs2, VGx4], where offs1 is a multiple of 2 in the range 0-6 inclusive, when the operand consists of four double-vector groups.
- ILZRIK
 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:



 I_{JYQTB} The mapping between multi-versor operands of double-vector groups and 64-bit element ZA tile slices when SVL is 2^f c. is illusive to following diagram:



B2.4.3.3 ZA multi-vector c arands of quad-vector groups

- D_{WSTWB} In instructions when the ZA multi-vector period consists of one quad-vector group, the vector group is held in ZA array vectors A[n+0] ZA[n+3], where n is a multiple of 4 in the range 0 to (SVL_B 4) inclusive.
- D_{QJXHS} In instructions where 'ZA multi-ctor operand consists of two quad-vector groups, each vector group is held in a separat that 'the Z rray,' the halves of the ZA array are as follows, where *n* is a multiple of 4 in the range 0 to $(SV_B/2 4)$ inclusive:

•
$$A[n+1, w=2^{n}]$$
.
• $Z_{L} = \sqrt{L_{B}/2 + n + 0}$ to $ZA[SVL_{B}/2 + n + 3]$.

- D_{BQWJD} In instruction 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. The quarters of the ZA array are as follows, where *n* is a multiple of 4 in the range 0 to (SVL_B/4 4) inclusive:
 - 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 $ZA[SVL_B/2 + n+3]$.
 - $ZA[SVL_B*3/4 + n+0]$ to $ZA[SVL_B*3/4 + n+3]$.

D_{TTNGH} The preferred disassembly for a ZA multi-vector operand of quad-vector groups is as follows, where offs4 = offs1 + 3, and T is one of B, H, S, or D:

- ZA.*T*[Wv, offs1:offs4], where: offs1 is a multiple of 4 in the range 0-12 inclusive, when the operand consists of one quad-vector group.
- ZA.T[Wv, offs1:offs4, VGx2], where offs1 is 0 or 4, when the operand consists of two quad-vector groups.
- ZA.T[Wv, offs1:offs4, VGx4], where offs1 is 0 or 4, when the operand consists of four quad-vector groups.

Chapter B2. Architectural state B2.4. SME2 Multi-vector operands

IThe 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:



 $\mathbb{I}_{\text{KBMLX}}$ The mapping between ZA multi-vector o_{F} and s of quad-vector groups and 64-bit element ZA tile slices when SVL is 256 bits industrate in the following diagram:



B2.5 SME2 Multi-vector predication

- D_{JFJZX} SME2 introduces the multi-vector predication concept in *Streaming SVE mode*, named *predicate-as-counter*.
- I_{LLLXP} The existing SVE predication concept is referred to as *predicate-as-mask*.
- I_{DSFKR} SME2 multi-vector instructions interpret the lowest-numbered 16 bits of SVE predicate registers *P0-P15* as a *predicate-as-counter* encoding.
- D_{QBDRH} A *predicate-as-counter* encoding includes:
 - An invert bit, that encodes whether the element count field is referring to the number of TRUE or FALSE predicate elements.
 - A variable-width element count field. This field holds an unsigned integer value of up to 14 bits, that encodes:
 - When the invert bit is 0, the number of consecutive elements of the mean of that are TRUE, with the remaining elements being FALSE.
 - When the invert bit is 1, the number of consecutive eleme. starting from 'ement 0 that are FALSE, with the remaining elements being TRUE.
 - A variable-width element size field of up to 4 bits, wher the number trailing zeroes encodes LSZ, log₂ of the element size in bytes, or an all-FALSE predicate all 4 bits the zero.

[15] Invert	[14:(LSZ+1)] Element count	[LSZ:0] .nem. t size	Meing
Х	*****	1	Byte elements, count in [14:1]
Х	xxxxxxxxx. ′x		Halfword elements, count in [14:2]
Х	хххху ххххх	10.	Word elements, count in [14:3]
Х	x xxxxx	000	Doubleword elements, count in [14:4]
x	XXXXXXX. 'X	0000	All-FALSE predicate (any element size)

- D_{JGSYR} The canonical al RUE *p icate-as-counter* encoding has an element count of zero, with the invert bit set to 1 and a nonzero elen. * *i* e field de rmined by the generating instruction.
- D_{BJMYH} The can can FALS. *redi le-as-counter* encoding has an element count of zero, with the invert bit set to 0 and *r* element ze field s. J 0b0000.
- $\mathbb{I}_{\mathbb{ZPLKP}} \qquad \text{A pre-rate s-count} \quad \text{ncoding can represent a consecutive element count in the range of 0 to the maximum number syste elements in four vectors, minus 1. The architectural maximum vector length of 2048 bits or 256 bytes there shares an element count of <math>\log_2(1024) = 10$ bits, plus one element size bit, plus the invert bit. The additional 4 bit in the element count field are reserved.
- D_{HWRFM} In assembler syntax:
 - The name *Pg* is used for *predicate-as-mask*.
 - The name *PNg* is used for *predicate-as-counter*.

Both Pg and PNg refer to the same predicate register.

- R_{YCRMW} If VL is greater than 128 bits, then an instruction which writes a *predicate-as-counter* encoding to a predicate register sets bits 16 and higher of that register to 0.
- R_{SGVTC} If VL is greater than 128 bits, then an instruction which reads a predicate register using the *predicate-as-counter* encoding ignores bits 16 and higher of that register.
- R_{YSTVC} An instruction uses only the least significant bits in the element count field of the *predicate-as-counter* register that are required to represent the number of bytes in the current vector length times four, minus 1. The instruction ignores the more significant bits in the element count field.

For example, when VL is 512 bits there are 256 byte elements in four vectors, so the *predicate-as-counter* encoding ISKKTX requires at most an 8-bit element count field [8:1], a 1-bit size field [0], and the invert bit [15]. Therefore, when VL is 512 bits, an instruction uses bits [15] and [8:0] from the predicate-as-counter register and ignores bits [14:9] and [63:16]. The SME2 WHILE instructions generate a predicate-as-counter encoding. These instructions have an operand that IDKPNO 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. A canonical all-TRUE encoding is generated when the number of TRUE elements is equal to or exceeds the limit of the number of elements in a vector times the number of vectors. The canonical all-FALSE encoding is generated when the number of TR^V . etc., ts is zero. The SME2 *PTRUE* instruction generates a canonical all-TRUE *pred[:] te-as-counter* coding. I JBYVM The SVE *PFALSE* instruction generates the canonical all-FALS^r predic. -as-counte encoding. IBXWKF The SME2 PEXT instruction converts a predicate-as-counter acoding into a di e-as-mask encoding. Since a ITRVOO predicate-as-counter encoding allows more predicate e' ents the can be represented in a predicate-as-mask encoding, this instruction takes an operand to extract da action of a wider mask corresponding to a predicate-as-counter encoding. The SME2 CNTP instruction converts a predic 2-as-cou er encod. Into a total Active element count value that IOSHRN is placed in a general-purpose register. CNTP has an operand that indicates the 'mit of number of elements to be counted. The limit corresponds to the total number of elements in eith 2 orvector Predicated SME2 multi-vector structions in. pret the value in their Governing predicate register using the IZMDPF predicate-as-counter encore g to 'etermine the umber and size of consecutive Active elements. When the

predicate-as-counter encoding to bettermine the number and size of consecutive Active elements. When the element size of the instruction operation is different from the element size in the *predicate-as-counter* encoding, the number of Active elements of the instruction operation is also different from the number of *predicate-as-counter* Active elements.

See also:

- CNT
- P^r xT (pr icate).
- EXT (p^r licate pair).
- **FALS**
- P1 É.
- WHIL 'GE.
- WHILE
- WHILEHI.
- WHILEHS.
- WHILELE.
- WHILELO.
- WHILELS.
- WHILELT.

B2.6 SME2 Lookup table

D_{KHZMV} When SME2 is implemented, a PE has a 512-bit architectural register *ZT0* to support the lookup table feature.

- D_{WDJBC} The ZTO 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 lookup table in the *ZT0* register can be accessed using fully packed 2-bit or 4-bit indices from a numbered portion of one source *Z* vector register.
- I_{BRRGG} When the lookup table *ZT0* is addressed by 2-bit indices, four different table elements (0-3) of a given element size can be accessed. When the lookup table *ZT0* 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 *ZT* .cgr. and packed into consecutive elements of an SVE *Z* vector or *Z* multi-vector operand.
- I_{HQTSR} The validity and accessibility of the ZTO register are enabled by PSI. 7.ZA. For m e information, see B1.2 Process state and B1.2.2 PSTATE.ZA.

See also:

- C1.1.2 Traps and exceptions.
- C1.4.8 *SVCR*.

Chapter B3 Floating-point behaviors

B3.1 Overview

SME modifies so c of the profile floating-point behaviors when a PE is in *Streaming SVE mode*, and introduces an FPCR control w. he and BFloat16 dot product calculations to support a wider range of numeric behaviors.



B3.2 Supported floating-point data types

R_{vschz}

The following BFloat16 instructions operate on the BFloat16 and the IEEE 754-2008 Single-precision floating-point data types, as defined respectively in sections *BFloat16 floating-point format* and *Single-precision floating-point format* of *Arm[®] Architecture Reference Manual for A-profile architecture* [1]:

- The SME BEMOPA and BEMOPS floating-point instructions defined in D1.1 SME and SME2 data-processing instructions.
- The SME2 multi-vector BFCVT, BFCVTN, BFDOT, BFMLAL, BFMLSL, and BFVDOT floating-point instructions defined in D1.1 *SME and SME2 data-processing instructions*.
- The SVE2 BFMLSLB and BFMLSLT floating-point instructions that are introduced by SME2 and defined in D1.2 *SVE2 data-processing instructions*.

 $\mathsf{R}_{\texttt{BYPSW}}$

- The floating-point instructions defined in Chapter D1 *SME instructions* operations (IEEE 754-2008 floating-point data types as defined in the following *Arm[®] Architecture Reference Mc* al for *A-pr*, *'e architecture* [1] sections:
 - Half-precision floating-point formats (but not the Arm alternative olf-precision ormat).
 - Single-precision floating-point format.
 - Double-precision floating-point format.

B3.3 BFloat16 behaviors

If FEAT EBF16 is implemented, the Extended BFloat16 behaviors can be enabled for the BFloat16 instructions. This section describes how the BFloat16 instruction behaviors are changed by FEAT EBF16.

When ID_AA642FR0_EL1.BF16 and ID_AA64ISAR1_EL1.BF16 have the value 0b0010, the PE implements R_{BSHYK} FEAT EBF16 and supports the FPCR.EBF control.

- If FEAT_EBF16 is implemented, then: R_{RKGSJ}
 - FEAT_EBF16 is enabled when FPCR.EBF is 1.
 - FEAT_EBF16 is not enabled when FPCR.EBF is 0.

Unless stated otherwise, the rules in this section describe the behaviors of the following instructions: D_{MVPSG}

- The SME BEMOPA and BEMOPS instructions.
- The SME2 BFDOT and BFVDOT instructions.
- The Advanced SIMD and SVE BFDOT and BFMMLA instructions.

B3.3.1 Common BFloat16 behaviors

- The Common BFloat16 behaviors are the behaviors currently and in A^{\prime} Architecture Reference Manual for ICYTKF A-profile architecture [1] which are not changed ', the tional. SR.J control.
- The instructions specified in D_{MVPSG} that tect exc tional floating-point conditions produce the expected R_{RDCPW} single-precision default result but do no mo ly une .nulative floating-point exception flag bits, FPSR. {IDC, IXC, UFC, OFC, DZC, IOC}.
- The instructions specified in D_M, genera, lefault N values, behaving as if FPCR.DN has an Effective value R_{PFFFF} of 1.

See also:

- FPSR, Floating-r ... nt Status Registe Arm[®] Architecture Reference Manual for A-profile architecture [1]. • FPCR.

B3.3.2 Standard PT'nat1 ehavio

The andard F loat16 beh. viors are the behaviors currently defined in Arm[®] Architecture Reference Manual for ICMSBQ *A-propert are neurous* which can be changed by the FPCR.EBF control provided by FEAT_EBF16.

- If FEAT __________F16 is either not implemented or not enabled, then the instructions specified in D_{MVPSG} ignore the R_{JFWDV} FPCR.RMode ntrol and use the rounding mode defined for BFloat16 in section Round to Odd mode of Arm® Architecture Reference Manual for A-profile architecture [1].
- If FEAT_EBF16 is either not implemented or not enabled, then the instructions specified in D_{MVPSG} flush R_{WBLQD} denormalized inputs and outputs to zero, behaving as if FPCR.FZ has an Effective value of 1.

If FEAT_EBF16 is either not implemented or not enabled, then the instructions specified in D_{MVPSG} perform R_{JPNSN} unfused multiplies and additions with intermediate rounding of all products and sums.

B3.3.3 Extended BFloat16 behaviors

- The Extended BFloat16 behaviors are the behaviors that can be enabled by the FPCR.EBF control provided by IMFJMJ FEAT EBF16.
- If FEAT_EBF16 is implemented and enabled, then the instructions specified in D_{MVPSG} support all four IEEE 754 R_{RQKQZ} rounding modes selected by the FPCR.RMode control.

Chapter B3. Floating-point behaviors B3.3. BFloat16 behaviors

- R_{JZVPD} If FEAT_EBF16 is implemented and enabled, then the instructions specified in D_{MVPSG} honor the FPCR.FZ control.
- $R_{LJGTX} \qquad \mbox{If FEAT_EBF16 is implemented and enabled, then the instructions specified in D_{MVPSG} perform a fused two-way sum-of-products for each pair of adjacent BFloat16 elements in the source vectors, without intermediate rounding of the products, but rounding the single-precision sum before addition to the single-precision accumulator element.$
- R_{CQFQT} If FEAT_EBF16 is implemented and enabled, then the instructions specified in D_{MVPSG} generate the default NaN as intermediate sum-of-products when any of the following are true:
 - Any multiplier input is a NaN.
 - Any product is infinity $\times 0.0$.
 - There are infinite products with differing signs.
- R_{WQDBR} If FEAT_EBF16 is implemented and enabled, then the instructions specified in D_{MVPSG} generate an intermediate sum-of-products of the same infinity when there are infinite products all with the same sign.
- RYPGLE
 When FEAT_AFP is implemented and FEAT_EBF16 is implemented a enable. the instructions specified in D_{MVPSG} honor the FPCR.AH and FPCR.FIZ controls.
- I_{WKNML} When FEAT_AFP is implemented and FEAT_EBF16 is implemented on the following alternate floating-point behaviors affect the instructions specified in D __PSG:
 - When FPCR. AH is 1, the sign bit of a generated defa NaN rent is set to instead of 0.
 - When FPCR.AH is 1 and FPCR.FZ is 1, a denormal result of the definition of the definition of the second s
 - When FPCR. AH is 1, the FPCR. FZ control d's not use den me zed inputs to be flushed to zero.
 - When FPCR.FIZ is 1, all denormalized puts are f .shed to zer .

B3.4 Floating-point behaviors in Streaming SVE mode

 D_{DMPBW} Unless stated otherwise, the rules in this section describe the behaviors of the following instructions:

- The floating-point instructions that are legal in Streaming SVE mode, and operate on half-precision, single-precision, and double-precision input data types, placing their results in SIMD&FP registers or SVE *Z* vector registers.
- The SVE $\ensuremath{\mathsf{BFMLALB}}$ and $\ensuremath{\mathsf{BFMLALT}}$ instructions.
- The floating-point instructions introduced by SME2 that place their results in one or more SVE Z vector registers:
 - The BFCVT, BFCVTN, FCLAMP, FCVT, FCVTN, FCVTZS, FCVTZU, FMAX, FMAXNM, FMIN, FMINNM, FRINTA, FRINTM, FRINTN, FRINTP, SCVTF, and UCVTF instructions, as defined in D1.1 SME and SME2 data-processing instructions.
 - The BFMLSLB, BFMLSLT, FDOT, and FCLAMP instructions, as fined in 1.2 SVE2 data-processing instructions.
- RPHDZLWhen the PE is in Streaming SVE mode, the instructions specifier in DDM.w honor thNon-streaming scalar andSVE floating-point behaviors, as governed by the FPCR. {DN, *, RMode, FZ1H, FTcontrols.
- R_{GTYSK} When the PE is in *Streaming SVE mode*, the instructions scified in _{DMPBW} the Jetect exceptional floating-point conditions produce the expected default result and can upde the ppropriate cumulative floating-point exception flag bits in FPSR. {IDC, IXC, UFC, OFC, DZC, IOC}.
- R_{PYSCC} The floating-point behaviors followed by the CLAMP is truction .dentical to the behaviors followed when executing FMAXNM and FMINNM in order.
- R_{FBFNT} When the PE is in *Streaming SVE mod*, nd FE, *SME_FA64* is not implemented or not enabled at the current Exception level, the *Effective value* of the pcR is . if all of the IDE, IXE, UFE, OFE, DZE, and IOE floating-point exception trap enable controls, ar the NEP element preve control, are 0 for all purposes other than a direct read or write of the register.

See also:

- FPSR, Floating Int Status Register Arm[®] Architecture Reference Manual for A-profile architecture [1].
- FPCR.

B3.5 Floating-point behaviors targeting the ZA array

D _{HTZVK}	Unless stated otherwise, the rules in this section describe the behaviors of the SME and SME2 floating-point instructions that place their results in the ZA array, except BFMOPA, BFMOPS, BFDOT, and BFVDOT.
	For the behaviors of the BFloat16 instructions, see B3.3 BFloat16 behaviors.
R _{tgskg}	The instructions specified in D_{HTZVK} that detect exceptional floating-point conditions produce the expected IEEE 754 default result but do not modify any of the cumulative floating-point exception flag bits, FPSR. {IDC, IXC, UFC, OFC, DZC, IOC}.
$R_{\rm RKHHZ}$	The instructions specified in D_{HTZVK} generate default NaN values, behaving as if FPCR.DN has an Effective value of 1.
R_{TCLRM}	The instructions specified in D_{HTZVK} support all four IEEE 754 round: selected by the FPCR.RMode control.
R _{VVVNR}	The instructions specified in D _{HTZVK} honor the FPCR.FZ control.
R _{txkvk}	The instructions specified in D_{HTZVK} that accumulate dot projects of pairs adjace chalf-precision elements in the source vectors into single-precision elements in the ZA ray hop the FPC 16 control.
R_{JRRMJ}	The instructions specified in D_{HTZVK} that multiply single e. per from eac' source vector and accumulate their product into the ZA array perform a fused multiply to eac. ccumula tile or multi-vector operand element without intermediate rounding.
R _{NNCFV}	The instructions specified in D_{HTZVK} that a simulate standucts of pairs of adjacent half-precision elements in the source vectors into single-precision element in the ZA array perform a fused sum-of-products without intermediate rounding of the products, b rounding the single-precision sum before addition to the accumulator tile or multi-vector operand eleme .
R _{qpkjc}	The instructions specified in $\mathcal{L}_{\mathbf{n}} \to \mathcal{K}$ that accumulate dot products of pairs of adjacent half-precision elements in the source vectors into angle-public sign elements in the ZA array generate the default NaN as intermediate sum-of-products when \mathcal{L} y of the following are true:
	 Any multiplic input is a NaN. Any products infinit < 0.0. There are init. terminducts with differing signs.
R_{ZBLND}	The instruction specify in D_{ZVK} that accumulate dot products of pairs of adjacent half-precision elements in the source vector into sing precision elements in the ZA array generate an intermediate sum-of-products of the sanging infinity in there are infinite products all with the same sign.
R _{RPSLK}	When FAFP is implemented, the instructions specified in D_{HTZVK} honor the FPCR.AH and FPCR.FIZ controls.
I _{YPCHJ}	When FEA1_ FP is implemented, the following alternate floating-point behaviors affect the instructions specified in D_{HTZVK} :
	 When FPCR.AH is 1, the sign bit of a generated default NaN result is set to 1 instead of 0. When FPCR.AH is 1 and FPCR.FZ is 1, a denormal result, detected after rounding with an unbounded exponent has been applied, is flushed to zero.

- When FPCR.AH is 1, the FPCR.FZ control does not cause denormalized inputs to be flushed to zero.
- When FPCR.FIZ is 1, all denormalized single-precision and double-precision inputs are flushed to zero.

Part C هد `vsi、m Level Programmers' Model

Chapter C1 System management

C1.1 Overview

I_{ZTOKZ} The SME Syster Janager At architecture provides mechanisms for system software to:

- Dis ver e capa 'ities' SME.
- / Jntrol S E usage.
- Monitor ME usage.

The arc. cture consists of extensions to processor mode, the Exception model, and System registers for trap control and 'entification.

I_{PZVYW} SME extends the AArch64 System registers and processor state, by introducing the following:

- An SME presence identification field added to ID_AA64PFR1_EL1.
- An SME-specific ID register, ID_AA64SMFR0_EL1, for SME feature discovery.
- Configuration settings for SME in CPACR_EL1, CPTR_EL2, CPTR_EL3, HCR_EL2, HCRX_EL2, HFGRTR_EL2, and HFGWTR_EL2.
- An SME exception type, with new ESR_ELX.EC and ESR_ELX.ISS encodings.
- A field in ESR_ELX.ISS to signal that an imprecise FAR_ELX value has been reported on a synchronous Data Abort exception.
- SME controls to set the Effective Streaming SVE vector length in SMCR_EL1, SMCR_EL2, and SMCR_EL3.
- ID and control registers that influence streaming execution priority in multiprocessor systems: SMIDR_EL1, SMPRI_EL1, and SMPRIMAP_EL2.
- Fields that enable the software thread ID register TPIDR2_EL0 in SCR_EL3, SCTLR_EL2, and SCTLR_EL1.

I_{HVHCP} SME2 extends the SME System registers as follows:

- The SMTC exception syndrome field for an exception due to SME functionality in ESR_EL1, ESR_EL2, and ESR_EL3 is extended to identify trapping of accesses to the ZTO register.
- A value is added to the ID_AA64PFR1_EL1.SME field, to identify the presence of the ZT0 register.
- An SMEVER field is added to ID_AA64SMFR0_EL1 to indicate the presence of the mandatory SME2 instructions.
- The existing F64F64 and I16I64 fields in ID_AA64SMFR0_EL1 are extended to cover multi-vector instructions that generate double-precision and 64-bit integer results.
- A BI32I32 field is added to ID_AA64SMFR0_EL1 to identify the presence of SME2 instructions that accumulate thirty-two 1-bit binary outer products into 32-bit integer tiles.
- A 116132 field is added to ID_AA64SMFR0_EL1 to identify the presence of SME2 instructions that accumulate 16-bit outer products into 32-bit integer tiles.
- An EZTO field is added to the SMCR_EL1, SMCR_EL2, and SMCR_EL3 registers, to enable access to the ZTO register.

See also:

- C1.3.1 CPACR_EL1.
- C1.3.2 *CPTR_EL2*.
- C1.3.3 *CPTR_EL3*.
- C1.3.4 ESR EL1, ESR EL2, and ESR EL3.
- C1.3.5 HCR EL2.
- C1.3.6 HCRX EL2.
- C1.3.7 *HFGRTR EL2*.
- C1.3.8 *HFGWTR_EL2*.
- C1.3.9 *ID_AA64PFR1_EL1*.
- C1.3.11 SCR EL3.
- C1.3.12 SCTLR EL1.
- C1.3.13 *SCTLR_EL2*.
- C1.4.1 *ID_AA64SMFR0_E*
- C1.4.2 *SMCR_EL1*.
- C1.4.3 SMCR_EL2.
- C1.4.4 SMCR_EL3.
- C1.4.5 *SMIDR* _1.
- C1.4.6 SMP^r EL1.
- C1.4.7 SM </br>
- C1.4.8 SVC

C1.1.1 Identif ,ation

IXTNNP ID_AA6. 1_EL1. SME indicates whether SME is implemented in a PE.

I_{RLFXX} If SME is in the sME features that are implemented in a PE are determined from ID_AA64SMFR0_EL1. See also:

- C1.3.9 *ID_AA64PFR1_EL1*.
- C1.4.1 *ID_AA64SMFR0_EL1*.

C1.1.2 Traps and exceptions

D_{DMBHW}

The SME-related instructions that can be configured to trap by CPACR_EL1, CPTR_EL2, and CPTR_EL3 controls, unless otherwise stated, include all of the following:

- SME data-processing instructions.
- SME mode change instructions SMSTART and SMSTOP.
- AArch64 MRs and MSR instructions which directly access any of the SVCR, SMCR_EL1, SMCR_EL2, or SMCR_EL3 registers.

Chapter C1. System management C1.1. Overview

The following SME2 instructions that access the ZTO register can be configured to trap by SMCR_ELX.EZTO controls: D_{RGBVJ} • LDR (ZT0). • LUTI2, LUTI4. • MOVT. • STR (ZT0). • ZERO (ZTO). Execution of SME-related instructions can be trapped by supervisor software. The mechanisms provided are: IMOBEG • CPACR_EL1, which enables execution of SME-related instructions at EL0 or EL1 to be trapped to EL1 or EL2. • CPTR_EL2, which enables execution of SME-related instructions at EL0, EL1 or EL2 to be trapped to EL2. • CPTR_EL3, which enables execution of SME-related instructions at any Exception level to be trapped to EL3. Execution of SME2 instructions that access ZTO can be trapped by supervisor software. The mechanisms provided ITLWWF are: • SMCR_EL1.EZTO, which enables execution of SME2 instructions at access ZTO EL0 or EL1 to be trapped to EL1 or EL2. • SMCR_EL2.EZTO, which enables execution of SME2 instr Jons that cess ZT/ it ELO, EL1, or EL2 to be trapped to EL2. • SMCR_EL3.EZTO, which enables execution of SME2 structior .nat access .10 at any Exception level to be trapped to EL3. SME adds an exception syndrome value 0b01110⁻ JXL, which, used identify instructions that are trapped by IXKMKY any of the following: • The SME controls in CPACR_EL1, CPTR_ ap∕i • The SME2 controls SMCR_EL1.EZ. .EZTO, and SMCR_EL3.EZTO. SMCR_ • The PSTATE.SM and PSTATE.7 mod. Exceptions reported with the evision syndron. value ob011101 (0x1D) are mapped onto the Trap type in section IFWJZB Exceptions to Exception eler ... t.e. oding of chap. The Embedded Trace Extension and section Filtering on type of chapter The Branch Record Buffer rension in Arm[®] Architecture Reference Manual for A-profile architecture [1]. See also: • C1.2.1 Except on corities. "PACR_ 1. C1 2 € .3.2 C. TR_EL∠ C1.3.3 C *IR_EL3*. SR_EL2, and ESR_EL3. 1.3.4 .on__ 2 SMCR_EL1. • (C1.4. SMCR EL2. • C1.4.4 CR_EL3. C1.1.3 Vector lengths

DQORNRThe Effective Streaming SVE vector length (SVL) is the accessible length in bits of the ZA array vectors and
Streaming SVE vector registers at the current Exception level. SVL is determined by the LEN field of the appropriate
SME SMCR_EL* registers, as defined in rule RGWVHP.IBHFWGSVL is used explicitly by the unpredicated SME LDR (vector), STR (vector), and ZERO (tile) instructions which can
access the ZA array irrespective of whether the PE is in Streaming SVE mode.RVCQBBThe Effective SVE vector length (VL) is equal to SVL when the PE is in Streaming SVE mode.I_LRBQYVL is determined by the LEN field of the ZCR_EL* registers when the PE is not in Streaming SVE mode and
FEAT_SVE is implemented.

Chapter C1. System management C1.1. Overview

R _{JRCSH}	An implementation is permitted to support any subset of the architecturally defined SVL values.
I _{FQKMN}	For example, this means that the set of supported SVLs might be discontiguous and might not start at the smallest permitted SVL.
R _{RZNVH}	An implementation is permitted but not required to support more than one SVL.
R _{WDKGR}	An implementation is permitted to support a set of SVLs that do not overlap with the set of VLs that are supported when the PE is not in <i>Streaming SVE mode</i> .
I _{GZRPL}	There is no requirement for the Maximum implemented Streaming SVE vector length to be greater than or equal to the Maximum implemented SVE vector length.
R _{gwvhp}	The Effective Streaming SVE vector length at a given Exception level is determined from the requested length, encoded as a multiple of 16 bytes in SMCR_EL1.LEN, SMCR_EL2.LEN, or SMCR_EL3.LEN, according to the Exception level, following these steps:
	 If the requested length is less than the minimum implemented S caming SVL vector length, the Effective length is the minimum implemented Streaming SVE vector length. If this is the highest implemented Exception level and the reques of length is cleater than the maximum implemented Streaming SVE vector length, then the Effective length is cleater than the maximum SVE vector length. If this is not the highest implemented Exception level and the requested length is greater than the Effective length at the next more privileged implemented Fxception level and the requested length is greater than the Effective length at the more privileged Exception level as used. If the requested length is not supported by the inplementation, then the Effective length is the highest supported Streaming SVE vector lengt, that is least the requested length. Otherwise, the Effective length is the requester that is least the requested length.
I _{vrryr}	The set of supported values of SVL Exc. ion lev ELx (where ELx is EL1, EL2 or EL3) can be discovered by privileged software in a similar v f to detern. ing the t of supported values of VL. For example, when SME is enabled by the appropriate co fields in CPAC EL1, CPTR_EL2 and CPTR_EL3:
	 Request an out of range vector with of 8192 bytes by writing 0x1ff to SMCR_ELx[8:0]. Use the SME RD we instruction to wid SVL. If SMCR_ELx requests a supported Streaming SVE vector length, the requested length in bytes will be returned by RDSVL. If SMCR_ELx we as an unsw ported Streaming SVE vector length, a supported length in bytes will be returned by RDSV. If smcR_ELx we as an unsw ported Streaming SVE vector length, a supported length in bytes will be returned by RDSV. If smcR_ELx we as an unsw ported Streaming SVE vector length, a supported length in bytes will be returned by RDSV. If we returned by RDSV. If we returned length of us less than or equal to the requested vector length, and greater than 16 bytes (128 ofts), there equest the we return length by writing (len/16)-2 to SMCR_ELx[8:0] and go to step 2.
R _{yrpdh}	When T nanges from a smaller to a larger value without leaving <i>Streaming SVE mode</i> , a new area of storage becomes hitecturally visible in the Streaming SVE registers and, if enabled, the <i>ZA</i> storage. The values in the area commo. The previous and current length are preserved, and the values in the newly accessible area are a CONSTRAINED UNPREDICTABLE choice between the following:
	 Zero. The value the bits had before executing at the more constrained size.
I _{hgypv}	The SVL might change without leaving <i>Streaming SVE mode</i> because of an explicit action such as a write to SMCR_ELx, or an implicit action such as taking an exception to an Exception level with a less constrained SVL.
I _{PDLWX}	The SVL can be raised and then restored to a previous value without affecting the original contents of the Streaming SVE registers and the ZA storage.
I _{FMWZZ}	Supervisory software must guarantee that values generated by one body of software are not observable by another body of software in a different trust or security scope. When SVL is increased, steps must be taken to ensure the newly accessible area does not contain values unrelated to another body of software. This might be achieved by ensuring that the PE exits <i>Streaming SVE mode</i> and disables the ZA storage when performing a context switch, or by explicitly resetting all register values.

Chapter C1. System management C1.1. Overview

I_{BRMMV} System software provides a maximum SVL to lower-privileged software, which might further constrain the SVL. However, system software must initialize and context switch values consistent with the maximum SVL provided and should not make assumptions about any smaller size being in use by lower-privileged software. For example, if a hypervisor exposes an SVL of 512 to a VM, that VM might choose to constrain SVL to 256. The hypervisor should 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:

- C1.4.2 SMCR_EL1.
- C1.4.3 *SMCR_EL2*.
- C1.4.4 *SMCR_EL3*.

C1.1.4 Streaming execution priority

- D_{DXMSW} Streaming execution refers to the execution of instructions by a ^r ^s whe. bat PE is in *treaming SVE mode*.
- I_{CMRVS} Arm expects a variety of implementation styles for SME, ir uding styles wire right retaining SVE compute resources.
- I_{PQNJS} Shared SME and Streaming SVE compute resources are can *streaming lode Compute Unit* (SMCU).
- I_{MZXWD} For implementations that share an SMCU, this *p* nitect *e* prover *p P E* mechanisms that software can use to dynamically prioritize performance character acts expended by *e* on PE.

See also:

• C1.2.4 Streaming execution priority r share implementations.

C1.2 Processor behavior

C1.2.1 Exception priorities

- I_{QZNRN} I_{ZFGJP} in the *Prioritization of Synchronous exceptions taken to AArch64 state* section of *Arm[®] Architecture Reference Manual for A-profile architecture* [1] provides a full list of exception priorities. The rules in this section provide additional detail for the prioritization of SME-related exceptions.
- R_{GTKQD} Exceptions due to configuration settings and modes resulting from the attempted execution of an SME data-processing instruction are evaluated in the following order from highest to lowest priority:
 - 1. If FEAT_SME is not implemented, then SME and SME2 instructions are UNDEFINED.
 - 2. If FEAT_SME2 is not implemented, then SME2 instructions are UNDETWED.
 - 3. If CPACR_EL1.SMEN configures the instruction to trap, it is reported sing E. ELX.EC value 0x1D with ISS code 0x0000000.
 - 4. If CPACR_EL1.FPEN configures the instruction to trap, it is reporte vsing ESR_EL EC value 0x07.
 - 5. If CPTR_EL2.SMEN configures the instruction to trap, it is _ported ing ESR_F ... EC value 0x1D with ISS code 0x0000000.
 - 6. If CPTR_EL2.FPEN configures the instruction to trap is report . using EL_ELX.EC value 0x07.
 - 7. If CPTR_EL2.TSM configures the instruction to trap, it provide using FOR_ELX.EC value 0x1D with ISS code 0x00000000.
 - 8. If CPTR_EL2.TFP configures the instruction trap, is rep. od r ig ESR_ELX.EC value 0x07.
 - 9. If CPTR_EL3.ESM configures the instruction to trap, is reported using ESR_ELX.EC value 0x1D with ISS code 0x0000000.

 - 11. If the PE is not in *Streaming SVE no* SME 1 SME2 instructions that access the SVE registers Z0-Z31 or *P0-P15* generate an SME exception, rejected using the segmentation of the segm
 - 12. If the ZA storage is disab' SME and SN. ? instructions that access ZA and SME2 instructions that access ZTO generate an SMF Acepu. reported us. g ESR_ELX.EC value 0x1D with ISS code 0x0000003.
 - 13. If accesses to ZTO re not enable occording to R_{CSPYJ}, then the SME2 instructions that access ZTO generate an SME except; , reported using L ELX.EC value 0x1D with ISS code 0x0000004.
 - 14. Otherwise, t¹ instructⁱ n executes.
- R_{XQKHH} Exceptions due to onfinitation settings resulting from attempted execution of MRS or MSR instructions which directly accorriging one one of e SVCR, or R_EL1, SMCR_EL2, or SMCR_EL3 registers, are evaluated in the following order from hister to set to se
 - 1 f FEAT ME is not implemented, then the instruction is UNDEFINED.
 - 2. **PP** _EL1.SPM_ configures the instruction to trap, it is reported using ESR_ELX.EC value 0x1D, with ISS coc. x0000000.
 - 3. If CPTL TL2.SMEN configures the instruction to trap, it is reported using ESR_ELX.EC value 0x1D, with ISS code 0x00,00000.
 - 4. If CPTR_EL2.TSM configures the instruction to trap, it is reported using ESR_ELX.EC value 0x1D, with ISS code 0x00000000.
 - 5. If CPTR_EL3.ESM configures the instruction to trap, it is reported using ESR_ELx.EC value 0x1D, with ISS code 0x00000000.
 - 6. Otherwise, the instruction executes.
- R_{PLYVH} Exceptions due to configuration settings and modes resulting from the attempted execution of an SVE or SVE2 instruction when FEAT_SVE is not implemented or when the PE is in *Streaming SVE mode*, are evaluated in the following order from highest to lowest priority:
 - 1. If FEAT_SME is not implemented and FEAT_SVE is not implemented, then the instruction is UNDEFINED.
 - 2. If FEAT_SVE is not implemented and the instruction is illegal when the PE is in *Streaming SVE mode*, then the instruction is UNDEFINED.

- 3. If FEAT_SME is not implemented and the instruction is defined as part of the SME architecture in D1.2 *SVE2 data-processing instructions*, then the instruction is UNDEFINED.
- 4. If CPACR_EL1.SMEN configures the instruction to trap, it is reported using ESR_ELX.EC value 0x1D with ISS code 0x0000000.
- 5. If CPACR_EL1.FPEN configures the instruction to trap, it is reported using ESR_ELX.EC value 0x07.
- 6. If CPTR_EL2.SMEN configures the instruction to trap, it is reported using ESR_ELX.EC value 0x1D with ISS code 0x0000000.
- 7. If CPTR_EL2.FPEN configures the instruction to trap, it is reported using ESR_ELX.EC value 0x07.
- 8. If CPTR_EL2.TSM configures the instruction to trap, it is reported using ESR_ELX.EC value 0x1D with ISS code 0x00000000.
- 9. If CPTR_EL2.TFP configures the instruction to trap, it is reported using ESR_ELX.EC value 0x07.
- 10. If CPTR_EL3.ESM configures the instruction to trap, it is reported using ESR_ELX.EC value 0x1D with ISS code 0x00000000.
- 11. If CPTR_EL3.TFP configures the instruction to trap, it is reported using LSK_ ... EC value 0x07.
- 12. If the PE is in *Streaming SVE mode* and the SVE instruction is *ill d* in that mother then an SME exception is taken, using ESR_ELX.EC value 0x1D with ISS code 0x0000001.
- 13. If the PE is not in *Streaming SVE mode* and FEAT_SVE is r timple anted, then a SME exception is taken, using ESR_ELX.EC value 0x1D with ISS code 0x0000000
- 14. Otherwise, the instruction executes.

R _{ztkxf}	Exceptions due to configuration settings resulting from the 'ter jued exection of an SVE or SVE2 instruction
	when FEAT_SVE is implemented and when the PFt in Sming S'mode, are evaluated in the following
	order from highest to lowest priority:

- 1. If FEAT_SME is not implemented an, be instruction is defined as part of the SME architecture in D1.2 *SVE2 data-processing instructions* then a incruction is oNDEFINED.
- 2. If CPACR_EL1.ZEN configures the in viction 'rap, it is reported using ESR_ELX.EC value 0x19.
- 3. If CPACR_EL1.FPEN configur the instantion to p, it is reported using ESR_ELX.EC value 0x07.
- 4. If CPTR_EL2.ZEN configur the instruction to trap, it is reported using ESR_ELX.EC value 0x19.
- 5. If CPTR_EL2.FPEN con^r, are. `e instruction trap, it is reported using ESR_ELX.EC value 0x07.
- 6. If CPTR_EL2.TZ configures the . truction to trap, it is reported using ESR_ELX.EC value 0x19.
- 7. If CPTR_EL2.TFF onfigures the in. to trap, it is reported using ESR_ELX.EC value 0x07.
- 8. If CPTR_EL3, configures the instruction to trap, it is reported using ESR_ELX.EC value 0x19.
- 9. If CPTR_EL TFP con^c ares the instruction to trap, it is reported using ESR_ELX.EC value 0x07.
- 10. Otherwise, the 'ns' action executes.
- R_{DTCLZ} Exception on to contract settings and modes resulting from the attempted execution of an AArch64 Adverted SIM and floc g-point instruction when the PE is in *Streaming SVE mode* are evaluated in the folic sing ord for highest to lowest priority:
 - 1. If CR_EL1.FPEN configures the instruction to trap, it is reported using ESR_ELX.EC value 0x07.
 - 2. If CP1 EL2.FPEN configures the instruction to trap, it is reported using ESR_ELX.EC value 0x07.
 - 3. If CPTR___2.TFP configures the instruction to trap, it is reported using ESR_ELX.EC value 0x07.
 - 4. If CPTR_EL3.TFP configures the instruction to trap, it is reported using ESR_ELX.EC value 0x07.
 - 5. If the instruction is *illegal* when the PE is in *Streaming SVE mode*, then an SME exception is taken, using ESR_ELX.EC value 0x1D with ISS code 0x0000001.
 - 6. Otherwise, the instruction executes.
- INWNQZ
 When the PE is in Streaming SVE mode or FEAT_SVE is not implemented, the CPACR_EL1.SMEN, CPTR_EL2.SMEN, CPTR_EL2.TSM, and CPTR_EL3.ESM controls configure SVE instructions to trap, and the CPACR_EL1.ZEN, CPTR_EL2.ZEN, CPTR_EL2.TZ, and CPTR_EL3.EZ controls do not cause any SVE instructions to be trapped.
- IPKGPR
 When the PE is not in Streaming SVE mode and FEAT_SVE is implemented, the CPACR_EL1.ZEN, CPTR_EL2.ZEN, CPTR_EL2.ZEN, and CPTR_EL3.EZ controls configure SVE instructions to trap, and the CPACR_EL1.SMEN, CPTR_EL2.SMEN, CPTR_EL2.TSM, and CPTR_EL3.ESM controls do not cause any SVE instructions to be trapped.
- R_{ZZBRC} An Undefined Instruction exception due to the pairing of an SVE MOVPRFX with an instruction which cannot be predictably prefixed has a higher exception priority than a PSTATE mode-dependent SME exception with
ESR_ELX.EC value 0x1D and an ISS code that is not 0x0000000.

See also:

- Prioritization of Synchronous exceptions taken to AArch64 state in Arm[®] Architecture Reference Manual for A-profile architecture [1].
- C1.3.1 CPACR_EL1.
- C1.3.2 *CPTR_EL2*.
- C1.3.3 CPTR_EL3.
- C1.3.4 ESR_EL1, ESR_EL2, and ESR_EL3.
- Chapter E1 Instructions affected by SME.

C1.2.2 Synchronous Data Abort

R_{JXPNL} If SME is implemented, then when the PE takes a Data Abort exception and sets ES ELX.ISV to 0 and is caused by either an SME load/store instruction, or an SVE contiguous vec. load/store in. uction when the PE is in Streaming SVE mode, the PE:

- Sets ESR_ELX.FnP to 1 if the value written to the corresr _iding FAR_ELX ight of be the same as the faulting virtual address that generated the Data Abort.
- Otherwise, sets $ESR_ELx.FnP$ to 0.

R_{XMWQT} If the PE sets ESR_ELX.ISV to 0 and ESR_ELX.Fr ... on tak, a De' Abort exception, then the PE sets the corresponding FAR_ELX to any address within the *naturali* alignea, granule which contains the faulting virtual address that generated the Data Abort.

- D_{BRGHW} The naturally-aligned fault granule is \bigcirc of the lowing:
 - A 16-byte tag granule when *K_ELX*. SC is 0. 10001, indicating a Synchronous Tag Check fault.
 - An IMPLEMENTATION DF VED granule in ESK_ELX.DFSC is Ob11010x, indicating an IMPLEMENTATION DEFINED fault.
 - Otherwise, the smallest implen ted translation granule.

See also:

- FAR_EL1
- FAR_EL2.
- FAR ---- 3
- *C* .5.4 *E*, '_*EL1*, '*R* .2, and ESR_EL3.

C1.2.3 Validity f ME and SVE state

I_{VBJBR} The Effective 'ues of pSTATE.SM and pSTATE.ZA configure whether SME architectural state is valid and accessible.

- IWFHKZ
 Fields in CPACR_EL1, CPTR_EL2, and CPTR_EL3 configure whether SME-related instructions can be executed or are trapped.
- IKKEXN
 The fields SMCR_EL1.EZTO, SMCR_EL2.EZTO, and SMCR_EL3.EZTO configure whether SME2 instructions that access the ZTO register can be executed or are trapped.

R_{CSPYJ} When SME2 is implemented, accesses to the *ZT0* register are enabled when all of the following are true:

- The access is from EL0 and SMCR_EL1.EZTO is 1, if EL2 is implemented and enabled in the current Security state and HCR_EL2.{E2H, TGE} is not {1,1}.
- The access is from EL1 and SMCR_EL1.EZTO is 1.
- The access is from EL0, EL1, or EL2 and SMCR_EL2.EZTO is 1, if EL2 is implemented and enabled in the current Security state.
- The access is from EL0, EL1, EL2, or EL3 and SMCR_EL3.EZT0 is 1.

Chapter C1. System management C1.2. Processor behavior

R _{xccxw}	The controls for trapping SME-related instructions and the controls for the validity of SME architectural state are independent.
I _{JGRTR}	Because the trap and architectural state validity are controlled independently, the following scenarios are all permissible:
	 Instructions trap, state invalid. For example, an OS traps the first usage of SME-related instructions by a process. Instructions trap, state valid. For example, a process was running with valid SME architectural state and an OS configures traps to detect when the next usage of SME architectural state occurs. Enabling the trap does not affect or corrupt the SME architectural state. Instructions permitted, state invalid. For example, a process is permitted to execute SME-related instructions but is currently not running in <i>Streaming SVE mode</i>. SME data-processing instructions which <i>cless</i> and <i>P</i> vector or predicate registers are <i>illegal</i> and are trapped, but SVE instructions operate on <i>cl</i> Non-streating SVE register state. The process can execute an SMSTART instruction to enter <i>Stream. SVE mode</i>. For example, a process is running in <i>Streaming SVE m le</i>, but <i>s</i> not enable access to the ZA storage. SME instructions permitted, state valid. Instructions permitted, state valid. For example, a process is running in <i>Streaming SVE m le</i>, and <i>P</i> venabled access to ZA storage.
R _{swqgh}	An exception return from AArch64 to AArch? Exect on state for not change the values of PSTATE.SM and PSTATE.ZA.
R_{MZLVB}	An exception taken from AArch32 to ^Arch F cution state does not change the values of PSTATE.SM and PSTATE.ZA.
R _{gxknk}	When a PE is executing in the A ch32 Exection state the Effective value of PSTATE.SM is 0.
I _{MWQNV}	When PSTATE.SM is 1, a chrogen Execution stal from AArch64 to AArch32, or from AArch32 to AArch64, causes all implemented bits of the SV registers (including SIMD&FP registers) and the FPSR to be reset to a fixed value, which software sust mitigate.
I _{WYKRM}	The Effective value of PSTATE.ZA does not change in AArch32 Execution state. Therefore, a transition between AArch64 and Ar. $h32$ F ecution state when PSTATE.ZA is 1 has no effect on the contents of ZA storage, or the ZTO register when S. I is implemented.
I _{VGWQW}	An important on mig we the activity of the PSTATE.SM and PSTATE.ZA bits to influence the choice of power-saving sets for both functional units and retention of architected state.
C1.2.4	Streaming xecution priority for shared implementations
I _{yyrzq}	Execution of certain instructions by a PE in <i>Streaming SVE mode</i> might experience a performance dependency on other PEs in the system that are also executing instructions in <i>Streaming SVE mode</i> . For example, this might occur when a <i>Streaming Mode Compute Unit</i> (SMCU) is shared between PEs.
I _{wpqvv}	The architecture provides a mechanism to control the streaming execution priority of a PE, in SMPRI_EL1. The streaming execution priority of a PE is relative to the streaming execution priority of other PEs, when a performance dependency exists between PEs executing in <i>Streaming SVE mode</i> .
D _{dgrts}	All PEs that share a given SMCU form a Priority domain.
$D_{\mathtt{YQFWM}}$	Different Priority domains represent unrelated SMCUs.

- $R_{\tt WPVQK} \qquad \mbox{All PEs in a Priority domain have the same value of {\tt SMIDR_EL1.Affinity}.}$
- R_{CVLSF} PEs in differing *Priority domains* have different values of SMIDR_EL1.Affinity.
- R_{GGDRC}

Chapter C1. System management C1.2. Processor behavior

The streaming execution priority in SMPRI_EL1 affects execution of a PE relative to all other PEs in the same *Priority domain*.

- I_{WVGGW} System software can use the streaming execution priority mechanism to manage scenarios where multiple concurrent software threads contend on shared SMCUs.
- R_{RQXFC} The streaming execution priority mechanism affects the execution of instructions by a shared SMCU when the PE is in *Streaming SVE mode* and does not directly control the execution of other types of instruction.
- I_{HQXBH} The streaming execution priority mechanism is optional.
- IKTYTDAn implementation that does not share SMCUs or has no performance dependency between PEs might not need to
limit or prioritize execution of one PE relative to another.
- IYBQNWThe architecture considers *Priority domains* to be non-overlapping sets, meaning that in a shared-SMCU system a
PE is associated with at most one SMCU.

C1.2.4.1 Streaming execution context management

IPRNMJArm expects that the SVE- and SME-related instructions used b save, resto. and cle routines for the Streaming
SVE mode SVE register state, the ZA array state, and the Zⁿ register when S. Tⁿ implemented, are limited to
using the following SME and SVE instructions:

- SME LDR (vector) and STR (vector) instruction
- SME2 LDR (ZT0) and STR (ZT0) instructio^{*} .
- SVE LDR (vector) and STR (vector) instr aons.
- SVE LDR (predicate) and STR (predicate "struc"
- SME ZERO (tile) instruction.
- SME2 ZERO (ZT0) instruction
- SVE DUP (immediate) instruction with . to imm. "ate.
- SVE PFALSE instruction

For implementations with a mared S. [°]U, PEs are expected to execute these instructions in a way that experiences a reduced effect of council for the [°]UCU from other PEs, compared to other SME and SVE instructions executed in *Streami*, *SVE mode*.

C1.2.4.2 Stre mir J execution priority control

- I_{RMDCP} The strong ecution. *vic* by is controlled by a 4-bit priority value. When the streaming execution priority mechanism is *r* c supported, the priority value is ignored.
- I_{FJQRG} A high r priority value corresponds to a higher streaming execution priority. Priority value 15 is the highest priority.
- I_{QHKWP} The behavior any given priority value relative to that of another PE is IMPLEMENTATION DEFINED.

C1.2.4.3 Streaming execution priority virtualization

- I SQBCZ The Effective streaming execution priority is either the value configured in SMPRI_EL1, or the value of SMPRI_EL1 mapped into a new value by indexing the fields in SMPRIMAP_EL2. This choice is affected by the current Exception level, and the HCRX_EL2.SMPME configuration.
- ILSZZG
 A hypervisor can use SMPRIMAP_EL2 to map the virtual streaming execution priority values written into SMPRI_EL1

 by a guest OS into different physical priority values.

See also:

- C1.3.6 *HCRX_EL2*.
- C1.4.5 *SMIDR_EL1*.
- C1.4.6 *SMPRI_EL1*.

• C1.4.7 *SMPRIMAP_EL2*.

C1.2.5 Security considerations

- IDXRGG
 All SME load and store instructions adhere to the memory access permissions model in The AArch64 Virtual Memory System Architecture chapter of Arm[®] Architecture Reference Manual for A-profile architecture [1].
- I_{MGLWR} SME architectural state can be access-controlled, meaning that higher levels of privilege can trap access to the state from the same or lower levels of privilege.

For example, execution of SME instructions including entry to or exit from *Streaming SVE mode* in EL0 might be trapped to EL2.

- I_{CYPJJ} System software has controls available to save and restore state between up toted pieces of software, and must ensure that steps are taken to preserve isolation and privacy.
- I_{TDPHC} Operations performed in *Streaming SVE mode* respect the requirements of P. ATE.DIT. DIT requires data-independent timing when enabled.

C1.3 Changes to existing System registers

C1.3.1 CPACR_EL1

- R_{QBTKS} If SME is implemented, the field CPACR_EL1.SMEN is defined at bits [25:24]. For more information, see CPACR_EL1.
- I_{ZNSLS} The set of SME-related instructions trapped by this control is defined by rule D_{DMBHW} in C1.1.2 Traps and *exceptions*.

See also:

- C1.1.2 Traps and exceptions.
- C1.2.1 Exception priorities.
- C1.2.3 Validity of SME and SVE state.
- C1.3.4 ESR_EL1, ESR_EL2, and ESR_EL3.

C1.3.2 CPTR_EL2

- R_{2X2ZC} If SME is implemented, FEAT_VHE is implemented, and HC V. E2H is 1 e field CPTR_EL2.SMEN is defined at bits [25:24]. For more information, see CPTR_EL2
- R_{QLKFH} When SME is implemented, FEAT_VHE is implemented and HCR_E .E2H is 0, the field CPTR_EL2.TSM is defined at bit [12]. For more information, see CPTK_ 1.2.
- I_{DYLZC} The set of SME-related instructions the by \cdot 's control is defined by rule D_{DMBHW} in C1.1.2 Traps and *exceptions*.

See also:

- C1.1.2 Traps and exc ptions.
- C1.2.1 Exception riorities.
- C1.2.3 Validit f SME and SVE st.
- C1.3.4 ESF LLI, ESF EL2, and ESR_EL3.

C1.3.3 CPTR_EI

R_{NPVSR} If S. S is improved the field CPTR_EL3.ESM is defined at bit [12]. For more information, see CPTR_EL3.

 I_{LGJZY} The set SME-related instructions trapped by this control is defined by rule D_{DMBHW} in C1.1.2 Traps and exceptions.

See also:

- C1.1.2 Traps and exceptions.
- C1.2.1 *Exception priorities*.
- C1.2.3 Validity of SME and SVE state.
- C1.3.4 ESR_EL1, ESR_EL2, and ESR_EL3.

C1.3.4 ESR_EL1, ESR_EL2, and ESR_EL3

D_{DZSWB} If SME is implemented, an Exception Class value 0x1D is added to ESR_EL1, ESR_EL2, and ESR_EL3, for exceptions taken from AArch64.

Chapter C1. System management C1.3. Changes to existing System registers

EC	ISS[2:0]	Meaning
0b011101	06000	Access to SME functionality trapped as a result of CPACR_EL1.SMEN, CPTR_EL2.SMEN, CPTR_EL2.TSM, or CPTR_EL3.ESM, that is not reported using EC 0b000000.
0b011101	0b001	Illegal Advanced SIMD, SVE, or SVE2 instruction trapped because ${\tt pstate.sm}$ is 1
0b011101	0b010	Illegal SME instruction trapped because PSTATE . SM is 0
0b011101	0b011	Illegal SME instruction trapped because $PSTATE.ZA$ is 0

Other values of ISS[2:0] are reserved.

 $\mathsf{D}_{\text{MSSSD}}$

If SME2 is implemented, for Exception Class 0x1D, the following ISS fue is a d to ESR_EL1, ESR_EL2, and ESR_EL3:

EC	ISS[2:0]	Meaning
0b011101	0b100	Access to the SME2 ZTO IC ster treped as a result of SMCR_EL1.EZTO, SMCR_EL2.EZTO, or SMCR_EL3.

R_{DYSFV} If SME is implemented, then the following 1d is defined in the ESR_EL1, ESR_EL2, and ESR_EL3 ISS encoding for an exception from a Data Abort, when the 1 V?' c is 0b100101 (0x24) or 0b100101 (0x25):

Field	Name	Meaning
[15]	FnP	Who 1SV - 9: F R not Precis
		 b0 The FAR ho. the faulting virtual address that generated the Data Abort. b1 The FAR holds any virtual address within the <i>naturally-aligned fault granule</i> (see D_{BRGHW}) that contains the faulting virtual address that generated a Data Abort excer on due to an SVE contiguous vector load/store instruction when the PE is in Stroming SVE mode, or by an SME load/store instruction. On Varm reset, this field resets to an architecturally UNKNOWN value.

C1.3.5 HCR_EL2

 $R_{\text{BWHVR}} \qquad \quad \text{If SME is implemented, the } \text{hcr_el2.tid3 field causes accesses to } \text{id_aa64smfr0_el1 to be trapped.}$

R_{ZRBBT} If SME is implemented, the HCR_EL2.TID1 field causes accesses to SMIDR_EL1 to be trapped.

See also:

- HCR_EL2.
- ID_AA64SMFR0_EL1.
- SMIDR_EL1.

C1.3.6 HCRX_EL2

RYDHJV If SME is implemented, the field HCRX_EL2. SMPME is defined at bit [5]. For more information, see HCRX_EL2.

See also:

- C1.2.4 Streaming execution priority for shared implementations.
- C1.4.6 *SMPRI_EL1*.
- C1.4.7 *SMPRIMAP_EL2*.

C1.3.7 HFGRTR_EL2

R_{SXDSB} If SME is implemented, the fields HFGRTR_EL2.nTPIDR2_EL0 and HFGRTR_EL2.nSMPRI_EL1 are defined at bits [55] and [54]. For more information, see HFGRTR_EL2.

See also:

- B1.2.4 *TPIDR2_EL0*.
- C1.4.6 *SMPRI_EL1*.

C1.3.8 HFGWTR_EL2

R_{XKLBN} If SME is implemented, the fields HFGWTR_EL2.nTPIDR2_F and HFC_R_EL2.n_PRI_EL1 are defined at bits [55] and [54]. For more information, see HFGWTR_EL2.

See also:

- B1.2.4 *TPIDR2_EL0*.
- C1.4.6 *SMPRI_EL1*.

C1.3.9 ID_AA64PFR1_EL1

- R_{KHPZL} If SME is implemented, the cld _ AA64PFR1_EL SME is defined at bits [27:24].
- RLYHCJ If SME2 is implemente the value Ob. 0 is added to ID_AA64PFR1_EL1. SME at bit position [27:24].
- I_{DJQVZ} A nonzero value in J_AA64PFR1_EL1.SME coes not imply that ID_AA64PFR0_EL1.SVE must also contain a nonzero value.

See also:

• 6 .3.9 IL AA64P. 1 L1

C1.3.10 ID_AA. ** . R0_EL1

- R_{SYRGK} If SME is imp_nented, the ID_AA64ZFR0_EL1 register identifies the implemented features of the SVE instruction set when any of ID_AA64PFR0_EL1.SVE and ID_AA64PFR1_EL1.SME are nonzero.
- R_{JLSQK} If SME is implemented, then ID_AA64ZFR0_EL1.SVEver has a nonzero value, indicating that *legal* SVE and SVE2 instructions can be executed when the PE is in *Streaming SVE mode*.
- R_{RFZRM} If SME is implemented and ID_AA64PFR0_EL1.SVE is nonzero, then FEAT_SVE2 is implemented and SVE and SVE2 instructions can be executed when the PE is not in *Streaming SVE mode*.
- R_{XFFLH} If SME is implemented and ID_AA64PFR0_EL1.SVE is zero, then FEAT_SVE is not implemented and the ID_AA64ZFR0_EL1 fields named F64MM, F32MM, SM4, SHA3, BitPerm, and AES hold the value zero.

See also:

- ID_AA64PFR0_EL1, defined in Arm[®] Architecture Registers, for A-profile architecture [2].
- C1.3.9 *ID_AA64PFR1_EL1*.
- C1.3.10 *ID_AA64ZFR0_EL1*.

C1.3.11 SCR_EL3

- R_{TCPTK} If SME is implemented, the field SCR_EL3.EnTP2 is defined at bit [41]. For more information, see SCR_EL3. See also:
 - B1.2.4 *TPIDR2_EL0*.

C1.3.12 SCTLR_EL1

- R_{NMVMQ} If SME is implemented, the field SCTLR_EL1.EnTP2 is defined at bit [60]. For more information, see SCTLR_EL1.
- R_{MXHMD} When EL2 is implemented and enabled in the current Security state and HCR_EL2.{E2H, TGE} is {1, 1}, the SCTLR_EL1.EnTP2 control has no effect on execution at EL0 and the SCT______ EnTP2 control is used for this purpose.

See also:

• B1.2.4 *TPIDR2_EL0*.

C1.3.13 SCTLR_EL2

R_{KGMHQ} If SME is implemented and HCR_EL2. {E2H, TGF 18 {1, }}, the 'd c lR_EL2.EnTP2 is defined at bit [60]. For more information, see SCTLR_EL2.

See also:

• B1.2.4 *TPIDR2_EL0*.

C1.3.14 ZCR_EL1, ZCR_EL2, 7 J 2 R_EL3

I_{CRKQJ} If FEAT_SVE is implemented, then the _____ELx registers have their described effect on the Effective SVE vector length only when ____ PE is p___ in *Streaming SVE mode*.

C1.4 SME-specific System registers

C1.4.1 ID_AA64SMFR0_EL1

I_{DVWNO} The AArch64 SME Feature ID Register describes the set of implemented SME data-processing instructions.

D_{GRPHH} If SME is implemented, the register ID_AA64SMFR0_EL1 is added. For more information, see ID_AA64SMFR0_EL1.

C1.4.2 SMCR_EL1

- IRKHZL
 The Streaming SVE Mode Control Register for EL1 configures the Effection paming SVE vector length and accessibility of the SME2 ZT0 register, when executing at EL1 or EL0
- R_{HRTZQ} If SME is implemented, the register SMCR_EL1 is added. For more intervation, see SN_CR_EL1.
- R_{NWXVG} When EL2 is implemented and enabled in the current Security state and CR_EL2 (E2H, TGE) is {1, 1}, the SMCR_EL1 register has no effect on execution at EL0 and EV and the CMCR_EL restored for this purpose.

C1.4.3 SMCR_EL2

- I_{WTNZY} The Streaming Mode Control Register for L2 conf uses the L active Streaming SVE vector length and accessibility of the SME2 ZTO register when $ecut^2$ and d at EL1 or EL0 in the same Security state as EL2.
- R_{JPZPH} If SME is implemented, the regist SMCR_E. is ado. For more information, see SMCR_EL2.

C1.4.4 SMCR_EL3

- I_VVCBLThe Streaming McControl Register 1EL3 configures the Effective Streaming SVE vector length and
accessibility of t'SME2 7J register, when executing at EL3, EL2, EL1, or EL0.
- R_{DBGWC} If SME is implement in registress smcr_EL3 is added. For more information, see SMCR_EL3.

C1.4.5 SMID _EL1

- IMZESJ
 The Streeting Mode Identification Register provides additional information about the Streaming SVE mode implementation.
- D_{NBDMK} If SME is implemented, the register SMIDR_EL1 is added. For more information, see SMIDR_EL1.

C1.4.6 SMPRI_EL1

- IPJDXFThe Streaming Mode Priority register configures the streaming execution priority for instructions executed in
Streaming SVE mode on a shared SMCU at any Exception level.
- R_{JKMFH} If SME is implemented, the register SMPRI_EL1 is added. For more information, see SMPRI_EL1.
- R_{DWGZP} In an implementation that shares execution resources between PEs, higher streaming execution priority values are allocated more processing resource than other PEs configured with lower streaming execution priority values in the same *Priority domain*.
- R_{DFQLX} The precise meaning and behavior of each streaming execution priority value is IMPLEMENTATION DEFINED.

Chapter C1. System management C1.4. SME-specific System registers

- I_{BLMYK} If system software does not support differentiation of streaming execution priority of threads, it is safe to use a value of 0 for all threads.
- R_{SBCRG} All SMCUs in the system have a consistent interpretation of the streaming execution priority values.

See also:

• C1.2.4 Streaming execution priority for shared implementations.

C1.4.7 SMPRIMAP_EL2

IThe Streaming Mode Priority Mapping register maps the current virtual streaming execution priority value to a
physical streaming execution priority value for instructions executed in *Streaming SVE mode* on a shared SMCU at
EL1 or EL0 in the same Security state as EL2.

D_{CHVZD} If SME is implemented, the register SMPRIMAP_EL2 is added. For more formation, e SMPRIMAP_EL2.

C1.4.8 SVCR

- I_{JGCTD} The Streaming Vector Control Register provides direct. PSTATE.SM and PSTATE.ZA mode bits from any Exception level.
- D_{JXVQJ} If SME is implemented, the register SVCR is add^r. For pre in pati, see SVCR.

See also:

• B1.2.3 Changing PSTATE.SM ar PSTA

Chapter C2 Interaction with other A-profile archity ctur at features

 I_{MDMBB} This section describes the interaction f SME with other aspects and features of the A-profile architecture.

It covers:

• Watchpoin*

- Self-hosted Ug
- Exter ¹ ⁴ebug.
- M nory gging h enc n (MTE).
- .eliabilit Availabih. , and Serviceability (RAS).
- femor 3 and Monitoring (MPAM).
- 1. .ctional Memory Extension (TME).
- Men. v consistency model.

See also:

- Arm[®] Architecture Reference Manual for A-profile architecture [1].
- Arm[®] Architecture Reference Manual Supplement, Memory System Resource Partitioning and Monitoring (MPAM), for A-profile architecture [4].
- Arm[®] Architecture Reference Manual Supplement, The Transactional Memory Extension (TME), for A-profile architecture [5].

C2.1 Watchpoints

- R_{PDGZL} For a memory access or set of contiguous memory accesses generated by an SVE contiguous vector load/store instruction when the PE is in *Streaming SVE mode*, or by an SME load/store instruction, if a watchpoint matches a range where the lowest accessed address is rounded down to the nearest multiple of 16 bytes and the highest accessed address is rounded up to the nearest multiple of 16 bytes minus 1, but the watchpoint does not the match the range of the original access or set of contiguous accesses, then it is CONSTRAINED UNPREDICTABLE whether or not a Watchpoint debug event is triggered.
- R_{XKRPV} If a watchpoint matches only the rounded access address ranges of Inactive elements in a predicated vector load/store instruction, then it does not trigger a Watchpoint debug event.
- If a Watchpoint debug event is triggered by a match on a rounded access address range that would not have been triggered by the original access address range, then this may report a fals poster match. Debug software must attempt to detect and step over false-positive matches. The architectur poes not per it missed, or false-negative matches.

C2.1.1 Reporting watchpoints

R_{KDRCX} If SME is implemented, then the following fields are added R_EL1 ar LSR_EL2 in the ISS encoding for an exception from a Watchpoint exception, when the provide is 0. Y or 0.

Field	Name	Meaning
[24]	ISV	RESO
[23:18]	WPT	Wate' oint number, 15 inclusive. $A^{1'}$ on values are recoved.
[17]	WPTV	 Watchpoin, Tymber Valid. Ob0 The WPi rold is invalid, and holds an UNKNOWN value. Ob1 The WPT field is valid, and holds the number of a watchpoint that triggered a Watchpoint debug event.
[16]	w. '	 Wate point might be false-positive. The watchpoint matched the original access or set of contiguous accesses. 0b1 The watchpoint matched an access or set of contiguous accesses where the lowest accessed address was rounded down to the nearest multiple of 16 bytes and the highest accessed address was rounded up to the nearest multiple of 16 bytes minus 1, but the watchpoint might not have matched the original access or set of contiguous accesses.
[15]	FnP	 FAR not Precise. This field only has meaning if the FAR is valid; that is, when the FnV field is 0. If the FnV field is 1, the FnP field is 0. Ob0 If the FnV field is 0, the FAR holds the virtual address of an access or set of contiguous accesses that triggered a Watchpoint debug event. Ob1 The FAR holds any address within the smallest implemented translation granule that contains the virtual address of an access or set of contiguous accesses that triggered a Watchpoint debug event.

Field	Name	Meaning
[10]	FnV	FAR not Valid.Ob0 The FAR is valid, and its value is as described by the FnP field.Ob1 The FAR is invalid, and holds an UNKNOWN value.

R_{MFRPZ} If SME is implemented, then the following field is added to the EDDEVID1 register:

Field	Name	Meaning
[7:4]	HSR	Indicates support for the External Debug Halt Surger (EDHSR). Defined values are: Ob0000 EDHSR not implemented, and the S follows behavors consistent with all of the EDHSR fields having a zero value. Ob0001 EDHSR implemented. All other values are reserved. If FEAT_SME is implemented, S performed values are Ob0000 and Ob0001. If FEAT_SME is not implemented. Solution only performed value is Ob0000.

 R_{QBKWY} If SME is implemented, then the read-only. ternal D \rightarrow Halt Status Register (EDHSR) may be implemented at offset 0x038. The field EDDEVID1.HSR1 indicate v¹ and the EDRSR is implemented.

R_{SDSFM} If the EDHSR is implemented, t¹ (it is only v. ¹ when the PE is in Debug state and EDSCR.STATUS indicates a Watchpoint debug event (05 101 otherwise it.) an UNKNOWN value.

The EDHSR fields are de^c ed as follow.

Field	Nar	V aning
[23:18]	<u>T</u>	Watch ² .nt number, 0-15 inclusive. All .ner values are reserved.
[17]	WT .V	 Watchpoint number Valid. Ob0 The WPT field is invalid, and holds an UNKNOWN value. Ob1 The WPT field is valid, and holds the number of a watchpoint that triggered a Watchpoint debug event. On a Cold reset, this field resets to an architecturally UNKNOWN value.
[16]	WPF	 Watchpoint might be false-positive. 0b0 The watchpoint matched the original access or set of contiguous accesses. 0b1 The watchpoint matched an access or set of contiguous accesses where the lowest accessed address was rounded down to the nearest multiple of 16 bytes and the highest accessed address was rounded up to the nearest multiple of 16 bytes minus 1, but the watchpoint might not have matched the original access or set of contiguous accesses.

On a Cold reset, this field resets to an architecturally UNKNOWN value.

 R_{LXGXN} If the EDHSR is implemented, it is in the Component main.

	Field	Name	Meaning
	[15]	FnP	 FAR not Precise. This field only has meaning if the EDWAR is valid; that is, when the FnV field is 0. If the FnV field is 1, the FnP field is 0. 0b0 If the FnV field is 0, the EDWAR holds the virtual address of an access or set of contiguous accesses that triggered a Watchpoint debug event. 0b1 The EDWAR holds any address within the smallest implemented translation granule that contains the virtual address of an access or set of contiguous accesses that triggered a Watchpoint debug event. 0n a Cold reset, this field resets to an architecturally UNKNOWN value.
	[10]	FnV	 FAR not Valid. Ob0 The EDWAR is valid, and its value is as destricted the FnP field. Ob1 The EDWAR is invalid, and holds an UN[*]. OWN valu. On a Cold reset, this field resets to an arch. cturally UNKN WN value.
R_{XWDJT}	If the r	EDHSR is not imp	lemented, then the PE must follow by aviors consistent and of the EDHSR fields having a
R _{CXZCY}	If a Ward of the second	atchpoint debug <i>node</i> , or by an S d from an addres	event is triggered by an SVF tiguou. ad/store struction when the PE is in <i>Streaming</i> ME load/store instruction then virtual 'dre recorded in FAR_ELX or EDWAR must be so that is both:
	• 1	n the inclusive r - The lowest a address as p - The highest the highest v Within a naturall	ange between: ddress accessed by 'he vector instruction that triggered the watchpoint, or the lowest rounded ermitted by $f_{,,DGZL}$. watchpoine ed address a lowest buy the vector instruction that triggered the watchpoint, or vatchmente inderess in the lidress range permitted by R_{PDGZL} . y-aligned bloc lowest for the lowest rounded by the vector instruction that triggered the watchpoint, or y-aligned bloc lowest rounded by R_{PDGZL} .
R _{sqdkj}	If an in not ace	nstruction gen/ cessed by the ins	the set watchpoint . the where the the watchpointed data address or data address range is struction, the PE:
	• § • §	Sets ESR_ELX P Sets TR.WPP r .n.	 1, on taking a Watchpoint exception generated by the watchpoint match. 1, on en ling Debug state on a Watchpoint debug event generated by the watchpoint
	Otl.	vise, ES' or W	PF or EDHSR.WPF (as applicable) is set to an IMPLEMENTATION DEFINED choice of 0 or 1.
	For ex SVE n addres applic	a. 2, when R _P node r an SM s range at the able) is set to 0 f	DGZL applies, an SVE contiguous vector load/store instruction when the PE is in <i>Streaming</i> E load/store instruction might generate a watchpoint match for a data address or data instruction does not access. Arm strongly recommends that ESR_ELX.WPF or EDHSR.WPF (as For all other cases.
R _{KSSHC}	If a w Strean	atchpoint match	es an access that is due to an SVE contiguous load/store instruction when the PE is in or is due to an SME load/store instruction, then the PE:
	• S	Sets ESR_ELx.Fn generated by the	\vee to an IMPLEMENTATION DEFINED value, 0 or 1, on taking a Watchpoint exception watchpoint match.
	• S	Sets EDHSR.FnV t lebug event gene	o an IMPLEMENTATION DEFINED value, 0 or 1, on entering Debug state on a Watchpoint erated by the watchpoint match.
	• (Otherwise, ESR_E	ELX.FNV or EDHSR.FNV (as applicable) is set to 0.
R _{rlwsf}	When	the PE sets ESR_	$_{\text{ELx.FnV}}$ to 0 on taking a Watchpoint exception generated by the watchpoint match:
	• I ł	f the lowest wat	chpointed address higher than or the same as the address recorded in FAR_ELx might not ed by the instruction, other than as permitted by R_{PDGZL} , then the PE sets ESR_ELx.FnP to 1.

	• Otherwise, the PE sets ESR_ELX.FnP to 0.	
R _{cjwyx}	When the PE sets EDHSR.FnV to 0 on entering Debug state on a Watchpoint debug event generated by a watchpoint match:	
	• If the lowest watchpointed address higher than or the same as the address recorded in EDWAR might not have been accessed by the instruction, other than as permitted by R_{PDGZL} , then the PE sets EDHSR.FnP to 1.	
	• Otherwise, the PE sets EDHSR.FnP to 0.	
R _{dtwth}	When a Watchpoint exception is triggered by a watchpoint match:	
	 If the PE sets any of ESR_ELX.FnV, ESR_ELX.FnP, or ESR_ELX.WPF to 1, then the PE sets ESR_ELX.WPTV to 1. If the PE sets all of ESR_ELX.FnV, ESR_ELX.FnP, and ESR_ELX.WPF to 0, then the PE sets ESR_ELX.WPTV to an IMPLEMENTATION DEFINED value, 0 or 1. 	
R _{BNXVL}	When an entry to Debug state is triggered by a watchpoint match:	
	 If the PE sets any of EDHSR.FnV, EDHSR.FnP, or EDHSR.WPF to 1 (en the PE set. DHSR.WPTV to 1. If the PE sets all of EDHSR.FnV, EDHSR.FnP, and EDHSR.WPF to 0, n the PE set EDHSR.WPTV to an IMPLE-MENTATION DEFINED value, 0 or 1. 	
R_{PVYNL}	On a watchpoint match generated by watchpoint <i><n></n></i> :	
	 If the PE sets ESR_ELX.WPTV to 1 on taking a Watchpo. evention generated by the watchpoint match, then ESR_ELX.WPT is set to <n>.</n> If the PE sets EDHSR.WPTV to 1 on entering Debut state converted by the watchpoint match, then EDHSR.WPT is solved on solve the watchpoint match, then EDHSR.WPT is solved on solve the watchpoint match, then EDHSR.WPT is solved on solve the watchpoint match, then EDHSR.WPT is solved on solve the watchpoint match, then EDHSR.WPT is solved on solve the watchpoint match, then EDHSR.WPT is solved on solve the watchpoint match, then EDHSR.WPT is solved on solve the watchpoint match, then EDHSR.WPT is solved on solve the watchpoint match, then EDHSR.WPT is solved on solve the watchpoint match, then EDHSR.WPT is solved on solve the watchpoint match, then EDHSR.WPT is solved on solve the watchpoint match, then EDHSR.WPT is solved on solve the watchpoint match, then EDHSR.WPT is solved on solve the watchpoint match, then EDHSR.WPT is solved on solve the watchpoint match, then EDHSR.WPT is solved on solve the watchpoint match watchpoint match watchpoint match, then EDHSR.WPT is solved on solve the watchpoint match watchpoint watchpoint match watchpoint match watchpoint watchpoint watchpoint watchpoint watchpoint watchpoint watchpoint watchpoin	
R _{KHSFH}	When an instruction generates multiple tchpo, matches and the PE sets ESR_ELX.WPTV or EDHSR.WPTV to 1, then it is UNPREDICTABLE which _atched tchpo. is reported in ESR_ELX.WPT or EDHSR.WPT (as applicable).	
D _{LXZPC}	The naturally-aligned block c . vory is all of following:	
	 A power-of-two size. No larger than the DC ZVA block similar Structure in the SR_ELX.FnP or EDHSR.FnP (as appropriate) is 0. No larger than the smallest implemented translation granule if ESR_ELX.FnP or EDHSR.FnP (as appropriate) is 1. Contains a way have inted address accessed by the memory access or set of contiguous memory accesses that trigger the way point, converted address in the address range permitted by R_{PDGZL}. 	
	The s' of the ock is IN. MENTATION DEFINED.	
	There no means of discovering the size.	
D _{ltgky}	A watchp. •ted address is an address that a watchpoint is watching.	

Chapter C2. Interaction with other A-profile architectural features C2.2. Self-hosted debug

C2.2 Self-hosted debug

 ${\tt I}_{\tt CDBZX} \qquad SME \mbox{ has no additional effect on self-hosted debug.}$



Chapter C2. Interaction with other A-profile architectural features C2.3. External debug

C2.3 External debug

R_{XQQRS} The following SME-related instructions are unchanged in Debug state:

- MOVA (tile to vector, single).
- MOVA (vector to tile, single).
- MRS SVCR.
- MSR SVCR.
- RDSVL.

R_{CSRRS}

- If SME2 is implemented, the following instructions are defined only when the PE is in Debug state:
 - MOVT (ZT0 to scalar).
 MOVT (scalar to ZT0).
- R_{TNZLR} All other SME-related instructions are CONSTRAINED UNPREDICTABLE in Debustate, with the same set of CONSTRAINED UNPREDICTABLE options as other instructions in D_s is state, as defined in *Arm[®]* Architecture *Reference Manual for A-profile architecture* [1].

Chapter C2. Interaction with other A-profile architectural features C2.4. Memory Tagging Extension (MTE)

C2.4 Memory Tagging Extension (MTE)

The following rules apply when the optional FEAT_MTE feature is implemented.

R_{BGGMD} When the Memory Tagging Extension is implemented, it is IMPLEMENTATION DEFINED whether memory accesses due to SME, SVE, and SIMD&FP load and store instructions executed when the PE is in *Streaming SVE mode* will perform a Tag Check.

R_{GLYMK} When the Memory Tagging Extension is implemented, it is IMPLEMENTATION DEFINED whether memory accesses due to the following instructions will perform a Tag Check:

- SME LDR (vector) and STR (vector) instructions.
- SME2 LDR (ZT0) and STR (ZT0) instructions.
- IRBPIN
 An implementation of FEAT_MTE is only expected to perform Tag Checking on the PE is in Streaming SVE mode if it can do so with a similar relative performance impact to Tag Checking men.
 Implementation of SVE and SIMD&FP load and store instructions executed when the PE is not in *treaming SVE ode*.

Chapter C2. Interaction with other A-profile architectural features C2.5. Reliability, Availability, and Serviceability (RAS)

C2.5 Reliability, Availability, and Serviceability (RAS)

 \mathbb{R}_{RVYHY} Rules I_{NTXKV} and R_{NQDWB} in the RAS PE Architecture chapter of Arm[®] Architecture Reference Manual for A-profile architecture [1] are extended by adding the ZA storage, and the ZT0 register when SME2 is implemented, to any list of program-visible architectural state or registers that includes the SIMD&FP or SVE registers.



Chapter C2. Interaction with other A-profile architectural features C2.6. Memory Partitioning and Monitoring (MPAM)

C2.6 Memory Partitioning and Monitoring (MPAM)

The following System registers are modified or added when the optional FEAT_MPAM feature is implemented.

C2.6.1 MPAMSM_EL1

R_{PPJJP} If SME and FEAT_MPAM are implemented, the register MPAMSM_EL1 is added to generate MPAM labels for memory requests issued at any Exception level by the following instructions:

- SME load/store instructions.
- When the PE is in *Streaming SVE mode*, SVE and SIMD&FP load/store instructions, and SVE prefetch instructions.

For more information, see MPAMSM_EL1.

C2.6.2 MPAM2_EL2

R_{LJVWP} If SME, FEAT_MPAM, and EL2 are implemented, the fite MPAM2, 2.EnMPAM2, 13 defined at bit [50]. For more information, see MPAM2_EL2.

Chapter C2. Interaction with other A-profile architectural features C2.7. Transactional Memory Extension (TME)

C2.7 Transactional Memory Extension (TME)

The following rules apply when the optional FEAT_TME feature is implemented.

- R_{KHVVR} Executing a TSTART instruction when PSTATE.SM is 1 fails the transaction with the ERR cause.
- R_{LYBMR} Executing any of the following instructions while in Transactional state will cause the transaction to fail with the ERR cause:
 - An SME LDR (vector), STR (vector), or ZERO (tile) instruction.
 - An SME2 LDR (ZT0), STR (ZT0), or ZERO (ZT0) instruction.
- I_{TNZSW} Any MSR instruction that writes to the PSTATE.SM OF PSTATE.ZA bits in Transactional state, including the SMSTART and SMSTOP aliases, are UNDEFINED according to the rules in Arm[®] Architecture Reference Manual Supplement, The Transactional Memory Extension (TME), for A-profile architecture [for architecture for and supplement] use the transaction to fail with the ERR cause, without trapping.

For more information about the rules, see the "MSR (register)" an. "MSR (imm liate)" sections in *Arm*[®] Architecture Reference Manual Supplement, The Transactional *N* nory Ex. sion (*TM* , for A-profile architecture [5].

Chapter C2. Interaction with other A-profile architectural features C2.8. Memory consistency model

C2.8 Memory consistency model

- R_{BQSCG} Any access to memory performed by an SME load/store instruction, or an SVE load/store instruction when the PE is in *Streaming SVE mode*, is subject to the same rules that govern an SVE memory access in *The AArch64 Application Level Memory Model* chapter of *Arm*[®] *Architecture Reference Manual for A-profile architecture* [1].
- R_{SMWPF} When the PE is in *Streaming SVE mode*, any access to memory performed by a SIMD&FP load/store instruction is subject to the same rules that govern a SIMD&FP memory access in *The AArch64 Application Level Memory Model* chapter of *Arm*[®] *Architecture Reference Manual for A-profile architecture* [1].
- 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 a SIMD&FP load/store instruction is relaxed such that it might behave as if:
 - The Gathering attribute is set, regardless of the configured value of the nG at the 'ute.
 - The Reordering attribute is set, regardless of the configured value of the nR attilute.
 - The Early Acknowledgement attribute is set, regardless of the co. rured value the nE attribute.

Whether or not attributes are classified as mismatched is deter and strictly. the malory attributes derived from the translation table entry.

R_{HBBTV} If a pair of memory reads access the same location, and at k, t or of the rect's is generated by a SIMD&FP load instruction then, for a given observer, the pair of rection not rectired to s sfy the *internal visibility* requirement when the PE is in *Streaming SVE mode* and FF ._SM, FA64 k, or uplemented or not enabled at the current Exception level.

Part D SME Instruction Set

Chapter D1 **SME instructions**

This chapter defines the instructions ¹ded to the A64 instruction set when SME is implemented.

This content is from t' **2022-12** version. *Arm® A64 Instruction Set Architecture, for A-profile architecture* [3], which contains the efinitive letails of the instruction set.

D1.1 SME and SME2 data-processing instructions

The following SME data-processing instructions are added by the SME or SME2 architecture.

The SME data-processing instructions are available when SME or SME2 is implemented, and are identified by the presence of the FEAT_SME symbol, or a call to one of the HaveSME pseudocode functions.

The SME2 data-processing instructions are available when SME2 is implemented, and are identified by the presence of the FEAT_SME2 symbol, or a call to the HaveSME2 pseudocode function.

D1.1.1 ADD (to vector)

Add replicated single vector to multi-vector with multi-vector result

Add elements of the second source vector to the corresponding element of the two of our first source vectors and destructively place the results in the corresponding elements of the two or four first source vectors.

This instruction is unpredicated.

It has encodings from 2 classes: Two registers and Four resters

Two registers (FEAT_SME2)



0

1

0 1

1

0 0

1

ADD { <Zdn1>.<T>-<Zdn² .<T> }, <Zdn1 <T>-<Zdn2>.<T> }, <Zm>.<T>

- 1 if !HaveSME2() then UNP INEL
- 2 constant integer esize = 8 << t t(size);</pre>
- 3 integer dn = UInt(7 .: '0');
- 4 integer m = UInt(':Zm);
- 5 constant **intege** nreg =

Four registers

(FEAT S ...

1

ADD { <_ 11>.<T>-<Zdn4>.<T> }, { <Zdn1>.<T>-<Zdn4>.<T> }, <Zm>.<T>

7m

```
1 if !HaveSME2() then UNDEFINED;
```

```
2 constant integer esize = 8 << UInt(size);</pre>
```

-0

1 size 1 0

0

- 3 integer dn = UInt(Zdn:'00');
- 4 integer m = UInt('0':Zm);
- 5 constant integer nreg = 4;

Assembler Symbols

<Zdn1> For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2.

For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4.

<T> Is the size specifier, encoded in "size":

<t></t>
В
Н
S
D

- <Zdn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4 plus 3.
- <Zdn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2 plus 1.
- <Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

Operation

```
1
   CheckStreamingSVEEnabled();
2
   constant integer VL = CurrentVL;
3
   constant integer elements = VL DIV esize;
4
   array [0..3] of bits(VL) results;
5
6
   for r = 0 to nreg-1
7
       bits(VL) operand1 = Z[dn+r, VL];
       bits(VL) operand2 = Z[m, VL];
8
9
        for e = 0 to elements-1
10
           bits(esize) element1 = Elem[op/
                                            und1,
                                                      esize
           bits(esize) element2 = Elem[
11
                                           -rand2,
                                                      esize];
12
            Elem[results[r], e, esize] = e
                                             ment
                                                            2:
13
14
   for r = 0 to nreg-1
15
    Z[dn+r, VL] = results[r]
```

D1.1.2 ADD (array accumulators)

Add multi-vector to ZA array vector accumulators

The instruction operates on two or four ZA single-vector groups.

Destructively add all elements of the two or four source vectors to the corresponding elements of the two or four ZA single-vector groups. The vector numbers forming the single-vector group within each half or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

ID_AA64SMFR0_EL1.116I64 indicates whether the 64-bit integer v ant is implem ted.

It has encodings from 2 classes: Two ZA single-vectors and For ZA single-vectors

Two ZA single-vectors (FEAT_SME2)



7 constant integer nreg = 4;

Assembler Symbols

<T> Is the size specifier, encoded in "sz":

SZ	< T >
0	S
1	D

- <Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
- <offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.
- <Zm1> For the two ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 2.

For the four ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 4.

- <Zm4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zm" times 4 plus 3.
- <Zm2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zm" times 2 plus 1.

Operation

```
1
   CheckStreamingSVEAndZAEnabled();
   constant integer VL = CurrentVL;
2
3
   constant integer elements = VL DIV esize;
   integer vectors = VL DIV 8;
4
   integer vstride = vectors DIV nreg;
5
6 bits(32) vbase = X[v, 32];
7
   integer vec = (UInt(vbase) + offset) MOD
8 bits(VL) result;
9
10
   for r = 0 to nreg-1
        bits(VL) operand1 = ZAvector['ec,
11
                                               V.
        bits(VL) operand2 = Z[m+r, VL,
12
13
        for e = 0 to elements-1
            bits(esize) element = Elem[or and1, esize];
bits(esize) eleme = Elem[ope. d2, e, esize];
14
15
        Elem[result, e, siz = element1 element2;
ZAvector[vec, VL] = result,
16
17
18
        vec = vec + vst de;
```

D1.1.3 ADD (array results, multiple and single vector)

Add replicated single vector to multi-vector with ZA array vector results

The instruction operates on two or four ZA single-vector groups.

Add all corresponding elements of the second source vector and the two or four first source vectors and place the results in the corresponding elements of the two or four ZA single-vector groups. The vector numbers forming the single-vector group within each half or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

ID_AA64SMFR0_EL1.116I64 indicates whether the 64-bit integer v ant is implem ted.

It has encodings from 2 classes: Two ZA single-vectors and For ZA single-vectors

Two ZA single-vectors (FEAT_SME2)



8 constant integer nreg = 4;

Assembler Symbols

<T> Is the size specifier, encoded in "sz":

SZ	<t></t>
0	S
1	D

- <Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
- <offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.
- <Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn".
- <Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" plus 3 modulo 32.
- <Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" plus 1 modulo 32.
- <Zm> Is the name of the second source scalable vector register Z0-Z1^r encoded in e "Zm" field.

Operation

```
CheckStreamingSVEAndZAEnabled();
1
2
   constant integer VL = CurrentVL;
3
   constant integer elements = VL DIV esize;
4
   integer vectors = VL DIV 8;
   integer vstride = vectors DIV nreg;
5
   bits(32) vbase = X[v, 32];
6
   integer vec = (UInt(vbase) + offset) M
7
                                            vstri
8
   bits(VL) result;
9
10
   for r = 0 to nreg-1
11
       bits(VL) operand1 = Z[(n+r) MO. 32],
                                          VL.
       bits(VL) operand2 = Z[m, _];
12
13
       for e = 0 to elements-'
14
          bits(esize) elem
                                = Elem[oper ]1, e, esize];
       Flem[operan 2, e, esize];
15
16
                                    element1 + element2;
17
18
       vec = vec + ·
                     .ride;
```

D1.1.4 ADD (array results, multiple vectors)

Add multi-vector to multi-vector with ZA array vector results

The instruction operates on two or four ZA single-vector groups.

Add all corresponding elements of the two or four second source vectors and first source vectors and place the results in the corresponding elements of the two or four ZA single-vector groups. The vector numbers forming the single-vector group within each half or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

ID_AA64SMFR0_EL1.I16I64 indicates whether the 64-bit integer v ant is implem ted.

It has encodings from 2 classes: Two ZA single-vectors and For ZA single-vectors

Two ZA single-vectors (FEAT_SME2)

	$\begin{array}{cccccccccccccccccccccccccccccccccccc$		
	Ls		
	ADD $ZA.[, {, V, '}], { Zn1>.- }, { - }$		
1	if HaveSME2() then INDER		
2	if sz = '1' && 'HaveSN 161, ') then UNDE (NED:		
3	<pre>integer v = UInt('010':Rv);</pre>		
4	constant integer e ^s .e = 32 << UL (sz);		
5	<pre>integer n = UInt' .:'0');</pre>		
6	integer m = UIr Zm:'0'		
7	<pre>integer offset UInt (f3);</pre>		
8	constant integei re = 2;		
	Four 7 singly rectors (FF 1_SME2		
	31 <u>23 22 21 20 18 17 15 14 13 12 10 9 7 6 4 3 2 0</u>		
	1 0 0 0 0 1 1 sz 1 Zm 0 1 0 Rv 1 1 0 Zn 0 0 1 0 off3		
	Ls		
	ADD $ZA.<1>[, {, VGx4}], { .<1>-.<1> }, { .<1>-.<1> \leftrightarrow}$		
1	if !HaveSME2() then UNDEFINED;		
2 if sz == '1' && !HaveSMEI16164() then UNDEFINED;			
3 1	<pre>integer v = UIAL('UIU':KV); constant integer esize = 32 << UIAt(sz);</pre>		
+ 5	integer n = HInt (Zn:'00').		
6	<pre>integer m = UInt(Zm:'00');</pre>		

Assembler Symbols

integer offset = UInt(off3); constant integer nreg = 4;

<T> Is the size specifier, encoded in "sz":

7

8

SZ	<t></t>
0	S
1	D

- <Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
- <offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.
- <Zn1> For the two ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.

For the four ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

- <Zn4> Is the name of the fourth scalable vector register of a multi-vector sector encoded as "Zn" times 4 plus 3.
- <Zn2> Is the name of the second scalable vector register of a multi-ctor sequence encoded as "Zn" times 2 plus 1.
- <Zm1> For the two ZA single-vectors variant: is the name cone first scalar vec c register of a multi-vector sequence, encoded as "Zm" times 2.

For the four ZA single-vectors variant: is the name c^{-1} first scale is vector register of a multi-vector sequence, encoded as "Zm" $t^2 = c_0$

- <Zm4> Is the name of the fourth scalable ve or regist of a mult. Jector sequence, encoded as "Zm" times 4 plus 3.
- <Zm2> Is the name of the second scalar vector gister of a multi-vector sequence, encoded as "Zm" times 2 plus 1.

Operation

```
CheckStreamingSVEAnd"AEnabled()
1
    constant integer VV - CurrentVL;
2
   constant integer v currentvu,
constant integer lements = VL DIV ize;
integer vector v VL DIV 3;
integer vstric = vect is DIV nreg;
bits(32) vbase = [v 32];
3
4
5
6
    integer v - (UIn. vbase)
                                       offset) MOD vstride;
7
    bits(V' res. ;;
8
9
    for = 0 to real
10
11
         . 's (V open
                             = Z[n+r, VL];
12
         bi. L) operand2 = Z[m+r, VL];
         for 0 to elements-1
13
14
              bi 'esize) element1 = Elem[operand1, e, esize];
15
              bits .size) element2 = Elem[operand2, e, esize];
              Elem[result, e, esize] = element1 + element2;
16
17
         ZAvector[vec, VL] = result;
18
         vec = vec + vstride;
```

D1.1.5 ADDHA

Add horizontally vector elements to ZA tile

Add each element of the source vector to the corresponding active element of each horizontal slice of a ZA tile. The tile elements are predicated by a pair of governing predicates. An element of a horizontal slice is considered active if its corresponding element in the second governing predicate is TRUE and the element corresponding to its horizontal slice number in the first governing predicate is TRUE. Inactive elements in the destination tile remain unmodified.

ID_AA64SMFR0_EL1.I16I64 indicates whether the 64-bit integer variant is implemented.

It has encodings from 2 classes: 32-bit and 64-bit



For the 64-bit variant: is the name of the ZA tile ZA0-ZA7, encoded in the "ZAda" field.

- <Pn> Is the name of the first governing scalable predicate register P0-P7, encoded in the "Pn" field.
- <Pm> Is the name of the second governing scalable predicate register P0-P7, encoded in the "Pm" field.
- $<\!\! Zn\!\! >$ Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

```
CheckStreamingSVEAndZAEnabled();
1
2
   constant integer VL = CurrentVL;
3
   constant integer PL = VL DIV 8;
   constant integer dim = VL DIV esize;
4
5
   bits(PL) mask1 = P[a, PL];
6
   bits(PL) mask2 = P[b, PL];
   bits(VL) operand_src = Z[n, VL];
7
8
   bits(dim*dim*esize) operand_acc = ZAtile[da, esize, dim*dim*esize];
9
   bits(dim*dim*esize) result;
10
11
   for col = 0 to dim-1
12
       bits(esize) element = Elem[operand_src, col, esize];
13
        for row = 0 to dim-1
14
           bits(esize) res = Elem[operand_acc, row*dim+col, esize];
15
            if (ActivePredicateElement(mask1, row, esize) &&
16
                  ActivePredicateElement(mask2, col, esize)) then
17
                res = res + element;
18
            Elem[result, row*dim+col, esize] = res;
19
20
   ZAtile[da, esize, dim*dim*esize] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent or
 - The values of the data supplied in ar of its erand is 's' s when its governing predicate registers contain the same value for each evolution.
 - The values of the NZCV flag^e
- The response of this instruction to as phrone exceptions does not vary based on:
 - The values of the description of its operand registers when its governing predicate registers contain the same since are each executive.
 - The values of he NZCV flag

D1.1.6 ADDSPL

12

3 4 Add multiple of Streaming SVE predicate register size to scalar register

Add the Streaming SVE predicate register size in bytes multiplied by an immediate in the range -32 to 31 to the 64-bit source general-purpose register or current stack pointer and place the result in the 64-bit destination general-purpose register or current stack pointer.

This instruction does not require the PE to be in Streaming SVE mode.

SME (FEAT_SME)



Assembler Symbols

- <Xd|SP> Is the 64-bit name of the destination general-put register or stack pointer, encoded in the "Rd" field.
- <Xn|SP> Is the 64-bit name of the solice georal-pullose register or stack pointer, encoded in the "Rn" field.
 - <imm> Is the signed immedⁱ $_{2}$ o_p $_{3}$ nd, in the ra. $_{2}$ -32 to 31, encoded in the "imm6" field.

Operation

```
1
   CheckSMEEnabled
   constant integ · SVL =
integer len = 1. *
2
                                arrentSVL;
3
                         ★ √L DIV 61);
4
               ororandl
                            ⊥f n ==
                                         then SP[] else X[n, 64];
   bits(64)
5
   bits(64' res
                     t = o_1 rand<sup>1</sup>
                                       len;
6
7
          -= 31 t' n
    if
8
           [] =
9
    else
10
         X[d, 4] = result;
```

Operational in *cormation*

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
 - The values of the data supplied in any of its registers.
 - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
 - The values of the data supplied in any of its registers.
 - The values of the NZCV flags.

D1.1.7 ADDSVL

1 2

3

Add multiple of Streaming SVE vector register size to scalar register

Add the Streaming SVE vector register size in bytes multiplied by an immediate in the range -32 to 31 to the 64-bit source general-purpose register or current stack pointer, and place the result in the 64-bit destination general-purpose register or current stack pointer.

This instruction does not require the PE to be in Streaming SVE mode.

SME (FEAT_SME)



4

Assembler Symbols

- < Xd |SP>Is the 64-bit name of the destination \geq eral-pu resister or stack pointer, encoded in the "Rd" field.
- Is the 64-bit name of the so ce g, pral-pu pse register or stack pointer, encoded in the <Xn|SP> "Rn" field.
 - <imm> Is the signed immediately $o_{\rm h}$ and, in the rate of -32 to 31, encoded in the "imm6" field.

Operation

```
1
   CheckSMEEnabled
   constant integ · SVL =
integer len = 1. *
2
                                arrentSVL;
3
                         * /L DIV 8
4
               ororandl
                            ⊥f n ==
                                         then SP[] else X[n, 64];
   bits(64)
5
   bits(64' res
                     t = o_1 rand<sup>1</sup>
                                       len;
6
7
           -= 31 t' n
    if
8
           [] =
9
    else
10
         X[d, 4] = result;
```

Operational h. ormation

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
 - The values of the data supplied in any of its registers.
 - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
 - The values of the data supplied in any of its registers.
 - The values of the NZCV flags.
D1.1.8 ADDVA

Add vertically vector elements to ZA tile

Add each element of the source vector to the corresponding active element of each vertical slice of a ZA tile. The tile elements are predicated by a pair of governing predicates. An element of a vertical slice is considered active if its corresponding element in the first governing predicate is TRUE and the element corresponding to its vertical slice number in the second governing predicate is TRUE. Inactive elements in the destination tile remain unmodified.

ID_AA64SMFR0_EL1.I16I64 indicates whether the 64-bit integer variant is implemented.

It has encodings from 2 classes: 32-bit and 64-bit



For the 64-bit variant: is the name of the ZA tile ZA0-ZA7, encoded in the "ZAda" field.

- <Pn> Is the name of the first governing scalable predicate register P0-P7, encoded in the "Pn" field.
- <Pm> Is the name of the second governing scalable predicate register P0-P7, encoded in the "Pm" field.
- $<\!\! Zn\!\!>$ Is the name of the source scalable vector register, encoded in the "Zn" field.

```
CheckStreamingSVEAndZAEnabled();
1
   constant integer VL = CurrentVL;
2
3
   constant integer PL = VL DIV 8;
   constant integer dim = VL DIV esize;
4
5
   bits(PL) mask1 = P[a, PL];
6
   bits(PL) mask2 = P[b, PL];
   bits(VL) operand_src = Z[n, VL];
7
8
   bits(dim*dim*esize) operand_acc = ZAtile[da, esize, dim*dim*esize];
9
   bits(dim*dim*esize) result;
10
11
   for row = 0 to dim-1
       bits(esize) element = Elem[operand_src, row, esize];
12
13
        for col = 0 to dim-1
14
           bits(esize) res = Elem[operand_acc, row*dim+col, esize];
15
            if (ActivePredicateElement(mask1, row, esize) &&
16
                  ActivePredicateElement(mask2, col, esize)) then
17
                res = res + element;
18
            Elem[result, row*dim+col, esize] = res;
19
20
   ZAtile[da, esize, dim*dim*esize] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of
 - The values of the data supplied in ar of its erand is 's' s when its governing predicate registers contain the same value for each evolution.
 - The values of the NZCV flag^e
- The response of this instruction to as phrone exceptions does not vary based on:
 - The values of the description of its operand registers when its governing predicate registers contain the same since we each executive.
 - The values of he NZCV flag

D1.1.9 BFCVT

Multi-vector floating-point convert from single-precision to packed BFloat16 format

Convert to BFloat16 from single-precision, each element of the two source vectors, and place the results in the half-width destination elements.

This instruction follows SME2 floating-point numerical behaviors corresponding to instructions that place their results in one or more SVE Z vectors.

This instruction is unpredicated.

SME2 (FEAT_SME2)



<Zn2> Is the name of the sound blable vector gister of a multi-vector sequence, encoded as "Zn" times 2 plus 1.

```
1
   CheckStreaming. "Enab' d();
2
   constant integer
                             Curren+
3
   constant
                   ver e.
                           rents
                                     L DIV 32;
   bits(V
4
              resi
                    ;;
5
   bit VL) opr
                          Z[n+0, VL];
6
                rand2 [n+0, VL];
7
   bits
8
              to elements-1
   for e
9
        bits | element1 = Elem[operand1, e, 32];
        bits(32 element2 = Elem[operand2, e, 32];
bits(16) res1 = FPConvertBF(element1, FPCR[]);
10
11
12
        bits(16) res2 = FPConvertBF(element2, FPCR[]);
13
        Elem[result, e, 16] = res1;
14
        Elem[result, elements+e, 16] = res2;
15
16
   Z[d, VL] = result;
```

D1.1.10 BFCVTN

Multi-vector floating-point convert from single-precision to interleaved BFloat16 format

Convert to BFloat16 from single-precision, each element of the two source vectors, and place the two-way interleaved results in the half-width destination elements.

This instruction follows SME2 floating-point numerical behaviors corresponding to instructions that place their results in one or more SVE Z vectors.

This instruction is unpredicated.

SME2 (FEAT_SME2)



```
1
    CheckStreaming. "Enab' d();
2
    constant integer
                               Curren+
3
    constant
                    rer e.
                                        L DIV 32;
                             rents
    bits(V
4
               resi
                      ;;
5
    bit VL) opr
                            Z[n+0, VL];
6
                 ranaz [n+1, VL];
7
    bits
8
                to elements-1
    for e
9
         bits | element1 = Elem[operand1, e, 32];
         bits(32 element2 = Elem[operand2, e, 32];
bits(16) res1 = FPConvertBF(element1, FPCR[]);
10
11
12
         bits(16) res2 = FPConvertBF(element2, FPCR[]);
         Elem[result, 2*e + 0, 16] = res1;
Elem[result, 2*e + 1, 16] = res2;
13
14
15
16
   Z[d, VL] = result;
```

D1.1.11 BFDOT (multiple and indexed vector)

Multi-vector BFloat16 floating-point dot-product by indexed element

The instruction operates on two or four ZA single-vector groups.

The instruction computes the dot product of a pair of BF16 values held in the corresponding 32-bit elements of the two or four first source vectors and the indexed 32-bit element of the second source vector. The single-precision dot product results are destructively added to the corresponding single-precision elements of the two or four ZA single-vector groups.

The BF16 pairs within the second source vector are specified using an immediate index which selects the same BF16 pair position within each 128-bit vector segment. The element index range is from 0 to 3. The vector numbers forming the single-vector group within each half or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the the ber of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA oper .d consists o. vo or four ZA single-vector groups respectively. The VECTOR GROUP symbol is preferred for dis embly, but o₁ onal in assembler source code.

This instruction follows SME2 ZA-targeting BFloat16 num .cal behaviors.

This instruction is unpredicated.

It has encodings from 2 classes: Two ZA single-ve and by ZA single-vectors

Two ZA single-vectors (FEAT_SME2)

```
19
                                                                                                        off3
                1
                   0
                      0
                         0
                            0
                               0
                                  1
                                     0
                                            0
                                                              0
                                                                  Rv
                                                                       1
                                                                            i2
                                                                                     Zn
                                                                                             0
                                                                                                1
                                                                                                   1
   BFDOT
             ZA.S[\langle Wv \rangle, \langle offs \rangle \{,
                                            ?}], { <Zn1>.H-<Zn2>.H }, <Zm>.H[<index>]
   if !HaveSME2() t<sup>+</sup> . UNDEFINED;
1
   integer v = UIr ('010':F
2
                                   ;:
   integer n = U1 (Zn:')
3
4
   integer m = UInt 0'
                              _m);
                            t(off3)
5
   integer o
                      - UInu
6
   integer .nde.
                               2);
7
   const it inte ir nreg
   Four.
             s' gle-vectors
   (FEAT_\ 'E2)
                                     0
                                           0
                                                      Zm
                1
                      0
                         0
                            0
                               0
                                   1
                                         1
                                               1
                                                              1
                                                                   Rv
                                                                            i2
                                                                                    Zn
                                                                                          0
                                                                                             0
                                                                                                1
                                                                                                        off3
```

BFDOT ZA.S[<Wv>, <offs>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H[<index>]

```
if !HaveSME2() then UNDEFINED;
```

```
2 integer v = UInt('010':Rv);
```

```
3 integer n = UInt(Zn:'00');
4 integer m = UInt('0':Zm);
```

```
5 integer offset = UInt(off3);
```

```
6 integer index = UInt(i2);
```

```
7 constant integer nreg = 4;
```

Assembler Symbols

<Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.

Chapter D1. SME instructions D1.1. SME and SME2 data-processing instructions

- <offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.
- <Zn1> For the two ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.

For the four ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

- <Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.
- <Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.
- <Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

<index> Is the element index, in the range 0 to 3, encoded in the "i2" field.

```
1
     CheckStreamingSVEAndZAEnabled();
    constant integer VL = CurrentVL;
 2
 3
     constant integer elements = VL DIV 32;
    integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
 4
 5
 6
     integer eltspersegment = 128 DIV 32;
 7
     bits(32) vbase = X[v, 32];
     integer vec = (UInt(vbase) + offset) MC
 8
                                                               vstri
 9
     bits(VL) result;
10
     for r = 0 to nreg-1
11
          bits(VL) operand1 = Z[n+r, VL,
12
13
          bits(VL) operand2 = Z[m, ];
          bits(VL) operand3 = ZAv
14
                                             cor[vec,
           for e = 0 to elements
15
               bits(16) elt1_a El 'operand1, 2 e + 0, 16];
bits(16) elt1_b = Elem perand1, 2 * e + 1, 16];
integer seg ntbase = e 'e MOD eltspersegment);
integer s segmentbase + dex;
16
17
18
19
                bits(16' elt2_a Elem[operand2, 2 * s + 0, 16];
bits(' elt2_' = Elem[operand2, 2 * s + 1, 16];
bits(32, sum Elem[operand3, e, 32];
20
21
22
                sum = BFL id(sum, c1_a, elt1_b, elt2_a, elt2_b, FPCR[]);
rem esult a, 32 = sum;
23
24
25
           Z' ector[ c, VL]
                                        sult;
             ec = ver + vstride
26
```

D1.1.12 BFDOT (multiple and single vector)

Multi-vector BFloat16 floating-point dot-product by vector

The instruction operates on two or four ZA single-vector groups.

The instruction computes the dot product of a pair of BF16 values held in the corresponding 32-bit elements of the two or four first source vectors and the second source vector. The single-precision dot product results are destructively added to the corresponding single-precision elements of the two or four ZA single-vector groups. The vector numbers forming the single-vector group within each half or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction follows SME2 ZA-targeting BFloat16 numerical behr .ors.

This instruction is unpredicated.

It has encodings from 2 classes: Two ZA single-vectors and ^r ar ZA single ⁻ ctor

Two ZA single-vectors (FEAT_SME2)

0 0 0 0 0 0 off3 1 1 0 0 1 0 Ζn Rv 0 Zn 1

BFDOT ZA.S[<Wv>, <offs>{, 'x2}, { <Zn .H-<Zn2>.H }, <Zm>.H

```
1 if !HaveSME2() then UNDEF
```

- 2 integer v = UInt('010':' /;
- 3 integer n = UInt(Zn);
- 4 integer m = UInt('0' Lm);
- 5 integer offset = ' .it(off3);

```
Four ZA single-ve
```

1

1



ß

0 0 6 1

BFDOT ZA.. (Wv>, <offs>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H

Zm

0 Rv

0 0

1

Zn

0

1

off3

0

0 1 1

```
1 if !HaveSME2() then UNDEFINED;
2 integer v = UInt('010':Rv);
3 integer n = UInt(Zn);
4 integer m = UInt('0':Zm);
```

- 5 integer offset = UInt(off3);
- 6 constant integer nreg = 4;

Assembler Symbols

- <Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
- <offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.
- <Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn".
- <Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn"

plus 3 modulo 32.

- <Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" plus 1 modulo 32.
- <Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

```
1
    CheckStreamingSVEAndZAEnabled();
2
   constant integer VL = CurrentVL;
3
    constant integer elements = VL DIV 32;
   integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
 4
5
6
   bits(32) vbase = X[v, 32];
7
    integer vec = (UInt(vbase) + offset) MOD vstride;
8
    bits(VL) result;
9
10
    for r = 0 to nreg-1
11
        bits(VL) operand1 = Z[(n+r) MOD 32, VL];
12
        bits(VL) operand2 = Z[m, VL];
13
        bits(VL) operand3 = ZAvector[vec, VL];
14
        for e = 0 to elements-1
                                                            , 16];
15
             bits(16) elt1_a = Elem[operand1, 2 * e
             bits(16) elt1_b = Elem[operand1, 2 * e +
16
                                                              16
            bits(16) elt2_a = Elem[operand2, 2 * e + 0,
bits(16) elt2_b = Elem[operand2, ^ + 1,
17
                                                                 1
18
19
             bits(32) sum = Elem[operand3, e .2];
20
             sum = BFDotAdd(sum, elt1_a, e' _b, el' _a, elt2_ , FPCR[]);
             Elem[result, e, 32] = sum;
21
22
        ZAvector[vec, VL] = result;
23
        vec = vec + vstride;
```

D1.1.13 BFDOT (multiple vectors)

Multi-vector BFloat16 floating-point dot-product

The instruction operates on two or four ZA single-vector groups.

The instruction computes the dot product of a pair of BF16 values held in the corresponding 32-bit elements of the two or four first and second source vectors. The single-precision dot product results are destructively added to the corresponding single-precision elements of the two or four ZA single-vector groups. The vector numbers forming the single-vector group within each half or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction follows SME2 ZA-targeting BFloat16 numerical behr lors.

This instruction is unpredicated.

It has encodings from 2 classes: Two ZA single-vectors and ^r ar ZA single ⁻ ctor

Two ZA single-vectors (FEAT_SME2)

 31
 30
 29
 23
 22
 21
 20
 17
 1x
 5
 14
 2
 9
 6
 5
 4
 3
 2
 0

 1
 1
 0
 0
 0
 1
 1
 0
 1
 0
 Rv
 0
 Zn
 0
 1
 0
 off3

```
BFDOT ZA.S[<Wv>, <offs>{, '`x2} { <Zn .H-<Zn2>.H }, { <Zm1>.H-<Zm2>.H }
```

```
1 if !HaveSME2() then UNDEFT
```

- 2 integer v = UInt('010':F
- 3 integer n = UInt(Zn:'0
 4 integer m = UInt(Zm:')')
- 4 integer m = UInt(Zm·)');
 5 integer offset = V .t(off3);
- 6 constant integer reg = 2:

```
Four ZA single-v. 'or'
(FEAT S'
```



BFDOT ZA. <Wv>, <offs>{, VGx4}], { <Zn1>.H-<Zn4>.H }, { <Zm1>.H-<Zm4>.H }

1 0 Rv

0

Zm

Zn

0 0 1 0

0 0

off3

```
1 if !HaveSME2() then UNDEFINED;
```

```
2 integer v = UInt('010':Rv);
```

```
3 integer n = UInt(Zn:'00');
```

```
4 integer m = UInt(Zm:'00');
```

```
5 integer offset = UInt(off3);
6 constant integer nreg = 4;
```

```
_____
```

Assembler Symbols

- <Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
- <offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.

0

1

<Zn1> For the two ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.

For the four ZA single-vectors variant: is the name of the first scalable vector register of a

multi-vector sequence, encoded as "Zn" times 4.

- <Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.
- <Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.
- <Zm1> For the two ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 2.

For the four ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 4.

- <Zm4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zm" times 4 plus 3.
- <Zm2> Is the name of the second scalable vector register of a multi-v for sequence encoded as "Zm" times 2 plus 1.

```
1
   CheckStreamingSVEAndZAEnabled();
 2
   constant integer VL = CurrentVL;
3
   constant integer elements = VL DIV 32;
4
   integer vectors = VL DIV 8;
 5
    integer vstride = vectors DIV nreg;
   bits(32) vbase = X[v, 32];
 6
   integer vec = (UInt(vbase) + offset)
 7
                                              D vstr
 8
   bits(VL) result;
9
10
   for r = 0 to nreg-1
11
        bits(VL) operand1 = Z[n+
                                      VL];
        bits(VL) operand2 = Z[ c, VL];
12
                                    or[vec, VL]
13
        bits(VL) operand3 =
14
        for e = 0 to elemen s-1
            bits(16) elt a = Elem[ rand1, 2 * e + 0, 16];
15
            bits(16) e' __b = Elem[op __nd1, 2 * e + 1, 16];
16
            bits(16) it2_a = Elem[oper..d2, 2 * e + 0, 16];
bits(1 elt2_b Elem[operand2, 2 * e + 1, 16];
17
18
            bits(3. sum _lem[operand3, e, 32];
19
             sum = BF. ____d(sum, e__1_a, elt1_b, elt2_a, elt2_b, FPCR[]);
20
            ctor, c, VL = r
21
                                    sum;
22
        7. A 1
                                alt:
23
            = vec · vstria
```

D1.1.14 BFMLAL (multiple and indexed vector)

Multi-vector BFloat16 floating-point multiply-add long by indexed element

The instruction operates on one, two, or four ZA double-vector groups.

This BFloat16 floating-point multiply-add long instruction widens all 16-bit BFloat16 elements in the one, two, or four first source vectors and the indexed element of the second source vector to single-precision format, then multiplies the corresponding elements and destructively adds these values without intermediate rounding to the overlapping 32-bit single-precision elements of the one, two, or four ZA double-vector groups.

The BF16 elements within the second source vector are specified using a 3-bit immediate index which selects the same element position within each 128-bit vector segment.

The lowest of the two consecutive vector numbers forming the double-vector group within all, each half, or each quarter of the ZA array are selected by the sum of the vector select register and module offset, modulo all, half, or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the λ operand consists of two or four ZA double-vector groups respectively. The VECTOR GROUP states in a symbol is provided in assembler source code.

This instruction follows SME ZA-targeting floating-point having

This instruction is unpredicated.

It has encodings from 3 classes: One ZA dou¹ -vector Iwo ZA c Je-vectors and Four ZA double-vectors

One ZA double-vector (FEAT_SME2)



BFMLAL ZA.S[<Wy , <offsf>:<offsl, , <Zn>.H, <Zm>.H[<index>]

```
1 if !HaveSME2() 'en U LFINED;
```

2 integer v = UInt (2); 3 integer z Int (2);

```
5 inte c offse = UInt( 3:'0');
```

```
6 int ar inde - Wint(i3h:i3l);
```

```
7 boole sr op =  L;
```

```
8 constal. .nteger nreg =
```

Two ZA dou. -vectors (FEAT_SME2)

BFMLAL ZA.S[<Wv>, <offsf>:<offsl>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H[<index>]

```
1 if !HaveSME2() then UNDEFINED;
```

```
2 integer v = UInt('010':Rv);
```

```
3 integer n = UInt(Zn:'0');
```

```
4 integer m = UInt('0':Zm);
```

```
5 integer offset = UInt(off2:'0');
6 integer index = UInt(i3h:i3l);
```

```
7 boolean sub_op = FALSE;
```

```
8 constant integer nreg = 2;
```



```
1 if !HaveSME2() then UNDEFINED;
```

```
2 integer v = UInt('010':Rv);
```

```
3 integer n = UInt(Zn:'00');
```

```
4 integer m = UInt('0':Zm);
```

```
5 integer offset = UInt(off2:'0');
6 integer '...'
```

6 integer index = UInt(i3h:i3l); 7 boolean sub_op = FALSE;

```
8 constant integer nreg = 4;
```

Assembler Symbols

- $\langle Wv \rangle$ Is the 32-bit name of the vector select register W8. 11, encoding the select register W8. 11, encoding the select register W8.
- <offsf> For the one ZA double-vector variant: is the vector electoffset, rointing to first of two consecutive vectors, encoded as "off3" field s 2.

For the four ZA double-vectors and two A double vectors v at: is the vector select offset, pointing to first of two consecutive v tors, enc v and v for r field times 2.

<offsl> For the one ZA double-vector riant: . .e vector select offset, pointing to last of two consecutive vectors, encoder as "o, "" field nes 2 plus 1.

- <Zn> Is the name of the first source lable vector register, encoded in the "Zn" field.
- <Zn1> For the two value-vectors value: is the name of the first scalable vector register of a multi-vector sequered as "Zn" times 2.

For the four \land ouble-vectors variant: is the name of the first scalable vector register of a m or sequence, ϵ oded as "Zn" times 4.

- <Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times r
- <Zn2> ls e name of the second scalable vector register of a multi-vector sequence, encoded as "Zn nes 2 plus 1.
- <Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

<index> Is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields.

```
CheckStreamingSVEAndZAEnabled();
1
2
   constant integer VL = CurrentVL;
   constant integer elements = VL DIV 32;
3
4
   integer vectors = VL DIV 8;
   integer vstride = vectors DIV nreg;
5
   integer eltspersegment = 128 DIV 32;
6
   bits(32) vbase = X[v, 32];
7
   integer vec = (UInt(vbase) + offset) MOD vstride;
8
   bits(VL) result;
9
10
   vec = vec - (vec MOD 2);
11
12 for r = 0 to nreg-1
```

```
Chapter D1. SME instructions
D1.1. SME and SME2 data-processing instructions
```

```
bits(VL) operand1 = Z[n+r, VL];
13
14
         bits(VL) operand2 = Z[m, VL];
15
          for i = 0 to 1
16
              bits(VL) operand3 = ZAvector[vec + i, VL];
17
               for e = 0 to elements-1
18
                    integer segmentbase = e - (e MOD eltspersegment);
                    integer s = 2 * segmentbase + index;
19
                   bits(16) element1 = Elem[operand1, 2 * e + i, 16];
bits(16) element2 = Elem[operand2, s, 16];
bits(32) element3 = Elem[operand3, e, 32];
20
21
22
23
                    if sub_op then element1 = BFNeg(element1);
               Elem[result, e, 32] = BFMulAddH_ZA(element3, element1, element2, FPCR[]);
ZAvector[vec + i, VL] = result;
24
25
26
         vec = vec + vstride;
```

D1.1.15 BFMLAL (multiple and single vector)

Multi-vector BFloat16 floating-point multiply-add long by vector

The instruction operates on one, two, or four ZA double-vector groups.

This BFloat16 floating-point multiply-add long instruction widens all 16-bit BFloat16 elements in the one, two, or four first source vectors and the second source vector to single-precision format, then multiplies the corresponding elements and destructively adds these values without intermediate rounding to the overlapping 32-bit single-precision elements of the one, two, or four ZA double-vector groups. The lowest of the two consecutive vector numbers forming the double-vector group within all, each half, or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo all, half, or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA d consists of two or four ZA double-vector groups respectively. The VECTOR GROUP symbol is referred disassembly, but optional in assembler source code.

This instruction follows SME ZA-targeting floating-point behaviors.

This instruction is unpredicated.

It has encodings from 3 classes: One ZA double-vector, ZA _ uble-vectors and Four ZA double-vectors

One ZA double-vector (FEAT_SME2)



BFMLAL ZA.S[<Wv>, <offsf>:<offsl>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H

```
1 if !HaveSME2() then UNDEFINED;
```

- 2 integer v = UInt('010':Rv);
- 3 integer n = UInt(Zn);
- 4 integer m = UInt('0':Zm);
 5 integer offset = UInt(off2:'0')
- 5 integer offset = UInt(off2:'0'); 6 boolean sub_op = FALSE;
- 7 constant integer nreg = 2;

Four ZA double-vectors (FEAT_SME2)



17

18 19

20

21

22

23

Elem[result, e, 32] = BFMulAddH_ZA(element3, element1, element2, FPCR[]);

bits(16) element1 = Elem[operand1, 2 * e + i, 16]; bits(16) element2 = Elem[operand2, 2 * e + i, 16];

bits(32) element3 = Elem[operand3, e, 32];

if sub_op then element1 = BFNeg(element1);

ZAvector[vec + i, VL] = result;

vec = vec + vstride;

D1.1.16 BFMLAL (multiple vectors)

Multi-vector BFloat16 floating-point multiply-add long

The instruction operates on two or four ZA double-vector groups.

This BFloat16 floating-point multiply-add long instruction widens all 16-bit BFloat16 elements in the two or four first and second source vectors to single-precision format, then multiplies the corresponding elements and destructively adds these values without intermediate rounding to the overlapping 32-bit single-precision elements of the two or four ZA double-vector groups. The lowest of the two consecutive vector numbers forming the double-vector group within each half or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA double-vector groups respectively. The VECTOR GROUP symbol is profor disassembly, but optional in assembler source code.

This instruction follows SME ZA-targeting floating-point behaviors.

This instruction is unpredicated.

It has encodings from 2 classes: Two ZA double-vectors Four Z' Jouble

Two ZA double-vectors (FEAT_SME2)

```
13
1
   0
       0
          0
              0
                 0
                     1
                         1
                            0
                                1
                                         7m
                                                      0
                                                           Rv
                                                                0
                                                                        0
                                                                                 7n
                                                                                          0
```

BFMLAL ZA.S[<Wv>, <offsf <offsl>{ x2}] <Zn1>.H-<Zn2>.H }, { <Zm1>.H-<Zm2>.H }

```
if !HaveSME2() then UNL_FINED;
1
```

```
:Rv);
2
  integer v = UInt('0'
```

- 3 integer n = UInt(7 0');
- integer m = UIn+ _m:'0') 4
- 5 integer offset UInt (r 2: (0):FAL

= 2:

- 6 boolean sub op
- constant integer

Four' A doub vectors (FI **5** SMF



```
BFMLAL ZA.S[<Wv>, <offsf>:<offsl>{, VGx4}], { <Zn1>.H-<Zn4>.H }, { <Zm1>.H-<Zm4>.H }
```

```
if !HaveSME2() then UNDEFINED;
1
```

```
2
  integer v = UInt('010':Rv);
```

```
integer n = UInt(Zn:'00');
3
```

```
4
  integer m = UInt(Zm:'00');
```

```
integer offset = UInt(off2:'0');
boolean sub_op = FALSE;
5
```

```
6
  constant integer nreg = 4;
```

Assembler Symbols

Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field. $\langle Wv \rangle$

<offsf> Is the vector select offset, pointing to first of two consecutive vectors, encoded as "off2" field times 2.

- <offsl> Is the vector select offset, pointing to last of two consecutive vectors, encoded as "off2" field times 2 plus 1.
- <Zn1> For the two ZA double-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.

For the four ZA double-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

- <Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.
- <Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.
- <Zm1> For the two ZA double-vectors variant: is the name of the first alable vec, register of a multi-vector sequence, encoded as "Zm" times 2.

For the four ZA double-vectors variant: is the name of *t* first sc. ble vecto egister of a multi-vector sequence, encoded as "Zm" times 4.

- <Zm4> Is the name of the fourth scalable vector register f a mu' vector sequence, encoded as "Zm" times 4 plus 3.
- <Zm2> Is the name of the second scalable vector regist of a noticity for sequence, encoded as "Zm" times 2 plus 1.

```
1
   CheckStreamingSVEAndZAEnabled
   constant integer VL = Curre
2
                                   /T.:
3
   constant integer elements
                                  VL DIV 32;
   integer vectors = VL DI' ;
integer vstride = vectors DIV
 4
5
                                       ٦g;
 6
   bits(32) vbase = X[, 32];
    integer vec = (UIr (vbase) + offse
                                            MOD vstride;
 7
 8
   bits(VL) result
 9
   vec = vec - (
                     MOD 2
10
11
    for r = 0 + o nreg
12
        bit
                  peran.
                            = Z[
                                 1, VL];
        (VL) د <sup>ن</sup>ط
13
                   erand∠
                                .a+r, VL];
14
         Jr i =
                 f co 1
15
                          rand3 = ZAvector[vec + i, VL];
            bit
                 e = 0 to elements-1
16
            f
17
                 bits(16) element1 = Elem[operand1, 2 * e + i, 16];
                 bits(16) element2 = Elem[operand2, 2 * e + i, 16];
18
19
                  `ts(32) element3 = Elem[operand3, e, 32];
20
                 if sub_op then element1 = BFNeg(element1);
21
                 Elem[result, e, 32] = BFMulAddH_ZA(element3, element1, element2, FPCR[]);
22
            ZAvector[vec + i, VL] = result;
23
        vec = vec + vstride;
```

D1.1.17 BFMLSL (multiple and indexed vector)

Multi-vector BFloat16 floating-point multiply-subtract long by indexed element

The instruction operates on one, two, or four ZA double-vector groups.

This BFloat16 floating-point multiply-subtract long instruction widens all 16-bit BFloat16 elements in the one, two, or four first source vectors and the indexed element of the second source vector to single-precision format, then multiplies the corresponding elements and destructively subtracts these values without intermediate rounding from the overlapping 32-bit single-precision elements of the one, two, or four ZA double-vector groups.

The BF16 elements within the second source vector are specified using a 3-bit immediate index which selects the same element position within each 128-bit vector segment.

The lowest of the two consecutive vector numbers forming the double-vector group within all, each half, or each quarter of the ZA array are selected by the sum of the vector select register and module offset, modulo all, half, or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the λ operand consists of two or four ZA double-vector groups respectively. The VECTOR GROUP symbol is provided for disassembly, but optional in assembler source code.

This instruction follows SME ZA-targeting floating-point having

This instruction is unpredicated.

It has encodings from 3 classes: One ZA dou¹ -vector Iwo ZA c Je-vectors and Four ZA double-vectors

One ZA double-vector (FEAT_SME2)



BFMLSL ZA.S[<Wy , <offsf>:<offsl ,, <Zn>.H, <Zm>.H[<index>]

```
1 if !HaveSME2() 'en U' EFINED;
2 integer v = UInt ':Rv);
3 integer ' Int(Z.')
```

```
3 integer ... Tnt(2...
4 integer ... = u nt('0' n);
```

```
5 inte c offse = UInt( 3:'0');
```

```
6 int or inde - "Int(i3h:i3l);
```

```
7 boole st op = i
```

```
8 constan .nteger nreg =
```

Two ZA dou. -vectors (FEAT_SME2)

BFMLSL ZA.S[<Wv>, <offsf>:<offsl>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H[<index>]

```
1 if !HaveSME2() then UNDEFINED;
```

```
2 integer v = UInt('010':Rv);
```

```
3 integer n = UInt(Zn:'0');
```

```
4 integer m = UInt('0':Zm);
```

```
5 integer offset = UInt(off2:'0');
```

```
6 integer index = UInt(i3h:i3l);
7 boolean sub on = TRUE:
```

```
7 boolean sub_op = TRUE;
8 constant integer nreg = 2;
```



```
8 constant integer nreg = 4;
```

Assembler Symbols

- <Wv> Is the 32-bit name of the vector select register W8. 11, encrod in the r field.
- <offsf> For the one ZA double-vector variant: is the vector electoffset, pointing to first of two consecutive vectors, encoded as "off3" field s 2.

For the four ZA double-vectors and two A double vectors v at: is the vector select offset, pointing to first of two consecutive v fors, enc of as "off2" field times 2.

<offsl> For the one ZA double-vector riant: . . .e vector select offset, pointing to last of two consecutive vectors, encoder as "o, "" field thes 2 plus 1.

For the four ZA double- tors and two \ double-vectors variant: is the vector select offset, pointing to last of two concentrative vectors incoded as "off2" field times 2 plus 1.

- <Zn> Is the name of the first source lable vector register, encoded in the "Zn" field.
- <Zn1> For the two soluble-vectors van. ht: is the name of the first scalable vector register of a multi-vec sequer, encoded as "Zn" times 2.

For the four Λ suble-vec rs variant: is the name of the first scalable vector register of a m and tor sequence, ϵ solded as "Zn" times 4.

- <Zn4> Is the name of the to the scalable vector register of a multi-vector sequence, encoded as "Zn" times r
- <Zn2> ls e name of the second scalable vector register of a multi-vector sequence, encoded as "Zn nes 2 plus 1.
- <Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

<index> Is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields.

```
CheckStreamingSVEAndZAEnabled();
1
   constant integer VL = CurrentVL;
2
   constant integer elements = VL DIV 32;
3
4
   integer vectors = VL DIV 8;
   integer vstride = vectors DIV nreg;
5
   integer eltspersegment = 128 DIV 32;
6
   bits(32) vbase = X[v, 32];
7
   integer vec = (UInt(vbase) + offset) MOD vstride;
8
   bits(VL) result;
9
10
   vec = vec - (vec MOD 2);
11
12 for r = 0 to nreg-1
```

```
Chapter D1. SME instructions
D1.1. SME and SME2 data-processing instructions
```

```
bits(VL) operand1 = Z[n+r, VL];
13
14
         bits(VL) operand2 = Z[m, VL];
15
          for i = 0 to 1
16
              bits(VL) operand3 = ZAvector[vec + i, VL];
17
               for e = 0 to elements-1
18
                    integer segmentbase = e - (e MOD eltspersegment);
                    integer s = 2 * segmentbase + index;
19
                   bits(16) element1 = Elem[operand1, 2 * e + i, 16];
bits(16) element2 = Elem[operand2, s, 16];
bits(32) element3 = Elem[operand3, e, 32];
20
21
22
23
                    if sub_op then element1 = BFNeg(element1);
               Elem[result, e, 32] = BFMulAddH_ZA(element3, element1, element2, FPCR[]);
ZAvector[vec + i, VL] = result;
24
25
26
         vec = vec + vstride;
```

D1.1.18 BFMLSL (multiple and single vector)

Multi-vector BFloat16 floating-point multiply-subtract long by vector

The instruction operates on one, two, or four ZA double-vector groups.

This BFloat16 floating-point multiply-subtract long instruction widens all 16-bit BFloat16 elements in the one, two, or four first source vectors and the second source vector to single-precision format, then multiplies the corresponding elements and destructively subtracts these values without intermediate rounding from the overlapping 32-bit single-precision elements of the one, two, or four ZA double-vector groups. The lowest of the two consecutive vector numbers forming the double-vector group within all, each half, or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo all, half, or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA d consists of two or four ZA double-vector groups respectively. The VECTOR GROUP symbol is referred disassembly, but optional in assembler source code.

This instruction follows SME ZA-targeting floating-point behaviors.

This instruction is unpredicated.

It has encodings from 3 classes: One ZA double-vector, ZA _ uble-vectors and Four ZA double-vectors

One ZA double-vector (FEAT_SME2)



BFMLSL ZA.S[<Wv>, <offsf>:<offsl>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H

```
1 if !HaveSME2() then UNDEFINED;
```

```
2 integer v = UInt('010':Rv);
```

```
3 integer n = UInt(Zn);
4 integer m = UInt('0':
```

```
4 integer m = UInt('0':Zm);
5 integer offset = UInt(off2:'0');
```

```
6 boolean sub_op = TRUE;
```

```
7 constant integer nreg = 2;
```

Four ZA double-vectors (FEAT_SME2)



19

20

21

22

23

Elem[result, e, 32] = BFMulAddH_ZA(element3, element1, element2, FPCR[]);

bits(32) element3 = Elem[operand3, e, 32];

if sub_op then element1 = BFNeg(element1);

ZAvector[vec + i, VL] = result;

vec = vec + vstride;

D1.1.19 BFMLSL (multiple vectors)

Multi-vector BFloat16 floating-point multiply-subtract long

The instruction operates on two or four ZA double-vector groups.

This BFloat16 floating-point multiply-subtract long instruction widens all 16-bit BFloat16 elements in the two or four first and second source vectors to single-precision format, then multiplies the corresponding elements and destructively subtracts these values without intermediate rounding from the overlapping 32-bit single-precision elements of the two or four ZA double-vector groups. The lowest of the two consecutive vector numbers forming the double-vector group within each half or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA double-vector groups respectively. The VECTOR GROUP symbol is pro for disassembly, but optional in assembler source code.

This instruction follows SME ZA-targeting floating-point behaviors.

This instruction is unpredicated.

It has encodings from 2 classes: Two ZA double-vectors Four Z' Jouble

Two ZA double-vectors (FEAT_SME2)

```
13
1
   0
      0
          0
             0
                 0
                     1
                        1
                            0
                                1
                                        7m
                                                     0
                                                          Rv
                                                               0
                                                                       0
                                                                               7n
                                                                                        0
```

<offsl>{, BFMLSL ZA.S[<Wv>, <offsf x2}] <Zn1>.H-<Zn2>.H }, { <Zm1>.H-<Zm2>.H }

```
if !HaveSME2() then UNL_FINED;
1
```

```
:Rv);
2
  integer v = UInt('0'
```

- 3 integer n = UInt(7 0');
- integer m = UIn+ _m:'0') 4
- 5 integer offset UInt (r 2: (0):TRUF

= 2;

- boolean sub_op 6
- constant integer

Four' A doub vectors (FI **5** SMF



```
BFMLSL ZA.S[<Wv>, <offsf>:<offsl>{, VGx4}], { <Zn1>.H-<Zn4>.H }, { <Zm1>.H-<Zm4>.H }
```

```
if !HaveSME2() then UNDEFINED;
1
```

```
2
  integer v = UInt('010':Rv);
```

```
integer n = UInt(Zn:'00');
3
```

```
4
  integer m = UInt(Zm:'00');
5
```

```
integer offset = UInt(off2:'0');
6
```

boolean sub_op = TRUE; constant integer nreg = 4;

Assembler Symbols

Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field. $\langle Wv \rangle$

<offsf> Is the vector select offset, pointing to first of two consecutive vectors, encoded as "off2" field times 2.

- <offsl> Is the vector select offset, pointing to last of two consecutive vectors, encoded as "off2" field times 2 plus 1.
- <Zn1> For the two ZA double-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.

For the four ZA double-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

- <Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.
- <Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.
- <Zm1> For the two ZA double-vectors variant: is the name of the first alable vec, register of a multi-vector sequence, encoded as "Zm" times 2.

For the four ZA double-vectors variant: is the name of *t* first sc. ble vecto egister of a multi-vector sequence, encoded as "Zm" times 4.

- <Zm4> Is the name of the fourth scalable vector register f a mu' vector sequence, encoded as "Zm" times 4 plus 3.
- <Zm2> Is the name of the second scalable vector regist of a noticity for sequence, encoded as "Zm" times 2 plus 1.

```
1
   CheckStreamingSVEAndZAEnabled
   constant integer VL = Curre
2
                                   /T.:
3
   constant integer elements
                                  VL DIV 32;
   integer vectors = VL DI' ;
integer vstride = vectors DIV
 4
5
                                       ٦g;
 6
   bits(32) vbase = X[, 32];
    integer vec = (UIr (vbase) + offse
                                            MOD vstride;
 7
 8
   bits(VL) result
 9
   vec = vec - (
                     MOD 2
10
11
    for r = 0 + o nreg
12
        bit
                  peran.
                            = Z[
                                 1, VL];
        (VL) د <sup>ن</sup>ط
13
                   erand∠
                                .a+r, VL];
14
         Jr i =
                 f co 1
15
                          rand3 = ZAvector[vec + i, VL];
            bit
                 e = 0 to elements-1
16
            f
17
                 bits(16) element1 = Elem[operand1, 2 * e + i, 16];
                 bits(16) element2 = Elem[operand2, 2 * e + i, 16];
18
19
                  `ts(32) element3 = Elem[operand3, e, 32];
20
                 if sub_op then element1 = BFNeg(element1);
21
                 Elem[result, e, 32] = BFMulAddH_ZA(element3, element1, element2, FPCR[]);
22
            ZAvector[vec + i, VL] = result;
23
        vec = vec + vstride;
```

D1.1.20 BFMOPA

BFloat16 sum of outer products and accumulate

The BFloat16 floating-point sum of outer products and accumulate instruction works with a 32-bit element ZA tile.

This instruction multiplies the $SVL_S \times 2$ sub-matrix of BFloat16 values held in the first source vector by the $2 \times SVL_S$ sub-matrix of BFloat16 values in the second source vector.

Each source vector is independently predicated by a corresponding governing predicate. When a 16-bit source element is Inactive it is treated as having the value +0.0, but if both pairs of source vector elements that correspond to a 32-bit destination element contain Inactive elements, then the destination element remains unmodified.

The resulting $SVL_S \times SVL_S$ single-precision floating-point sum of outer products is then destructively added to the single-precision floating-point destination tile. This is equivalent to performing a 2-way dot product and accumulate to each of the destination tile elements.

Each 32-bit container of the first source vector holds 2 consecutive summer of each row of a $SVL_S \times 2$ sub-matrix. Similarly, each 32-bit container of the second source vector olds 2 consecutive row elements of each column of a $2 \times SVL_S$ sub-matrix.

This instruction follows SME BFloat16 numerical behavi/

```
SME
(FEAT_SME)
```



BFMOPA <ZAda>.S, <Pn>/M <Pm>/M, <2 .H, <_m>

1 if !HaveSME() then UNDL.INED;

```
2 integer a = UInt(Pn)
```

- 3 integer b = UInt(F /;
- 4 integer n = UIn⁺ In);
- 5 integer $m = U^{+}$ (Zm);
- 6 integer $da = U_{\perp}$ (ZAc)
- 7 **boolean** sub op = Jf

Asser her Syr ols

```
<ZAda> Is the ..... ZA tile ZA0-ZA3, encoded in the "ZAda" field.
```

- <Pn> Is name of the first governing scalable predicate register P0-P7, encoded in the "Pn" field.
- <Pm> Is the . me of the second governing scalable predicate register P0-P7, encoded in the "Pm" field.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
- <Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

```
1
   CheckStreamingSVEAndZAEnabled();
2
   constant integer VL = CurrentVL;
   constant integer PL = VL DIV 8;
3
Δ
   constant integer dim = VL DIV 32;
5
   bits(PL) mask1 = P[a, PL];
6
   bits(PL) mask2 = P[b, PL];
   bits(VL) operand1 = Z[n, VL];
7
   bits(VL) operand2 = Z[m, VL];
8
9
   bits(dim*dim*32) operand3 = ZAtile[da, 32, dim*dim*32];
10
   bits(dim*dim*32) result;
```

11

```
12
    for row = 0 to dim-1
13
        for col = 0 to dim-1
14
             // determine row/col predicates
15
             boolean prow_0 = (ActivePredicateElement(mask1, 2*row + 0, 16));
             boolean prow_1 = (ActivePredicateElement(mask1, 2*row + 1, 16));
boolean pcol_0 = (ActivePredicateElement(mask2, 2*col + 0, 16));
16
17
18
             boolean pcol_1 = (ActivePredicateElement(mask2, 2*col + 1, 16));
19
20
             bits(32) sum = Elem[operand3, row*dim+col, 32];
21
             if (prow_0 && pcol_0) || (prow_1 && pcol_1) then
22
                 bits(16) erow_0 = (if prow_0 then Elem[operand1, 2*row + 0, 16] else FPZero('0',
                       →16));
23
                 bits(16) erow_1 = (if prow_1 then Elem[operand1, 2*row + 1, 16] else FPZero('0',
                       →16));
24
                 bits(16) ecol_0 = (if pcol_0 then Elem[operand2, 2*col + 0, 16] else FPZero('0',
                      \rightarrow16));
25
                 bits(16) ecol_1 = (if pcol_1 then Elem[operand2 _*col +
                                                                                      16] else FPZero('0',
                       →16));
26
                  if sub_op then
27
                      boolean honor_altfp = FALSE;
                                                         // Altr nate
                                                                           ndling ig red
                      if prow_0 then erow_0 = BFNeg(erow_0 honor_al o);
if prow_1 then erow_1 = BFNeg(erov_1, honor_alt)
28
29
30
                  sum = BFDotAdd(sum, erow_0, erow_1, _ol_0, < _l_1, Fr _(]);</pre>
31
32
             Elem[result, row*dim+col, 32] = sum;
33
34
    ZAtile[da, 32, dim*dim*32] = result;
```

D1.1.21 BFMOPS

BFloat16 sum of outer products and subtract

The BFloat16 floating-point sum of outer products and subtract instruction works with a 32-bit element ZA tile.

This instruction multiplies the $SVL_S \times 2$ sub-matrix of BFloat16 values held in the first source vector by the $2 \times SVL_S$ sub-matrix of BFloat16 values in the second source vector.

Each source vector is independently predicated by a corresponding governing predicate. When a 16-bit source element is Inactive it is treated as having the value +0.0, but if both pairs of source vector elements that correspond to a 32-bit destination element contain Inactive elements, then the destination element remains unmodified.

The resulting $SVL_S \times SVL_S$ single-precision floating-point sum of outer products is then destructively subtracted from the single-precision floating-point destination tile. This is equivalent to performing a 2-way dot product and subtract from each of the destination tile elements.

Each 32-bit container of the first source vector holds 2 consecutive summer of each row of a $SVL_S \times 2$ sub-matrix. Similarly, each 32-bit container of the second source vector olds 2 consecutive row elements of each column of a $2 \times SVL_S$ sub-matrix.

This instruction follows SME BFloat16 numerical behavi/

```
SME
(FEAT_SME)
```



BFMOPS <ZAda>.S, <Pn>/M Pm>/M, <\u03e4 .H, <\u03e4m>

1 if !HaveSME() then UNDL.INED;

```
2 integer a = UInt(Pn)
```

- 3 integer b = UInt(F ,;
- 4 integer n = UIn+ _n);
- 5 integer $m = U^{T}$ (Zm);
- 6 integer da = U1 (ZAc 7 boolean sub op =

```
7 boolean sub op =
```

Asser /ier Syr)ols

```
<ZAda> Is the ..... ZA tile ZA0-ZA3, encoded in the "ZAda" field.
```

- <Pn> Is name of the first governing scalable predicate register P0-P7, encoded in the "Pn" field.
- <Pm> Is the . me of the second governing scalable predicate register P0-P7, encoded in the "Pm" field.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
- <Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

```
1
   CheckStreamingSVEAndZAEnabled();
2
   constant integer VL = CurrentVL;
   constant integer PL = VL DIV 8;
3
Δ
   constant integer dim = VL DIV 32;
5
   bits(PL) mask1 = P[a, PL];
6
   bits(PL) mask2 = P[b, PL];
   bits(VL) operand1 = Z[n, VL];
7
   bits(VL) operand2 = Z[m, VL];
8
9
   bits(dim*dim*32) operand3 = ZAtile[da, 32, dim*dim*32];
10
   bits(dim*dim*32) result;
```

11

```
12
    for row = 0 to dim-1
13
        for col = 0 to dim-1
14
             // determine row/col predicates
15
             boolean prow_0 = (ActivePredicateElement(mask1, 2*row + 0, 16));
             boolean prow_1 = (ActivePredicateElement(mask1, 2*row + 1, 16));
boolean pcol_0 = (ActivePredicateElement(mask2, 2*col + 0, 16));
16
17
18
             boolean pcol_1 = (ActivePredicateElement(mask2, 2*col + 1, 16));
19
20
             bits(32) sum = Elem[operand3, row*dim+col, 32];
21
             if (prow_0 && pcol_0) || (prow_1 && pcol_1) then
22
                 bits(16) erow_0 = (if prow_0 then Elem[operand1, 2*row + 0, 16] else FPZero('0',
                       →16));
23
                 bits(16) erow_1 = (if prow_1 then Elem[operand1, 2*row + 1, 16] else FPZero('0',
                       →16));
24
                 bits(16) ecol_0 = (if pcol_0 then Elem[operand2, 2*col + 0, 16] else FPZero('0',
                      \rightarrow16));
25
                 bits(16) ecol_1 = (if pcol_1 then Elem[operand2 _*col +
                                                                                      16] else FPZero('0',
                       →16));
26
                  if sub_op then
27
                      boolean honor_altfp = FALSE;
                                                         // Altr nate
                                                                           ndling ig red
                      if prow_0 then erow_0 = BFNeg(erow_0 honor_al o);
if prow_1 then erow_1 = BFNeg(erov_1, honor_alt)
28
29
30
                  sum = BFDotAdd(sum, erow_0, erow_1, _ol_0, < _l_1, Fr _(]);</pre>
31
32
             Elem[result, row*dim+col, 32] = sum;
33
34
    ZAtile[da, 32, dim*dim*32] = result;
```

D1.1.22 BFVDOT

Multi-vector BFloat16 floating-point vertical dot-product by indexed element

The instruction operates on two ZA single-vector groups.

The instruction computes the vertical dot product of the corresponding BF16 elements held in the two first source vectors with pair of BF16 values held in the indexed 32-bit element of the second source vector. The single-precision dot product results are destructively added to the corresponding single-precision elements of the two ZA single-vector groups.

The BF16 pairs within the second source vector are specified using an immediate index which selects the same BF16 pair position within each 128-bit vector segment. The element index range is from 0 to 3.

The vector numbers forming the single-vector group within each half of the 7A array are selected by the sum of the vector select register and immediate offset, modulo half the number $c \Box A$ are vectors.

The VECTOR GROUP symbol VGx2 indicates that the ZA operand c sists of two Z₄ single-vector groups. The VECTOR GROUP symbol is preferred for disassembly, but optional in a mbler source code.

This instruction follows SME2 ZA-targeting BFloat16 nume al behaviors.

This instruction is unpredicated.

SME2 (FEAT_SME2)

```
13 | 12
   0
      0
          0
              0
                  0
                          0
                                                                                    Zn
1
                      1
                             1
                                 0
                                              Zm
                                                                         i2
                                                                                              0
                                                                                                  1
                                                                                                            off3
                                     1
```

BFVDOT ZA.S[<Wv>, <offs' VGx2}], { Zn1>..-<Zn2>.H }, <Zm>.H[<index>]

```
1 if !HaveSME2() then UNL FINED;
```

```
2 integer v = UInt('01 ':Rv);
```

```
3 integer n = UInt(7 '0');
4 integer m = UInt 0':Zm);
```

```
5 integer offset UInt(c .3);
```

```
6 integer index - 'Int' .);
```

Assemb' . sy. 'ols

```
<Wv> Is the 3' oit name of the vector select register W8-W11, encoded in the "Rv" field.
```

```
<offs> t<sup>1</sup> vector select offset, in the range 0 to 7, encoded in the "off3" field.
```

- <Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2
- <Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.
- <Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

<index> Is the element index, in the range 0 to 3, encoded in the "i2" field.

```
1 CheckStreamingSVEAndZAEnabled();
2 constant integer VL = CurrentVL;
3 constant integer elements = VL DIV 32;
4 integer vectors = VL DIV 8;
5 integer vstride = vectors DIV 2;
6 integer eltspersegment = 128 DIV 32;
7 bits(32) vbase = X[v, 32];
8 integer vec = (UInt(vbase) + offset) MOD vstride;
```

```
bits(VL) result;
 9
10
11
     for r = 0 to 1
12
           bits(VL) operand1a = Z[n, VL];
           bits(VL) operand1b = Z[n+1, VL];
13
           bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = ZAvector[vec, VL];
14
15
16
           for e = 0 to elements-1
                 integer segmentbase = e - (e MOD eltspersegment);
integer s = segmentbase + index;
17
18
19
                 bits(16) elt1_a = Elem[operand1a, 2 * e + r, 16];
                 bits(16) elt1_a = Elem[operand1a, 2 * e + 1, 16];
bits(16) elt1_b = Elem[operand1b, 2 * e + r, 16];
bits(16) elt2_a = Elem[operand2, 2 * s + 0, 16];
bits(16) elt2_b = Elem[operand2, 2 * s + 1, 16];
20
21
22
                 bits(32) sum = Elem[operand3, e, 32];
23
                 sum = BFDotAdd(sum, elt1_a, elt1_b, elt2_a, elt2_b, FPCR[]);
Elem[result, e, 32] = sum;
24
25
           ZAvector[vec, VL] = result;
26
27
           vec = vec + vstride;
```

D1.1.23 BMOPA

Bitwise exclusive NOR population count outer product and accumulate

This instruction works with 32-bit element ZA tile. This instruction generates an outer product of the first source $SVL_S \times 1$ vector and the second source $1 \times SVL_S$ vector. Each outer product element is obtained as population count of the bitwise XNOR result of the corresponding 32-bit elements of the first source vector and the second source vector. Each source vector is independently predicated by a corresponding governing predicate. When either source vector element is inactive the corresponding destination tile element remains unmodified. The resulting $SVL_S \times SVL_S$ product is then destructively added to the destination tile.

SME2 (FEAT_SME2)



- <ZAda> Is the name of the Z ale 2 9-ZA3, encor J in the "ZAda" field.
 - <Pn> Is the name of t¹ first governin, calable predicate register P0-P7, encoded in the "Pn" field.
 - <Pm> Is the name 1 the second governing scalable predicate register P0-P7, encoded in the "Pm" field.
 - <Zn> Is the name of e first sor e scalable vector register, encoded in the "Zn" field.
 - <Zm> , the name of the provide scalable vector register, encoded in the "Zm" field.

Open 'on

```
1
   CheckSt1 mingSVEAndZAEnabled();
    constant _ eger VL = CurrentVL;
2
   constant in er PL = VL DIV 8;
3
4
   constant integer dim = VL DIV esize;
5
   bits(PL) mask1 = P[a, PL];
   bits(PL) mask2 = P[b, PL];
6
7
   bits(VL) operand1 = Z[n, VL];
   bits(VL) operand2 = Z[m, VL];
8
   bits(dim*dim*esize) operand3 = ZAtile[da, esize, dim*dim*esize];
0
10
   bits(dim*dim*esize) result;
11
12
   for row = 0 to dim-1
13
       bits(esize) element1 = Elem[operand1, row, esize];
14
        for col = 0 to dim-1
15
            bits(esize) element2 = Elem[operand2, col, esize];
            bits(esize) element3 = Elem[operand3, row*dim + col, esize];
16
17
            if (ActivePredicateElement(mask1, row, esize) &&
18
                  ActivePredicateElement(mask2, col, esize)) then
19
                integer res = BitCount(NOT(element1 EOR element2));
20
                if sub_op then res = -res;
21
                Elem[result, row*dim + col, esize] = element3 + res;
```

22 else 23 Elem[result, row*dim + col, esize] = element3; 24 ZAtile[da, esize, dim*dim*esize] = result;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
 - The values of the data supplied in any of its operand registers when its governing predicate registers contain the same value for each execution.
 - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
 - The values of the data supplied in any of its operand register when its verning predicate registers contain the same value for each execution.
 - The values of the NZCV flags.

D1.1.24 BMOPS

Bitwise exclusive NOR population count outer product and subtract

This instruction works with 32-bit element ZA tile. This instruction generates an outer product of the first source $SVL_S \times 1$ vector and the second source $1 \times SVL_S$ vector. Each outer product element is obtained as population count of the bitwise XNOR result of the corresponding 32-bit elements of the first source vector and the second source vector. Each source vector is independently predicated by a corresponding governing predicate. When either source vector element is inactive the corresponding destination tile element remains unmodified. The resulting $SVL_S \times SVL_S$ product is then destructively subtracted from the destination tile.

SME2 (FEAT_SME2)



- <ZAda> Is the name of the Z ale 2 9-ZA3, encor J in the "ZAda" field.
 - <Pn> Is the name of t¹ first governin, calable predicate register P0-P7, encoded in the "Pn" field.
 - <Pm> Is the name 1 the second governing scalable predicate register P0-P7, encoded in the "Pm" field.
 - <Zn> Is the name of e first sor e scalable vector register, encoded in the "Zn" field.
 - <Zm> , the name of the provide scalable vector register, encoded in the "Zm" field.

Open 'on

```
1
   CheckSt1 mingSVEAndZAEnabled();
    constant _ eger VL = CurrentVL;
2
   constant in er PL = VL DIV 8;
3
4
   constant integer dim = VL DIV esize;
5
   bits(PL) mask1 = P[a, PL];
   bits(PL) mask2 = P[b, PL];
6
7
   bits(VL) operand1 = Z[n, VL];
   bits(VL) operand2 = Z[m, VL];
8
   bits(dim*dim*esize) operand3 = ZAtile[da, esize, dim*dim*esize];
0
10
   bits(dim*dim*esize) result;
11
12
   for row = 0 to dim-1
13
       bits(esize) element1 = Elem[operand1, row, esize];
14
        for col = 0 to dim-1
15
            bits(esize) element2 = Elem[operand2, col, esize];
            bits(esize) element3 = Elem[operand3, row*dim + col, esize];
16
17
            if (ActivePredicateElement(mask1, row, esize) &&
18
                  ActivePredicateElement(mask2, col, esize)) then
19
                integer res = BitCount(NOT(element1 EOR element2));
20
                if sub_op then res = -res;
21
                Elem[result, row*dim + col, esize] = element3 + res;
```

Copyright © 2022-2023 Arm Limited or its affiliates. All rights reserved. Non-confidential

22 else 23 Elem[result, row*dim + col, esize] = element3; 24 ZAtile[da, esize, dim*dim*esize] = result;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
 - The values of the data supplied in any of its operand registers when its governing predicate registers contain the same value for each execution.
 - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
 - The values of the data supplied in any of its operand register when its verning predicate registers contain the same value for each execution.
 - The values of the NZCV flags.

D1.1.25 CNTP

Set scalar to count from predicate-as-counter

Counts the number of true elements in the source predicate and places the scalar result in the destination general-purpose register.

SME2 (FEAT_SME2)



```
CNTP <Xd>, <PNn>.<T>, <v1>
```

```
1 if !HaveSME2() then UNDEFINED;
```

```
2 constant integer esize = 8 << UInt(size);</pre>
```

```
3 integer n = UInt(PNn);
4 integer d = UInt(Rd);
```

```
4 integer d = UInt(Rd);
5 constant integer width = 2 << UInt(vl);</pre>
```

Assembler Symbols

- <Xd> Is the 64-bit name of the destination gen *u*-pun se regular, e oded in the "Rd" field.
- <PNn> Is the name of the first source scalab! redicate gister, with predicate-as-counter encoding, encoded in the "PNn" field.
 - <T> Is the size specifier, encoded in "s.

size	<t></t>
00	В
01	Н
10	S
11	D

<vl> Is the vl sp. fer _ncoded i "vl":



Operation

```
CheckStreamingSVEEnabled();
1
   constant integer VL = CurrentVL;
2
   constant integer PL = VL DIV 8;
3
4
   constant integer elements = VL DIV esize;
   bits(PL) pred = P[n, PL];
5
   bits(PL*4) mask = CounterToPredicate(pred<15:0>, PL*4);
6
7
   bits(64) sum = Zeros(64);
   constant integer limit = elements * width;
8
9
10
   for e = 0 to limit-1
11
       if ActivePredicateElement(mask, e, esize) then
12
           sum = sum + 1;
13
   X[d, 64] = sum;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
 - The values of the data supplied in any of its registers.
 - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
 - The values of the data supplied in any of its registers.
 - The values of the NZCV flags.
D1.1.26 FADD

Floating-point add multi-vector to ZA array vector accumulators

The instruction operates on two or four ZA single-vector groups.

Destructively add all elements of the two or four source vectors to the corresponding elements of the two or four ZA single-vector groups. The vector numbers forming the single-vector group within each half or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction follows SME ZA-targeting floating-point behaviors.

This instruction is unpredicated.

ID_AA64SMFR0_EL1.F64F64 indicates whether the double-pr ision iant is imp mented.

It has encodings from 2 classes: Two ZA single-vectors and our ZA single-

Two ZA single-vectors (FEAT_SME2)



VGx2 FADD 7A. <T>[<Wv>. <off<T>-<Zm2>.

```
1
   if !HaveSME2() then UNP INEL
2
   if sz == '1' && !HaveSMEF64F64, then UNDEFINED;
   integer v = UInt('0
3
                             ':Rv);
   constant integer ize = 32 << UInt z);</pre>
4
   integer m = UIr _2m:'0')
integer offset UInt' .f3);
constant intege. re = 2;
5
6
```

Four 7 singly vectors (FF ___SME2

> 0 0 0 0 0 1 1 1 0 0 0

FADD ZA.<T>[<Wv>, <offs>{, VGx4}], { <Zm1>.<T>-<Zm4>.<T> }

```
if !HaveSME2() then UNDEFINED;
1
2
  if sz == '1' && !HaveSMEF64F64() then UNDEFINED;
  integer v = UInt('010':Rv);
3
4
  constant integer esize = 32 << UInt(sz);</pre>
5
  integer m = UInt(Zm:'00');
6
  integer offset = UInt(off3);
  constant integer nreg = 4;
```

Assembler Symbols

SZ	<t></t>
0	S
1	D

- <Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
- <offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.
- <Zm1> For the two ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 2.

For the four ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 4.

- <Zm4> Is the name of the fourth scalable vector register of a multi-vector rence, encoded as "Zm" times 4 plus 3.
- <Zm2> Is the name of the second scalable vector register of a multi-ctor sequence encoded as "Zm" times 2 plus 1.

```
CheckStreamingSVEAndZAEnabled();
 1
    constant integer VL = CurrentVL;
2
 3
    constant integer elements = VL DIV esize;
    integer vectors = VL DIV 8;
 4
    integer vstride = vectors DIV nreg;
5
 6
    bits(32) vbase = X[v, 32];
    integer vec = (UInt(vbase) + offsot) Mo.
 7
                                                               ide;
8
    bits(VL) result;
0
    for r = 0 to nreg-1
10
11
         bits(VL) operand1 = Z<sup>7</sup>
                                        rtor[vec,
         bits(VL) operand2 = __m+
12
                                            VL];
         for e = 0 to elemen.s-1
13
              bits(esize) lement1 = L m[operand1, e, esize];
bits(esize element2 = Ele operand2, e, esize];
Elem[res_tt, e, esize] = FPA.d_ZA(element1, element2, FPCR[]);
ector[v_, VL] = esult;
14
15
16
17
         ZAvector[, , VL] =
18
         vec = vec
                           str: .;
```

D1.1.27 FCLAMP

Multi-vector floating-point clamp to minimum/maximum number

Clamp each floating-point element in the two or four destination vectors to between the floating-point minimum value in the corresponding element of the first source vector and the floating-point maximum value in the corresponding element of the second source vector and destructively place the clamped results in the corresponding elements of the two or four destination vectors. If at least one element value contributing to a result is numeric and the other is either numeric or a quiet NaN, then the result is the numeric value.

This instruction follows SME2 floating-point numerical behaviors corresponding to instructions that place their results in one or more SVE Z vectors.

This instruction is unpredicated.

It has encodings from 2 classes: Two registers and Four registers



Assembler Symbols

<Zd1> For the two registers variant: is the name of the first destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 2.

For the four registers variant: is the name of the first destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 4.

size	<t></t>
00	RESERVED
01	Н
10	S
11	D

- <Zd4> Is the name of the fourth destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 4 plus 3.
- <Zd2> Is the name of the second destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 2 plus 1.
- $<\!\!Zn\!\!>$ $\!$ Is the name of the first source scalable vector register, encoded in the "Zn" field.
- <Zm> Is the name of the second source scalable vector register, encoded us "field.

```
CheckStreamingSVEEnabled();
1
   constant integer VL = CurrentVL;
2
   constant integer elements = VL DIV esize;
3
4
   array [0..3] of bits(VL) results;
5
6
   for r = 0 to nreg-1
7
        bits(VL) operand1 = Z[n, VL];
8
        bits(VL) operand2 = Z[m, VL];
        bits(VL) operand3 = Z[d+r, VL];
9
10
        for e = 0 to elements-1
11
             bits(esize) element1 = E<sup>1</sup> m[oper
                                                      , e, esize];
             bits(esize) element2 = Ele operal, e, esize];
bits(esize) element3 Elem1 erand, e, esize];
12
13
             Elem[results[r], e. size] = r 'inNum 'MaxNum(element1, element3, FPCR[]),
14
                  \hookrightarrowelement2,
                                     (1);
15
16
   for r = 0 to nreg-1
                           its[r];
17
        Z[d+r, VL] = re
```

D1.1.28 FCVT

Multi-vector floating-point convert from single-precision to packed half-precision

Convert to half-precision from single-precision, each element of the two source vectors, and place the results in the half-width destination elements.

This instruction follows SME2 floating-point numerical behaviors corresponding to instructions that place their results in one or more SVE Z vectors.

This instruction is unpredicated.

SME2 (FEAT_SME2)



```
1
   CheckStreaming. "Enab' d();
2
   constant integer
                             Curren+
3
   constant
                   ver e.
                                     L DIV 32;
                           rents
   bits(V
4
              resi
                    ;;
5
   bit VL) opr
                          Z[n+0, VL];
6
                rand2 [n+1, VL];
7
   bits
8
              to elements-1
   for e
9
        bits | element1 = Elem[operand1, e, 32];
        bits(32 element2 = Elem[operand2, e, 32];
bits(16) res1 = FPConvertSVE(element1, FPCR[], 16);
10
11
        bits(16) res2 = FPConvertSVE(element2, FPCR[], 16);
12
13
        Elem[result, e, 16] = res1;
14
        Elem[result, elements+e, 16] = res2;
15
16
   Z[d, VL] = result;
```

D1.1.29 FCVTN

Multi-vector floating-point convert from single-precision to interleaved half-precision

Convert to half-precision from single-precision, each element of the two source vectors, and place the two-way interleaved results in the half-width destination elements.

This instruction follows SME2 floating-point numerical behaviors corresponding to instructions that place their results in one or more SVE Z vectors.

This instruction is unpredicated.

SME2 (FEAT_SME2)



```
1
    CheckStreaming. "Enab' d();
2
    constant integer
                               Curren+
3
    constant
                    ver e.
                                       L DIV 32;
                             ents =
    bits(V
4
               resi
                      ;;
5
    bit VL) opr
                            Z[n+0, VL];
6
                 ranaz [n+1, VL];
7
    bits
8
                to elements-1
    for e
9
         bits | element1 = Elem[operand1, e, 32];
         bits(32 element2 = Elem[operand2, e, 32];
bits(16) res1 = FPConvertSVE(element1, FPCR[], 16);
10
11
         bits(16) res2 = FPConvertSVE(element2, FPCR[], 16);
12
         Elem[result, 2*e + 0, 16] = res1;
Elem[result, 2*e + 1, 16] = res2;
13
14
15
16
   Z[d, VL] = result;
```

D1.1.30 FCVTZS

Multi-vector floating-point convert to signed integer, rounding toward zero

Convert to the signed 32-bit integer nearer to zero from single-precision, each element of the two or four source vectors, and place the results in the corresponding elements of the two or four destination vectors.

This instruction follows SME2 floating-point numerical behaviors corresponding to instructions that place their results in one or more SVE Z vectors.

This instruction is unpredicated.

It has encodings from 2 classes: Two registers and Four registers

Two registers (FEAT_SME2)



Assembler Symbols

<Zd1> For the two registers variant: is the name of the first destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 2.

For the four registers variant: is the name of the first destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 4.

- <Zd4> Is the name of the fourth destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 4 plus 3.
- <Zd2> Is the name of the second destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 2 plus 1.
- <Zn1> For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.

For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

- <Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.
- <Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.

```
CheckStreamingSVEEnabled();
1
2
   constant integer VL = CurrentVL;
3
   constant integer elements = VL DIV 32;
4
   array [0..3] of bits(VL) results;
5
6
   for r = 0 to nreg-1
7
       bits(VL) operand = Z[n+r, VL];
       for e = 0 to elements-1
8
           bits(32) element = Elem[operand, e, 32];
9
10
           Elem[results[r], e, 32] = FPToFixed(element)
                                                             unsig
                                                                       FPCR
                                                                                rounding, 32);
11
   for r = 0 to nreg-1
12
13
      Z[d+r, VL] = results[r];
```

D1.1.31 FCVTZU

Multi-vector floating-point convert to unsigned integer, rounding toward zero

Convert to the unsigned 32-bit integer nearer to zero from single-precision, each element of the two or four source vectors, and place the results in the corresponding elements of the two or four destination vectors.

This instruction follows SME2 floating-point numerical behaviors corresponding to instructions that place their results in one or more SVE Z vectors.

This instruction is unpredicated.

It has encodings from 2 classes: Two registers and Four registers

Two registers (FEAT_SME2)



Assembler Symbols

<Zd1> For the two registers variant: is the name of the first destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 2.

For the four registers variant: is the name of the first destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 4.

- <Zd4> Is the name of the fourth destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 4 plus 3.
- <Zd2> Is the name of the second destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 2 plus 1.
- <Zn1> For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.

For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

- <Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.
- <Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.

```
CheckStreamingSVEEnabled();
1
2
   constant integer VL = CurrentVL;
3
   constant integer elements = VL DIV 32;
4
   array [0..3] of bits(VL) results;
5
6
   for r = 0 to nreg-1
7
       bits(VL) operand = Z[n+r, VL];
       for e = 0 to elements-1
8
           bits(32) element = Elem[operand, e, 32];
9
10
           Elem[results[r], e, 32] = FPToFixed(element)
                                                             unsig
                                                                       FPCR
                                                                                rounding, 32);
11
   for r = 0 to nreg-1
12
13
      Z[d+r, VL] = results[r];
```

D1.1.32 FDOT (multiple and indexed vector)

Multi-vector half-precision floating-point dot-product by indexed element

The instruction operates on two or four ZA single-vector groups.

The instruction computes the fused sum-of-products of a pair of half-precision floating-point values held in the corresponding 32-bit elements of the two or four first source vectors and the indexed 32-bit element of the second source vector, without intermediate rounding. The single-precision sum-of-products results are destructively added to the corresponding single-precision elements of the two or four ZA single-vector groups.

The half-precision floating-point pairs within the second source vector are specified using an immediate index which selects the same pair position within each 128-bit vector segment. The element index range is from 0 to 3. The vector numbers forming the single-vector group within each half or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo he warter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA ope. d consists of to or four ZA single-vector groups respectively. The VECTOR GROUP symbol is preferred f disass bly, but o onal in assembler source code.

This instruction follows SME ZA-targeting floating-poir. ehavior

This instruction is unpredicated.

It has encodings from 2 classes: Two ZA single ectors nd Foun gle-vectors

Two ZA single-vectors (FEAT_SME2)

14 13 12 11 10 15 1 0 0 0 0 0 1 0 Zm 0 Rv i2 Zn 0 0 1 off3 1 1 1

```
FDOT
       ZA.S[<Wv>
                    offs>{, VGx.
                                     { <Zn1>.H-<Zn2>.H }, <Zm>.H[<index>]
```

```
if !HaveSME2() nen UND' INED;
1
2
  integer v = UI. ('010 Rv);
3
  integer n = UInt
                        J');
4
                ⊤nt('
                        Zm);
  integer
5
  intege
           offse
                  = UIn.
                          hff
```

```
inte _ index = UInt(i.
6
                           2:
```

```
7
        ant in*
  COL
                      rea =
```

```
Four ZA rgle-vectors
(FEAT_SM.
```

1 0 0 0 0 0 1 0 1 0 1 Zm Rv i2 Zn 0 0 off3 1 1

FDOT ZA.S[<Wv>, <offs>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H[<index>]

```
if !HaveSME2() then UNDEFINED;
1
```

```
2
  integer v = UInt('010':Rv);
```

```
3
  integer n = UInt(Zn:'00');
```

```
integer m = UInt('0':Zm);
4
```

```
integer offset = UInt(off3);
5
```

```
integer index = UInt(i2);
6
  constant integer nreg = 4;
7
```

Assembler Symbols

- <Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
- <offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.
- <Zn1> For the two ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.

For the four ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

- <Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.
- <Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.
- <Zm> Is the name of the second source scalable vector register Z0-Z['], encoded in "Zm" field.

<index> Is the element index, in the range 0 to 3, encoded in the "2" field

```
CheckStreamingSVEAndZAEnabled();
 1
 2
    constant integer VL = CurrentVL;
    constant integer elements = VL DIV 32;
 3
 4
    integer vectors = VL DIV 8;
     integer vstride = vectors DIV nreg;
 5
    integer eltspersegment = 128 DIV 32;
 6
 7
    bits(32) vbase = X[v, 32];
 8
    integer vec = (UInt(vbase) + offs
                                                      MOD
                                                                ride;
 9
    bits(VL) result;
10
    for r = 0 to nreg-1
11
                                     , nh
12
          bits(VL) operand1 =
                                              VL];
          bits(VL) operand2 = Z[m,
13
14
          bits(VL) operanc' = ZAvector
                                                  ec, VL];
15
          for e = 0 to \epsilon .ments-1
               bits(16) .ttl_a = Elem[opera.dl, 2 * e + 0, 16];
bits(1 elt1_b Elem[opera.dl, 2 * e + 1, 16];
integel egme base = e - (e MOD eltspersegment);
integer egmentbase + index;
16
17
18
19
               6) el a = F m[operand2, 2 * s + 0, 16];

bits j) eltz = slem[operand2, 2 * s + 1, 16];

bits 2) sum = lem[operand3, e, 32];
20
21
22
23
                             +Add_ZA(sum, elt1_a, elt1_b, elt2_a, elt2_b, FPCR[]);
               sum
24
               F' ... [result, e, 32] = sum;
                  or[vec, VL] = result;
25
          ZΑv
26
                    >c + vstride;
          vec
```

D1.1.33 FDOT (multiple and single vector)

Multi-vector half-precision floating-point dot-product by vector

The instruction operates on two or four ZA single-vector groups.

The instruction computes the fused sum-of-products of a pair of half-precision floating-point values held in the corresponding 32-bit elements of the two or four first source vectors and the second source vector, without intermediate rounding. The single-precision sum-of-products results are destructively added to the corresponding single-precision elements of the two or four ZA single-vector groups. The vector numbers forming the single-vector group within each half or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The VECTOR GROUP symbol is preferred for disasser rt optional in assembler source code.

This instruction follows SME ZA-targeting floating-point behaviors.

This instruction is unpredicated.

It has encodings from 2 classes: Two ZA single-vectors ar Four ZA ingle-

Two ZA single-vectors (FEAT_SME2)

> 13 10 1 0 0 0 0 0 1 0 0 1 0 7m 0 Rv 1 0 0 7n 0 0 off3

√Gx2}1 FDOT ZA.S[<Wv>, <offs> 7n1 <Zn2>.H }, <Zm>.H

- 1 if !HaveSME2() then UND
- integer v = UInt('010':Rv); 2
- integer n = UInt(Zp) 3
- 4 integer m = UInt() :Zm);
- 5 integer offset JInt (of 3);
- constant inte 6 nreq



1

Zm

0 Rv 0

0

Zn

0 0 off3

ZA.S[<Wv>, <offs>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H FDOT

0

0

0 1

- if !HaveSME2() then UNDEFINED; 1
- 2 integer v = UInt('010':Rv);
- 3 integer n = UInt(Zn);
- 4 integer m = UInt('0':Zm);
- integer offset = UInt(off3); 6 constant integer nreg = 4;

Assembler Symbols

- $\langle Wv \rangle$ Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
- <offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.
- Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn". $\langle Zn1 \rangle$
- <Zn4>Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn"

plus 3 modulo 32.

- <Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" plus 1 modulo 32.
- <Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

```
1
    CheckStreamingSVEAndZAEnabled();
2
   constant integer VL = CurrentVL;
3
    constant integer elements = VL DIV 32;
   integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
 4
5
6
   bits(32) vbase = X[v, 32];
7
    integer vec = (UInt(vbase) + offset) MOD vstride;
8
    bits(VL) result;
9
10
    for r = 0 to nreg-1
11
        bits(VL) operand1 = Z[(n+r) MOD 32, VL];
12
        bits(VL) operand2 = Z[m, VL];
13
        bits(VL) operand3 = ZAvector[vec, VL];
14
        for e = 0 to elements-1
                                                            , 16];
15
             bits(16) elt1_a = Elem[operand1, 2 * e
             bits(16) elt1_b = Elem[operand1, 2 * e +
16
                                                               16
            bits(16) elt2_a = Elem[operand2, 2 * e +
bits(16) elt2_b = Elem[operand2, 2 * e +
                                                           0,
17
                                                                 ....
18
                                                            1,
19
             bits(32) sum = Elem[operand3, e
                                                  52];
20
             sum = FPDotAdd_ZA(sum, elt1_a elt1_b,
                                                          lt2_a, e. 2_b, FPCR[]);
21
             Elem[result, e, 32] = sum;
22
        ZAvector[vec, VL] = result;
23
        vec = vec + vstride;
```

D1.1.34 FDOT (multiple vectors)

Multi-vector half-precision floating-point dot-product

The instruction operates on two or four ZA single-vector groups.

The instruction computes the fused sum-of-products of a pair of half-precision floating-point values held in the corresponding 32-bit elements of the two or four first and second source vectors, without intermediate rounding. The single-precision sum-of-products results are destructively added to the corresponding single-precision elements of the two or four ZA single-vector groups. The vector numbers forming the single-vector group within each half or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The VECTOR GROUP symbol is preferred for disasser to prove optional in assembler source code.

This instruction follows SME ZA-targeting floating-point behaviors.

This instruction is unpredicated.

It has encodings from 2 classes: Two ZA single-vectors a Four ZA ingle-ve

Two ZA single-vectors (FEAT_SME2)

16 13 10 Zm 0 0 0 0 1 1 0 1 0 Rv 1 0 0 7n 0 0 0 off3

FDOT ZA.S[<Wv>, <offs>/ VGx2}], <Zn1> -<Zn2>.H }, { <Zm1>.H-<Zm2>.H }

```
1 if !HaveSME2() then UNP INE
```

```
2 integer v = UInt('010':Rv);
```

- 3 integer n = UInt(Zr 0');
- 4 integer m = UInt (.: '0');
- 5 integer offset JInt(of 3);
- 6 constant inte, nreg 2;



0 1 1 Zm 1 0 Rv 0 0 Zn 0 off3 0 0 1 0

FDOT ZA.S[<Wv>, <offs>{, VGx4}], { <Zn1>.H-<Zn4>.H }, { <Zm1>.H-<Zm4>.H }

```
1 if !HaveSME2() then UNDEFINED;
```

```
2 integer v = UInt('010':Rv);
```

```
3 integer n = UInt(Zn:'00');
```

```
4 integer m = UInt(Zm:'00');
```

```
5 integer offset = UInt(off3);
```

```
6 constant integer nreg = 4;
```

Assembler Symbols

- <Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
- <offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.
- <Zn1> For the two ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.

For the four ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

- <Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.
- <Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.
- <Zm1> For the two ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 2.

For the four ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 4.

- <Zm4> Is the name of the fourth scalable vector register of a multi-vector sence, encoded as "Zm" times 4 plus 3.
- <Zm2> Is the name of the second scalable vector register of a multi-ctor sequence encoded as "Zm" times 2 plus 1.

```
CheckStreamingSVEAndZAEnabled();
 1
 2
     constant integer VL = CurrentVL;
     constant integer elements = VL DIV 32;
 3
 4
     integer vectors = VL DIV 8;
 5
     integer vstride = vectors DIV nreg;
     bits(32) vbase = X[v, 32];
 6
     integer vec = (UInt(vbase) + offsot) Mc
 7
                                                                        ide;
 8
     bits(VL) result;
 9
10
     for r = 0 to nreg-1
           bits(VL) operand1 = Z' r, VL];
bits(VL) operand2 = _m, VL];
bits(VL) operand3 = ZAvect [vec, VL];
11
12
13
           for e = 0 to ele ents-1
14
15
                 bits(16)
                                 .1_a = Elem[ope nd1, 2 * e + 0, 16];
                bits(16) .1t1_b = Elem[opera.dl, 2 * e + 0, 16];
bits(16) .1t1_b = Elem[opera.dl, 2 * e + 1, 16];
bits(1 elt2_r Elem[opera.d2, 2 * e + 0, 16];
bits(1 elt2_r Elem[opera.d2, 2 * e + 1, 16];
bits(32) = Elem[opera.d3, e, 32];
16
17
18
19
                          FPDot 'd_ZA(* d, elt1_a, elt1_b, elt2_a, elt2_b, FPCR[]);
20
21
             Elem | sult,
Avector ec, VL]
                                           _ = sum;
                                      - .esult;
22
23
                = v
                                -ride;
             ٦C
```

D1.1.35 FMAX (multiple and single vector)

Multi-vector floating-point maximum by vector

Determine the maximum of floating-point elements of the second source vector and the corresponding floating-point elements of the two or four first source vectors and destructively place the results in the corresponding elements of the two or four first source vectors. If either element value is NaN then the result is NaN.

This instruction follows SME2 floating-point numerical behaviors corresponding to instructions that place their results in one or more SVE Z vectors.

This instruction is unpredicated.

It has encodings from 2 classes: Two registers and Four registers

	Two registers (FEAT_SME2)
	1 1 0 0 0 0 0 1 size 1 0 Zm 1 0 1 0 0 0 1 0 0 Zdn 0
	FMAX { <zdn1>.<t>-<zdn2>.<t> }, { <zdn1>.< -< .n2>.<t` <zm="" },="">.<t></t></t`></zdn1></t></zdn2></t></zdn1>
1	if !HaveSME2() then UNDEFINED;
23	IT SIZE == 00° THEN UNDEFINED;
4	<pre>integer dn = UInt(Zdn:'0');</pre>
5	<pre>integer m = UInt('0':Zm);</pre>
6	<pre>constant integer nreg = 2;</pre>
	Four registers
	(FEAT SME2)
	<u>31</u> <u>24</u> <u>23</u> <u>22</u> <u>28</u> <u>19</u> <u>16</u> <u>15</u> <u>10</u> <u>9</u> <u>6</u> <u>5</u> <u>4</u> <u>2</u> <u>1</u> <u>8</u> <u>7</u> <u>1</u> <u>8</u> <u>15</u> <u>10</u> <u>9</u> <u>6</u> <u>5</u> <u>4</u> <u>2</u> <u>1</u> <u>8</u> <u>15</u> <u>10</u> <u>15</u> <u>15</u> <u>10</u> <u>10</u> <u>15</u> <u>15</u> <u>10</u> <u>15</u> <u>15</u>
	FMAX { Sn1>. >- <zdn' .<t=""> }, { <zdn1>.<t> -<zdn4>.<t> }, <zm>.<t></t></zm></t></zdn4></t></zdn1></zdn'>
1	if !' JeSME2 (then UN. INED;
2	if ze == ' ' then UNDEFINED;
3	const t, ; .eger = 8 << UInt(size);
4	<pre>integel (= UInt(Zdn:'00');</pre>
5	integer h. Olnt('0':Zm);
0	constant In yer meg - 4;
	Assembler Symbols

<Zdn1> For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2.

For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4.

size	<t></t>
00	RESERVED
01	Н
10	S
11	D

- <Zdn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4 plus 3.
- <Zdn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2 plus 1.
 - <Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

```
CheckStreamingSVEEnabled();
1
   constant integer VL = CurrentVL;
2
3
   constant integer elements = VL DIV esize;
4
   array [0..3] of bits(VL) results;
5
6
   for r = 0 to nreg-1
7
        bits(VL) operand1 = Z[dn+r, VL];
8
        bits(VL) operand2 = Z[m, VL];
9
        for e = 0 to elements-1
10
            bits(esize) element1 = Elem[operand1, e, esize];
11
            bits(esize) element2 = Elem[operand2, e, esi7
            Elem[results[r], e, esize] = FPMax(element<sup>1</sup>
12
                                                           element2
                                                                         CRſ
13
14
   for r = 0 to nreg-1
    Z[dn+r, VL] = results[r];
15
```

D1.1.36 FMAX (multiple vectors)

Multi-vector floating-point maximum

Determine the maximum of floating-point elements of the two or four second source vectors and the corresponding floating-point elements of the two or four first source vectors and destructively place the results in the corresponding elements of the two or four first source vectors. If either element value is NaN then the result is NaN.

This instruction follows SME2 floating-point numerical behaviors corresponding to instructions that place their results in one or more SVE Z vectors.

This instruction is unpredicated.

It has encodings from 2 classes: Two registers and Four registers

```
Two registers
   (FEAT_SME2)
                                                                     0
                                                                                0
                                                                                   0
               1
                  0
                    0
                       0
                          0
                             0
                                   size
                                                          1
                                                            0
                                                                                            Zdn
            1
                                1
                                        1
                                                       0
                                                                             1
            { <Zdn1>.<T>-<Zdn2>.<T> }, { <Zdn1>.
   FMAX
                                                                                 \{ < 7m1 > . < T > - < 7m2 > .
        \hookrightarrow
1
   if !HaveSME2() then UNDEFINED;
   if size == '00' then UNDEFINED;
2
3
   constant integer esize = 8 << UInt(size)
4
   integer dn = UInt(Zdn:'0');
5
   integer m = UInt(Zm:'0');
6
   constant integer nreg = 2;
   Four registers
   (FEAT_SME2)
                                                                                0
                                                                                   0
                  0
                          0
                             0
                                              Zm
                                                       0
                                                         1
                                                            0
                                                               1
                                                                  1
                                                                     1
                                                                        0
                                                                          0
                                                                             1
                                                                                           Zdn
               1
                                1
                                   size
                                        1
                                                    0
   FMAX
                                               { <Zdn1>.<T>-<Zdn4>.<T> }, { <Zm1>.<T>-<Zm4>.<T>
                                      \langle T \rangle
1
   if
           ve SM
                            DEFINED;
                '00' then UNDEFINED;
2
   if siz
3
   constanı
              nteger esize = 8 << UInt(size);</pre>
4
   integer dh.
                  UInt(Zdn:'00');
                nt (Zm: '00');
5
   integer m =
   constant integer nreg = 4;
6
   Assembler Symbols
```

<Zdn1> For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2.

For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4.

size	<t></t>
00	RESERVED
01	Н
10	S
11	D

- <Zdn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4 plus 3.
- <Zdn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2 plus 1.
- <Zm1> For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 2.

For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 4.

- <Zm4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zm" times 4 plus 3.
- <Zm2> Is the name of the second scalable vector register of a multi-v for sequence encoded as "Zm" times 2 plus 1.

```
1
   CheckStreamingSVEEnabled();
2
   constant integer VL = CurrentVL;
   constant integer elements = VL DIV esize;
3
4
   array [0..3] of bits(VL) results;
5
6
   for r = 0 to nreg-1
       bits(VL) operand1 = Z[dn+r, VL];
7
8
       bits(VL) operand2 = Z[m+r, VL];
       for e = 0 to elements-1
9
10
           bits(esize) element1 = Ele.
                                         peran
                                                   e, esize];
11
           bits(esize) element? Elem[ rand2, ,
                                                     esize];
12
           Elem[results[r], e
                                esize] = Fi
                                             x(element1, element2, FPCR[]);
13
14
   for r = 0 to nreg-1
15
       Z[dn+r, VL] = re ults[r];
```

D1.1.37 FMAXNM (multiple and single vector)

Multi-vector floating-point maximum number by vector

Determine the maximum number value of floating-point elements of the second source vector and the corresponding floating-point elements of the two or four first source vectors and destructively place the results in the corresponding elements of the two or four first source vectors. If one element value is numeric and the other is a quiet NaN, then the result is the numeric value.

This instruction follows SME2 floating-point numerical behaviors corresponding to instructions that place their results in one or more SVE Z vectors.

This instruction is unpredicated.

It has encodings from 2 classes: Two registers and Four registers



<Zdn1> For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2.

For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4.

size	<t></t>
00	RESERVED
01	Н
10	S
11	D

- <Zdn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4 plus 3.
- <Zdn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2 plus 1.
 - <Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

```
CheckStreamingSVEEnabled();
1
2
   constant integer VL = CurrentVL;
   constant integer elements = VL DIV esize;
3
4
   array [0..3] of bits(VL) results;
5
6
   for r = 0 to nreg-1
7
       bits(VL) operand1 = Z[dn+r, VL];
8
       bits(VL) operand2 = Z[m, VL];
        for e = 0 to elements-1
9
10
           bits(esize) element1 = Elem[operand1, e, es
                                                           e];
                                                        _ze];
11
           bits(esize) element2 = Elem[operand2, e,
           Elem[results[r], e, esize] = FPMaxNum(e)
                                                                ement2,
                                                                         . CR[]);
12
                                                      nent1,
13
14
   for r = 0 to nreg-1
15
       Z[dn+r, VL] = results[r];
```

D1.1.38 FMAXNM (multiple vectors)

Multi-vector floating-point maximum number

Determine the maximum number value of floating-point elements of the two or four second source vectors and the corresponding floating-point elements of the two or four first source vectors and destructively place the results in the corresponding elements of the two or four first source vectors. If one element value is numeric and the other is a quiet NaN, then the result is the numeric value.

This instruction follows SME2 floating-point numerical behaviors corresponding to instructions that place their results in one or more SVE Z vectors.

This instruction is unpredicated.

It has encodings from 2 classes: Two registers and Four registers



Assembler Symbols

<Zdn1> For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2.

For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4.

size	<t></t>
00	RESERVED
01	Н
10	S
11	D

- <Zdn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4 plus 3.
- <Zdn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2 plus 1.
- <Zm1> For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 2.

For the four registers variant: is the name of the first scalable version register or multi-vector sequence, encoded as "Zm" times 4.

- <Zm4> Is the name of the fourth scalable vector register of a _alti-vect_sequence encoded as "Zm" times 4 plus 3.
- <Zm2> Is the name of the second scalable vector registe. f a mv⁻¹-vector sequence, encoded as "Zm" times 2 plus 1.

```
1
   CheckStreamingSVEEnabled();
2
   constant integer VL = CurrentVL;
3
   constant integer elements = VL D1
                                             size;
4
   array [0..3] of bits(VL) resu
                                       3;
5
6
   for r = 0 to nreg-1
7
        bits(VL) operand1
                                , dn
                                        VL];
        bits(VL) operand2 = Z[m+r, ];
8
9
        for e = 0 to el/ ents-1
            bits(esize element1 = Ele operand1, e, esize];
bits(esize) element2 = Elemioperand2, e, esize];
10
11
12
             Elem[: ults[r' e, esize] = FPMaxNum(element1, element2, FPCR[]);
13
14
    for r = 0 \pm 0 nreg
        Z[dr
15
                         ı ults
```

D1.1.39 FMIN (multiple and single vector)

Multi-vector floating-point minimum by vector

Determine the mininum of floating-point elements of the second source vector and the corresponding floating-point elements of the two or four first source vectors and destructively place the results in the corresponding elements of the two or four first source vectors. If either element value is NaN then the result is NaN.

This instruction follows SME2 floating-point numerical behaviors corresponding to instructions that place their results in one or more SVE Z vectors.

This instruction is unpredicated.

It has encodings from 2 classes: Two registers and Four registers

	Two registers (FEAT_SME2)
	31 24 23 22 21 20 19 16 15 10 9 6 4 1 0
	1 1 0 0 0 0 0 1 size 1 0 Zm 1 0 1 0 0 0 1 0 0 Zdn 1
	FMIN { <zdn1>.<t> -<zdn2>.<t> }, { <zdn1>.<</zdn1></t></zdn2></t></zdn1>
1	if HaveSME2() then INDEFINED.
2	if size == '00' then UNDEFINED:
3	constant integer esize = 8 << UInt(s; -);
4	<pre>integer dn = UInt(Zdn:'0');</pre>
5	<pre>integer m = UInt('0':Zm);</pre>
6	constant integer nreg = 2;
	Four registers (FEAT_SME2)
	31 24 23 22 20 19 16 15 10 9 6 5 4 2 1 0 1 1 0 0 0 1 size 1 Zm 1 0 1 0 0 0 0 Zdn 0 1
	FMIN / dn1>. >- <zdn <="" td=""> .<t> }, { <zdn1>.<t> -<zdn4>.<t> }, <zm>.<t></t></zm></t></zdn4></t></zdn1></t></zdn>
1	if V veSME2 (then UN (NED:
2	if ze == ' ' then UNDEFINED:
3	const t ; .eger t e = 8 << UInt(size);
4	<pre>intege. 1 = UInt(Zdn:'00');</pre>
5	<pre>integer n UInt('0':Zm);</pre>
6	constant i. ger nreg = 4;
	Assembler Symbols

<Zdn1> For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2.

For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4.

size	<t></t>
00	RESERVED
01	Н
10	S
11	D

- <Zdn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4 plus 3.
- <Zdn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2 plus 1.
 - <Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

```
CheckStreamingSVEEnabled();
1
   constant integer VL = CurrentVL;
2
3
   constant integer elements = VL DIV esize;
4
   array [0..3] of bits(VL) results;
5
6
   for r = 0 to nreg-1
        bits(VL) operand1 = Z[dn+r, VL];
7
8
        bits(VL) operand2 = Z[m, VL];
9
        for e = 0 to elements-1
10
            bits(esize) element1 = Elem[operand1, e, esize];
11
            bits(esize) element2 = Elem[operand2, e, esi7
            Elem[results[r], e, esize] = FPMin(element<sup>1</sup>
12
                                                           element2
                                                                         CRſ
13
14
   for r = 0 to nreg-1
    Z[dn+r, VL] = results[r];
15
```

D1.1.40 FMIN (multiple vectors)

Multi-vector floating-point minimum

Determine the mininum of floating-point elements of the two or four second source vectors and the corresponding floating-point elements of the two or four first source vectors and destructively place the results in the corresponding elements of the two or four first source vectors. If either element value is NaN then the result is NaN.

This instruction follows SME2 floating-point numerical behaviors corresponding to instructions that place their results in one or more SVE Z vectors.

This instruction is unpredicated.

It has encodings from 2 classes: Two registers and Four registers

```
Two registers
   (FEAT_SME2)
                                                                   0
                                                                              0
                                                                                 0
               1
                 0
                    0
                       0
                         0
                            0
                                  size
                                              Zm
                                                        1
                                                           0
                                                                                          Zdn
            1
                               1
                                       1
                                                      0
                                                                            1
         { <Zdn1>.<T>-<Zdn2>.<T> }, { <Zdn1>.
   FMTN
                                                                               { <7m1>.<T>-<7m2>.<T
        \hookrightarrow}
1
   if !HaveSME2() then UNDEFINED;
   if size == '00' then UNDEFINED;
2
3
   constant integer esize = 8 << UInt(size)
4
   integer dn = UInt(Zdn:'0');
5
   integer m = UInt(Zm:'0');
6
  constant integer nreg = 2;
   Four registers
   (FEAT_SME2)
                                                           0
                                                                           1
                                                                              0
                                                                                 0
                 0
                          0
                            0
                                  size
                                            Zm
                                                     0
                                                        1
                                                              1
                                                                1
                                                                   1
                                                                      0
                                                                         0
                                                                                         Zdn
               1
                               1
                                       1
                                                   0
   FMIN
                                              { <Zdn1>.<T>-<Zdn4>.<T> }, { <Zm1>.<T>-<Zm4>.<T>
                                     \langle T \rangle
1
   if
           ve SM
                           DEFINED;
               '00' then UNDEFINED;
2
   if siz
3
   constanı
              nteger esize = 8 << UInt(size);</pre>
4
   integer dh.
                 UInt(Zdn:'00');
                nt (Zm: '00');
5
   integer m =
   constant integer nreg = 4;
6
   Assembler Symbols
```

<Zdn1> For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2.

For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4.

size	<t></t>
00	RESERVED
01	Н
10	S
11	D

- <Zdn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4 plus 3.
- <Zdn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2 plus 1.
- <Zm1> For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 2.

For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 4.

- <Zm4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zm" times 4 plus 3.
- <Zm2> Is the name of the second scalable vector register of a multi-v for sequence encoded as "Zm" times 2 plus 1.

```
CheckStreamingSVEEnabled();
1
2
   constant integer VL = CurrentVL;
   constant integer elements = VL DIV esize;
3
   array [0..3] of bits(VL) results;
4
5
6
   for r = 0 to nreg-1
        bits(VL) operand1 = Z[dn+r, VL];
7
8
        bits(VL) operand2 = Z[m+r, VL];
        for e = 0 to elements-1
9
             bits(esize) element1 = Ele. perant e, esize];
bits(esize) element2 Elem[trand2, trand2, esize];
10
11
12
             Elem[results[r], e
                                   esize] = Fi n(element1, element2, FPCR[]);
13
14
   for r = 0 to nreg-1
15
        Z[dn+r, VL] = re ults[r];
```

D1.1.41 FMINNM (multiple and single vector)

Multi-vector floating-point minimum number by vector

Determine the minimum number value of floating-point elements of the second source vector and the corresponding floating-point elements of the two or four first source vectors and destructively place the results in the corresponding elements of the two or four first source vectors. If one element value is numeric and the other is a quiet NaN, then the result is the numeric value.

This instruction follows SME2 floating-point numerical behaviors corresponding to instructions that place their results in one or more SVE Z vectors.

This instruction is unpredicated.

It has encodings from 2 classes: Two registers and Four registers



<Zdn1> For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2.

For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4.

size	<t></t>
00	RESERVED
01	Н
10	S
11	D

- <Zdn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4 plus 3.
- <Zdn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2 plus 1.
 - <Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

```
CheckStreamingSVEEnabled();
1
2
   constant integer VL = CurrentVL;
   constant integer elements = VL DIV esize;
3
4
   array [0..3] of bits(VL) results;
5
6
   for r = 0 to nreg-1
7
       bits(VL) operand1 = Z[dn+r, VL];
8
       bits(VL) operand2 = Z[m, VL];
       for e = 0 to elements-1
9
10
           bits(esize) element1 = Elem[operand1, e, es
                                                           e];
                                                        _ze];
11
           bits(esize) element2 = Elem[operand2, e,
           Elem[results[r], e, esize] = FPMinNum(e
                                                               ement2,
                                                                        . CR[]);
12
                                                      nent1,
13
14
   for r = 0 to nreg-1
15
       Z[dn+r, VL] = results[r];
```

D1.1.42 FMINNM (multiple vectors)

Multi-vector floating-point minimum number

Determine the minimum number value of floating-point elements of the two or four second source vectors and the corresponding floating-point elements of the two or four first source vectors and destructively place the results in the corresponding elements of the two or four first source vectors. If one element value is numeric and the other is a quiet NaN, then the result is the numeric value.

This instruction follows SME2 floating-point numerical behaviors corresponding to instructions that place their results in one or more SVE Z vectors.

This instruction is unpredicated.

It has encodings from 2 classes: Two registers and Four registers



Assembler Symbols

<Zdn1> For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2.

For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4.

size	<t></t>
00	RESERVED
01	Н
10	S
11	D

- <Zdn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4 plus 3.
- <Zdn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2 plus 1.
- <Zm1> For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 2.

For the four registers variant: is the name of the first scalable version register or multi-vector sequence, encoded as "Zm" times 4.

- <Zm4> Is the name of the fourth scalable vector register of a _alti-vect_sequence encoded as "Zm" times 4 plus 3.
- <Zm2> Is the name of the second scalable vector registe. f a mv⁻¹-vector sequence, encoded as "Zm" times 2 plus 1.

```
1
   CheckStreamingSVEEnabled();
2
   constant integer VL = CurrentVL;
3
   constant integer elements = VL D1
                                             size;
4
   array [0..3] of bits(VL) resu
                                       3;
5
6
   for r = 0 to nreg-1
7
        bits(VL) operand1
                                , dn
                                        VL];
        bits(VL) operand2 = Z[m+r, ];
8
9
        for e = 0 to el/ ents-1
            bits(esize element1 = Ele operand1, e, esize];
bits(esize) element2 = Elemioperand2, e, esize];
10
11
12
             Elem[: ults[r' e, esize] = FPMinNum(element1, element2, FPCR[]);
13
14
    for r = 0 \pm 0 nreg
        Z[dr
15
                        1 ults
```

D1.1.43 FMLA (multiple and indexed vector)

Multi-vector floating-point fused multiply-add by indexed element

The instruction operates on two or four ZA single-vector groups.

Multiply the indexed element of the second source vector by the corresponding floating-point elements of the two or four first source vectors and destructively add without intermediate rounding to the corresponding elements of the two or four ZA single-vector groups.

The elements within the second source vector are specified using an immediate element index which selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 1 to 2 bits depending on the size of the element. The vector numbers forming the single-vector group within each half or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA oper disconsists of the vector groups respectively. The VECTOR GROUP symbol is preferred for disconsists of the vector on all in assembler source code.

This instruction follows SME ZA-targeting floating-point be aviors.

This instruction is unpredicated.

ID_AA64SMFR0_EL1.F64F64 indicates whether _____uble-____cision y___ant is implemented.

It has encodings from 4 classes: Two ZA sing -vectors f single is in elements, Two ZA single-vectors of double precision elements, Four ZA single ctors of close classes and Four ZA single-vectors of double precision elements

Two ZA single-vectors of single p cisio. \lemen. (FEAT_SME2)



```
1 if !Har oME2 then DEF LD;
2 inter r v = U it ('010 ;;
3 cor ant int /or esize = 32;
4 integ n orne, J');
5 intege. = UInt ('0':Zm);
6 integen Cort Wirt (652);
```

```
6 integer ( set = UInt(off3);
```

```
7 integer inc = UInt(i2);
8 boolean sub_p = FALSE;
```

```
8 boolean sub_p = FALSE;
9 constant integer nreg = 2;
```

```
Two ZA single-vectors of double precision elements (FEAT_SME_F64F64)
```



```
FMLA ZA.D[<Wv>, <offs>{, VGx2}], { <Zn1>.D-<Zn2>.D }, <Zm>.D[<index>]
```

```
1 if !(HaveSME2() && HaveSMEF64F64()) then UNDEFINED;
2 integer v = UInt('010':Rv);
```

```
3 constant integer esize = 64;
```

```
Chapter D1. SME instructions
D1.1. SME and SME2 data-processing instructions
```

```
4 integer n = UInt(Zn:'0');
5 integer m = UInt('0':Zm);
6 integer offset = UInt(off3);
7 integer index = UInt(i1);
```

8 boolean sub_op = FALSE; 9 constant integer nreg = 2;

Four ZA single-vectors of single precision elements



Assembler Symbols

- <Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
- <offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.
- <Zn1> For the two ZA single-vectors of double precision elements and two ZA single-vectors of single precision elements variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.

For the four ZA single-vectors of double precision elements and four ZA single-vectors of single precision elements variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

- <Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.
- <Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as

"Zn" times 2 plus 1.

- <Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.
- <index> For the four ZA single-vectors of single precision elements and two ZA single-vectors of single precision elements variant: is the element index, in the range 0 to 3, encoded in the "i2" field.

For the four ZA single-vectors of double precision elements and two ZA single-vectors of double precision elements variant: is the element index, in the range 0 to 1, encoded in the "i1" field.

```
CheckStreamingSVEAndZAEnabled();
 1
    constant integer VL = CurrentVL;
 2
 3
    constant integer elements = VL DIV esize;
 4
    integer vectors = VL DIV 8;
    integer vstride = vectors DIV nreg;
 5
    integer eltspersegment = 128 DIV esize;
 6
 7
    bits(32) vbase = X[v, 32];
 8
    integer vec = (UInt(vbase) + offset) MOD vstride;
 9
    bits(VL) result;
10
11
    for r = 0 to nreg-1
12
         bits(VL) operand1 = Z[n+r, VL];
         bits(VL) operand2 = Z[m, VL];
13
14
         bits(VL) operand3 = ZAvector[vec,
                                                        1;
15
          for e = 0 to elements-1
16
              bits(esize) element1 = Elem[op
                                                         nd'
              integer segmentbase = e 'e MOL
integer s = segmentbase + 'ex;
17
                                                            .spersegment);
               integer s = segmentbase +
18
         bits(esize) element2 Elem[\_randz_3, e
bits(esize) element = Elem[op\_nd3, , e
if sub_op then e' u '1 = FPNeg(\_ment1);
Elem[result, e, ssize = FPMulAdd_.A(eleme
ZAvector[vec, VL' = result;
19
                                                                   esize];
20
                                                                   esize];
21
22
                                             FPMulAdd_.A(element3, element1, element2, FPCR[]);
23
         vec = vec + vs<sup>+</sup> _de;
24
```

D1.1.44 FMLA (multiple and single vector)

Multi-vector floating-point fused multiply-add by vector

The instruction operates on two or four ZA single-vector groups.

Multiply the corresponding floating-point elements of the two or four first source vector with corresponding elements of the second source vector and destructively add without intermediate rounding to the corresponding elements of the two or four ZA single-vector groups. The vector numbers forming the single-vector group within each half or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction follows SME ZA-targeting floating-point behaviors.

This instruction is unpredicated.

ID AA64SMFR0 EL1.F64F64 indicates whether the double recision var. t is ir remented.

It has encodings from 2 classes: Two ZA single-vectors (Four 7) single-vec ors

Two ZA single-vectors (FEAT_SME2)



FMLA ZA. <T>[<Wv>,VGx2}], [Zn1>.<T>-<Zn2>.<T> }, <Zm>.<T>

```
if !HaveSME2() then "NDEFINED;
1
2
  if sz == '1' && !H _SMEF64F64()
                                      uNDEFINED;
3
  integer v = UInt J10':Rv);
  constant integ esize
4
                            32 << UInt(sz);
  integer n = U1. (Zn);
5
6
  integer m = UInt V
                       _m);
7
  integer
                 = U ~ (off3)
                 = FAL
  boolea sub_c
8
9
  const it inte ir nreg
```

Four **L** ົ .gle-vectors (FEAT 5 'E2)



FMLA ZA.<T>[<Wv>, <offs>{, VGx4}], { <Zn1>.<T>-<Zn4>.<T> }, <Zm>..<T>

```
1
  if !HaveSME2() then UNDEFINED;
2
  if sz == '1' && !HaveSMEF64F64() then UNDEFINED;
```

```
3
  integer v = UInt('010':Rv);
```

```
constant integer esize = 32 << UInt(sz);</pre>
4
```

```
integer n = UInt(Zn);
5
```

```
6
  integer m = UInt('0':Zm);
```

```
7
```

```
integer offset = UInt(off3);
boolean sub_op = FALSE;
8
```

```
9
  constant integer nreg = 4;
```
Assembler Symbols

<T> Is the size specifier, encoded in "sz":

SZ	<t></t>
0	S
1	D

- <Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
- <offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.
- <Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn".
- <Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" plus 3 modulo 32.
- <Zn2> Is the name of the second scalable vector register of a multi-¹ ctor sequence encoded as "Zn" plus 1 modulo 32.
- $\langle Zm \rangle$ Is the name of the second source scalable vector registe 20-Z15, e. $\forall ded in t = "Zm" field.$

```
1
    CheckStreamingSVEAndZAEnabled();
    constant integer VL = CurrentVL;
 2
3
    constant integer elements = VL DIV esize
 4
    integer vectors = VL DIV 8;
    integer vstride = vectors DIV nreg;
 5
 6
    bits(32) vbase = X[v, 32];
 7
    integer vec = (UInt(vbase) + offs
                                                MOD
                                                         ride;
    bits(VL) result;
8
9
    for r = 0 to nreg-1
10
                                (h. )
11
         bits(VL) operand1 =
                                         MOD 32,
         bits(VL) operand2 = Z[m,
12
13
         bits(VL) operanc' = ZAvector
                                             ec, VL];
14
         for e = 0 to \epsilon .ments-1
             bits(esi ) element1 = Elem_perand1, e, esize];
15
             bits(c .ze) elc .nt2 = Elem[operand2, e, esize];
16
             bits(e, e) e ment3 = Elem[operand3, e, esize];
if sub_op in element = FPNeg(element1);
esult e, esize] = FPMulAdd_ZA(element3, element1, element2, FPCR[]);
17
18
19
            ctorl c, VL,
c = vec - vstria
20
                                 / Jult;
21
                      vstrid
           Ć
```

D1.1.45 FMLA (multiple vectors)

Multi-vector floating-point fused multiply-add

The instruction operates on two or four ZA single-vector groups.

Multiply the corresponding floating-point elements of the two or four first and second source vectors and destructively add without intermediate rounding to the corresponding elements of the two or four ZA single-vector groups. The vector numbers forming the single-vector group within each half or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction follows SME ZA-targeting floating-point behaviors.

This instruction is unpredicated.

ID_AA64SMFR0_EL1.F64F64 indicates whether the double recision var. t is in remented.

It has encodings from 2 classes: Two ZA single-vectors (Four 7 single-vectors

Two ZA single-vectors (FEAT_SME2)

17 0 0 0 0 1 0 0 sz 1

FMLA ZA.<T>[<Wv>, <c {, VGx2}], <Zn1>.<T>-<Zn2>.<T> }, { <Zm1>.<T>-<Zm2>.<T> →}

```
1 if !HaveSME2() ther INDEFINED;
2 if sz == '1' && ' IVeSMEF64F64() th. UNDEFINED;
3 integer v = UIr ('010':F';
4 constant integ esize 32 << UInt(sz);
5 integer n = UInt n. ');
6 integer m ''Int(z '0');
7 integer rfs = UInt off',
8 boole . sub_o = FALSL
```

```
9 cor ant int ,er nreg = 2;
```

```
Four Z<sub>2</sub>. ngle-vectors
(FEAT_SN. '?)
```



```
if !HaveSME2() then UNDEFINED;
  if sz == '1' && !HaveSMEF64F64() then UNDEFINED;
2
3
  integer v = UInt('010':Rv);
4
  constant integer esize = 32 << UInt(sz);</pre>
  integer n = UInt(Zn:'00');
5
  integer m = UInt(Zm:'00');
6
  integer offset = UInt(off3);
7
8
  boolean sub_op = FALSE;
9
  constant integer nreg = 4;
```

Assembler Symbols

<T> Is the size specifier, encoded in "sz":

SZ	< T >
0	S
1	D

- <Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
- <offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.
- <Zn1> For the two ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.

For the four ZA single-vectors variant: is the name of the first scal otor register of a multi-vector sequence, encoded as "Zn" times 4.

- <Zn4> Is the name of the fourth scalable vector register of a multi-vector regulation of a multi-vector register of a multi-vecto
- <Zn2> Is the name of the second scalable vector register *c* a multi-vector source, encoded as "Zn" times 2 plus 1.
- <Zm1> For the two ZA single-vectors variant: is the name of first scall be vector register of a multi-vector sequence, encoded as "Zm" to les 2

For the four ZA single-vectors varia⁷ is the n² is of the finit scalable vector register of a multi-vector sequence, encoded as "Z₁, time r.

- <Zm4> Is the name of the fourth scalab. vector vister of a multi-vector sequence, encoded as "Zm" times 4 plus 3.
- <Zm2> Is the name of the sec in calable vector register of a multi-vector sequence, encoded as "Zm" times 2 plus 1

```
CheckStreaming EAndZAF .oled();
 1
    constant intege VL = urrentVL;
constant integer .ents = V D
 2
    constant integer ents = v DIV entremediate
integer s = DIV 8
integer vstri = vec rr JIV nreg;
bits 2) vbas = X[v, ...;
                                           DIV esize;
 3
4
 5
 6
 7
    int vec
                           + (vbase) + offset) MOD vstride;
 8
    bits (
                sult;
9
    for r = 0 > nreg-1
10
         bits(VL operand1 = Z[n+r, VL];
bits(VL) operand2 = Z[m+r, VL];
11
12
         bits(VL) operand3 = ZAvector[vec, VL];
13
         for e = 0 to elements-1
14
              bits(esize) element1 = Elem[operand1, e, esize];
15
              bits(esize) element2 = Elem[operand2, e, esize];
16
17
              bits(esize) element3 = Elem[operand3, e, esize];
18
              if sub_op then element1 = FPNeg(element1);
              Elem[result, e, esize] = FPMulAdd_ZA(element3, element1, element2, FPCR[]);
19
         ZAvector[vec, VL] = result;
20
21
         vec = vec + vstride;
```

D1.1.46 FMLAL (multiple and indexed vector)

Multi-vector floating-point multiply-add long by indexed element

The instruction operates on one, two, or four ZA double-vector groups.

This half-precision floating-point multiply-add long instruction widens all 16-bit half-precision elements in the one, two, or four first source vectors and the indexed element of the second source vector to single-precision format, then multiplies the corresponding elements and destructively adds these values without intermediate rounding to the overlapping 32-bit single-precision elements of the one, two, or four ZA double-vector groups.

The half-precision elements within the second source vector are specified using a 3-bit immediate index which selects the same element position within each 128-bit vector segment.

The lowest of the two consecutive vector numbers forming the double-vector group within all, each half, or each quarter of the ZA array are selected by the sum of the vector select register and module offset, modulo all, half, or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the λ operand consists of two or four ZA double-vector groups respectively. The VECTOR GROUP symbol is provided for disassembly, but optional in assembler source code.

This instruction follows SME ZA-targeting floating-point having

This instruction is unpredicated.

It has encodings from 3 classes: One ZA dou¹ -vector Iwo ZA c Je-vectors and Four ZA double-vectors

One ZA double-vector (FEAT_SME2)



FMLAL ZA.S[<Wv , <offsf>:<offsl , <Zn>.H, <Zm>.H[<index>]

- 1 if !HaveSME2() `en U _FINED;
- 2 integer v = UInt > :Rv);
- 3 integer Tht (2.
- 4 integer .n = t nt('0' n);
- 5 inte c offse = UInt(3:'0');
- 6 int **ar** inde urnt(i3h:i3l);
- 7 boole sr _op = ____; 8 constal. _nteger nreg = 1

Two ZA dou. -vectors (FEAT SME2)



FMLAL ZA.S[<Wv>, <offsf>:<offsl>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H[<index>]

```
1 if !HaveSME2() then UNDEFINED;
```

```
2 integer v = UInt('010':Rv);
```

```
3 integer n = UInt(Zn:'0');
```

```
4 integer m = UInt('0':Zm);
```

```
5 integer offset = UInt(off2:'0');
6 integer index = UInt(i3h:i3l);
```

```
6 integer index = UInt(i3)
7 boolean sub on = FALSE:
```

```
7 boolean sub_op = FALSE;
8 constant integer nreg = 2;
```

Four ZA double-vectors (FEAT_SME2) 0 0 1 0 0 0 Zm Rv Zn 1 0 0 1 1 i3h 1 FMLAL ZA.S[<Wv>, <offsf>:<offsl>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H[<index>] 1 if !HaveSME2() then UNDEFINED; 2 integer v = UInt('010':Rv); integer n = UInt(Zn:'00'); 3 4 integer m = UInt('0':Zm); integer offset = UInt(off2:'0'); 5 integer index = UInt(i3h:i31); 6 boolean sub_op = FALSE; 7 constant integer nreg = 4; **Assembler Symbols** /' field. $\langle Wv \rangle$ Is the 32-bit name of the vector select register W8.7 11, encrypt d in the For the one ZA double-vector variant: is the vector electron offset, r inting to first of two <offsf> consecutive vectors, encoded as "off3" field s 2. For the four ZA double-vectors and two .A double vectors w At: is the vector select offset, pointing to first of two consecutive v 'ors, enc int as "off2" field times 2. <offsl> For the one ZA double-vector riant: . e vector select offset, pointing to last of two consecutive vectors, encoder'ns "o. "" fiela res 2 plus 1. For the four ZA double- tors and two \double-vectors variant: is the vector select offset, pointing to last of tw _on. "utive vectors nooded as "off2" field times 2 plus 1. Is the name of the first source _____able vector register, encoded in the "Zn" field. $\langle Zn \rangle$ $\langle Zn1 \rangle$ For the two double-vectors val. at: is the name of the first scalable vector register of a multi-vec c sequer , encoded as "Zn" times 2. For the four ' ouble-ve' is variant: is the name of the first scalable vector register of a tor sec, ince, i loded as "Zn" times 4. m <Zn4>is the nr le of the full in scalable vector register of a multi-vector sequence, encoded as "Zn" times ls of name of the second scalable vector register of a multi-vector sequence, encoded as <Zn2>"Zn nes 2 plus 1. Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field. $\langle Zm \rangle$ <index> Is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields.

```
CheckStreamingSVEAndZAEnabled();
1
2
   constant integer VL = CurrentVL;
   constant integer elements = VL DIV 32;
3
4
   integer vectors = VL DIV 8;
   integer vstride = vectors DIV nreg;
5
   integer eltspersegment = 128 DIV 32;
6
   bits(32) vbase = X[v, 32];
7
   integer vec = (UInt(vbase) + offset) MOD vstride;
8
   bits(VL) result;
9
10
   vec = vec - (vec MOD 2);
11
12 for r = 0 to nreg-1
```

```
Chapter D1. SME instructions
D1.1. SME and SME2 data-processing instructions
```

```
bits(VL) operand1 = Z[n+r, VL];
13
14
         bits(VL) operand2 = Z[m, VL];
15
          for i = 0 to 1
16
              bits(VL) operand3 = ZAvector[vec + i, VL];
17
               for e = 0 to elements-1
18
                    integer segmentbase = e - (e MOD eltspersegment);
                    integer s = 2 * segmentbase + index;
19
                   bits(16) element1 = Elem[operand1, 2 * e + i, 16];
bits(16) element2 = Elem[operand2, s, 16];
bits(32) element3 = Elem[operand3, e, 32];
20
21
22
23
                    if sub_op then element1 = FPNeg(element1);
               Elem[result, e, 32] = FPMulAddH_ZA(element3, element1, element2, FPCR[]);
ZAvector[vec + i, VL] = result;
24
25
26
         vec = vec + vstride;
```

D1.1.47 FMLAL (multiple and single vector)

Multi-vector floating-point multiply-add long by vector

The instruction operates on one, two, or four ZA double-vector groups.

This half-precision floating-point multiply-add long instruction widens all 16-bit half-precision elements in the one, two, or four first source vectors and the second source vector to single-precision format, then multiplies the corresponding elements and destructively adds these values without intermediate rounding to the overlapping 32-bit single-precision elements of the one, two, or four ZA double-vector groups. The lowest of the two consecutive vector numbers forming the double-vector group within all, each half, or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo all, half, or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA d consists of two or four ZA double-vector groups respectively. The VECTOR GROUP symbol is referred disassembly, but optional in assembler source code.

This instruction follows SME ZA-targeting floating-point behaviors.

This instruction is unpredicated.

It has encodings from 3 classes: One ZA double-vector, and ZA puble-vectors and Four ZA double-vectors

One ZA double-vector (FEAT_SME2)



```
\label{eq:FMLAL} FMLAL \quad ZA.S[<\!Wv\!>, <\!offsf\!>:<\!offsl\!>\{, VGx2\}], \{ <\!Znl\!>.H - <\!Zn2\!>.H \}, <\!Zm\!>.H \\
```

```
1 if !HaveSME2() then UNDEFINED;
```

```
2 integer v = UInt('010':Rv);
```

```
3 integer n = UInt(Zn);
```

```
4 integer m = UInt('0':Zm);
5 integer offset = UInt(off2.'0')
```

```
5 integer offset = UInt(off2:'0');
6 boolean sub_op = FALSE;
```

```
7 constant integer nreg = 2;
```

Four ZA double-vectors (FEAT_SME2)



23

vec = vec + vstride;

Copyright © 2022-2023 Arm Limited or its affiliates. All rights reserved. Non-confidential

D1.1.48 FMLAL (multiple vectors)

Multi-vector floating-point multiply-add long

The instruction operates on two or four ZA double-vector groups.

This half-precision floating-point multiply-add long instruction widens all 16-bit half-precision elements in the two or four first and second source vectors to single-precision format, then multiplies the corresponding elements and destructively adds these values without intermediate rounding to the overlapping 32-bit single-precision elements of the two or four ZA double-vector groups. The lowest of the two consecutive vector numbers forming the double-vector group within each half or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA double-vector groups respectively. The VECTOR GROUP symbol is profor disassembly, but optional in assembler source code.

This instruction follows SME ZA-targeting floating-point behaviors.

This instruction is unpredicated.

It has encodings from 2 classes: Two ZA double-vectors Four Z' Jouble

Two ZA double-vectors (FEAT_SME2)

```
13
1
   0
       0
          0
              0
                 0
                     1
                        1
                            0
                                1
                                         7m
                                                     0
                                                          Rv
                                                                0
                                                                       0
                                                                                7n
                                                                                         0
                                                                                            0
```

<offsl>{, FMLAL ZA.S[<Wv>, <offsf x2}] <Zn1>.H-<Zn2>.H }, { <Zm1>.H-<Zm2>.H }

```
if !HaveSME2() then UNL_FINED;
1
```

```
:Rv);
2
  integer v = UInt('0'
```

- 3 integer n = UInt(7 0');
- integer m = UIn+ _m:'0') 4
- 5 integer offset UInt (2: (0):FAL

= 2:

6 boolean sub op

```
7
  constant integer
```

Four' A doub vectors (FI **5** SMF



```
FMLAL ZA.S[<Wv>, <offsf>:<offsl>{, VGx4}], { <Zn1>.H-<Zn4>.H }, { <Zm1>.H-<Zm4>.H }
```

```
if !HaveSME2() then UNDEFINED;
1
```

```
2
  integer v = UInt('010':Rv);
```

```
integer n = UInt(Zn:'00');
3
```

```
4
  integer m = UInt(Zm:'00');
5
```

```
integer offset = UInt(off2:'0');
boolean sub_op = FALSE;
6
```

constant integer nreg = 4;

Assembler Symbols

Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field. $\langle Wv \rangle$

<offsf> Is the vector select offset, pointing to first of two consecutive vectors, encoded as "off2" field times 2.

- <offsl> Is the vector select offset, pointing to last of two consecutive vectors, encoded as "off2" field times 2 plus 1.
- <Zn1> For the two ZA double-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.

For the four ZA double-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

- <Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.
- <Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.
- <Zm1> For the two ZA double-vectors variant: is the name of the first alable vec, register of a multi-vector sequence, encoded as "Zm" times 2.

For the four ZA double-vectors variant: is the name of *t* first sc. ble vecto egister of a multi-vector sequence, encoded as "Zm" times 4.

- <Zm4> Is the name of the fourth scalable vector register f a mu' vector sequence, encoded as "Zm" times 4 plus 3.
- <Zm2> Is the name of the second scalable vector regist of a notice vector regist. The vector sequence, encoded as "Zm" times 2 plus 1.

```
1
   CheckStreamingSVEAndZAEnabled
   constant integer VL = Curre
2
                                   /T.:
3
   constant integer elements
                                  VL DIV 32;
   integer vectors = VL DI' ;
integer vstride = vectors DIV
 4
5
                                       ٦g;
 6
   bits(32) vbase = X[, 32];
   integer vec = (UIr __vbase) + offse
                                           MOD vstride;
 7
 8
   bits(VL) result
 9
   vec = vec - (
                     MOD 2
10
11
    for r = 0 + o nreg
                                 ______;
12
        bit
                  peran.
                            = Z[
        (VL) د <sup>ن</sup>ط
13
                   erand∠
                                .a+r, VL];
14
         Jr i =
                 f co 1
15
                          rand3 = ZAvector[vec + i, VL];
            bit
                 e = 0 to elements-1
16
            f
17
                 bits(16) element1 = Elem[operand1, 2 * e + i, 16];
                 bits(16) element2 = Elem[operand2, 2 * e + i, 16];
18
19
                  `ts(32) element3 = Elem[operand3, e, 32];
20
                 if sub_op then element1 = FPNeg(element1);
21
                 Elem[result, e, 32] = FPMulAddH_ZA(element3, element1, element2, FPCR[]);
22
            ZAvector[vec + i, VL] = result;
23
        vec = vec + vstride;
```

D1.1.49 FMLS (multiple and indexed vector)

Multi-vector floating-point fused multiply-subtract by indexed element

The instruction operates on two or four ZA single-vector groups.

Multiply the indexed element of the second source vector by the corresponding floating-point elements of the two or four first source vectors and destructively subtract without intermediate rounding from the corresponding elements of the two or four ZA single-vector groups.

The elements within the second source vector are specified using an immediate element index which selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 1 to 2 bits depending on the size of the element. The vector numbers forming the single-vector group within each half or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA oper disconsists of the vector groups respectively. The VECTOR GROUP symbol is preferred for disconsists of the vector on all in assembler source code.

This instruction follows SME ZA-targeting floating-point b aviors.

This instruction is unpredicated.

ID_AA64SMFR0_EL1.F64F64 indicates whether _____uble-____cision y___ant is implemented.

It has encodings from 4 classes: Two ZA sing -vectors f single is in elements, Two ZA single-vectors of double precision elements, Four ZA single ctors of close classes and Four ZA single-vectors of double precision elements

Two ZA single-vectors of single p cisio. \lemen. (FEAT_SME2)



```
1 if !Ha' oME2 then DEF ED,
2 inter r v = U it ('010 ;;
3 cor ant int er esize = 32;
4 inter n office, J');
```

```
5 intege. = UInt('0':Zm);
```

```
6 integer (set = UInt(off3);
```

```
7 integer in = UInt(i2);
```

```
8 boolean sub_cp = TRUE;
```

```
9 constant integer nreg = 2;
```

Two ZA single-vectors of double precision elements (FEAT_SME_F64F64)



```
FMLS ZA.D[<Wv>, <offs>{, VGx2}], { <Zn1>.D-<Zn2>.D }, <Zm>.D[<index>]
```

```
1 if !(HaveSME2() && HaveSMEF64F64()) then UNDEFINED;
2 integer v = UInt('010':Rv);
```

```
3 constant integer esize = 64;
```

```
Chapter D1. SME instructions
D1.1. SME and SME2 data-processing instructions
```

```
4 integer n = UInt(Zn:'0');
5 integer m = UInt('0':Zm);
6 integer offset = UInt(off3);
7 integer index = UInt(i1);
8 boolean sub_op = TRUE;
```

9 constant integer nreg = 2;

Four ZA single-vectors of single precision elements (FEAT_SME2)



Assembler Symbols

- <Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
- <offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.
- <Zn1> For the two ZA single-vectors of double precision elements and two ZA single-vectors of single precision elements variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.

For the four ZA single-vectors of double precision elements and four ZA single-vectors of single precision elements variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

- <Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.
- <Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as

"Zn" times 2 plus 1.

- <Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.
- <index> For the four ZA single-vectors of single precision elements and two ZA single-vectors of single precision elements variant: is the element index, in the range 0 to 3, encoded in the "i2" field.

For the four ZA single-vectors of double precision elements and two ZA single-vectors of double precision elements variant: is the element index, in the range 0 to 1, encoded in the "i1" field.

```
CheckStreamingSVEAndZAEnabled();
 1
    constant integer VL = CurrentVL;
 2
 3
    constant integer elements = VL DIV esize;
 4
    integer vectors = VL DIV 8;
    integer vstride = vectors DIV nreg;
 5
    integer eltspersegment = 128 DIV esize;
 6
 7
    bits(32) vbase = X[v, 32];
 8
    integer vec = (UInt(vbase) + offset) MOD vstride;
 9
    bits(VL) result;
10
11
    for r = 0 to nreg-1
12
         bits(VL) operand1 = Z[n+r, VL];
         bits(VL) operand2 = Z[m, VL];
13
14
         bits(VL) operand3 = ZAvector[vec,
                                                        1;
15
          for e = 0 to elements-1
16
              bits(esize) element1 = Elem[op
                                                         nd'
              integer segmentbase = e 'e MOL
integer s = segmentbase + 'ex;
17
                                                            .spersegment);
               integer s = segmentbase +
18
         bits(esize) element2 Elem[\_randz_3, e
bits(esize) element = Elem[op\_nd3, , e
if sub_op then e' u '1 = FPNeg(\_ment1);
Elem[result, e, ssize = FPMulAdd_.A(eleme
ZAvector[vec, VL' = result;
19
                                                                   esize];
20
                                                                   esize];
21
22
                                             FPMulAdd_.A(element3, element1, element2, FPCR[]);
23
         vec = vec + vs<sup>+</sup> _de;
24
```

D1.1.50 FMLS (multiple and single vector)

Multi-vector floating-point fused multiply-subtract by vector

The instruction operates on two or four ZA single-vector groups.

Multiply the corresponding floating-point elements of the two or four first source vector with corresponding elements of the second source vector and destructively subtract without intermediate rounding from the corresponding elements of the two or four ZA single-vector groups. The vector numbers forming the single-vector group within each half or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction follows SME ZA-targeting floating-point behaviors.

This instruction is unpredicated.

ID AA64SMFR0 EL1.F64F64 indicates whether the double recision var. t is ir remented.

It has encodings from 2 classes: Two ZA single-vectors (Four 7) single-vec ors

Two ZA single-vectors (FEAT_SME2)



FMLS ZA. <T>[<Wv>,VGx2}], [Zn1>.<T>-<Zn2>.<T> }, <Zm>.<T>

```
if !HaveSME2() then "NDEFINED;
1
2
  if sz == '1' && !H _SMEF64F64()
                                      uNDEFINED;
3
  integer v = UInt J10':Rv);
  constant integ esize
4
                            32 << UInt(sz);
  integer n = U1. (Zn);
5
6
  integer m = UInt V
                        _m);
                 = D.
7
  integer
                       └(off3)
                 = TRU
  boolea sub_c
8
9
  const it inte ir nreg
```

Four **L** ົ້ .gle-vectors (FEAT 5 'E2)



FMLS ZA.<T>[<Wv>, <offs>{, VGx4}], { <Zn1>.<T>-<Zn4>.<T> }, <Zm>..<T>

```
1
  if !HaveSME2() then UNDEFINED;
2
  if sz == '1' && !HaveSMEF64F64() then UNDEFINED;
```

```
3
  integer v = UInt('010':Rv);
```

```
constant integer esize = 32 << UInt(sz);</pre>
4
```

```
integer n = UInt(Zn);
5
```

```
6
  integer m = UInt('0':Zm);
```

```
7
```

```
integer offset = UInt(off3);
boolean sub_op = TRUE;
8
```

```
9
  constant integer nreg = 4;
```

Assembler Symbols

<T> Is the size specifier, encoded in "sz":

SZ	<t></t>
0	S
1	D

- <Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
- <offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.
- <Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn".
- <Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" plus 3 modulo 32.
- <Zn2> Is the name of the second scalable vector register of a multi-¹ ctor sequence encoded as "Zn" plus 1 modulo 32.
- $\langle Zm \rangle$ Is the name of the second source scalable vector registe 20-Z15, e. $\forall ded in t = "Zm" field.$

```
1
    CheckStreamingSVEAndZAEnabled();
    constant integer VL = CurrentVL;
 2
3
    constant integer elements = VL DIV esize
 4
    integer vectors = VL DIV 8;
    integer vstride = vectors DIV nreg;
 5
 6
    bits(32) vbase = X[v, 32];
 7
    integer vec = (UInt(vbase) + offs
                                                MOD
                                                         ride;
    bits(VL) result;
8
9
    for r = 0 to nreg-1
10
                                (h. )
11
         bits(VL) operand1 =
                                         MOD 32,
         bits(VL) operand2 = Z[m,
12
13
         bits(VL) operanc' = ZAvector
                                             ec, VL];
14
         for e = 0 to \epsilon .ments-1
             bits(esi ) element1 = Elem_perand1, e, esize];
15
             bits(c .ze) elc .nt2 = Elem[operand2, e, esize];
16
             bits(e, e) e ment3 = Elem[operand3, e, esize];
if sub_op in element = FPNeg(element1);
esult e, esize] = FPMulAdd_ZA(element3, element1, element2, FPCR[]);
17
18
19
            ctorl c, VL,
c = vec - vstria
20
                                 / Jult;
21
                      vstrid
           Ć
```

Chapter D1. SME instructions D1.1. SME and SME2 data-processing instructions

D1.1.51 FMLS (multiple vectors)

Multi-vector floating-point fused multiply-subtract

The instruction operates on two or four ZA single-vector groups.

Multiply the corresponding floating-point elements of the two or four first and second source vectors and destructively subtract without intermediate rounding from the corresponding elements of the two or four ZA single-vector groups. The vector numbers forming the single-vector group within each half or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction follows SME ZA-targeting floating-point behaviors.

This instruction is unpredicated.

ID_AA64SMFR0_EL1.F64F64 indicates whether the double recision var. t is in remented.

It has encodings from 2 classes: Two ZA single-vectors (Four 7 single-vectors

Two ZA single-vectors (FEAT_SME2)

17 0 0 0 0 1 0 sz 1

FMLS ZA.<T>[<Wv>, <c {, VGx2}], <Zn1>.<T>-<Zn2>.<T> }, { <Zm1>.<T>-<Zm2>..<T>

```
1 if !HaveSME2() ther INDEFINED;
2 if sz == '1' && ' VeSMEF64F64() th. UNDEFINED;
3 integer v = UIr ('010':F';
4 constant integ esize 32 << UInt(sz);
5 integer n = UInt n. ');
6 integer m ''Int(z '0');
7 integer rfs = UInt off',
8 boole . sub_o = TRUE,
```

cor ant int ,er nreg = 2;

```
Four Z<sub>2</sub>. ngle-vectors
(FEAT_SN. '?)
```



```
1
  if !HaveSME2() then UNDEFINED;
2
  if sz == '1' && !HaveSMEF64F64() then UNDEFINED;
3
  integer v = UInt('010':Rv);
4
  constant integer esize = 32 << UInt(sz);</pre>
  integer n = UInt(Zn:'00');
5
  integer m = UInt(Zm:'00');
6
  integer offset = UInt(off3);
7
8
  boolean sub_op = TRUE;
9
  constant integer nreg = 4;
```

Assembler Symbols

<T> Is the size specifier, encoded in "sz":

SZ	< T >
0	S
1	D

- <Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
- <offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.
- <Zn1> For the two ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.

For the four ZA single-vectors variant: is the name of the first scal otor register of a multi-vector sequence, encoded as "Zn" times 4.

- <Zn4> Is the name of the fourth scalable vector register of a multi-vector regulation of a multi-vector register of a multi-vecto
- <Zn2> Is the name of the second scalable vector register *c* a multi-vector source, encoded as "Zn" times 2 plus 1.
- <Zm1> For the two ZA single-vectors variant: is the name of first scall be vector register of a multi-vector sequence, encoded as "Zm" to les 2

For the four ZA single-vectors variates is the n_{z} is the n_{z} is of the finite scalable vector register of a multi-vector sequence, encoded as "Z₁, time r.

- <Zm4> Is the name of the fourth scalab. vector vister of a multi-vector sequence, encoded as "Zm" times 4 plus 3.
- <Zm2> Is the name of the sec in calable vector register of a multi-vector sequence, encoded as "Zm" times 2 plus 1

```
CheckStreaming EAndZAF .oled();
 1
    constant intege VL = JurrentVL;
constant integer dents = V DIV d
integer s = DIV 8
integer vstri = vec rr JIV nreg;
bits 2) vbas = X[v, z];
 2
                                           DIV esize;
 3
4
 5
6
 7
    int vec
                           + (vbase) + offset) MOD vstride;
 8
    bits(
                sult;
9
    for r = 0 > nreg-1
10
         bits(VL operand1 = Z[n+r, VL];
bits(VL) operand2 = Z[m+r, VL];
11
12
         bits(VL) operand3 = ZAvector[vec, VL];
13
         for e = 0 to elements-1
14
              bits(esize) element1 = Elem[operand1, e, esize];
15
              bits(esize) element2 = Elem[operand2, e, esize];
16
17
              bits(esize) element3 = Elem[operand3, e, esize];
18
              if sub_op then element1 = FPNeg(element1);
              Elem[result, e, esize] = FPMulAdd_ZA(element3, element1, element2, FPCR[]);
19
         ZAvector[vec, VL] = result;
20
21
         vec = vec + vstride;
```

D1.1.52 FMLSL (multiple and indexed vector)

Multi-vector floating-point multiply-subtract long by indexed element

The instruction operates on one, two, or four ZA double-vector groups.

This half-precision floating-point multiply-subtract long instruction widens all 16-bit half-precision elements in the one, two, or four first source vectors and the indexed element of the second source vector to single-precision format, then multiplies the corresponding elements and destructively subtracts these values without intermediate rounding from the overlapping 32-bit single-precision elements of the one, two, or four ZA double-vector groups.

The half-precision elements within the second source vector are specified using a 3-bit immediate index which selects the same element position within each 128-bit vector segment.

The lowest of the two consecutive vector numbers forming the double-vector group within all, each half, or each quarter of the ZA array are selected by the sum of the vector select registre and modulo all, half, or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the λ operand consists of two or four ZA double-vector groups respectively. The VECTOR GROUP states in a symbol is provided in assembler source code.

This instruction follows SME ZA-targeting floating-point having

This instruction is unpredicated.

It has encodings from 3 classes: One ZA dou¹ -vector Iwo ZA c Je-vectors and Four ZA double-vectors

One ZA double-vector (FEAT_SME2)



FMLSL ZA.S[<Wv , <offsf>:<offsl, , <Zn>.H, <Zm>.H[<index>]

```
1 if !HaveSME2() `en U' FINED;
```

- 2 integer v = UInt γ^{*} :Rv);
- 3 integer r Tht (2.
- 4 **intege** *i* = *i* it ('0' 'n);
- 5 inter c offse = UInt(3:'0'); 6 int er inder urnt(i3h:i31);
- 6 int or inde "Int (i3 7 boole sr op = . ;
- 8 constal. .nteger nreg =

Two ZA dou. -vectors (FEAT_SME2)

FMLSL ZA.S[<Wv>, <offsf>:<offsl>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H[<index>]

```
1 if !HaveSME2() then UNDEFINED;
```

```
2 integer v = UInt('010':Rv);
```

```
3 integer n = UInt(Zn:'0');
```

```
4 integer m = UInt('0':Zm);
```

```
5 integer offset = UInt(off2:'0');
6 integer index = UInt(i3h:i3l);
```

```
6 integer index = UInt(i
7 boolean sub on = TRUE:
```

```
7 boolean sub_op = TRUE;
8 constant integer nreg = 2;
```

Four ZA double-vectors (FEAT_SME2) 0 0 1 0 0 0 Zm Rv Zn 1 0 0 1 1 i3h 1 FMLSL ZA.S[<Wv>, <offsf>:<offsl>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H[<index>] 1 if !HaveSME2() then UNDEFINED; 2 integer v = UInt('010':Rv); integer n = UInt(Zn:'00'); 3 4 integer m = UInt('0':Zm); integer offset = UInt(off2:'0'); 5 integer index = UInt(i3h:i31); 6 boolean sub_op = TRUE; 7 constant integer nreg = 4; **Assembler Symbols** /' field. $\langle Wv \rangle$ Is the 32-bit name of the vector select register W8.7 11, encrypt d in the For the one ZA double-vector variant: is the vector electron offset, r inting to first of two <offsf> consecutive vectors, encoded as "off3" field s 2. For the four ZA double-vectors and two .A double vectors w At: is the vector select offset, pointing to first of two consecutive v 'ors, enc int as "off2" field times 2. <offsl> For the one ZA double-vector riant: . e vector select offset, pointing to last of two consecutive vectors, encoder'ns "o. "" fiela res 2 plus 1. For the four ZA double- tors and two \double-vectors variant: is the vector select offset, pointing to last of tw _on. "utive vectors nooded as "off2" field times 2 plus 1. Is the name of the first source _____able vector register, encoded in the "Zn" field. $\langle Zn \rangle$ $\langle Zn1 \rangle$ For the two double-vectors val. at: is the name of the first scalable vector register of a multi-vec c sequer , encoded as "Zn" times 2. For the four ' ouble-ve' is variant: is the name of the first scalable vector register of a tor sec, ince, i loded as "Zn" times 4. m <Zn4>is the nr le of the full in scalable vector register of a multi-vector sequence, encoded as "Zn" times ls of name of the second scalable vector register of a multi-vector sequence, encoded as $\langle Zn2 \rangle$ "Zn nes 2 plus 1. Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field. $\langle Zm \rangle$ <index> Is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields.

```
CheckStreamingSVEAndZAEnabled();
1
2
   constant integer VL = CurrentVL;
   constant integer elements = VL DIV 32;
3
4
   integer vectors = VL DIV 8;
   integer vstride = vectors DIV nreg;
5
   integer eltspersegment = 128 DIV 32;
6
   bits(32) vbase = X[v, 32];
7
   integer vec = (UInt(vbase) + offset) MOD vstride;
8
   bits(VL) result;
9
10
   vec = vec - (vec MOD 2);
11
12 for r = 0 to nreg-1
```

```
Chapter D1. SME instructions
D1.1. SME and SME2 data-processing instructions
```

```
bits(VL) operand1 = Z[n+r, VL];
13
14
         bits(VL) operand2 = Z[m, VL];
15
         for i = 0 to 1
16
             bits(VL) operand3 = ZAvector[vec + i, VL];
17
              for e = 0 to elements-1
18
                   integer segmentbase = e - (e MOD eltspersegment);
                   integer s = 2 * segmentbase + index;
19
                  bits(16) element1 = Elem[operand1, 2 * e + i, 16];
bits(16) element2 = Elem[operand2, s, 16];
bits(32) element3 = Elem[operand3, e, 32];
20
21
22
23
                  if sub_op then element1 = FPNeg(element1);
                  Elem[result, e, 32] = FPMulAddH_ZA(element3, element1, element2, FPCR[]);
24
25
              ZAvector[vec + i, VL] = result;
26
         vec = vec + vstride;
```

D1.1.53 FMLSL (multiple and single vector)

Multi-vector floating-point multiply-subtract long by vector

The instruction operates on one, two, or four ZA double-vector groups.

This half-precision floating-point multiply-subtract long instruction widens all 16-bit half-precision elements in the one, two, or four first source vectors and the second source vector to single-precision format, then multiplies the corresponding elements and destructively subtracts these values without intermediate rounding from the overlapping 32-bit single-precision elements of the one, two, or four ZA double-vector groups. The lowest of the two consecutive vector numbers forming the double-vector group within all, each half, or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo all, half, or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA d consists of two or four ZA double-vector groups respectively. The VECTOR GROUP symbol is referred disassembly, but optional in assembler source code.

This instruction follows SME ZA-targeting floating-point behaviors.

This instruction is unpredicated.

It has encodings from 3 classes: One ZA double-vector, and ZA puble-vectors and Four ZA double-vectors

One ZA double-vector (FEAT_SME2)



```
FMLSL ZA.S[<Wv>, <offsf>:<offsl>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H
```

```
1 if !HaveSME2() then UNDEFINED;
```

```
2 integer v = UInt('010':Rv);
```

```
3 integer n = UInt(Zn);
```

```
4 integer m = UInt('0':Zm);
5 integer offset = UInt(off2:'0'
```

```
5 integer offset = UInt(off2:'0');
6 boolean sub_op = TRUE;
```

```
7 constant integer nreg = 2;
```

Four ZA double-vectors (FEAT_SME2)



D1.1.54 FMLSL (multiple vectors)

Multi-vector floating-point multiply-subtract long

The instruction operates on two or four ZA double-vector groups.

This half-precision floating-point multiply-subtract long instruction widens all 16-bit half-precision elements in the two or four first and second source vectors to single-precision format, then multiplies the corresponding elements and destructively subtracts these values without intermediate rounding from the overlapping 32-bit single-precision elements of the two or four ZA double-vector groups. The lowest of the two consecutive vector numbers forming the double-vector group within each half or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA double-vector groups respectively. The VECTOR GROUP symbol is project of disassembly, but optional in assembler source code.

This instruction follows SME ZA-targeting floating-point behaviors.

This instruction is unpredicated.

It has encodings from 2 classes: Two ZA double-vectors / J Four Z / Jouble- Jr

Two ZA double-vectors (FEAT_SME2)

```
13
1
   0
       0
          0
             0
                 0
                     1
                        1
                            0
                                1
                                         7m
                                                     0
                                                          Rv
                                                                0
                                                                       0
                                                                                7n
                                                                                         0
                                                                                            0
```

 $\label{eq:FMLSL ZA.S[<Wv>, <offsf <offsl>{, `x2}], { <Zn1>.H-<Zn2>.H }, { <Zm1>.H-<Zm2>.H }$

```
1 if !HaveSME2() then UNL_FINED;
```

```
2 integer v = UInt('0' ':Rv);
```

```
3 integer n = UInt(7 .'0');
```

- 4 integer m = UIn⁺ .m:'0')
- 5 integer offset UInt(r .2:'0');
 6 boolean sub_op TRUF

= 2;

6 **boolean** sub_op T 7 constant **integer**

```
7 constant integer
```

Four A doub vectors (FI ^C_SMF



```
FMLSL ZA.S[<Wv>, <offsf>:<offsl>{, VGx4}], { <Zn1>.H-<Zn4>.H }, { <Zm1>.H-<Zm4>.H }
```

```
1 if !HaveSME2() then UNDEFINED;
```

```
2 integer v = UInt('010':Rv);
```

```
3 integer n = UInt(Zn:'00');
```

```
4 integer m = UInt(Zm:'00');
```

```
5 integer offset = UInt(off2:'0');
```

```
6 boolean sub_op = TRUE;
7 constant integer nreg = 4;
```

Assembler Symbols

<Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.

<offsf> Is the vector select offset, pointing to first of two consecutive vectors, encoded as "off2" field

times 2.

- <offsl> Is the vector select offset, pointing to last of two consecutive vectors, encoded as "off2" field times 2 plus 1.
- <Zn1> For the two ZA double-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.

For the four ZA double-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

- <Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.
- <Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.
- <Zm1> For the two ZA double-vectors variant: is the name of the first alable vec, register of a multi-vector sequence, encoded as "Zm" times 2.

For the four ZA double-vectors variant: is the name of *t* first sc. ble vecto egister of a multi-vector sequence, encoded as "Zm" times 4.

- <Zm4> Is the name of the fourth scalable vector register f a mu vector sequence, encoded as "Zm" times 4 plus 3.
- <Zm2> Is the name of the second scalable vector regist of a notice vector regist. The vector sequence, encoded as "Zm" times 2 plus 1.

```
1
   CheckStreamingSVEAndZAEnabled
   constant integer VL = Curre
2
                                   /T.:
3
   constant integer elements
                                  VL DIV 32;
   integer vectors = VL DI' ;
integer vstride = vectors DIV
 4
5
                                       ٦g;
 6
   bits(32) vbase = X[, 32];
   integer vec = (UIr __vbase) + offse
                                           MOD vstride;
 7
 8
   bits(VL) result
 9
   vec = vec - (
                     MOD 2
10
11
    for r = 0 + o nreg
                                 ______;
12
        bit
                  peran.
                            = Z[
        (VL) د <sup>ن</sup>ط
13
                   erand∠
                                .a+r, VL];
14
         Jr i =
                 f co 1
15
                          rand3 = ZAvector[vec + i, VL];
            bit
                 e = 0 to elements-1
16
            f
17
                 bits(16) element1 = Elem[operand1, 2 * e + i, 16];
                 bits(16) element2 = Elem[operand2, 2 * e + i, 16];
18
19
                  `ts(32) element3 = Elem[operand3, e, 32];
20
                 if sub_op then element1 = FPNeg(element1);
21
                 Elem[result, e, 32] = FPMulAddH_ZA(element3, element1, element2, FPCR[]);
22
            ZAvector[vec + i, VL] = result;
23
        vec = vec + vstride;
```

D1.1.55 FMOPA (widening)

Half-precision floating-point sum of outer products and accumulate

The half-precision floating-point sum of outer products and accumulate instruction works with a 32-bit element ZA tile.

This instruction widens the $SVL_S \times 2$ sub-matrix of half-precision floating-point values held in the first source vector to single-precision floating-point values and multiplies it by the widened $2 \times SVL_S$ sub-matrix of half-precision floating-point values in the second source vector to single-precision floating-point values.

Each source vector is independently predicated by a corresponding governing predicate. When a 16-bit source element is Inactive it is treated as having the value +0.0, but if both pairs of source vector elements that correspond to a 32-bit destination element contain Inactive elements, then the destination element remains unmodified.

The resulting $SVL_S \times SVL_S$ single-precision floating-point sum of outer source is then destructively added to the single-precision floating-point destination tile. This is equivalence of performine a 2-way dot product and accumulate to each of the destination tile elements.

Each 32-bit container of the first source vector holds 2 consective colunt element of each row of a $SVL_S \times 2$ sub-matrix. Similarly, each 32-bit container of the second source vector holds for ecutive row elements of each column of a $2 \times SVL_S$ sub-matrix.

This instruction follows SME ZA-targeting floating-point be. ors.

SME (FEAT_SME)



FMOPA <ZAda>.S, <Pn>/M, <Ph. 'M, <Zn>.H, <Zm>.H

```
1
   if !HaveSME() thr UNDEFINED;
   integer a = UIr (Pn);
2
3
   integer b = U1 (Pm);
4
   integer n = UInt
5
   integer m
                 TInt (A
6
   intege<sup>,</sup> Ja
                   Int (ZA
   boolr . sub_o
                    = FALS
```

```
Assen. 'ar ymbols
```

- <ZAda> Is i. name of the ZA tile ZA0-ZA3, encoded in the "ZAda" field.
 - <Pn> Is the name of the first governing scalable predicate register P0-P7, encoded in the "Pn" field.
 - <Pm> Is the name of the second governing scalable predicate register P0-P7, encoded in the "Pm" field.
 - <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
 - <Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

```
1 CheckStreamingSVEAndZAEnabled();
2 constant integer VL = CurrentVL;
3 constant integer PL = VL DIV 8;
4 constant integer dim = VL DIV 32;
5 bits(PL) mask1 = P[a, PL];
6 bits(PL) mask2 = P[b, PL];
7 bits(VL) operand1 = Z[n, VL];
```

```
8 bits(VL) operand2 = Z[m, VL];
9
    bits(dim*dim*32) operand3 = ZAtile[da, 32, dim*dim*32];
10
    bits(dim*dim*32) result;
11
12
    for row = 0 to dim-1
13
         for col = 0 to dim-1
              // determine row/col predicates
14
15
             boolean prow_0 = (ActivePredicateElement(mask1, 2*row + 0, 16));
             boolean prow_1 = (ActivePredicateElement(mask1, 2*row + 1, 16));
boolean pcol_0 = (ActivePredicateElement(mask2, 2*col + 0, 16));
16
17
18
             boolean pcol_1 = (ActivePredicateElement(mask2, 2*col + 1, 16));
19
20
             bits(32) sum = Elem[operand3, row*dim+col, 32];
21
              if (prow_0 && pcol_0) || (prow_1 && pcol_1) then
22
                  bits(16) erow_0 = (if prow_0 then Elem[operand1, 2*row + 0, 16] else FPZero('0',
                       →16));
23
                  bits(16) erow_1 = (if prow_1 then Elem[operand1, _____1, 16] else FPZero('0',
                       \rightarrow16));
24
                  bits(16) ecol_0 = (if pcol_0 then Elem[operanc'
                                                                            2 \star col + 0,
                                                                                         16] else FPZero('0',
                       \rightarrow16));
25
                  bits(16) ecol_1 = (if pcol_1 then Elem[opr and2, +col + 1, 6] else FPZero('0',
                       →16));
26
                  if sub_op then
                  if prow_0 then erow_0 = FPNeg(er __0);
if prow_1 then erow_1 = FPNeg(er __1);
sum = FPDotAdd_ZA(sum, erow_0, erow_1,
27
28
                                                                1);
                                                                     __0, eco' _, FPCR[]);
29
30
31
              Elem[result, row*dim+col, 32] =
                                                   am;
32
    ZAtile[da, 32, dim*dim*32] = result;
33
```

D1.1.56 FMOPA (non-widening)

Floating-point outer product and accumulate

The single-precision variant works with a 32-bit element ZA tile.

The double-precision variant works with a 64-bit element ZA tile.

These instructions generate an outer product of the first source vector and the second source vector. In case of the single-precision variant, the first source is $SVL_S \times 1$ vector and the second source is $1 \times SVL_S$ vector. In case of the double-precision variant, the first source is $SVL_D \times 1$ vector and the second source is $1 \times SVL_D$ vector.

Each source vector is independently predicated by a corresponding governing predicate. When either source vector element is Inactive the corresponding destination tile element remains unmodified.

The resulting outer product, $SVL_S \times SVL_S$ in case of single-precision \leftarrow or $SVL_D \times SVL_D$ in case of double-precision variant, is then destructively added to the destinationale. The sequivalent to performing a single multiply-accumulate to each of the destination tile elements

This instruction follows SME ZA-targeting floating-point behaviors.

ID_AA64SMFR0_EL1.F64F64 indicates whether the doub' precision varia. is .plemented.

It has encodings from 2 classes: Single-precision and Dou -- pre ion

Single-precision (FEAT_SME)



FMOPA <ZAda>.D, <Pn>/M, <Pm>/M, <Zn>.D, <Zm>.D

- 1 if !HaveSMEF64F64() then UNDEFINED;
- 2 constant **integer** esize = 64;
- 3 integer a = UInt(Pn);
- 4 integer b = UInt (Pm);
- 5 integer n = UInt(Zn); 6 integer m = UInt(Zm);
- 7 integer da = UInt(ZAda);
- 8 boolean sub_op = FALSE;

Assembler Symbols

<ZAda> For the single-precision variant: is the name of the ZA tile ZA0-ZA3, encoded in the "ZAda" field.

For the double-precision variant: is the name of the ZA tile ZA0-ZA7, encoded in the "ZAda" field.

- <Pn> Is the name of the first governing scalable predicate register P0-P7, encoded in the "Pn" field.
- <Pm> Is the name of the second governing scalable predicate register P0-P7, encoded in the "Pm" field.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

```
CheckStreamingSVEAndZAEnabled();
1
   constant integer VL = CurrentVL;
2
   constant integer PL = VL DIV 8;
3
Δ
   constant integer dim = VL DIV esize;
   bits(PL) mask1 = P[a, PL];
5
   bits(PL) mask2 = P[b, PL];
6
7
   bits(VL) operand1 = Z[n, VL];
   bits(VL) operand2 = Z[m, VL];
8
9
   bits(dim*dim*esize) operand3 = ZAtile[d
                                               esiz€
                                                       dim*a.
                                                                   2e1;
10
   bits(dim*dim*esize) result;
11
12
   for row = 0 to dim-1
13
        for col = 0 to dim-1
                                   Elen.
14
            bits(esize) element1
                                           peranc
                                                    row, esize];
15
            bits(esize) element<sup>2</sup>
                                    Elem[o, rand2, ol, esize];
16
                                  = Elem[ope hd3, row*dim+col, esize];
            bits (esize) elemer
17
18
            if (ActivePred_ateEle nt(mask1, row, esize) &&
19
                  Active redicateEle nt(mask2, col, esize)) then
20
                if sub p then element
                                          FPNeg(element1);
                     esult, row*dim+col, esize] = FPMulAdd_ZA(element3, element1, element2,
21
                Eler
                      →FPCR
22
            else
23
                         .it, row _m+col, esize] = element3;
                Elem
24
25
    ZAtile
           .a, es
                  ie, din 'ir .size] = result;
```

D1.1.57 FMOPS (widening)

Half-precision floating-point sum of outer products and subtract

The half-precision floating-point sum of outer products and subtract instruction works with a 32-bit element ZA tile.

This instruction widens the $SVL_S \times 2$ sub-matrix of half-precision floating-point values held in the first source vector to single-precision floating-point values and multiplies it by the widened $2 \times SVL_S$ sub-matrix of half-precision floating-point values in the second source vector to single-precision floating-point values.

Each source vector is independently predicated by a corresponding governing predicate. When a 16-bit source element is Inactive it is treated as having the value +0.0, but if both pairs of source vector elements that correspond to a 32-bit destination element contain Inactive elements, then the destination element remains unmodified.

The resulting $SVL_S \times SVL_S$ single-precision floating-point sum of outer r_{soucco} then destructively subtracted from the single-precision floating-point destination tile. This is equivalent to perform g a 2-way dot product and subtract from each of the destination tile elements.

Each 32-bit container of the first source vector holds 2 consective colunt element of each row of a $SVL_S \times 2$ sub-matrix. Similarly, each 32-bit container of the second scale vector holds for ecutive row elements of each column of a $2 \times SVL_S$ sub-matrix.

This instruction follows SME ZA-targeting floating-point be. ors.

SME (FEAT_SME)



FMOPS <ZAda>.S, <Pn>/M, <Ph. 'M, <Zn>.H, <Zm>.H

```
if !HaveSME() thr UNDEFINED;
1
   integer a = UIr (Pn);
2
3
   integer b = U1 (Pm);
4
   integer n = UInt
5
   integer m
                 TInt (A
6
   intege<sup>,</sup> Ja
                   Int (ZA
   boolr . sub_o
                    = TRUE:
```

Assen. 'er ymbols

- <ZAda> Is i. name of the ZA tile ZA0-ZA3, encoded in the "ZAda" field.
 - <Pn> Is the name of the first governing scalable predicate register P0-P7, encoded in the "Pn" field.
 - <Pm> Is the name of the second governing scalable predicate register P0-P7, encoded in the "Pm" field.
 - <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
 - <Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

```
1 CheckStreamingSVEAndZAEnabled();
2 constant integer VL = CurrentVL;
3 constant integer PL = VL DIV 8;
4 constant integer dim = VL DIV 32;
5 bits(PL) mask1 = P[a, PL];
6 bits(PL) mask2 = P[b, PL];
7 bits(VL) operand1 = Z[n, VL];
```

```
8 bits(VL) operand2 = Z[m, VL];
9
    bits(dim*dim*32) operand3 = ZAtile[da, 32, dim*dim*32];
10
    bits(dim*dim*32) result;
11
12
    for row = 0 to dim-1
13
         for col = 0 to dim-1
              // determine row/col predicates
14
15
             boolean prow_0 = (ActivePredicateElement(mask1, 2*row + 0, 16));
             boolean prow_1 = (ActivePredicateElement(mask1, 2*row + 1, 16));
boolean pcol_0 = (ActivePredicateElement(mask2, 2*col + 0, 16));
16
17
18
             boolean pcol_1 = (ActivePredicateElement(mask2, 2*col + 1, 16));
19
20
             bits(32) sum = Elem[operand3, row*dim+col, 32];
21
              if (prow_0 && pcol_0) || (prow_1 && pcol_1) then
22
                  bits(16) erow_0 = (if prow_0 then Elem[operand1, 2*row + 0, 16] else FPZero('0',
                       →16));
23
                  bits(16) erow_1 = (if prow_1 then Elem[operand1, _____1, 16] else FPZero('0',
                       \rightarrow16));
24
                  bits(16) ecol_0 = (if pcol_0 then Elem[operanc'
                                                                            2 \star col + 0,
                                                                                         16] else FPZero('0',
                       \rightarrow16));
25
                  bits(16) ecol_1 = (if pcol_1 then Elem[opr and2, +col + 1, 6] else FPZero('0',
                       →16));
                  if sub_op then
26
                  if prow_0 then erow_0 = FPNeg(er __0);
if prow_1 then erow_1 = FPNeg(er __1);
sum = FPDotAdd_ZA(sum, erow_0, erow_1,
27
28
                                                                1);
                                                                     __0, eco' _, FPCR[]);
29
30
31
              Elem[result, row*dim+col, 32] =
                                                   am;
32
    ZAtile[da, 32, dim*dim*32] = result;
33
```

D1.1.58 FMOPS (non-widening)

Floating-point outer product and subtract

The single-precision variant works with a 32-bit element ZA tile.

The double-precision variant works with a 64-bit element ZA tile.

These instructions generate an outer product of the first source vector and the second source vector. In case of the single-precision variant, the first source is $SVL_S \times 1$ vector and the second source is $1 \times SVL_S$ vector. In case of the double-precision variant, the first source is $SVL_D \times 1$ vector and the second source is $1 \times SVL_D$ vector.

Each source vector is independently predicated by a corresponding governing predicate. When either source vector element is Inactive the corresponding destination tile element remains unmodified.

The resulting outer product, $SVL_S \times SVL_S$ in case of single-precision \uparrow or $SVL_D \times SVL_D$ in case of double-precision variant, is then destructively subtracted from the destine on tile. It is equivalent to performing a single multiply-subtract from each of the destination tile elements.

This instruction follows SME ZA-targeting floating-point behaviors.

ID_AA64SMFR0_EL1.F64F64 indicates whether the doub' precision varia. is .plemented.

It has encodings from 2 classes: Single-precision and Dou. --pre ion

Single-precision (FEAT_SME)



FMOPS <ZAda>.D, <Pn>/M, <Pm>/M, <Zn>.D, <Zm>.D

- 1 if !HaveSMEF64F64() then UNDEFINED;
- 2 constant **integer** esize = 64;
- 3 integer a = UInt(Pn);
- 4 integer b = UInt(Pm);
- 5 integer n = UInt(Zn); 6 integer m = UInt(Zm);
- 7 integer da = UInt(ZAda);
- 8 boolean sub_op = TRUE;

```
DDI0616
B.a
```

Assembler Symbols

<ZAda> For the single-precision variant: is the name of the ZA tile ZA0-ZA3, encoded in the "ZAda" field.

For the double-precision variant: is the name of the ZA tile ZA0-ZA7, encoded in the "ZAda" field.

- <Pn> Is the name of the first governing scalable predicate register P0-P7, encoded in the "Pn" field.
- <Pm> Is the name of the second governing scalable predicate register P0-P7, encoded in the "Pm" field.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

```
CheckStreamingSVEAndZAEnabled();
1
   constant integer VL = CurrentVL;
2
   constant integer PL = VL DIV 8;
3
Δ
   constant integer dim = VL DIV esize;
   bits(PL) mask1 = P[a, PL];
5
   bits(PL) mask2 = P[b, PL];
6
7
   bits(VL) operand1 = Z[n, VL];
   bits(VL) operand2 = Z[m, VL];
8
9
   bits(dim*dim*esize) operand3 = ZAtile[d
                                               esiz€
                                                       dim*a.
                                                                   2e1;
10
   bits(dim*dim*esize) result;
11
12
   for row = 0 to dim-1
13
        for col = 0 to dim-1
14
            bits(esize) element1
                                    Elen
                                           peranc
                                                    row, esize];
15
            bits(esize) element<sup>2</sup>
                                    Elem[o, rand2, ol, esize];
16
                                  = Elem[ope hd3, row*dim+col, esize];
            bits (esize) elemer
17
18
            if (ActivePred_ateEle nt(mask1, row, esize) &&
19
                  Active redicateEle nt(mask2, col, esize)) then
20
                if sub p then element
                                          FPNeg(element1);
                     esult, row*dim+col, esize] = FPMulAdd_ZA(element3, element1, element2,
21
                Eler
                      .
→FPCR/
22
            else
23
                         .it, row _m+col, esize] = element3;
                Elem
24
25
    ZAtile
           .a, es
                  ie, din 'ir .size] = result;
```

D1.1.59 FRINTA

Multi-vector floating-point round to integral value, to nearest with ties away from zero

Round to the nearest integral floating-point value, with ties rounding away from zero, each element of the two or four source vectors, and place the results in the corresponding elements of the two or four destination vectors.

This instruction follows SME2 floating-point numerical behaviors corresponding to instructions that place their results in one or more SVE Z vectors.

This instruction is unpredicated.

It has encodings from 2 classes: Two registers and Four registers

Two registers (FEAT_SME2)



```
FRINTA { <Zd1>.S-<Zd2>.S }, { <Zn1>.S-<Zn2>.
```

- 1 if !HaveSME2() then UNDEFINED;
- 2 integer n = UInt(Zn:'0');
- 3 integer d = UInt(Zd:'0');
- 4 constant integer nreg = 2;
- 5 boolean exact = FALSE;
- 6 FPRounding rounding = FPRounding_ TAWAY,

Four registers (FEAT_SME2)



FRINTA / 1>.S 7d4>.S / { <Zn1>.S-<Zn4>.S }

```
1 if / .veSME2 / then UNL .INED;
```

- 2 int r n = '...'00');
- 3 integ d UInt(Zu. 00');
- 4 constan nteger nreg = 4;
- 5 boolean e. rt = FALSE;
- 6 FPRounding __unding = FPRounding_TIEAWAY;

Assembler Symbols

<Zd1> For the two registers variant: is the name of the first destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 2.

For the four registers variant: is the name of the first destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 4.

- <Zd4> Is the name of the fourth destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 4 plus 3.
- <Zd2> Is the name of the second destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 2 plus 1.
- <Zn1> For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.

For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

- <Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.
- <Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.

```
CheckStreamingSVEEnabled();
1
2
   constant integer VL = CurrentVL;
3
   constant integer elements = VL DIV 32;
4
   array [0..3] of bits(VL) results;
5
6
   for r = 0 to nreg-1
7
       bits(VL) operand = Z[n+r, VL];
       for e = 0 to elements-1
8
           bits(32) element = Elem[operand, e, 32];
9
                                                             PCR[]
10
            Elem[results[r], e, 32] = FPRoundInt(element
                                                                     ounding
                                                                               exact);
11
   for r = 0 to nreg-1
12
13
      Z[d+r, VL] = results[r];
```

D1.1.60 FRINTM

Multi-vector floating-point round to integral value, toward minus Infinity

Round down to an integral floating-point value, each element of the two or four source vectors, and place the results in the corresponding elements of the two or four destination vectors.

This instruction follows SME2 floating-point numerical behaviors corresponding to instructions that place their results in one or more SVE Z vectors.

This instruction is unpredicated.

It has encodings from 2 classes: Two registers and Four registers

Two registers (FEAT_SME2)



FRINTM { <Zd1>.S-<Zd2>.S }, { <Zn1>.S-<Zn2>.

- 1 if !HaveSME2() then UNDEFINED;
- 2 integer n = UInt(Zn:'0');
- 3 integer d = UInt(Zd:'0');
- 4 constant integer nreg = 2;
- 5 boolean exact = FALSE;
- 6 FPRounding rounding = FPRounding_ GINF;

Four registers (FEAT_SME2)



FRINTM / 1>.S 7d4>.S / { <Zn1>.S-<Zn4>.S }

```
1 if / .veSME2/ then UNL .INED;
```

- 2 int r n = '00');
- 3 integ d UInt(Zu. 00');
- 4 constan nteger nreg = 4;
- 5 boolean e. rt = FALSE;
- 6 FPRounding ______ unding = FPRounding_NEGINF;

Assembler Symbols

<Zd1> For the two registers variant: is the name of the first destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 2.

For the four registers variant: is the name of the first destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 4.

- <Zd4> Is the name of the fourth destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 4 plus 3.
- <Zd2> Is the name of the second destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 2 plus 1.
- <Zn1> For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.

For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

- <Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.
- <Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.

```
CheckStreamingSVEEnabled();
1
2
   constant integer VL = CurrentVL;
3
   constant integer elements = VL DIV 32;
4
   array [0..3] of bits(VL) results;
5
6
   for r = 0 to nreg-1
7
       bits(VL) operand = Z[n+r, VL];
       for e = 0 to elements-1
8
           bits(32) element = Elem[operand, e, 32];
9
                                                             PCR[]
10
            Elem[results[r], e, 32] = FPRoundInt(element
                                                                     ounding
                                                                               exact);
11
   for r = 0 to nreg-1
12
13
      Z[d+r, VL] = results[r];
```
D1.1.61 FRINTN

Multi-vector floating-point round to integral value, to nearest with ties to even

Round to the nearest integral floating-point value, with ties rounding to an even value, each element of the two or four source vectors, and place the results in the corresponding elements of the two or four destination vectors.

This instruction follows SME2 floating-point numerical behaviors corresponding to instructions that place their results in one or more SVE Z vectors.

This instruction is unpredicated.

It has encodings from 2 classes: Two registers and Four registers

Two registers (FEAT_SME2)



FRINTN { <Zd1>.S-<Zd2>.S }, { <Zn1>.S-<Zn2>.

- 1 if !HaveSME2() then UNDEFINED;
- 2 integer n = UInt(Zn:'0');
- 3 integer d = UInt(Zd:'0');
- 4 constant integer nreg = 2;
- 5 boolean exact = FALSE;
- 6 FPRounding rounding = FPRounding_ TEVEN;

Four registers (FEAT_SME2)



FRINTN / 1>.S 7d4>.S / { <Zn1>.S-<Zn4>.S }

```
1 if / .veSME2 / then UNL .INED;
```

- 2 int r n = '00');
- 3 integ d UInt(Zu. 00');
- 4 constan nteger nreg = 4;
- 5 boolean e. rt = FALSE;
- 6 FPRounding __unding = FPRounding_TIEEVEN;

Assembler Symbols

<Zd1> For the two registers variant: is the name of the first destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 2.

For the four registers variant: is the name of the first destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 4.

- <Zd4> Is the name of the fourth destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 4 plus 3.
- <Zd2> Is the name of the second destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 2 plus 1.
- <Zn1> For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.

For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

- <Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.
- <Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.

```
CheckStreamingSVEEnabled();
1
2
   constant integer VL = CurrentVL;
3
   constant integer elements = VL DIV 32;
4
   array [0..3] of bits(VL) results;
5
6
   for r = 0 to nreg-1
7
       bits(VL) operand = Z[n+r, VL];
       for e = 0 to elements-1
8
           bits(32) element = Elem[operand, e, 32];
9
                                                             PCR[]
10
            Elem[results[r], e, 32] = FPRoundInt(element
                                                                     ounding
                                                                               exact);
11
   for r = 0 to nreg-1
12
13
      Z[d+r, VL] = results[r];
```

D1.1.62 FRINTP

Multi-vector floating-point round to integral value, toward plus Infinity

Round up to an integral floating-point value, each element of the two or four source vectors, and place the results in the corresponding elements of the two or four destination vectors.

This instruction follows SME2 floating-point numerical behaviors corresponding to instructions that place their results in one or more SVE Z vectors.

This instruction is unpredicated.

It has encodings from 2 classes: Two registers and Four registers

Two registers (FEAT_SME2)



FRINTP { <Zd1>.S-<Zd2>.S }, { <Zn1>.S-<Zn2>.

- 1 if !HaveSME2() then UNDEFINED;
- 2 integer n = UInt(Zn:'0');
- 3 integer d = UInt(Zd:'0');
- 4 constant integer nreg = 2;
- 5 boolean exact = FALSE;
- 6 FPRounding rounding = FPRounding_ SINF;

Four registers (FEAT_SME2)

0 0 0 1 1 0 0 1 0 0 1 0 0 1 1 1 size<1>⅃ Lsize<

FRINTP '1>.5 7d4>.5 (<Zn1>.S-<Zn4>.S }

```
1 if / .veSME2/ then UNL .INED;
```

- 2 int r n = ''00');
- 3 integ d UInt(Zu. 00');
- 4 constan **nteger** nreg = 4;
- 5 boolean e. rt = FALSE;
- 6 FPRounding _____nding = FPRounding_POSINF;

Assembler Symbols

<Zd1> For the two registers variant: is the name of the first destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 2.

For the four registers variant: is the name of the first destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 4.

- <Zd4> Is the name of the fourth destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 4 plus 3.
- <Zd2> Is the name of the second destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 2 plus 1.
- <Zn1> For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.

For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

- <Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.
- <Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.

```
CheckStreamingSVEEnabled();
1
2
   constant integer VL = CurrentVL;
3
   constant integer elements = VL DIV 32;
4
   array [0..3] of bits(VL) results;
5
6
   for r = 0 to nreg-1
7
       bits(VL) operand = Z[n+r, VL];
       for e = 0 to elements-1
8
           bits(32) element = Elem[operand, e, 32];
9
                                                             PCR[]
10
            Elem[results[r], e, 32] = FPRoundInt(element
                                                                     ounding
                                                                               exact);
11
   for r = 0 to nreg-1
12
13
      Z[d+r, VL] = results[r];
```

D1.1.63 FSUB

Floating-point subtract multi-vector from ZA array vector accumulators

The instruction operates on two or four ZA single-vector groups.

Destructively subtract all elements of the two or four source vectors from the corresponding elements of the two or four ZA single-vector groups. The vector numbers forming the single-vector group within each half or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction follows SME ZA-targeting floating-point behaviors.

This instruction is unpredicated.

ID_AA64SMFR0_EL1.F64F64 indicates whether the double-pr ision iant is imp mented.

It has encodings from 2 classes: Two ZA single-vectors and our ZA single-

Two ZA single-vectors (FEAT_SME2)



VGx2 FSUB 7A. <T > [< Wv >. < off<T>-<Zm2>.<T>

```
1
   if !HaveSME2() then UNP INEL
2
   if sz == '1' && !HaveSMEF64F64, then UNDEFINED;
   integer v = UInt('0
3
                             ':Rv);
   constant integer ize = 32 << UInt z);</pre>
4
   integer m = UIr _2m:'0')
integer offset UInt' .f3);
constant intege. re = 2;
5
6
```

Four 7 singly vectors (FF ___SME2

> 0 0 0 0 0 1 1 1 0 0 0 0

FSUB ZA.<T>[<Wv>, <offs>{, VGx4}], { <Zm1>.<T>-<Zm4>.<T> }

```
if !HaveSME2() then UNDEFINED;
1
2
  if sz == '1' && !HaveSMEF64F64() then UNDEFINED;
  integer v = UInt('010':Rv);
3
4
  constant integer esize = 32 << UInt(sz);</pre>
5
  integer m = UInt(Zm:'00');
6
  integer offset = UInt(off3);
  constant integer nreg = 4;
```

Assembler Symbols

<T> Is the size specifier, encoded in "sz":

SZ	<t></t>
0	S
1	D

- <Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
- <offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.
- <Zm1> For the two ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 2.

For the four ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 4.

- <Zm4> Is the name of the fourth scalable vector register of a multi-vector rence, encoded as "Zm" times 4 plus 3.
- <Zm2> Is the name of the second scalable vector register of a multi-ctor sequence encoded as "Zm" times 2 plus 1.

```
CheckStreamingSVEAndZAEnabled();
 1
    constant integer VL = CurrentVL;
2
 3
    constant integer elements = VL DIV esize;
    integer vectors = VL DIV 8;
 4
    integer vstride = vectors DIV nreg;
5
 6
    bits(32) vbase = X[v, 32];
    integer vec = (UInt(vbase) + offsot) Mo.
 7
                                                               ide;
8
    bits(VL) result;
0
    for r = 0 to nreg-1
10
11
         bits(VL) operand1 = Z<sup>7</sup>
                                        rtor[vec,
         bits(VL) operand2 = __m+
12
                                            VL];
         for e = 0 to elemen.s-1
13
              bits(esize) lement1 = L m[operand1, e, esize];
bits(esize element2 = Ele operand2, e, esize];
Elem[res_tt, e, esize] = FPsub_ZA(element1, element2, FPCR[]);
ector[v_, VL] = lesult;
14
15
16
17
         ZAvector[, , VL] =
18
         vec = vec
                           str: .;
```

D1.1.64 FVDOT

Multi-vector half-precision floating-point vertical dot-product by indexed element

The instruction operates on two ZA single-vector groups.

The instruction computes the vertical fused sum-of-products of the corresponding half-precision floating-point values held in the two first source vectors with pair of half-precision floating-point values in the indexed 32-bit element of the second source vector, without intermediate rounding. The single-precision sum-of-products results are destructively added to the corresponding single-precision elements of the two ZA single-vector groups.

The half-precision floating-point pairs within the second source vector are specified using an immediate index which selects the same pair position within each 128-bit vector segment. The element index range is from 0 to 3.

The vector numbers forming the single-vector group within each half of the 7A array are selected by the sum of the vector select register and immediate offset, modulo half the number c = A are vectors.

The VECTOR GROUP symbol VGx2 indicates that the ZA operand c sists of two Z₄ single-vector groups. The VECTOR GROUP symbol is preferred for disassembly, but optional in a mbler source code.

This instruction follows SME ZA-targeting floating-point be' viors.

This instruction is unpredicated.

SME2 (FEAT_SME2)

```
13 | 12
       0
          0
              0
                  0
                                                                                     Zn
                                                                                               0
                                                                                                   0
1
   0
                      1
                          0
                              1
                                 0
                                               Zm
                                                                    0
                                                                          i2
                                                                                                       1
                                                                                                             off3
                                      1
```

FVDOT ZA.S[<Wv>, <offs> VGx2}], { Zn1>..-<Zn2>.H }, <Zm>.H[<index>]

```
1 if !HaveSME2() then UNL FINED,
```

```
2 integer v = UInt('01 ':Rv);
```

```
3 integer n = UInt(7 '0');
4 integer m = UInt 0':Zm);
```

```
5 integer offset UInt(c .3);
```

```
6 integer index - 'Int'.);
```

Assemb' . sy. 'ols

```
<Wv> Is the 3' oit name of the vector select register W8-W11, encoded in the "Rv" field.
```

```
<offs> t' vector select offset, in the range 0 to 7, encoded in the "off3" field.
```

- <Zn1> Is the mame of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2
- <Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.
- <Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

<index> Is the element index, in the range 0 to 3, encoded in the "i2" field.

```
1 CheckStreamingSVEAndZAEnabled();
2 constant integer VL = CurrentVL;
3 constant integer elements = VL DIV 32;
4 integer vectors = VL DIV 8;
5 integer vstride = vectors DIV 2;
6 integer eltspersegment = 128 DIV 32;
7 bits(32) vbase = X[v, 32];
8 integer vec = (UInt(vbase) + offset) MOD vstride;
```

```
Chapter D1. SME instructions
D1.1. SME and SME2 data-processing instructions
```

```
bits(VL) result;
9
10
11
    for r = 0 to 1
12
         bits(VL) operand1a = Z[n, VL];
         bits(VL) operand1b = Z[n+1, VL];
13
         bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = ZAvector[vec, VL];
14
15
16
         for e = 0 to elements-1
              integer segmentbase = e - (e MOD eltspersegment);
integer s = segmentbase + index;
17
18
19
              bits(16) elt1_a = Elem[operand1a, 2 * e + r, 16];
              bits(16) elt1_b = Elem[operand1b, 2 * e + r, 16];
bits(16) elt2_a = Elem[operand2, 2 * s + 0, 16];
20
21
              bits(16) elt2_b = Elem[operand2, 2 * s + 1, 16];
22
23
              bits(32) sum = Elem[operand3, e, 32];
24
              sum = FPDotAdd_ZA(sum, elt1_a, elt1_b, elt2_a, elt2_b, FPCR[]);
25
              Elem[result, e, 32] = sum;
         ZAvector[vec, VL] = result;
26
27
         vec = vec + vstride;
```

D1.1.65 LD1B (scalar plus immediate, consecutive registers)

Contiguous load of bytes to multiple consecutive vectors (immediate index)

Contiguous load of unsigned bytes to elements of two or four consecutive vector registers from the memory address generated by a 64-bit scalar base and immediate index which is multiplied by the vector's in-memory size, irrespective of predication, and 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.

It has encodings from 2 classes: Two registers and Four registers

Two registers (FEAT_SME2)



For the four registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 4.

- <Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" times 4 plus 3.
- <Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" times 2 plus 1.
- <PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter

encoding, encoded in the "PNg" 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 two registers variant: is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.

For the four registers variant: is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.

```
1
   CheckStreamingSVEEnabled();
2
   constant integer VL = CurrentVL;
   constant integer PL = VL DIV 8;
3
4
   constant integer elements = VL DIV esize;
5
   constant integer mbytes = esize DIV 8;
6
   bits(64) base;
   bits(PL) pred = P[g, PL];
7
   bits(PL * nreg) mask = CounterToPredicate(pred<15:0>)
8
                                                                ni
                                                            Ť.
9
   array [0..3] of bits(VL) values;
10 boolean contiguous = TRUE;
11
   boolean nontemporal = FALSE;
12
   boolean tagchecked = n != 31;
13
   AccessDescriptor accdesc = CreateAccDescSVF ("_mOp_L
                                                             nonte
                                                                    oral, contiguous, tagchecked);
14
15
   if !AnyActiveElement(mask, esize) then
       if n == 31 && ConstrainUnpredict? eBool(U predictable_CHECKSPNONEACTIVE) then
16
17
            CheckSPAlignment();
18
   else
19
       if n == 31 then CheckSPAlignme ();
20
       base = if n == 31 then SP
                                    else
                                           ſn,
21
22
   for r = 0 to nreg-1
23
        for e = 0 to elemer
                             -1
24
            if ActivePredicateElem. (mask, r * elements + e, esize) then
                                       (offset * nreg * elements + r * elements + e) * mbytes;
25
                bits(6<sup>4</sup> addr = base
                Elem [ lues[r], e, esi. = Mem[addr, mbytes, accdesc];
26
27
            else
28
                ΕŚ
                    [value r], e, esize] = Zeros(esize);
29
30
   for r = 0 -- nreg
31
       Z[t
                   = va. s[r
```

D1.1.66 LD1B (scalar plus scalar, consecutive registers)

Contiguous load of bytes to multiple consecutive vectors (scalar index)

Contiguous load of unsigned bytes to elements of two or four consecutive vector registers from the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 2 classes: Two registers and Four registers

Two registers (FEAT_SME2)



<Zt1> For the two registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 2.

For the four registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 4.

- <Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" times 4 plus 3.
- <Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" times 2 plus 1.
- <PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter

encoding, encoded in the "PNg" field.

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

```
1
    CheckStreamingSVEEnabled();
   constant integer VL = CurrentVL;
 2
 3
    constant integer PL = VL DIV 8;
 4
    constant integer elements = VL DIV esize;
    constant integer mbytes = esize DIV 8;
 5
 6 bits(64) offset;
    bits(64) base;
 7
   bits(PL) pred = P[g, PL];
 8
   bits(PL * nreg) mask = CounterToPredicate(pred<15:0>, PL *
 9
                                                                       eα);
10
   array [0..3] of bits(VL) values;
11
    boolean contiguous = TRUE;
12 boolean nontemporal = FALSE;
13
    boolean tagchecked = TRUE;
14
    AccessDescriptor accdesc = CreateAccDescSVE (MemOp _ JAD, nontempo _ 1
                                                                               ontiguous, tagchecked);
15
16
    if !AnyActiveElement(mask, esize) then
                                                               able_CHF SPNONEACTIVE) then
17
        if n == 31 && ConstrainUnpredictableBool(Unpre.
18
            CheckSPAlignment();
19
    else
20
        if n == 31 then CheckSPAlignment()
        base = if n == 31 then SP[] else
21
                                              ٦,
                                                 641
22
        offset = X[m, 64];
23
24
    for r = 0 to nreg-1
25
        for e = 0 to elements-1
                                ement(mask, * elements + e, esize) then
= se + (UInt( fset) + r * elements + e) * mbytes;
26
            if ActivePredicate
27
                 bits(64) adc'
                 Elem[values,r], e, size] = Mem[addr, mbytes, accdesc];
28
29
             else
30
                 Elem[y
                        .ues[r], e, es.
                                           1 = Zeros(esize);
31
    for r = 0 to r .g-1
    Z[t+r, VL] valv
32
33
                             [r];
```

D1.1.67 LD1B (scalar plus immediate, strided registers)

Contiguous load of bytes to multiple strided vectors (immediate index)

Contiguous load of unsigned bytes to elements of two or four strided vector registers from the memory address generated by a 64-bit scalar base and immediate index which is multiplied by the vector's in-memory size, irrespective of predication, and 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.

It has encodings from 2 classes: Two registers and Four registers

Two registers (FEAT_SME2)



Assembler Symbols

<Zt1> For the two registers variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as "T:'0':Zt".

For the four registers variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as "T:'00':Zt".

<Zt2> For the two registers variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as "T:'1':Zt".

For the four registers variant: is the name of the second scalable vector register Z4-Z7 or Z20-Z23 to be transferred, encoded as "T:'01':Zt".

- <Zt3> Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as "T:'10':Zt".
- <Zt4> Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as "T:'11':Zt".
- <PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" 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 two registers variant: is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.

For the four registers variant: is the optional signed immediate vector freet, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" fiet.

```
CheckStreamingSVEEnabled();
 1
   constant integer VL = CurrentVL;
2
3
   constant integer PL = VL DIV 8;
4
   constant integer elements = VL DIV esize;
5
   constant integer mbytes = esize DIV 8;
 6
   bits(64) base;
   bits(PL) pred = P[g, PL];
7
8
   bits(PL * nreg) mask = CounterToPredic
                                              _e(pred 5:0>, PL
                                                                    nreg);
 9
   array [0..3] of bits(VL) values;
10 boolean contiguous = TRUE;
   boolean nontemporal = FALSE;
11
   boolean tagchecked = n != 31;
12
13
   AccessDescriptor accdesc = C .ateAccL
                                               cSVE(N_Op_LOAD, nontemporal, contiguous, tagchecked);
14
15
   if !AnyActiveElement(mas 6 ze) then
        if n == 31 && Const ainUnp dictableBool(Unpredictable_CHECKSPNONEACTIVE) then
16
17
            CheckSPAligr ont();
18
    else
19
        if n == 31 t' n CheckSPAlignmen...);
20
        base = if _== 31 t' .n SP[] else X[n, 64];
21
22
    for r = 0 to nrey
23
                  to e. rents-1
        for
                   vePre at lement(mask, r * elements + e, esize) then
s(64) a. = base + (offset * nreg * elements + r * elements + e) * mbytes;
24
            1f Ac
25
                t :s(64) a.
26
                       ralues[r], e, esize] = Mem[addr, mbytes, accdesc];
27
28
                 Elem[values[r], e, esize] = Zeros(esize);
29
   for r = 0 t nreg-1
    Z[t, VL] = values[r];
30
31
32
        t = t + tstride;
```

D1.1.68 LD1B (scalar plus scalar, strided registers)

Contiguous load of bytes to multiple strided vectors (scalar index)

Contiguous load of unsigned bytes to elements of two or four strided vector registers from the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 2 classes: Two registers and Four registers

Two registers (FEAT_SME2)



Assembler Symbols

<Zt1> For the two registers variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as "T:'0':Zt".

For the four registers variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as "T:'00':Zt".

<Zt2> For the two registers variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as "T:'1':Zt".

For the four registers variant: is the name of the second scalable vector register Z4-Z7 or Z20-Z23 to be transferred, encoded as "T:'01':Zt".

- <Zt3> Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as "T:'10':Zt".
- <Zt4> Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as "T:'11':Zt".
- <PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
 - <Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

```
1 CheckStreamingSVEEnabled();
2
   constant integer VL = CurrentVL;
   constant integer PL = VL DIV 8;
3
   constant integer elements = VL DIV esize;
4
5
   constant integer mbytes = esize DIV 8;
6
   bits(64) offset;
   bits(64) base;
7
8 bits(PL) pred = P[g, PL];
   bits(PL * nreg) mask = CounterToPredicate(pred<15.</pre>
9
                                                                 nreg
10
   array [0..3] of bits(VL) values;
   boolean contiguous = TRUE;
11
12
   boolean nontemporal = FALSE;
13
   boolean tagchecked = TRUE;
14
   AccessDescriptor accdesc = CreateAccDe
                                              SVE /
                                                   nontemporal, contiguous, tagchecked);
15
16
   if !AnyActiveElement(mask, esize)
17
        if n == 31 && ConstrainUr edict 'eBoo predictable_CHECKSPNONEACTIVE) then
18
            CheckSPAlignment() ·
19
   else
20
        if n == 31 then Chr .SPAL
                                    nment();
21
        base = if n == 31 then SP[] lse X[n, 64];
22
        offset = X[m,
                       E i
23
   for r = 0 to nr _-1
for e = 0 eleme .s-1
    if Active rescateE
24
25
                      re_cateEler nt(mask, r * elements + e, esize) then
26
                   ts(\_ addr = _se + (UInt(offset) + r * elements + e
em[val s[r' e, esize] = Mem[addr, mbytes, accdesc];
27
                                  hits (
28
29
            else
30
                  .em[values,r], e, esize] = Zeros(esize);
31
32
   for r
              co nreg-1
             ] = values[r];
33
        Z[t,
34
        t = t
                 `stride;
```

D1.1.69 LD1B (scalar plus scalar, tile slice)

Contiguous load of bytes to 8-bit element ZA tile slice

The slice number within the tile is selected by the sum of the slice index register and immediate offset, modulo the number of 8-bit elements in a vector. The immediate offset is in the range 0 to 15. The memory address is generated by a 64-bit scalar base and an optional 64-bit scalar offset 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.

SME (FEAT_SME)



```
2
   constant integer VL = CurrentVL;
3
   constant integer PL = VL DIV 8;
4
   constant integer dim = VL DIV esize;
5
   bits(64) base;
6
   bits(64) addr;
7
   bits(PL) mask = P[g, PL];
8
   bits(64) moffs = X[m, 64];
   bits(32) index = X[s, 32];
9
10
   integer slice = (UInt(index) + offset) MOD dim;
11
   bits(VL) result;
   constant integer mbytes = esize DIV 8;
12
13 boolean contiguous = TRUE;
14
   boolean nontemporal = FALSE;
```

Chapter D1. SME instructions D1.1. SME and SME2 data-processing instructions

```
boolean tagchecked = TRUE;
15
16
   AccessDescriptor accdesc = CreateAccDescSME (MemOp_LOAD, nontemporal, contiguous, tagchecked);
17
18
   if n == 31 then
19
        if AnyActiveElement(mask, esize) ||
20
              ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
21
            CheckSPAlignment();
22
        base = SP[];
23
   else
24
        base = X[n, 64];
25
   for e = 0 to dim - 1
    addr = base + UInt(moffs) * mbytes;
26
27
28
        if ActivePredicateElement(mask, e, esize) then
29
            Elem[result, e, esize] = Mem[addr, mbytes, accdesc];
30
        else
        Elem[result, e, esize] = Zeros(esize);
moffs = moffs + 1;
31
32
33
34 ZAslice[t, esize, vertical, slice, VL] = result;
```

D1.1.70 LD1D (scalar plus immediate, consecutive registers)

Contiguous load of doublewords to multiple consecutive vectors (immediate index)

Contiguous load of unsigned doublewords to elements of two or four consecutive vector registers from the memory address generated by a 64-bit scalar base and immediate index which is multiplied by the vector's in-memory size, irrespective of predication, and 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.

It has encodings from 2 classes: Two registers and Four registers

Two registers (FEAT_SME2)



encoded as "Zt" times 4. <Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" times 4

- plus 3.
 <Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" times 2
 plus 1.
- <PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter

encoding, encoded in the "PNg" 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 two registers variant: is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.

For the four registers variant: is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.

```
1
   CheckStreamingSVEEnabled();
2
   constant integer VL = CurrentVL;
   constant integer PL = VL DIV 8;
3
4
   constant integer elements = VL DIV esize;
5
   constant integer mbytes = esize DIV 8;
6
   bits(64) base;
   bits(PL) pred = P[g, PL];
7
   bits(PL * nreg) mask = CounterToPredicate(pred<15:0>)
8
                                                                ni
                                                            Ť.
9
   array [0..3] of bits(VL) values;
10 boolean contiguous = TRUE;
11
   boolean nontemporal = FALSE;
12
   boolean tagchecked = n != 31;
13
   AccessDescriptor accdesc = CreateAccDescSVF ("_mOp_L
                                                             nonte
                                                                    oral, contiguous, tagchecked);
14
15
   if !AnyActiveElement(mask, esize) then
       if n == 31 && ConstrainUnpredict? eBool(U predictable_CHECKSPNONEACTIVE) then
16
17
            CheckSPAlignment();
18
   else
19
       if n == 31 then CheckSPAlignme ();
20
       base = if n == 31 then SP
                                    else
                                           ſn,
21
22
   for r = 0 to nreg-1
23
        for e = 0 to elemer
                             -1
24
            if ActivePredicateElem. (mask, r * elements + e, esize) then
                                       (offset * nreg * elements + r * elements + e) * mbytes;
25
                bits(6<sup>4</sup> addr = base
                Elem [ lues[r], e, esi. = Mem[addr, mbytes, accdesc];
26
27
            else
28
                ΕŚ
                    [value r], e, esize] = Zeros(esize);
29
30
   for r = 0 -- nreg
31
       Z[t
                   = va. s[r
```

D1.1.71 LD1D (scalar plus scalar, consecutive registers)

Contiguous load of doublewords to multiple consecutive vectors (scalar index)

Contiguous load of unsigned doublewords to elements of two or four consecutive vector registers from the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 2 classes: Two registers and Four registers

Two registers (FEAT_SME2)



```
encoded as "Zt" times 2.
```

For the four registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 4.

- <Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" times 4 plus 3.
- <Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" times 2 plus 1.
- <PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter

encoding, encoded in the "PNg" field.

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

```
1
    CheckStreamingSVEEnabled();
   constant integer VL = CurrentVL;
 2
 3
    constant integer PL = VL DIV 8;
 4
    constant integer elements = VL DIV esize;
    constant integer mbytes = esize DIV 8;
 5
 6 bits(64) offset;
    bits(64) base;
 7
   bits(PL) pred = P[g, PL];
 8
   bits(PL * nreg) mask = CounterToPredicate(pred<15:0>, PL *
 9
                                                                       eα);
10
   array [0..3] of bits(VL) values;
11
    boolean contiguous = TRUE;
12 boolean nontemporal = FALSE;
13
    boolean tagchecked = TRUE;
14
    AccessDescriptor accdesc = CreateAccDescSVE (MemOp _ JAD, nontempo _ 1
                                                                               ontiguous, tagchecked);
15
16
    if !AnyActiveElement(mask, esize) then
                                                               able_CHF SPNONEACTIVE) then
17
        if n == 31 && ConstrainUnpredictableBool(Unpre.
18
            CheckSPAlignment();
19
    else
20
        if n == 31 then CheckSPAlignment()
        base = if n == 31 then SP[] else
21
                                              ٦,
                                                 641
22
        offset = X[m, 64];
23
24
    for r = 0 to nreg-1
25
        for e = 0 to elements-1
                                ement(mask, * elements + e, esize) then
= se + (UInt( fset) + r * elements + e) * mbytes;
26
            if ActivePredicate
27
                 bits(64) adc'
                 Elem[values,r], e, size] = Mem[addr, mbytes, accdesc];
28
29
             else
30
                 Elem[y
                        .ues[r], e, es.
                                           1 = Zeros(esize);
31
    for r = 0 to r .g-1
    Z[t+r, VL] valv
32
33
                             [r];
```

D1.1.72 LD1D (scalar plus immediate, strided registers)

Contiguous load of doublewords to multiple strided vectors (immediate index)

Contiguous load of unsigned doublewords to elements of two or four strided vector registers from the memory address generated by a 64-bit scalar base and immediate index which is multiplied by the vector's in-memory size, irrespective of predication, and 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.

It has encodings from 2 classes: Two registers and Four registers

Two registers (FEAT_SME2)



Assembler Symbols

<Zt1> For the two registers variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as "T:'0':Zt".

For the four registers variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as "T:'00':Zt".

<Zt2> For the two registers variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as "T:'1':Zt".

For the four registers variant: is the name of the second scalable vector register Z4-Z7 or Z20-Z23 to be transferred, encoded as "T:'01':Zt".

- <Zt3> Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as "T:'10':Zt".
- <Zt4> Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as "T:'11':Zt".
- <PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" 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 two registers variant: is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.

For the four registers variant: is the optional signed immediate vector freet, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" fiet.

```
CheckStreamingSVEEnabled();
 1
   constant integer VL = CurrentVL;
2
3
   constant integer PL = VL DIV 8;
4
   constant integer elements = VL DIV esize;
5
   constant integer mbytes = esize DIV 8;
 6
   bits(64) base;
   bits(PL) pred = P[g, PL];
7
8
   bits(PL * nreg) mask = CounterToPredic
                                              _e(pred 5:0>, PL
                                                                    nreg);
 9
   array [0..3] of bits(VL) values;
10 boolean contiguous = TRUE;
   boolean nontemporal = FALSE;
11
   boolean tagchecked = n != 31;
12
13
   AccessDescriptor accdesc = C .ateAccL
                                               cSVE(N_Op_LOAD, nontemporal, contiguous, tagchecked);
14
15
   if !AnyActiveElement(mas 6 ze) then
        if n == 31 && Const ainUnp dictableBool(Unpredictable_CHECKSPNONEACTIVE) then
16
17
            CheckSPAligr ont();
18
    else
19
        if n == 31 t' n CheckSPAlignmen...);
20
        base = if _== 31 t' .n SP[] else X[n, 64];
21
22
    for r = 0 to nrey
23
                  to e. rents-1
        for
                   vePre at lement(mask, r * elements + e, esize) then
s(64) a. = base + (offset * nreg * elements + r * elements + e) * mbytes;
24
            1f Ac
25
                t :s(64) a.
26
                       ralues[r], e, esize] = Mem[addr, mbytes, accdesc];
27
28
                 Elem[values[r], e, esize] = Zeros(esize);
29
   for r = 0 t nreg-1
    Z[t, VL] = values[r];
30
31
32
        t = t + tstride;
```

D1.1.73 LD1D (scalar plus scalar, strided registers)

Contiguous load of doublewords to multiple strided vectors (scalar index)

Contiguous load of unsigned doublewords to elements of two or four strided vector registers from the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 2 classes: Two registers and Four registers

Two registers (FEAT_SME2)



Assembler Symbols

<Zt1> For the two registers variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as "T:'0':Zt".

For the four registers variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as "T:'00':Zt".

<Zt2> For the two registers variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as "T:'1':Zt".

For the four registers variant: is the name of the second scalable vector register Z4-Z7 or Z20-Z23 to be transferred, encoded as "T:'01':Zt".

- <Zt3> Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as "T:'10':Zt".
- <Zt4> Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as "T:'11':Zt".
- <PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
 - <Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

```
1 CheckStreamingSVEEnabled();
2
   constant integer VL = CurrentVL;
   constant integer PL = VL DIV 8;
3
   constant integer elements = VL DIV esize;
4
5
   constant integer mbytes = esize DIV 8;
6
   bits(64) offset;
   bits(64) base;
7
8 bits(PL) pred = P[g, PL];
   bits(PL * nreg) mask = CounterToPredicate(pred<15.</pre>
9
                                                                 nreg
10
   array [0..3] of bits(VL) values;
   boolean contiguous = TRUE;
11
12
   boolean nontemporal = FALSE;
13
   boolean tagchecked = TRUE;
14
   AccessDescriptor accdesc = CreateAccDe
                                              SVE /
                                                   nontemporal, contiguous, tagchecked);
15
16
   if !AnyActiveElement(mask, esize)
17
        if n == 31 && ConstrainUr edict 'eBoo predictable_CHECKSPNONEACTIVE) then
18
            CheckSPAlignment() ·
19
   else
20
        if n == 31 then Chr .SPAL
                                    nment();
21
        base = if n == 31 then SP[] lse X[n, 64];
22
        offset = X[m,
                       E i
23
   for r = 0 to nr _-1
for e = 0 eleme .s-1
    if Active rescateE
24
25
                      re_cateEler nt(mask, r * elements + e, esize) then
26
                   ts(\_ addr = _se + (UInt(offset) + r * elements + e
em[val s[r' e, esize] = Mem[addr, mbytes, accdesc];
27
                                  hits (
28
29
            else
30
                  .em[values,r], e, esize] = Zeros(esize);
31
32
   for r
              co nreg-1
             ] = values[r];
33
        Z[t,
34
        t = t
                 `stride;
```

D1.1.74 LD1D (scalar plus scalar, tile slice)

Contiguous load of doublewords to 64-bit element ZA tile slice

The slice number within the tile is selected by the sum of the slice index register and immediate offset, modulo the number of 64-bit elements in a vector. The immediate offset is in the range 0 to 1. The memory address is generated by a 64-bit scalar base and an optional 64-bit scalar offset which is multiplied by 8 and 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.

SME (FEAT_SME)



Assembler Symbols

<ZAt> Is the name of the Z[^] .ie. `0-ZA7 to be cessed, encoded in the "ZAt" field.

<HV> Is the horizontal "vertical shu indicator, encoded in "V":

 V
 <HV</th>

 0
 r

 1
 r

<Ws> , the 32 vit name c , slice index register W12-W15, encoded in the "Rs" field.

- <offs. Is the ______ offset, in the range 0 to 1, encoded in the "o1" field.
- <Pg> h. e name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 1-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
 - <Xm> Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the "Rm" field.

```
1
   CheckStreamingSVEAndZAEnabled();
2
   constant integer VL = CurrentVL;
3
   constant integer PL = VL DIV 8;
4
   constant integer dim = VL DIV esize;
5
   bits(64) base;
   bits(64) addr;
6
7
   bits(PL) mask = P[g, PL];
   bits(64) moffs = X[m, 64];
8
9 bits(32) index = X[s, 32];
10 integer slice = (UInt(index) + offset) MOD dim;
11
   bits(VL) result;
```

Chapter D1. SME instructions D1.1. SME and SME2 data-processing instructions

```
12 constant integer mbytes = esize DIV 8;
13 boolean contiguous = TRUE;
14
   boolean nontemporal = FALSE;
15 boolean tagchecked = TRUE;
   AccessDescriptor accdesc = CreateAccDescSME(MemOp_LOAD, nontemporal, contiguous, tagchecked);
16
17
   if n == 31 then
18
19
       if AnyActiveElement(mask, esize) ||
20
              ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
21
            CheckSPAlignment();
22
       base = SP[];
23
   else
24
       base = X[n, 64];
25
26
   for e = 0 to dim - 1
       addr = base + UInt(moffs) * mbytes;
27
28
       if ActivePredicateElement(mask, e, esize) then
29
           Elem[result, e, esize] = Mem[addr, mbytes, accdesc]
30
       else
31
           Elem[result, e, esize] = Zeros(esize);
32
       moffs = moffs + 1;
33
34
   ZAslice[t, esize, vertical, slice, VL] = result;
```

D1.1.75 LD1H (scalar plus immediate, consecutive registers)

Contiguous load of halfwords to multiple consecutive vectors (immediate index)

Contiguous load of unsigned halfwords to elements of two or four consecutive vector registers from the memory address generated by a 64-bit scalar base and immediate index which is multiplied by the vector's in-memory size, irrespective of predication, and 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.

It has encodings from 2 classes: Two registers and Four registers

Two registers (FEAT_SME2)



For the four registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 4.

- <Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" times 4 plus 3.
- <Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" times 2 plus 1.
- <PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter

encoding, encoded in the "PNg" 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 two registers variant: is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.

For the four registers variant: is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.

```
1
   CheckStreamingSVEEnabled();
2
   constant integer VL = CurrentVL;
   constant integer PL = VL DIV 8;
3
4
   constant integer elements = VL DIV esize;
5
   constant integer mbytes = esize DIV 8;
6
   bits(64) base;
   bits(PL) pred = P[g, PL];
7
   bits(PL * nreg) mask = CounterToPredicate(pred<15:0>)
8
                                                                ni
                                                            Ť.
9
   array [0..3] of bits(VL) values;
10 boolean contiguous = TRUE;
11
   boolean nontemporal = FALSE;
12
   boolean tagchecked = n != 31;
13
   AccessDescriptor accdesc = CreateAccDescSVF ("_mOp_L
                                                             nonte
                                                                    oral, contiguous, tagchecked);
14
15
   if !AnyActiveElement(mask, esize) then
       if n == 31 && ConstrainUnpredict? eBool(U predictable_CHECKSPNONEACTIVE) then
16
17
            CheckSPAlignment();
18
   else
19
       if n == 31 then CheckSPAlignme ();
20
       base = if n == 31 then SP
                                    else
                                           ſn,
21
22
   for r = 0 to nreg-1
23
        for e = 0 to elemer
                             -1
24
            if ActivePredicateElem. (mask, r * elements + e, esize) then
                                       (offset * nreg * elements + r * elements + e) * mbytes;
25
                bits(6<sup>4</sup> addr = base
                Elem [ lues[r], e, esi. = Mem[addr, mbytes, accdesc];
26
27
            else
28
                ΕŚ
                    [value r], e, esize] = Zeros(esize);
29
30
   for r = 0 -- nreg
31
       Z[t
                   = va. s[r
```

D1.1.76 LD1H (scalar plus scalar, consecutive registers)

Contiguous load of halfwords to multiple consecutive vectors (scalar index)

Contiguous load of unsigned halfwords to elements of two or four consecutive vector registers from the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 2 classes: Two registers and Four registers

Two registers (FEAT_SME2)



<Zt1> For the two registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 2.

For the four registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 4.

- <Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" times 4 plus 3.
- <Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" times 2 plus 1.
- <PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter

encoding, encoded in the "PNg" field.

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

```
1
    CheckStreamingSVEEnabled();
   constant integer VL = CurrentVL;
 2
 3
    constant integer PL = VL DIV 8;
 4
    constant integer elements = VL DIV esize;
    constant integer mbytes = esize DIV 8;
 5
 6 bits(64) offset;
    bits(64) base;
 7
   bits(PL) pred = P[g, PL];
 8
   bits(PL * nreg) mask = CounterToPredicate(pred<15:0>, PL *
 9
                                                                      eα);
10
   array [0..3] of bits(VL) values;
11
    boolean contiguous = TRUE;
12 boolean nontemporal = FALSE;
13
    boolean tagchecked = TRUE;
14
    AccessDescriptor accdesc = CreateAccDescSVE (MemOp _ JAD, nontempo _ 1
                                                                               ontiguous, tagchecked);
15
16
    if !AnyActiveElement(mask, esize) then
                                                               able_CHF SPNONEACTIVE) then
17
        if n == 31 && ConstrainUnpredictableBool(Unpre.
18
            CheckSPAlignment();
19
    else
20
        if n == 31 then CheckSPAlignment()
        base = if n == 31 then SP[] else
21
                                              ٦,
                                                 641
22
        offset = X[m, 64];
23
24
    for r = 0 to nreg-1
25
        for e = 0 to elements-1
                                ement(mask, * elements + e, esize) then
= se + (UInt( fset) + r * elements + e) * mbytes;
26
            if ActivePredicate
27
                 bits(64) adc'
                 Elem[values,r], e, size] = Mem[addr, mbytes, accdesc];
28
29
             else
30
                 Elem[y
                        .ues[r], e, es.
                                           1 = Zeros(esize);
31
    for r = 0 to r .g-1
    Z[t+r, VL] valv
32
33
                             [r];
```

D1.1.77 LD1H (scalar plus immediate, strided registers)

Contiguous load of halfwords to multiple strided vectors (immediate index)

Contiguous load of unsigned halfwords to elements of two or four strided vector registers from the memory address generated by a 64-bit scalar base and immediate index which is multiplied by the vector's in-memory size, irrespective of predication, and 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.

It has encodings from 2 classes: Two registers and Four registers

Two registers (FEAT_SME2)



Assembler Symbols

<Zt1> For the two registers variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as "T:'0':Zt".

For the four registers variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as "T:'00':Zt".

<Zt2> For the two registers variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as "T:'1':Zt".

For the four registers variant: is the name of the second scalable vector register Z4-Z7 or Z20-Z23 to be transferred, encoded as "T:'01':Zt".

- <Zt3> Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as "T:'10':Zt".
- <Zt4> Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as "T:'11':Zt".
- <PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" 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 two registers variant: is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.

For the four registers variant: is the optional signed immediate vector freet, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" fiet.

```
CheckStreamingSVEEnabled();
 1
   constant integer VL = CurrentVL;
2
3
   constant integer PL = VL DIV 8;
4
   constant integer elements = VL DIV esize;
5
   constant integer mbytes = esize DIV 8;
 6
   bits(64) base;
   bits(PL) pred = P[g, PL];
7
8
   bits(PL * nreg) mask = CounterToPredic
                                              _e(pred 5:0>, PL
                                                                    nreg);
 9
   array [0..3] of bits(VL) values;
10 boolean contiguous = TRUE;
   boolean nontemporal = FALSE;
11
   boolean tagchecked = n != 31;
12
13
   AccessDescriptor accdesc = C .ateAccL
                                               cSVE(N_Op_LOAD, nontemporal, contiguous, tagchecked);
14
15
   if !AnyActiveElement(mas 6 ze) then
        if n == 31 && Const ainUnp dictableBool(Unpredictable_CHECKSPNONEACTIVE) then
16
17
            CheckSPAligr ont();
18
    else
19
        if n == 31 t' n CheckSPAlignmen...);
20
        base = if _== 31 t' .n SP[] else X[n, 64];
21
22
    for r = 0 to nrey
23
                  to e. rents-1
        for
                   vePre at lement(mask, r * elements + e, esize) then
s(64) a. = base + (offset * nreg * elements + r * elements + e) * mbytes;
24
            1f Ac
25
                t :s(64) a.
26
                       ralues[r], e, esize] = Mem[addr, mbytes, accdesc];
27
28
                 Elem[values[r], e, esize] = Zeros(esize);
29
   for r = 0 t nreg-1
    Z[t, VL] = values[r];
30
31
32
        t = t + tstride;
```

D1.1.78 LD1H (scalar plus scalar, strided registers)

Contiguous load of halfwords to multiple strided vectors (scalar index)

Contiguous load of unsigned halfwords to elements of two or four strided vector registers from the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 2 classes: Two registers and Four registers

Two registers (FEAT_SME2)



Assembler Symbols

<Zt1> For the two registers variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as "T:'0':Zt".

For the four registers variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as "T:'00':Zt".

<Zt2> For the two registers variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as "T:'1':Zt".

For the four registers variant: is the name of the second scalable vector register Z4-Z7 or Z20-Z23 to be transferred, encoded as "T:'01':Zt".

- <Zt3> Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as "T:'10':Zt".
- <Zt4> Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as "T:'11':Zt".
- <PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
 - <Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

```
1 CheckStreamingSVEEnabled();
2
   constant integer VL = CurrentVL;
   constant integer PL = VL DIV 8;
3
   constant integer elements = VL DIV esize;
4
5
   constant integer mbytes = esize DIV 8;
6
   bits(64) offset;
   bits(64) base;
7
8 bits(PL) pred = P[g, PL];
   bits(PL * nreg) mask = CounterToPredicate(pred<15.</pre>
9
                                                                 nreg
10
   array [0..3] of bits(VL) values;
   boolean contiguous = TRUE;
11
12
   boolean nontemporal = FALSE;
13
   boolean tagchecked = TRUE;
14
   AccessDescriptor accdesc = CreateAccDe
                                              SVE /
                                                   nontemporal, contiguous, tagchecked);
15
16
   if !AnyActiveElement(mask, esize)
17
        if n == 31 && ConstrainUr edict 'eBoo predictable_CHECKSPNONEACTIVE) then
18
            CheckSPAlignment() ·
19
   else
20
        if n == 31 then Chr .SPAL
                                    nment();
21
        base = if n == 31 then SP[] lse X[n, 64];
22
        offset = X[m,
                       E i
23
   for r = 0 to nr _-1
for e = 0 eleme .s-1
    if Active rescateE
24
25
                      re_cateEler nt(mask, r * elements + e, esize) then
26
                   ts(\_ addr = _se + (UInt(offset) + r * elements + e
em[val s[r' e, esize] = Mem[addr, mbytes, accdesc];
27
                                  hits (
28
29
            else
30
                  .em[values,r], e, esize] = Zeros(esize);
31
32
   for r
              co nreg-1
             ] = values[r];
33
        Z[t,
34
        t = t
                 `stride;
```
D1.1.79 LD1H (scalar plus scalar, tile slice)

Contiguous load of halfwords to 16-bit element ZA tile slice

The slice number within the tile is selected by the sum of the slice index register and immediate offset, modulo the number of 16-bit elements in a vector. The immediate offset is in the range 0 to 7. The memory address is generated by a 64-bit scalar base and an optional 64-bit scalar offset which is multiplied by 2 and 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.

SME (FEAT_SME)



Assembler Symbols

<ZAt> Is the name of the Z[^] .ie. `0-ZA1 to be cessed, encoded in the "ZAt" field.

<HV> Is the horizontal "vertical shu indicator, encoded in "V":

 V
 <HV>

 0
 1

 1
 1

<Ws> , the 32 vit name c , slice index register W12-W15, encoded in the "Rs" field.

- <offs. Is the ______ offset, in the range 0 to 7, encoded in the "off3" field.
- <Pg> 1. e name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 1-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
 - <Xm> Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the "Rm" field.

```
1
   CheckStreamingSVEAndZAEnabled();
2
   constant integer VL = CurrentVL;
3
   constant integer PL = VL DIV 8;
4
   constant integer dim = VL DIV esize;
5
   bits(64) base;
   bits(64) addr;
6
7
   bits(PL) mask = P[g, PL];
   bits(64) moffs = X[m, 64];
8
9 bits(32) index = X[s, 32];
10 integer slice = (UInt(index) + offset) MOD dim;
11
   bits(VL) result;
```

Chapter D1. SME instructions D1.1. SME and SME2 data-processing instructions

```
12 constant integer mbytes = esize DIV 8;
13 boolean contiguous = TRUE;
14
   boolean nontemporal = FALSE;
15 boolean tagchecked = TRUE;
   AccessDescriptor accdesc = CreateAccDescSME(MemOp_LOAD, nontemporal, contiguous, tagchecked);
16
17
   if n == 31 then
18
19
       if AnyActiveElement(mask, esize) ||
20
              ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
21
            CheckSPAlignment();
22
       base = SP[];
23
   else
24
       base = X[n, 64];
25
26
   for e = 0 to dim - 1
       addr = base + UInt(moffs) * mbytes;
27
28
       if ActivePredicateElement(mask, e, esize) then
29
           Elem[result, e, esize] = Mem[addr, mbytes, accdesc]
30
       else
31
           Elem[result, e, esize] = Zeros(esize);
32
       moffs = moffs + 1;
33
34
   ZAslice[t, esize, vertical, slice, VL] = result;
```

D1.1.80 LD1Q

Contiguous load of quadwords to 128-bit element ZA tile slice

The slice number in the tile is selected by the slice index register, modulo the number of 128-bit elements in a Streaming SVE vector. The memory address is generated by scalar base and optional scalar offset which is multiplied by 16 and 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.

SME (FEAT_SME)



```
4
   constant integer dim = VL DIV esize;
5
   bits(64) base;
   bits(64) addr;
6
   bits(PL) mask = P[g, PL];
7
   bits(64) moffs = X[m, 64];
8
9
   bits(32) index = X[s, 32];
10
   integer slice = (UInt(index) + offset) MOD dim;
11 bits(VL) result;
12
   constant integer mbytes = esize DIV 8;
13
   boolean contiguous = TRUE;
```

Chapter D1. SME instructions D1.1. SME and SME2 data-processing instructions

```
14
   boolean nontemporal = FALSE;
15
   boolean tagchecked = TRUE;
16
   AccessDescriptor accdesc = CreateAccDescSME (MemOp_LOAD, nontemporal, contiguous, tagchecked);
17
18
   if n == 31 then
19
       if AnyActiveElement(mask, esize) ||
20
             ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
21
            CheckSPAlignment();
22
       base = SP[];
23
   else
24
       base = X[n, 64];
25
26
   for e = 0 to dim - 1
27
       addr = base + UInt(moffs) * mbytes;
       if ActivePredicateElement(mask, e, esize) then
28
29
           Elem[result, e, esize] = Mem[addr, mbytes, accdesc];
30
       else
31
           Elem[result, e, esize] = Zeros(esize);
32
       moffs = moffs + 1;
33
34
   ZAslice[t, esize, vertical, slice, VL] = result;
```

D1.1.81 LD1W (scalar plus immediate, consecutive registers)

Contiguous load of words to multiple consecutive vectors (immediate index)

Contiguous load of unsigned words to elements of two or four consecutive vector registers from the memory address generated by a 64-bit scalar base and immediate index which is multiplied by the vector's in-memory size, irrespective of predication, and 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.

It has encodings from 2 classes: Two registers and Four registers

Two registers (FEAT_SME2)



For the four registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 4.

- <Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" times 4 plus 3.
- <Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" times 2 plus 1.
- <PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter

encoding, encoded in the "PNg" 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 two registers variant: is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.

For the four registers variant: is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.

```
1
   CheckStreamingSVEEnabled();
2
   constant integer VL = CurrentVL;
   constant integer PL = VL DIV 8;
3
4
   constant integer elements = VL DIV esize;
5
   constant integer mbytes = esize DIV 8;
6
   bits(64) base;
   bits(PL) pred = P[g, PL];
7
   bits(PL * nreg) mask = CounterToPredicate(pred<15:0>)
8
                                                                ni
                                                            Ť.
9
   array [0..3] of bits(VL) values;
10 boolean contiguous = TRUE;
11
   boolean nontemporal = FALSE;
12
   boolean tagchecked = n != 31;
13
   AccessDescriptor accdesc = CreateAccDescSVF ("_mOp_L
                                                             nonte
                                                                    oral, contiguous, tagchecked);
14
15
   if !AnyActiveElement(mask, esize) then
       if n == 31 && ConstrainUnpredict? eBool(U predictable_CHECKSPNONEACTIVE) then
16
17
            CheckSPAlignment();
18
   else
19
       if n == 31 then CheckSPAlignme ();
20
       base = if n == 31 then SP
                                    else
                                           ſn,
21
22
   for r = 0 to nreg-1
23
        for e = 0 to elemer
                             -1
24
            if ActivePredicateElem. (mask, r * elements + e, esize) then
                                       (offset * nreg * elements + r * elements + e) * mbytes;
25
                bits(6<sup>4</sup> addr = base
                Elem [ lues[r], e, esi. = Mem[addr, mbytes, accdesc];
26
27
            else
28
                ΕŚ
                    [value r], e, esize] = Zeros(esize);
29
30
   for r = 0 -- nreg
31
       Z[t
                   = va. s[r
```

D1.1.82 LD1W (scalar plus scalar, consecutive registers)

Contiguous load of words to multiple consecutive vectors (scalar index)

Contiguous load of unsigned words to elements of two or four consecutive vector registers from the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 2 classes: Two registers and Four registers

Two registers (FEAT_SME2)



encoded as "Zt" times 2.

For the four registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 4.

- <Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" times 4 plus 3.
- <Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" times 2 plus 1.
- <PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter

encoding, encoded in the "PNg" field.

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

```
1
    CheckStreamingSVEEnabled();
   constant integer VL = CurrentVL;
 2
 3
    constant integer PL = VL DIV 8;
 4
    constant integer elements = VL DIV esize;
    constant integer mbytes = esize DIV 8;
 5
 6 bits(64) offset;
    bits(64) base;
 7
   bits(PL) pred = P[g, PL];
 8
   bits(PL * nreg) mask = CounterToPredicate(pred<15:0>, PL *
 9
                                                                       eα);
10
   array [0..3] of bits(VL) values;
11
    boolean contiguous = TRUE;
12 boolean nontemporal = FALSE;
13
    boolean tagchecked = TRUE;
14
    AccessDescriptor accdesc = CreateAccDescSVE (MemOp _ JAD, nontempo _ 1
                                                                               ontiguous, tagchecked);
15
16
    if !AnyActiveElement(mask, esize) then
                                                               able_CHF SPNONEACTIVE) then
17
        if n == 31 && ConstrainUnpredictableBool(Unpre.
18
            CheckSPAlignment();
19
    else
20
        if n == 31 then CheckSPAlignment()
        base = if n == 31 then SP[] else
21
                                              ٦,
                                                 641
22
        offset = X[m, 64];
23
24
    for r = 0 to nreg-1
25
        for e = 0 to elements-1
                                ement(mask, * elements + e, esize) then
= se + (UInt( fset) + r * elements + e) * mbytes;
26
            if ActivePredicate
27
                 bits(64) adc'
                 Elem[values,r], e, size] = Mem[addr, mbytes, accdesc];
28
29
             else
30
                 Elem[y
                        .ues[r], e, es.
                                           1 = Zeros(esize);
31
    for r = 0 to r .g-1
    Z[t+r, VL] valv
32
33
                             [r];
```

D1.1.83 LD1W (scalar plus immediate, strided registers)

Contiguous load of words to multiple strided vectors (immediate index)

Contiguous load of unsigned words to elements of two or four strided vector registers from the memory address generated by a 64-bit scalar base and immediate index which is multiplied by the vector's in-memory size, irrespective of predication, and 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.

It has encodings from 2 classes: Two registers and Four registers

Two registers (FEAT_SME2)



Assembler Symbols

<Zt1> For the two registers variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as "T:'0':Zt".

For the four registers variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as "T:'00':Zt".

<Zt2> For the two registers variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as "T:'1':Zt".

For the four registers variant: is the name of the second scalable vector register Z4-Z7 or Z20-Z23 to be transferred, encoded as "T:'01':Zt".

- <Zt3> Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as "T:'10':Zt".
- <Zt4> Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as "T:'11':Zt".
- <PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" 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 two registers variant: is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.

For the four registers variant: is the optional signed immediate vector freet, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" fiet.

```
CheckStreamingSVEEnabled();
 1
2
   constant integer VL = CurrentVL;
3
   constant integer PL = VL DIV 8;
4
   constant integer elements = VL DIV esize;
5
   constant integer mbytes = esize DIV 8;
 6
   bits(64) base;
   bits(PL) pred = P[g, PL];
7
8
   bits(PL * nreg) mask = CounterToPredic
                                              _e(pred 5:0>, PL
                                                                    nreg);
 9
   array [0..3] of bits(VL) values;
10 boolean contiguous = TRUE;
   boolean nontemporal = FALSE;
11
   boolean tagchecked = n != 31;
12
13
   AccessDescriptor accdesc = C .ateAccL
                                               cSVE(N_Op_LOAD, nontemporal, contiguous, tagchecked);
14
15
   if !AnyActiveElement(mas 6 ze) then
        if n == 31 && Const ainUnp dictableBool(Unpredictable_CHECKSPNONEACTIVE) then
16
17
            CheckSPAligr ont();
18
    else
19
        if n == 31 t' n CheckSPAlignmen...);
20
        base = if _== 31 t' .n SP[] else X[n, 64];
21
22
    for r = 0 to nrey
23
                  to e. rents-1
        for
                   vePre at lement(mask, r * elements + e, esize) then
s(64) a. = base + (offset * nreg * elements + r * elements + e) * mbytes;
24
            1f Ac
25
                t :s(64) a.
26
                       ralues[r], e, esize] = Mem[addr, mbytes, accdesc];
27
28
                 Elem[values[r], e, esize] = Zeros(esize);
29
   for r = 0 t nreg-1
    Z[t, VL] = values[r];
30
31
32
        t = t + tstride;
```

D1.1.84 LD1W (scalar plus scalar, strided registers)

Contiguous load of words to multiple strided vectors (scalar index)

Contiguous load of unsigned words to elements of two or four strided vector registers from the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 2 classes: Two registers and Four registers

Two registers (FEAT_SME2)



Assembler Symbols

<Zt1> For the two registers variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as "T:'0':Zt".

For the four registers variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as "T:'00':Zt".

<Zt2> For the two registers variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as "T:'1':Zt".

For the four registers variant: is the name of the second scalable vector register Z4-Z7 or Z20-Z23 to be transferred, encoded as "T:'01':Zt".

- <Zt3> Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as "T:'10':Zt".
- <Zt4> Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as "T:'11':Zt".
- <PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
 - <Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

```
1 CheckStreamingSVEEnabled();
2
   constant integer VL = CurrentVL;
   constant integer PL = VL DIV 8;
3
   constant integer elements = VL DIV esize;
4
5
   constant integer mbytes = esize DIV 8;
6
   bits(64) offset;
   bits(64) base;
7
8 bits(PL) pred = P[g, PL];
   bits(PL * nreg) mask = CounterToPredicate(pred<15.</pre>
9
                                                                 nreg
10
   array [0..3] of bits(VL) values;
   boolean contiguous = TRUE;
11
12
   boolean nontemporal = FALSE;
13
   boolean tagchecked = TRUE;
14
   AccessDescriptor accdesc = CreateAccDe
                                              SVE /
                                                   nontemporal, contiguous, tagchecked);
15
16
   if !AnyActiveElement(mask, esize)
17
        if n == 31 && ConstrainUr edict 'eBoo predictable_CHECKSPNONEACTIVE) then
18
            CheckSPAlignment() ·
19
   else
20
        if n == 31 then Chr .SPAL
                                    nment();
21
        base = if n == 31 then SP[] lse X[n, 64];
22
        offset = X[m,
                       E i
23
   for r = 0 to nr _-1
for e = 0 eleme .s-1
    if Active rescateE
24
25
                      re_cateEler nt(mask, r * elements + e, esize) then
26
                   ts(\ addr = _se + (UInt(offset) + r * elements + e
em[val s[r' e, esize] = Mem[addr, mbytes, accdesc];
27
                                  hits (
28
29
            else
30
                  .em[values,r], e, esize] = Zeros(esize);
31
32
   for r
              co nreg-1
             ] = values[r];
33
        Z[t,
34
        t = t
                `stride;
```

D1.1.85 LD1W (scalar plus scalar, tile slice)

Contiguous load of words to 32-bit element ZA tile slice

The slice number within the tile is selected by the sum of the slice index register and immediate offset, modulo the number of 32-bit elements in a vector. The immediate offset is in the range 0 to 3. The memory address is generated by a 64-bit scalar base and an optional 64-bit scalar offset which is multiplied by 4 and 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.

SME (FEAT_SME)



Assembler Symbols

<ZAt> Is the name of the Z[^] .ie. `0-ZA3 to be cessed, encoded in the "ZAt" field.

<HV> Is the horizontal "vertical shu indicator, encoded in "V":

 V
 <HV^</th>

 0
 I

 1
 I

<Ws> , the 32 vit name c , slice index register W12-W15, encoded in the "Rs" field.

- <offs> Is the ______ offset, in the range 0 to 3, encoded in the "off2" field.
- <Pg> h. e name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 1-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
 - <Xm> Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the "Rm" field.

```
1
   CheckStreamingSVEAndZAEnabled();
2
   constant integer VL = CurrentVL;
3
   constant integer PL = VL DIV 8;
4
   constant integer dim = VL DIV esize;
5
   bits(64) base;
   bits(64) addr;
6
7
   bits(PL) mask = P[g, PL];
   bits(64) moffs = X[m, 64];
8
9 bits(32) index = X[s, 32];
10 integer slice = (UInt(index) + offset) MOD dim;
11
   bits(VL) result;
```

Chapter D1. SME instructions D1.1. SME and SME2 data-processing instructions

```
constant integer mbytes = esize DIV 8;
12
13 boolean contiguous = TRUE;
14
   boolean nontemporal = FALSE;
15 boolean tagchecked = TRUE;
   AccessDescriptor accdesc = CreateAccDescSME(MemOp_LOAD, nontemporal, contiguous, tagchecked);
16
17
   if n == 31 then
18
19
       if AnyActiveElement(mask, esize) ||
20
              ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
21
            CheckSPAlignment();
22
       base = SP[];
23
   else
24
       base = X[n, 64];
25
26
   for e = 0 to dim - 1
       addr = base + UInt(moffs) * mbytes;
27
28
       if ActivePredicateElement(mask, e, esize) then
29
           Elem[result, e, esize] = Mem[addr, mbytes, accdesc]
30
       else
31
           Elem[result, e, esize] = Zeros(esize);
32
       moffs = moffs + 1;
33
34
   ZAslice[t, esize, vertical, slice, VL] = result;
```

D1.1.86 LDNT1B (scalar plus immediate, consecutive registers)

Contiguous load non-temporal of bytes to multiple consecutive vectors (immediate index)

Contiguous load non-temporal of bytes to elements of two or four consecutive vector registers from the memory address generated by a 64-bit scalar base and immediate index which is multiplied by the vector's in-memory size, irrespective of predication, and 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.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: Two registers and Four registers



- <Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" times 4 plus 3.
- <Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" times 2 plus 1.

Chapter D1. SME instructions D1.1. SME and SME2 data-processing instructions

- <PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" 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 two registers variant: is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.

For the four registers variant: is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.

```
CheckStreamingSVEEnabled();
1
2 constant integer VL = CurrentVL;
3
   constant integer PL = VL DIV 8;
   constant integer elements = VL DIV esize;
4
5
   constant integer mbytes = esize DIV 8;
   bits(64) base;
6
7
   bits(PL) pred = P[g, PL];
8 bits(PL * nreg) mask = CounterToPredicate(pred<15;</pre>
                                                           PT. *
                                                                 nrea.
9
   array [0..3] of bits(VL) values;
10 boolean contiguous = TRUE;
11 boolean nontemporal = TRUE;
12
   boolean tagchecked = n != 31;
                                                              nor mporal, contiguous, tagchecked);
13
   AccessDescriptor accdesc = CreateAccDesc _ (Me. p_LOAL
14
15
   if !AnvActiveElement(mask, esize) th
                                                  16
        if n == 31 && ConstrainUnpredictab.
                                              2001
17
            CheckSPAlignment();
18
   else
19
        if n == 31 then CheckSPA' gnment
20
        base = if n == 31 then 2[] else X
                                                64];
21
22
    for r = 0 to nreg-1
23
        for e = 0 to elements-1
            if ActiveP sicateElement >sk, r * elements + e, esize) then
    bits _) addr = base + ffset * nreg * elements + r * elements + e) * mbytes;
24
25
26
                    _values', e, esize] = Mem[addr, mbytes, accdesc];
                Elr
27
            else
28
                Elen.
                         esize] = Zeros(esize);
29
30
    for r
              to
                   eg-1
31
           +r, VI
                   = value
                              1
```

D1.1.87 LDNT1B (scalar plus scalar, consecutive registers)

Contiguous load non-temporal of bytes to multiple consecutive vectors (scalar index)

Contiguous load non-temporal of bytes to elements of two or four consecutive vector registers from the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: Two registers and Four registers



- <Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" times 4 plus 3.
- <Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" times 2 plus 1.

- <PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
 - <Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

```
1
   CheckStreamingSVEEnabled();
   constant integer VL = CurrentVL;
2
   constant integer PL = VL DIV 8;
3
4
   constant integer elements = VL DIV esize;
 5
    constant integer mbytes = esize DIV 8;
 6 bits(64) offset;
   bits(64) base;
7
8
   bits(PL) pred = P[g, PL];
 9 bits(PL * nreg) mask = CounterToPredicate(pred<15:0>, PL
                                                                   nreg);
10 array [0..3] of bits(VL) values;
11
   boolean contiguous = TRUE;
12 boolean nontemporal = TRUE;
13
   boolean tagchecked = TRUE;
   AccessDescriptor accdesc = CreateAccDescSVE(Mem
                                                         LOAD,
                                                                             contiguous, tagchecked);
14
                                                                 ntempora.
15
16
   if !AnyActiveElement(mask, esize) then
17
        if n == 31 && ConstrainUnpredictableP
                                                      redic ble_C_CKSPNONEACTIVE) then
18
            CheckSPAlignment();
19
    else
20
        if n == 31 then CheckSPAlignment(),
21
        base = if n == 31 then SP[] e se X[.
                                                   11
22
        offset = X[m, 64];
23
24
    for r = 0 to nreg-1
25
        for e = 0 to elements
26
            if ActivePredic eEl nt(mask, r elements + e, esize) then
                bits(64) addr = ba. + (UInt(offset) + r * elements + e) * mbytes;
Elem[values[r], e, e. e] = Mem[addr, mbytes, accdesc];
27
28
29
            else
30
                Ele values[], e, esize] = Zeros(esize);
31
32
    for r = 0 to nr
                       :1
        Z[t+r.VL] =
33
                         ues[r];
```

D1.1.88 LDNT1B (scalar plus immediate, strided registers)

Contiguous load non-temporal of bytes to multiple strided vectors (immediate index)

Contiguous load non-temporal of bytes to elements of two or four strided vector registers from the memory address generated by a 64-bit scalar base and immediate index which is multiplied by the vector's in-memory size, irrespective of predication, and 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.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: Two registers and Four registers



Assembler Symbols

<Zt1> For the two registers variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as "T:'0':Zt".

For the four registers variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as "T:'00':Zt".

<Zt2> For the two registers variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as "T:'1':Zt".

For the four registers variant: is the name of the second scalable vector register Z4-Z7 or Z20-Z23 to be transferred, encoded as "T:'01':Zt".

- <Zt3> Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as "T:'10':Zt".
- <Zt4> Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as "T:'11':Zt".
- <PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" 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 two registers variant: is the optional signed immediate vector from a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" fier.

For the four registers variant: is the optional signed immediate octor offset, a ultiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "im o4" fields.



D1.1.89 LDNT1B (scalar plus scalar, strided registers)

Contiguous load non-temporal of bytes to multiple strided vectors (scalar index)

Contiguous load non-temporal of bytes to elements of two or four strided vector registers from the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: Two registers and Four registers



Assembler Symbols

<Zt1> For the two registers variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as "T:'0':Zt".

For the four registers variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as "T:'00':Zt".

<Zt2> For the two registers variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as "T:'1':Zt".

For the four registers variant: is the name of the second scalable vector register Z4-Z7 or Z20-Z23 to be transferred, encoded as "T:'01':Zt".

- <Zt3> Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as "T:'10':Zt".
- <Zt4> Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as "T:'11':Zt".
- <PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
 - <Xm> Is the 64-bit name of the general-purpose offset register, encoded ip """ m" field.

```
CheckStreamingSVEEnabled();
1
2
   constant integer VL = CurrentVL;
3
   constant integer PL = VL DIV 8;
4
   constant integer elements = VL DIV esize;
   constant integer mbytes = esize DIV 8;
5
6
   bits(64) offset;
7
   bits(64) base;
   bits(PL) pred = P[g, PL];
8
0
   bits(PL * nreg) mask = CounterToPredic
                                                      5:0>,
                                             e(pred
                                                            Ρ.
                                                                  nreq);
10
   array [0..3] of bits(VL) values;
11
   boolean contiguous = TRUE;
   boolean nontemporal = TRUE;
12
   boolean tagchecked = TRUE;
13
14
   AccessDescriptor accdesc = C
                                  uteAcc.
                                            cSVE(. "Op_LOAD, nontemporal, contiguous, tagchecked);
15
   if !AnyActiveElement(mas'
                                 ize) then
16
17
        if n == 31 && Const _inUn_ >dictableBo_1(Unpredictable_CHECKSPNONEACTIVE) then
18
            CheckSPAlignment();
19
   else
20
        if n == 31 t' . CheckSPAlignmen );
21
        base = if 
                      = 31 t' . SP[] else X[n, 64];
22
        offset = \lambda
                      641
23
24
   for r =
            nreg
            s = to ele n
if Ac .vePredi
25
        fo' = =
                           nts
26
                              eElement(mask, r * elements + e, esize) then
27
                  .+ < (64) adur = base + (UInt(offset) + r * elements + e) * mbytes;
28
                         es[r], e, esize] = Mem[addr, mbytes, accdesc];
29
              .se
30
                Elem[values[r], e, esize] = Zeros(esize);
31
   for r = 0 to ..reg-1
32
33
        Z[t, VL] = values[r];
34
        t = t + tstride;
```

D1.1.90 LDNT1D (scalar plus immediate, consecutive registers)

Contiguous load non-temporal of doublewords to multiple consecutive vectors (immediate index)

Contiguous load non-temporal of doublewords to elements of two or four consecutive vector registers from the memory address generated by a 64-bit scalar base and immediate index which is multiplied by the vector's in-memory size, irrespective of predication, and 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.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: Two registers and Four registers



- <Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" times 4 plus 3.
- <Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" times 2 plus 1.

Chapter D1. SME instructions D1.1. SME and SME2 data-processing instructions

- <PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" 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 two registers variant: is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.

For the four registers variant: is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.

```
CheckStreamingSVEEnabled();
1
2 constant integer VL = CurrentVL;
3
   constant integer PL = VL DIV 8;
   constant integer elements = VL DIV esize;
4
5
   constant integer mbytes = esize DIV 8;
   bits(64) base;
6
7
   bits(PL) pred = P[g, PL];
8 bits(PL * nreg) mask = CounterToPredicate(pred<15;</pre>
                                                           PT. *
                                                                 nrea.
9
   array [0..3] of bits(VL) values;
10 boolean contiguous = TRUE;
11 boolean nontemporal = TRUE;
12
   boolean tagchecked = n != 31;
                                                              nor mporal, contiguous, tagchecked);
13
   AccessDescriptor accdesc = CreateAccDesc _ (Me. p_LOAL
14
15
   if !AnyActiveElement(mask, esize) th
                                                  16
        if n == 31 && ConstrainUnpredictab.
                                              2001
17
            CheckSPAlignment();
18
   else
19
        if n == 31 then CheckSPA' gnment
20
        base = if n == 31 then 2[] else X
                                                64];
21
22
    for r = 0 to nreg-1
23
        for e = 0 to elements-1
            if ActiveP sicateElement >sk, r * elements + e, esize) then
    bits _) addr = base + ffset * nreg * elements + r * elements + e) * mbytes;
24
25
26
                    _values', e, esize] = Mem[addr, mbytes, accdesc];
                Elr
27
            else
28
                Elen.
                         esize] = Zeros(esize);
29
30
    for r
              to
                   eg-1
31
           +r, VI
                   = value
                              1
```

D1.1.91 LDNT1D (scalar plus scalar, consecutive registers)

Contiguous load non-temporal of doublewords to multiple consecutive vectors (scalar index)

Contiguous load non-temporal of doublewords to elements of two or four consecutive vector registers from the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: Two registers and Four registers



<Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" times 2 plus 1.

- <PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
 - <Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

```
1
   CheckStreamingSVEEnabled();
   constant integer VL = CurrentVL;
2
   constant integer PL = VL DIV 8;
3
4
   constant integer elements = VL DIV esize;
 5
    constant integer mbytes = esize DIV 8;
 6 bits(64) offset;
   bits(64) base;
7
8
   bits(PL) pred = P[g, PL];
 9 bits(PL * nreg) mask = CounterToPredicate(pred<15:0>, PL
                                                                   nreg);
10 array [0..3] of bits(VL) values;
11
   boolean contiguous = TRUE;
12 boolean nontemporal = TRUE;
13
   boolean tagchecked = TRUE;
   AccessDescriptor accdesc = CreateAccDescSVE(Mem
                                                         LOAD,
                                                                             contiguous, tagchecked);
14
                                                                 ntempora.
15
16
   if !AnyActiveElement(mask, esize) then
17
        if n == 31 && ConstrainUnpredictableP
                                                       redic ble_C_CKSPNONEACTIVE) then
18
            CheckSPAlignment();
19
    else
20
        if n == 31 then CheckSPAlignment(),
21
        base = if n == 31 then SP[] e se X[.
                                                   11
22
        offset = X[m, 64];
23
24
    for r = 0 to nreg-1
25
        for e = 0 to elements
26
            if ActivePredic eEl nt(mask, r elements + e, esize) then
                bits(64) addr = ba. + (UInt(offset) + r * elements + e) * mbytes;
Elem[values[r], e, e. e] = Mem[addr, mbytes, accdesc];
27
28
29
            else
30
                Ele values[], e, esize] = Zeros(esize);
31
32
    for r = 0 to nr
                       :1
        Z[t+r.VL] =
33
                         ues[r];
```

D1.1.92 LDNT1D (scalar plus immediate, strided registers)

Contiguous load non-temporal of doublewords to multiple strided vectors (immediate index)

Contiguous load non-temporal of doublewords to elements of two or four strided vector registers from the memory address generated by a 64-bit scalar base and immediate index which is multiplied by the vector's in-memory size, irrespective of predication, and 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.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: Two registers and Four registers



Assembler Symbols

<Zt1> For the two registers variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as "T:'0':Zt".

For the four registers variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as "T:'00':Zt".

<Zt2> For the two registers variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as "T:'1':Zt".

For the four registers variant: is the name of the second scalable vector register Z4-Z7 or Z20-Z23 to be transferred, encoded as "T:'01':Zt".

- <Zt3> Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as "T:'10':Zt".
- <Zt4> Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as "T:'11':Zt".
- <PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" 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 two registers variant: is the optional signed immediate vector from a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" fier.

For the four registers variant: is the optional signed immediate octor offset, a ultiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "im o4" fields.



D1.1.93 LDNT1D (scalar plus scalar, strided registers)

Contiguous load non-temporal of doublewords to multiple strided vectors (scalar index)

Contiguous load non-temporal of doublewords to elements of two or four strided vector registers from the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: Two registers and Four registers



Assembler Symbols

<Zt1> For the two registers variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as "T:'0':Zt".

For the four registers variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as "T:'00':Zt".

<Zt2> For the two registers variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as "T:'1':Zt".

For the four registers variant: is the name of the second scalable vector register Z4-Z7 or Z20-Z23 to be transferred, encoded as "T:'01':Zt".

- <Zt3> Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as "T:'10':Zt".
- <Zt4> Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as "T:'11':Zt".
- <PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
 - <Xm> Is the 64-bit name of the general-purpose offset register, encoded ip """ m" field.

```
CheckStreamingSVEEnabled();
1
2
   constant integer VL = CurrentVL;
3
   constant integer PL = VL DIV 8;
4
   constant integer elements = VL DIV esize;
   constant integer mbytes = esize DIV 8;
5
6
   bits(64) offset;
7
   bits(64) base;
   bits(PL) pred = P[g, PL];
8
0
   bits(PL * nreg) mask = CounterToPredic
                                                       5:0>,
                                             e(pred
                                                            Ρ.
                                                                  nreq);
10
   array [0..3] of bits(VL) values;
11
   boolean contiguous = TRUE;
   boolean nontemporal = TRUE;
12
   boolean tagchecked = TRUE;
13
14
   AccessDescriptor accdesc = C
                                  uteAcc.
                                            cSVE(. nOp_LOAD, nontemporal, contiguous, tagchecked);
15
   if !AnyActiveElement(mas'
16
                                 'ze) then
17
        if n == 31 && Const _inUn_ >dictableBo_1(Unpredictable_CHECKSPNONEACTIVE) then
18
            CheckSPAlignment();
19
   else
20
        if n == 31 t' . CheckSPAlignmen );
21
        base = if 
                      = 31 t' . SP[] else X[n, 64];
22
        offset = \lambda
                      641
23
24
   for r =
            nreg
            s = to ele n
if Ac .vePredi
25
        fo' =
                           nts
26
                              eElement(mask, r * elements + e, esize) then
27
                  .+ < (64) adur = base + (UInt(offset) + r * elements + e) * mbytes;
28
                         es[r], e, esize] = Mem[addr, mbytes, accdesc];
29
              .se
30
                Elem[values[r], e, esize] = Zeros(esize);
31
   for r = 0 to ..reg-1
32
33
        Z[t, VL] = values[r];
34
        t = t + tstride;
```

D1.1.94 LDNT1H (scalar plus immediate, consecutive registers)

Contiguous load non-temporal of halfwords to multiple consecutive vectors (immediate index)

Contiguous load non-temporal of halfwords to elements of two or four consecutive vector registers from the memory address generated by a 64-bit scalar base and immediate index which is multiplied by the vector's in-memory size, irrespective of predication, and 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.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: Two registers and Four registers



- <Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" times 4 plus 3.
- <Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" times 2 plus 1.

Chapter D1. SME instructions D1.1. SME and SME2 data-processing instructions

- <PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" 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 two registers variant: is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.

For the four registers variant: is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.

```
CheckStreamingSVEEnabled();
1
2 constant integer VL = CurrentVL;
3
   constant integer PL = VL DIV 8;
   constant integer elements = VL DIV esize;
4
5
   constant integer mbytes = esize DIV 8;
   bits(64) base;
6
7
   bits(PL) pred = P[g, PL];
8 bits(PL * nreg) mask = CounterToPredicate(pred<15;</pre>
                                                           PT. *
                                                                 nrea.
9
   array [0..3] of bits(VL) values;
10 boolean contiguous = TRUE;
11 boolean nontemporal = TRUE;
12
   boolean tagchecked = n != 31;
                                                              nor mporal, contiguous, tagchecked);
13
   AccessDescriptor accdesc = CreateAccDesc _ (Me. p_LOAL
14
15
   if !AnvActiveElement(mask, esize) th
                                                  16
        if n == 31 && ConstrainUnpredictab.
                                              2001
17
            CheckSPAlignment();
18
   else
19
        if n == 31 then CheckSPA' gnment
20
        base = if n == 31 then 2[] else X
                                                64];
21
22
    for r = 0 to nreg-1
23
        for e = 0 to elements-1
            if ActiveP sicateElement >sk, r * elements + e, esize) then
    bits _) addr = base + ffset * nreg * elements + r * elements + e) * mbytes;
24
25
26
                    _values', e, esize] = Mem[addr, mbytes, accdesc];
                Elr
27
            else
28
                Elen.
                         esize] = Zeros(esize);
29
30
    for r
              to
                   eg-1
31
           +r, VI
                   = value
                              1
```

D1.1.95 LDNT1H (scalar plus scalar, consecutive registers)

Contiguous load non-temporal of halfwords to multiple consecutive vectors (scalar index)

Contiguous load non-temporal of halfwords to elements of two or four consecutive vector registers from the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: Two registers and Four registers



<Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" times 2 plus 1.

- <PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
 - <Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

```
1
   CheckStreamingSVEEnabled();
2
   constant integer VL = CurrentVL;
   constant integer PL = VL DIV 8;
3
4
   constant integer elements = VL DIV esize;
 5
    constant integer mbytes = esize DIV 8;
 6 bits(64) offset;
   bits(64) base;
7
8
   bits(PL) pred = P[g, PL];
  bits(PL * nreg) mask = CounterToPredicate(pred<15:0>, PL
 9
                                                                   nreg);
10 array [0..3] of bits(VL) values;
11
   boolean contiguous = TRUE;
12 boolean nontemporal = TRUE;
13
   boolean tagchecked = TRUE;
   AccessDescriptor accdesc = CreateAccDescSVE(Mem
                                                         LOAD,
                                                                             contiguous, tagchecked);
14
                                                                 ntempora.
15
16
   if !AnyActiveElement(mask, esize) then
17
        if n == 31 && ConstrainUnpredictableP
                                                      redic ble_C_CKSPNONEACTIVE) then
18
            CheckSPAlignment();
19
    else
20
        if n == 31 then CheckSPAlignment(),
21
        base = if n == 31 then SP[] e se X[.
                                                   11
22
        offset = X[m, 64];
23
24
    for r = 0 to nreg-1
25
        for e = 0 to elements
26
            if ActivePredic eEl nt(mask, r elements + e, esize) then
                bits(64) addr = ba. + (UInt(offset) + r * elements + e) * mbytes;
Elem[va' es[r], e, e. e] = Mem[addr, mbytes, accdesc];
27
28
29
            else
30
                Ele values[], e, esize] = Zeros(esize);
31
32
    for r = 0 to nr
                       :1
        Z[t+r.VL] =
33
                         ues[r];
```

D1.1.96 LDNT1H (scalar plus immediate, strided registers)

Contiguous load non-temporal of halfwords to multiple strided vectors (immediate index)

Contiguous load non-temporal of halfwords to elements of two or four strided vector registers from the memory address generated by a 64-bit scalar base and immediate index which is multiplied by the vector's in-memory size, irrespective of predication, and 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.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: Two registers and Four registers



Assembler Symbols

<Zt1> For the two registers variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as "T:'0':Zt".

For the four registers variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as "T:'00':Zt".

<Zt2> For the two registers variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as "T:'1':Zt".

For the four registers variant: is the name of the second scalable vector register Z4-Z7 or Z20-Z23 to be transferred, encoded as "T:'01':Zt".

- <Zt3> Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as "T:'10':Zt".
- <Zt4> Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as "T:'11':Zt".
- <PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" 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 two registers variant: is the optional signed immediate vector from a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" fier.

For the four registers variant: is the optional signed immediate octor offset, a ultiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "im o4" fields.


D1.1.97 LDNT1H (scalar plus scalar, strided registers)

Contiguous load non-temporal of halfwords to multiple strided vectors (scalar index)

Contiguous load non-temporal of halfwords to elements of two or four strided vector registers from the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: Two registers and Four registers



Assembler Symbols

<Zt1> For the two registers variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as "T:'0':Zt".

For the four registers variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as "T:'00':Zt".

<Zt2> For the two registers variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as "T:'1':Zt".

For the four registers variant: is the name of the second scalable vector register Z4-Z7 or Z20-Z23 to be transferred, encoded as "T:'01':Zt".

- <Zt3> Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as "T:'10':Zt".
- <Zt4> Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as "T:'11':Zt".
- <PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
 - <Xm> Is the 64-bit name of the general-purpose offset register, encoded ip """ m" field.

```
CheckStreamingSVEEnabled();
1
2
   constant integer VL = CurrentVL;
3
   constant integer PL = VL DIV 8;
4
   constant integer elements = VL DIV esize;
   constant integer mbytes = esize DIV 8;
5
6
   bits(64) offset;
7
   bits(64) base;
   bits(PL) pred = P[g, PL];
8
0
   bits(PL * nreg) mask = CounterToPredic
                                                      5:0>,
                                             e(pred
                                                             Ρ.
                                                                  nreq);
10
   array [0..3] of bits(VL) values;
11
   boolean contiguous = TRUE;
   boolean nontemporal = TRUE;
12
   boolean tagchecked = TRUE;
13
14
   AccessDescriptor accdesc = C
                                  uteAcc.
                                            cSVE(. "Op_LOAD, nontemporal, contiguous, tagchecked);
15
   if !AnyActiveElement(mas'
16
                                 'ze) then
17
        if n == 31 && Const _inUn_ >dictableBo_1(Unpredictable_CHECKSPNONEACTIVE) then
18
            CheckSPAlignment();
19
   else
20
        if n == 31 t' . CheckSPAlignmen );
21
        base = if 
                      = 31 t' . SP[] else X[n, 64];
22
        offset = \lambda
                      641
23
24
   for r =
            nreg
            s = to ele n
if Ac .vePredi
25
        fo' = =
                           nts
26
                              eElement(mask, r * elements + e, esize) then
27
                  .+ < (64) adur = base + (UInt(offset) + r * elements + e) * mbytes;
28
                         es[r], e, esize] = Mem[addr, mbytes, accdesc];
29
              .se
30
                Elem[values[r], e, esize] = Zeros(esize);
31
   for r = 0 to ..reg-1
32
33
        Z[t, VL] = values[r];
34
        t = t + tstride;
```

D1.1.98 LDNT1W (scalar plus immediate, consecutive registers)

Contiguous load non-temporal of words to multiple consecutive vectors (immediate index)

Contiguous load non-temporal of words to elements of two or four consecutive vector registers from the memory address generated by a 64-bit scalar base and immediate index which is multiplied by the vector's in-memory size, irrespective of predication, and 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.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: Two registers and Four registers



- <Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" times 4 plus 3.
- <Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" times 2 plus 1.

Chapter D1. SME instructions D1.1. SME and SME2 data-processing instructions

- <PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" 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 two registers variant: is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.

For the four registers variant: is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.

```
CheckStreamingSVEEnabled();
1
2 constant integer VL = CurrentVL;
3
   constant integer PL = VL DIV 8;
   constant integer elements = VL DIV esize;
4
5
   constant integer mbytes = esize DIV 8;
   bits(64) base;
6
7
   bits(PL) pred = P[g, PL];
8 bits(PL * nreg) mask = CounterToPredicate(pred<15;</pre>
                                                           PT. *
                                                                 nrea.
9
   array [0..3] of bits(VL) values;
10 boolean contiguous = TRUE;
11 boolean nontemporal = TRUE;
12
   boolean tagchecked = n != 31;
                                                              nor mporal, contiguous, tagchecked);
13
   AccessDescriptor accdesc = CreateAccDesc _ (Me. p_LOAL
14
15
   if !AnvActiveElement(mask, esize) th
                                                  16
        if n == 31 && ConstrainUnpredictab.
                                              2001
17
            CheckSPAlignment();
18
   else
19
        if n == 31 then CheckSPA' gnment
20
        base = if n == 31 then 2[] else X
                                                64];
21
22
    for r = 0 to nreg-1
23
        for e = 0 to elements-1
            if ActiveP sicateElement >sk, r * elements + e, esize) then
    bits _) addr = base + ffset * nreg * elements + r * elements + e) * mbytes;
24
25
26
                    _values', e, esize] = Mem[addr, mbytes, accdesc];
                Elr
27
            else
28
                Elen.
                         esize] = Zeros(esize);
29
30
    for r
              to
                   eg-1
31
           +r, VI
                   = value
                              1
```

D1.1.99 LDNT1W (scalar plus scalar, consecutive registers)

Contiguous load non-temporal of words to multiple consecutive vectors (scalar index)

Contiguous load non-temporal of words to elements of two or four consecutive vector registers from the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: Two registers and Four registers



<Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" times 2 plus 1.

- <PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
 - <Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

```
1
   CheckStreamingSVEEnabled();
2
   constant integer VL = CurrentVL;
   constant integer PL = VL DIV 8;
3
4
   constant integer elements = VL DIV esize;
 5
    constant integer mbytes = esize DIV 8;
 6 bits(64) offset;
   bits(64) base;
7
8
   bits(PL) pred = P[g, PL];
 9 bits(PL * nreg) mask = CounterToPredicate(pred<15:0>, PL
                                                                   nreg);
10 array [0..3] of bits(VL) values;
11
   boolean contiguous = TRUE;
12 boolean nontemporal = TRUE;
13
   boolean tagchecked = TRUE;
   AccessDescriptor accdesc = CreateAccDescSVE(Mem
                                                         LOAD,
                                                                             contiguous, tagchecked);
14
                                                                 ntempora.
15
16
   if !AnyActiveElement(mask, esize) then
17
        if n == 31 && ConstrainUnpredictableP
                                                       redic ble_C_CKSPNONEACTIVE) then
18
            CheckSPAlignment();
19
    else
20
        if n == 31 then CheckSPAlignment(),
21
        base = if n == 31 then SP[] e se X[.
                                                   11
22
        offset = X[m, 64];
23
24
    for r = 0 to nreg-1
25
        for e = 0 to elements
26
            if ActivePredic eEl nt(mask, r elements + e, esize) then
                bits(64) addr = ba. + (UInt(offset) + r * elements + e) * mbytes;
Elem[values[r], e, e. e] = Mem[addr, mbytes, accdesc];
27
28
29
            else
30
                Ele values[], e, esize] = Zeros(esize);
31
32
    for r = 0 to nr
                       :1
        Z[t+r.VL] =
33
                         ues[r];
```

D1.1.100 LDNT1W (scalar plus immediate, strided registers)

Contiguous load non-temporal of words to multiple strided vectors (immediate index)

Contiguous load non-temporal of words to elements of two or four strided vector registers from the memory address generated by a 64-bit scalar base and immediate index which is multiplied by the vector's in-memory size, irrespective of predication, and 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.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: Two registers and Four registers



Assembler Symbols

<Zt1> For the two registers variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as "T:'0':Zt".

For the four registers variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as "T:'00':Zt".

<Zt2> For the two registers variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as "T:'1':Zt".

For the four registers variant: is the name of the second scalable vector register Z4-Z7 or Z20-Z23 to be transferred, encoded as "T:'01':Zt".

- <Zt3> Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as "T:'10':Zt".
- <Zt4> Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as "T:'11':Zt".
- <PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" 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 two registers variant: is the optional signed immediate vector from a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" fier.

For the four registers variant: is the optional signed immediate octor offset, a ultiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "im o4" fields.



D1.1.101 LDNT1W (scalar plus scalar, strided registers)

Contiguous load non-temporal of words to multiple strided vectors (scalar index)

Contiguous load non-temporal of words to elements of two or four strided vector registers from the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: Two registers and Four registers



Assembler Symbols

<Zt1> For the two registers variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as "T:'0':Zt".

For the four registers variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as "T:'00':Zt".

<Zt2> For the two registers variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as "T:'1':Zt".

For the four registers variant: is the name of the second scalable vector register Z4-Z7 or Z20-Z23 to be transferred, encoded as "T:'01':Zt".

- <Zt3> Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as "T:'10':Zt".
- <Zt4> Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as "T:'11':Zt".
- <PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
 - <Xm> Is the 64-bit name of the general-purpose offset register, encoded ip """ m" field.

```
CheckStreamingSVEEnabled();
1
2
   constant integer VL = CurrentVL;
3
   constant integer PL = VL DIV 8;
4
   constant integer elements = VL DIV esize;
   constant integer mbytes = esize DIV 8;
5
6
   bits(64) offset;
7
   bits(64) base;
   bits(PL) pred = P[g, PL];
8
0
   bits(PL * nreg) mask = CounterToPredic
                                                      5:0>,
                                             e(pred
                                                            Ρ.
                                                                  nreq);
10
   array [0..3] of bits(VL) values;
11
   boolean contiguous = TRUE;
   boolean nontemporal = TRUE;
12
   boolean tagchecked = TRUE;
13
14
   AccessDescriptor accdesc = C
                                  uteAcc.
                                            cSVE(. "Op_LOAD, nontemporal, contiguous, tagchecked);
15
   if !AnyActiveElement(mas'
16
                                 'ze) then
17
        if n == 31 && Const _inUn_ >dictableBo_1(Unpredictable_CHECKSPNONEACTIVE) then
18
            CheckSPAlignment();
19
   else
20
        if n == 31 t' . CheckSPAlignmen );
21
        base = if 
                      = 31 t' . SP[] else X[n, 64];
22
        offset = \lambda
                      641
23
24
   for r =
            nreg
            s = to ele n
if Ac .vePredi
25
        fo' =
                           nts
26
                              eElement(mask, r * elements + e, esize) then
27
                  .+ < (64) adur = base + (UInt(offset) + r * elements + e) * mbytes;
28
                         es[r], e, esize] = Mem[addr, mbytes, accdesc];
29
              .se
30
                Elem[values[r], e, esize] = Zeros(esize);
31
   for r = 0 to ..reg-1
32
33
        Z[t, VL] = values[r];
34
        t = t + tstride;
```

D1.1.102 LDR (vector)

Load ZA array vector

The ZA array vector is selected by the sum of the vector select register and immediate offset, modulo the number of bytes in a Streaming SVE vector. The immediate offset is in the range 0 to 15. The memory address is generated by a 64-bit scalar base, plus the same optional immediate offset multiplied by the current vector length in bytes. This instruction is unpredicated.

The load is performed as contiguous byte accesses, with no endian conversion and no guarantee of single-copy atomicity larger than a byte. However, if alignment is checked, then the base register must be aligned to 16 bytes.

This instruction does not require the PE to be in Streaming SVE mode, and it is expected that this instruction will not experience a significant slowdown due to contention with other PEs that are executing in Streaming SVE mode.

SME (FEAT_SME)



Mu

11

```
LDR ZA[<Wv>, <offs>], [<Xn|SP>{, #<offs>,
```

```
1 if !HaveSME() then UNDEFINED;
```

```
2 integer n = UInt(Rn);
```

```
3 integer v = UInt('011':Rv);
4 integer offset = UInt(off4);
```

Assembler Symbols

- $\langle Wv \rangle$ Is the 32-bit name of t' or select regime to respect to the W12-W15, encoded in the "Rv" field.
- <offs> Is the vector select offset and ptional memory offset, in the range 0 to 15, defaulting to 0, encoded in the off4" field.
- <Xn|SP> Is the 64-b name of e general-purpose base register or stack pointer, encoded in the "Rn" field.

Operati

```
Chec MEAndZA habled(),
1
2
   con. nt i
                      CVL =
                            CurrentSVL;
               iteger and = SVL DIV 8;
3
   const.
4
   bits(64
              base;
5
               offs = offset * dim;
   integer
6
   bits(SVL) r vlt;
7
   bits(32)
              vbase = X[v, 32];
              vec = (UInt(vbase) + offset) MOD dim;
8
   integer
9
   boolean
              contiguous = TRUE;
10
   boolean
              nontemporal = FALSE;
              tagchecked = n != 31;
11
   boolean
   AccessDescriptor accdesc = CreateAccDescSME (MemOp_LOAD, nontemporal, contiguous, tagchecked);
12
13
14
   if HaveTME() && TSTATE.depth > 0 then
15
        FailTransaction(TMFailure_ERR, FALSE);
16
   if n == 31 then
17
18
        CheckSPAlignment();
19
        base = SP[];
20
   else
21
        base = X[n, 64];
22
23
   boolean aligned = IsAligned(base + offset, 16);
24
```

Chapter D1. SME instructions D1.1. SME and SME2 data-processing instructions

D1.1.103 LDR (ZT0)

Load ZT0 register

Load the 64-byte ZT0 register from the memory address provided in the 64-bit scalar base register. This instruction is unpredicated.

The load is performed as contiguous byte accesses, with no endian conversion and no guarantee of single-copy atomicity larger than a byte. However, if alignment is checked, then the base register must be aligned to 16 bytes.

This instruction does not require the PE to be in Streaming SVE mode, and it is expected that this instruction will not experience a significant slowdown due to contention with other PEs that are executing in Streaming SVE mode.

SME2 (FEAT_SME2)



D1.1.104 LUTI2 (two registers)

Lookup table read with 2-bit indexes

Copy 8-bit, 16-bit or 32-bit elements from ZT0 to two destination vectors using packed 2-bit indices from a segment of the source vector register. A segment corresponds to a portion of the source vector that is consumed in order to fill the destination vector. The segment is selected by the vector segment index modulo the total number of segments.

This instruction is unpredicated.

SME2 (FEAT_SME2)



Assembler Symbols

- <Zd1> Is the name of the first d nation scalab. "ector register of a multi-vector sequence, encoded as "Zd" times 2.
 - <T> Is the size specific, encoded in ize":

 size
 I>

 00
 B

 01
 H

 10
 I

 11
 ESERV

- <Zd2> t¹ name or the second destination scalable vector register of a multi-vector sequence, e., ded as "Zd" times 2 plus 1.
- <Zn> Is the . me of the source scalable vector register, encoded in the "Zn" field.

<index> Is the vector segment index, in the range 0 to 7, encoded in the "i3" field.

```
1
   CheckStreamingSVEEnabled();
   CheckSMEZT0Enabled();
2
   constant integer VL = CurrentVL;
3
   constant integer elements = VL DIV esize;
4
5
   integer segments = esize DIV (isize * nreg);
6
   integer segment = imm MOD segments;
   bits(VL) indexes = Z[n, VL];
7
8
   bits(512) table = ZT0[512];
9
   for r = 0 to nreg-1
10
11
       integer base = (segment * nreg + r) * elements;
       bits(VL) result = Z[d+r, VL];
12
13
       for e = 0 to elements-1
```

Chapter D1. SME instructions D1.1. SME and SME2 data-processing instructions

14	<pre>integer index = UInt(Elem[indexes, base+e, isize]);</pre>
15	<pre>Elem[result, e, esize] = Elem[table, index, 32]<esize-1:0>;</esize-1:0></pre>
16	<pre>Z[d+r, VL] = result;</pre>

D1.1.105 LUTI2 (four registers)

Lookup table read with 2-bit indexes

Copy 8-bit, 16-bit or 32-bit elements from ZT0 to four destination vectors using packed 2-bit indices from a segment of the source vector register. A segment corresponds to a portion of the source vector that is consumed in order to fill the destination vector. The segment is selected by the vector segment index modulo the total number of segments.

This instruction is unpredicated.

SME2 (FEAT_SME2)



Assembler Symbols

- <Zd1> Is the name of the first d nation scalab. "ector register of a multi-vector sequence, encoded as "Zd" times 4.
 - <T> Is the size specific, encoded in ize":

 size
 ...>

 00
 B

 01
 H

 1^
 ESERV

- <Zd4> t¹ name or the fourth destination scalable vector register of a multi-vector sequence, e., ded as "Zd" times 4 plus 3.
- <Zn> Is the . me of the source scalable vector register, encoded in the "Zn" field.

<index> Is the vector segment index, in the range 0 to 3, encoded in the "i2" field.

```
1
   CheckStreamingSVEEnabled();
   CheckSMEZT0Enabled();
2
   constant integer VL = CurrentVL;
3
   constant integer elements = VL DIV esize;
4
5
   integer segments = esize DIV (isize * nreg);
6
   integer segment = imm MOD segments;
   bits(VL) indexes = Z[n, VL];
7
8
   bits(512) table = ZT0[512];
9
   for r = 0 to nreg-1
10
11
       integer base = (segment * nreg + r) * elements;
       bits(VL) result = Z[d+r, VL];
12
13
       for e = 0 to elements-1
```

Chapter D1. SME instructions D1.1. SME and SME2 data-processing instructions

14	<pre>integer index = UInt(Elem[indexes, base+e, isize]);</pre>
15	<pre>Elem[result, e, esize] = Elem[table, index, 32]<esize-1:0>;</esize-1:0></pre>
16	<pre>Z[d+r, VL] = result;</pre>

D1.1.106 LUTI2 (single)

Lookup table read with 2-bit indexes

Copy 8-bit, 16-bit or 32-bit elements from ZT0 to one destination vector using packed 2-bit indices from a segment of the source vector register. A segment corresponds to a portion of the source vector that is consumed in order to fill the destination vector. The segment is selected by the vector segment index modulo the total number of segments.

This instruction is unpredicated.

SME2 (FEAT_SME2)



Assembler Symbols



<T> Is the size specifier, _ncodea "size":

 size
 <T'</th>

 00
 5

 01
 H

 10
 S

 1.1
 RESL (ED)

<Zn. Is the *r* model the source scalable vector register, encoded in the "Zn" field.

e vector segment index, in the range 0 to 15, encoded in the "i4" field.

Operation

<index>

```
CheckStreamingSVEEnabled();
1
2
   CheckSMEZTOEnabled();
   constant integer VL = CurrentVL;
3
4
   constant integer elements = VL DIV esize;
   integer segments = esize DIV (isize * nreg);
5
   integer segment = imm MOD segments;
6
7
   bits(VL) indexes = Z[n, VL];
8
   bits(512) table = ZT0[512];
9
10
   for r = 0 to nreg-1
11
       integer base = (segment * nreg + r) * elements;
12
       bits(VL) result = Z[d+r, VL];
13
       for e = 0 to elements-1
            integer index = UInt(Elem[indexes, base+e, isize]);
14
15
            Elem[result, e, esize] = Elem[table, index, 32]<esize-1:0>;
16
       Z[d+r, VL] = result;
```

D1.1.107 LUTI4 (two registers)

Lookup table read with 4-bit indexes

Copy 8-bit, 16-bit or 32-bit elements from ZT0 to two destination vectors using packed 4-bit indices from a segment of the source vector register. A segment corresponds to a portion of the source vector that is consumed in order to fill the destination vector. The segment is selected by the vector segment index modulo the total number of segments.

This instruction is unpredicated.

SME2 (FEAT_SME2)



Assembler Symbols

- <Zd1> Is the name of the first d nation scalab. "ector register of a multi-vector sequence, encoded as "Zd" times 2.
 - <T> Is the size specific, encoded in ize":

 size
 I>

 00
 B

 01
 H

 10
 I

 11
 ESERV

- <Zd2> t¹ name or the second destination scalable vector register of a multi-vector sequence, e., ded as "Zd" times 2 plus 1.
- <Zn> Is the . me of the source scalable vector register, encoded in the "Zn" field.

<index> Is the vector segment index, in the range 0 to 3, encoded in the "i2" field.

```
1
   CheckStreamingSVEEnabled();
   CheckSMEZT0Enabled();
2
   constant integer VL = CurrentVL;
3
   constant integer elements = VL DIV esize;
4
5
   integer segments = esize DIV (isize * nreg);
6
   integer segment = imm MOD segments;
   bits(VL) indexes = Z[n, VL];
7
8
   bits(512) table = ZT0[512];
9
   for r = 0 to nreg-1
10
11
       integer base = (segment * nreg + r) * elements;
       bits(VL) result = Z[d+r, VL];
12
13
       for e = 0 to elements-1
```

Chapter D1. SME instructions D1.1. SME and SME2 data-processing instructions

14	<pre>integer index = UInt(Elem[indexes, base+e, isize]);</pre>
15	<pre>Elem[result, e, esize] = Elem[table, index, 32]<esize-1:0>;</esize-1:0></pre>
16	<pre>Z[d+r, VL] = result;</pre>

D1.1.108 LUTI4 (four registers)

Lookup table read with 4-bit indexes

Copy 16-bit or 32-bit elements from ZT0 to four destination vectors using packed 4-bit indices from a segment of the source vector register. A segment corresponds to a portion of the source vector that is consumed in order to fill the destination vector. The segment is selected by the vector segment index modulo the total number of segments.

This instruction is unpredicated.

SME2 (FEAT_SME2)



Assembler Symbols

- <Zd1> Is the name of the first destination scall 'e vector egister of a multi-vector sequence, encoded as "Zd" times 4.
 - <T> Is the size specifier, .ncodea "size":

 size
 <T'</th>

 00
 RES RVED

 01
 H

 10
 S

 11
 RESL (ED)

- <Zd4: Is the proof the fourth destination scalable vector register of a multi-vector sequence, vector ed as "Zu- ames 4 plus 3.
- <Zn> Is i. name of the source scalable vector register, encoded in the "Zn" field.

<index> Is the vector segment index, in the range 0 to 1, encoded in the "i1" field.

```
CheckStreamingSVEEnabled();
1
   CheckSMEZT0Enabled();
2
3
   constant integer VL = CurrentVL;
4
   constant integer elements = VL DIV esize;
   integer segments = esize DIV (isize * nreg);
5
   integer segment = imm MOD segments;
6
   bits(VL) indexes = Z[n, VL];
7
8
   bits(512) table = ZTO[512];
9
10
   for r = 0 to nreg-1
11
       integer base = (segment * nreg + r) * elements;
12
       bits(VL) result = Z[d+r, VL];
13
       for e = 0 to elements-1
14
            integer index = UInt(Elem[indexes, base+e, isize]);
```

Chapter D1. SME instructions D1.1. SME and SME2 data-processing instructions

> 15 Elem[result, e, esize] = Elem[table, index, 32]<esize-1:0>; 16 Z[d+r, VL] = result;



D1.1.109 LUTI4 (single)

Lookup table read with 4-bit indexes

Copy 8-bit, 16-bit or 32-bit elements from ZT0 to one destination vector using packed 4-bit indices from a segment of the source vector register. A segment corresponds to a portion of the source vector that is consumed in order to fill the destination vector. The segment is selected by the vector segment index modulo the total number of segments.

This instruction is unpredicated.

SME2 (FEAT_SME2)



Assembler Symbols



<T> Is the size specifier, _ncodea "size":

 size
 <T'</th>

 00
 5

 01
 H

 10
 S

 1.1
 RESL (ED)

<Zn: Is the *r* model the source scalable vector register, encoded in the "Zn" field.

e vector segment index, in the range 0 to 7, encoded in the "i3" field.

Operation

<index>

```
CheckStreamingSVEEnabled();
1
2
   CheckSMEZTOEnabled();
   constant integer VL = CurrentVL;
3
4
   constant integer elements = VL DIV esize;
   integer segments = esize DIV (isize * nreg);
5
   integer segment = imm MOD segments;
6
7
   bits(VL) indexes = Z[n, VL];
8
   bits(512) table = ZT0[512];
9
10
   for r = 0 to nreg-1
11
       integer base = (segment * nreg + r) * elements;
12
       bits(VL) result = Z[d+r, VL];
13
       for e = 0 to elements-1
            integer index = UInt(Elem[indexes, base+e, isize]);
14
15
            Elem[result, e, esize] = Elem[table, index, 32]<esize-1:0>;
16
       Z[d+r, VL] = result;
```

D1.1.110 MOV (tile to vector, two registers)

Move two ZA tile slices to two vector registers

The instruction operates on two consecutive horizontal or vertical slices within a named ZA tile of the specified element size.

The consecutive slice numbers within the tile are selected starting from the sum of the slice index register and immediate offset, modulo the number of such elements in a vector. The immediate offset is a multiple of 2 in the range 0 to the number of elements in a 128-bit vector segment minus 2.

This instruction is unpredicated.

This is an alias of MOVA (tile to vector, two registers). This means:

- The encodings in this description are named to match the encodings carry (A (tile to vector, two registers).
- The description of MOVA (tile to vector, two registers) gives the contractional pse boode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for is instruction.

It has encodings from 4 classes: 8-bit, 16-bit, 32-bit and 64-b

8-bit

1 0 0 0 0 0 0 0 0 0 0 0 off3 Zd v 0 0 Θ 1 ß ß sizer1>

MOV { <Zd1>.B-<Zd2>.B }, ZA 'HV>.1 s>, <offsf>:<offsl>]

is equivalent to

```
MOVA { <Zd1>.B-<Zd2
                           ZAO<HV>.b
                                       Ws>.
                                           <offsf>:<offsl>]
```

and is always the prefer d disassembly

16-bit



MOV dl>.n_.d2>.H }, <ZAn><HV>.H[<Ws>, <offsf>:<offsl>]

is equivalen.

```
MOVA { <2d1>.H-<Zd2>.H }, <ZAn><HV>.H[<Ws>, <offsf>:<offsl>]
```

and is always the preferred disassembly.

32-bit



```
MOV { <Zdl>.S-<Zd2>.S }, <ZAn><HV>.S[<Ws>, <offsf>:<offsl>]
```

is equivalent to

```
MOVA { <Zd1>.S-<Zd2>.S }, <ZAn><HV>.S[<Ws>, <offsf>:<offsl>]
```

and is always the preferred disassembly.

64-bit



MOV { <Zd1>.D-<Zd2>.D }, <ZAn><HV>.D[<Ws>, <offsf>:<offsl>]

is equivalent to

MOVA { <Zdl>.D-<Zd2>.D }, <ZAn><HV>.D[<Ws>, <offsf>:<offsl>]

and is always the preferred disassembly.

Assembler Symbols

- $\langle Zd1 \rangle$ Is the name of the first destination scalable vector register *c* a multiple ctor sequence, encoded as "Zd" times 2.
- $\langle Zd2 \rangle$ Is the name of the second destination scalable v or regist of a multivector sequence, encoded as "Zd" times 2 plus 1.
- For the 16-bit variant: is the name of the ZAO-2 1 to b ccessed, encoded in the $\langle ZAn \rangle$ "ZAn" field.

For the 32-bit variant: is the name of \neg ZA 3 to be accessed, encoded in the "ZAn" field.

For the 64-bit variant: is the name whe ZA be ZAO-ZA7 to be accessed, encoded in the "ZAn" field.

Is the horizontal or tical 're indicator, coded in "V": $\langle HV \rangle$

V	<hv></hv>
0	Н
1	V

- -bit n. of the .ce index register W12-W15, encoded in the "Rs" field. $\langle Ws \rangle$ Is *
- <offsf> s the slice index offset, pointing to first of two consecutive slices, For the bit varia. "field times 2. encod

the 16-bit variant: is the slice index offset, pointing to first of two consecutive slices, enc. d as "off2" field times 2.

For the 32-bit variant: is the slice index offset, pointing to first of two consecutive slices, encoded as "o1" field times 2.

For the 64-bit variant: is the slice index offset, pointing to first of two consecutive slices, with implicit value 0.

<offsl> For the 8-bit variant: is the slice index offset, pointing to last of two consecutive slices, encoded as "off3" field times 2 plus 1.

> For the 16-bit variant: is the slice index offset, pointing to last of two consecutive slices, encoded as "off2" field times 2 plus 1.

> For the 32-bit variant: is the slice index offset, pointing to last of two consecutive slices, encoded as "o1" field times 2 plus 1.

> For the 64-bit variant: is the slice index offset, pointing to last of two consecutive slices, with implicit value 1.

Operation

The description of MOVA (tile to vector, two registers) gives the operational pseudocode for this instruction.

Operational information

- The execution time of this instruction is independent of:
 - The values of the data supplied in any of its registers.
 - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
 - The values of the data supplied in any of its registers.
 - The values of the NZCV flags.

D1.1.111 MOV (tile to vector, four registers)

Move four ZA tile slices to four vector registers

The instruction operates on four consecutive horizontal or vertical slices within a named ZA tile of the specified element size.

The consecutive slice numbers within the tile are selected starting from the sum of the slice index register and immediate offset, modulo the number of such elements in a vector. The immediate offset is a multiple of 4 in the range 0 to the number of elements in a 128-bit vector segment minus 4.

This instruction is unpredicated.

This is an alias of MOVA (tile to vector, four registers). This means:

- The encodings in this description are named to match the encodings c² (tile to vector, four registers).
- The description of MOVA (tile to vector, four registers) gives the *c*_rational psc 'ocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for is instruction.

It has encodings from 4 classes: 8-bit, 16-bit, 32-bit and 64-b

8-bit

1 0 0 0 0 0 0 0 0 0 0 0 v 0 0 0 Θ off2 7d 1 sizer1>

MOV { <Zd1>.B-<Zd4>.B }, ZA 'HV>.1 s>, <offsf>:<offsl>]

is equivalent to

```
MOVA { <Zd1>.B-<Zd4
                           ZAO<HV>.b
                                       Ws>.
                                           <offsf>:<offsl>]
```

and is always the prefer d disassembly

16-bit



MOV Ldl>.n_.d4>.H }, <ZAn><HV>.H[<Ws>, <offsf>:<offsl>]

is equivalen.

```
MOVA { <2d1>.H-<2d4>.H }, <2An><HV>.H[<Ws>, <offsf>:<offsl>]
```

and is always the preferred disassembly.

32-bit



```
MOV { <Zdl>.S-<Zd4>.S }, <ZAn><HV>.S[<Ws>, <offsf>:<offsl>]
```

is equivalent to

```
MOVA { <Zd1>.S-<Zd4>.S }, <ZAn><HV>.S[<Ws>, <offsf>:<offs1>]
```

and is always the preferred disassembly.

64-bit



MOV { <Zd1>.D-<Zd4>.D }, <ZAn><HV>.D[<Ws>, <offsf>:<offsl>]

is equivalent to

MOVA { <Zdl>.D-<Zd4>.D }, <ZAn><HV>.D[<Ws>, <offsf>:<offsl>]

and is always the preferred disassembly.

Assembler Symbols

- $\langle Zd1 \rangle$ Is the name of the first destination scalable vector register *c* a multiple ctor sequence, encoded as "Zd" times 4.
- Is the name of the fourth destination scalable v_{c} or regist of a multivector sequence, <Zd4>encoded as "Zd" times 4 plus 3.
- For the 16-bit variant: is the name of the ZAO-2 1 to b ccessed, encoded in the $\langle ZAn \rangle$ "ZAn" field.

For the 32-bit variant: is the name of >ZA 3 to be accessed, encoded in the "ZAn" field.

For the 64-bit variant: is the name whe ZA be ZAO-ZA7 to be accessed, encoded in the "ZAn" field.

Is the horizontal or tical 're indicator, coded in "V": $\langle HV \rangle$

V	<hv></hv>
0	Н
1	Ţ

- -bit n. of the .ce index register W12-W15, encoded in the "Rs" field. $\langle Ws \rangle$ Is *
- <offsf> s the slice index offset, pointing to first of four consecutive slices, For the bit varia. " ff?" field times 4. encod

the 16-bit variant: is the slice index offset, pointing to first of four consecutive slices, ency d as "o1" field times 4.

For the 32-bit and 64-bit variant: is the slice index offset, pointing to first of four consecutive slices, with implicit value 0.

<offsl> For the 8-bit variant: is the slice index offset, pointing to last of four consecutive slices, encoded as "off2" field times 4 plus 3.

For the 16-bit variant: is the slice index offset, pointing to last of four consecutive slices, encoded as "o1" field times 4 plus 3.

For the 32-bit and 64-bit variant: is the slice index offset, pointing to last of four consecutive slices, with implicit value 3.

Operation

The description of MOVA (tile to vector, four registers) gives the operational pseudocode for this instruction.

Operational information

- The execution time of this instruction is independent of:
 - The values of the data supplied in any of its registers.
 - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
 - The values of the data supplied in any of its registers.
 - The values of the NZCV flags.



D1.1.112 MOV (array to vector, two registers)

Move two ZA single-vector groups to two vector registers

The instruction operates on two ZA single-vector groups. The vector numbers forming the single-vector group within each half of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half the number of ZA array vectors.

The VECTOR GROUP symbol VGx2 indicates that the instruction operates on two ZA single-vector groups.

The preferred disassembly syntax uses a 64-bit element size, but an assembler should accept any element size if it is used consistently for all operands. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

This is an alias of MOVA (array to vector, two registers). This means:

- The encodings in this description are named to match the encodin. of MOVA (ar y to vector, two registers).
- The description of MOVA (array to vector, two regist a) gives the operational pseudocode, any CON-STRAINED UNPREDICTABLE behavior, and any operational information. Transmission of the second seco



MOV { <Zd1>.D-<Zd2>.D }, ZA.D[</

is equivalent to

MOVA	{ <zd1>.D-<zd2>.D</zd2></zd1>	4	ZA.D[<w< td=""><td><pre>v</pre>, V</td><td>Gx2}</td></w<>	<pre>v</pre> , V	Gx2}

and is always the preferred insassen. 'v

Assembler Symbols

- <Zd1> Is the name of the first estination scalable vector register of a multi-vector sequence, encoded as "Zd" tin. 2.
- <Zd2> Is me on a second estination scalable vector register of a multi-vector sequence, ncoded 3 "Zd" u. a plus 1.

<Wv> Is the ______ me of the vector select register W8-W11, encoded in the "Rv" field.

<offs> h e vector select offset, in the range 0 to 7, encoded in the "off3" field.

Operation

The description of MOVA (array to vector, two registers) gives the operational pseudocode for this instruction.

Operational information

- The execution time of this instruction is independent of:
 - The values of the data supplied in any of its registers.
 - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
 - The values of the data supplied in any of its registers.
 - The values of the NZCV flags.

D1.1.113 MOV (array to vector, four registers)

Move four ZA single-vector groups to four vector registers

The instruction operates on four ZA single-vector groups. The vector numbers forming the single-vector group within each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo quarter the number of ZA array vectors.

The VECTOR GROUP symbol VGx4 indicates that the instruction operates on four ZA single-vector groups.

The preferred disassembly syntax uses a 64-bit element size, but an assembler should accept any element size if it is used consistently for all operands. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

This is an alias of MOVA (array to vector, four registers). This means:

- The encodings in this description are named to match the encoding of MOVA (ar v to vector, four registers).
- The description of MOVA (array to vector, four regist 3) gives 1. operatic al pseudocode, any CON-STRAINED UNPREDICTABLE behavior, and any oper? anal information. Dr. s instruction.



MOV { <Zd1>.D-<Zd4>.D }, ZA.D[<N , <

is equivalent to

MOVA	{ <zd1>.D-<zd4>.D</zd4></zd1>	4	ZA.D[<w< td=""><td><of>{,</of></td><td>VGx4}</td></w<>	<of>{,</of>	VGx4}

and is always the preferred insassen. 'v

Assembler Symbols

- <Zd1> Is the name of the first estination scalable vector register of a multi-vector sequence, encoded as "Zd" tin. 4.
- <Zd4> Is me on e fourt' aestination scalable vector register of a multi-vector sequence, needed s "Zd" i. e plus 3.

<Wv> Is the ______ me of the vector select register W8-W11, encoded in the "Rv" field.

<offs> 1. e vector select offset, in the range 0 to 7, encoded in the "off3" field.

Operation

The description of MOVA (array to vector, four registers) gives the operational pseudocode for this instruction.

Operational information

- The execution time of this instruction is independent of:
 - The values of the data supplied in any of its registers.
 - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
 - The values of the data supplied in any of its registers.
 - The values of the NZCV flags.

D1.1.114 MOV (tile to vector, single)

Move ZA tile slice to vector register

The instruction operates on individual horizontal or vertical slices within a named ZA tile of the specified element size. The slice number within the tile is selected by the sum of the slice index register and immediate offset, modulo the number of such elements in a vector. The immediate offset is in the range 0 to the number of elements in a 128-bit vector segment minus 1.

Inactive elements in the destination vector remain unmodified.

This is an alias of MOVA (tile to vector, single). This means:

- The encodings in this description are named to match the encodings of MOVA (tile to vector, single).
- The description of MOVA (tile to vector, single) gives the operation udocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for t^{1} instruct.

It has encodings from 5 classes: 8-bit, 16-bit, 32-bit, 64-bit and 128



MOV <Zd>.S, <Pg>/M, <ZAn><HV>.S[<Ws>, <offs>]

is equivalent to

MOVA <Zd>.S, <Pg>/M, <ZAn><HV>.S[<Ws>, <offs>]

Chapter D1. SME instructions D1.1. SME and SME2 data-processing instructions

and is always the preferred disassembly.

64-bit



is equivalent to

MOVA <Zd>.Q, <Pg>/M, <ZAn><FV>.QL 's' <offs>_

and is always the preferred disasse .oly.

Assembler Symbols

- <Zd> Is the name of the destination. alable vector register, encoded in the "Zd" field.
- <Pg> Is the name of the governing scale by predicate register P0-P7, encoded in the "Pg" field.
- <ZAn> For the 1 sit variation is the name of the ZA tile ZA0-ZA1 to be accessed, encoded in the "ZAn" field
 - F and 2-bit v fant: if the name of the ZA tile ZA0-ZA3 to be accessed, encoded in the ZAn" f d.
 - For t¹ ant: is the name of the ZA tile ZA0-ZA7 to be accessed, encoded in the a" field.

For L 128-bit variant: is the name of the ZA tile ZA0-ZA15 to be accessed, encoded in the "ZAn" reld.

<HV> Is the horizontal or vertical slice indicator, encoded in "V":

V	<hv></hv>
0	Н
1	V

- <Ws> Is the 32-bit name of the slice index register W12-W15, encoded in the "Rs" field.
- <offs> For the 8-bit variant: is the slice index offset, in the range 0 to 15, encoded in the "off4" field. For the 16-bit variant: is the slice index offset, in the range 0 to 7, encoded in the "off3" field. For the 32-bit variant: is the slice index offset, in the range 0 to 3, encoded in the "off2" field. For the 64-bit variant: is the slice index offset, in the range 0 to 1, encoded in the "o1" field.

For the 128-bit variant: is the slice index offset 0.

Operation

The description of MOVA (tile to vector, single) gives the operational pseudocode for this instruction.

Operational information

- The execution time of this instruction is independent of:
 - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
 - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does \star , vary ba \pm on:
 - The values of the data supplied in any of its operand reg. rs when its g erning predicate register contains the same value for each execution.
 - The values of the NZCV flags.

D1.1.115 MOV (vector to tile, two registers)

Move two vector registers to two ZA tile slices

The instruction operates on two consecutive horizontal or vertical slices within a named ZA tile of the specified element size.

The consecutive slice numbers within the tile are selected starting from the sum of the slice index register and immediate offset, modulo the number of such elements in a vector. The immediate offset is a multiple of 2 in the range 0 to the number of elements in a 128-bit vector segment minus 2.

This instruction is unpredicated.

This is an alias of MOVA (vector to tile, two registers). This means:

- The encodings in this description are named to match the encodings of VA (vector to tile, two registers).
- The description of MOVA (vector to tile, two registers) gives the *c* crational pse bcode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for is instruction.

It has encodings from 4 classes: 8-bit, 16-bit, 32-bit and 64-b

8-bit



MOV ZAO<HV>.B[<Ws>, <offsf; <offs. { <Zn1>.B-<Zn2>.B

is equivalent to

```
MOVA ZAO<HV>.B[<Ws> `sf>:<offs1 ', { <Zn1>.B-<Zn2>.B }
```

and is always the prefer d disassemb.

16-bit



MOV

1><Hv, (<Ws>, <offsf>:<offsl>], { <Zn1>.H-<Zn2>.H }

is equivalen

MOVA <ZAd><HV>.H[<Ws>, <offsf>:<offsl>], { <Zn1>.H-<Zn2>.H }

and is always the preferred disassembly.

32-bit



```
MOV <ZAd><HV>.S[<Ws>, <offsf>:<offsl>], { <Zn1>.S-<Zn2>.S }
```

is equivalent to

```
MOVA <_ZAd><HV>.S[<Ws>, <offsf>:<offsl>], { <Zn1>.S-<Zn2>.S }
```

and is always the preferred disassembly.

64-bit



```
MOV <ZAd><HV>.D[<Ws>, <offsf>:<offsl>], { <Zn1>.D-<Zn2>.D }
```

is equivalent to

MOVA <ZAd><HV>.D[<Ws>, <offsf>:<offsl>], { <Zn1>.D-<Zn2>.D }

and is always the preferred disassembly.

Assembler Symbols

<ZAd> For the 16-bit variant: is the name of the ZA tile ZA0-Z 1 to by cressed, er oded in the "ZAd" field.

> For the 32-bit variant: is the name of the ZA tile ' x0-ZA3 be acces. I, encoded in the "ZAd" field.

> For the 64-bit variant: is the name of the 7.1. ZAO-2 7 to b sccessed, encoded in the "ZAd" field.

<HV>Is the horizontal or vertical slice indic vr, en

V	<hv></hv>
0	Н
1	V

- <Ws> Is the 32-bit name of the slice dex register W12-W15, encoded in the "Rs" field.
- <offsf> For the 8-bit ariant: is the slice 1ex offset, pointing to first of two consecutive slices, encoded ? off3" fie' times 2.

For the 16-, v ant: is the slice index offset, pointing to first of two consecutive slices, 'as "on " field ti es 2. ep

.-bit varı. is the slice index offset, pointing to first of two consecutive slices, for the "¹" field times 2. encod

the 64-bit variant: is the slice index offset, pointing to first of two consecutive slices, with. pplicit value 0.

<offsl> For the 8-bit variant: is the slice index offset, pointing to last of two consecutive slices, encoded as "off3" field times 2 plus 1.

> For the 16-bit variant: is the slice index offset, pointing to last of two consecutive slices, encoded as "off2" field times 2 plus 1.

> For the 32-bit variant: is the slice index offset, pointing to last of two consecutive slices, encoded as "o1" field times 2 plus 1.

> For the 64-bit variant: is the slice index offset, pointing to last of two consecutive slices, with implicit value 1.

- <Zn1>Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.
- Is the name of the second scalable vector register of a multi-vector sequence, encoded as $\langle Zn2 \rangle$ "Zn" times 2 plus 1.
Operation

The description of MOVA (vector to tile, two registers) gives the operational pseudocode for this instruction.

Operational information

- The execution time of this instruction is independent of:
 - The values of the data supplied in any of its registers.
 - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
 - The values of the data supplied in any of its registers.
 - The values of the NZCV flags.

D1.1.116 MOV (vector to tile, four registers)

Move four vector registers to four ZA tile slices

The instruction operates on four consecutive horizontal or vertical slices within a named ZA tile of the specified element size.

The consecutive slice numbers within the tile are selected starting from the sum of the slice index register and immediate offset, modulo the number of such elements in a vector. The immediate offset is a multiple of 4 in the range 0 to the number of elements in a 128-bit vector segment minus 4.

This instruction is unpredicated.

This is an alias of MOVA (vector to tile, four registers). This means:

- The encodings in this description are named to match the encodings of th
- The description of MOVA (vector to tile, four registers) gives the curational pse 'ocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for is instruction.

It has encodings from 4 classes: 8-bit, 16-bit, 32-bit and 64-b

8-bit



MOV ZAO<HV>.B[<Ws>, <offsf; <offs. { <Zn1>.B-<Zn4>.B

is equivalent to

```
MOVA ZA0<HV>.B[<Ws> $sf>:<offs1 ', { <Zn1>.B-<Zn4>.B }
```

and is always the prefer d disassemb.

16-bit



MOV

1><Hv, (<Ws>, <offsf>:<offsl>], { <Zn1>.H-<Zn4>.H }

is equivalen

MOVA <ZAd><HV>.H[<Ws>, <offsf>:<offsl>], { <Zn1>.H-<Zn4>.H }

and is always the preferred disassembly.

32-bit



```
MOV <ZAd><HV>.S[<Ws>, <offsf>:<offsl>], { <Zn1>.S-<Zn4>.S }
```

is equivalent to

```
MOVA <_ZAd><HV>.S[<Ws>, <offsf>:<offsl>], { <Zn1>.S-<Zn4>.S }
```

and is always the preferred disassembly.

64-bit



```
MOV <ZAd><HV>.D[<Ws>, <offsf>:<offsl>], { <Zn1>.D-<Zn4>.D }
```

is equivalent to

MOVA <ZAd><HV>.D[<Ws>, <offsf>:<offsl>], { <Zn1>.D-<Zn4>.D }

and is always the preferred disassembly.

Assembler Symbols

<ZAd> For the 16-bit variant: is the name of the ZA tile ZA0-Z 1 to by cressed, er oded in the "ZAd" field.

> For the 32-bit variant: is the name of the ZA tile ' x0-ZA3 be acces. I, encoded in the "ZAd" field.

> For the 64-bit variant: is the name of the ZAO-2 7 to b ccessed, encoded in the "ZAd" field.

Is the horizontal or vertical slice indic vr, en $\langle HV \rangle$

V	<hv></hv>
0	Н
1	V

- <Ws> Is the 32-bit name of the slice dex register W12-W15, encoded in the "Rs" field.
- <offsf> For the 8-bit ariant: is the slice lex offset, pointing to first of four consecutive slices, encoded ? off2" fie' times 4.

For the 16-, v ant: is the slice index offset, pointing to first of four consecutive slices, en as "o. Geld tir 4.

For the first of four consecutive index offset, pointing to first of four consecutive icit value 0. slices

<offsl> the 8-bit variant: is the slice index offset, pointing to last of four consecutive slices, ency d as "off2" field times 4 plus 3.

> For the 16-bit variant: is the slice index offset, pointing to last of four consecutive slices, encoded as "o1" field times 4 plus 3.

> For the 32-bit and 64-bit variant: is the slice index offset, pointing to last of four consecutive slices, with implicit value 3.

- <Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.
- <Zn4>Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.

Operation

The description of MOVA (vector to tile, four registers) gives the operational pseudocode for this instruction.

Operational information

- The execution time of this instruction is independent of:
 - The values of the data supplied in any of its registers.
 - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
 - The values of the data supplied in any of its registers.
 - The values of the NZCV flags.



D1.1.117 MOV (vector to array, two registers)

Move two vector registers to two ZA single-vector groups

The instruction operates on two ZA single-vector groups. The vector numbers forming the single-vector group within each half of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half the number of ZA array vectors.

The VECTOR GROUP symbol VGx2 indicates that the instruction operates on two ZA single-vector groups.

The preferred disassembly syntax uses a 64-bit element size, but an assembler should accept any element size if it is used consistently for all operands. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

This is an alias of MOVA (vector to array, two registers). This means:

- The encodings in this description are named to match the encodin. of MOVA (ve or to array, two registers).
- The description of MOVA (vector to array, two regist) gives upperational pseudocode, any CON-STRAINED UNPREDICTABLE behavior, and any oper 2 nal information. It is instruction.

MOV ZA.D[<Wv>, <offs>{, VGx2}], <Zn2

```
is equivalent to
```

and is always the preferred assen. 'v

Assembler Symbols

<Wv> Is the 32-1 name of the vector select register W8-W11, encoded in the "Rv" field.

<offs> Is the vector 1/ . offset, ir he range 0 to 7, encoded in the "off3" field.

- <Zn1> V ine n, he of the first alable vector register of a multi-vector sequence, encoded as "Zn" arms 2,
- <Zn2> 's the name and second scalable vector register of a multi-vector sequence, encoded as times 2 plus 1.

Operation

The description of MOVA (vector to array, two registers) gives the operational pseudocode for this instruction.

Operational information

- The execution time of this instruction is independent of:
 - The values of the data supplied in any of its registers.
 - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
 - The values of the data supplied in any of its registers.
 - The values of the NZCV flags.

D1.1.118 MOV (vector to array, four registers)

Move four vector registers to four ZA single-vector groups

The instruction operates on four ZA single-vector groups. The vector numbers forming the single-vector group within each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo quarter the number of ZA array vectors.

The VECTOR GROUP symbol VGx4 indicates that the instruction operates on four ZA single-vector groups.

The preferred disassembly syntax uses a 64-bit element size, but an assembler should accept any element size if it is used consistently for all operands. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

This is an alias of MOVA (vector to array, four registers). This means:

- The encodings in this description are named to match the encoding of MOVA (ve or to array, four registers).
- The description of MOVA (vector to array, four regist 3) gives 1. operatic al pseudocode, any CON-STRAINED UNPREDICTABLE behavior, and any operation and information. Transmission of the structure of the stru

MOV ZA.D[<Wv>, <offs>{, VGx4}], <Zn1

```
is equivalent to
```

and is always the preferred insassen. 'v

Assembler Symbols

```
<Wv> Is the 32-1 name of the vector select register W8-W11, encoded in the "Rv" field.
```

<offs> Is the vector 1 offset, ir he range 0 to 7, encoded in the "off3" field.

- <Zn1> V ine n, he of the first alable vector register of a multi-vector sequence, encoded as "Zn" arms 4,
- <Zn4> 's the lame of a multi-vector sequence, encoded as "Zn" u s 4 plus 3.

Operation

The description of MOVA (vector to array, four registers) gives the operational pseudocode for this instruction.

Operational information

- The execution time of this instruction is independent of:
 - The values of the data supplied in any of its registers.
 - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
 - The values of the data supplied in any of its registers.
 - The values of the NZCV flags.

D1.1.119 MOV (vector to tile, single)

Move vector register to ZA tile slice

The instruction operates on individual horizontal or vertical slices within a named ZA tile of the specified element size. The slice number within the tile is selected by the sum of the slice index register and immediate offset, modulo the number of such elements in a vector. The immediate offset is in the range 0 to the number of elements in a 128-bit vector segment minus 1.

Inactive elements in the destination slice remain unmodified.

This is an alias of MOVA (vector to tile, single). This means:

- The encodings in this description are named to match the encodings of MOVA (vector to tile, single).
- The description of MOVA (vector to tile, single) gives the operation of Dudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for t¹ instruct.

It has encodings from 5 classes: 8-bit, 16-bit, 32-bit, 64-bit and 128

8-bit



MOV <ZAd><HV>.S[<Ws>, <offs>], <Pg>/M, <Zn>.S

is equivalent to

MOVA <ZAd><HV>.S[<Ws>, <offs>], <Pg>/M, <Zn>.S

Chapter D1. SME instructions D1.1. SME and SME2 data-processing instructions

and is always the preferred disassembly.

64-bit



Assembler Symbols

<ZAd> For the 16-bit variant: is the . me of the ZA tile ZA0-ZA1 to be accessed, encoded in the "ZAd" field.

For the 32 , a variant is the name of the ZA tile ZA0-ZA3 to be accessed, encoded in the "ZAd" fie

For 64-bh riant: is 2 name of the ZA tile ZA0-ZA7 to be accessed, encoded in the $2Ad^{"}h$ d.

For the 28 bit variant: is the name of the ZA tile ZA0-ZA15 to be accessed, encoded in the ZA tiled.

<HV> Is . horizontal or vertical slice indicator, encoded in "V":

V	'HV>
0	Н
1	V

<Ws> Is the 32-bit name of the slice index register W12-W15, encoded in the "Rs" field.

<offs> For the 8-bit variant: is the slice index offset, in the range 0 to 15, encoded in the "off4" field. For the 16-bit variant: is the slice index offset, in the range 0 to 7, encoded in the "off3" field. For the 32-bit variant: is the slice index offset, in the range 0 to 3, encoded in the "off2" field. For the 64-bit variant: is the slice index offset, in the range 0 to 1, encoded in the "o1" field. For the 128-bit variant: is the slice index offset 0.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

 $<\!\! Zn\!\!>$ Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

The description of MOVA (vector to tile, single) gives the operational pseudocode for this instruction.

Operational information

- The execution time of this instruction is independent of:
 - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
 - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does \star , vary ba \pm on:
 - The values of the data supplied in any of its operand reg₁, rs when its g erning predicate register contains the same value for each execution.
 - The values of the NZCV flags.

D1.1.120 MOVA (tile to vector, two registers)

Move two ZA tile slices to two vector registers

The instruction operates on two consecutive horizontal or vertical slices within a named ZA tile of the specified element size.

The consecutive slice numbers within the tile are selected starting from the sum of the slice index register and immediate offset, modulo the number of such elements in a vector. The immediate offset is a multiple of 2 in the range 0 to the number of elements in a 128-bit vector segment minus 2.

This instruction is unpredicated.

This instruction is used by the alias MOV (tile to vector, two registers).

It has encodings from 4 classes: 8-bit, 16-bit, 32-bit and 64-bit



size<1>」 Lsize<0>

```
MOVA { <Zdl>.S-<Zd2>.S }, <ZAn><HV>.S[<Ws>, <offsf>:<offsl>]
1
  if !HaveSME2() then UNDEFINED;
  integer s = UInt('011':Rs);
2
3
  constant integer nreg = 2;
4
  constant integer esize = 32;
5
  integer d = UInt(Zd:'0');
  integer n = UInt(ZAn);
6
  integer offset = UInt(o1:'0');
8
  boolean vertical = V == '1';
   64-bit
   (FEAT_SME2)
               0
                  0
                    0
                       0
                         0
                            0
                                      0
                                         0
                                           1
                                                                         7An
                                                                                  7d
                          size<1>
                                 Lsize<0>
  MOVA { <Zd1>.D-<Zd2>.D }, <ZAn><HV>.D[<Ws>,
                                                                    s1>1
1
  if !HaveSME2() then UNDEFINED;
2
  integer s = UInt('011':Rs);
  constant integer nreg = 2;
3
  constant integer esize = 64;
4
5
  integer d = UInt(Zd:'0');
  integer n = UInt(ZAn);
6
  integer offset = 0;
8 boolean vertical = V == '1';
```

Assembler Symbols

- <Zd1> Is the name of the first d nation scalab. rector register of a multi-vector sequence, encoded as "Zd" times 2.
- <Zd2> Is the name of the second desumation scalable vector register of a multi-vector sequence, encoded as "" at times 2 plus 1.
- <ZAn> For the 1 pit varie .. is the name of the ZA tile ZA0-ZA1 to be accessed, encoded in the "ZAn" field.

F. the 2-bit v. 2nt: ine name of the ZA tile ZA0-ZA3 to be accessed, encoded in the ZAn" f. d.

<HV> Is the prizontal or vertical slice indicator, encoded in "V":

V	<hv></hv>
0	Н
1	V

- <Ws> Is the 32-bit name of the slice index register W12-W15, encoded in the "Rs" field.
- <offsf> For the 8-bit variant: is the slice index offset, pointing to first of two consecutive slices, encoded as "off3" field times 2.

For the 16-bit variant: is the slice index offset, pointing to first of two consecutive slices, encoded as "off2" field times 2.

For the 32-bit variant: is the slice index offset, pointing to first of two consecutive slices, encoded as "01" field times 2.

For the 64-bit variant: is the slice index offset, pointing to first of two consecutive slices,

with implicit value 0.

<offsl> For the 8-bit variant: is the slice index offset, pointing to last of two consecutive slices, encoded as "off3" field times 2 plus 1.

For the 16-bit variant: is the slice index offset, pointing to last of two consecutive slices, encoded as "off2" field times 2 plus 1.

For the 32-bit variant: is the slice index offset, pointing to last of two consecutive slices, encoded as "01" field times 2 plus 1.

For the 64-bit variant: is the slice index offset, pointing to last of two consecutive slices, with implicit value 1.

Operation

```
CheckStreamingSVEAndZAEnabled();
1
2
   constant integer VL = CurrentVL;
   if nreg == 4 && esize == 64 && VL == 128 then UNDEFINED;
3
   integer slices = VL DIV esize;
4
5
   bits(32) index = X[s, 32];
6
   integer slice = ((UInt(index) - (UInt(index) MOD n
                                                                offse
                                                                         MOD
                                                                              lices;
                                                           (\Lambda)
7
8
   for r = 0 to nreg-1
       bits(VL) result = ZAslice[n, esize, vertical,
9
                                                                    VI
                                                                 r.
10
       Z[d + r, VL] = result;
```

Operational information

- The execution time of this instruction is independent of:
 - The values of the dat applied in a. of its h gisters.
 - The values of the $\angle ZCV$ rgs.
- The response of *t*'s instruction to vnchronous exceptions does not vary based on:
 - The values of the c' ta supplied in any of its registers.
 - The value of z NZCV flags.

D1.1.121 MOVA (tile to vector, four registers)

Move four ZA tile slices to four vector registers

The instruction operates on four consecutive horizontal or vertical slices within a named ZA tile of the specified element size.

The consecutive slice numbers within the tile are selected starting from the sum of the slice index register and immediate offset, modulo the number of such elements in a vector. The immediate offset is a multiple of 4 in the range 0 to the number of elements in a 128-bit vector segment minus 4.

This instruction is unpredicated.

This instruction is used by the alias MOV (tile to vector, four registers).

It has encodings from 4 classes: 8-bit, 16-bit, 32-bit and 64-bit



```
MOVA { <Zdl>.S-<Zd4>.S }, <ZAn><HV>.S[<Ws>, <offsf>:<offsl>]
1
  if !HaveSME2() then UNDEFINED;
  integer s = UInt('011':Rs);
2
3
  constant integer nreg = 4;
4
  constant integer esize = 32;
5
  integer d = UInt(Zd:'00');
  integer n = UInt(ZAn);
6
  integer offset = 0;
7
8
  boolean vertical = V == '1';
   64-bit
   (FEAT_SME2)
               0
                  0
                     0
                       0
                          0
                                      0
                                         0
                                           1
                                                                         7An
                          size<1>
                                  Lsize<0>
   MOVA { <Zd1>.D-<Zd4>.D }, <ZAn><HV>.D[<Ws>,
                                                                    s1>1
1
  if !HaveSME2() then UNDEFINED;
2
  integer s = UInt('011':Rs);
  constant integer nreg = 4;
3
  constant integer esize = 64;
4
5
  integer d = UInt(Zd:'00');
  integer n = UInt(ZAn);
6
  integer offset = 0;
8 boolean vertical = V == '1';
```

Assembler Symbols

- <Zd1> Is the name of the first d nation scalab. rector register of a multi-vector sequence, encoded as "Zd" times 4.
- <Zd4> Is the name of c fourth dest. tion scalable vector register of a multi-vector sequence, encoded as "" ' times 4 plus 3.
- <ZAn> For the 1 pit varie .. is the name of the ZA tile ZA0-ZA1 to be accessed, encoded in the "ZAn" field.

F. the 2-bit v. Int: ine name of the ZA tile ZA0-ZA3 to be accessed, encoded in the ZAn" f. d.

<HV> Is the prizontal or vertical slice indicator, encoded in "V":

V	<hv></hv>
0	Н
1	V

- <Ws> Is the 32-bit name of the slice index register W12-W15, encoded in the "Rs" field.
- <offsf> For the 8-bit variant: is the slice index offset, pointing to first of four consecutive slices, encoded as "off2" field times 4.

For the 16-bit variant: is the slice index offset, pointing to first of four consecutive slices, encoded as "o1" field times 4.

For the 32-bit and 64-bit variant: is the slice index offset, pointing to first of four consecutive slices, with implicit value 0.

<offsl> For the 8-bit variant: is the slice index offset, pointing to last of four consecutive slices,

encoded as "off2" field times 4 plus 3.

For the 16-bit variant: is the slice index offset, pointing to last of four consecutive slices, encoded as "01" field times 4 plus 3.

For the 32-bit and 64-bit variant: is the slice index offset, pointing to last of four consecutive slices, with implicit value 3.

Operation

```
1
   CheckStreamingSVEAndZAEnabled();
   constant integer VL = CurrentVL;
2
3
   if nreg == 4 && esize == 64 && VL == 128 then UNDEFINED;
   integer slices = VL DIV esize;
4
   bits(32) index = X[s, 32];
5
6
   integer slice = ((UInt(index) - (UInt(index) MOD nreg)) + offse+' MOD slices;
7
8
   for r = 0 to nreg-1
       bits(VL) result = ZAslice[n, esize, vertical, slice
                                                                  VL];
9
10
       Z[d + r, VL] = result;
```

Operational information

- The execution time of this instruction is indep____ont of.
 - The values of the data supplied in ar of its resters.
 - The values of the NZCV flags.
- The response of this instruction to vnchro. vs exceptions does not vary based on:
 - The values of the data pplied in v of its visters.
 - The values of the * _C ^qags.

D1.1.122 MOVA (array to vector, two registers)

Move two ZA single-vector groups to two vector registers

The instruction operates on two ZA single-vector groups. The vector numbers forming the single-vector group within each half of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half the number of ZA array vectors.

The VECTOR GROUP symbol VGx2 indicates that the instruction operates on two ZA single-vector groups.

The preferred disassembly syntax uses a 64-bit element size, but an assembler should accept any element size if it is used consistently for all operands. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

This instruction is used by the alias MOV (array to vector, two registers)



13 1 1 0 0 0 0 0 Θ ß Rv 0 0 0 off3 Zd 0 0 0 0 1 1 0 Θ

MOVA { <Zdl>.D-<Zd2>.D }, ZA.D[<Wv , <off >{, VG>

```
1 if !HaveSME2() then UNDEFINED;
```

```
2 integer v = UInt('010':Rv);
```

3 integer offset = UInt(off3);

```
4 integer d = UInt(Zd:'0');
5 constant integer nreg = 2;
```

Assembler Symbols

- <Zd1> Is the name of first destination hable vector register of a multi-vector sequence, encoded as "Zd" tim 2.
- $\langle Zd2 \rangle$ Is the name of the second destination scalable vector register of a multi-vector sequence, encoded as '2 aimes 2 p' s 1.

<Wv> the 32 it name '+' vector select register W8-W11, encoded in the "Rv" field.

<offs. Is the sect offset, in the range 0 to 7, encoded in the "off3" field.

Operatio.

```
1
   CheckStream. SVEAndZAEnabled();
2
   constant integer VL = CurrentVL;
   integer vectors = VL DIV 8;
3
   integer vstride = vectors DIV nreg;
4
   bits(32) vbase = X[v, 32];
5
6
   integer vec = (UInt(vbase) + offset) MOD vstride;
7
8
   for r = 0 to nreg-1
9
       bits(VL) result = ZAvector[vec, VL];
10
       Z[d + r, VL] = result;
11
       vec = vec + vstride;
```

Operational information

- The execution time of this instruction is independent of:
 - The values of the data supplied in any of its registers.

- The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
 - The values of the data supplied in any of its registers.
 - The values of the NZCV flags.



D1.1.123 MOVA (array to vector, four registers)

Move four ZA single-vector groups to four vector registers

The instruction operates on four ZA single-vector groups. The vector numbers forming the single-vector group within each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo quarter the number of ZA array vectors.

The VECTOR GROUP symbol VGx4 indicates that the instruction operates on four ZA single-vector groups.

The preferred disassembly syntax uses a 64-bit element size, but an assembler should accept any element size if it is used consistently for all operands. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

This instruction is used by the alias MOV (array to vector, four register

```
SME2
(FEAT_SME2)
```

13 1 1 0 0 0 0 0 Θ ß Rv 0 1 0 off3 Zd 0 0 0 0 1 1 0 0

```
MOVA { <Zdl>.D-<Zd4>.D }, ZA.D[<Wv , <off >{, VGx
```

```
1 if !HaveSME2() then UNDEFINED;
```

```
2 integer v = UInt('010':Rv);
```

3 integer offset = UInt(off3); 4 integer d = UInt(7d:'00');

```
4 integer d = UInt(Zd:'00');
5 constant integer nreg = 4;
```

Assembler Symbols

- <Zd1> Is the name of first destination hable vector register of a multi-vector sequence, encoded as "Zd" tim 4.
- $\langle Zd4 \rangle$ Is the name of the fourth destination scalable vector register of a multi-vector sequence, encoded as '2 times 4 r' 33.

<Wv> the 32 it name '+' vector select register W8-W11, encoded in the "Rv" field.

<offs. Is the sect offset, in the range 0 to 7, encoded in the "off3" field.

Operatio.

```
1
   CheckStream. SVEAndZAEnabled();
2
   constant integer VL = CurrentVL;
   integer vectors = VL DIV 8;
3
   integer vstride = vectors DIV nreg;
4
   bits(32) vbase = X[v, 32];
5
6
   integer vec = (UInt(vbase) + offset) MOD vstride;
7
8
   for r = 0 to nreg-1
9
       bits(VL) result = ZAvector[vec, VL];
10
       Z[d + r, VL] = result;
11
       vec = vec + vstride;
```

Operational information

- The execution time of this instruction is independent of:
 - The values of the data supplied in any of its registers.

- The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
 - The values of the data supplied in any of its registers.
 - The values of the NZCV flags.



D1.1.124 MOVA (tile to vector, single)

Move ZA tile slice to vector register

The instruction operates on individual horizontal or vertical slices within a named ZA tile of the specified element size. The slice number within the tile is selected by the sum of the slice index register and immediate offset, modulo the number of such elements in a vector. The immediate offset is in the range 0 to the number of elements in a 128-bit vector segment minus 1.

Inactive elements in the destination vector remain unmodified.

This instruction is used by the alias MOV (tile to vector, single).

It has encodings from 5 classes: 8-bit, 16-bit, 32-bit, 64-bit and 128-bit



MOVA <Zd>.S, <Pg>/M, <ZAn><HV>.S[<Ws>, <offs>]

```
Chapter D1. SME instructions
D1.1. SME and SME2 data-processing instructions
```

- 1 if !HaveSME() then UNDEFINED;
- 2 integer g = UInt(Pg);
- 3 integer s = UInt('011':Rs);
- 4 integer n = UInt(ZAn);
- 5 integer offset = UInt(off2);
- 6 constant integer esize = 32; 7 integer d = UInt(7d):
- 7 integer d = UInt(Zd);
- 8 boolean vertical = V == '1';

64-bit (FEAT_SME)



```
1 if !HaveSME() the DEFINED
2 integer Int (Ps
3 integer S = Lot ('01) Rs
```

- 3 intege s = t it('01. Re
 4 inte .r n = U it(ZAn);
- 5 int r off
- 6 const .eger e. .e = 128;
- 7 integer = UInt(Zd);
- 8 boolean v +ical = V == '1';

Assembler Symbols

- <Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <ZAn> For the 16-bit variant: is the name of the ZA tile ZA0-ZA1 to be accessed, encoded in the "ZAn" field.

For the 32-bit variant: is the name of the ZA tile ZA0-ZA3 to be accessed, encoded in the "ZAn" field.

For the 64-bit variant: is the name of the ZA tile ZA0-ZA7 to be accessed, encoded in the "ZAn" field.

For the 128-bit variant: is the name of the ZA tile ZA0-ZA15 to be accessed, encoded in the "ZAn" field.

<HV> Is the horizontal or vertical slice indicator, encoded in "V":

V	<hv></hv>
0	Н
1	V

- <Ws> Is the 32-bit name of the slice index register W12-W15, encoded in the "Rs" field.
- <offs> For the 8-bit variant: is the slice index offset, in the range 0 to 15, encoded in the "off4" field. For the 16-bit variant: is the slice index offset, in the range 0 to 7, encoded in the "off3" field. For the 32-bit variant: is the slice index offset, in the range 0 to 3, encoded in the "off2" field. For the 64-bit variant: is the slice index offset, in the range 0 to 1, encoded in the "o1" field. For the 128-bit variant: is the slice index offset 0.

Operation

```
CheckStreamingSVEAndZAEnabled();
1
   constant integer VL = CurrentVL;
2
3
   constant integer PL = VL DIV 8;
   constant integer dim = VL DIV esize;
4
5
   bits(PL) mask = P[g, PL];
6
   bits(32) index = X[s, 32];
   integer slice = (UInt(index) + offset)
                                               Jdi
7
8
   bits(VL) operand = ZAslice[n, esize, v cical,
                                                      lice,
9
   bits(VL) result = Z[d, VL];
10
11
   for e = 0 to dim-1
       bits(esize) element = Elemfope rd, e, rize];
12
       if ActivePredicateElemen* .ask,
13
                                            esize hen
                                 3] = eleme.
14
            Elem[result, e, es<sup>;</sup>
15
   Z[d, VL] = result;
16
```

Operational inform

If PSTATE.DIT : 1:

- The execution \checkmark of this in action is independent of:
 - The lues of e d a supplied in any of its operand registers when its governing predicate register contains the same alue for each execution.

T' values ne NZCV flags.

- The . ponse of this instruction to asynchronous exceptions does not vary based on:
 - The alues of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
 - The values of the NZCV flags.

D1.1.125 MOVA (vector to tile, two registers)

Move two vector registers to two ZA tile slices

The instruction operates on two consecutive horizontal or vertical slices within a named ZA tile of the specified element size.

The consecutive slice numbers within the tile are selected starting from the sum of the slice index register and immediate offset, modulo the number of such elements in a vector. The immediate offset is a multiple of 2 in the range 0 to the number of elements in a 128-bit vector segment minus 2.

This instruction is unpredicated.

This instruction is used by the alias MOV (vector to tile, two registers).

It has encodings from 4 classes: 8-bit, 16-bit, 32-bit and 64-bit



```
MOVA <_ZAd><HV>.S[<Ws>, <offsf>:<offsl>], { <Zn1>.S-<Zn2>.S }
1
  if !HaveSME2() then UNDEFINED;
   integer s = UInt('011':Rs);
2
3
  constant integer nreg = 2;
4
  constant integer esize = 32;
5
  integer n = UInt(Zn:'0');
  integer d = UInt(ZAd);
6
  integer offset = UInt(o1:'0');
8
  boolean vertical = V == '1';
   64-bit
   (FEAT_SME2)
                  0 0
                          0
                            0 1
                                      0
                                         0
                                            1
                                                                             0
                                                                               0
                                                                                  0
                                                                                      7Ad
                          size<1>
                                  Lsize<0>
   MOVA <ZAd><HV>.D[<Ws>, <offsf>:<offsl>], {
                                                                     .D }
1
  if !HaveSME2() then UNDEFINED;
2
  integer s = UInt('011':Rs);
  constant integer nreg = 2;
3
  constant integer esize = 64;
4
5
  integer n = UInt(Zn:'0');
  integer d = UInt(ZAd);
6
  integer offset = 0;
  boolean vertical = V == '1';
8
   Assembler Symbols
```

<ZAd> For the 16-bit variant: 're name of th. 'A tile ZA0-ZA1 to be accessed, encoded in the "ZAd" field.

For the 32-bit v _ant: is the na _ of the ZA tile ZA0-ZA3 to be accessed, encoded in the "ZAd" field.

For the ℓ pit varie \cdot is the name of the ZA tile ZA0-ZA7 to be accessed, encoded in the "ZAd" field.

<HV> V me n 'zonta' ver' al slice indicator, encoded in "V":



- <Ws> Is the 3.-bit name of the slice index register W12-W15, encoded in the "Rs" field.
- <offsf> For the 8-bit variant: is the slice index offset, pointing to first of two consecutive slices, encoded as "off3" field times 2.

For the 16-bit variant: is the slice index offset, pointing to first of two consecutive slices, encoded as "off2" field times 2.

For the 32-bit variant: is the slice index offset, pointing to first of two consecutive slices, encoded as "o1" field times 2.

For the 64-bit variant: is the slice index offset, pointing to first of two consecutive slices, with implicit value 0.

<offsl> For the 8-bit variant: is the slice index offset, pointing to last of two consecutive slices, encoded as "off3" field times 2 plus 1.

For the 16-bit variant: is the slice index offset, pointing to last of two consecutive slices,

encoded as "off2" field times 2 plus 1.

For the 32-bit variant: is the slice index offset, pointing to last of two consecutive slices, encoded as "01" field times 2 plus 1.

For the 64-bit variant: is the slice index offset, pointing to last of two consecutive slices, with implicit value 1.

- <Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.
- <Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.

Operation

```
CheckStreamingSVEAndZAEnabled();
1
2
   constant integer VL = CurrentVL;
   if nreg == 4 && esize == 64 && VL == 128 then UNDEFINED;
3
   integer slices = VL DIV esize;
4
5
   bits(32) index = X[s, 32];
6
   integer slice = ((UInt(index) - (UInt(index) MOD n
                                                                offse
                                                                         MOD
                                                                              lices;
                                                          (\Lambda)
7
8
   for r = 0 to nreg-1
9
       bits(VL) result = Z[n + r, VL];
10
                                                     = re.
       ZAslice[d, esize, vertical, slice + r,
```

Operational information

- The execution time of this instruction is independent of:
 - The values of the data applied in a of its a gisters.
 - The values of the $\angle ZCV$ yes.
- The response of *t*'s instruction to vnchronous exceptions does not vary based on:
 - The values of the c' ta supplied in any of its registers.
 - The value of z NZCV flags.

D1.1.126 MOVA (vector to tile, four registers)

Move four vector registers to four ZA tile slices

The instruction operates on four consecutive horizontal or vertical slices within a named ZA tile of the specified element size.

The consecutive slice numbers within the tile are selected starting from the sum of the slice index register and immediate offset, modulo the number of such elements in a vector. The immediate offset is a multiple of 4 in the range 0 to the number of elements in a 128-bit vector segment minus 4.

This instruction is unpredicated.

This instruction is used by the alias MOV (vector to tile, four registers).

It has encodings from 4 classes: 8-bit, 16-bit, 32-bit and 64-bit





```
MOVA <_ZAd><HV>.S[<Ws>, <offsf>:<offsl>], { <Zn1>.S-<Zn4>.S }
1
  if !HaveSME2() then UNDEFINED;
   integer s = UInt('011':Rs);
2
3
  constant integer nreg = 4;
4
  constant integer esize = 32;
5
  integer n = UInt(Zn:'00');
  integer d = UInt(ZAd);
6
  integer offset = 0;
7
8
  boolean vertical = V == '1';
   64-bit
   (FEAT_SME2)
                  0 0
                         0
                                      0
                                         0
                                            1
                                                                               0
                                                                                      7Ad
                          size<1>
                                  _size<0
   MOVA <ZAd><HV>.D[<Ws>, <offsf>:<offsl>], {
                                                                     D }
1
  if !HaveSME2() then UNDEFINED;
2
  integer s = UInt('011':Rs);
  constant integer nreg = 4;
3
  constant integer esize = 64;
4
5
  integer n = UInt(Zn:'00');
  integer d = UInt(ZAd);
6
  integer offset = 0;
  boolean vertical = V == '1';
8
   Assembler Symbols
```

<ZAd> For the 16-bit variant: pe name of the "A tile ZA0-ZA1 to be accessed, encoded in the "ZAd" field.

For the 32-bit v _ant: is the na _ of the ZA tile ZA0-ZA3 to be accessed, encoded in the "ZAd" field.

For the ℓ pit varie \cdot is the name of the ZA tile ZA0-ZA7 to be accessed, encoded in the "ZAd" field.

<HV> V me n 'zonta' ver' al slice indicator, encoded in "V":

V < V0V

- <Ws> Is the 3.-bit name of the slice index register W12-W15, encoded in the "Rs" field.
- <offsf> For the 8-bit variant: is the slice index offset, pointing to first of four consecutive slices, encoded as "off2" field times 4.

For the 16-bit variant: is the slice index offset, pointing to first of four consecutive slices, encoded as "o1" field times 4.

For the 32-bit and 64-bit variant: is the slice index offset, pointing to first of four consecutive slices, with implicit value 0.

<offsl> For the 8-bit variant: is the slice index offset, pointing to last of four consecutive slices, encoded as "off2" field times 4 plus 3.

For the 16-bit variant: is the slice index offset, pointing to last of four consecutive slices, encoded as "01" field times 4 plus 3.

For the 32-bit and 64-bit variant: is the slice index offset, pointing to last of four consecutive

slices, with implicit value 3.

- <Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.
- <Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.

Operation

```
1
   CheckStreamingSVEAndZAEnabled();
   constant integer VL = CurrentVL;
2
3
   if nreg == 4 && esize == 64 && VL == 128 then UNDEFINED;
   integer slices = VL DIV esize;
4
5
   bits(32) index = X[s, 32];
6
   integer slice = ((UInt(index) - (UInt(index) MOD nreg)) + offse+' MOD slices;
7
8
   for r = 0 to nreg-1
       bits(VL) result = Z[n + r, VL];
9
10
       ZAslice[d, esize, vertical, slice + r, VL] = result;
```

Operational information

- The execution time of this instruction is indep____ont of.
 - The values of the data supplied in ar of its resters.
 - The values of the NZCV flags.
- The response of this instruction to vnchro. vs exceptions does not vary based on:
 - The values of the data pplied in . y of its gisters.
 - The values of the * _C ^qags.

D1.1.127 MOVA (vector to array, two registers)

Move two vector registers to two ZA single-vector groups

The instruction operates on two ZA single-vector groups. The vector numbers forming the single-vector group within each half of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half the number of ZA array vectors.

The VECTOR GROUP symbol VGx2 indicates that the instruction operates on two ZA single-vector groups.

The preferred disassembly syntax uses a 64-bit element size, but an assembler should accept any element size if it is used consistently for all operands. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

This instruction is used by the alias MOV (vector to array, two registers)

```
SME2
(FEAT_SME2)
```

13 1 1 0 0 0 0 0 0 0 0 Rv 0 0 0 0 0 off3 0 0 0 1 0 0

MOVA ZA.D[<Wv>, <offs>{, VGx2}], { Zn1>.1 <Z

```
1 if !HaveSME2() then UNDEFINED;
```

```
2 integer v = UInt('010':Rv);
```

3 integer offset = UInt(off3);

```
4 integer n = UInt(Zn:'0');
5 constant integer nreg = 2;
```

Assembler Symbols

- <Wv> Is the 32-bit p is the vector s is ct register W8-W11, encoded in the "Rv" field.
- $\langle offs \rangle$ Is the vect select of et, in the range 0 to 7, encoded in the "off3" field.
- <Zn1> Is the name `t' urst scal²' 'e vector register of a multi-vector sequence, encoded as "Zn" tir

<Zn2> is the n; is of the ond scalable vector register of a multi-vector sequence, encoded as "Zn" t⁻¹ - 1 us 1.

Operatio.

```
1
   CheckStream. SVEAndZAEnabled();
2
   constant integer VL = CurrentVL;
3
   integer vectors = VL DIV 8;
   integer vstride = vectors DIV nreg;
4
   bits(32) vbase = X[v, 32];
5
6
   integer vec = (UInt(vbase) + offset) MOD vstride;
7
8
   for r = 0 to nreg-1
9
       bits(VL) result = Z[n + r, VL];
10
       ZAvector[vec, VL] = result;
11
       vec = vec + vstride;
```

Operational information

- The execution time of this instruction is independent of:
 - The values of the data supplied in any of its registers.

- The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
 - The values of the data supplied in any of its registers.
 - The values of the NZCV flags.



D1.1.128 MOVA (vector to array, four registers)

Move four vector registers to four ZA single-vector groups

The instruction operates on four ZA single-vector groups. The vector numbers forming the single-vector group within each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo quarter the number of ZA array vectors.

The VECTOR GROUP symbol VGx4 indicates that the instruction operates on four ZA single-vector groups.

The preferred disassembly syntax uses a 64-bit element size, but an assembler should accept any element size if it is used consistently for all operands. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

This instruction is used by the alias MOV (vector to array, four register

```
SME2
(FEAT_SME2)
```

13 0 1 1 0 0 0 0 0 0 0 Rv 0 1 0 0 0 0 off3 0 0 0 1 0 0

MOVA ZA.D[<Wv>, <offs>{, VGx4}], { Zn1>.1 <Zr

```
1 if !HaveSME2() then UNDEFINED;
```

```
2 integer v = UInt('010':Rv);
```

3 integer offset = UInt(off3); 4 integer n = UInt(7n:'00');

```
4 integer n = UInt(Zn:'00');
5 constant integer nreg = 4;
```

Assembler Symbols

- <Wv> Is the 32-bit p is the vector s is ct register W8-W11, encoded in the "Rv" field.
- $\langle offs \rangle$ Is the vect select of et, in the range 0 to 7, encoded in the "off3" field.
- <Zn1> Is the name `t' arst scale''' vector register of a multi-vector sequence, encoded as "Zn" tir
- <Zn4> is the name of the normalized in scalable vector register of a multi-vector sequence, encoded as "Zn" times

Operatio.

```
1
   CheckStream. SVEAndZAEnabled();
2
   constant integer VL = CurrentVL;
   integer vectors = VL DIV 8;
3
   integer vstride = vectors DIV nreg;
4
   bits(32) vbase = X[v, 32];
5
6
   integer vec = (UInt(vbase) + offset) MOD vstride;
7
8
   for r = 0 to nreg-1
9
       bits(VL) result = Z[n + r, VL];
10
       ZAvector[vec, VL] = result;
11
       vec = vec + vstride;
```

Operational information

- The execution time of this instruction is independent of:
 - The values of the data supplied in any of its registers.

- The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
 - The values of the data supplied in any of its registers.
 - The values of the NZCV flags.



D1.1.129 MOVA (vector to tile, single)

Move vector register to ZA tile slice

The instruction operates on individual horizontal or vertical slices within a named ZA tile of the specified element size. The slice number within the tile is selected by the sum of the slice index register and immediate offset, modulo the number of such elements in a vector. The immediate offset is in the range 0 to the number of elements in a 128-bit vector segment minus 1.

Inactive elements in the destination slice remain unmodified.

This instruction is used by the alias MOV (vector to tile, single).

It has encodings from 5 classes: 8-bit, 16-bit, 32-bit, 64-bit and 128-bit



MOVA <ZAd><HV>.S[<Ws>, <offs>], <Pg>/M, <Zn>.S

```
Chapter D1. SME instructions
D1.1. SME and SME2 data-processing instructions
```

- 1 if !HaveSME() then UNDEFINED;
- 2 integer g = UInt(Pg);
- 3 integer s = UInt('011':Rs);
- 4 integer n = UInt(Zn);
- 5 integer d = UInt(ZAd);
- 6 integer offset = UInt(off2); 7 constant integer esize = 32;
- 8 boolean vertical = V == '1';

64-bit (FEAT_SME)



Assembler S_____bols

<ZAd> For the 16-bit variant: is the name of the ZA tile ZA0-ZA1 to be accessed, encoded in the "ZAd" field.

For the 32-bit variant: is the name of the ZA tile ZA0-ZA3 to be accessed, encoded in the "ZAd" field.

For the 64-bit variant: is the name of the ZA tile ZA0-ZA7 to be accessed, encoded in the "ZAd" field.

For the 128-bit variant: is the name of the ZA tile ZA0-ZA15 to be accessed, encoded in the "ZAd" field.

<HV> Is the horizontal or vertical slice indicator, encoded in "V":

- <Ws> Is the 32-bit name of the slice index register W12-W15, encoded in the "Rs" field.
- <offs> For the 8-bit variant: is the slice index offset, in the range 0 to 15, encoded in the "off4" field. For the 16-bit variant: is the slice index offset, in the range 0 to 7, encoded in the "off3" field. For the 32-bit variant: is the slice index offset, in the range 0 to 3, encoded in the "off2" field. For the 64-bit variant: is the slice index offset, in the range 0 to 1, encoded in the "o1" field. For the 128-bit variant: is the slice index offset 0.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

```
CheckStreamingSVEAndZAEnabled();
1
2
   constant integer VL = CurrentVL;
3
   constant integer PL = VL DIV 8;
   constant integer dim = VL DIV esize;
4
5
   bits(PL) mask = P[g, PL];
6
   bits(VL) operand = Z[n, VL];
7
   bits(32) index = X[s, 32];
   integer slice = (UInt(index) + offset) MC
8
9
   bits(VL) result = ZAslice[d, esize, ver+
                                               .al.
                                                      ice.
10
11
   for e = 0 to dim-1
       bits(esize) element = Elem[operand,
12
                                                   1zel:
13
       if ActivePredicateElement(mas
                                         e, es
                                                 ) then
14
           Elem[result, e, esize' = e.
                                           ent:
15
   ZAslice[d, esize, vertica]
                                 slice, VL]
16
                                               esul.;
```

Operational information

If PSTATE.DIT is 1[.]

- The execution is independent of:
 - The value the data pplied in any of its operand registers when its governing predicate register controls the the value for each execution.
 - The Jues of the JZCV flags.
- r ponse of uns instruction to asynchronous exceptions does not vary based on:
 - values of the data supplied in any of its operand registers when its governing predicate register cont as the same value for each execution.
 - The values of the NZCV flags.

D1.1.130 MOVT (ZT0 to scalar)

Move 8 bytes from ZT0 to general-purpose register

Move 8 bytes to a general-purpose register from the ZT0 register at the byte offset specified by the immediate index. This instruction is UNDEFINED in Non-debug state.

SME2 (FEAT_SME2)



```
MOVT <Xt>, ZT0[<offs>]
```

```
1 if !HaveSME2() || !Halted() then UNDEFINED;
2 integer t = UInt(Rt);
```

```
3 integer offset = UInt(off3);
```

Assembler Symbols

<Xt> Is the 64-bit name of the general-purpose register to the sferred, excluded in the "Rt" field.

< offs > Is the immediate byte offset, a multiple 6 in 2 range 6 t - 36, encoded in the "off3" field as < offs > /8.

Operation

```
1 CheckSMEEnabled();
```

- 2 CheckSMEZT0Enabled();
- 3 bits(512) operand = ZT0[57
 4
- 5 X[t, 64] = Elem[operand, offse 64];
D1.1.131 MOVT (scalar to ZT0)

Move 8 bytes from general-purpose register to ZTO

Move 8 bytes to the ZT0 register at the byte offset specified by the immediate index from a general-purpose register. This instruction is UNDEFINED in Non-debug state.

SME2 (FEAT_SME2)



MOVT ZT0[<offs>], <Xt>

```
1 if !HaveSME2() || !Halted() then UNDEFINED;
2 integer t = UInt(Rt);
```

3 integer offset = UInt(off3);

Assembler Symbols

- <offs> Is the immediate byte offset, a multiple of 8 in the r of 0 to 56 ncoded in the "off3" field as <offs>/8.
- <Xt> Is the 64-bit name of the general-purper register to be trans. ed, encoded in the "Rt" field.

```
1 CheckSMEEnabled();
2 CheckSMEZTOEnabled();
3 bits(512) result = ZTO[512
4
5 Elem[result, offset, 64, = X[t 64];
6 ZTO[512] = result;
```

D1.1.132 PEXT (predicate)

Set predicate from predicate-as-counter

Expands the source predicate-as-counter into a four-predicate wide mask and copies one quarter of it into the destination predicate register.

SME2 (FEAT_SME2)



```
PEXT <Pd>.<T>, <PNn>[<imm>]
```

```
1 if !HaveSME2() then UNDEFINED;
```

```
2 constant integer esize = 8 << UInt(size);</pre>
```

```
3 integer n = UInt('1':PNn);
4 integer d = UInt(Pd);
```

```
4 integer d = UInt(Pd);
5 integer part = UInt(imm2);
```

Assembler Symbols

- <Pd> Is the name of the destination scalable pre_cate_gister, 'code_in the "Pd" field.
- <T> Is the size specifier, encoded in "size"

size	<t></t>	
00	В	
01	Н	
10	S	
11	D	

<PNn> Is the name of c first source scale. predicate register PN8-PN15, with predicate-as-counter encoding, coded in the "PNn" field.

```
<imm> Is the elen. tine', in the range 0 to 3, encoded in the "imm2" field.
```

Operat' .1

```
1
   Che streamir SVEEnable ();
2
   cons nt ir
                          CurrentVL;
             .teger PL = VL DIV 8;
3
   consta
4
   constant nteger elements = VL DIV esize;
5
   bits(PL) \downarrow d = P[n, PL];
   bits(PL*4) h k = CounterToPredicate(pred<15:0>, PL*4);
6
7
   bits(PL) result;
8
   constant integer psize = esize DIV 8;
9
10
   for e = 0 to elements-1
11
       bit = PredicateElement(mask, part * elements + e, esize);
12
       Elem[result, e, psize] = ZeroExtend(pbit, psize);
13
14
   P[d, PL] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
 - The values of the data supplied in any of its registers.
 - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
 - The values of the data supplied in any of its registers.
 - The values of the NZCV flags.



D1.1.133 PEXT (predicate pair)

Set pair of predicates from predicate-as-counter

Expands the source predicate-as-counter into a four-predicate wide mask and copies two quarters of it into the destination predicate registers.

SME2 (FEAT_SME2)



```
PEXT { <Pd1>.<T>, <Pd2>.<T> }, <PNn>[<imm>]
```

```
1 if !HaveSME2() then UNDEFINED;
```

```
2 constant integer esize = 8 << UInt(size);</pre>
```

```
3 integer n = UInt('1':PNn);
```

```
4 integer d0 = UInt(Pd);
```

```
5 integer d1 = (UInt(Pd) + 1) MOD 16;
```

```
6 integer part = UInt(i1);
```

Assembler Symbols

<Pd1> Is the name of the first destination scal he predi te registe hcoded in the "Pd" field.

<T> Is the size specifier, encoded in "size"

В
Н
S
D

- <Pd2> Is the name of the second destination scalable predicate register, encoded in the "Pd" field.
- <PNn> Is the name "the st source calable predicate register PN8-PN15, with predicate-as-counter encoding, encoding, encoding the "An" field.

<imm> s the element ino. the range 0 to 1, encoded in the "i1" field.

Oper. 'on

```
CheckStr ringSVEEnabled();
1
2
   constant i. yer VL = CurrentVL;
   constant int _er PL = VL DIV 8;
3
4
   constant integer elements = VL DIV esize;
5
   bits(PL) pred = P[n, PL];
6
   bits(PL*4) mask = CounterToPredicate(pred<15:0>, PL*4);
7
   bits(PL) result0;
   bits(PL) result1;
8
9
   constant integer psize = esize DIV 8;
10
11
   for e = 0 to elements-1
       bit pbit = PredicateElement(mask, part * 2 * elements + e, esize);
12
13
        Elem[result0, e, psize] = ZeroExtend(pbit, psize);
14
   for e = 0 to elements-1
15
16
       bit pbit = PredicateElement(mask, part * 2 * elements + elements + e, esize);
17
       Elem[result1, e, psize] = ZeroExtend(pbit, psize);
18
19
   P[d0, PL] = result0;
20 P[d1, PL] = result1;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
 - The values of the data supplied in any of its registers.
 - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
 - The values of the data supplied in any of its registers.
 - The values of the NZCV flags.



D1.1.134 PTRUE

Initialise predicate-as-counter to all active

Set the destination predicate as all-active elements, using the predicate-as-counter encoding.

SME2 (FEAT_SME2)



PTRUE <PNd>.<T>

- 1 if !HaveSME2() then UNDEFINED;
- 2 constant integer esize = 8 << UInt(size);</pre>
- 3 integer d = UInt('1':PNd);

Assembler Symbols

- <PNd> Is the name of the destination scalable predicate reg or PN8 15, with predicate-as-counter encoding, encoded in the "PNd" field.
 - <T> Is the size specifier, encoded in "size":

size	<t></t>
00	В
01	Н
10	S
11	D

Operation

```
1 CheckStreamingSVT labled();
2 constant integr VL = Cr centVL;
3 constant integr PL = ___ DIV 8;
4 constant integer 1r ents = Vr DIV esize;
5 bits(PL) lt = rodePre ount(esize, elements, elements, FALSE, PL);
6 P[d, Pr = re ult;
```

Ope tional tion

If PSTA. OIT is 1:

- The exc tion time of this instruction is independent of:
 - The values of the data supplied in any of its registers.
 - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
 - The values of the data supplied in any of its registers.
 - The values of the NZCV flags.

D1.1.135 RDSVL

Read multiple of Streaming SVE vector register size to scalar register

Multiply the Streaming SVE vector register size in bytes by an immediate in the range -32 to 31 and place the result in the 64-bit destination general-purpose register.

This instruction does not require the PE to be in Streaming SVE mode.

SME (FEAT_SME)



RDSVL <Xd>, #<imm>

```
1 if !HaveSME() then UNDEFINED;
```

```
2 integer d = UInt(Rd);
```

3 integer imm = SInt(imm6);

Assembler Symbols

<Xd> Is the 64-bit name of the destination gene .1-put se reg. r, e uded in the "Rd" field.

<imm> Is the signed immediate operand, in the "range -2" to 31, encoded in the "imm6" field.

Operation

```
1 CheckSMEEnabled();
2 constant integer SVL = Cure .itSVL;
```

- 3 integer len = imm * (SVI 1 ?);
- 4 X[d, 64] = len<63:0>;

Operational inform .on

If PSTATE.DIT ' 1:

• The execution • of this in action is independent of:

The 'ues of d' supplied in any of its registers.

- The dues of the NZCV flags.

- 1. sponse of this instruction to asynchronous exceptions does not vary based on:
 - 1. values of the data supplied in any of its registers.
 - The values of the NZCV flags.

D1.1.136 SCLAMP

Multi-vector signed clamp to minimum/maximum vector

Clamp each signed element in the two or four destination vectors to between the signed minimum value in the corresponding element of the first source vector and the signed maximum value in the corresponding element of the second source vector and destructively place the clamped results in the corresponding elements of the two or four destination vectors.

This instruction is unpredicated.

It has encodings from 2 classes: Two registers and Four registers



Assembler Symbols

<Zd1> For the two registers variant: is the name of the first destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 2.

For the four registers variant: is the name of the first destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 4.

<T> Is the size specifier, encoded in "size":

size	< T >
00	В
01	Н
10	S
11	D

- <Zd4> Is the name of the fourth destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 4 plus 3.
- <Zd2> Is the name of the second destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 2 plus 1.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
- <Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

```
CheckStreamingSVEEnabled();
1
2
   constant integer VL = CurrentVL;
   constant integer elements = VL DIV esize;
3
4
   array [0..3] of bits(VL) results;
5
6
   for r = 0 to nreg-1
7
       bits(VL) operand1 = Z[n, VL];
8
       bits(VL) operand2 = Z[m, VL];
       bits(VL) operand3 = Z[d+r, VL];
9
10
       for e = 0 to elements-1
                                                       size]);
11
            integer element1 = SInt(Elem[operand1, e,
            integer element2 = SInt(Elem[operand2,
12
                                                       esizel
            integer element3 = SInt(Elem[operand3, e, siz );
13
14
            integer res = Min(Max(element1, element3),
                                                           ment.2):
15
            Elem[results[r], e, esize] = res
                                                     :0>;
16
   for r = 0 to nreg-1
17
18
       Z[d+r, VL] = results[r];
```

D1.1.137 SCVTF

Multi-vector signed integer convert to floating-point

Convert to single-precision from signed 32-bit integer, each element of the two or four source vectors, and place the results in the corresponding elements of the two or four destination vectors.

This instruction follows SME2 floating-point numerical behaviors corresponding to instructions that place their results in one or more SVE Z vectors.

This instruction is unpredicated.

It has encodings from 2 classes: Two registers and Four registers

Two registers (FEAT_SME2)



Assembler Symbols

<Zd1> For the two registers variant: is the name of the first destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 2.

For the four registers variant: is the name of the first destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 4.

- <Zd4> Is the name of the fourth destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 4 plus 3.
- <Zd2> Is the name of the second destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 2 plus 1.
- <Zn1> For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.

For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

- <Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.
- <Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.

```
CheckStreamingSVEEnabled();
1
2
   constant integer VL = CurrentVL;
3
   constant integer elements = VL DIV 32;
4
   array [0..3] of bits(VL) results;
5
6
   for r = 0 to nreg-1
7
       bits(VL) operand = Z[n+r, VL];
       for e = 0 to elements-1
8
           bits(32) element = Elem[operand, e, 32];
9
10
           Elem[results[r], e, 32] = FixedToFP(element,
                                                             unsig
                                                                       FPCR
                                                                                rounding, 32);
11
   for r = 0 to nreg-1
12
13
      Z[d+r, VL] = results[r];
```

D1.1.138 SDOT (2-way, multiple and indexed vector)

Multi-vector signed integer dot-product by indexed element

The instruction operates on two or four ZA single-vector groups.

The signed integer dot product instruction computes the dot product of two signed 16-bit integer values held in each 32-bit element of the two or four first source vectors and two signed 16-bit integer values in the corresponding indexed 32-bit element of the second source vector. The widened dot product result is destructively added to the corresponding 32-bit element of the two or four ZA single-vector groups.

The groups within the second source vector are specified using an immediate element index which selects the same group position within each 128-bit vector segment. The index range is from 0 to 3, encoded in 2 bits. The vector numbers forming the single-vector group within each half or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the the of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA oper disconsists of voor four ZA single-vector groups respectively. The VECTOR GROUP symbol is preferred for disconsists of the vector on a sembler source code.

This instruction is unpredicated.

It has encodings from 2 classes: Two ZA single-vectors a. Four' A single-vectors

Two ZA single-vectors (FEAT_SME2)



SDOT ZA.S[<Wv>, <offs>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H[<index>]

```
1 if !HaveSME2() then UNDEFINED;
```

```
2 integer v = UInt('010':Rv);
```

```
3 constant integer esize = 32;
```

```
4 integer n = UInt(Zn:'00');
5 integer m = UInt('0':Zm);
```

```
5 integer m = UInt('0':Zm);
6 integer offset = UInt(off3);
```

```
7 integer index = UInt(i2);
```

```
8 constant integer nreg = 4;
```

Assembler Symbols

- <Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
- <offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.
- <Zn1> For the two ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.

For the four ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

- <Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.
- <Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.
- <Zm> Is the name of the second source scalable vector register Z0-Z['], encoded in "Zm" field.

<index> Is the element index, in the range 0 to 3, encoded in the "?" field

```
CheckStreamingSVEAndZAEnabled();
 1
 2
   constant integer VL = CurrentVL;
   constant integer elements = VL DIV esize;
 3
4
   integer vectors = VL DIV 8;
    integer vstride = vectors DIV nreg;
 5
   integer eltspersegment = 128 DIV esiz
6
7
   bits(32) vbase = X[v, 32];
 8
   integer vec = (UInt(vbase) + offs
                                             MOD
                                                     ride;
9
   bits(VL) result;
10
11
   for r = 0 to nreg-1
                              , nh
12
        bits(VL) operand1 =
                                      VL];
        bits(VL) operand2 = Z[m,
13
14
        bits(VL) operanc' = ZAvector
                                          ec, VL];
15
        for e = 0 to \epsilon .ments-1
            bits(esi ) sum = Elem[oper.d3, e, esize];
intege segment se = e - (e MOD eltspersegment);
16
17
             intege.
18
                       = s _mentbase + index;
19
             for i =
                           4
                   `tege. lemen* = SInt(Elem[operand1, 2 * e + i, esize DIV 2]);
20
                 1 :eger m 2 = SInt(Elem[operand2, 2 * s + i, esize DIV 2]);
21
                             .lement1 * element2;
                 s n = sum
22
                       '+ e, esize] = sum;
'-_ = result;
23
             Elei
24
             c' ⊥[vec,
25
              vec + vstride;
        vec
```

D1.1.139 SDOT (2-way, multiple and single vector)

Multi-vector signed integer dot-product by vector

The instruction operates on two or four ZA single-vector groups.

The signed integer dot product instruction computes the dot product of two signed 16-bit integer values held in each 32-bit element of the two or four first source vectors and two signed 16-bit integer values in the corresponding 32-bit element of the second source vector. The widened dot product result is destructively added to the corresponding 32-bit element of the two or four ZA single-vector groups. The vector numbers forming the single-vector group within each half or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The VECTOR GROUP symbol is preferred for disasser "t optional in assembler source code.

This instruction is unpredicated.

It has encodings from 2 classes: Two ZA single-vectors and F ZA sing. vectors

Two ZA single-vectors (FEAT_SME2)



<Zh SDOT $ZA.S[\langle Wv \rangle, \langle offs \rangle \}$ $H - \langle 7n2 \rangle, H \}, \langle 7m \rangle, H$ Sx2

```
if !HaveSME2() then UNDEF
1
```

- integer v = UInt('010';" 2 1;
- 3 constant **integer** esize 32;
- 4 integer n = UInt(Zn)
- integer m = UInt(' :Zm); 5
- 6 integer offset JInt (of 3); constant inte
- nreq



off3 1 1 0 Zn 0 Rv 0 Zn 0 1 Lu

SDOT ZA.S[<Wv>, <offs>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H

```
if !HaveSME2() then UNDEFINED;
1
```

```
2
  integer v = UInt('010':Rv);
```

```
3
  constant integer esize = 32;
```

```
4
  integer n = UInt(Zn);
```

```
5
  integer m = UInt('0':Zm);
```

```
integer offset = UInt(off3);
6
7
  constant integer nreg = 4;
```

Assembler Symbols

- $\langle Wv \rangle$ Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
- <offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.
- $\langle Zn1 \rangle$ Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn".

off3

0 1 Lu

Zn

0

- <Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" plus 3 modulo 32.
- <Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" plus 1 modulo 32.
- <Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

```
1
    CheckStreamingSVEAndZAEnabled();
    constant integer VL = CurrentVL;
2
 3
    constant integer elements = VL DIV esize;
   integer vectors = VL DIV 8;
4
    integer vstride = vectors DIV nreg;
 5
 6
   bits(32) vbase = X[v, 32];
    integer vec = (UInt(vbase) + offset) MOD vstride;
7
8
    bits(VL) result;
9
10
   for r = 0 to nreg-1
11
        bits(VL) operand1 = Z[(n+r) MOD 32, VL];
        bits(VL) operand2 = Z[m, VL];
12
13
        bits(VL) operand3 = ZAvector[vec, VL];
14
        for e = 0 to elements-1
15
             bits(esize) sum = Elem[operand3, e, esize
16
             for i = 0 to 1
                 integer element1 = SInt(Elem f ere '1,
integer element2 = SInt(Ele operar 2,
17
                                                                          esize DIV 2]);
18
                                                             2
                                                                         esize DIV 2]);
19
                 sum = sum + element1 * e]
                                               .ent2;
        Elem[result, e, esize] = sum;
ZAvector[vec, VL] = result;
20
21
22
        vec = vec + vstride;
```

D1.1.140 SDOT (2-way, multiple vectors)

Multi-vector signed integer dot-product

The instruction operates on two or four ZA single-vector groups.

The signed integer dot product instruction computes the dot product of two signed 16-bit integer values held in each 32-bit element of the two or four first source vectors and two signed 16-bit integer values in the corresponding 32-bit element of the two or four second source vectors. The widened dot product result is destructively added to the corresponding 32-bit element of the two or four ZA single-vector groups. The vector numbers forming the single-vector group within each half or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The VECTOR GROUP symbol is preferred for disasser "t optional in assembler source code.

This instruction is unpredicated.

It has encodings from 2 classes: Two ZA single-vectors and F . ZA sing. vectors

Two ZA single-vectors (FEAT_SME2)

1

2 3

4

5 6

1

2

3



```
5
  integer offset = UInt(off3);
6
```

```
7
  constant integer nreg = 4;
```

Assembler Symbols

- $\langle Wv \rangle$ Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
- <offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.
- $\langle Zn1 \rangle$ For the two ZA single-vectors variant: is the name of the first scalable vector register of a

multi-vector sequence, encoded as "Zn" times 2.

For the four ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

- <Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.
- <Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.
- <Zm1> For the two ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 2.

For the four ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 4.

- <Zm4> Is the name of the fourth scalable vector register of a multi-v_tor sequence encoded as "Zm" times 4 plus 3.

```
CheckStreamingSVEAndZAEnabled();
1
   constant integer VL = CurrentVL;
2
3
   constant integer elements = VL DIV es;
   integer vectors = VL DIV 8;
4
   integer vstride = vectors DIV nreg;
5
6
   bits(32) vbase = X[v, 32];
   integer vec = (UInt(vbase) +
7
                                c fsei
                                       MOD
                                           v.
                                               ide:
   bits(VL) result;
8
9
10
   for r = 0 to nreg-1
       bits(VL) operand1 = 2[n+r, 'L];
11
12
       bits(VL) operand = Z[m+r,
       bits(VL) opera .3 = ZAvector[
13
                                        VL1:
14
       for e = 0 to lements-1
           bits(e ze) sur Elem[operand3, e, esize];
for i     to
15
16
               17
18
19
                          f .ment1 * element2;
                 ı = su
20
           Elem[ esult, e, size] = sum;
21
        Avecto<sup>.</sup>
                    VIJ = result;
22
                +
                   vs....de;
```

D1.1.141 SDOT (4-way, multiple and indexed vector)

Multi-vector signed integer dot-product by indexed element

The instruction operates on two or four ZA single-vector groups.

The signed integer dot product instruction computes the dot product of four signed 8-bit or 16-bit integer values held in each 32-bit or 64-bit element of the two or four first source vectors and four signed 8-bit or 16-bit integer values in the corresponding indexed 32-bit or 64-bit element of the second source vector. The widened dot product result is destructively added to the corresponding 32-bit or 64-bit element of the two or four ZA single-vector groups.

The groups within the second source vector are specified using an immediate element index which selects the same group position within each 128-bit vector segment. The index range is from 0 to one less than the number of groups per 128-bit segment, encoded in 1 to 2 bits depending on the simple group. The vector numbers forming the single-vector group within each half or each quarter of the A array selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA ray vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the Z operan onsists of t o or four ZA single-vector groups respectively. The VECTOR GROUP symbol is preferre for disassen. v, but ptional in assembler source code.

This instruction is unpredicated.

ID_AA64SMFR0_EL1.I16I64 indicates whether leng it integeneration is implemented.

It has encodings from 4 classes: Two ZA si gle-vectors of 32-bit elements, Two ZA single-vectors of 64-bit elements, Four ZA single-vectors of 32-bit elements are single-vectors of 64-bit elements

Two ZA single-vectors of 32-bit element (FEAT_SME2)

 31
 20
 10
 16
 15
 14
 13
 12
 11
 10
 9
 6
 5
 4
 3
 2
 0

 1
 1
 0
 0
 1
 6
 0
 1
 2
 0
 Rv
 1
 i2
 Zn
 1
 0
 0
 off3

 Lu
 <t

```
1
  if !Have
                then. WDEFI
2
  integr v = U
               it('01
                      R۰
  cons int inte esize
3
                         32:
4
                  ···'0');
  int r n =
  5
6
  integer
           fset = UInt(off3);
  integer 1. \circle x = UInt(i2);
7
```

```
8 constant in ver nreg = 2;
```

Two ZA single-vectors of 64-bit elements (FEAT_SME_I16I64)



```
SDOT ZA.D[<Wv>, <offs>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H[<index>]
```

```
1 if !(HaveSME2() && HaveSMEI16164()) then UNDEFINED;
2 integer v = UInt('010':Rv);
3 constant integer esize = 64;
4 integer n = UInt(Zn:'0');
5 integer m = UInt('0':Zm);
```

Chapter D1. SME instructions D1.1. SME and SME2 data-processing instructions

- 6 integer offset = UInt(off3);
- 7 integer index = UInt(i1); 8 constant integer nreg = 2;
 - Four ZA single-vectors of 32-bit elements (FEAT_SME2)



elements variant: is the element index, in the range 0 to 3, encoded in the "i2" field.

For the four ZA single-vectors of 64-bit elements and two ZA single-vectors of 64-bit elements variant: is the element index, in the range 0 to 1, encoded in the "i1" field.

```
CheckStreamingSVEAndZAEnabled();
1
   constant integer VL = CurrentVL;
2
   constant integer elements = VL DIV esize;
3
4
   integer vectors = VL DIV 8;
   integer vstride = vectors DIV nreq;
5
6
   integer eltspersegment = 128 DIV esize;
7
   bits(32) vbase = X[v, 32];
   integer vec = (UInt(vbase) + offset) MOD vstride;
8
9
   bits(VL) result;
10
   for r = 0 to nreg-1
11
12
        bits(VL) operand1 = Z[n+r, VL];
13
        bits(VL) operand2 = Z[m, VL];
        bits(VL) operand3 = ZAvector[vec, VL];
14
15
        for e = 0 to elements-1
            bits(esize) sum = Elem[operand3, e, esize
16
17
            integer segmentbase = e - (e MOD eltspe
                                                       egment)
18
            integer s = segmentbase + index;
19
            for i = 0 to 3
                                                     11,
20
                integer element1 = SInt(Elem[
                                                                     esize DIV 4]);
21
                integer element2 = SInt(Ele<sup>-</sup> operat
                                                     2.
                                                        4
                                                                    esize DIV 4]);
22
                sum = sum + element1 * el ent2;
23
            Elem[result, e, esize] = sum;
        ZAvector[vec, VL] = result;
24
25
        vec = vec + vstride;
```

D1.1.142 SDOT (4-way, multiple and single vector)

Multi-vector signed integer dot-product by vector

The instruction operates on two or four ZA single-vector groups.

The signed integer dot product instruction computes the dot product of four signed 8-bit or 16-bit integer values held in each 32-bit or 64-bit element of the two or four first source vectors and four signed 8-bit or 16-bit integer values in the corresponding 32-bit or 64-bit element of the second source vector. The widened dot product result is destructively added to the corresponding 32-bit or 64-bit element of the two or four ZA single-vector groups. The vector numbers forming the single-vector group within each half or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The VECTOR GROUP symbol is preferred for disasser to perform on the source code.

This instruction is unpredicated.

ID_AA64SMFR0_EL1.I16I64 indicates whether the 16-bit int _er varian. implem_ted.

It has encodings from 2 classes: Two ZA single-vectors a Four ZA ingle-vectors

Two ZA single-vectors (FEAT_SME2)

13 10 0 0 0 0 1 0 sz 1 0 7m 0 BV 1 • 0 1 7n 0 0 off3 Lu

SDOT ZA.<T>[<Wv>, <of >{, VGx2}], <<Tb>-<Zn2>.<Tb> }, <Zm>.<Tb>

```
1
   if !HaveSME2() then UNL FINED,
   if sz == '1' && !Hay CMEI16I64()
                                          hen UNDEFINED:
2
3
   integer v = UInt(' .0':Rv);
   constant integer .size = 32 << UInt(.sz);
4
5
   integer n = U<sup>T</sup> .(Zn);
6
   integer m = UIA ''0'
                             .);
   integer offset =
                           .(off3);
7
8
                   •er n.
                             = 2;
   constan<sup>+</sup>
```

For **ZA sing'** vectors (FEA SM^{*} Z)



```
SDOT ZA.<T>[<Wv>, <offs>{, VGx4}], { <Zn1>.<Tb>-<Zn4>.<Tb> }, <Zm>.<Tb>
```

```
1 if !HaveSME2() then UNDEFINED;
```

```
2 if sz == '1' && !HaveSMEI16I64() then UNDEFINED;
```

```
3 integer v = UInt('010':Rv);
```

```
4 constant integer esize = 32 << UInt(sz);</pre>
```

```
5 integer n = UInt(Zn);
```

```
6 integer m = UInt('0':Zm);
7 integer offset = UInt(off
```

```
7 integer offset = UInt(off3);
8 constant integer nreg = 4;
```

Assembler Symbols

<T> Is the size specifier, encoded in "sz":

SZ	<t></t>
0	S
1	D

- <Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
- <offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.
- <Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn".
- <Tb> Is the size specifier, encoded in "sz":

SZ	< Tb >
0	В
1	Н

- <Zn4> Is the name of the fourth scalable vector register of a multi-vec sequence, enc led as "Zn" plus 3 modulo 32.
- <Zn2> Is the name of the second scalable vector register of multi-vector quer 2, encoded as "Zn" plus 1 modulo 32.
- <Zm> Is the name of the second source scalable vector reg. r⁻_-Z15, enc ded in the "Zm" field.

```
CheckStreamingSVEAndZAEnabled();
1
   constant integer VL = CurrentVL;
2
3
   constant integer elements = VL D. esize,
4
   integer vectors = VL DIV 8;
   integer vstride = vectors DV nreg;
5
6
   bits(32) vbase = X[v, 32];
   integer vec = (UInt(vbas
7
                                  ffset) MOD tride;
8
   bits(VL) result;
0
   for r = 0 to nreq-
10
       bits(VL) ope .nd1 = Z[(n+r) MOD 2, VL];
bits(VL) or rand2 = [m, VL];
bits(VL) or rand3 ZAvector[vec, VL];
11
12
13
       14
15
16
           for = 0 to
17
                i :eger el .nt1 = SInt(Elem[operand1, 4 * e + i, esize DIV 4]);
               sum = ____ + element1 * element2;
18
19
20
             em[result, e, esize] = sum;
21
       ZAvec r[vec, VL] = result;
22
       vec =
             v + vstride;
```

D1.1.143 SDOT (4-way, multiple vectors)

Multi-vector signed integer dot-product

The instruction operates on two or four ZA single-vector groups.

The signed integer dot product instruction computes the dot product of four signed 8-bit or 16-bit integer values held in each 32-bit or 64-bit element of the two or four first source vectors and four signed 8-bit or 16-bit integer values in the corresponding 32-bit or 64-bit element of the two or four second source vectors. The widened dot product result is destructively added to the corresponding 32-bit or 64-bit element of the two or four group within each half or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand control of two or four ZA single-vector groups respectively. The VECTOR GROUP symbol is preferred for disact mbly, but period in assembler source code.

This instruction is unpredicated.

ID_AA64SMFR0_EL1.I16I64 indicates whether the 16-bit ...eger variant is ______. ...ented.

It has encodings from 2 classes: Two ZA single-vectors a. Four ' A single-vectors

Two ZA single-vectors (FEAT_SME2)



```
4 constant integer esize = 32 << UInt(sz);</pre>
```

```
5 integer n = UInt(Zn:'00');
```

```
6 integer m = UInt(Zm:'00');
```

```
7 integer offset = UInt(off3);
```

```
8 constant integer nreg = 4;
```

Assembler Symbols

<T> Is the size specifier, encoded in "sz":

SZ	< T >
0	S
1	D

- <Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
- <offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.
- <Zn1> For the two ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.

For the four ZA single-vectors variant: is the name of the first scale of a multi-vector sequence, encoded as "Zn" times 4.

<Tb> Is the size specifier, encoded in "sz":

SZ	< Tb >
0	В
1	Н

- <Zn4> Is the name of the fourth scalable vector region for a multiple vector quence, encoded as "Zn" times 4 plus 3.
- <Zn2> Is the name of the second scalable ve or regimetrivector sequence, encoded as "Zn" times 2 plus 1.
- <Zm1> For the two ZA single-vect's variation is the time of the first scalable vector register of a multi-vector sequence, e coded as "Zn_times 2.

For the four ZA sin -vec. s variant: is the name of the first scalable vector register of a multi-vector sequence, encode is "Zm" times 4.

- <Zm4> Is the name the fourth scalable actor register of a multi-vector sequence, encoded as "Zm" tim 4 plus 3
- $\langle Zm2 \rangle$ Is the name γ second able vector register of a multi-vector sequence, encoded as "7 les 2 μ 1.

Op ation

```
.ingSVEAnoZAEnabled();
 1
   Check
   constan. nteger VL = CurrentVL;
constant . eger elements = VL DIV esize;
 2
 3
 4
   integer vect s = VL DIV 8;
   integer vstride = vectors DIV nreg;
 5
   bits(32) vbase = X[v, 32];
 6
   integer vec = (UInt(vbase) + offset) MOD vstride;
 7
8
   bits(VL) result;
 0
10
   for r = 0 to nreg-1
11
        bits(VL) operand1 = Z[n+r, VL];
12
        bits(VL) operand2 = Z[m+r, VL];
        bits(VL) operand3 = ZAvector[vec, VL];
13
14
        for e = 0 to elements-1
15
            bits(esize) sum = Elem[operand3, e, esize];
16
            for i = 0 to 3
17
                 integer element1 = SInt(Elem[operand1, 4 * e + i, esize DIV 4]);
18
                 integer element2 = SInt(Elem[operand2, 4 * e + i, esize DIV 4]);
19
                 sum = sum + element1 * element2;
20
            Elem[result, e, esize] = sum;
21
        ZAvector[vec, VL] = result;
22
        vec = vec + vstride;
```

D1.1.144 SEL

Multi-vector conditionally select elements from two vectors

Read active elements from the two or four first source vectors and inactive elements from the two or four second source vectors and place in the corresponding elements of the two or four destination vectors.

It has encodings from 2 classes: Two registers and Four registers

Two registers (FEAT_SME2)



Assembler S____bols

<Zd1> For the two registers variant: is the name of the first destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 2.

For the four registers variant: is the name of the first destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 4.

<T> Is the size specifier, encoded in "size":

<t></t>
В
Н
S
D

<Zd4> Is the name of the fourth destination scalable vector register of a multi-vector sequence,

encoded as "Zd" times 4 plus 3.

- <Zd2> Is the name of the second destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 2 plus 1.
- <PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.
- <Zn1> For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.

For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

- <Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.
- <Zn2> Is the name of the second scalable vector register of a multi-v for sequence encoded as "Zn" times 2 plus 1.
- <Zm1> For the two registers variant: is the name of the first scale is vector gister of relativector sequence, encoded as "Zm" times 2.

For the four registers variant: is the name of the fine calable ector register of a multi-vector sequence, encoded as "Zm" times 4.

- <Zm4> Is the name of the fourth scalable vector legist of a n. 'ti-v lor sequence, encoded as "Zm" times 4 plus 3.
- <Zm2> Is the name of the second scalable vec rester or a ulti-vector sequence, encoded as "Zm" times 2 plus 1.

```
1
   CheckStreamingSVEEnable /;
    constant integer VL = CarrentV
2
   constant integer PI VL DIV 8;
3
   constant integer ements = VL DIV
array [0..3] of ts(VL) results;
bits(PL) pred P[g, P<sup>*</sup>,
4
                                              ize;
5
6
7
   bits(PL * nreg)
                        rsk CounterToPredicate(pred<15:0>, PL * nreg);
8
9
   for r =
                   reg-l
10
        b' s(VL)
                   erandì
                                 .n+r, VL];
11
          _ts(VL) perand2 = 2[m+r, VL];
12
             е
                            onts-1
13
               ActivePredicateElement(mask, r * elements + e, esize) then
14
                 Elem[results[r], e, esize] = Elem[operand1, e, esize];
15
             e.
16
                  em[results[r], e, esize] = Elem[operand2, e, esize];
17
18
   for r = 0 to nreg-1
        Z[d+r, VL] = results[r];
19
```

D1.1.145 SMAX (multiple and single vector)

Multi-vector signed maximum by vector

Determine the signed maximum of elements of the second source vector and the corresponding elements of the two or four first source vectors and destructively place the results in the corresponding elements of the two or four first source vectors.

This instruction is unpredicated.

It has encodings from 2 classes: Two registers and Four registers

Two registers (FEAT_SME2)



Assembler Sy bols

<Zdn1> For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2.

For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4.

<T> Is the size specifier, encoded in "size":

size	<t></t>
00	В
01	Η
10	S
11	D

<Zdn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as

"Zdn" times 4 plus 3.

- <Zdn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2 plus 1.
 - <Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

```
1
   CheckStreamingSVEEnabled();
   constant integer VL = CurrentVL;
2
3
   constant integer elements = VL DIV esize;
4
   array [0..3] of bits(VL) results;
5
6
   for r = 0 to nreg-1
        bits(VL) operand1 = Z[dn+r, VL];
bits(VL) operand2 = Z[m, VL];
7
8
9
        for e = 0 to elements-1
            integer element1 = Int(Elem[operand1, e, esize],
10
                                                                    signed);
            integer element2 = Int(Elem[operand2, e, esize],
11
                                                                  ù
                                                                     'gned);
12
            integer res = Max(element1, element2);
13
            Elem[results[r], e, esize] = res<esize-1:0>
14
   for r = 0 to nreg-1
15
16
    Z[dn+r, VL] = results[r];
```

D1.1.146 SMAX (multiple vectors)

Multi-vector signed maximum

Determine the signed maximum of elements of the two or four second source vectors and the corresponding elements of the two or four first source vectors and destructively place the results in the corresponding elements of the two or four first source vectors.

This instruction is unpredicated.

It has encodings from 2 classes: Two registers and Four registers

Two registers (FEAT_SME2)



Assembler Symbols

<Zdn1> For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2.

For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4.

<T> Is the size specifier, encoded in "size":

size	<t></t>
00	В
01	Н
10	S
11	D

- <Zdn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4 plus 3.
- <Zdn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2 plus 1.
- <Zm1> For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 2.

For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 4.

- <Zm4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zm" times 4 plus 3.
- <Zm2> Is the name of the second scalable vector register of a multi-vector ence, encoded as "Zm" times 2 plus 1.

```
1
   CheckStreamingSVEEnabled();
   constant integer VL = CurrentVL;
2
3
   constant integer elements = VL DIV esize;
4
   array [0..3] of bits(VL) results;
5
6
   for r = 0 to nreg-1
       bits(VL) operand1 = Z[dn+r, VL];
7
       bits(VL) operand2 = Z[m+r, VL];
8
9
       for e = 0 to elements-1
10
                                              nd
                                                      es___, unsigned);
            integer element1 = Int(Elem[ope
                                                   e,
11
            integer element2 = Int(E1 [opera
                                                 2, e, esize], unsigned);
            integer res = Max(eler nt1,
12
                                          lemen
13
            Elem[results[r], e,
                                  .ize] =
                                            s<esiz
                                                    1:0>;
14
15
   for r = 0 to nreg-1
16
       Z[dn+r, VL] = resul .s[r];
```

D1.1.147 SMIN (multiple and single vector)

Multi-vector signed minimum by vector

Determine the signed minimum of elements of the second source vector and the corresponding elements of the two or four first source vectors and destructively place the results in the corresponding elements of the two or four first source vectors.

This instruction is unpredicated.

It has encodings from 2 classes: Two registers and Four registers

Two registers (FEAT_SME2)



Assembler Sy bols

<Zdn1> For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2.

For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4.

<T> Is the size specifier, encoded in "size":

size	<t></t>
00	В
01	Н
10	S
11	D

<Zdn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as

"Zdn" times 4 plus 3.

- <Zdn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2 plus 1.
 - <Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

```
1
   CheckStreamingSVEEnabled();
   constant integer VL = CurrentVL;
2
3
   constant integer elements = VL DIV esize;
4
   array [0..3] of bits(VL) results;
5
6
   for r = 0 to nreg-1
        bits(VL) operand1 = Z[dn+r, VL];
bits(VL) operand2 = Z[m, VL];
7
8
9
        for e = 0 to elements-1
            integer element1 = Int(Elem[operand1, e, esize],
10
                                                                    signed);
            integer element2 = Int(Elem[operand2, e, esize],
11
                                                                  ù
                                                                     'gned);
12
            integer res = Min(element1, element2);
13
            Elem[results[r], e, esize] = res<esize-1:0>
14
   for r = 0 to nreg-1
15
16
    Z[dn+r, VL] = results[r];
```

D1.1.148 SMIN (multiple vectors)

Multi-vector signed minimum

Determine the signed minimum of elements of the two or four second source vectors and the corresponding elements of the two or four first source vectors and destructively place the results in the corresponding elements of the two or four first source vectors.

This instruction is unpredicated.

It has encodings from 2 classes: Two registers and Four registers

Two registers (FEAT_SME2)



Assembler Symbols

<Zdn1> For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2.

For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4.

<T> Is the size specifier, encoded in "size":

size	<t></t>
00	В
01	Η
10	S
11	D

- <Zdn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4 plus 3.
- <Zdn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2 plus 1.
- <Zm1> For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 2.

For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 4.

- <Zm4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zm" times 4 plus 3.
- <Zm2> Is the name of the second scalable vector register of a multi-vector ence, encoded as "Zm" times 2 plus 1.

```
1
   CheckStreamingSVEEnabled();
   constant integer VL = CurrentVL;
2
3
   constant integer elements = VL DIV esize;
4
   array [0..3] of bits(VL) results;
5
6
   for r = 0 to nreg-1
       bits(VL) operand1 = Z[dn+r, VL];
7
       bits(VL) operand2 = Z[m+r, VL];
8
9
       for e = 0 to elements-1
10
                                              nd
                                                      es___, unsigned);
            integer element1 = Int(Elem[ope
                                                   e,
11
            integer element2 = Int(E1 [opera
                                                 2, e, esize], unsigned);
12
            integer res = Min(eler nt1,
                                          lemen
13
            Elem[results[r], e,
                                  .ize] =
                                            s<esiz
                                                    1:0>;
14
15
   for r = 0 to nreg-1
16
       Z[dn+r, VL] = resul .s[r];
```

D1.1.149 SMLAL (multiple and indexed vector)

Multi-vector signed integer multiply-add long by indexed element

The instruction operates on one, two, or four ZA double-vector groups.

This signed integer multiply-add long instruction multiplies each signed 16-bit element in the one, two, or four first source vectors with each signed 16-bit indexed element of the second source vector, widens each product to 32-bits and destructively adds these values to the corresponding 32-bit elements of the one, two, or four ZA double-vector groups.

The elements within the second source vector are specified using an immediate element index which selects the same element position within each 128-bit vector segment. The index range is from 0 to 7, encoded in 3 bits. The lowest of the two consecutive vector numbers forming the double-vector group within all, each half, or each quarter of the ZA array are selected by the sum of the vector select register mediate offset, modulo all, half, or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the \land operand consists of two or four ZA double-vector groups respectively. The VECTOR GROUP symbol is $_{\rm F}$ ferred for isassembly, but optional in assembler source code.

This instruction is unpredicated.

It has encodings from 3 classes: One ZA double-vector, Two double-vectors and Four ZA double-vectors

One ZA double-vector (FEAT_SME2)



```
if !HaveSME2() t<sup>1</sup> . UNDEFINED;
1
2
  constant integr esize = 32;
  integer v = U1 ('010' .v);
3
4
  integer n = UInt
                      יר '
                        •Zm):
5
  integer m
                VInt (
6
  intege offs.
                   = UIn. off
                                 (0')
   inter _ index = UInt(1
7
                              131);
  cor ant int /er nreg = 1;
8
```

```
Two ZA uble-vectors
(FEAT_SN.?)
```



SMLAL ZA.S[<Wv>, <offsf>:<offsl>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H[<index>]

```
1 if !HaveSME2() then UNDEFINED;
```

```
2 constant integer esize = 32;
```

```
3 integer v = UInt('010':Rv);
```

```
4 integer n = UInt(Zn:'0');
```

```
5 integer m = UInt('0':Zm);
```

```
6 integer offset = UInt(off2:'0');
7 integer index = UInt(i3h:i3l);
```

```
8 constant integer nreg = 2;
```

Four ZA double-vectors (FEAT_SME2) 0 0 0 1 0 0 Zm Rv Zn 1 1 0 1 i3h 1 0 SMLAL ZA.S[<Wv>, <offsf>:<offsl>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H[<index>] 1 if !HaveSME2() then UNDEFINED; 2 constant integer esize = 32; integer v = UInt('010':Rv); 3 4 integer n = UInt(Zn:'00'); integer m = UInt('0':Zm); 5 integer offset = UInt(off2:'0'); 6 integer index = UInt(i3h:i3l); constant **integer** nreg = 4; **Assembler Symbols** / field. $\langle Wv \rangle$ Is the 32-bit name of the vector select register W8.⁷ 11, encc d in the For the one ZA double-vector variant: is the vector electron offset, r inting to first of two <offsf> consecutive vectors, encoded as "off3" field <u>s</u> 2. For the four ZA double-vectors and two A double vectors At: is the vector select offset, pointing to first of two consecutive v 'ors, enc int as "off2" field times 2. <offsl> For the one ZA double-vector riant: . e vector select offset, pointing to last of two consecutive vectors, encoder'ns "o. "" fiela res 2 plus 1. For the four ZA double- tors and two \double-vectors variant: is the vector select offset, pointing to last of tw _on_ utive vectors needed as "off2" field times 2 plus 1. Is the name of the first source _____able vector register, encoded in the "Zn" field. $\langle Zn \rangle$ $\langle Zn1 \rangle$ For the two double-vectors val. at: is the name of the first scalable vector register of a multi-vec c sequer , encoded as "Zn" times 2. For the four <u>v</u> ouble-vec rs variant: is the name of the first scalable vector register of a tor sec, ince, i loded as "Zn" times 4. m <Zn4>is the nr le of the full in scalable vector register of a multi-vector sequence, encoded as "Zn" times ls of name of the second scalable vector register of a multi-vector sequence, encoded as <Zn2>"Zn nes 2 plus 1. Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field. $\langle Zm \rangle$ <index> Is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields. Operation

```
CheckStreamingSVEAndZAEnabled();
1
2
   constant integer VL = CurrentVL;
   constant integer elements = VL DIV esize;
3
4
   integer vectors = VL DIV 8;
   integer vstride = vectors DIV nreg;
5
   integer eltspersegment = 128 DIV esize;
6
   bits(32) vbase = X[v, 32];
7
8
   integer vec = (UInt(vbase) + offset) MOD vstride;
   bits(VL) result;
9
10
   vec = vec - (vec MOD 2);
11
12 for r = 0 to nreg-1
```
Chapter D1. SME instructions D1.1. SME and SME2 data-processing instructions

```
bits(VL) operand1 = Z[n+r, VL];
13
14
         bits(VL) operand2 = Z[m, VL];
15
          for i = 0 to 1
16
              bits(VL) operand3 = ZAvector[vec + i, VL];
17
               for e = 0 to elements-1
18
                    integer segmentbase = e - (e MOD eltspersegment);
                    integer s = 2 * segmentbase + index;
19
                   integer element1 = SInt(Elem[operand1, 2 * e + i, esize DIV 2]);
integer element2 = SInt(Elem[operand2, s, esize DIV 2]);
bits(esize) product = (element1 * element2)<esize-1:0>;
20
21
22
23
                   Elem[result, e, esize] = Elem[operand3, e, esize] + product;
24
               ZAvector[vec + i, VL] = result;
25
         vec = vec + vstride;
```

D1.1.150 SMLAL (multiple and single vector)

Multi-vector signed integer multiply-add long by vector

The instruction operates on one, two, or four ZA double-vector groups.

This signed integer multiply-add long instruction multiplies each signed 16-bit element in the one, two, or four first source vectors with each signed 16-bit element in the second source vector, widens each product to 32-bits and destructively adds these values to the corresponding 32-bit elements of the one, two, or four ZA double-vector groups. The lowest of the two consecutive vector numbers forming the double-vector group within all, each half, or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo all, half, or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA double-vector groups respectively. The VECTOR GROUP symbol is proved for disassembly, but optional in assembler source code.

This instruction is unpredicated.

It has encodings from 3 classes: One ZA double-vector, Two 7, double- tors and our ZA double-vectors

```
One ZA double-vector (FEAT_SME2)
```



SMLAL ZA.S[<Wv>, <offsf>:< ffsl . <Zn H, <Zm>.H

```
1 if !HaveSME2() then UNDEF
```

- 2 constant integer esize > .2;
- 3 integer v = UInt('010':xv);
- 4 integer n = UInt(Zn)
- 5 integer m = UInt(' :Zm);
- 6 integer offset = JInt(off3:'0');
- constant **inte** nreg

```
Two ZA de-vec.
(FEAT ME2
```



SMLAL ZA.S[<Wv>, <offsf>:<offsl>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H

- 1 if !HaveSME2() then UNDEFINED;
- 2 constant integer esize = 32;
- 3 integer v = UInt('010':Rv);
- 4 integer n = UInt(Zn);
- 5 integer m = UInt('0':Zm);
- 6 integer offset = UInt(off2:'0'); 7 constant integer nreg = 2;

```
Four ZA double-vectors (FEAT_SME2)
```



 $\label{eq:smlal_$

- 1 if !HaveSME2() then UNDEFINED;
- 2 constant integer esize = 32;
- 3 integer v = UInt('010':Rv);
- 4 integer n = UInt(Zn);
- 5 integer m = UInt('0':Zm);
- 6 integer offset = UInt(off2:'0');
 7
- 7 constant **integer** nreg = 4;

Assembler Symbols

- <Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
- <offsf> For the one ZA double-vector variant: is the vector select offset r to first of two consecutive vectors, encoded as "off3" field times 2.

For the four ZA double-vectors and two ZA double-vectors variables is the vector elect offset, pointing to first of two consecutive vectors, encoded as " .f2" field imes 2.

<offsl> For the one ZA double-vector variant: is the vect select offset, put is to last of two consecutive vectors, encoded as "off3" field times plus 1

For the four ZA double-vectors and two ZA double-vector variant: the vector select offset, pointing to last of two consecutive vector encoded as "to 2" for times 2 plus 1.

- <Zn> Is the name of the first source scalab' vector rester, encoded in the "Zn" field.
- $\langle Zn1 \rangle$ Is the name of the first scalable vector region of a multi-vector sequence, encoded as "Zn".
- <Zn4> Is the name of the fourth sc?' ble ver regis of a multi-vector sequence, encoded as "Zn" plus 3 modulo 32.
- <Zn2> Is the name of the sound plable vector gister of a multi-vector sequence, encoded as "Zn" plus 1 modplo 32.
- <Zm> Is the name c .ne second source s 'able vector register Z0-Z15, encoded in the "Zm" field.

```
CheckStre qSVEA 7AEnabl ();
constar int, rVL Cur .tVL;
const .t inte :r eleme = VL DIV esize;
 1
2
 3
    int er vect s = VL DIV 8;
inte. VS rate Cors D
 4
                           .ors DIV nreg;
 5
    bits(3. pase = X[v, 32];
integer = (UInt(vbase) + offset) MOD vstride;
6
 7
    bits(VL) r lt;
vec = vec - vec MOD 2);
 8
9
10
11
    for r = 0 to nreg-1
         bits(VL) operand1 = Z[(n+r) MOD 32, VL];
12
13
         bits(VL) operand2 = Z[m, VL];
14
         for i = 0 to 1
15
             bits(VL) operand3 = ZAvector[vec + i, VL];
16
              for e = 0 to elements-1
17
                  integer element1 = SInt(Elem[operand1, 2 * e + i, esize DIV 2]);
18
                  integer element2 = SInt(Elem[operand2, 2 * e + i, esize DIV 2]);
19
                  bits(esize) product = (element1 * element2)<esize-1:0>;
20
                  Elem[result, e, esize] = Elem[operand3, e, esize] + product;
21
             ZAvector[vec + i, VL] = result;
22
      vec = vec + vstride;
```

Chapter D1. SME instructions D1.1. SME and SME2 data-processing instructions

D1.1.151 SMLAL (multiple vectors)

Multi-vector signed integer multiply-add long

The instruction operates on two or four ZA double-vector groups.

This signed integer multiply-add long instruction multiplies each signed 16-bit element in the two or four first source vectors with each signed 16-bit element in the two or four second source vectors, widens each product to 32-bits and destructively adds these values to the corresponding 32-bit elements of the two or four ZA double-vector groups. The lowest of the two consecutive vector numbers forming the double-vector group within each half or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA double-vector groups respectively. The VECTOR GROUP symbol is profor disassembly, but optional in assembler source code.

This instruction is unpredicated.

It has encodings from 2 classes: Two ZA double-vectors and F Tr ZA dou -vector

Two ZA double-vectors (FEAT_SME2)

1

1

2



```
5
  integer m = UInt(Zm:'00');
```

```
integer offset = UInt(off2:'0');
6
```

```
7
  constant integer nreg = 4;
```

Assembler Symbols

Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field. $\langle Wv \rangle$

Is the vector select offset, pointing to first of two consecutive vectors, encoded as "off2" field <offsf> times 2.

Chapter D1. SME instructions

D1.1. SME and SME2 data-processing instructions

- <offsl> Is the vector select offset, pointing to last of two consecutive vectors, encoded as "off2" field times 2 plus 1.
- <Zn1> For the two ZA double-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.

For the four ZA double-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

- <Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.
- <Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.
- <Zm1> For the two ZA double-vectors variant: is the name of the first scale¹ vector register of a multi-vector sequence, encoded as "Zm" times 2.

For the four ZA double-vectors variant: is the name of the fin. calable vecto. egister of a multi-vector sequence, encoded as "Zm" times 4.

- <Zm4> Is the name of the fourth scalable vector register of multi-vector rue c, encoded as "Zm" times 4 plus 3.
- <Zm2> Is the name of the second scalable vector register o. _____iti-vecto__equence, encoded as "Zm" times 2 plus 1.

```
1
   CheckStreamingSVEAndZAEnabled();
2
   constant integer VL = CurrentVL;
   constant integer elements = V DIV
3
                                            ze;
4
   integer vectors = VL DIV 8;
   integer vstride = vectors
5
                                  1 nreq;
6
   bits(32) vbase = X[v, 3',
   integer vec = (UInt(vbase) + o. et) MOD vstride;
7
   bits(VL) result;
8
9
                       5 2);
   vec = vec - (vec)
10
   for r = 0 to i g-1
11
       bits(VL) op and = Z[n+r. VL];
bits(VT) oper Z = Z[m+ VL];
12
13
        bits(VT ) oper
14
        for
                  to 1
            bits( ) oper
                            A = ZAvector[vec + i, VL];
15
16
            for r = 0 to el ments-1
17
                          lement1 = SInt(Elem[operand1, 2 * e + i, esize DIV 2]);
                integer element2 = SInt(Elem[operand2, 2 * e + i, esize DIV 2]);
18
19
                bits(esize) product = (element1 * element2)<esize-1:0>;
20
                Elem[result, e, esize] = Elem[operand3, e, esize] + product;
21
            ZAve or[vec + i, VL] = result;
22
        vec = vec + vstride;
```

D1.1.152 SMLALL (multiple and indexed vector)

Multi-vector signed integer multiply-add long long by indexed element

The instruction operates on one, two, or four ZA quad-vector groups.

This signed integer multiply-add long long instruction multiplies each signed 8-bit or 16-bit element in the one, two, or four first source vectors with each signed 8-bit or 16-bit indexed element of second source vector, widens each product to 32-bits or 64-bits and destructively adds these values to the corresponding 32-bit or 64-bit elements of the one, two, or four ZA quad-vector groups.

The elements within the second source vector are specified using an immediate element index which selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 3 to 4 bits depending on the size of the element. The lowest of the four consecutive vector numbers forming the quad-vector group within all half, or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, not all half, or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the 7 operation consists of voor four ZA quad-vector groups respectively. The VECTOR GROUP symbol is preferre for disassen. v, but ptional in assembler source code.

This instruction is unpredicated.

ID_AA64SMFR0_EL1.I16I64 indicates whether leng it integination is implemented.

It has encodings from 6 classes: One ZA quad-vector of 2-bit elements, One ZA quad-vector of 64-bit elements, Two ZA quad-vectors of 32-bit elements, Tv ZA quad-vectors of 64-bit elements, Four ZA quad-vectors of 64-bit elements

One ZA quad-vector of 32-bit e' nents (FEAT_SME2)



SMLALL ZA.S[<Wv, offsf>: ffsl>], <Zn>.B, <Zm>.B[<index>]

```
if !H eSME2 ( then
1
                             NED.
2
  con
       ant int er esize
                            32:
3
  inte
        v 7
                        :Rv);
4
               UInt(Zn);
  intege
             = UInt('0':Zm);
5
  integer
  integer on et = UInt(off2:'00');
6
  integer inde = UInt(i4h:i4l);
7
8
  constant integer nreg = 1;
```

One ZA quad-vector of 64-bit elements (FEAT_SME_I16I64)



SMLALL ZA.D[<Wv>, <offsf>:<offsl>], <Zn>.H, <Zm>.H[<index>]

if !(HaveSME2() && HaveSMEI16I64()) then UNDEFINED;

```
2 constant integer esize = 64;
```

```
3 integer v = UInt('010':Rv);
```

```
4 integer n = UInt(Zn);
```

1

Chapter D1. SME instructions D1.1. SME and SME2 data-processing instructions

- integer m = UInt('0':Zm); 5
- 6 integer offset = UInt(off2:'00');
- integer index = UInt(i3h:i3l); 7
- 8 constant integer nreg = 1;

Two ZA quad-vectors of 32-bit elements (FEAT_SME2)



SMLALL ZA.S[<Wv>, <offsf>:<offsl>{, VGx2}], { <Zn1>.B-<Zn2>.B }, <Zm>.B[<index>]

- if !HaveSME2() then UNDEFINED; 1
- 2 constant integer esize = 32; 3
- integer v = UInt('010':Rv); 4
- integer n = UInt(Zn:'0'); 5
- integer m = UInt('0':Zm); 6
- integer offset = UInt(o1:'00'); integer index = UInt(i4h:i4l); 7
- 8 constant integer nreg = 2;

Two ZA quad-vectors of 64-bit elements

(FEAT SME I16I64)

```
0 0
          0
1
              0
                                            Zm
                                                                    0 i3h
                                                                                Zn
                  0
                            0
                                                      0
                                                                                                     i3l
                                                                                                           o1
                         1
                                0
                                    1
                                                                0
                                                                                             0
                                                                                                0
```

SMLALL ZA.D[<Wv>, <off >> offsl>{, Vc }], { <Zn1>.H-<Zn2>.H }, <Zm>.H[<index>]

1

- constant integer < _ze = 64; integer v = UIn⁺ 010':R^x); integer n = U⁺ (Zn:'0' 2
- 3
- 4
- 5
- 6 7
- integer m = UIh. '0' ...);
 integer offset = _ (o1:'00'
 integer = UIh. 'i3h:i',
 constat inte ;r nreg 8

Fou 'A qv f 32-bit elements (FEA1_ .£2) 0 0 0 0 1 1 0 0 1 0 0 1 Zm 1 Rv 0 i4h 0 0 0 i4l П_

SMLALL ZA.S[<Wv>, <offsf>:<offsl>{, VGx4}], { <Zn1>.B-<Zn4>.B }, <Zm>.B[<index>]

- 1 if !HaveSME2() then UNDEFINED;
- 2 constant integer esize = 32;
- 3 integer v = UInt('010':Rv);
- 4 integer n = UInt(Zn:'00');
- 5 integer m = UInt('0':Zm);
- 6 integer offset = UInt(o1:'00'); 7
- integer index = UInt(i4h:i4l); 8

constant integer nreg = 4;

Four ZA quad-vectors of 64-bit elements (FEAT_SME_I16I64)

01



SMLALL ZA.D[<Wv>, <offsf>:<offsl>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H[<index>]

- if !(HaveSME2() && HaveSMEI16I64()) then UNDEFINED;
- 2 constant integer esize = 64;
- 3 integer v = UInt('010':Rv); 4
- integer n = UInt(Zn:'00');
- 5 integer m = UInt('0':Zm);
- 6 integer offset = UInt(o1:'00'); integer index = UInt(i3h:i3l);
- 7 8 constant integer nreg = 4;

Assembler Symbols

<Wv> Is the 32-bit name of the vector select register W8-W11, encode in the "Rv" t ld.

For the one ZA quad-vector of 32-bit elements and or $\angle A$ quad-vector of f bit elements <offsf> variant: is the vector select offset, pointing to first four corecutive .ors, encoded as "off2" field times 4.

> four. quad-y cors of 64-bit elements, For the four ZA quad-vectors of 32-bit eler two ZA quad-vectors of 32-bit elements a two. A quad- vector of 64-bit elements variant: is the vector select offset, pointing to st of four consecutive vectors, encoded as "o1" field times 4.

For the one ZA quad-vector of 32 it elen. 's and one ZA quad-vector of 64-bit elements <offsl> variant: is the vector selec' ifset, porting to st of four consecutive vectors, encoded as "off2" field times 4 plus

> For the four ZA qu. -vecto. of 32-bit elements, four ZA quad-vectors of 64-bit elements, two ZA quad-vectors of 32-bit ments and two ZA quad-vectors of 64-bit elements variant: is the vector ect offset, pointing last of four consecutive vectors, encoded as "o1" field times 4 ply 5.

- Is the name ^cth *inst source scalable vector register, encoded in the "Zn" field.* $\langle Zn \rangle$
- <Zn1> F inc 'o ZA ad-ve is of 32-bit elements and two ZA quad-vectors of 64-bit elements √ariant: the name the first scalable vector register of a multi-vector sequence, encoded

he four ZA quad-vectors of 32-bit elements and four ZA quad-vectors of 64-bit elements var. t: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn imes 4.

- <Zn4>Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.
- <Zn2>Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.
- <Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.
- <index> For the four ZA quad-vectors of 32-bit elements, one ZA quad-vector of 32-bit elements and two ZA quad-vectors of 32-bit elements variant: is the element index, in the range 0 to 15, encoded in the "i4h:i4l" fields.

For the four ZA quad-vectors of 64-bit elements, one ZA quad-vector of 64-bit elements and two ZA quad-vectors of 64-bit elements variant: is the element index, in the range 0 to 7, encoded in the "i3h:i31" fields.

```
CheckStreamingSVEAndZAEnabled();
 1
    constant integer VL = CurrentVL;
 2
3
   constant integer elements = VL DIV esize;
4 integer vectors = VL DIV 8;
    integer vstride = vectors DIV nreg;
5
   integer eltspersegment = 128 DIV esize;
6
7
    bits(32) vbase = X[v, 32];
    integer vec = (UInt(vbase) + offset) MOD vstride;
8
9
    bits(VL) result;
   vec = vec - (vec MOD 4);
10
11
12
    for r = 0 to nreg-1
        bits(VL) operand1 = Z[n+r, VL];
13
14
        bits(VL) operand2 = Z[m, VL];
15
         for i = 0 to 3
             bits(VL) operand3 = ZAvector[vec + i, VL];
16
17
             for e = 0 to elements-1
18
                  integer segmentbase = e - (e MOD eltspersegme. `;
19
                  integer s = 4 * segmentbase + index;
                 integer element1 = SInt(Elem[operand1, 4 e + i, size i
integer element2 = SInt(Elem[operand2, , esize DIV ');
bits(esize) product = (element1 * el ent2)<e ze-1:0</pre>
20
                                                                              rize DIV
                                                                                         ·]);
21
22
23
                  Elem[result, e, esize] = Elem[operal <sup>2</sup>, e,
                                                                    _ze] + product;
24
             ZAvector[vec + i, VL] = result;
25
        vec = vec + vstride;
```

D1.1.153 SMLALL (multiple and single vector)

Multi-vector signed integer multiply-add long long by vector

The instruction operates on one, two, or four ZA quad-vector groups.

This signed integer multiply-add long long instruction multiplies each signed 8-bit or 16-bit element in the one, two, or four first source vectors with each signed 8-bit or 16-bit element in the second source vector, widens each product to 32-bits or 64-bits and destructively adds these values to the corresponding 32-bit or 64-bit elements of the one, two, or four ZA quad-vector groups. The lowest of the four consecutive vector numbers forming the quad-vector group within all, each half, or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo all, half, or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA quad-vector groups respectively. The VECTOR GROUP symbol is preferred for disasser to prove optional in assembler source code.

This instruction is unpredicated.

ID_AA64SMFR0_EL1.I16I64 indicates whether the 16-bit int _er varian. implem_ted.

It has encodings from 3 classes: One ZA quad-vector, Tv ZA quad ectors a our ZA quad-vectors

One ZA quad-vector (FEAT_SME2)



SMLALL ZA.<T>[<Wv>, <of t>:<offsl> <Zn>.<Tb>, <Zm>.<Tb>

```
if !HaveSME2() then UNL_FINED;
1
                                           hen UNDEFINED;
2
   if sz == '1' && !Hay CMEI16I64()
3
   constant integer < _ze = 32 << UIn sz);</pre>
                       010':R');
4
   integer v = UIn+
5
   integer n = U<sup>T</sup> .(Zn);
   integer m = UIn
integer offset =
6
                        '0'
                              .0 ;
7
                           .(off2:
8
   constan+
                    er ni
                             = 1;
```

Tw 'A quad ectors (FEA SM .2)



SMLALL ZA.<T>[<Wv>, <offsf>:<offsl>{, VGx2}], { <Zn1>.<Tb>-<Zn2>.<Tb> }, <Zm>.<Tb>

```
1
  if !HaveSME2() then UNDEFINED;
  if sz == '1' && !HaveSMEI16I64() then UNDEFINED;
2
3
  constant integer esize = 32 << UInt(sz);</pre>
  integer v = UInt('010':Rv);
4
5
  integer n = UInt(Zn);
  integer m = UInt('0':Zm);
6
  integer offset = UInt(o1:'00');
7
  constant integer nreg = 2;
8
```

Four ZA quad-vectors (FEAT_SME2)



SMLALL ZA.<T>[<Wv>, <offsf>:<offsl>{, VGx4}], { <Zn1>.<Tb>-<Zn4>.<Tb> }, <Zm>.<Tb>

```
1 if !HaveSME2() then UNDEFINED;
```

- 2 if sz == '1' && !HaveSMEI16I64() then UNDEFINED;
- 3 constant integer esize = 32 << UInt(sz);</pre>
- 4 integer v = UInt('010':Rv);
- 5 integer n = UInt(Zn);
- 6 integer m = UInt('0':Zm);
- 7 integer offset = UInt(o1:'00');
- 8 constant **integer** nreg = 4;

Assembler Symbols

<T> Is the size specifier, encoded in "sz":

SZ	< T >
0	S
1	D

- <Wv> Is the 32-bit name of the vector select regimer v, W11, ordgiment with the "Rv" field.
- $\langle offsf \rangle$ For the one ZA quad-vector variant is the vector select outset, pointing to first of four consecutive vectors, encoded as "off2 vectors 4.

For the four ZA quad-vectors and vo ZA ad-vectors variant: is the vector select offset, pointing to first of four conjuctive v tors, ended as "o1" field times 4.

<offsl> For the one ZA quad out variant: is evector select offset, pointing to last of four consecutive vectors, encoded s "off2" field times 4 plus 3.

For the four Z quad-vectors an wo ZA quad-vectors variant: is the vector select offset, pointing to 'st of four consecutive vectors, encoded as "o1" field times 4 plus 3.

- <Zn> Is the nan. If the 1st source scalable vector register, encoded in the "Zn" field.
- <Zn1> Is "me on first so uble vector register of a multi-vector sequence, encoded as "Zn".
- <Tb> is the size specifier coded in "sz":



- <Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" plus 3 modulo 32.
- <Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" plus 1 modulo 32.
- <Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

```
1 CheckStreamingSVEAndZAEnabled();
2 constant integer VL = CurrentVL;
3 constant integer elements = VL DIV esize;
4 integer vectors = VL DIV 8;
5 integer vstride = vectors DIV nreg;
6 bits(32) vbase = X[v, 32];
7 integer vec = (UInt(vbase) + offset) MOD vstride;
```

```
8 bits(VL) result;
9
    vec = vec - (vec MOD 4);
10
11
    for r = 0 to nreg-1
        bits(VL) operand1 = Z[(n+r) MOD 32, VL];
12
13
        bits(VL) operand2 = Z[m, VL];
        for i = 0 to 3
14
15
             bits(VL) operand3 = ZAvector[vec + i, VL];
             for e = 0 to elements-1
16
                 integer element1 = SInt(Elem[operand1, 4 * e + i, esize DIV 4]);
17
                 integer element2 = SInt(Elem[operand2, 4 * e + i, esize DIV 4]);
18
                 bits(esize) product = (element1 * element2)<esize-1:0>;
Elem[result, e, esize] = Elem[operand3, e, esize] + product;
19
20
21
             ZAvector[vec + i, VL] = result;
22
        vec = vec + vstride;
```

DDI0616 B.a

D1.1.154 SMLALL (multiple vectors)

Multi-vector signed integer multiply-add long long

The instruction operates on two or four ZA quad-vector groups.

This signed integer multiply-add long long instruction multiplies each signed 8-bit or 16-bit element in the two or four first source vectors with each signed 8-bit or 16-bit element in the one, two, or four second source vectors, widens each product to 32-bits or 64-bits and destructively adds these values to the corresponding 32-bit or 64-bit elements of the two or four ZA quad-vector groups. The lowest of the four consecutive vector numbers forming the quad-vector group within each half or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA quad-vector groups respectively. The VECTOR GROUP symbol is preferred for disasser "t optional in assembler source code.

This instruction is unpredicated.

ID_AA64SMFR0_EL1.I16I64 indicates whether the 16-bit int er varian, implem ited.

It has encodings from 2 classes: Two ZA quad-vectors an our ZA ad-vectors

Two ZA quad-vectors (FEAT_SME2)

1 2

3 4 5

6

7

8



↔<Zm1>.<Tb>-<Zm4>.<Tb> }

```
if !HaveSME2() then UNDEFINED;
  if sz == '1' && !HaveSMEI16I64() then UNDEFINED;
2
3
   constant integer esize = 32 << UInt(sz);</pre>
4
  integer v = UInt('010':Rv);
5
  integer n = UInt(Zn:'00');
  integer m = UInt(Zm:'00');
6
  integer offset = UInt(o1:'00');
7
8 constant integer nreg = 4;
```

Assembler Symbols

<T> Is the size specifier, encoded in "sz":

SZ	<t></t>
0	S
1	D

- <Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
- <offsf> Is the vector select offset, pointing to first of four consecutive vectors, encoded as "o1" field times 4.
- <offsl> Is the vector select offset, pointing to last of four consecutive vectors, encoded as "o1" field times 4 plus 3.
- <Zn1> For the two ZA quad-vectors variant: is the name of the first smable vector register of a multi-vector sequence, encoded as "Zn" times 2.

For the four ZA quad-vectors variant: is the name of the lirst sc. ble vector egister of a multi-vector sequence, encoded as "Zn" times 4.

<Tb> Is the size specifier, encoded in "sz":

SZ	<tb></tb>
0	В
1	Н

- <Zn4> Is the name of the fourth scalable vector in the rector in the rector is a multi-vector sequence, encoded as "Zn" times 4 plus 3.
- <Zn2> Is the name of the secone scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.
- <Zm1> For the two ZA vad-vectors riant: is the name of the first scalable vector register of a multi-vector s valuence, encoded a. 'Zm" times 2.

For the fr ZA que vectors variant: is the name of the first scalable vector register of a multi-vector equation of the sector equation of th

- <Zm4> Is _____ me on ____ fourt' _scalable vector register of a multi-vector sequence, encoded as Zm" tin s 4 plus
- <Zm2> Is the times 2 plus 1.

```
CheckStreamingSVEAndZAEnabled();
1
2
   constant integer VL = CurrentVL;
   constant integer elements = VL DIV esize;
3
   integer vectors = VL DIV 8;
4
   integer vstride = vectors DIV nreg;
5
6
   bits(32) vbase = X[v, 32];
7
   integer vec = (UInt(vbase) + offset) MOD vstride;
8
   bits(VL) result;
9
   vec = vec - (vec MOD 4);
10
   for r = 0 to nreg-1
11
12
       bits(VL) operand1 = Z[n+r, VL];
       bits(VL) operand2 = Z[m+r, VL];
13
14
       for i = 0 to 3
           bits(VL) operand3 = ZAvector[vec + i, VL];
15
16
           for e = 0 to elements-1
17
                integer element1 = SInt(Elem[operand1, 4 * e + i, esize DIV 4]);
```

Chapter D1. SME instructions D1.1. SME and SME2 data-processing instructions

18	<pre>integer element2 = SInt(Elem[operand2, 4 * e + i, esize DIV 4]);</pre>
19	<pre>bits(esize) product = (element1 * element2)<esize-1:0>;</esize-1:0></pre>
20	<pre>Elem[result, e, esize] = Elem[operand3, e, esize] + product;</pre>
21	ZAvector[vec + i, VL] = result;
22	<pre>vec = vec + vstride;</pre>



D1.1.155 SMLSL (multiple and indexed vector)

Multi-vector signed integer multiply-subtract long by indexed element

The instruction operates on one, two, or four ZA double-vector groups.

This signed integer multiply-subtract long instruction multiplies each signed 16-bit element in the one, two, or four first source vectors with each signed 16-bit indexed element of the second source vector, widens each product to 32-bits and destructively subtracts these values from the corresponding 32-bit elements of the one, two, or four ZA double-vector groups.

The elements within the second source vector are specified using an immediate element index which selects the same element position within each 128-bit vector segment. The index range is from 0 to 7, encoded in 3 bits. The lowest of the two consecutive vector numbers forming the double-vector group within all, each half, or each quarter of the ZA array are selected by the sum of the vector select register mediate offset, modulo all, half, or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the \land operand consists of two or four ZA double-vector groups respectively. The VECTOR GROUP symbol is $_{\rm F}$ ferred for isassembly, but optional in assembler source code.

This instruction is unpredicated.

It has encodings from 3 classes: One ZA double-vector, Two double-vectors and Four ZA double-vectors

One ZA double-vector (FEAT_SME2)



```
3 integer v = UL ('010' .v);
4 integer n = UInt n'
```

1 2

```
5 integer " 'Int( ·Zm);
```

```
6 intege* Jffs = UIn. Sff 0');
```

```
7 inter index UInt(1 131);
```

```
8 cor ant int /er nreg = 1;
```

```
Two ZA uble-vectors
(FEAT_SN. ?)
```



 SMLSL
 ZA.S[<Wv>, <offsf>:<offsl>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H[<index>]

```
1 if !HaveSME2() then UNDEFINED;
```

```
2 constant integer esize = 32;
```

```
3 integer v = UInt('010':Rv);
```

```
4 integer n = UInt(Zn:'0');
```

```
5 integer m = UInt('0':Zm);
6 integer affinite VIT if (500 if it);
```

```
6 integer offset = UInt(off2:'0');
7 integer index = UInt(i3h:i3l);
```

```
8 constant integer nreg = 2;
```

Four ZA double-vectors (FEAT_SME2) 0 0 0 1 Zm 0 0 Rv Zn 1 1 0 1 i3h 1 0 SMLSL ZA.S[<Wv>, <offsf>:<offsl>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H[<index>] 1 if !HaveSME2() then UNDEFINED; 2 constant integer esize = 32; integer v = UInt('010':Rv); 3 4 integer n = UInt(Zn:'00'); integer m = UInt('0':Zm); 5 integer offset = UInt(off2:'0'); 6 integer index = UInt(i3h:i3l); constant **integer** nreg = 4; **Assembler Symbols** / field. $\langle Wv \rangle$ Is the 32-bit name of the vector select register W8.³ 11, encc d in the For the one ZA double-vector variant: is the vector electron offset, r inting to first of two <offsf> consecutive vectors, encoded as "off3" field <u>s</u> 2. For the four ZA double-vectors and two A double vectors At: is the vector select offset, pointing to first of two consecutive v 'ors, enc int as "off2" field times 2. <offsl> For the one ZA double-vector riant: . e vector select offset, pointing to last of two consecutive vectors, encoder'ns "o. "" fiela res 2 plus 1. For the four ZA double- tors and two \double-vectors variant: is the vector select offset, pointing to last of tw _on. "utive vectors nooded as "off2" field times 2 plus 1. Is the name of the first source _____able vector register, encoded in the "Zn" field. $\langle Zn \rangle$ $\langle Zn1 \rangle$ For the two double-vectors val. at: is the name of the first scalable vector register of a multi-vec c sequer , encoded as "Zn" times 2. For the four <u>v</u> ouble-vec rs variant: is the name of the first scalable vector register of a tor sec, ince, i loded as "Zn" times 4. m <Zn4>is the nr le of the full in scalable vector register of a multi-vector sequence, encoded as "Zn" times ls of name of the second scalable vector register of a multi-vector sequence, encoded as $\langle Zn2 \rangle$ "Zn nes 2 plus 1. Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field. $\langle Zm \rangle$ <index> Is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields. Operation

```
CheckStreamingSVEAndZAEnabled();
1
2
   constant integer VL = CurrentVL;
   constant integer elements = VL DIV esize;
3
4
   integer vectors = VL DIV 8;
   integer vstride = vectors DIV nreg;
5
   integer eltspersegment = 128 DIV esize;
6
   bits(32) vbase = X[v, 32];
7
   integer vec = (UInt(vbase) + offset) MOD vstride;
8
   bits(VL) result;
9
10
   vec = vec - (vec MOD 2);
11
12 for r = 0 to nreg-1
```

Chapter D1. SME instructions D1.1. SME and SME2 data-processing instructions

```
bits(VL) operand1 = Z[n+r, VL];
13
14
         bits(VL) operand2 = Z[m, VL];
15
          for i = 0 to 1
16
              bits(VL) operand3 = ZAvector[vec + i, VL];
17
               for e = 0 to elements-1
18
                    integer segmentbase = e - (e MOD eltspersegment);
                    integer s = 2 * segmentbase + index;
19
                   integer element1 = SInt(Elem[operand1, 2 * e + i, esize DIV 2]);
integer element2 = SInt(Elem[operand2, s, esize DIV 2]);
bits(esize) product = (element1 * element2)<esize-1:0>;
20
21
22
23
                   Elem[result, e, esize] = Elem[operand3, e, esize] - product;
24
               ZAvector[vec + i, VL] = result;
25
         vec = vec + vstride;
```

D1.1.156 SMLSL (multiple and single vector)

Multi-vector signed integer multiply-subtract long by vector

The instruction operates on one, two, or four ZA double-vector groups.

This signed integer multiply-subtract long instruction multiplies each signed 16-bit element in the one, two, or four first source vectors with each signed 16-bit element in the second source vector, widens each product to 32-bits and destructively subtracts these values from the corresponding 32-bit elements of the one, two, or four ZA double-vector groups. The lowest of the two consecutive vector numbers forming the double-vector group within all, each half, or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo all, half, or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA double-vector groups respectively. The VECTOR GROUP symbol is prover of the vector disassembly, but optional in assembler source code.

This instruction is unpredicated.

It has encodings from 3 classes: One ZA double-vector, Two 7, double-vectors and our ZA double-vectors

One ZA double-vector (FEAT_SME2)



SMLSL ZA.S[<Wv>, <offsf>:< ffsl . <Zn H, <Zm>.H

```
1 if !HaveSME2() then UNDEF
```

- 2 constant integer esize > .2;
- 3 integer v = UInt('010':xv);
- 4 integer n = UInt(Zn'
- 5 integer m = UInt(' :Zm);
- 6 integer offset = JInt(off3:'0');
- 7 constant **inte**(nreg



SMLSL ZA.S[<Wv>, <offsf>:<offsl>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H

- 1 if !HaveSME2() then UNDEFINED;
- 2 constant integer esize = 32;
- 3 integer v = UInt('010':Rv);
- 4 integer n = UInt(Zn);
- 5 integer m = UInt('0':Zm); 6 integer offset = UInt(off.
- 6 integer offset = UInt(off2:'0'); 7 constant integer nreg = 2;

Four ZA double-vectors (FEAT_SME2)



SMLSL ZA.S[<Wv>, <offsf>:<offsl>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H

- 1 if !HaveSME2() then UNDEFINED;
- 2 constant integer esize = 32;
- 3 integer v = UInt('010':Rv);
- 4 integer n = UInt(Zn);
- 5 integer m = UInt('0':Zm);
- 6 integer offset = UInt(off2:'0');
 7
- 7 constant **integer** nreg = 4;

Assembler Symbols

- <Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.

For the four ZA double-vectors and two ZA double-vectors variables is the vector elect offset, pointing to first of two consecutive vectors, encoded as "af2" field imes 2.

<offsl> For the one ZA double-vector variant: is the vect select offset, put is to last of two consecutive vectors, encoded as "off3" field times plus 1

For the four ZA double-vectors and two ZA double-vector variant: the vector select offset, pointing to last of two consecutive vector encoded as "to 2" for a times 2 plus 1.

- <Zn> Is the name of the first source scalab' vector rester, encoded in the "Zn" field.
- $\langle Zn1 \rangle$ Is the name of the first scalable \sim ctor region of a multi-vector sequence, encoded as "Zn".
- <Zn4> Is the name of the fourth sc?' ble ver regis of a multi-vector sequence, encoded as "Zn" plus 3 modulo 32.
- <Zn2> Is the name of the sond plable vector gister of a multi-vector sequence, encoded as "Zn" plus 1 modplo 32.
- <Zm> Is the name one second source so "able vector register Z0-Z15, encoded in the "Zm" field.

```
CheckStre qSVEA 7AEnabl ();
constar int, rVL Cur .tVL;
const .t inte :r eleme = VL DIV esize;
 1
2
 3
    int er vect s = VL DIV 8;
inte. VS rate Cors D
 4
                           .ors DIV nreg;
 5
    bits(3. pase = X[v, 32];
integer = (UInt(vbase) + offset) MOD vstride;
6
 7
    bits(VL) r lt;
vec = vec - vec MOD 2);
 8
9
10
11
    for r = 0 to nreg-1
         bits(VL) operand1 = Z[(n+r) MOD 32, VL];
12
13
         bits(VL) operand2 = Z[m, VL];
14
         for i = 0 to 1
15
             bits(VL) operand3 = ZAvector[vec + i, VL];
16
              for e = 0 to elements-1
17
                  integer element1 = SInt(Elem[operand1, 2 * e + i, esize DIV 2]);
18
                  integer element2 = SInt(Elem[operand2, 2 * e + i, esize DIV 2]);
19
                  bits(esize) product = (element1 * element2)<esize-1:0>;
20
                  Elem[result, e, esize] = Elem[operand3, e, esize] - product;
21
             ZAvector[vec + i, VL] = result;
22
      vec = vec + vstride;
```

D1.1.157 SMLSL (multiple vectors)

Multi-vector signed integer multiply-subtract long

The instruction operates on two or four ZA double-vector groups.

This signed integer multiply-subtract long instruction multiplies each signed 16-bit element in the two or four first source vectors with each signed 16-bit element in the two or four second source vectors, widens each product to 32-bits and destructively subtracts these values from the corresponding 32-bit elements of the two or four ZA double-vector groups. The lowest of the two consecutive vector numbers forming the double-vector group within each half or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA double-vector groups respectively. The VECTOR GROUP symbol is project of disassembly, but optional in assembler source code.

This instruction is unpredicated.

It has encodings from 2 classes: Two ZA double-vectors and F at ZA dou. 2-vector

Two ZA double-vectors (FEAT_SME2)



```
6 integer offset = UInt(off2:'0');
```

```
7 constant integer nreg = 4;
```

Assembler Symbols

- <Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
- <offsf> Is the vector select offset, pointing to first of two consecutive vectors, encoded as "off2" field times 2.

Chapter D1. SME instructions

D1.1. SME and SME2 data-processing instructions

- <offsl> Is the vector select offset, pointing to last of two consecutive vectors, encoded as "off2" field times 2 plus 1.
- <Zn1> For the two ZA double-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.

For the four ZA double-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

- <Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.
- <Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.
- <Zm1> For the two ZA double-vectors variant: is the name of the first scale¹ vector register of a multi-vector sequence, encoded as "Zm" times 2.

For the four ZA double-vectors variant: is the name of the fin. calable vecto. egister of a multi-vector sequence, encoded as "Zm" times 4.

- <Zm4> Is the name of the fourth scalable vector register of multi-vector rue c, encoded as "Zm" times 4 plus 3.
- <Zm2> Is the name of the second scalable vector register o. _____iti-vecto__equence, encoded as "Zm" times 2 plus 1.

```
1
   CheckStreamingSVEAndZAEnabled();
2
   constant integer VL = CurrentVL;
   constant integer elements = V DIV
3
                                            ze;
4
   integer vectors = VL DIV 8;
   integer vstride = vectors
5
                                  1 nreq;
6
   bits(32) vbase = X[v, 3'
   integer vec = (UInt(vbase) + o. et) MOD vstride;
7
   bits(VL) result;
8
9
                       5 2);
   vec = vec - (vec)
10
   for r = 0 to i g-1
11
       bits(VL) op and = Z[n+r. VL];
bits(VT) oper Z = Z[m+ VL];
12
13
        bits(VT ) oper
14
        for
                  to 1
            bits( ) oper
                            A = ZAvector[vec + i, VL];
15
16
            for r = 0 to el ments-1
17
                          lement1 = SInt(Elem[operand1, 2 * e + i, esize DIV 2]);
                integer element2 = SInt(Elem[operand2, 2 * e + i, esize DIV 2]);
18
19
                bits(esize) product = (element1 * element2)<esize-1:0>;
20
                Elem[result, e, esize] = Elem[operand3, e, esize] - product;
21
            ZAve or[vec + i, VL] = result;
22
        vec = vec + vstride;
```

D1.1.158 SMLSLL (multiple and indexed vector)

Multi-vector signed integer multiply-subtract long long by indexed element

The instruction operates on one, two, or four ZA quad-vector groups.

This signed integer multiply-subtract long long instruction multiplies each signed 8-bit or 16-bit element in the one, two, or four first source vectors with each signed 8-bit or 16-bit indexed element of second source vector, widens each product to 32-bits or 64-bits and destructively subtracts these values from the corresponding 32-bit or 64-bit elements of the one, two, or four ZA quad-vector groups.

The elements within the second source vector are specified using an immediate element index which selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 3 to 4 bits depending on the size of the element. The lowest of the four consecutive vector numbers forming the quad-vector group within all half, or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, not all half, or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the 7 operation consists of voor four ZA quad-vector groups respectively. The VECTOR GROUP symbol is preferre for disassen. v, but ptional in assembler source code.

This instruction is unpredicated.

ID_AA64SMFR0_EL1.I16I64 indicates whether leng it integination is implemented.

It has encodings from 6 classes: One ZA quad-vector of 2-bit elements, One ZA quad-vector of 64-bit elements, Two ZA quad-vectors of 32-bit elements, Tv ZA quad-vectors of 64-bit elements, Four ZA quad-vectors of 64-bit elements

One ZA quad-vector of 32-bit e' ...ents (FEAT_SME2)



SMLSLL ZA.S[<Wv, offsf>: ffsl>], <Zn>.B, <Zm>.B[<index>]

```
if !H eSME2 ( then
1
                             NED.
2
  con
       ant int er esize
                            32:
3
  inte
        ~ v
                        :Rv);
4
               UInt(Zn);
  intege
             = UInt('0':Zm);
5
  integer
  integer on et = UInt(off2:'00');
6
  integer inde = UInt(i4h:i4l);
7
8
```

```
constant integer nreg = 1;
```

One ZA quad-vector of 64-bit elements (FEAT_SME_I16I64)



SMLSLL ZA.D[<Wv>, <offsf>:<offsl>], <Zn>.H, <Zm>.H[<index>]

if !(HaveSME2() && HaveSMEI16I64()) then UNDEFINED;

```
2 constant integer esize = 64;
```

```
3 integer v = UInt('010':Rv);
```

```
4 integer n = UInt(Zn);
```

1

Chapter D1. SME instructions D1.1. SME and SME2 data-processing instructions

- integer m = UInt('0':Zm); 5
- 6 integer offset = UInt(off2:'00');
- integer index = UInt(i3h:i3l); 7
- 8 constant integer nreg = 1;

Two ZA quad-vectors of 32-bit elements (FEAT_SME2)



SMLSLL ZA.S[<Wv>, <offsf>:<offsl>{, VGx2}], { <Zn1>.B-<Zn2>.B }, <Zm>.B[<index>]

- if !HaveSME2() then UNDEFINED; 1
- 2 constant integer esize = 32;
- 3 integer v = UInt('010':Rv);
- integer n = UInt(Zn:'0'); 4 5
- integer m = UInt('0':Zm); 6
- integer offset = UInt(o1:'00'); integer index = UInt(i4h:i4l); 7
- 8 constant integer nreg = 2;

Two ZA quad-vectors of 64-bit elements (FEAT SME I16I64)



SMLSLL ZA.D[<Wv>, <off >> offsl>{, Vc }], { <Zn1>.H-<Zn2>.H }, <Zm>.H[<index>]

1

- constant integer < _ze = 64; integer v = UIn⁺ 010':R^x); integer n = U⁺ (Zn:'0' 2
- 3
- 4
- 5
- 6 7
- integer m = UIh. '0' ...);
 integer offset = _ (o1:'00'
 integer = UIh. 'i3h:i',
 constat inte ;r nreg 8

Fou 'A qv f 32-bit elements (FEA1_ .£2) 0 0 0 0 1 1 0 Zm 0 0 1 0 0 1 1 Rv 0 i4h 0 1 П_

SMLSLL ZA.S[<Wv>, <offsf>:<offsl>{, VGx4}], { <Zn1>.B-<Zn4>.B }, <Zm>.B[<index>]

- 1 if !HaveSME2() then UNDEFINED;
- 2 constant integer esize = 32;
- 3 integer v = UInt('010':Rv);
- 4 integer n = UInt(Zn:'00');
- 5 integer m = UInt('0':Zm);
- 6 integer offset = UInt(o1:'00'); integer index = UInt(i4h:i4l); 7
- 8 constant integer nreg = 4;

Four ZA quad-vectors of 64-bit elements (FEAT_SME_I16I64)

01

i4l



SMLSLL ZA.D[<Wv>, <offsf>:<offsl>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H[<index>]

- if !(HaveSME2() && HaveSMEI16I64()) then UNDEFINED;
- 2 constant integer esize = 64;
- 3 integer v = UInt('010':Rv); 4
- integer n = UInt(Zn:'00');
- 5 integer m = UInt('0':Zm);
- 6 integer offset = UInt(o1:'00');
- integer index = UInt(i3h:i3l); 7
- 8 constant integer nreg = 4;

Assembler Symbols

<Wv> Is the 32-bit name of the vector select register W8-W11, encode in the "Rv" t ld.

For the one ZA quad-vector of 32-bit elements and or $\angle A$ quad-vector of f bit elements <offsf> variant: is the vector select offset, pointing to first four corecutive .ors, encoded as "off2" field times 4.

> four. quad-y cors of 64-bit elements, For the four ZA quad-vectors of 32-bit eler two ZA quad-vectors of 32-bit elements J two. A quad- re of 64-bit elements variant: is the vector select offset, pointing to st of four consecutive vectors, encoded as "o1" field times 4.

For the one ZA quad-vector of 32 it elen. 's and one ZA quad-vector of 64-bit elements <offsl> variant: is the vector selec' ifset, porting to st of four consecutive vectors, encoded as "off2" field times 4 plus

> For the four ZA qu. -vecto. of 32-bit elements, four ZA quad-vectors of 64-bit elements, two ZA quad-vectors of 32-bit ments and two ZA quad-vectors of 64-bit elements variant: is the vector ect offset, pointing last of four consecutive vectors, encoded as "o1" field times 4 ply 5.

- Is the name ^cth *inst source scalable vector register, encoded in the "Zn" field.* $\langle Zn \rangle$
- <Zn1> F inc 'o ZA ad-ve is of 32-bit elements and two ZA quad-vectors of 64-bit elements √ariant: the name the first scalable vector register of a multi-vector sequence, encoded

he four ZA quad-vectors of 32-bit elements and four ZA quad-vectors of 64-bit elements var. t: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn imes 4.

- <Zn4>Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.
- Is the name of the second scalable vector register of a multi-vector sequence, encoded as <Zn2>"Zn" times 2 plus 1.
- <Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.
- <index> For the four ZA quad-vectors of 32-bit elements, one ZA quad-vector of 32-bit elements and two ZA quad-vectors of 32-bit elements variant: is the element index, in the range 0 to 15, encoded in the "i4h:i4l" fields.

For the four ZA quad-vectors of 64-bit elements, one ZA quad-vector of 64-bit elements and two ZA quad-vectors of 64-bit elements variant: is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields.

```
CheckStreamingSVEAndZAEnabled();
 1
    constant integer VL = CurrentVL;
 2
3
   constant integer elements = VL DIV esize;
4 integer vectors = VL DIV 8;
    integer vstride = vectors DIV nreg;
5
   integer eltspersegment = 128 DIV esize;
6
7
    bits(32) vbase = X[v, 32];
    integer vec = (UInt(vbase) + offset) MOD vstride;
8
9
    bits(VL) result;
   vec = vec - (vec MOD 4);
10
11
12
    for r = 0 to nreg-1
        bits(VL) operand1 = Z[n+r, VL];
13
14
        bits(VL) operand2 = Z[m, VL];
15
         for i = 0 to 3
             bits(VL) operand3 = ZAvector[vec + i, VL];
16
17
             for e = 0 to elements-1
18
                  integer segmentbase = e - (e MOD eltspersegme. `;
19
                  integer s = 4 * segmentbase + index;
                 integer element1 = SInt(Elem[operand1, 4 e + i, size
integer element2 = SInt(Elem[operand2, , esize DIV ');
bits(esize) product = (element1 * el ent2)<e ze-1:0</pre>
20
                                                                              rize DIV
                                                                                        ·]);
21
22
23
                  Elem[result, e, esize] = Elem[operal <sup>2</sup>, e,
                                                                    _ze] - product;
24
             ZAvector[vec + i, VL] = result;
25
        vec = vec + vstride;
```

D1.1.159 SMLSLL (multiple and single vector)

Multi-vector signed integer multiply-subtract long long by vector

The instruction operates on one, two, or four ZA quad-vector groups.

This signed integer multiply-subtract long long instruction multiplies each signed 8-bit or 16-bit element in the one, two, or four first source vectors with each signed 8-bit or 16-bit element in the second source vector, widens each product to 32-bits or 64-bits and destructively subtracts these values from the corresponding 32-bit or 64-bit elements of the one, two, or four ZA quad-vector groups. The lowest of the four consecutive vector numbers forming the quad-vector group within all, each half, or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo all, half, or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA quad-vector groups respectively. The VECTOR GROUP symbol is preferred for disasser to prove optional in assembler source code.

This instruction is unpredicated.

ID_AA64SMFR0_EL1.I16I64 indicates whether the 16-bit int _er varian. implem_ted.

It has encodings from 3 classes: One ZA quad-vector, Tv ZA quad ectors a our ZA quad-vectors

One ZA quad-vector (FEAT_SME2)



SMLSLL ZA.<T>[<Wv>, <of t>:<offsl> <Zn>.<Tb>, <Zm>.<Tb>

```
if !HaveSME2() then UNL_FINED;
1
                                           hen UNDEFINED;
2
   if sz == '1' && !Hay CMEI16I64()
3
   constant integer < _ze = 32 << UIn sz);</pre>
                       010':R');
4
   integer v = UIn+
5
   integer n = U<sup>T</sup> .(Zn);
   integer m = UIn
integer offset =
6
                        '0'
                              .0 ;
7
                           .(off2:
8
   constan+
                    er ni
                             = 1;
```

Tw 'A quad ectors (FEA SN .2)



SMLSLL ZA.<T>[<Wv>, <offsf>:<offsl>{, VGx2}], { <Zn1>.<Tb>-<Zn2>.<Tb> }, <Zm>.<Tb>

```
1
  if !HaveSME2() then UNDEFINED;
  if sz == '1' && !HaveSMEI16I64() then UNDEFINED;
2
3
  constant integer esize = 32 << UInt(sz);</pre>
  integer v = UInt('010':Rv);
4
5
  integer n = UInt(Zn);
  integer m = UInt('0':Zm);
6
  integer offset = UInt(o1:'00');
7
8
  constant integer nreg = 2;
```

Four ZA quad-vectors (FEAT_SME2)



SMLSLL ZA.<T>[<Wv>, <offsf>:<offsl>{, VGx4}], { <Zn1>.<Tb>-<Zn4>.<Tb> }, <Zm>.<Tb>

```
1 if !HaveSME2() then UNDEFINED;
```

- 2 if sz == '1' && !HaveSMEI16I64() then UNDEFINED;
- 3 constant integer esize = 32 << UInt(sz);</pre>
- 4 integer v = UInt('010':Rv);
- 5 integer n = UInt(Zn);
- 6 integer m = UInt('0':Zm);
- 7 integer offset = UInt(o1:'00');
- 8 constant **integer** nreg = 4;

Assembler Symbols

<T> Is the size specifier, encoded in "sz":

SZ	< T >
0	S
1	D

- <Wv> Is the 32-bit name of the vector select regimer v, W11, ordgiment with the "Rv" field.
- $\langle offsf \rangle$ For the one ZA quad-vector variant is the vector select outset, pointing to first of four consecutive vectors, encoded as "off2 vectors 4.

For the four ZA quad-vectors and vo ZA ad-vectors variant: is the vector select offset, pointing to first of four conjuctive v tors, ended as "o1" field times 4.

<offsl> For the one ZA quad our variant: is evector select offset, pointing to last of four consecutive vectors, encode of "off2" field times 4 plus 3.

For the four Z quad-vectors an wo ZA quad-vectors variant: is the vector select offset, pointing to 'st of four consecutive vectors, encoded as "o1" field times 4 plus 3.

- <Zn> Is the nan. If the 1st source scalable vector register, encoded in the "Zn" field.
- <Zn1> Is "me on first so uble vector register of a multi-vector sequence, encoded as "Zn".
- <Tb> is the size specifier coded in "sz":



- <Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" plus 3 modulo 32.
- <Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" plus 1 modulo 32.
- <Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

```
1 CheckStreamingSVEAndZAEnabled();
2 constant integer VL = CurrentVL;
3 constant integer elements = VL DIV esize;
4 integer vectors = VL DIV 8;
5 integer vstride = vectors DIV nreg;
6 bits(32) vbase = X[v, 32];
7 integer vec = (UInt(vbase) + offset) MOD vstride;
```

```
8 bits(VL) result;
9
    vec = vec - (vec MOD 4);
10
11
    for r = 0 to nreg-1
        bits(VL) operand1 = Z[(n+r) MOD 32, VL];
12
13
        bits(VL) operand2 = Z[m, VL];
        for i = 0 to 3
14
15
             bits(VL) operand3 = ZAvector[vec + i, VL];
             for e = 0 to elements-1
16
                 integer element1 = SInt(Elem[operand1, 4 * e + i, esize DIV 4]);
17
                 integer element2 = SInt(Elem[operand2, 4 * e + i, esize DIV 4]);
18
                 bits(esize) product = (element1 * element2)<esize-1:0>;
Elem[result, e, esize] = Elem[operand3, e, esize] - product;
19
20
21
             ZAvector[vec + i, VL] = result;
22
        vec = vec + vstride;
```

DDI0616 B.a

D1.1.160 SMLSLL (multiple vectors)

Multi-vector signed integer multiply-subtract long long

The instruction operates on two or four ZA quad-vector groups.

This signed integer multiply-subtract long long instruction multiplies each signed 8-bit or 16-bit element in the two or four first source vectors with each signed 8-bit or 16-bit element in the one, two, or four second source vectors, widens each product to 32-bits or 64-bits and destructively subtracts these values from the corresponding 32-bit or 64-bit elements of the two or four ZA quad-vector groups. The lowest of the four consecutive vector numbers forming the quad-vector group within each half or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA quad-vector groups respectively. The VECTOR GROUP symbol is preferred for disasser to provide the provide the preferred for disasser to provide the provide the preferred for disasser to provide the providet

This instruction is unpredicated.

ID_AA64SMFR0_EL1.I16I64 indicates whether the 16-bit int _er varian. implem_ted.

It has encodings from 2 classes: Two ZA quad-vectors an our ZA ad-vector

Two ZA quad-vectors (FEAT_SME2)



```
1 if !HaveSME2() then UNDEFINED;
2 if sz == '1' && !HaveSMEI16I64() then UNDEFINED;
3 constant integer esize = 32 << UInt(sz);
4 integer v = UInt('010':Rv);
5 integer n = UInt(Zn:'00');
6 integer m = UInt(Zm:'00');
7 integer offset = UInt(o1:'00');
8 constant integer nreg = 4;
```

Assembler Symbols

<T> Is the size specifier, encoded in "sz":

SZ	<t></t>
0	S
1	D

- <Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
- <offsf> Is the vector select offset, pointing to first of four consecutive vectors, encoded as "o1" field times 4.
- <offsl> Is the vector select offset, pointing to last of four consecutive vectors, encoded as "o1" field times 4 plus 3.
- <Zn1> For the two ZA quad-vectors variant: is the name of the first smable vector register of a multi-vector sequence, encoded as "Zn" times 2.

For the four ZA quad-vectors variant: is the name of the lirst sc. ble vector egister of a multi-vector sequence, encoded as "Zn" times 4.

<Tb> Is the size specifier, encoded in "sz":

SZ	<tb></tb>
0	В
1	Н

- <Zn4> Is the name of the fourth scalable vector in the rector in the rector is a multi-vector sequence, encoded as "Zn" times 4 plus 3.
- <Zn2> Is the name of the secone scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.
- <Zm1> For the two ZA vad-vectors riant: is the name of the first scalable vector register of a multi-vector s valuence, encoded a. 'Zm" times 2.

For the fr ZA que vectors variant: is the name of the first scalable vector register of a multi-vector equation of the sector equation of th

- <Zm4> Is _____ me on ____ fourt' _scalable vector register of a multi-vector sequence, encoded as Zm" tin s 4 plus
- <Zm2> Is the times 2 plus 1.

```
CheckStreamingSVEAndZAEnabled();
1
2
   constant integer VL = CurrentVL;
   constant integer elements = VL DIV esize;
3
   integer vectors = VL DIV 8;
4
   integer vstride = vectors DIV nreg;
5
6
   bits(32) vbase = X[v, 32];
7
   integer vec = (UInt(vbase) + offset) MOD vstride;
8
   bits(VL) result;
9
   vec = vec - (vec MOD 4);
10
   for r = 0 to nreg-1
11
12
       bits(VL) operand1 = Z[n+r, VL];
       bits(VL) operand2 = Z[m+r, VL];
13
14
       for i = 0 to 3
           bits(VL) operand3 = ZAvector[vec + i, VL];
15
16
           for e = 0 to elements-1
17
                integer element1 = SInt(Elem[operand1, 4 * e + i, esize DIV 4]);
```

Chapter D1. SME instructions D1.1. SME and SME2 data-processing instructions

18	<pre>integer element2 = SInt(Elem[operand2, 4 * e + i, esize DIV 4]);</pre>
19	<pre>bits(esize) product = (element1 * element2)<esize-1:0>;</esize-1:0></pre>
20	<pre>Elem[result, e, esize] = Elem[operand3, e, esize] - product;</pre>
21	ZAvector[vec + i, VL] = result;
22	vec = vec + vstride;



D1.1.161 SMOPA (2-way)

Signed integer sum of outer products and accumulate

This instruction works with a 32-bit element ZA tile.

The signed integer sum of outer products and accumulate instructions multiply the sub-matrix in the first source vector by the sub-matrix in the second source vector. The first source holds $SVL_S \times 2$ sub-matrix of signed 16-bit integer values, and the second source holds $2 \times SVL_S$ sub-matrix of signed 16-bit integer values.

Each source vector is independently predicated by a corresponding governing predicate. When a 16-bit source element is inactive, it is treated as having the value 0.

The resulting $SVL_S \times SVL_S$ widened 32-bit integer sum of outer products is then destructively added to the 32-bit integer destination tile. This is equivalent to performing a 2-way dot product and accumulate to each of the destination tile elements.

Each 32-bit container of the first source vector holds 2 consecutive summer of each row of a $SVL_S \times 2$ sub-matrix, and each 32-bit container of the second source vector holds 2 onsecutive root elements of each column of a $2 \times SVL_S$ sub-matrix.

SME2 (FEAT_SME2)

0 0 0 0 0 Pm 7n 7Ada 0 1 0 Pn 0 1 0 Ls Lu0

SMOPA <ZAda>.S, <Pn>/M, <P >/M, <Zn>.h. <Zm>.H

```
1 if !HaveSME2() then UNDEF
```

- 2 constant **integer** esize 32
- 3 integer a = UInt(Pn);
- 4 integer b = UInt (Pm'
- 5 integer n = UInt(7 /;
- 6 integer m = UIn⁺ am);
- 7 integer da = V (ZAda)
- 8 boolean sub_op FAL
- 9 boolean unsigned ALSE;

Asser her Syr ols

- <ZAda> Is the ZA tile ZA0-ZA3, encoded in the "ZAda" field.
 - <Pn> Is name of the first governing scalable predicate register P0-P7, encoded in the "Pn" field.
 - <Pm> Is the . ne of the second governing scalable predicate register P0-P7, encoded in the "Pm" field.
 - <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
 - <Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

```
1
   CheckStreamingSVEAndZAEnabled();
2
   constant integer VL = CurrentVL;
3
   constant integer PL = VL DIV 8;
Δ
   constant integer dim = VL DIV esize;
5
   bits(PL) mask1 = P[a, PL];
6
   bits(PL) mask2 = P[b, PL];
   bits(VL) operand1 = Z[n, VL];
7
   bits(VL) operand2 = Z[m, VL];
8
9
   bits(dim*dim*esize) operand3 = ZAtile[da, esize, dim*dim*esize];
10
   bits(dim*dim*esize) result;
```

```
Chapter D1. SME instructions
D1.1. SME and SME2 data-processing instructions
```

```
integer prod;
11
12
13
   for row = 0 to dim-1
14
      for col = 0 to dim-1
15
          bits(esize) sum = Elem[operand3, row*dim+col, esize];
16
          for k = 0 to 1
              if ActivePredicateElement(mask1, 2*row + k, esize DIV 2) &&
17
18
                     ActivePredicateElement(mask2, 2*col + k, esize DIV 2) then
                 19
20
21
                 if sub_op then prod = -prod;
22
                 sum = sum + prod;
23
24
          Elem[result, row*dim+col, esize] = sum;
25
26
   ZAtile[da, esize, dim*dim*esize] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of
 - The values of the data supplied in any of its o rand registers when a governing predicate registers contain the same value for each execution.
 - The values of the NZCV flags.
- The response of this instruction to asyn fromous eleptions de s not vary based on:
 - The values of the data supplied in a. of .s operand registers when its governing predicate registers contain the same value for each vecution
 - The values of the NZC' dags.

D1.1.162 SMOPA (4-way)

Signed integer sum of outer products and accumulate

The 8-bit integer variant works with a 32-bit element ZA tile.

The 16-bit integer variant works with a 64-bit element ZA tile.

The signed integer sum of outer products and accumulate instructions multiply the sub-matrix in the first source vector by the sub-matrix in the second source vector. In case of the 8-bit integer variant, the first source holds $SVL_S \times 4$ sub-matrix of signed 8-bit integer values, and the second source holds $4 \times SVL_S$ sub-matrix of signed 8-bit integer values. In case of the 16-bit integer variant, the first source holds $SVL_D \times 4$ sub-matrix of signed 16-bit integer values, and the second source holds $4 \times SVL_D \times 4$ sub-matrix of signed 16-bit integer values.

Each source vector is independently predicated by a corresponding governing predicate. When an 8-bit source element in case of 8-bit integer variant or a 16-bit source element in case 10-c. Pteger variant is Inactive, it is treated as having the value 0.

The resulting $SVL_S \times SVL_S$ widened 32-bit integer or $SVL_D \times SVL_D$ where d64-bit in ger sum of outer products is then destructively added to the 32-bit integer or 64-bit integer destination ile, resp. dively for 8-bit integer and 16-bit integer instruction variants. This is equivalent to perfer hing a 4-way derived and accumulate to each of the destination tile elements.

In case of the 8-bit integer variant, each 32-bit container of first sour vector holds 4 consecutive column elements of each row of a $SVL_S \times 4$ sub-matrix and the h 32-b. contrast of the second source vector holds 4 consecutive row elements of each column of a f SVL_S sub-matrix. The second source vector holds 4 consecutive row of the first source vector holds 4 consecutive row of a $SVL_D \times 4$ sub-matrix, and each 64-bit container of the second source vector holds 4 consecutive row elements of each column of a $4 \times SVL_S$ sub-matrix.

It has encodings from 2 classis, bit and 64-bit



```
SMOPA <ZAda>.D, <Pn>/M, <Pm>/M, <Zn>.H, <Zm>.H
1
   if !HaveSMEI16I64() then UNDEFINED;
2
   constant integer esize = 64;
  integer a = UInt(Pn);
3
4
   integer b = UInt(Pm);
5
   integer n = UInt(Zn);
  integer m = UInt(Zm);
6
   integer da = UInt(ZAda);
7
8
   boolean sub_op = FALSE;
9
   boolean op1_unsigned = FALSE;
10 boolean op2_unsigned = FALSE;
   Assembler Symbols
```

- <ZAda> For the 32-bit variant: is the name of the ZA tile ZA0-ZA3, encoded in the "ZAda" field. For the 64-bit variant: is the name of the ZA tile ZA0-ZA7, enc., ed in the Ada" field.
 - For the 64-bit variant: is the name of the ZA tile ZA0-ZA7, enc. led in the Ada" field. <Pn> Is the name of the first governing scalable predicate register P0-. encoded in t "Pn" field.
 - <Pm> Is the name of the second governing scalable predicate _gister P0-, encod _ in the "Pm"
- field.
 - <Zn> Is the name of the first source scalable vector regis. enc. ed in the "Zn" field.
- <Zm> Is the name of the second source scalable y registe 'ncode' in the "Zm" field.

Operation

```
1
   CheckStreamingSVEAndZAEnabled();
2
   constant integer VL = CurrentVL;
3
   constant integer PL = VL DIV
   constant integer dim = VL D
4
                                 esize;
5
   bits(PL) mask1 = P[a, PL]
   bits(PL) mask2 = P[b, P'
6
7
   bits(VL) operand1 = Z[n, VL];
8
   bits(VL) operand2 = [m, VL];
   bits(dim*dim*esize operand3 = ZAt >[da, esize, dim*dim*esize];
9
10
   bits(dim*dim*es' .) resu't;
11
   integer prod
12
13
   for row = 0 to d_{\perp}
14
        for
                 0 to
                        m-1
15
            bits ( ize)
                              _lem[operand3, row*dim+col, esize];
                   = 0 to 3
16
            for }
17
                        wePredicateElement(mask1, 4*row + k, esize DIV 4) &&
18
                        ActivePredicateElement(mask2, 4*col + k, esize DIV 4) then
19
                    prod = (Int(Elem[operand1, 4*row + k, esize DIV 4], op1_unsigned) *
                            Int(Elem[operand2, 4*col + k, esize DIV 4], op2_unsigned));
20
21
                    if sub_op then prod = -prod;
22
                    sum = sum + prod;
23
24
            Elem[result, row*dim+col, esize] = sum;
25
26
   ZAtile[da, esize, dim*dim*esize] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
 - The values of the data supplied in any of its operand registers when its governing predicate registers contain the same value for each execution.
 - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
- The values of the data supplied in any of its operand registers when its governing predicate registers contain the same value for each execution.
- The values of the NZCV flags.



D1.1.163 SMOPS (2-way)

Signed integer sum of outer products and subtract

This instruction works with a 32-bit element ZA tile.

The signed integer sum of outer products and subtract instructions multiply the sub-matrix in the first source vector by the sub-matrix in the second source vector. The first source holds $SVL_S \times 2$ sub-matrix of signed 16-bit integer values, and the second source holds $2 \times SVL_{S}$ sub-matrix of signed 16-bit integer values.

Each source vector is independently predicated by a corresponding governing predicate. When a 16-bit source element is inactive, it is treated as having the value 0.

The resulting $SVL_S \times SVL_S$ widened 32-bit integer sum of outer products is then destructively subtracted from the 32-bit integer destination tile. This is equivalent to performing a 2-way dot product and subtract from each of the destination tile elements.

Each 32-bit container of the first source vector holds 2 consecutive (umn element) f each row of a $SVL_S \times 2$ sub-matrix, and each 32-bit container of the second source vector bolds _____nsecutive rc elements of each column of a $2 \times SVL_S$ sub-matrix.

SME2 (FEAT SME2)

0 0 0 0 0 Pm 7n 7Ada 0 1 0 Pn 1 1 0 Ĺs Lu0

SMOPS <7Ada>.S. <Pn>/M. 7m>.H7.n>

```
if !HaveSME2() then UNDEF
1
```

- 2 constant **integer** esize
- 3 integer a = UInt(Pn);
- integer b = UInt(Pm' 4
- integer n = UInt(" /; 5
- 6 integer m = UIn⁺ _m); 7
- integer da = V (t (ZAda
- boolean sub_op 8 TRUT LSE;
- boolean unsigned

Asser her Syr ols

- <ZAda> 's the ZA tile ZA0-ZA3, encoded in the "ZAda" field.
 - Is name of the first governing scalable predicate register P0-P7, encoded in the "Pn" field. <Pn>
 - Is the . ne of the second governing scalable predicate register P0-P7, encoded in the "Pm" <Pm> field.
 - $\langle Zn \rangle$ Is the name of the first source scalable vector register, encoded in the "Zn" field.
 - <Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

```
1
   CheckStreamingSVEAndZAEnabled();
2
   constant integer VL = CurrentVL;
   constant integer PL = VL DIV 8;
3
Δ
   constant integer dim = VL DIV esize;
5
   bits(PL) mask1 = P[a, PL];
6
   bits(PL) mask2 = P[b, PL];
   bits(VL) operand1 = Z[n, VL];
7
   bits(VL) operand2 = Z[m, VL];
8
9
   bits(dim*dim*esize) operand3 = ZAtile[da, esize, dim*dim*esize];
10
   bits(dim*dim*esize) result;
```

```
Chapter D1. SME instructions
D1.1. SME and SME2 data-processing instructions
```

```
integer prod;
11
12
13
   for row = 0 to dim-1
14
      for col = 0 to dim-1
15
          bits(esize) sum = Elem[operand3, row*dim+col, esize];
16
          for k = 0 to 1
              if ActivePredicateElement(mask1, 2*row + k, esize DIV 2) &&
17
18
                     ActivePredicateElement(mask2, 2*col + k, esize DIV 2) then
                 19
20
21
                 if sub_op then prod = -prod;
22
                 sum = sum + prod;
23
24
          Elem[result, row*dim+col, esize] = sum;
25
26
   ZAtile[da, esize, dim*dim*esize] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of
 - The values of the data supplied in any of its contain the same value for each execution.
 - The values of the NZCV flags.
- The response of this instruction to asyn fromous eleptions de s not vary based on:
 - The values of the data supplied in a. of .s operand registers when its governing predicate registers contain the same value for each vecution
 - The values of the NZC' dags.

D1.1.164 SMOPS (4-way)

Signed integer sum of outer products and subtract

The 8-bit integer variant works with a 32-bit element ZA tile.

The 16-bit integer variant works with a 64-bit element ZA tile.

The signed integer sum of outer products and subtract instructions multiply the sub-matrix in the first source vector by the sub-matrix in the second source vector. In case of the 8-bit integer variant, the first source holds $SVL_S \times 4$ sub-matrix of signed 8-bit integer values, and the second source holds $4 \times SVL_S$ sub-matrix of signed 8-bit integer values. In case of the 16-bit integer variant, the first source holds $SVL_D \times 4$ sub-matrix of signed 16-bit integer values, and the second source holds $SVL_D \times 4$ sub-matrix of signed 16-bit integer values.

Each source vector is independently predicated by a corresponding governing predicate. When an 8-bit source element in case of 8-bit integer variant or a 16-bit source element in case 10-0. Teger variant is Inactive, it is treated as having the value 0.

The resulting $SVL_S \times SVL_S$ widened 32-bit integer or $SVL_D \times SVL_D$ where d64-bit in ger sum of outer products is then destructively subtracted from the 32-bit integer or 64 at integer stination dile, respectively for 8-bit integer and 16-bit integer instruction variants. This is equivalent to perform, a vary dot product and subtract from each of the destination tile elements.

In case of the 8-bit integer variant, each 32-bit container on dirst sour vector holds 4 consecutive column elements of each row of a $SVL_S \times 4$ sub-matrix and the h 32-b. Fonto der of the second source vector holds 4 consecutive row elements of each column of a f SVL_S sub-matrix. The second source vector holds 4 consecutive row of the first source vector holds 4 consecutive row elements of each column of a f SVL_S sub-matrix. The second source vector holds 4 consecutive row elements of each column of a f SVL_S sub-matrix. The second source vector holds 4 consecutive row elements of each column of a $4 \times SVL_D$ sub-matrix.

It has encodings from 2 classes by bit and 64-bit



```
SMOPS <ZAda>.D, <Pn>/M, <Pm>/M, <Zn>.H, <Zm>.H
1
   if !HaveSMEI16I64() then UNDEFINED;
2
   constant integer esize = 64;
  integer a = UInt(Pn);
3
4
   integer b = UInt(Pm);
5
   integer n = UInt(Zn);
6 integer m = UInt(Zm);
   integer da = UInt(ZAda);
7
8
   boolean sub_op = TRUE;
9
   boolean op1_unsigned = FALSE;
10 boolean op2_unsigned = FALSE;
   Assembler Symbols
```

- <ZAda> For the 32-bit variant: is the name of the ZA tile ZA0-ZA3, encoded in the "ZAda" field.
 - For the 64-bit variant: is the name of the ZA tile ZA0-ZA7, enclued in the Ada" field.
 - <Pn> Is the name of the first governing scalable predicate register P0-. encoded in t "Pn" field.
 - <Pm> Is the name of the second governing scalable predicate _gister P0-, encod in the "Pm" field.
 - <Zn> Is the name of the first source scalable vector regis. enc .ed in the "Zn" field.
 - <Zm> Is the name of the second source scalable y registe 'ncode' in the "Zm" field.

Operation

```
1
   CheckStreamingSVEAndZAEnabled();
2
   constant integer VL = CurrentVL;
3
   constant integer PL = VL DIV
   constant integer dim = VL D
                                 esize;
4
5
   bits(PL) mask1 = P[a, PL]
   bits(PL) mask2 = P[b, P'
6
   bits(VL) operand1 = Z[n, VL];
7
8 bits(VL) operand2 = [m, VL];
   bits(dim*dim*esize operand3 = ZAt >[da, esize, dim*dim*esize];
9
10
   bits(dim*dim*es' .) resu't;
11
   integer prod.
12
13
   for row = 0 to d_{\perp}
14
        for
                 0 to
                        m-1
15
            bits ( ize)
                              _lem[operand3, row*dim+col, esize];
                   = 0 to 3
16
            for }
17
                        wePredicateElement(mask1, 4*row + k, esize DIV 4) &&
18
                        ActivePredicateElement(mask2, 4*col + k, esize DIV 4) then
19
                    prod = (Int(Elem[operand1, 4*row + k, esize DIV 4], op1_unsigned) *
                            Int(Elem[operand2, 4*col + k, esize DIV 4], op2_unsigned));
20
21
                    if sub_op then prod = -prod;
22
                    sum = sum + prod;
23
24
            Elem[result, row*dim+col, esize] = sum;
25
26
   ZAtile[da, esize, dim*dim*esize] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
 - The values of the data supplied in any of its operand registers when its governing predicate registers contain the same value for each execution.
 - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:

- The values of the data supplied in any of its operand registers when its governing predicate registers contain the same value for each execution.
- The values of the NZCV flags.



D1.1.165 SQCVT (two registers)

Multi-vector signed saturating extract narrow

Saturate the signed integer value in each element of the two source vectors to half the orginal source element width, and place the results in the half-width destination elements.

This instruction is unpredicated.

SME2 (FEAT_SME2)



SQCVT <Zd>.H, { <Zn1>.S-<Zn2>.S }

```
1 if !HaveSME2() then UNDEFINED;
```

- 2 constant integer esize = 16;
- 3 integer n = UInt(Zn:'0');
- 4 integer d = UInt(Zd);

Assembler Symbols

- <Zd> Is the name of the destination scalab¹ vector revister, encoded in the "Zd" field.
- <Zn1> Is the name of the first scalable vector noise of a multi-vector sequence, encoded as "Zn" times 2.
- <Zn2> Is the name of the second scalable velow regner of a multi-vector sequence, encoded as "Zn" times 2 plus 1.

```
CheckStreamingSV _______...abled();
1
  constant integ VL = C lentVL;
2
  constant integ elem _s = VL DIV (2 * esize);
3
4
  bits(VL) result;
5
  6
7
8
9
10
          .em[result, r*elements + e, esize] = SignedSat(element, esize);
11
  Z[d, VL] = sult;
12
```

D1.1.166 SQCVT (four registers)

Multi-vector signed saturating extract narrow

Saturate the signed integer value in each element of the four source vectors to quarter the orginal source element width, and place the results in the quarter-width destination elements.

This instruction is unpredicated.

SME2 (FEAT_SME2)

 31
 24
 23
 22
 21

 1
 1
 0
 0
 0
 0
 1
 Sz
 0
 1
 1 0 0 1 1 1 1 1 0 0 Zn 0 7d 0 0 τυ N_

```
SQCVT <Zd>.<T>, { <Zn1>.<Tb>-<Zn4>.<Tb> }
```

```
1 if !HaveSME2() then UNDEFINED;
```

```
2 constant integer esize = 8 << UInt(sz);</pre>
```

```
3 integer n = UInt(Zn:'00');
```

```
4 integer d = UInt(Zd);
```

Assembler Symbols

SZ

1

- <Zd> Is the name of the destination scalab¹ vector revister, encoded in the "Zd" field.
- <T> Is the size specifier, encoded in "<z":

SZ	<t></t>
0	В
1	Н

- <Zn1> Is the name of t first scalable register of a multi-vector sequence, encoded as "Zn" times 4.
- <Tb> Is the siz pecifier needed in "sz":

<<u>Tb</u>

<Zn4> is a name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" time 4 plus 3.

```
CheckStreamingSVEEnabled();
1
2
   constant integer VL = CurrentVL;
3
   constant integer elements = VL DIV (4 * esize);
4
   bits(VL) result;
5
6
   for r = 0 to 3
       bits(VL) operand = Z[n+r, VL];
7
8
       for e = 0 to elements-1
9
           integer element = SInt(Elem[operand, e, 4 * esize]);
10
           Elem[result, r*elements + e, esize] = SignedSat(element, esize);
11
12 Z[d, VL] = result;
```

D1.1.167 SQCVTN

Multi-vector signed saturating extract narrow and interleave

Saturate the signed integer value in each element of the four source vectors to quarter the orginal source element width, and place the four-way interleaved results in the quarter-width destination elements.

This instruction is unpredicated.

SME2 (FEAT_SME2)

 31
 24
 23
 22
 21

 1
 1
 0
 0
 0
 0
 1
 SZ
 0
 1
 1 0 0 1 1 1 1 1 0 0 Zn 0 7d 0 1 τυ Ν_

```
SQCVTN <Zd>.<T>, { <Zn1>.<Tb>-<Zn4>.<Tb> }
```

```
1 if !HaveSME2() then UNDEFINED;
```

```
2 constant integer esize = 8 << UInt(sz);</pre>
```

```
3 integer n = UInt(Zn:'00');
```

```
4 integer d = UInt(Zd);
```

Assembler Symbols

SZ

1

- <Zd> Is the name of the destination scalab¹ vector revister, encoded in the "Zd" field.
- <T> Is the size specifier, encoded in "2":

SZ	<t></t>
0	В
1	Н

- <Zn1> Is the name of t first scalable register of a multi-vector sequence, encoded as "Zn" times 4.
- <Tb> Is the siz pecifier needed in "sz":

<<u>Tb</u>

<Zn4> is a name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.

```
CheckStreamingSVEEnabled();
1
2
    constant integer VL = CurrentVL;
3
    constant integer elements = VL DIV (4 * esize);
4
   bits(VL) result;
5
6
    for e = 0 to elements-1
        for i = 0 to 3
7
8
             bits(VL) operand = Z[n+i, VL];
             integer element = SInt(Elem[operand, e, 4 * esize]);
Elem[result, 4*e + i, esize] = SignedSat(element, esize);
9
10
11
12 Z[d, VL] = result;
```

D1.1.168 SQCVTU (two registers)

Multi-vector signed saturating unsigned extract narrow

Saturate the signed integer value in each element of the two source vectors to unsigned integer value that is half the orginal source element width, and place the results in the half-width destination elements.

This instruction is unpredicated.

SME2 (FEAT_SME2)



SQCVTU <Zd>.H, { <Zn1>.S-<Zn2>.S }

```
1 if !HaveSME2() then UNDEFINED;
```

- 2 constant integer esize = 16;
- 3 integer n = UInt(Zn:'0');
- 4 integer d = UInt(Zd);

Assembler Symbols

- <Zd> Is the name of the destination scalab¹ vector revister, encoded in the "Zd" field.
- <Zn1> Is the name of the first scalable vector noise of a multi-vector sequence, encoded as "Zn" times 2.
- <Zn2> Is the name of the second scalable velor register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.

```
CheckStreamingSV _______...abled();
      1
                          constant integ VL = C lentVL;
    2
                          constant integ elem _s = VL DIV (2 * esize);
    3
    4
                         bits(VL) result;
    5
      6
                                                           r > 0 to
' cs(VL) perand = .n+r, VL];
'sr e = _to_elements-1
ir .ger c____nt = SInt(Elem[operand, e, 2 * esize]);
' coult r+elements + e, esize] = UnsignedSat(elements);
' coult r+elements + e,
                          for r
                                                                                          J to
    7
    8
    9
 10
                                                                                                        .em[result, r*elements + e, esize] = UnsignedSat(element, esize);
 11
                   Z[d, VL] = sult;
12
```

D1.1.169 SQCVTU (four registers)

Multi-vector signed saturating unsigned extract narrow

Saturate the signed integer value in each element of the four source vectors to unsigned integer value that is quarter the orginal source element width, and place the results in the quarter-width destination elements.

This instruction is unpredicated.

SME2 (FEAT_SME2)

 31
 24
 23
 22
 21

 1
 1
 0
 0
 0
 0
 1
 SZ
 1
 1
 1 0 0 1 1 1 1 1 0 0 Zn 0 7d 0 0 Τu N_

```
SQCVTU <Zd>.<T>, { <Zn1>.<Tb>-<Zn4>.<Tb> }
```

```
1 if !HaveSME2() then UNDEFINED;
```

```
2 constant integer esize = 8 << UInt(sz);</pre>
```

```
3 integer n = UInt(Zn:'00');
```

```
4 integer d = UInt(Zd);
```

Assembler Symbols

SZ

1

- <Zd> Is the name of the destination scalab¹ vector revister, encoded in the "Zd" field.
- <T> Is the size specifier, encoded in "<z":

SZ	<t></t>
0	В
1	Н

- <Zn1> Is the name of t first scalable register of a multi-vector sequence, encoded as "Zn" times 4.
- <Tb> Is the siz pecifier needed in "sz":

<<u>Tb</u>

<Zn4> is a name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" time 4 plus 3.

```
CheckStreamingSVEEnabled();
1
2
   constant integer VL = CurrentVL;
3
   constant integer elements = VL DIV (4 * esize);
4
   bits(VL) result;
5
6
   for r = 0 to 3
       bits(VL) operand = Z[n+r, VL];
7
8
       for e = 0 to elements-1
9
           integer element = SInt(Elem[operand, e, 4 * esize]);
10
           Elem[result, r*elements + e, esize] = UnsignedSat(element, esize);
11
12 Z[d, VL] = result;
```

D1.1.170 SQCVTUN

2

3

Multi-vector signed saturating unsigned extract narrow and interleave

Saturate the signed integer value in each element of the four source vectors to unsigned integer value that is quarter the orginal source element width, and place the four-way interleaved results in the quarter-width destination elements.

This instruction is unpredicated.

SME2 (FEAT_SME2)



Is the name of the destination scalable ded in the "Zd" field. $\langle Zd \rangle$ ctor

<T> Is the size specifier, encoded in

SZ	<t></t>
0	В
1	Н

- the first scalable vec r register of a multi-vector sequence, encoded as "Zn" <Zn1> Is the name times 4.
- < Tb >y' *r*, encode _n "sz": Is the size $s_{\rm F}$



Is the number of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" <Zn4>times - lus 3.

```
CheckStreamingSVEEnabled();
1
2
   constant integer VL = CurrentVL;
3
   constant integer elements = VL DIV (4 * esize);
4
   bits(VL) result;
5
   for e = 0 to elements-1
6
       for i = 0 to 3
7
8
           bits(VL) operand = Z[n+i, VL];
9
           integer element = SInt(Elem[operand, e, 4 * esize]);
10
           Elem[result, 4*e + i, esize] = UnsignedSat(element, esize);
11
12 Z[d, VL] = result;
```

D1.1.171 SQDMULH (multiple and single vector)

Multi-vector signed saturating doubling multiply high by vector

Multiply then double the corresponding signed elements of the two or four first source vectors and the signed elements of the second source vector, and destructively place the most significant half of the result in the corresponding elements of the two or four first source vectors. Each result element is saturated to the N-bit element's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$.

This instruction is unpredicated.

It has encodings from 2 classes: Two registers and Four registers

Two registers (FEAT_SME2) 0 0 0 0 0 1 size 0 0 1 0 0 0 0 0 Zdn SQDMULH { <Zdn1>.<T>-<Zdn2>.<T> { <Zdn1> <Zdr }. if !HaveSME2() then UNDEFINED; constant integer esize = 8 << UInt(size);</pre> 2 integer dn = UInt(Zdn:'0'); 3 integer m = UInt('0':Zm); 4 5 constant integer nreg = 2; Four registers (FEAT_SME2) 0 0 0 0 0 0 0 1 0 1 0 1 0 0 1 0 1 .<T>dn4>.<T> }, { <Zdn1>.<T>-<Zdn4>.<T> }, <Zm>.<T> SQDMULH { <Zdr NDEFINE 1 if !HaveSME2() th 2 constan+ er es e = 8 UInt(size); integr dn = int (Zdı. 0.0 3 . : 4 int∈ ⊿r m = V nt('0':Zn , con. nt ir' 5 4;

Assemble Symbols

<Zdn1> For the vo registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2.

For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4.

<T> Is the size specifier, encoded in "size":

size	<t></t>
00	В
01	Н
10	S
11	D

<Zdn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4 plus 3.

- <Zdn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2 plus 1.
- <Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

```
1
   CheckStreamingSVEEnabled();
   constant integer VL = CurrentVL;
2
3
    constant integer elements = VL DIV esize;
4
    array [0..3] of bits(VL) results;
5
6
    for r = 0 to nreg-1
7
        bits(VL) operand1 = Z[dn+r, VL];
8
        bits(VL) operand2 = Z[m, VL];
9
        for e = 0 to elements-1
             integer element1 = SInt(Elem[operand1, e, esize]);
integer element2 = SInt(Elem[operand2, e, esize]);
10
11
12
             integer res = 2 * element1 * element2;
13
             Elem[results[r], e, esize] = SignedSat(res >> esiz.
                                                                         esize);
14
15
    for r = 0 to nreg-1
16
    Z[dn+r, VL] = results[r];
```

D1.1.172 SQDMULH (multiple vectors)

Multi-vector signed saturating doubling multiply high

Multiply then double the corresponding signed elements of the two or four first and second source vectors, and destructively place the most significant half of the result in the corresponding elements of the two or four first source vectors. Each result element is saturated to the N-bit element's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$.

This instruction is unpredicated.

It has encodings from 2 classes: Two registers and Four registers

Two registers (FEAT_SME2)



Assembles vmbols

<Zdn1> For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2.

For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4.

<T> Is the size specifier, encoded in "size":

size	<t></t>
00	В
01	Н
10	S
11	D

<Zdn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4 plus 3.

- <Zdn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2 plus 1.
- <Zm1> For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 2.

For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 4.

- <Zm4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zm" times 4 plus 3.
- <Zm2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zm" times 2 plus 1.

```
CheckStreamingSVEEnabled();
1
2
   constant integer VL = CurrentVL;
3
   constant integer elements = VL DIV esize;
4
   array [0..3] of bits(VL) results;
5
6
   for r = 0 to nreg-1
7
       bits(VL) operand1 = Z[dn+r, VL];
       bits(VL) operand2 = Z[m+r, VL];
8
9
       for e = 0 to elements-1
10
            integer element1 = SInt(Elem[ope .nd1,
                                                       esiz
            integer element2 = SInt(Elem[c rand2,
11
                                                       esize
12
            integer res = 2 * element1 *
                                           ement2
13
            Elem[results[r], e, esize]
                                       = S_
                                             ed
                                                        _____size, esize);
                                                  c(res
14
15
   for r = 0 to nreg-1
16
    Z[dn+r, VL] = results[r
```

D1.1.173 SQRSHR (two registers)

Multi-vector signed saturating rounding shift right narrow by immediate

Shift right by an immediate value, the signed integer value in each element of the two source vectors and place the rounded results in the half-width destination elements. Each result element is saturated to the half-width N-bit element's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$. The immediate shift amount is an unsigned value in the range 1 to 16.

This instruction is unpredicated.

SME2 (FEAT_SME2)



- <Zd> Is the name of the destination scale 'e vector 'egister, encoded in the "Zd" field.
- <Zn1> Is the name of the first stable vector sequence, encoded as "Zn" times 2.
- $\langle Zn2 \rangle$ Is the name of the second scall be vector register of a multi-vector sequence, encoded as "Zn" times 2r, s 1.
- <const> Is the imr diate shi' amount, in the range 1 to 16, encoded in the "imm4" field.

```
Check<sup>c</sup> _eamin VEEnaL
1
                            d
   cons int int or VL = C rentVL;
2
3
   con. nt ir
                        oments = VL DIV (2 * esize);
             sult;
4
   bits
5
   integer vund_const = 1 << (shift-1);</pre>
6
   for r = 0 t.
7
8
        bits(VL) operand = Z[n+r, VL];
        for e = 0 to elements-1
9
10
            bits(2 * esize) element = Elem[operand, e, 2 * esize];
11
            integer res = (SInt(element) + round_const) >> shift;
12
            Elem[result, r*elements + e, esize] = SignedSat(res, esize);
13
14
   Z[d, VL] = result;
```

D1.1.174 SQRSHR (four registers)

Multi-vector signed saturating rounding shift right narrow by immediate

Shift right by an immediate value, the signed integer value in each element of the four source vectors and place the rounded results in the quarter-width destination elements. Each result element is saturated to the quarter-width N-bit element's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$. The immediate shift amount is an unsigned value in the range 1 to number of bits per source element.

This instruction is unpredicated.

SME2 (FEAT_SME2)

1

2

9



Assembler Symbols

<Zd $>$	Is the name of the d	.cinatic	calable vec	etcr register,	encoded in the	"Zd" field.
---------	----------------------	----------	-------------	----------------	----------------	-------------

Is the size spec'er, encoded in <T> ze":

- tsize $\mathbf{f} \geq$ 00 SERVF 01 R
- <Zn1> Is the the first scalable vector register of a multi-vector sequence, encoded as "Zn" 4
- < Tb >Is the 'ize specifier, encoded in "tsize":

tsize	<tb></tb>
00	RESERVED
01	S
1x	D

- <Zn4>Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.
- <const> Is the immediate shift amount, in the range 1 to number of bits per source element, encoded in "tsize:imm5".

```
CheckStreamingSVEEnabled();
1
```

```
2
  constant integer VL = CurrentVL;
```

```
constant integer elements = VL DIV (4 * esize);
3
```

```
4 bits(VL) result;
5
   integer round_const = 1 << (shift-1);</pre>
6
7
   for r = 0 to 3
       bits(VL) operand = Z[n+r, VL];
8
9
       for e = 0 to elements-1
10
           bits(4 * esize) element = Elem[operand, e, 4 * esize];
            integer res = (SInt(element) + round_const) >> shift;
11
12
           Elem[result, r*elements + e, esize] = SignedSat(res, esize);
13
14 Z[d, VL] = result;
```

D1.1.175 SQRSHRN

2

7

9

Multi-vector signed saturating rounding shift right narrow by immediate and interleave

Shift right by an immediate value, the signed integer value in each element of the four source vectors and place the four-way interleaved rounded results in the quarter-width destination elements. Each result element is saturated to the quarter-width N-bit element's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$. The immediate shift amount is an unsigned value in the range 1 to number of bits per source element.

This instruction is unpredicated.

SME2 (FEAT_SME2)



Assembler Symbols

<Zd $>$	Is the name of the d	.cinatic	calable vec	etcr register,	encoded in the	"Zd" field.
---------	----------------------	----------	-------------	----------------	----------------	-------------

Is the size spec'er, encoded in <T> ze":

- tsize $\mathbf{f} \geq$ 00 SERVF 01 R
- <Zn1> Is the the first scalable vector register of a multi-vector sequence, encoded as "Zn" 4
- < Tb >Is the ize specifier, encoded in "tsize":

tsize	<tb></tb>
00	RESERVED
01	S
1x	D

- <Zn4>Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.
- <const> Is the immediate shift amount, in the range 1 to number of bits per source element, encoded in "tsize:imm5".

```
CheckStreamingSVEEnabled();
1
```

```
2
  constant integer VL = CurrentVL;
```

```
constant integer elements = VL DIV (4 * esize);
3
```

```
4 bits(VL) result;
5
   integer round_const = 1 << (shift-1);</pre>
6
7
    for e = 0 to elements-1
8
        for i = 0 to 3
            bits(VL) operand = Z[n+i, VL];
bits(4 * esize) element = Elem[operand, e, 4 * esize];
9
10
             integer res = (SInt(element) + round_const) >> shift;
11
12
             Elem[result, 4*e + i, esize] = SignedSat(res, esize);
13
14 Z[d, VL] = result;
```

D1.1.176 SQRSHRU (two registers)

Multi-vector signed saturating rounding shift right unsigned narrow by immediate

Shift right by an immediate value, the signed integer value in each element of the two source vectors and place the rounded results in the half-width destination elements. Each result element is saturated to the half-width N-bit element's unsigned integer range 0 to (2^N) -1. The immediate shift amount is an unsigned value in the range 1 to 16.

This instruction is unpredicated.

SME2 (FEAT_SME2)



- $\langle Zd \rangle$ Is the name of the destination sc 'able v. register, encoded in the "Zd" field.
- <Zn1> Is the name of the first scale, e vector register of a multi-vector sequence, encoded as "Zn" times 2.
- <Zn2> Is the name of the sound lable vector gister of a multi-vector sequence, encoded as "Zn" times 2 plus 1.
- <const> Is the immer '.e shift amount, in u range 1 to 16, encoded in the "imm4" field.

```
CheckStr TSVEL bled();
constation interver VL our ntVL;
1
2
3
    cons' nt inte ar eleme.
                                  = VL DIV (2 * esize);
4
    bit VL) res
    integ rr id_cons. 1 << (shift-1);</pre>
5
6
7
    for r = 0
                 o 1
        bits(VL operand = Z[n+r, VL];
for e = . to elements-1
8
9
10
             bits(2 * esize) element = Elem[operand, e, 2 * esize];
11
             integer res = (SInt(element) + round_const) >> shift;
12
             Elem[result, r*elements + e, esize] = UnsignedSat(res, esize);
13
14
   Z[d, VL] = result;
```

D1.1.177 SQRSHRU (four registers)

Multi-vector signed saturating rounding shift right unsigned narrow by immediate

Shift right by an immediate value, the signed integer value in each element of the four source vectors and place the rounded results in the quarter-width destination elements. Each result element is saturated to the quarter-width N-bit element's unsigned integer range 0 to (2^N) -1. The immediate shift amount is an unsigned value in the range 1 to number of bits per source element.

This instruction is unpredicated.

SME2 (FEAT_SME2)

1

2

4

5

6 7

9



Assembler Symbols

<Zd $>$	Is the name of the d	.cinatic	calable vec	etcr register,	encoded in the	"Zd" field.
---------	----------------------	----------	-------------	----------------	----------------	-------------

Is the size spec'er, encoded in <T> ze":

- tsize $\mathbf{f} \geq$ 00 SERVF 01 R
- <Zn1> Is the the first scalable vector register of a multi-vector sequence, encoded as "Zn" 4
- < Tb >Is the ize specifier, encoded in "tsize":

tsize	<tb></tb>
00	RESERVED
01	S
1x	D

- <Zn4>Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.
- <const> Is the immediate shift amount, in the range 1 to number of bits per source element, encoded in "tsize:imm5".

```
CheckStreamingSVEEnabled();
1
```

```
2
  constant integer VL = CurrentVL;
```

```
constant integer elements = VL DIV (4 * esize);
3
```

```
4 bits(VL) result;
5
   integer round_const = 1 << (shift-1);</pre>
6
7
   for r = 0 to 3
       bits(VL) operand = Z[n+r, VL];
8
9
       for e = 0 to elements-1
10
           bits(4 * esize) element = Elem[operand, e, 4 * esize];
            integer res = (SInt(element) + round_const) >> shift;
11
12
           Elem[result, r*elements + e, esize] = UnsignedSat(res, esize);
13
14 Z[d, VL] = result;
```

D1.1.178 SQRSHRUN

1

2

4

5

6 7

9

Multi-vector signed saturating rounding shift right unsigned narrow by immediate and interleave

Shift right by an immediate value, the signed integer value in each element of the four source vectors and place the four-way interleaved rounded results in the quarter-width destination elements. Each result element is saturated to the quarter-width N-bit element's unsigned integer range 0 to (2^N) -1. The immediate shift amount is an unsigned value in the range 1 to number of bits per source element.

This instruction is unpredicated.

SME2 (FEAT_SME2)



Assembler Symbols

<Zd $>$	Is the name of the d	.cinatic	calable vec	etcr register,	encoded in the	"Zd" field.
---------	----------------------	----------	-------------	----------------	----------------	-------------

Is the size spec'er, encoded in <T> ze":

- tsize $\mathbf{f} \geq$ 00 SERVF 01 R
- <Zn1> Is the the first scalable vector register of a multi-vector sequence, encoded as "Zn" 4
- < Tb >Is the ize specifier, encoded in "tsize":

tsize	< Tb >		
00	RESERVED		
01	S		
1x	D		

- <Zn4>Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.
- <const> Is the immediate shift amount, in the range 1 to number of bits per source element, encoded in "tsize:imm5".

```
CheckStreamingSVEEnabled();
1
```

```
2
  constant integer VL = CurrentVL;
```

```
constant integer elements = VL DIV (4 * esize);
3
```

```
4 bits(VL) result;
5
   integer round_const = 1 << (shift-1);</pre>
6
7
    for e = 0 to elements-1
8
        for i = 0 to 3
            bits(VL) operand = Z[n+i, VL];
bits(4 * esize) element = Elem[operand, e, 4 * esize];
9
10
             integer res = (SInt(element) + round_const) >> shift;
11
12
             Elem[result, 4*e + i, esize] = UnsignedSat(res, esize);
13
14 Z[d, VL] = result;
```

D1.1.179 SRSHL (multiple and single vector)

Multi-vector signed rounding shift left by vector

Shift active signed elements of the two or four first source vectors by corresponding elements of the second source vector and destructively place the rounded results in the corresponding elements of the two or four first source vectors. A positive shift amount performs a left shift, otherwise a right shift by the negated shift amount is performed.

This instruction is unpredicated.

It has encodings from 2 classes: Two registers and Four registers

```
Two registers
   (FEAT_SME2)
                 0
                    0
                       0
                          0
                             0
                               1
                                   size
                                        1
                                           0
                                                 Ζm
                                                           0
                                                              1
                                                                 0
                                                                       0
                                                                            0
                                                                               0
                                                                                           Zdr
                                                                                                   0
                                                                                                   SRSHL { <Zdn1>.<T>-<Zdn2>.<T> }, { <Zdn1>
                                                                                (7m) <T
                                                              <70
1
   if !HaveSME2() then UNDEFINED;
2
   constant integer esize = 8 << UInt(size)
3
   integer dn = UInt(Zdn:'0');
   integer m = UInt('0':Zm);
4
5
   constant integer nreg = 2;
   Four registers
   (FEAT SME2)
                                  s.
               1
                 0
                    0
                       0
                          0
                                1
                                       1
                                          0
                                                           0
                                                              1
                                                                 0
                                                                       0
                                                                            0
                                                                               0
                                                                                 0
                                                                                         7dn
                                                                                                  0
   SRSHL
            { < 7.dn
                              Zdn4>
                                             { <Zdn1>.<T>-<Zdn4>.<T> }, <Zm>.<T>
                                     \langle T \rangle \}.
1
   if !Hav
                    then 'DEF'
                                  ; <u>۵</u>.
2
           c inte r esiz
                                 << UInt(size);
   const
   int er dn = Int(Zdn:'\0');
3
4
   inte
            m
                          ~m);
               .teger nreg = 4;
5
   consta
```

Assembler **b**, **b**ols

<Zdn1> For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2.

For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4.

<T> Is the size specifier, encoded in "size":

<t></t>
В
Η
S
D

<Zdn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as

"Zdn" times 4 plus 3.

- <Zdn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2 plus 1.
 - <Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

```
1
   CheckStreamingSVEEnabled();
   constant integer VL = CurrentVL;
2
3
   constant integer elements = VL DIV esize;
4
   array [0..3] of bits(VL) results;
5
6
   for r = 0 to nreg-1
        bits(VL) operand1 = Z[dn+r, VL];
bits(VL) operand2 = Z[m, VL];
7
8
9
        for e = 0 to elements-1
10
            integer element = SInt(Elem[operand1, e, esize])
11
             integer shift = ShiftSat(SInt(Elem[operand2, e, es.
                                                                       ]), esize)
12
            integer res;
            if shift >= 0 then
13
14
                 res = element << shift;</pre>
15
             else
16
                 shift = -shift;
17
                 res = (element + (1 << (shift - 1)))
                                                                .ft;
18
            Elem[results[r], e, esize] = res
                                                         :0>;
19
20
   for r = 0 to nreg-1
21
        Z[dn+r, VL] = results[r];
```

D1.1.180 SRSHL (multiple vectors)

Multi-vector signed rounding shift left

Shift active signed elements of the two or four first source vectors by corresponding elements of the two or four second source vectors and destructively place the rounded results in the corresponding elements of the two or four first source vectors. A positive shift amount performs a left shift, otherwise a right shift by the negated shift amount is performed.

This instruction is unpredicated.

It has encodings from 2 classes: Two registers and Four registers



Assembler Symbols

<Zdn1> For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2.

For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4.

<T> Is the size specifier, encoded in "size":

size	< T >
00	В
01	Н
10	S
11	D

- <Zdn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4 plus 3.
- <Zdn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2 plus 1.
- <Zm1> For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 2.

For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 4.

- <Zm4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zm" times 4 plus 3.
- <Zm2> Is the name of the second scalable vector register of a multi-vector encoded as "Zm" times 2 plus 1.

```
1
   CheckStreamingSVEEnabled();
   constant integer VL = CurrentVL;
2
3
   constant integer elements = VL DIV esize;
4
   array [0..3] of bits(VL) results;
5
   for r = 0 to nreg-1
6
       bits(VL) operand1 = Z[dn+r, VL];
7
8
       bits(VL) operand2 = Z[m+r, VL];
9
       for e = 0 to elements-1
10
           integer element = SInt(Elem[op
                                          ar , e, es_ ]);
11
           12
           integer res;
13
           if shift >= 0 then
14
               res = element
                               shift;
15
           else
16
               shift = -sh.ft;
           res = (e oment + (1
Elem[resul [r], e, esize]
17
                                     (shift - 1))) >> shift;
18
                                      res<esize-1:0>;
19
   for r = 0 to r = -1
20
21
       Z[dn+r, VL]
                    res
                        cs[r];
```

D1.1.181 ST1B (scalar plus immediate, consecutive registers)

Contiguous store of bytes from multiple consecutive vectors (immediate index)

Contiguous store of bytes from elements of two or four consecutive vector registers to the memory address generated by a 64-bit scalar base and immediate index which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

Inactive elements are not written to memory.

It has encodings from 2 classes: Two registers and Four registers

Two registers (FEAT_SME2)



encoded as "Zt" times 2.

For the four registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 4.

- <Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" times 4 plus 3.
- <Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" times 2 plus 1.
- <PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" 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 two registers variant: is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.

For the four registers variant: is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.

```
CheckStreamingSVEEnabled();
1
2
   constant integer VL = CurrentVL;
   constant integer PL = VL DIV 8;
3
4
   constant integer elements = VL DIV esize;
5
   constant integer mbytes = esize DIV 8;
6 bits(64) base;
7
   bits(VL) src;
8
   bits(PL) pred = P[g, PL];
   bits(PL * nreg) mask = CounterToPredicate(pred<15:0>, PL *
9
                                                                   a);
10 boolean contiguous = TRUE;
11 boolean nontemporal = FALSE;
12 boolean tagchecked = n != 31;
   AccessDescriptor accdesc = CreateAccDescSVE (Mem. STORE
13
                                                              .ontemporai, contiguous,
        →tagchecked);
14
15
   if !AnyActiveElement(mask, esize) then
                                            Jool (Ur redicta.
                                                               CHECKSPNONEACTIVE) then
16
       if n == 31 && ConstrainUnpredictab'
17
           CheckSPAlignment();
18
   else
19
       if n == 31 then CheckSPAlignr t();
20
       base = if n == 31 then SP[1 el
                                         X[n,
21
22
   for r = 0 to nreg-1
23
       src = Z[t+r, VL];
24
       for e = 0 to elemer _-1
25
            if ActivePredicateEleme (mask, r * elements + e, esize) then
               bits(6^ addr = base (offset * nreg * elements + r * elements + e) * mbytes;
26
27
               Mem[/ ,r, mbytes, accde ] = Elem[src, e, esize];
```

D1.1.182 ST1B (scalar plus scalar, consecutive registers)

Contiguous store of bytes from multiple consecutive vectors (scalar index)

Contiguous store of bytes from elements of two or four consecutive vector registers to the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated.

Inactive elements are not written to memory.

It has encodings from 2 classes: Two registers and Four registers

Two registers (FEAT_SME2)



<Zt1> For the two registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 2.

For the four registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 4.

- <Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" times 4 plus 3.
- <Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" times 2 plus 1.
- <PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.

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

```
1
   CheckStreamingSVEEnabled();
   constant integer VL = CurrentVL;
2
3
   constant integer PL = VL DIV 8;
4
   constant integer elements = VL DIV esize;
   constant integer mbytes = esize DIV 8;
5
6
   bits(64) offset;
7
   bits(64) base;
   bits(VL) src;
8
9
   bits(PL) pred = P[g, PL];
10
   bits(PL * nreg) mask = CounterToPredicate(pred<15:0>, PL * nr
   boolean contiguous = TRUE;
11
12 boolean nontemporal = FALSE;
13
   boolean tagchecked = TRUE;
   AccessDescriptor accdesc = CreateAccDescSVE(MemOp_STO), non poral, c
14
                                                                             tiguous,
        →tagchecked);
15
16
   if !AnyActiveElement(mask, esize) then
17
        if n == 31 && ConstrainUnpredictableBool(Unp. 'ict
                                                            .re_CHECKSPNONEACTIVE) then
18
           CheckSPAlignment();
19
    else
20
       if n == 31 then CheckSPAlignment();
21
       base = if n == 31 then SP[] else
                                           n, 64];
22
       offset = X[m, 64];
23
24
   for r = 0 to nreg-1
25
        src = Z[t+r, VL];
        for e = 0 to elements-1
26
                                rment(mask,
27
            if ActivePredicat
                                               * elements + e, esize) then
28
                                  re + (UInt( fset) + r * elements + e) * mbytes;
                bits(64) ad
                                     rdesc] = Elem[src, e, esize];
29
                Mem[addr, m.Jytes,
```

D1.1.183 ST1B (scalar plus immediate, strided registers)

Contiguous store of bytes from multiple strided vectors (immediate index)

Contiguous store of bytes from elements of two or four strided vector registers to the memory address generated by a 64-bit scalar base and immediate index which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

Inactive elements are not written to memory.

It has encodings from 2 classes: Two registers and Four registers

Two registers (FEAT_SME2)



<Zt1> For the two registers variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as "T:'0':Zt".

For the four registers variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as "T:'00':Zt".

<Zt2> For the two registers variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as "T:'1':Zt".

For the four registers variant: is the name of the second scalable vector register Z4-Z7 or Z20-Z23 to be transferred, encoded as "T:'01':Zt".

- <Zt3> Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as "T:'10':Zt".
- <Zt4> Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as "T:'11':Zt".
- <PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" 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 two registers variant: is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.

For the four registers variant: is the optional signed immediate vector freet, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" fiet.

```
1
   CheckStreamingSVEEnabled();
   constant integer VL = CurrentVL;
2
3
   constant integer PL = VL DIV 8;
4
   constant integer elements = VL DIV esize;
5
   constant integer mbytes = esize DIV 8;
6 bits(64) base;
   bits(VL) src;
7
8
   bits(PL) pred = P[g, PL];
9
   bits(PL * nreg) mask = CounterToPred_ te(pre
                                                     ΡL
                                                        * nreg);
10 boolean contiguous = TRUE;
11
   boolean nontemporal = FALSE;
12 boolean tagchecked = n != 31;
13
   AccessDescriptor accdesc = C .ateAccL
                                       SVE(1. Op_STORE, nontemporal, contiguous,
       \hookrightarrowtagchecked);
14
15
   if !AnyActiveElement(ma k, esi ) then
       if n == 31 && ConstrainUnpre
CheckSPAlinent();
16
                                  ctableBool(Unpredictable_CHECKSPNONEACTIVE) then
17
18
   else
       if n == 31 nen Che .SPAlignment();
19
20
       base = if 1 = 31 nen SP[] else X[n, 64];
21
22
   for r =
               rea-
          = Z[t VL];
23
       sr
24
        \mathbf{r} = 0 to element -1
25
          if
                26
27
              Mem[addr, mbytes, accdesc] = Elem[src, e, esize];
          t
28
       t =
              tstride;
```
D1.1.184 ST1B (scalar plus scalar, strided registers)

Contiguous store of bytes from multiple strided vectors (scalar index)

Contiguous store of bytes from elements of two or four strided vector registers to the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated.

Inactive elements are not written to memory.

It has encodings from 2 classes: Two registers and Four registers

Two registers (FEAT_SME2)



<Zt1> For the two registers variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as "T:'0':Zt".

For the four registers variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as "T:'00':Zt".

<Zt2> For the two registers variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as "T:'1':Zt".

For the four registers variant: is the name of the second scalable vector register Z4-Z7 or Z20-Z23 to be transferred, encoded as "T:'01':Zt".

- <Zt3> Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as "T:'10':Zt".
- <Zt4> Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as "T:'11':Zt".
- <PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
 - <Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

```
1 CheckStreamingSVEEnabled();
2
   constant integer VL = CurrentVL;
   constant integer PL = VL DIV 8;
3
   constant integer elements = VL DIV esize;
4
5
   constant integer mbytes = esize DIV 8;
6
   bits(64) offset;
   bits(64) base;
7
8 bits(VL) src;
9
   bits(PL) pred = P[g, PL];
                                                           PL * nr
10
   bits(PL * nreg) mask = CounterToPredicate(r
                                                   <15:0.
   boolean contiguous = TRUE;
11
12 boolean nontemporal = FALSE;
13
   boolean tagchecked = TRUE;
14
   AccessDescriptor accdesc = CreateAccDe SVE/
                                                              nontemporal, contiguous,
        →tagchecked);
15
16
   if !AnyActiveElement(mask, e le) th
        if n == 31 && Constrair predictab Bool(L predictable_CHECKSPNONEACTIVE) then
17
18
            CheckSPAlignment/
19
   else
20
        if n == 31 then CheckSPAlig ont();
       base = if n == then SP[] & X[n, 64];
offset = X[m 4];
21
22
23
24
   for r = 0 to 1. q-1
        src = Z[t,
25
26
        for
                 tοε
                        ments-1
            e
27
            .r A ivePre cate lement(mask, r * elements + e, esize) then
28
                b .s(64)
                              = base + (UInt(offset) + r * elements + e) * mbytes;
29
                 _m[addr, muytes, accdesc] = Elem[src, e, esize];
30
```

D1.1.185 ST1B (scalar plus scalar, tile slice)

Contiguous store of bytes from 8-bit element ZA tile slice

The slice number within the tile is selected by the sum of the slice index register and immediate offset, modulo the number of 8-bit elements in a vector. The immediate offset is in the range 0 to 15. The memory address is generated by a 64-bit scalar base and an optional 64-bit scalar offset which is added to the base address. Inactive elements are not written to memory.

SME (FEAT_SME)



```
2
   constant integer VL = CurrentVL;
3
   constant integer PL = VL DIV 8;
4
   constant integer dim = VL DIV esize;
5
   bits(64) base;
6
  bits(64) addr;
7
   bits(PL) mask = P[g, PL];
8
   bits(64) moffs = X[m, 64];
   bits(32) index = X[s, 32];
9
10
   integer slice = (UInt(index) + offset) MOD dim;
11
   bits(VL) src;
   constant integer mbytes = esize DIV 8;
12
13 boolean contiguous = TRUE;
14
   boolean nontemporal = FALSE;
```

```
15 boolean tagchecked = TRUE;
16
   AccessDescriptor accdesc = CreateAccDescSME(MemOp_STORE, nontemporal, contiguous,
         \hookrightarrowtagchecked);
17
    if n == 31 then
18
19
        if AnyActiveElement(mask, esize) ||
              ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
20
21
             CheckSPAlignment();
22
        base = SP[];
23
    else
24
        base = X[n, 64];
25
26
    src = ZAslice[t, esize, vertical, slice, VL];
27
    for e = 0 to dim-1
28
        addr = base + UInt(moffs) * mbytes;
        if ActivePredicateElement(mask, e, esize) then
    Mem[addr, mbytes, accdesc] = Elem[src, e, esize];
29
30
31
        moffs = moffs + 1;
```

D1.1.186 ST1D (scalar plus immediate, consecutive registers)

Contiguous store of doublewords from multiple consecutive vectors (immediate index)

Contiguous store of doublewords from elements of two or four consecutive vector registers to the memory address generated by a 64-bit scalar base and immediate index which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

Inactive elements are not written to memory.

It has encodings from 2 classes: Two registers and Four registers

Two registers (FEAT_SME2)



encoded as "Zt" times 2.

For the four registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 4.

- <Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" times 4 plus 3.
- <Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" times 2 plus 1.
- <PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" 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 two registers variant: is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.

For the four registers variant: is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.

```
1
   CheckStreamingSVEEnabled();
2
   constant integer VL = CurrentVL;
   constant integer PL = VL DIV 8;
3
4
   constant integer elements = VL DIV esize;
5
   constant integer mbytes = esize DIV 8;
6 bits(64) base;
7
   bits(VL) src;
8
   bits(PL) pred = P[g, PL];
   bits(PL * nreg) mask = CounterToPredicate(pred<15:0>, PL *
9
                                                                   a);
10 boolean contiguous = TRUE;
11 boolean nontemporal = FALSE;
12 boolean tagchecked = n != 31;
   AccessDescriptor accdesc = CreateAccDescSVE (Mem. STORE
13
                                                              .ontemporai, contiguous,
        →tagchecked);
14
15
   if !AnyActiveElement(mask, esize) then
                                            Jool (Ur redicta.
                                                               CHECKSPNONEACTIVE) then
16
       if n == 31 && ConstrainUnpredictab'
17
           CheckSPAlignment();
18
   else
19
       if n == 31 then CheckSPAlignr t();
20
       base = if n == 31 then SP[1 el
                                         X[n,
21
22
   for r = 0 to nreg-1
23
       src = Z[t+r, VL];
24
       for e = 0 to elemer _-1
25
            if ActivePredicateEleme (mask, r * elements + e, esize) then
               bits(6^ addr = base (offset * nreg * elements + r * elements + e) * mbytes;
26
27
               Mem[/ ,r, mbytes, accde ] = Elem[src, e, esize];
```

D1.1.187 ST1D (scalar plus scalar, consecutive registers)

Contiguous store of doublewords from multiple consecutive vectors (scalar index)

Contiguous store of doublewords from elements of two or four consecutive vector registers to the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated.

Inactive elements are not written to memory.

It has encodings from 2 classes: Two registers and Four registers

Two registers (FEAT_SME2)



encoded as "Zt" times 2.

For the four registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 4.

- <Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" times 4 plus 3.
- <Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" times 2 plus 1.
- <PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.

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

```
1
   CheckStreamingSVEEnabled();
   constant integer VL = CurrentVL;
2
3
   constant integer PL = VL DIV 8;
4
   constant integer elements = VL DIV esize;
   constant integer mbytes = esize DIV 8;
5
6
   bits(64) offset;
7
   bits(64) base;
   bits(VL) src;
8
9
   bits(PL) pred = P[g, PL];
10
   bits(PL * nreg) mask = CounterToPredicate(pred<15:0>, PL * nr
   boolean contiguous = TRUE;
11
12 boolean nontemporal = FALSE;
13
   boolean tagchecked = TRUE;
   AccessDescriptor accdesc = CreateAccDescSVE(MemOp_STO), non poral, c
14
                                                                              tiguous,
        \hookrightarrowtagchecked);
15
16
   if !AnyActiveElement(mask, esize) then
17
        if n == 31 && ConstrainUnpredictableBool(Unp. 'ict
                                                            .re_CHECKSPNONEACTIVE) then
18
            CheckSPAlignment();
19
    else
20
       if n == 31 then CheckSPAlignment();
21
       base = if n == 31 then SP[] else
                                           n, 64];
22
       offset = X[m, 64];
23
24
   for r = 0 to nreg-1
25
        src = Z[t+r, VL];
        for e = 0 to elements-1
26
                                rment(mask,
27
            if ActivePredicat
                                               * elements + e, esize) then
28
                                   re + (UInt( fset) + r * elements + e) * mbytes;
                bits(64) ad
                                     rdesc] = Elem[src, e, esize];
29
                Mem[addr, m.Jytes,
```

D1.1.188 ST1D (scalar plus immediate, strided registers)

Contiguous store of doublewords from multiple strided vectors (immediate index)

Contiguous store of doublewords from elements of two or four strided vector registers to the memory address generated by a 64-bit scalar base and immediate index which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

Inactive elements are not written to memory.

It has encodings from 2 classes: Two registers and Four registers

Two registers (FEAT_SME2)



Assembler Symbols

<Zt1> For the two registers variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as "T:'0':Zt".

For the four registers variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as "T:'00':Zt".

<Zt2> For the two registers variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as "T:'1':Zt".

For the four registers variant: is the name of the second scalable vector register Z4-Z7 or Z20-Z23 to be transferred, encoded as "T:'01':Zt".

- <Zt3> Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as "T:'10':Zt".
- <Zt4> Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as "T:'11':Zt".
- <PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" 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 two registers variant: is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.

For the four registers variant: is the optional signed immediate vector freet, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" fiet.

```
1
   CheckStreamingSVEEnabled();
   constant integer VL = CurrentVL;
2
3
   constant integer PL = VL DIV 8;
4
   constant integer elements = VL DIV esize;
5
   constant integer mbytes = esize DIV 8;
6
   bits(64) base;
   bits(VL) src;
7
8
   bits(PL) pred = P[g, PL];
9
   bits(PL * nreg) mask = CounterToPred_ te(pre
                                                        ΡL
                                                           * nreg);
10 boolean contiguous = TRUE;
11
   boolean nontemporal = FALSE;
12 boolean tagchecked = n != 31;
13
   AccessDescriptor accdesc = C .ateAccL
                                          SVE(1. Op_STORE, nontemporal, contiguous,
        \hookrightarrowtagchecked);
14
15
   if !AnyActiveElement(ma k, esi ) then
       if n == 31 && ConstrainUnpre
CheckSPAlinent();
16
                                     ctableBool(Unpredictable_CHECKSPNONEACTIVE) then
17
18
   else
       if n == 31 nen Che .SPAlignment();
19
20
       base = if 1 = 31 nen SP[] else X[n, 64];
21
22
   for r =
                rea-
          = Z[t VL];
23
       sr
24
         \mathbf{r} = 0 to element -1
25
           if
                    PredicateElement(mask, r * elements + e, esize) then
               26
27
               Mem[addr, mbytes, accdesc] = Elem[src, e, esize];
           t
28
       t =
               tstride;
```

D1.1.189 ST1D (scalar plus scalar, strided registers)

Contiguous store of doublewords from multiple strided vectors (scalar index)

Contiguous store of doublewords from elements of two or four strided vector registers to the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated.

Inactive elements are not written to memory.

It has encodings from 2 classes: Two registers and Four registers

Two registers (FEAT_SME2)



<Zt1> For the two registers variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as "T:'0':Zt".

For the four registers variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as "T:'00':Zt".

<Zt2> For the two registers variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as "T:'1':Zt".

For the four registers variant: is the name of the second scalable vector register Z4-Z7 or Z20-Z23 to be transferred, encoded as "T:'01':Zt".

- <Zt3> Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as "T:'10':Zt".
- <Zt4> Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as "T:'11':Zt".
- <PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
 - <Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

```
1 CheckStreamingSVEEnabled();
2
   constant integer VL = CurrentVL;
   constant integer PL = VL DIV 8;
3
   constant integer elements = VL DIV esize;
4
5
   constant integer mbytes = esize DIV 8;
6
   bits(64) offset;
   bits(64) base;
7
8 bits(VL) src;
9
   bits(PL) pred = P[g, PL];
                                                           PL * nr
10
   bits(PL * nreg) mask = CounterToPredicate(r
                                                   <15:0.
   boolean contiguous = TRUE;
11
12 boolean nontemporal = FALSE;
13
   boolean tagchecked = TRUE;
14
   AccessDescriptor accdesc = CreateAccDe SVE/
                                                              nontemporal, contiguous,
        →tagchecked);
15
16
   if !AnyActiveElement(mask, e le) th
        if n == 31 && Constrair predictab Bool(L predictable_CHECKSPNONEACTIVE) then
17
18
            CheckSPAlignment/
19
   else
20
        if n == 31 then CheckSPAlig ont();
       base = if n == then SP[] e > X[n, 64];
offset = X[m _4];
21
22
23
24
   for r = 0 to 1. q-1
        src = Z[t,
25
26
        for
                 tοε
                        ments-1
            e
27
            .r A ivePre cate lement(mask, r * elements + e, esize) then
28
                b .s(64)
                              = base + (UInt(offset) + r * elements + e) * mbytes;
29
                  _m[addr, muytes, accdesc] = Elem[src, e, esize];
30
```

D1.1.190 ST1D (scalar plus scalar, tile slice)

Contiguous store of doublewords from 64-bit element ZA tile slice

The slice number within the tile is selected by the sum of the slice index register and immediate offset, modulo the number of 64-bit elements in a vector. The immediate offset is in the range 0 to 1. The memory address is generated by a 64-bit scalar base and an optional 64-bit scalar offset which is multiplied by 8 and added to the base address. Inactive elements are not written to memory.

SME (FEAT_SME)



```
6 bits(64) addr;
7 bits(PL) mask = P[g, PL];
8 bits(64) moffs = X[m, 64];
9 bits(32) index = X[s, 32];
10 integer slice = (UInt(index) + offset) MOD dim;
11 bits(VL) src;
12 constant integer mbytes = esize DIV 8;
```

```
13 boolean contiguous = TRUE;
14 boolean nontemporal = FALSE;
15
    boolean tagchecked = TRUE;
    AccessDescriptor accdesc = CreateAccDescSME(MemOp_STORE, nontemporal, contiguous,
16
         \hookrightarrowtagchecked);
17
    if n == 31 then
18
19
        if AnyActiveElement(mask, esize) ||
20
               ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
21
             CheckSPAlignment();
22
        base = SP[];
23
    else
24
        base = X[n, 64];
25
    src = ZAslice[t, esize, vertical, slice, VL];
26
27
    for e = 0 to dim-1
        addr = base + UInt(moffs) * mbytes;
28
29
        \quad \text{if } \texttt{ActivePredicateElement} \ (\texttt{mask}, \ \texttt{e}, \ \texttt{esize}) \ \text{then} \\
30
            Mem[addr, mbytes, accdesc] = Elem[src, e, esize];
31
        moffs = moffs + 1;
```

D1.1.191 ST1H (scalar plus immediate, consecutive registers)

Contiguous store of halfwords from multiple consecutive vectors (immediate index)

Contiguous store of halfwords from elements of two or four consecutive vector registers to the memory address generated by a 64-bit scalar base and immediate index which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

Inactive elements are not written to memory.

It has encodings from 2 classes: Two registers and Four registers

Two registers (FEAT_SME2)



- - encoded as "Zt" times 2.

For the four registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 4.

- $\langle Zt4 \rangle$ Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" times 4 plus 3.
- Is the name of the second scalable vector register to be transferred, encoded as "Zt" times 2 <Zt2> plus 1.
- Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter <PNg> encoding, encoded in the "PNg" 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 two registers variant: is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.

For the four registers variant: is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.

```
CheckStreamingSVEEnabled();
1
2
   constant integer VL = CurrentVL;
   constant integer PL = VL DIV 8;
3
4
   constant integer elements = VL DIV esize;
5
   constant integer mbytes = esize DIV 8;
6 bits(64) base;
7
   bits(VL) src;
8
   bits(PL) pred = P[g, PL];
   bits(PL * nreg) mask = CounterToPredicate(pred<15:0>, PL *
9
                                                                   a);
10 boolean contiguous = TRUE;
11 boolean nontemporal = FALSE;
12 boolean tagchecked = n != 31;
   AccessDescriptor accdesc = CreateAccDescSVE (Mem. STORE
13
                                                              .ontemporai, contiguous,
        →tagchecked);
14
15
   if !AnyActiveElement(mask, esize) then
                                            Jool (Ur redicta.
                                                               CHECKSPNONEACTIVE) then
16
       if n == 31 && ConstrainUnpredictab'
17
           CheckSPAlignment();
18
   else
19
       if n == 31 then CheckSPAlignr t();
20
       base = if n == 31 then SP[1 el
                                         X[n,
21
22
   for r = 0 to nreg-1
23
       src = Z[t+r, VL];
24
       for e = 0 to elemer _-1
25
            if ActivePredicateEleme (mask, r * elements + e, esize) then
               bits(6^ addr = base (offset * nreg * elements + r * elements + e) * mbytes;
26
27
               Mem[/ ,r, mbytes, accde ] = Elem[src, e, esize];
```

D1.1.192 ST1H (scalar plus scalar, consecutive registers)

Contiguous store of halfwords from multiple consecutive vectors (scalar index)

Contiguous store of halfwords from elements of two or four consecutive vector registers to the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated.

Inactive elements are not written to memory.

It has encodings from 2 classes: Two registers and Four registers

Two registers (FEAT_SME2)



encoded as "Zt" times 2.

For the four registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 4.

- <Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" times 4 plus 3.
- <Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" times 2 plus 1.
- <PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.

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

```
1
   CheckStreamingSVEEnabled();
   constant integer VL = CurrentVL;
2
3
   constant integer PL = VL DIV 8;
4
   constant integer elements = VL DIV esize;
   constant integer mbytes = esize DIV 8;
5
6
   bits(64) offset;
7
   bits(64) base;
   bits(VL) src;
8
9
   bits(PL) pred = P[g, PL];
10
   bits(PL * nreg) mask = CounterToPredicate(pred<15:0>, PL * nr
   boolean contiguous = TRUE;
11
12 boolean nontemporal = FALSE;
13
   boolean tagchecked = TRUE;
   AccessDescriptor accdesc = CreateAccDescSVE(MemOp_STO), non poral, c
14
                                                                             tiguous,
        →tagchecked);
15
16
   if !AnyActiveElement(mask, esize) then
17
        if n == 31 && ConstrainUnpredictableBool(Unp. 'ict
                                                            .re_CHECKSPNONEACTIVE) then
18
           CheckSPAlignment();
19
    else
20
       if n == 31 then CheckSPAlignment();
21
       base = if n == 31 then SP[] else
                                           n, 64];
22
       offset = X[m, 64];
23
24
   for r = 0 to nreg-1
25
        src = Z[t+r, VL];
        for e = 0 to elements-1
26
                                rment(mask,
27
            if ActivePredicat
                                               * elements + e, esize) then
28
                                  re + (UInt( fset) + r * elements + e) * mbytes;
                bits(64) ad
                                     rdesc] = Elem[src, e, esize];
29
                Mem[addr, m.Jytes,
```

D1.1.193 ST1H (scalar plus immediate, strided registers)

Contiguous store of halfwords from multiple strided vectors (immediate index)

Contiguous store of halfwords from elements of two or four strided vector registers to the memory address generated by a 64-bit scalar base and immediate index which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

Inactive elements are not written to memory.

It has encodings from 2 classes: Two registers and Four registers

Two registers (FEAT_SME2)



<Zt1> For the two registers variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as "T:'0':Zt".

For the four registers variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as "T:'00':Zt".

<Zt2> For the two registers variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as "T:'1':Zt".

For the four registers variant: is the name of the second scalable vector register Z4-Z7 or Z20-Z23 to be transferred, encoded as "T:'01':Zt".

- <Zt3> Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as "T:'10':Zt".
- <Zt4> Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as "T:'11':Zt".
- <PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" 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 two registers variant: is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.

For the four registers variant: is the optional signed immediate vector freet, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" fiet.

```
1
   CheckStreamingSVEEnabled();
   constant integer VL = CurrentVL;
2
3
   constant integer PL = VL DIV 8;
4
   constant integer elements = VL DIV esize;
5
   constant integer mbytes = esize DIV 8;
6 bits(64) base;
   bits(VL) src;
7
8
   bits(PL) pred = P[g, PL];
9
   bits(PL * nreg) mask = CounterToPred_ te(pre
                                                     ΡL
                                                        * nreg);
10 boolean contiguous = TRUE;
11
   boolean nontemporal = FALSE;
12 boolean tagchecked = n != 31;
13
   AccessDescriptor accdesc = C .ateAccL
                                       SVE(1. Op_STORE, nontemporal, contiguous,
       \hookrightarrowtagchecked);
14
15
   if !AnyActiveElement(ma k, esi ) then
       if n == 31 && ConstrainUnpre
CheckSPAlinent();
16
                                  ctableBool(Unpredictable_CHECKSPNONEACTIVE) then
17
18
   else
       if n == 31 nen Che .SPAlignment();
19
20
       base = if 1 = 31 nen SP[] else X[n, 64];
21
22
   for r =
               rea-
          = Z[t VL];
23
       sr
24
        \mathbf{r} = 0 to element -1
25
          if
                26
27
              Mem[addr, mbytes, accdesc] = Elem[src, e, esize];
          t
28
       t =
              tstride;
```

D1.1.194 ST1H (scalar plus scalar, strided registers)

Contiguous store of halfwords from multiple strided vectors (scalar index)

Contiguous store of halfwords from elements of two or four strided vector registers to the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated.

Inactive elements are not written to memory.

It has encodings from 2 classes: Two registers and Four registers

Two registers (FEAT_SME2)



<Zt1> For the two registers variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as "T:'0':Zt".

For the four registers variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as "T:'00':Zt".

<Zt2> For the two registers variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as "T:'1':Zt".

For the four registers variant: is the name of the second scalable vector register Z4-Z7 or Z20-Z23 to be transferred, encoded as "T:'01':Zt".

- <Zt3> Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as "T:'10':Zt".
- <Zt4> Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as "T:'11':Zt".
- <PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
 - <Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

```
1 CheckStreamingSVEEnabled();
2
   constant integer VL = CurrentVL;
   constant integer PL = VL DIV 8;
3
   constant integer elements = VL DIV esize;
4
5
   constant integer mbytes = esize DIV 8;
6
   bits(64) offset;
   bits(64) base;
7
8 bits(VL) src;
9
   bits(PL) pred = P[g, PL];
                                                           PL * nr
10
   bits(PL * nreg) mask = CounterToPredicate(r
                                                   <15:0.
   boolean contiguous = TRUE;
11
12 boolean nontemporal = FALSE;
13
   boolean tagchecked = TRUE;
14
   AccessDescriptor accdesc = CreateAccDe SVE/
                                                              nontemporal, contiguous,
        →tagchecked);
15
16
   if !AnyActiveElement(mask, e le) th
        if n == 31 && Constrair predictab Bool(L predictable_CHECKSPNONEACTIVE) then
17
18
            CheckSPAlignment/
19
   else
20
        if n == 31 then CheckSPAlig ont();
       base = if n == then SP[] e > X[n, 64];
offset = X[m _4];
21
22
23
24
   for r = 0 to 1. q-1
        src = Z[t,
25
26
        for
                 tοε
                        ments-1
            е
27
            .r A ivePre cate lement(mask, r * elements + e, esize) then
28
                b .s(64)
                              = base + (UInt(offset) + r * elements + e) * mbytes;
29
                  _m[addr, muytes, accdesc] = Elem[src, e, esize];
30
```

D1.1.195 ST1H (scalar plus scalar, tile slice)

Contiguous store of halfwords from 16-bit element ZA tile slice

The slice number within the tile is selected by the sum of the slice index register and immediate offset, modulo the number of 16-bit elements in a vector. The immediate offset is in the range 0 to 7. The memory address is generated by a 64-bit scalar base and an optional 64-bit scalar offset which is multiplied by 2 and added to the base address. Inactive elements are not written to memory.

SME (FEAT_SME)



```
5 bits(64) base;
6 bits(64) addr;
7 bits(PL) mask = P[g, PL];
8 bits(64) moffs = X[m, 64];
9 bits(32) index = X[s, 32];
10 integer slice = (UInt(index) + offset) MOD dim;
11 bits(VL) src;
12 constant integer mbytes = esize DIV 8;
```

```
13 boolean contiguous = TRUE;
14 boolean nontemporal = FALSE;
15
    boolean tagchecked = TRUE;
    AccessDescriptor accdesc = CreateAccDescSME(MemOp_STORE, nontemporal, contiguous,
16
         \hookrightarrowtagchecked);
17
    if n == 31 then
18
19
        if AnyActiveElement(mask, esize) ||
20
               ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
21
             CheckSPAlignment();
22
        base = SP[];
23
    else
24
        base = X[n, 64];
25
    src = ZAslice[t, esize, vertical, slice, VL];
26
27
    for e = 0 to dim-1
        addr = base + UInt(moffs) * mbytes;
28
29
        \quad \text{if } \texttt{ActivePredicateElement} \ (\texttt{mask}, \ \texttt{e}, \ \texttt{esize}) \ \text{then} \\
30
            Mem[addr, mbytes, accdesc] = Elem[src, e, esize];
31
        moffs = moffs + 1;
```

D1.1.196 ST1Q

Contiguous store of quadwords from 128-bit element ZA tile slice

The slice number in the tile is selected by the slice index register, modulo the number of 128-bit elements in a Streaming SVE vector. The memory address is generated by scalar base and optional scalar offset which is multiplied by 16 and added to the base address. Inactive elements are not written to memory.

SME (FEAT_SME)



constant integer VL = CurrentVL; 2 3 constant integer PL = VL DIV 8; 4 constant **integer** dim = VL DIV esize; bits(64) base; 5 6 bits(64) addr; bits(PL) mask = P[g, PL]; 7 8 bits(64) moffs = X[m, 64]; **bits**(32) index = X[s, 32]; 9 10 integer slice = (UInt(index) + offset) MOD dim; 11 bits(VL) src; 12 constant **integer** mbytes = esize DIV 8; boolean contiguous = TRUE; 13 14 boolean nontemporal = FALSE;

```
15 boolean tagchecked = TRUE;
16
   AccessDescriptor accdesc = CreateAccDescSME(MemOp_STORE, nontemporal, contiguous,
         \hookrightarrowtagchecked);
17
    if n == 31 then
18
19
        if AnyActiveElement(mask, esize) ||
              ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
20
21
             CheckSPAlignment();
22
        base = SP[];
23
    else
24
        base = X[n, 64];
25
26
   src = ZAslice[t, esize, vertical, slice, VL];
27
    for e = 0 to dim-1
28
        addr = base + UInt(moffs) * mbytes;
        if ActivePredicateElement(mask, e, esize) then
    Mem[addr, mbytes, accdesc] = Elem[src, e, esize];
29
30
31
        moffs = moffs + 1;
```

D1.1.197 ST1W (scalar plus immediate, consecutive registers)

Contiguous store of words from multiple consecutive vectors (immediate index)

Contiguous store of words from elements of two or four consecutive vector registers to the memory address generated by a 64-bit scalar base and immediate index which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

Inactive elements are not written to memory.

It has encodings from 2 classes: Two registers and Four registers

Two registers (FEAT_SME2)



encoded as "Zt" times 2.

For the four registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 4.

- <Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" times 4 plus 3.
- <Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" times 2 plus 1.
- <PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" 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 two registers variant: is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.

For the four registers variant: is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.

```
CheckStreamingSVEEnabled();
1
2
   constant integer VL = CurrentVL;
   constant integer PL = VL DIV 8;
3
4
   constant integer elements = VL DIV esize;
5
   constant integer mbytes = esize DIV 8;
6 bits(64) base;
7
   bits(VL) src;
8
   bits(PL) pred = P[g, PL];
   bits(PL * nreg) mask = CounterToPredicate(pred<15:0>, PL *
9
                                                                   a);
10 boolean contiguous = TRUE;
11 boolean nontemporal = FALSE;
12 boolean tagchecked = n != 31;
   AccessDescriptor accdesc = CreateAccDescSVE (Mem. STORE
13
                                                              .ontemporai, contiguous,
        →tagchecked);
14
15
   if !AnyActiveElement(mask, esize) then
                                            Jool (Ur redicta.
                                                               CHECKSPNONEACTIVE) then
16
       if n == 31 && ConstrainUnpredictab'
17
           CheckSPAlignment();
18
   else
19
       if n == 31 then CheckSPAlignr t();
20
       base = if n == 31 then SP[1 el
                                         X[n,
21
22
   for r = 0 to nreg-1
23
       src = Z[t+r, VL];
24
       for e = 0 to elemer _-1
25
            if ActivePredicateEleme (mask, r * elements + e, esize) then
               bits(6^ addr = base (offset * nreg * elements + r * elements + e) * mbytes;
26
27
               Mem[/ ,r, mbytes, accde ] = Elem[src, e, esize];
```

D1.1.198 ST1W (scalar plus scalar, consecutive registers)

Contiguous store of words from multiple consecutive vectors (scalar index)

Contiguous store of words from elements of two or four consecutive vector registers to the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated.

Inactive elements are not written to memory.

It has encodings from 2 classes: Two registers and Four registers

Two registers (FEAT_SME2)



encoded as "Zt" times 2.

For the four registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 4.

- <Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" times 4 plus 3.
- <Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" times 2 plus 1.
- <PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.

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

```
1
   CheckStreamingSVEEnabled();
   constant integer VL = CurrentVL;
2
3
   constant integer PL = VL DIV 8;
4
   constant integer elements = VL DIV esize;
   constant integer mbytes = esize DIV 8;
5
6
   bits(64) offset;
7
   bits(64) base;
   bits(VL) src;
8
9
   bits(PL) pred = P[g, PL];
10
   bits(PL * nreg) mask = CounterToPredicate(pred<15:0>, PL * nr
   boolean contiguous = TRUE;
11
12 boolean nontemporal = FALSE;
13
   boolean tagchecked = TRUE;
   AccessDescriptor accdesc = CreateAccDescSVE(MemOp_STO), non poral, c
14
                                                                             tiguous,
        →tagchecked);
15
16
   if !AnyActiveElement(mask, esize) then
17
        if n == 31 && ConstrainUnpredictableBool(Unp. 'ict
                                                            .re_CHECKSPNONEACTIVE) then
18
           CheckSPAlignment();
19
    else
20
       if n == 31 then CheckSPAlignment();
21
       base = if n == 31 then SP[] else
                                           n, 64];
22
       offset = X[m, 64];
23
24
   for r = 0 to nreg-1
25
        src = Z[t+r, VL];
        for e = 0 to elements-1
26
                                rment(mask,
27
            if ActivePredicat
                                               * elements + e, esize) then
28
                                  re + (UInt( fset) + r * elements + e) * mbytes;
                bits(64) ad
                                     rdesc] = Elem[src, e, esize];
29
                Mem[addr, m.Jytes,
```

D1.1.199 ST1W (scalar plus immediate, strided registers)

Contiguous store of words from multiple strided vectors (immediate index)

Contiguous store of words from elements of two or four strided vector registers to the memory address generated by a 64-bit scalar base and immediate index which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

Inactive elements are not written to memory.

It has encodings from 2 classes: Two registers and Four registers

Two registers (FEAT_SME2)



Assembler Symbols

<Zt1> For the two registers variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as "T:'0':Zt".

For the four registers variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as "T:'00':Zt".

<Zt2> For the two registers variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as "T:'1':Zt".

For the four registers variant: is the name of the second scalable vector register Z4-Z7 or Z20-Z23 to be transferred, encoded as "T:'01':Zt".

- <Zt3> Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as "T:'10':Zt".
- <Zt4> Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as "T:'11':Zt".
- <PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" 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 two registers variant: is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.

For the four registers variant: is the optional signed immediate vector freet, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" fiet.

```
1
   CheckStreamingSVEEnabled();
   constant integer VL = CurrentVL;
2
3
   constant integer PL = VL DIV 8;
4
   constant integer elements = VL DIV esize;
5
   constant integer mbytes = esize DIV 8;
6 bits(64) base;
   bits(VL) src;
7
8
   bits(PL) pred = P[g, PL];
9
   bits(PL * nreg) mask = CounterToPred_ te(pre
                                                     ΡL
                                                        * nreg);
10 boolean contiguous = TRUE;
11
   boolean nontemporal = FALSE;
12 boolean tagchecked = n != 31;
13
   AccessDescriptor accdesc = C .ateAccL
                                       SVE(1. Op_STORE, nontemporal, contiguous,
       \hookrightarrowtagchecked);
14
15
   if !AnyActiveElement(ma k, esi ) then
       if n == 31 && ConstrainUnpre
CheckSPAlinent();
16
                                  ctableBool(Unpredictable_CHECKSPNONEACTIVE) then
17
18
   else
       if n == 31 nen Che .SPAlignment();
19
20
       base = if 1 = 31 nen SP[] else X[n, 64];
21
22
   for r =
               rea-
          = Z[t VL];
23
       sr
24
        \mathbf{r} = 0 to element -1
25
          if
                26
27
              Mem[addr, mbytes, accdesc] = Elem[src, e, esize];
          t
28
       t =
              tstride;
```

D1.1.200 ST1W (scalar plus scalar, strided registers)

Contiguous store of words from multiple strided vectors (scalar index)

Contiguous store of words from elements of two or four strided vector registers to the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated.

Inactive elements are not written to memory.

It has encodings from 2 classes: Two registers and Four registers

Two registers (FEAT_SME2)



<Zt1> For the two registers variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as "T:'0':Zt".

For the four registers variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as "T:'00':Zt".

<Zt2> For the two registers variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as "T:'1':Zt".

For the four registers variant: is the name of the second scalable vector register Z4-Z7 or Z20-Z23 to be transferred, encoded as "T:'01':Zt".

- <Zt3> Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as "T:'10':Zt".
- <Zt4> Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as "T:'11':Zt".
- <PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
 - <Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

```
1 CheckStreamingSVEEnabled();
2
   constant integer VL = CurrentVL;
   constant integer PL = VL DIV 8;
3
   constant integer elements = VL DIV esize;
4
5
   constant integer mbytes = esize DIV 8;
6
   bits(64) offset;
   bits(64) base;
7
8 bits(VL) src;
9
   bits(PL) pred = P[g, PL];
10
   bits(PL * nreg) mask = CounterToPredicate(r
                                                    <15:0.
                                                            PL * nr
   boolean contiguous = TRUE;
11
12 boolean nontemporal = FALSE;
   boolean tagchecked = TRUE;
13
14
   AccessDescriptor accdesc = CreateAccDe SVE/
                                                              nontemporal, contiguous,
        →tagchecked);
15
16
   if !AnyActiveElement(mask, e<sup>c</sup> .e) th
        if n == 31 && Constrair predictab Bool(L predictable_CHECKSPNONEACTIVE) then
17
18
            CheckSPAlignment/
19
   else
20
        if n == 31 then CheckSPAlig ont();
        base = if n == then SP[] & X[n, 64];
offset = X[m 4];
21
22
23
24
   for r = 0 to 1. q-1
        src = Z[t,
25
26
        for
                  tοε
                        ments-1
27
            .r A ivePre cate lement(mask, r * elements + e, esize) then
28
                b .s(64)
                               = base + (UInt(offset) + r * elements + e) * mbytes;
                  _m[addr, muytes, accdesc] = Elem[src, e, esize];
29
30
```

D1.1.201 ST1W (scalar plus scalar, tile slice)

Contiguous store of words from 32-bit element ZA tile slice

The slice number within the tile is selected by the sum of the slice index register and immediate offset, modulo the number of 32-bit elements in a vector. The immediate offset is in the range 0 to 3. The memory address is generated by a 64-bit scalar base and an optional 64-bit scalar offset which is multiplied by 4 and added to the base address. Inactive elements are not written to memory.

SME (FEAT_SME)



constant integer mbytes = esize DIV 8;

12

```
13 boolean contiguous = TRUE;
14 boolean nontemporal = FALSE;
15
    boolean tagchecked = TRUE;
    AccessDescriptor accdesc = CreateAccDescSME(MemOp_STORE, nontemporal, contiguous,
16
         \hookrightarrowtagchecked);
17
    if n == 31 then
18
19
        if AnyActiveElement(mask, esize) ||
20
               ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
21
             CheckSPAlignment();
22
        base = SP[];
23
    else
24
        base = X[n, 64];
25
    src = ZAslice[t, esize, vertical, slice, VL];
26
27
    for e = 0 to dim-1
        addr = base + UInt(moffs) * mbytes;
28
29
        \quad \text{if } \texttt{ActivePredicateElement} \ (\texttt{mask}, \ \texttt{e}, \ \texttt{esize}) \ \text{then} \\
30
             Mem[addr, mbytes, accdesc] = Elem[src, e, esize];
31
        moffs = moffs + 1;
```
D1.1.202 STNT1B (scalar plus immediate, consecutive registers)

Contiguous store non-temporal of bytes from multiple consecutive vectors (immediate index)

Contiguous store non-temporal of bytes from elements of two or four consecutive vector registers to the memory address generated by a 64-bit scalar base and immediate index which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

Inactive elements are not written to memory.

A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: Two registers and Four registers

Two registers (FEAT_SME2)



Assembler Symbols

<Zt1> For the two registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 2.

- <Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" times 4 plus 3.
- <Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" times 2 plus 1.

- <PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" 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 two registers variant: is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.

For the four registers variant: is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.

```
CheckStreamingSVEEnabled();
 1
2 constant integer VL = CurrentVL;
 3
   constant integer PL = VL DIV 8;
   constant integer elements = VL DIV esize;
 4
5
   constant integer mbytes = esize DIV 8;
   bits(64) base;
 6
7
   bits(VL) src;
8 bits(PL) pred = P[g, PL];
9
   bits(PL * nreg) mask = CounterToPredicate(pred<1/
                                                         J>. PL
                                                                  hreg)
10 boolean contiguous = TRUE;
11 boolean nontemporal = TRUE;
12
   boolean tagchecked = n != 31;
   AccessDescriptor accdesc = CreateAccDesc _ (Me. o_STO.
                                                                nc emporal, contiguous,
13
        \hookrightarrowtagchecked);
14
15
   if !AnyActiveElement(mask, esize) then
16
        if n == 31 && ConstrainUnpre. table.
                                                  (Unpredictable_CHECKSPNONEACTIVE) then
17
            CheckSPAlignment();
18
    else
        if n == 31 then CheckS<sup>r</sup> _ignment(),
19
20
        base = if n == 31 th
                                 [] else X[n,
                                                 41:
21
22
    for r = 0 to nreg-1
23
        src = Z[t+r, V^{T}]
24
        for e = 0 to _ements-1
25
            if Act<sup>i</sup> Predic<sup>i</sup> Element(mask, r * elements + e, esize) then
26
                b1 (64) f ar = base + (offset * nreg * elements + r * elements + e) * mbytes;
27
                Mem1 d' mbytes, ccdesc] = Elem[src, e, esize];
```

D1.1.203 STNT1B (scalar plus scalar, consecutive registers)

Contiguous store non-temporal of bytes from multiple consecutive vectors (scalar index)

Contiguous store non-temporal of bytes from elements of two or four consecutive vector registers to the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated.

Inactive elements are not written to memory.

A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: Two registers and Four registers

Two registers (FEAT_SME2)



Assembler Symbols

<Zt1> For the two registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 2.

- <Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" times 4 plus 3.
- <Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" times 2 plus 1.

- <PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
 - <Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

```
1
   CheckStreamingSVEEnabled();
2
   constant integer VL = CurrentVL;
   constant integer PL = VL DIV 8;
3
4
   constant integer elements = VL DIV esize;
5
   constant integer mbytes = esize DIV 8;
6 bits(64) offset;
   bits(64) base;
7
8
   bits(VL) src;
9 bits(PL) pred = P[g, PL];
10 bits(PL * nreg) mask = CounterToPredicate(pred<15:0>, PL *
                                                                     eq);
11
   boolean contiguous = TRUE;
12 boolean nontemporal = TRUE;
13 boolean tagchecked = TRUE;
                                                                 .ontempor. ., contiguous,
   AccessDescriptor accdesc = CreateAccDescSVE(Mem
                                                        STORE
14
        \hookrightarrowtagchecked);
15
16
   if !AnyActiveElement(mask, esize) then
                                                       redicta
                                                                   HECKSPNONEACTIVE) then
17
        if n == 31 && ConstrainUnpredictab]
                                              .ool(Ur
18
            CheckSPAlignment();
19
   else
20
        if n == 31 then CheckSPAlignm nt();
        base = if n == 31 then SP[] e.
21
                                          X[n,
                                                 1;
22
        offset = X[m, 64];
23
   for r = 0 to nreg-1
24
25
        src = Z[t+r, VL];
26
        for e = 0 to elemencs-1
            if ActivePr icateElemen pask, r * elements + e, esize) then
   bits( , addr = base + Uint(offset) + r * elements + e) * mbytes;
27
28
29
                Mem
```

D1.1.204 STNT1B (scalar plus immediate, strided registers)

Contiguous store non-temporal of bytes from multiple strided vectors (immediate index)

Contiguous store non-temporal of bytes from elements of two or four strided vector registers to the memory address generated by a 64-bit scalar base and immediate index which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

Inactive elements are not written to memory.

A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: Two registers and Four registers

Two registers (FEAT_SME2)



Assembler Symbols

<Zt1> For the two registers variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as "T:'0':Zt".

For the four registers variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as "T:'00':Zt".

<Zt2> For the two registers variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as "T:'1':Zt".

- <Zt3> Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as "T:'10':Zt".
- <Zt4> Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as "T:'11':Zt".
- <PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" 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 two registers variant: is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.

For the four registers variant: is the optional signed immediate _ ctor offset, multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4___ld.



D1.1.205 STNT1B (scalar plus scalar, strided registers)

Contiguous store non-temporal of bytes from multiple strided vectors (scalar index)

Contiguous store non-temporal of bytes from elements of two or four strided vector registers to the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated.

Inactive elements are not written to memory.

A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: Two registers and Four registers

Two registers (FEAT_SME2)



Assembler Symbols

<Zt1> For the two registers variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as "T:'0':Zt".

For the four registers variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as "T:'00':Zt".

<Zt2> For the two registers variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as "T:'1':Zt".

- <Zt3> Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as "T:'10':Zt".
- <Zt4> Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as "T:'11':Zt".
- <PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
 - <Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

```
CheckStreamingSVEEnabled();
1
   constant integer VL = CurrentVL;
2
   constant integer PL = VL DIV 8;
3
4
   constant integer elements = VL DIV esize;
   constant integer mbytes = esize DIV 8;
5
6
   bits(64) offset;
7
   bits(64) base;
8
   bits(VL) src;
9
   bits(PL) pred = P[g, PL];
10
   bits(PL * nreg) mask = CounterToPredica (pred
                                                     5:0>,
                                                                irea);
11 boolean contiguous = TRUE;
12 boolean nontemporal = TRUE;
13
   boolean tagchecked = TRUE;
   AccessDescriptor accdesc = Creater Descs. MemOp_STORE, nontemporal, contiguous,
14
        →tagchecked);
15
16
   if !AnyActiveElement(mask
                                ize) then
17
        if n == 31 && Const InU. edictableB. (Unpredictable_CHECKSPNONEACTIVE) then
18
            CheckSPAlignmenc();
19
    else
20
       if n == 31 tb . CheckSPAlignme. ();
        base = if p = 31 th SP[] else X[n, 64];
21
22
        offset = : , 64];
23
24
   for r = 0 -- nreg
25
        src
                   VL];
              4
            e = 0 o elem
26
        f
27
            if A ivePredic.teElement(mask, r * elements + e, esize) then
28
                         .ddr = base + (UInt(offset) + r * elements + e) * mbytes;
29
               Mem[addr, mbytes, accdesc] = Elem[src, e, esize];
30
              + tstride;
```

D1.1.206 STNT1D (scalar plus immediate, consecutive registers)

Contiguous store non-temporal of doublewords from multiple consecutive vectors (immediate index)

Contiguous store non-temporal of doublewords from elements of two or four consecutive vector registers to the memory address generated by a 64-bit scalar base and immediate index which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

Inactive elements are not written to memory.

A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: Two registers and Four registers

Two registers (FEAT_SME2)



Assembler Symbols

<Zt1> For the two registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 2.

- <Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" times 4 plus 3.
- <Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" times 2 plus 1.

- <PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" 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 two registers variant: is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.

For the four registers variant: is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.

```
CheckStreamingSVEEnabled();
 1
2 constant integer VL = CurrentVL;
 3
   constant integer PL = VL DIV 8;
   constant integer elements = VL DIV esize;
 4
5
   constant integer mbytes = esize DIV 8;
   bits(64) base;
 6
7
   bits(VL) src;
8 bits(PL) pred = P[g, PL];
9
   bits(PL * nreg) mask = CounterToPredicate(pred<1/
                                                         J>. PL
                                                                  hreg)
10 boolean contiguous = TRUE;
11 boolean nontemporal = TRUE;
12
   boolean tagchecked = n != 31;
   AccessDescriptor accdesc = CreateAccDesc _ (Me. o_STO.
                                                                nc emporal, contiguous,
13
        \hookrightarrowtagchecked);
14
15
   if !AnyActiveElement(mask, esize) then
16
        if n == 31 && ConstrainUnpre. table.
                                                  (Unpredictable_CHECKSPNONEACTIVE) then
17
            CheckSPAlignment();
18
    else
        if n == 31 then CheckS<sup>r</sup> _ignment(),
19
20
        base = if n == 31 th
                                 [] else X[n,
                                                 41:
21
22
    for r = 0 to nreg-1
23
        src = Z[t+r, V^{T}]
24
        for e = 0 to _ements-1
25
            if Act<sup>i</sup> Predic<sup>i</sup> Element(mask, r * elements + e, esize) then
26
                b1 (64) f ar = base + (offset * nreg * elements + r * elements + e) * mbytes;
27
                Mem1 d' mbytes, ccdesc] = Elem[src, e, esize];
```

D1.1.207 STNT1D (scalar plus scalar, consecutive registers)

Contiguous store non-temporal of doublewords from multiple consecutive vectors (scalar index)

Contiguous store non-temporal of doublewords from elements of two or four consecutive vector registers to the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated.

Inactive elements are not written to memory.

A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: Two registers and Four registers

Two registers (FEAT_SME2)



<Zt1> For the two registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 2.

- <Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" times 4 plus 3.
- <Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" times 2 plus 1.

- <PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
 - <Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

```
1
   CheckStreamingSVEEnabled();
   constant integer VL = CurrentVL;
2
   constant integer PL = VL DIV 8;
3
4
   constant integer elements = VL DIV esize;
5
   constant integer mbytes = esize DIV 8;
6 bits(64) offset;
   bits(64) base;
7
8
   bits(VL) src;
9 bits(PL) pred = P[g, PL];
10 bits(PL * nreg) mask = CounterToPredicate(pred<15:0>, PL *
                                                                     eq);
11
   boolean contiguous = TRUE;
12 boolean nontemporal = TRUE;
13 boolean tagchecked = TRUE;
                                                                 .ontempor. ., contiguous,
   AccessDescriptor accdesc = CreateAccDescSVE(Mem
                                                        STORE
14
        \hookrightarrowtagchecked);
15
16
   if !AnyActiveElement(mask, esize) then
                                                       redicta
                                                                   HECKSPNONEACTIVE) then
17
        if n == 31 && ConstrainUnpredictab]
                                              .ool(Ur
18
            CheckSPAlignment();
19
   else
20
        if n == 31 then CheckSPAlignm nt();
        base = if n == 31 then SP[] e.
21
                                          X[n,
                                                 1;
22
        offset = X[m, 64];
23
   for r = 0 to nreg-1
24
25
        src = Z[t+r, VL];
26
        for e = 0 to elemencs-1
            if ActivePr icateElemen pask, r * elements + e, esize) then
   bits( , addr = base + Uint(offset) + r * elements + e) * mbytes;
27
28
29
                Mem
```

D1.1.208 STNT1D (scalar plus immediate, strided registers)

Contiguous store non-temporal of doublewords from multiple strided vectors (immediate index)

Contiguous store non-temporal of doublewords from elements of two or four strided vector registers to the memory address generated by a 64-bit scalar base and immediate index which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

Inactive elements are not written to memory.

A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: Two registers and Four registers

Two registers (FEAT_SME2)



Assembler Symbols

<Zt1> For the two registers variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as "T:'0':Zt".

For the four registers variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as "T:'00':Zt".

<Zt2> For the two registers variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as "T:'1':Zt".

- <Zt3> Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as "T:'10':Zt".
- <Zt4> Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as "T:'11':Zt".
- <PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" 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 two registers variant: is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.

For the four registers variant: is the optional signed immediate _ ctor offset, multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4___ld.



D1.1.209 STNT1D (scalar plus scalar, strided registers)

Contiguous store non-temporal of doublewords from multiple strided vectors (scalar index)

Contiguous store non-temporal of doublewords from elements of two or four strided vector registers to the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated.

Inactive elements are not written to memory.

A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: Two registers and Four registers

Two registers (FEAT_SME2)



Assembler Symbols

<Zt1> For the two registers variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as "T:'0':Zt".

For the four registers variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as "T:'00':Zt".

<Zt2> For the two registers variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as "T:'1':Zt".

- <Zt3> Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as "T:'10':Zt".
- <Zt4> Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as "T:'11':Zt".
- <PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
 - <Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

```
CheckStreamingSVEEnabled();
1
   constant integer VL = CurrentVL;
2
   constant integer PL = VL DIV 8;
3
4
   constant integer elements = VL DIV esize;
   constant integer mbytes = esize DIV 8;
5
6
   bits(64) offset;
7
   bits(64) base;
8
   bits(VL) src;
9
   bits(PL) pred = P[g, PL];
10
   bits(PL * nreg) mask = CounterToPredica (pred
                                                     5:0>,
                                                                irea);
11 boolean contiguous = TRUE;
12 boolean nontemporal = TRUE;
13
   boolean tagchecked = TRUE;
   AccessDescriptor accdesc = Creater Descs. MemOp_STORE, nontemporal, contiguous,
14
        →tagchecked);
15
16
   if !AnyActiveElement(mask
                                ize) then
17
        if n == 31 && Const InU. edictableB. (Unpredictable_CHECKSPNONEACTIVE) then
18
            CheckSPAlignmenc();
19
    else
20
       if n == 31 tb . CheckSPAlignme. ();
        base = if p = 31 th SP[] else X[n, 64];
21
22
        offset = : , 64];
23
24
   for r = 0 -- nreg
25
        src
                   VL];
              4
            e = 0 o elem
26
        f
27
            if A ivePredic.teElement(mask, r * elements + e, esize) then
28
                         .ddr = base + (UInt(offset) + r * elements + e) * mbytes;
29
               Mem[addr, mbytes, accdesc] = Elem[src, e, esize];
30
              + tstride;
```

D1.1.210 STNT1H (scalar plus immediate, consecutive registers)

Contiguous store non-temporal of halfwords from multiple consecutive vectors (immediate index)

Contiguous store non-temporal of halfwords from elements of two or four consecutive vector registers to the memory address generated by a 64-bit scalar base and immediate index which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

Inactive elements are not written to memory.

A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: Two registers and Four registers

Two registers (FEAT_SME2)



Assembler Symbols

<Zt1> For the two registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 2.

- <Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" times 4 plus 3.
- <Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" times 2 plus 1.

- <PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" 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 two registers variant: is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.

For the four registers variant: is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.

```
CheckStreamingSVEEnabled();
 1
2 constant integer VL = CurrentVL;
 3
   constant integer PL = VL DIV 8;
   constant integer elements = VL DIV esize;
 4
5
   constant integer mbytes = esize DIV 8;
   bits(64) base;
 6
7
   bits(VL) src;
8 bits(PL) pred = P[g, PL];
 9
   bits(PL * nreg) mask = CounterToPredicate(pred<1/
                                                         J>. PL
                                                                  hreg)
10 boolean contiguous = TRUE;
11 boolean nontemporal = TRUE;
12
   boolean tagchecked = n != 31;
   AccessDescriptor accdesc = CreateAccDesc _ (Me. o_STO.
                                                                nc emporal, contiguous,
13
        \hookrightarrowtagchecked);
14
15
   if !AnyActiveElement(mask, esize) then
16
        if n == 31 && ConstrainUnpre. table.
                                                  (Unpredictable_CHECKSPNONEACTIVE) then
17
            CheckSPAlignment();
18
    else
        if n == 31 then CheckS<sup>r</sup> _ignment(),
19
20
        base = if n == 31 th
                                 [] else X[n,
                                                 41:
21
22
    for r = 0 to nreg-1
23
        src = Z[t+r, V^{T}]
24
        for e = 0 to _ements-1
25
            if Act<sup>i</sup> Predic<sup>i</sup> Element(mask, r * elements + e, esize) then
26
                b1 (64) f ar = base + (offset * nreg * elements + r * elements + e) * mbytes;
27
                Mem1 d' mbytes, ccdesc] = Elem[src, e, esize];
```

D1.1.211 STNT1H (scalar plus scalar, consecutive registers)

Contiguous store non-temporal of halfwords from multiple consecutive vectors (scalar index)

Contiguous store non-temporal of halfwords from elements of two or four consecutive vector registers to the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated.

Inactive elements are not written to memory.

A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: Two registers and Four registers

Two registers (FEAT_SME2)



<Zt1> For the two registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 2.

- <Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" times 4 plus 3.
- <Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" times 2 plus 1.

- <PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
 - <Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

```
1
   CheckStreamingSVEEnabled();
2
   constant integer VL = CurrentVL;
   constant integer PL = VL DIV 8;
3
4
   constant integer elements = VL DIV esize;
5
   constant integer mbytes = esize DIV 8;
6 bits(64) offset;
   bits(64) base;
7
8
   bits(VL) src;
9 bits(PL) pred = P[g, PL];
10 bits(PL * nreg) mask = CounterToPredicate(pred<15:0>, PL *
                                                                     eq);
11
   boolean contiguous = TRUE;
12 boolean nontemporal = TRUE;
13 boolean tagchecked = TRUE;
                                                                 .ontempor. ., contiguous,
   AccessDescriptor accdesc = CreateAccDescSVE(Mem
                                                        STORE
14
        \hookrightarrowtagchecked);
15
16
   if !AnyActiveElement(mask, esize) then
                                                       redicta
                                                                   HECKSPNONEACTIVE) then
17
        if n == 31 && ConstrainUnpredictab]
                                              .ool(Ur
18
            CheckSPAlignment();
19
   else
20
        if n == 31 then CheckSPAlignm nt();
        base = if n == 31 then SP[] e.
21
                                          X[n,
                                                 1;
22
        offset = X[m, 64];
23
   for r = 0 to nreg-1
24
25
        src = Z[t+r, VL];
26
        for e = 0 to elemencs-1
            if ActivePr icateElemen pask, r * elements + e, esize) then
   bits( , addr = base + Uint(offset) + r * elements + e) * mbytes;
27
28
29
                Mem
```

D1.1.212 STNT1H (scalar plus immediate, strided registers)

Contiguous store non-temporal of halfwords from multiple strided vectors (immediate index)

Contiguous store non-temporal of halfwords from elements of two or four strided vector registers to the memory address generated by a 64-bit scalar base and immediate index which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

Inactive elements are not written to memory.

A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: Two registers and Four registers

Two registers (FEAT_SME2)



8 integer offset = SInt(imm4);

Assembler Symbols

<Zt1> For the two registers variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as "T:'0':Zt".

For the four registers variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as "T:'00':Zt".

<Zt2> For the two registers variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as "T:'1':Zt".

- <Zt3> Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as "T:'10':Zt".
- <Zt4> Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as "T:'11':Zt".
- <PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" 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 two registers variant: is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.

For the four registers variant: is the optional signed immediate ctor offset, multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4___ld.



D1.1.213 STNT1H (scalar plus scalar, strided registers)

Contiguous store non-temporal of halfwords from multiple strided vectors (scalar index)

Contiguous store non-temporal of halfwords from elements of two or four strided vector registers to the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated.

Inactive elements are not written to memory.

A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: Two registers and Four registers

Two registers (FEAT_SME2)



Assembler Symbols

<Zt1> For the two registers variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as "T:'0':Zt".

For the four registers variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as "T:'00':Zt".

<Zt2> For the two registers variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as "T:'1':Zt".

- <Zt3> Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as "T:'10':Zt".
- <Zt4> Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as "T:'11':Zt".
- <PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
 - <Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

```
CheckStreamingSVEEnabled();
1
   constant integer VL = CurrentVL;
2
   constant integer PL = VL DIV 8;
3
4
   constant integer elements = VL DIV esize;
   constant integer mbytes = esize DIV 8;
5
6
   bits(64) offset;
7
   bits(64) base;
8
   bits(VL) src;
9
   bits(PL) pred = P[g, PL];
10
   bits(PL * nreg) mask = CounterToPredica (pred
                                                     5:0>,
                                                                irea);
11 boolean contiguous = TRUE;
12 boolean nontemporal = TRUE;
13
   boolean tagchecked = TRUE;
   AccessDescriptor accdesc = Creater Descs. MemOp_STORE, nontemporal, contiguous,
14
        →tagchecked);
15
16
   if !AnyActiveElement(mask
                                ize) then
17
        if n == 31 && Const InU. edictableB. (Unpredictable_CHECKSPNONEACTIVE) then
18
            CheckSPAlignmenc();
19
    else
20
       if n == 31 tb . CheckSPAlignme. ();
        base = if p = 31 th SP[] else X[n, 64];
21
22
        offset = : , 64];
23
24
   for r = 0 -- nreg
25
        src
                   VL];
              4
            e = 0 o elem
26
        f
27
            if A ivePredic.teElement(mask, r * elements + e, esize) then
28
                         .ddr = base + (UInt(offset) + r * elements + e) * mbytes;
29
               Mem[addr, mbytes, accdesc] = Elem[src, e, esize];
30
              + tstride;
```

D1.1.214 STNT1W (scalar plus immediate, consecutive registers)

Contiguous store non-temporal of words from multiple consecutive vectors (immediate index)

Contiguous store non-temporal of words from elements of two or four consecutive vector registers to the memory address generated by a 64-bit scalar base and immediate index which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

Inactive elements are not written to memory.

A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: Two registers and Four registers

Two registers (FEAT_SME2)



Assembler Symbols

<Zt1> For the two registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 2.

- <Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" times 4 plus 3.
- <Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" times 2 plus 1.

- <PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" 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 two registers variant: is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.

For the four registers variant: is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.

```
CheckStreamingSVEEnabled();
 1
2 constant integer VL = CurrentVL;
 3
   constant integer PL = VL DIV 8;
   constant integer elements = VL DIV esize;
 4
5
   constant integer mbytes = esize DIV 8;
   bits(64) base;
 6
7
   bits(VL) src;
8 bits(PL) pred = P[g, PL];
 9
   bits(PL * nreg) mask = CounterToPredicate(pred<1/
                                                         J>. PL
                                                                  hreg)
10 boolean contiguous = TRUE;
11 boolean nontemporal = TRUE;
12
   boolean tagchecked = n != 31;
   AccessDescriptor accdesc = CreateAccDesc _ (Me. o_STO.
                                                                nc emporal, contiguous,
13
        \hookrightarrowtagchecked);
14
15
   if !AnyActiveElement(mask, esize) then
16
        if n == 31 && ConstrainUnpre. table.
                                                  (Unpredictable_CHECKSPNONEACTIVE) then
17
            CheckSPAlignment();
18
    else
        if n == 31 then CheckS<sup>r</sup> _ignment(),
19
20
        base = if n == 31 th
                                 [] else X[n,
                                                 41:
21
22
    for r = 0 to nreg-1
23
        src = Z[t+r, V^{T}]
24
        for e = 0 to _ements-1
25
            if Act<sup>i</sup> Predic<sup>i</sup> Element(mask, r * elements + e, esize) then
26
                b1 (64) f ar = base + (offset * nreg * elements + r * elements + e) * mbytes;
27
                Mem1 d' mbytes, ccdesc] = Elem[src, e, esize];
```

D1.1.215 STNT1W (scalar plus scalar, consecutive registers)

Contiguous store non-temporal of words from multiple consecutive vectors (scalar index)

Contiguous store non-temporal of words from elements of two or four consecutive vector registers to the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated.

Inactive elements are not written to memory.

A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: Two registers and Four registers

Two registers (FEAT_SME2)



<Zt1> For the two registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 2.

- <Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" times 4 plus 3.
- <Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" times 2 plus 1.

- <PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
 - <Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

```
1
   CheckStreamingSVEEnabled();
2
   constant integer VL = CurrentVL;
   constant integer PL = VL DIV 8;
3
4
   constant integer elements = VL DIV esize;
5
   constant integer mbytes = esize DIV 8;
6 bits(64) offset;
   bits(64) base;
7
8
   bits(VL) src;
9 bits(PL) pred = P[g, PL];
10 bits(PL * nreg) mask = CounterToPredicate(pred<15:0>, PL *
                                                                     eq);
11
   boolean contiguous = TRUE;
12 boolean nontemporal = TRUE;
13 boolean tagchecked = TRUE;
                                                                 .ontempor. ., contiguous,
   AccessDescriptor accdesc = CreateAccDescSVE(Mem
                                                        STORE
14
        \hookrightarrowtagchecked);
15
16
   if !AnyActiveElement(mask, esize) then
                                                       redicta
                                                                   HECKSPNONEACTIVE) then
17
        if n == 31 && ConstrainUnpredictab]
                                              .ool(Ur
18
            CheckSPAlignment();
19
   else
20
        if n == 31 then CheckSPAlignm nt();
        base = if n == 31 then SP[] e.
21
                                          X[n,
                                                 1;
22
        offset = X[m, 64];
23
   for r = 0 to nreg-1
24
25
        src = Z[t+r, VL];
26
        for e = 0 to elemencs-1
            if ActivePr icateElemen pask, r * elements + e, esize) then
   bits( , addr = base + Uint(offset) + r * elements + e) * mbytes;
27
28
29
                Mem
```

D1.1.216 STNT1W (scalar plus immediate, strided registers)

Contiguous store non-temporal of words from multiple strided vectors (immediate index)

Contiguous store non-temporal of words from elements of two or four strided vector registers to the memory address generated by a 64-bit scalar base and immediate index which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

Inactive elements are not written to memory.

A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: Two registers and Four registers

Two registers (FEAT_SME2)



Assembler Symbols

<Zt1> For the two registers variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as "T:'0':Zt".

For the four registers variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as "T:'00':Zt".

<Zt2> For the two registers variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as "T:'1':Zt".

- <Zt3> Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as "T:'10':Zt".
- <Zt4> Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as "T:'11':Zt".
- <PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" 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 two registers variant: is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.

For the four registers variant: is the optional signed immediate _ ctor offset, multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4___ld.



D1.1.217 STNT1W (scalar plus scalar, strided registers)

Contiguous store non-temporal of words from multiple strided vectors (scalar index)

Contiguous store non-temporal of words from elements of two or four strided vector registers to the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated.

Inactive elements are not written to memory.

A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: Two registers and Four registers

Two registers (FEAT_SME2)



Assembler Symbols

<Zt1> For the two registers variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as "T:'0':Zt".

For the four registers variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as "T:'00':Zt".

<Zt2> For the two registers variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as "T:'1':Zt".

- <Zt3> Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as "T:'10':Zt".
- <Zt4> Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as "T:'11':Zt".
- <PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
 - <Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

```
CheckStreamingSVEEnabled();
1
   constant integer VL = CurrentVL;
2
   constant integer PL = VL DIV 8;
3
4
   constant integer elements = VL DIV esize;
   constant integer mbytes = esize DIV 8;
5
6
   bits(64) offset;
7
   bits(64) base;
8
   bits(VL) src;
9
   bits(PL) pred = P[g, PL];
10
   bits(PL * nreg) mask = CounterToPredica (pred
                                                     5:0>,
                                                                irea);
11 boolean contiguous = TRUE;
12 boolean nontemporal = TRUE;
13
   boolean tagchecked = TRUE;
   AccessDescriptor accdesc = Creater Descs. MemOp_STORE, nontemporal, contiguous,
14
        →tagchecked);
15
16
   if !AnyActiveElement(mask
                                ize) then
17
        if n == 31 && Const InU. edictableB. (Unpredictable_CHECKSPNONEACTIVE) then
18
            CheckSPAlignmenc();
19
    else
20
       if n == 31 tb . CheckSPAlignme. ();
        base = if p = 31 th SP[] else X[n, 64];
21
22
        offset = : , 64];
23
24
   for r = 0 -- nreg
25
        src
                   VL];
              4
            e = 0 o elem
26
        f
27
            if A ivePredic.teElement(mask, r * elements + e, esize) then
28
                         .ddr = base + (UInt(offset) + r * elements + e) * mbytes;
29
               Mem[addr, mbytes, accdesc] = Elem[src, e, esize];
30
              + tstride;
```

D1.1.218 STR (vector)

Store ZA array vector

The ZA array vector is selected by the sum of the vector select register and immediate offset, modulo the number of bytes in a Streaming SVE vector. The immediate offset is in the range 0 to 15. The memory address is generated by a 64-bit scalar base, plus the same optional immediate offset multiplied by the current vector length in bytes. This instruction is unpredicated.

The store is performed as contiguous byte accesses, with no endian conversion and no guarantee of single-copy atomicity larger than a byte. However, if alignment is checked, then the base register must be aligned to 16 bytes.

This instruction does not require the PE to be in Streaming SVE mode, and it is expected that this instruction will not experience a significant slowdown due to contention with other PEs that are executing in Streaming SVE mode.

SME (FEAT_SME)



Mu

11

```
STR ZA[<Wv>, <offs>], [<Xn|SP>{, #<offs>,
```

```
1 if !HaveSME() then UNDEFINED;
```

```
2 integer n = UInt(Rn);
```

```
3 integer v = UInt('011':Rv);
4 integer offset = UInt(off4);
```

Assembler Symbols

- $\langle Wv \rangle$ Is the 32-bit name of t' or select regiver W12-W15, encoded in the "Rv" field.
- <offs> Is the vector select offset and ptional memory offset, in the range 0 to 15, defaulting to 0, encoded in the off4" field.
- <Xn|SP> Is the 64-b name of e general-purpose base register or stack pointer, encoded in the "Rn" field.

Operati

```
Chec MEAndZA habled(),
1
2
   con. nt i
                      CVL =
                            CurrentSVL;
               iteger and = SVL DIV 8;
3
   const.
4
   bits(64
              base;
5
               offs = offset * dim;
   integer
6
   bits(SVL) s
7
   bits(32)
              vbase = X[v, 32];
              vec = (UInt(vbase) + offset) MOD dim;
8
   integer
9
   boolean
              contiguous = TRUE;
10
   boolean
              nontemporal = FALSE;
              tagchecked = n != 31;
11
   boolean
   AccessDescriptor accdesc = CreateAccDescSME (MemOp_STORE, nontemporal, contiguous,
12
        →tagchecked);
13
14
   if HaveTME() && TSTATE.depth > 0 then
15
        FailTransaction(TMFailure_ERR, FALSE);
16
17
   if n == 31 then
18
        CheckSPAlignment();
19
        base = SP[];
20
    else
21
        base = X[n, 64];
22
23
   src = ZAvector[vec, SVL];
```

Chapter D1. SME instructions D1.1. SME and SME2 data-processing instructions

> 24 25 boolean aligned = IsAligned(base + offset, 16); 26 27 if !aligned && AlignmentEnforced() then 28 AArch64.Abort(base + moffs, AlignmentFault(accdesc)); 29 30 for e = 0 to dim-1 31 AArch64.MemSingle[base + moffs, 1, accdesc, aligned] = Elem[src, e, 8]; 32 moffs = moffs + 1;

D1.1.219 STR (ZT0)

Store ZT0 register

Store the 64-byte ZT0 register to the memory address provided in the 64-bit scalar base register. This instruction is unpredicated.

The store is performed as contiguous byte accesses, with no endian conversion and no guarantee of single-copy atomicity larger than a byte. However, if alignment is checked, then the base register must be aligned to 16 bytes.

This instruction does not require the PE to be in Streaming SVE mode, and it is expected that this instruction will not experience a significant slowdown due to contention with other PEs that are executing in Streaming SVE mode.

SME2 (FEAT_SME2)



Chapter D1. SME instructions D1.1. SME and SME2 data-processing instructions

D1.1.220 SUB (array accumulators)

Subtract multi-vector from ZA array vector accumulators

The instruction operates on two or four ZA single-vector groups.

Destructively subtract all elements of the two or four source vectors from the corresponding elements of the two or four ZA single-vector groups. The vector numbers forming the single-vector group within each half or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

ID_AA64SMFR0_EL1.116I64 indicates whether the 64-bit integer v ant is implem ted.

It has encodings from 2 classes: Two ZA single-vectors and For ZA single-vectors

Two ZA single-vectors (FEAT_SME2)



Assembler Symbols

<T> Is the size specifier, encoded in "sz":

SZ	<t></t>
0	S
1	D
- <Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
- <offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.
- <Zm1> For the two ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 2.

For the four ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 4.

- <Zm4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zm" times 4 plus 3.
- <Zm2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zm" times 2 plus 1.

```
1
   CheckStreamingSVEAndZAEnabled();
   constant integer VL = CurrentVL;
2
3
   constant integer elements = VL DIV esize;
   integer vectors = VL DIV 8;
4
   integer vstride = vectors DIV nreg;
5
6 bits(32) vbase = X[v, 32];
7
   integer vec = (UInt(vbase) + offset) MOD
8 bits(VL) result;
9
10
   for r = 0 to nreg-1
        bits(VL) operand1 = ZAvector['ec,
11
                                               V.
        bits(VL) operand2 = Z[m+r, VL,
12
13
        for e = 0 to elements-1
            bits(esize) element = Elem[or and1, esize];
bits(esize) eleme = Elem[ope. d2, e, esize];
14
15
        Elem[result, e, siz = element1 element2;
ZAvector[vec, VL] = result,
16
17
18
        vec = vec + vst de;
```

D1.1.221 SUB (array results, multiple and single vector)

Subtract replicated single vector from multi-vector with ZA array vector results

The instruction operates on two or four ZA single-vector groups.

Subtract all corresponding elements of the second source vector from the two or four first source vectors and place the results in the corresponding elements of the two or four ZA single-vector groups. The vector numbers forming the single-vector group within each half or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

ID_AA64SMFR0_EL1.I16I64 indicates whether the 64-bit integer v ant is implem ted.

It has encodings from 2 classes: Two ZA single-vectors and For ZA single-vectors

Two ZA single-vectors (FEAT_SME2)



```
7 integer offset = UInt(off3)
8 constant integer nreg = 4;
```

Assembler Symbols

<T> Is the size specifier, encoded in "sz":

SZ	< T >
0	S
1	D

- <Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
- <offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.
- <Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn".
- <Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" plus 3 modulo 32.
- <Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" plus 1 modulo 32.
- <Zm> Is the name of the second source scalable vector register Z0-Z1^r encoded in e "Zm" field.

```
CheckStreamingSVEAndZAEnabled();
1
2
   constant integer VL = CurrentVL;
3
   constant integer elements = VL DIV esize;
4
   integer vectors = VL DIV 8;
   integer vstride = vectors DIV nreg;
5
   bits(32) vbase = X[v, 32];
6
   integer vec = (UInt(vbase) + offset) M
7
                                            vstri
8
   bits(VL) result;
9
10
   for r = 0 to nreg-1
11
       bits(VL) operand1 = Z[(n+r) MO. 32],
                                          VL.
       bits(VL) operand2 = Z[m, _];
12
13
       for e = 0 to elements-'
14
          bits(esize) elem
                                = Elem[oper ]1, e, esize];
       Flem[operan 2, e, esize];
15
16
                                    element1 - element2;
17
18
       vec = vec + ·
                     .ride;
```

D1.1.222 SUB (array results, multiple vectors)

Subtract multi-vector from multi-vector with ZA array vector results

The instruction operates on two or four ZA single-vector groups.

Subtract all corresponding elements of the two or four second source vectors from first source vectors and place the results in the corresponding elements of the two or four ZA single-vector groups. The vector numbers forming the single-vector group within each half or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

ID_AA64SMFR0_EL1.I16I64 indicates whether the 64-bit integer v ant is implem ted.

It has encodings from 2 classes: Two ZA single-vectors and For ZA sin *n*-vectors

Two ZA single-vectors (FEAT_SME2)

	(FFAT SMF2)
	$ \begin{array}{c ccccccccccccccccccccccccccccccccccc$
	SUB ZA. <t>[<wv>, <offs>{, V, `}], { Zn1>.<t>-<zn2>.<t> }, { <zm1>.<t>-<zm2>.<t> ↔}</t></zm2></t></zm1></t></zn2></t></offs></wv></t>
1	if HaveSME2() then UNDEF
3	<pre>integer v = UInt('010':Rv);</pre>
4	constant integer esse = 32 << Ul (sz);
5	<pre>integer n = UInt (:'0'); integer m = UInt (:'0');</pre>
7	<pre>integer m = 01 / 2m: 0 integer offset UInt / f3);</pre>
8	constant integer re = 2;
	Four 7 . singly vectors (FF 1_SME?
	31 1 0 0 0 0 0 1 1 sz 1 Zm 0 1 0 Rv 1 1 0 Zn 0 0 1 1 z c 0 0 0 1 1 sz 1 Sz 1 Zm 0 1 0 Sz 1 Sz 1 Sz 1 Sz 1 Sz 1 Sz 1 Sz
	SUB ZA. <t>[<wv>, <offs>{, VGx4}], { <zn1>.<t>-<zn4>.<t> }, { <zm1>.<t>-<zm4>.<t> →}</t></zm4></t></zm1></t></zn4></t></zn1></offs></wv></t>
1	if LUDYCSME2() then UNDEFINED.
2	<pre>if sz == '1' && !HaveSMEI16164() then UNDEFINED;</pre>
3	<pre>integer v = UInt('010':Rv);</pre>
4 5	<pre>constant integer esize = 32 << UInt(sz); integer n = UInt(Zn: '00');</pre>

Assembler Symbols

integer m = UInt(Zm:'00');

integer offset = UInt(off3); constant integer nreg = 4;

<T> Is the size specifier, encoded in "sz":

6 7

8

SZ	<t></t>
0	S
1	D

- <Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
- <offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.
- <Zn1> For the two ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.

For the four ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

- <Zn4> Is the name of the fourth scalable vector register of a multi-vector sector encoded as "Zn" times 4 plus 3.
- <Zn2> Is the name of the second scalable vector register of a multi-ctor sequence encoded as "Zn" times 2 plus 1.
- <Zm1> For the two ZA single-vectors variant: is the name cone first scalar vec c register of a multi-vector sequence, encoded as "Zm" times 2.

For the four ZA single-vectors variant: is the name c^{-1} first scale is vector register of a multi-vector sequence, encoded as "Zm" $t^2 = c_0$

- <Zm4> Is the name of the fourth scalable ve or regist of a mult. Jector sequence, encoded as "Zm" times 4 plus 3.
- <Zm2> Is the name of the second scalar vector gister of a multi-vector sequence, encoded as "Zm" times 2 plus 1.

```
CheckStreamingSVEAnd"AEnabled()
1
    constant integer VV - CurrentVL;
2
   constant integer v currentvu,
constant integer lements = VL DIV ize;
integer vector v VL DIV 3;
integer vstric = vect is DIV nreg;
bits(32) vbase = [v 32];
3
4
5
6
    integer v - (UIn. vbase)
                                       offset) MOD vstride;
7
    bits(V' res. ;;
8
9
    for = 0 to real
10
11
         . 's (V open
                             = Z[n+r, VL];
12
         bi. L) operand2 = Z[m+r, VL];
         for 0 to elements-1
13
14
              bi 'esize) element1 = Elem[operand1, e, esize];
15
              bits .size) element2 = Elem[operand2, e, esize];
              Elem[result, e, esize] = element1 - element2;
16
17
         ZAvector[vec, VL] = result;
18
         vec = vec + vstride;
```

D1.1.223 SUDOT (multiple and indexed vector)

Multi-vector signed by unsigned integer dot-product by indexed elements

The instruction operates on two or four ZA single-vector groups.

The signed by unsigned integer dot product instruction computes the dot product of four signed 8-bit integer values held in each 32-bit element of the two or four first source vectors and four unsigned 8-bit integer values in the corresponding indexed 32-bit element of the second source vector. The widened dot product result is destructively added to the corresponding 32-bit element of the two or four ZA single-vector groups.

The groups within the second source vector are specified using an immediate element index which selects the same group position within each 128-bit vector segment. The index range is from 0 to 3, encoded in 2 bits. The vector numbers forming the single-vector group within each half or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the the of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA oper disconsists of voor four ZA single-vector groups respectively. The VECTOR GROUP symbol is preferred for disconsists of the vector on a sembler source code.

This instruction is unpredicated.

It has encodings from 2 classes: Two ZA single-vectors a. Four' A single-vectors

Two ZA single-vectors (FEAT_SME2)



SUDOT ZA.S[<Wv>, <offs>{, VGx4}], { <Zn1>.B-<Zn4>.B }, <Zm>.B[<index>]

```
1 if !HaveSME2() then UNDEFINED;
```

```
2 integer v = UInt('010':Rv);
```

```
3 constant integer esize = 32;
```

```
4 integer n = UInt(Zn:'00');
5 integer m = UInt('0':Zm);
```

```
6 integer offset = UInt(off3);
```

```
7 integer index = UInt(i2);
```

```
8 constant integer nreg = 4;
```

Assembler Symbols

- <Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
- <offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.
- <Zn1> For the two ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.

For the four ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

- <Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.
- <Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.
- <Zm> Is the name of the second source scalable vector register Z0-Z['], encoded in "Zm" field.

<index> Is the element index, in the range 0 to 3, encoded in the "2" field

```
CheckStreamingSVEAndZAEnabled();
 1
 2
    constant integer VL = CurrentVL;
    constant integer elements = VL DIV esize;
 3
4
    integer vectors = VL DIV 8;
    integer vstride = vectors DIV nreg;
 5
    integer eltspersegment = 128 DIV esiz
6
7
    bits(32) vbase = X[v, 32];
 8
    integer vec = (UInt(vbase) + offs
                                             MOD
                                                     ride;
9
    bits(VL) result;
10
11
    for r = 0 to nreg-1
                              , nh
12
        bits(VL) operand1 =
                                      VL];
        bits(VL) operand2 = Z[m,
13
14
        bits(VL) operanc' = ZAvector
                                          ec, VL];
15
        for e = 0 to \epsilon .ments-1
            bits(esi ) sum = Elem[oper.d3, e, esize];
intege segment se = e - (e MOD eltspersegment);
16
17
             intege.
18
                       = s _mentbase + index;
19
             for i =
                   `tege. lemen* = SInt(Elem[operand1, 4 * e + i, esize DIV 4]);
20
                 1 :eger me _2 = UInt(Elem[operand2, 4 * s + i, esize DIV 4]);
21
                             .1ement1 * element2;
                 s n = sum
22
                       '' e, esize] = sum;
'___ = result;
23
             Elei
24
             c' ⊥[vec,
25
              vec + vstride;
        vec
```

D1.1.224 SUDOT (multiple and single vector)

Multi-vector signed by unsigned integer dot-product by vector

The instruction operates on two or four ZA single-vector groups.

The signed by unsigned integer dot product instruction computes the dot product of four signed 8-bit integer values held in each 32-bit element of the two or four first source vectors and four unsigned 8-bit integer values in the corresponding 32-bit element of the second source vector. The widened dot product result is destructively added to the corresponding 32-bit element of the two or four ZA single-vector groups. The vector numbers forming the single-vector group within each half or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The VECTOR GROUP symbol is preferred for disasser to prove optional in assembler source code.

This instruction is unpredicated.

It has encodings from 2 classes: Two ZA single-vectors and Fr ZA sing, vectors

Two ZA single-vectors (FEAT_SME2)



```
SUDOT ZA.S[<Wv>, <offs>{, '3x2}] { <Zn .B-<Zn2>.B }, <Zm>.B
```

```
1 if !HaveSME2() then UNDEF
```

- 2 integer v = UInt('010':' /;
- 3 constant integer esize = 32;
- 4 integer n = UInt(Zn)
- 5 integer m = UInt(' :Zm);
- 6 integer offset · JInt(of 3);
 7 separate integer
- 7 constant **inte**: nreg





SUDOT ZA.S[<Wv>, <offs>{, VGx4}], { <Zn1>.B-<Zn4>.B }, <Zm>.B

```
1 if !HaveSME2() then UNDEFINED;
```

```
2 integer v = UInt('010':Rv);
```

```
3 constant integer esize = 32;
```

```
4 integer n = UInt(Zn);
```

```
5 integer m = UInt('0':Zm);
6 integer affect UInt(aff
```

```
6 integer offset = UInt(off3);
7 constant integer nreg = 4;
```

Assembler Symbols

- <Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
- <offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.
- <Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn".

off3

- <Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" plus 3 modulo 32.
- <Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" plus 1 modulo 32.
- <Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

```
1
    CheckStreamingSVEAndZAEnabled();
    constant integer VL = CurrentVL;
2
3
    constant integer elements = VL DIV esize;
4
   integer vectors = VL DIV 8;
    integer vstride = vectors DIV nreg;
 5
 6
   bits(32) vbase = X[v, 32];
    integer vec = (UInt(vbase) + offset) MOD vstride;
7
8
    bits(VL) result;
9
10
   for r = 0 to nreg-1
11
        bits(VL) operand1 = Z[(n+r) MOD 32, VL];
        bits(VL) operand2 = Z[m, VL];
12
13
        bits(VL) operand3 = ZAvector[vec, VL];
14
        for e = 0 to elements-1
15
             bits(esize) sum = Elem[operand3, e, esize
16
             for i = 0 to 3
                 integer element1 = SInt(Elem / Grow '1,
integer element2 = UInt(Ele operar 2,
17
                                                                          esize DIV 4]);
18
                                                                         esize DIV 4]);
                                                             4
19
                 sum = sum + element1 * el .ent2;
        Elem[result, e, esize] = sum;
ZAvector[vec, VL] = result;
20
21
22
        vec = vec + vstride;
```

D1.1.225 SUMLALL (multiple and indexed vector)

Multi-vector signed by unsigned integer multiply-add long long by indexed element

The instruction operates on one, two, or four ZA quad-vector groups.

This signed by unsigned integer multiply-add long long instruction multiplies each signed 8-bit element in the one, two, or four first source vectors with each unsigned 8-bit indexed element of the second source vector, widens each product to 32-bits and destructively adds these values to the corresponding 32-bit elements of the one, two, or four ZA quad-vector groups.

The elements within the second source vector are specified using an immediate element index which selects the same element position within each 128-bit vector segment. The element index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 4 bits. The lowest of the four consecutive vector numbers forming the quad-vector group within all, each half, or each quarter of the 7 are selected by the sum of the vector select register and immediate offset, modulo all, half, or quarter ' _ number 'ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA opend consists of yo or four ZA quad-vector groups respectively. The VECTOR GROUP symbol is preferred f disass bly, but or lonal in assembler source code.

This instruction is unpredicated.

It has encodings from 3 classes: One ZA quad-vector Two 2 quad-vector and Four ZA quad-vectors

```
One ZA quad-vector
(FEAT_SME2)
```



```
if !HaveSME2() t<sup>1</sup> . UNDEFINED;
1
2
  constant integr esize = 32;
  integer v = U1 ('010' .v);
3
4
   integer n = UInt
                     יר
                        ·Zm);
5
  integer m
               VInt (
                               00');
6
  intege offs.
                   = UIn. off
   inter _ index = UInt(1
7
                             _41);
  cor ant int /er nreg = 1;
8
```

```
Two ZA 'ad-vectors
(FEAT_SN. ?)
```



SUMLALL ZA.S[<Wv>, <offsf>:<offsl>{, VGx2}], { <Zn1>.B-<Zn2>.B }, <Zm>.B[<index>]

```
1 if !HaveSME2() then UNDEFINED;
```

```
2 constant integer esize = 32;
```

```
3 integer v = UInt('010':Rv);
```

```
4 integer n = UInt(Zn:'0');
5 integer m = UInt('0':Zm);
```

```
5 integer m = UInt('0':Zm);
6 integer offset = UInt(o1:'00')
```

```
6 integer offset = UInt(o1:'00');
7 integer index = UInt(i4h:i4l);
```

```
8 constant integer nreg = 2;
```

Four ZA quad-vectors (FEAT_SME2) 1 0 0 0 0 0 1 0 Zm Rv Zn 0 0 1 1 0 i4h i4l SUMLALL ZA.S[<Wv>, <offsf>:<offsl>{, VGx4}], { <Zn1>.B-<Zn4>.B }, <Zm>.B[<index>] 1 if !HaveSME2() then UNDEFINED; 2 constant integer esize = 32; integer v = UInt('010':Rv); 3 4 integer n = UInt(Zn:'00'); integer m = UInt('0':Zm); 5 integer offset = UInt(o1:'00'); 6 integer index = UInt(i4h:i4l); constant **integer** nreg = 4; **Assembler Symbols** / field. $\langle Wv \rangle$ Is the 32-bit name of the vector select register W8.³ 11, encc d in the <offsf> For the one ZA quad-vector variant: is the vector le offset, printing to first of four consecutive vectors, encoded as "off2" field s 4. For the four ZA quad-vectors and two A quad- ectors va *c*: is the vector select offset, pointing to first of four consecutive vertors, en vert <offsl> For the one ZA quad-vector v. ant: is vector select offset, pointing to last of four consecutive vectors, encoder' as "o " field res 4 plus 3. For the four ZA quad-v ors and two \quad vectors variant: is the vector select offset, pointing to last of for con. outive vector. ncoded as "o1" field times 4 plus 3. Is the name of the first source _____able vector register, encoded in the "Zn" field. $\langle Zn \rangle$ $\langle Zn1 \rangle$ For the two A quad-vectors variant: is the name of the first scalable vector register of a multi-vec c sequer , encoded as "Zn" times 2. For the four *** uad-vect s variant: is the name of the first scalable vector register of a tor set, nce, ϵ loded as "Zn" times 4. m <Zn4>is the nr le of the full in scalable vector register of a multi-vector sequence, encoded as "Zn" times ls of name of the second scalable vector register of a multi-vector sequence, encoded as <Zn2>"Zn nes 2 plus 1.

<Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

<index> Is the element index, in the range 0 to 15, encoded in the "i4h:i4l" fields.

```
CheckStreamingSVEAndZAEnabled();
1
2
   constant integer VL = CurrentVL;
   constant integer elements = VL DIV esize;
3
4
   integer vectors = VL DIV 8;
   integer vstride = vectors DIV nreg;
5
   integer eltspersegment = 128 DIV esize;
6
   bits(32) vbase = X[v, 32];
7
   integer vec = (UInt(vbase) + offset) MOD vstride;
8
   bits(VL) result;
9
10
   vec = vec - (vec MOD 4);
11
12 for r = 0 to nreg-1
```

Chapter D1. SME instructions D1.1. SME and SME2 data-processing instructions

```
bits(VL) operand1 = Z[n+r, VL];
13
14
        bits(VL) operand2 = Z[m, VL];
15
        for i = 0 to 3
16
             bits(VL) operand3 = ZAvector[vec + i, VL];
17
             for e = 0 to elements-1
18
                  integer segmentbase = e - (e MOD eltspersegment);
                 integer s = 4 * segmentbase + index;
19
                 integer element1 = SInt(Elem[operand1, 4 * e + i, esize DIV 4]);
integer element2 = UInt(Elem[operand2, s, esize DIV 4]);
20
21
22
                 bits(esize) product = (element1 * element2)<esize-1:0>;
23
                 Elem[result, e, esize] = Elem[operand3, e, esize] + product;
24
             ZAvector[vec + i, VL] = result;
25
        vec = vec + vstride;
```

D1.1.226 SUMLALL (multiple and single vector)

Multi-vector signed by unsigned integer multiply-add long long by vector

The instruction operates on two or four ZA quad-vector groups.

This signed by unsigned integer multiply-add long long instruction multiplies each signed 8-bit element in the two or four first source vectors with each unsigned 8-bit element in the second source vector, widens each product to 32-bits and destructively adds these values to the corresponding 32-bit elements of the two or four ZA quad-vector groups. The lowest of the four consecutive vector numbers forming the quad-vector group within each half or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA quad-vector groups respectively. The VECTOR GROUP symbol is preferred for disasser to provide the provide the preferred for disasser to provide the provide the preferred for disasser to provide the preferred for disasser to provide the providet the providet the providet the provide the provide

This instruction is unpredicated.

It has encodings from 2 classes: Two ZA quad-vectors and For ZA quad- ctors

Two ZA quad-vectors (FEAT_SME2)



```
4 integer n = UInt(Zn);
```

```
5 integer m = UInt('0':Zm);
```

```
6 integer offset = UInt(o1:'00');
```

```
7 constant integer nreg = 4;
```

Assembler Symbols

- <Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
- <offsf> Is the vector select offset, pointing to first of four consecutive vectors, encoded as "o1" field times 4.

- <offsl> Is the vector select offset, pointing to last of four consecutive vectors, encoded as "o1" field times 4 plus 3.
- <Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn".
- <Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" plus 3 modulo 32.
- <Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" plus 1 modulo 32.
- <Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

```
1
    CheckStreamingSVEAndZAEnabled();
    constant integer VL = CurrentVL;
 2
 3
    constant integer elements = VL DIV esize;
    integer vectors = VL DIV 8;
 4
 5
    integer vstride = vectors DIV nreg;
 6
    bits(32) vbase = X[v, 32];
    integer vec = (UInt(vbase) + offset) MOD vstride;
 7
 8
    bits(VL) result;
 9
    vec = vec - (vec MOD 4);
10
11
    for r = 0 to nreg-1
12
          bits(VL) operand1 = Z[(n+r) MOD 32],
          bits(VL) operand2 = Z[m, VL];
13
14
          for i = 0 to 3
               bits(VL) operand3 = ZAvector[
15
16
               for e = 0 to elements-1
                    integer element1 = SIL 'Elem[\ rand1, 4 * e + i, esize DIV 4]);
integer element2 JInt em[op nd2, 4 * e + i, esize DIV 4]);
bits(esize) prc ct = (el nt1 * lement2)<esize-1:0>;
Elem[result, size] = El operand3, e, esize] + product;
17
18
19
                    Elem[result,
20
                                      VL.
21
               ZAvector[vec +
                                            result;
22
          vec = vec + vstride,
```

D1.1.227 SUMOPA

Signed by unsigned integer sum of outer products and accumulate

The 8-bit integer variant works with a 32-bit element ZA tile.

The 16-bit integer variant works with a 64-bit element ZA tile.

The signed by unsigned integer sum of outer products and accumulate instructions multiply the sub-matrix in the first source vector by the sub-matrix in the second source vector. In case of the 8-bit integer variant, the first source holds $SVL_S \times 4$ sub-matrix of signed 8-bit integer values, and the second source holds $4 \times SVL_S$ sub-matrix of unsigned 8-bit integer values. In case of the 16-bit integer variant, the first source holds $SVL_D \times 4$ sub-matrix of signed 16-bit integer values, and the second source holds $4 \times SVL_D \times 4$ sub-matrix of signed 16-bit integer values.

Each source vector is independently predicated by a corresponding governing predicate. When an 8-bit source element in case of 8-bit integer variant or a 16-bit source element in case 10-c. reger variant is Inactive, it is treated as having the value 0.

The resulting $SVL_S \times SVL_S$ widened 32-bit integer or $SVL_D \times SVL_D$ where d64-bit in ger sum of outer products is then destructively added to the 32-bit integer or 64-bit integer destination ile, resp. dively for 8-bit integer and 16-bit integer instruction variants. This is equivalent to perfer hing a 4-way derived and accumulate to each of the destination tile elements.

In case of the 8-bit integer variant, each 32-bit container of first sour vector holds 4 consecutive column elements of each row of a $SVL_S \times 4$ sub-matrix and the h 32-b. contrast of the second source vector holds 4 consecutive row elements of each column of a f SVL_S sub-matrix. The second source vector holds 4 consecutive row of the first source vector holds 4 consecutive row of a $SVL_D \times 4$ sub-matrix, and each 64-bit container of the second source vector holds 4 consecutive row elements of each column of a $4 \times SVL_S$ sub-matrix.

ID_AA64SMFR0_EL1.I16I64 *i* ________ cates whe. r the 1______ bit integer variant is implemented.

It has encodings from 2 classics, bit and 64-bit



```
SUMOPA <ZAda>.D, <Pn>/M, <Pm>/M, <Zn>.H, <Zm>.H
1
   if !HaveSMEI16I64() then UNDEFINED;
2
   constant integer esize = 64;
  integer a = UInt(Pn);
3
4
   integer b = UInt(Pm);
5
   integer n = UInt(Zn);
  integer m = UInt(Zm);
6
   integer da = UInt(ZAda);
7
8
   boolean sub_op = FALSE;
9
   boolean op1_unsigned = FALSE;
10 boolean op2_unsigned = TRUE;
   Assembler Symbols
```

- <ZAda> For the 32-bit variant: is the name of the ZA tile ZA0-ZA3, encoded in the "ZAda" field.
 - For the 64-bit variant: is the name of the ZA tile ZA0-ZA7, enclued in the Ada" field.
 - <Pn> Is the name of the first governing scalable predicate register P0-. encoded in t "Pn" field.
 - <Pm> Is the name of the second governing scalable predicate _gister P0-, encod in the "Pm" field.
 - <Zn> Is the name of the first source scalable vector regis. enc. ed in the "Zn" field.
 - <Zm> Is the name of the second source scalable y registe 'ncode' in the "Zm" field.

```
1
   CheckStreamingSVEAndZAEnabled();
2
   constant integer VL = CurrentVL;
3
   constant integer PL = VL DIV
   constant integer dim = VL D
4
                                 esize;
5
   bits(PL) mask1 = P[a, PL]
   bits(PL) mask2 = P[b, P'
6
7
   bits(VL) operand1 = Z[n, VL];
8
   bits(VL) operand2 = [m, VL];
   bits(dim*dim*esize operand3 = ZAt >[da, esize, dim*dim*esize];
9
10
   bits(dim*dim*es' .) resu't;
11
   integer prod
12
13
   for row = 0 to d_{\perp}
14
        for
                 0 to
                        m-1
15
           bits ( ize)
                              _lem[operand3, row*dim+col, esize];
                   0 to 3
16
            for }
17
                       wePredicateElement(mask1, 4*row + k, esize DIV 4) &&
18
                        ActivePredicateElement(mask2, 4*col + k, esize DIV 4) then
19
                    prod = (Int(Elem[operand1, 4*row + k, esize DIV 4], op1_unsigned) *
                            Int(Elem[operand2, 4*col + k, esize DIV 4], op2_unsigned));
20
21
                    if sub_op then prod = -prod;
22
                    sum = sum + prod;
23
24
            Elem[result, row*dim+col, esize] = sum;
25
26
   ZAtile[da, esize, dim*dim*esize] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
 - The values of the data supplied in any of its operand registers when its governing predicate registers contain the same value for each execution.
 - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:

- The values of the data supplied in any of its operand registers when its governing predicate registers contain the same value for each execution.
- The values of the NZCV flags.



D1.1.228 SUMOPS

Signed by unsigned integer sum of outer products and subtract

The 8-bit integer variant works with a 32-bit element ZA tile.

The 16-bit integer variant works with a 64-bit element ZA tile.

The signed by unsigned integer sum of outer products and subtract instructions multiply the sub-matrix in the first source vector by the sub-matrix in the second source vector. In case of the 8-bit integer variant, the first source holds $SVL_S \times 4$ sub-matrix of signed 8-bit integer values, and the second source holds $4 \times SVL_S$ sub-matrix of unsigned 8-bit integer values. In case of the 16-bit integer variant, the first source holds $SVL_D \times 4$ sub-matrix of signed 16-bit integer values, and the second source holds $4 \times SVL_D \times 4$ sub-matrix of signed 16-bit integer values.

Each source vector is independently predicated by a corresponding governing predicate. When an 8-bit source element in case of 8-bit integer variant or a 16-bit source element in case 10-c. reger variant is Inactive, it is treated as having the value 0.

The resulting $SVL_S \times SVL_S$ widened 32-bit integer or $SVL_D \times SVL_D$ where d64-bit in ger sum of outer products is then destructively subtracted from the 32-bit integer or 64 at integer stination dile, respectively for 8-bit integer and 16-bit integer instruction variants. This is equivalent to perform, a vary dot product and subtract from each of the destination tile elements.

In case of the 8-bit integer variant, each 32-bit container of first sour vector holds 4 consecutive column elements of each row of a $SVL_S \times 4$ sub-matrix and the 32-bit container of the second source vector holds 4 consecutive row elements of each column of a $4 \times SVL_S$ sub-matrix. The second source vector holds 4 consecutive row of a $SVL_D \times 4$ sub-matrix, and each 64-bit container of the second source vector holds 4 consecutive row elements of each column of a $4 \times SVL_S$ sub-matrix.

ID_AA64SMFR0_EL1.I16I64 *i* ________ cates whe. r the 1______ bit integer variant is implemented.

It has encodings from 2 classics, bit and 64-bit



```
SUMOPS <ZAda>.D, <Pn>/M, <Pm>/M, <Zn>.H, <Zm>.H
1
   if !HaveSMEI16I64() then UNDEFINED;
2
   constant integer esize = 64;
  integer a = UInt(Pn);
3
4
   integer b = UInt(Pm);
5
   integer n = UInt(Zn);
  integer m = UInt(Zm);
6
   integer da = UInt(ZAda);
7
8
   boolean sub_op = TRUE;
9
   boolean op1_unsigned = FALSE;
10 boolean op2_unsigned = TRUE;
   Assembler Symbols
```

- <ZAda> For the 32-bit variant: is the name of the ZA tile ZA0-ZA3, encoded in the "ZAda" field. For the 64-bit variant: is the name of the ZA tile ZA0-ZA7, enc., ed in the Ada" field.
 - For the 64-bit variant: is the name of the ZA tile ZA0-ZA7, enc. led in the Ada" field. <Pn> Is the name of the first governing scalable predicate register P0-. encoded in t "Pn" field.
 - <Pm> Is the name of the second governing scalable predicate _gister P0-, encod _ in the "Pm"
 - <Zn> Is the name of the first source scalable vector regis. enc. ed in the "Zn" field.
 - <Zm> Is the name of the second source scalable y register ncode? In the "Zm" field.

field.

```
1
   CheckStreamingSVEAndZAEnabled();
2
   constant integer VL = CurrentVL;
3
   constant integer PL = VL DIV
   constant integer dim = VL D
4
                                 esize;
5
   bits(PL) mask1 = P[a, PL]
   bits(PL) mask2 = P[b, P'
6
7
   bits(VL) operand1 = Z[n, VL];
8
   bits(VL) operand2 = [m, VL];
   bits(dim*dim*esize operand3 = ZAt >[da, esize, dim*dim*esize];
9
10
   bits(dim*dim*es' .) resu't;
11
   integer prod
12
13
   for row = 0 to d_{\perp}
14
        for
                 0 to
                        m-1
15
           bits ( ize)
                              _lem[operand3, row*dim+col, esize];
                   0 to 3
16
            for }
17
                       wePredicateElement(mask1, 4*row + k, esize DIV 4) &&
18
                        ActivePredicateElement(mask2, 4*col + k, esize DIV 4) then
19
                    prod = (Int(Elem[operand1, 4*row + k, esize DIV 4], op1_unsigned) *
                            Int(Elem[operand2, 4*col + k, esize DIV 4], op2_unsigned));
20
21
                    if sub_op then prod = -prod;
22
                    sum = sum + prod;
23
24
            Elem[result, row*dim+col, esize] = sum;
25
26
   ZAtile[da, esize, dim*dim*esize] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
 - The values of the data supplied in any of its operand registers when its governing predicate registers contain the same value for each execution.
 - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:

- The values of the data supplied in any of its operand registers when its governing predicate registers contain the same value for each execution.
- The values of the NZCV flags.



D1.1.229 SUNPK

Unpack and sign-extend multi-vector elements

Unpack elements from one or two source vectors and then sign-extend them to place in elements of twice their size within the two or four destination vectors.

This instruction is unpredicated.

It has encodings from 2 classes: Two registers and Four registers

Two registers (FEAT_SME2)



Assembler Symbols

<Zd1> For the two registers variant: is the name of the first destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 2.

For the four registers variant: is the name of the first destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 4.

<T> Is the size specifier, encoded in "size":

size	<t></t>
00	RESERVED
01	Н
10	S
11	D

- <Zd4> Is the name of the fourth destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 4 plus 3.
- <Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.
- <Zd2> Is the name of the second destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 2 plus 1.
- <Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.
- <Tb> Is the size specifier, encoded in "size":

size	< Tb >
00	RESERVED
01	В
10	Н
11	S

<Zn2> Is the name of the second scalable vector register of *r*_nulti-vector requence encoded as "Zn" times 2 plus 1.

```
CheckStreamingSVEEnabled();
1
2
   constant integer VL = CurrentVL;
3
   constant integer elements = VL DIV es
                                             .e;
4
   constant integer hsize = esize DIV 2;
5
   constant integer sreg = nreg DIV
   array [0..3] of bits(VL) results;
6
7
8
   for r = 0 to sreg-1
9
        bits(VL) operand = Z<sup>f</sup>
                                   VL];
10
        for i = 0 to 1
11
            for e = 0 to elements-
                bits(hs e) element
12
                                         'lem[operand, i*elements + e, hsize];
13
                Elem[ sults[2*r+i], e, size] = Extend(element, esize, unsigned);
14
   for r = 0 to :
15
                    ∙g−1
16
        Z[d+r, VL]
                      res
                           cs[r];
```

D1.1.230 SUVDOT

Multi-vector signed by unsigned integer vertical dot-product by indexed element

The instruction operates on four ZA single-vector groups.

The signed by unsigned integer vertical dot product instruction computes the vertical dot product of the corresponding signed 8-bit elements from the four first source vectors and four unsigned 8-bit integer values in the corresponding indexed 32-bit element of the second source vector. The widened dot product result is destructively added to the corresponding 32-bit element of the four ZA single-vector groups.

The groups within the second source vector are specified using an immediate element index which selects the same group position within each 128-bit vector segment. The index range is from 0 to 3, encoded in 2 bits.

The vector numbers forming the single-vector group within each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo quarter the number of 2t, ray vectors.

The VECTOR GROUP symbol VGx4 indicates that the ZA operand constst of four Z2 single-vector groups. The VECTOR GROUP symbol is preferred for disassembly, but optional in a mbler source code.

This instruction is unpredicated.

SME2 (FEAT_SME2)

0 0 0 0 0 1 1 0 1 0 **7π** Rv 7n 0 1 1 1 off3

SUVDOT ZA.S[<Wv>, <offs>{, 3x4}], <Zn1 B-<Zn4>.B }, <Zm>.B[<index>]

```
1 if !HaveSME2() then UNDF M
2 integer v = UInt('010'. v);
3 constant integer esi 2 = 32;
4 integer n = UInt(7 00');
5 integer m = UInt J':Zm);
6 integer offset UInt(c 3);
```

```
7 integer index "Int( );
```

Assemb' bols

```
<Wv> Is the 3' sit name o. .ne vector select register W8-W11, encoded in the "Rv" field.
```

- <offs> t¹ vector select offset, in the range 0 to 7, encoded in the "off3" field.
- <Zn1> Is a name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times
- <Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.
- <Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

<index> Is the element index, in the range 0 to 3, encoded in the "i2" field.

```
CheckStreamingSVEAndZAEnabled();
1
  constant integer VL = CurrentVL;
2
3
  constant integer elements = VL DIV esize;
4
  integer vectors = VL DIV 8;
  integer vstride = vectors DIV 4;
5
6
  integer eltspersegment = 128 DIV esize;
7
  bits(32) vbase = X[v, 32];
  integer vec = (UInt(vbase) + offset) MOD vstride;
8
```

```
Chapter D1. SME instructions
D1.1. SME and SME2 data-processing instructions
```

```
9 bits(VL) operand2 = Z[m, VL];
10 bits(VL) result;
11
12
    for r = 0 to 3
13
        bits(VL) operand3 = ZAvector[vec, VL];
14
         for e = 0 to elements-1
15
            integer segmentbase = e - (e MOD eltspersegment);
             integer s = segmentbase + index;
16
17
             bits(esize) sum = Elem[operand3, e, esize];
18
             for i = 0 to 3
19
                 bits(VL) operand1 = Z[n+i, VL];
                 integer element1 = SInt(Elem[operand1, 4 * e + r, esize DIV 4]);
integer element2 = UInt(Elem[operand2, 4 * s + i, esize DIV 4]);
20
21
22
                 sum = sum + element1 * element2;
23
             Elem[result, e, esize] = sum;
24
         ZAvector[vec, VL] = result;
25
        vec = vec + vstride;
```

D1.1.231 SVDOT (2-way)

Multi-vector signed integer vertical dot-product by indexed element

The instruction operates on two ZA single-vector groups.

The signed integer vertical dot product instruction computes the vertical dot product of the corresponding two signed 16-bit integer values held in the two first source vectors and two signed 16-bit integer values in the corresponding indexed 32-bit element of the second source vector. The widened dot product results are destructively added to the corresponding 32-bit element of two ZA single-vector groups.

The groups within the second source vector are specified using an immediate element index which selects the same group position within each 128-bit vector segment. The index range is from 0 to 3, encoded in 2 bits.

The vector numbers forming the single-vector group within each half of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half the number c __A an. vectors.

The VECTOR GROUP symbol VGx2 indicates that the ZA operand c sists of two Z single-vector groups. The VECTOR GROUP symbol is preferred for disassembly, but optional in a mbler source code.

This instruction is unpredicated.

SME2 (FEAT_SME2)

0 0 0 0 0 1 1 0 1 0 1 **7π** Rv 7n 1 0 0 off3 _11

SVDOT ZA.S[<Wv>, <offs>{ <Znl H-<Zn2>.H }, <Zm>.H[<index>] 4x2

```
1
  if !HaveSME2() then UNDF
  integer v = UInt('010'...v);
2
   constant integer esi ? = 32;
3
  integer n = UInt(7
                        0');
4
  integer m = UInt J':Zm);
5
6
  integer offset
                    UInt (c
                            3);
                          1:
```

```
7
  integer index
                   Unt(
```

Assemb' bols

```
Is the 3<sup>7</sup> bit name of the vector select register W8-W11, encoded in the "Rv" field.
\langle Wv \rangle
```

- t¹ vector second offset, in the range 0 to 7, encoded in the "off3" field. <offs>
- <Zn1>Is i. name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times .
- <Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.
- Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field. <Zm>

<index> Is the element index, in the range 0 to 3, encoded in the "i2" field.

```
CheckStreamingSVEAndZAEnabled();
1
  constant integer VL = CurrentVL;
2
3
  constant integer elements = VL DIV esize;
4
  integer vectors = VL DIV 8;
  integer vstride = vectors DIV 2;
5
6
  integer eltspersegment = 128 DIV esize;
7
  bits(32) vbase = X[v, 32];
  integer vec = (UInt(vbase) + offset) MOD vstride;
8
```

```
Chapter D1. SME instructions
D1.1. SME and SME2 data-processing instructions
```

```
9 bits(VL) operand2 = Z[m, VL];
10 bits(VL) result;
11
12
    for r = 0 to 1
13
        bits(VL) operand3 = ZAvector[vec, VL];
14
         for e = 0 to elements-1
15
            integer segmentbase = e - (e MOD eltspersegment);
             integer s = segmentbase + index;
16
17
             bits(esize) sum = Elem[operand3, e, esize];
18
             for i = 0 to 1
19
                 bits(VL) operand1 = Z[n+i, VL];
                 integer element1 = SInt(Elem[operand1, 2 * e + r, esize DIV 2]);
integer element2 = SInt(Elem[operand2, 2 * s + i, esize DIV 2]);
20
21
22
                 sum = sum + element1 * element2;
23
             Elem[result, e, esize] = sum;
24
         ZAvector[vec, VL] = result;
25
        vec = vec + vstride;
```

D1.1.232 SVDOT (4-way)

Multi-vector signed integer vertical dot-product by indexed element

The instruction operates on four ZA single-vector groups.

The signed integer vertical dot product instruction computes the vertical dot product of the corresponding four signed 8-bit or 16-bit integer values held in the four first source vectors and four signed 8-bit or 16-bit integer values in the corresponding indexed 32-bit or 64-bit element of the second source vector. The widened dot product results are destructively added to the corresponding 32-bit or 64-bit element of the four ZA single-vector groups.

The groups within the second source vector are specified using an immediate element index which selects the same group position within each 128-bit vector segment. The index range is from 0 to one less than the number of groups per 128-bit segment, encoded in 1 to 2 bits depending on the size of the group.

The vector numbers forming the single-vector group within each quarter $o^{c} = z_{c}$, and are selected by the sum of the vector select register and immediate offset, modulo quarter the nur ser of ZA a. vectors.

The VECTOR GROUP symbol VGx4 indicates that the ZA operand con. 's of four ZA ingle-vector groups. The VECTOR GROUP symbol is preferred for disassembly, but optir ... in asse. 1er sour code.

This instruction is unpredicated.

ID_AA64SMFR0_EL1.I16I64 indicates whether the 16-but 'er', variant i implemented.

It has encodings from 2 classes: 32-bit and 64-bi

32-bit (FEAT SME2)

1

6

7

```
13
                 0
                    0
                       0
                          0
                             0
                                                         1
                                                             Rv
                                                                             7n
                                                                                     1
                                                                                        0
                                                                                               off3
                                1
                                   0
                                        ß
                                                  7m
                                                                 0
                                                                     i2
                                                                                  Θ
                                                                                           0
                                                                                        Lu
                          offs>{, VGx.
   SVDOT ZA.S[<Wv>
                                              { <Zn1>.B-<Zn4>.B }, <Zm>.B[<index>]
   if !HaveSME2() nen UND INED;
integer v = UI. ''010 Rv);
2
3
   constant integer
                          e = 32;
                  Tnt(Z. '00');
4
   integer
   intege in = l it('0'
5
   inte r offse = UInt(
                               .3);
   int or ind
                      ™nt (i2):
   64-bit
   (FEAT_SM.
                 U16I64)
                  0
                    0
                       0
                          0
                             0
                               1
                                  1
                                     1
                                        0
                                                  Zm
                                                             Rv
                                                                                        0
                                           1
                                                                  0
                                                                       i1
                                                         1
                                                                    1
   SVDOT ZA.D[<Wv>, <offs>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H[<index>]
```

```
1
  if !(HaveSME2() && HaveSMEI16I64()) then UNDEFINED;
```

```
2
  integer v = UInt('010':Rv);
```

```
3
  constant integer esize = 64;
```

```
4
  integer n = UInt(Zn:'00');
5
  integer m = UInt('0':Zm);
```

```
6
```

```
integer offset = UInt(off3);
  integer index = UInt(i1);
7
```

```
DDI0616
B.a
```

Assembler Symbols

- <Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
- <offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.
- <Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.
- <Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.
- <Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

<index> For the 32-bit variant: is the element index, in the range 0 to 3, encoded in the "i2" field.

For the 64-bit variant: is the element index, in the range 0 to 1, encoded in the "i1" field.

```
1
   CheckStreamingSVEAndZAEnabled();
   constant integer VL = CurrentVL;
2
3
   constant integer elements = VL DIV esize;
4
   integer vectors = VL DIV 8;
   integer vstride = vectors DIV 4;
5
6
   integer eltspersegment = 128 DIV esize;
 7
   bits(32) vbase = X[v, 32];
 8
   integer vec = (UInt(vbase) + offset) MOD
                                                stri
9
   bits(VL) operand2 = Z[m, VL];
10
   bits(VL) result;
11
12
    for r = 0 to 3
13
        bits(VL) operand3 = ZAvector[v
                                            VL];
14
        for e = 0 to elements-1
15
            integer segmentbas
                                   e - (e M
                                               eltsp rsegment);
16
                                 se + index;
            integer s = seqm
                               Ele operand3,
17
            bits(esize) sur
                                               e. esize];
18
            for i = 0 to 3
                bits(VI operand1 = 2 +i, VL];
inter . element1 = SInt lem[operand1, 4 * e + r, esize DIV 4]);
19
20
21
                int ger eler it2 = SInt(Elem[operand2, 4 * s + i, esize DIV 4]);
22
                su = sum element1 * element2;
23
            Elem[res
                           ., esize' = sum;
                           = res
24
        ZAvec
                 vec.
25
        vec
                    vstı 'ə;
              VA.
```

D1.1.233 UCLAMP

Multi-vector unsigned clamp to minimum/maximum vector

Clamp each unsigned element in the two or four destination vectors to between the unsigned minimum value in the corresponding element of the first source vector and the unsigned maximum value in the corresponding element of the second source vector and destructively place the clamped results in the corresponding elements of the two or four destination vectors.

This instruction is unpredicated.

It has encodings from 2 classes: Two registers and Four registers



Assembler Symbols

<Zd1> For the two registers variant: is the name of the first destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 2.

For the four registers variant: is the name of the first destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 4.

<T> Is the size specifier, encoded in "size":

size	< T >
00	В
01	Н
10	S
11	D

- <Zd4> Is the name of the fourth destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 4 plus 3.
- <Zd2> Is the name of the second destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 2 plus 1.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
- <Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

```
CheckStreamingSVEEnabled();
1
2
   constant integer VL = CurrentVL;
   constant integer elements = VL DIV esize;
3
4
   array [0..3] of bits(VL) results;
5
   for r = 0 to nreg-1
6
7
       bits(VL) operand1 = Z[n, VL];
8
       bits(VL) operand2 = Z[m, VL];
       bits(VL) operand3 = Z[d+r, VL];
9
10
       for e = 0 to elements-1
                                                       size]);
11
            integer element1 = UInt(Elem[operand1, e,
            integer element2 = UInt(Elem[operand2,
12
                                                       esizel
            integer element3 = UInt(Elem[operand3, e, siz );
13
14
            integer res = Min(Max(element1, element3),
                                                           ment.2):
15
            Elem[results[r], e, esize] = res
                                                     :0>;
16
   for r = 0 to nreg-1
17
18
       Z[d+r, VL] = results[r];
```

D1.1.234 UCVTF

Multi-vector unsigned integer convert to floating-point

Convert to single-precision from unsigned 32-bit integer, each element of the two or four source vectors, and place the results in the corresponding elements of the two or four destination vectors.

This instruction follows SME2 floating-point numerical behaviors corresponding to instructions that place their results in one or more SVE Z vectors.

This instruction is unpredicated.

It has encodings from 2 classes: Two registers and Four registers

Two registers (FEAT_SME2)



Assembler Symbols

<Zd1> For the two registers variant: is the name of the first destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 2.

For the four registers variant: is the name of the first destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 4.

- <Zd4> Is the name of the fourth destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 4 plus 3.
- <Zd2> Is the name of the second destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 2 plus 1.
- <Zn1> For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.

For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

- <Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.
- <Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.

```
CheckStreamingSVEEnabled();
1
2
   constant integer VL = CurrentVL;
3
   constant integer elements = VL DIV 32;
4
   array [0..3] of bits(VL) results;
5
6
   for r = 0 to nreg-1
7
       bits(VL) operand = Z[n+r, VL];
       for e = 0 to elements-1
8
           bits(32) element = Elem[operand, e, 32];
9
10
           Elem[results[r], e, 32] = FixedToFP(element,
                                                             unsig
                                                                       FPCR
                                                                                rounding, 32);
11
   for r = 0 to nreg-1
12
13
      Z[d+r, VL] = results[r];
```

D1.1.235 UDOT (2-way, multiple and indexed vector)

Multi-vector unsigned integer dot-product by indexed element

The instruction operates on two or four ZA single-vector groups.

The unsigned integer dot product instruction computes the dot product of two unsigned 16-bit integer values held in each 32-bit element of the two or four first source vectors and two unsigned 16-bit integer values in the corresponding indexed 32-bit element of the second source vector. The widened dot product result is destructively added to the corresponding 32-bit element of the two or four ZA single-vector groups.

The groups within the second source vector are specified using an immediate element index which selects the same group position within each 128-bit vector segment. The index range is from 0 to 3, encoded in 2 bits. The vector numbers forming the single-vector group within each half or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the the of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA oper disconsists of voor four ZA single-vector groups respectively. The VECTOR GROUP symbol is preferred for disconsists of the vector on a sembler source code.

This instruction is unpredicated.

It has encodings from 2 classes: Two ZA single-vectors a. Four ' A single-vectors

Two ZA single-vectors (FEAT_SME2)



UDOT ZA.S[<Wv>, <offs>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H[<index>]

```
1 if !HaveSME2() then UNDEFINED;
```

```
2 integer v = UInt('010':Rv);
```

```
3 constant integer esize = 32;
4 integer n = UInt(Zn:'00');
```

```
4 integer n = UInt(Zn:'00');
5 integer m = UInt('0':Zm);
```

```
6 integer offset = UInt(off3);
```

```
7 integer index = UInt(i2);
```

```
8 constant integer nreg = 4;
```

Assembler Symbols

- <Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
- <offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.
- <Zn1> For the two ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.

For the four ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

- <Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.
- <Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.
- <Zm> Is the name of the second source scalable vector register Z0-Z['], encoded in "Zm" field.

<index> Is the element index, in the range 0 to 3, encoded in the "?" field

```
CheckStreamingSVEAndZAEnabled();
 1
 2
    constant integer VL = CurrentVL;
    constant integer elements = VL DIV esize;
 3
4
    integer vectors = VL DIV 8;
    integer vstride = vectors DIV nreg;
 5
    integer eltspersegment = 128 DIV esiz
6
7
    bits(32) vbase = X[v, 32];
 8
    integer vec = (UInt(vbase) + offs
                                             MOD
                                                     ride;
9
    bits(VL) result;
10
11
    for r = 0 to nreg-1
                              , nh
12
        bits(VL) operand1 =
                                      VL];
        bits(VL) operand2 = Z[m,
13
14
        bits(VL) operanc' = ZAvector
                                          ec, VL];
15
        for e = 0 to \epsilon .ments-1
            bits(esi ) sum = Elem[oper.d3, e, esize];
intege segment se = e - (e MOD eltspersegment);
16
17
             intege.
18
                       = s _mentbase + index;
19
             for i =
                           4
                   `tege. lemen* = UInt(Elem[operand1, 2 * e + i, esize DIV 2]);
20
                 1 :eger to me 2 = UInt(Elem[operand2, 2 * s + i, esize DIV 2]);
21
                             .lement1 * element2;
                 s n = sum
22
                       '' e, esize] = sum;
'___ = result;
23
             Elei
24
             c' ⊥[vec,
25
               vec + vstride;
        vec
```

D1.1.236 UDOT (2-way, multiple and single vector)

Multi-vector unsigned integer dot-product by vector

The instruction operates on two or four ZA single-vector groups.

The unsigned integer dot product instruction computes the dot product of two unsigned 16-bit integer values held in each 32-bit element of the two or four first source vectors and two unsigned 16-bit integer values in the corresponding 32-bit element of the second source vector. The widened dot product result is destructively added to the corresponding 32-bit element of the two or four ZA single-vector groups. The vector numbers forming the single-vector group within each half or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The VECTOR GROUP symbol is preferred for disasser to prove optional in assembler source code.

This instruction is unpredicated.

It has encodings from 2 classes: Two ZA single-vectors and Fr ZA sing, vectors

Two ZA single-vectors (FEAT_SME2)



UDOT ZA.S[<Wv>, <offs>{, 'x2}, { <Zh, .H-<Zn2>.H }, <Zm>.H

```
1 if !HaveSME2() then UNDEF
```

- 2 integer v = UInt('010':' /;
- 3 constant integer esize = 32;
- 4 integer n = UInt(Zn)
- 5 integer m = UInt(' :Zm); 6 integer offset = JInt(of 3);
- 6 integer offset · JInt(of 7 constant inter · nreg ·
- constant inter . nred



Zn

0 Rv

0

Zn

UDOT ZA.S[<Wv>, <offs>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H

1 0

```
1 if !HaveSME2() then UNDEFINED;
```

```
2 integer v = UInt('010':Rv);
```

```
3 constant integer esize = 32;
```

```
4 integer n = UInt(Zn);
```

```
5 integer m = UInt('0':Zm);
6 integer affect UInt(aff
```

```
6 integer offset = UInt(off3);
7 constant integer nreg = 4;
```

Assembler Symbols

- <Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
- <offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.
- <Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn".

off3

off3

1 1 Lu

- <Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" plus 3 modulo 32.
- <Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" plus 1 modulo 32.
- <Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

```
1
    CheckStreamingSVEAndZAEnabled();
    constant integer VL = CurrentVL;
2
 3
    constant integer elements = VL DIV esize;
   integer vectors = VL DIV 8;
4
    integer vstride = vectors DIV nreg;
 5
 6
   bits(32) vbase = X[v, 32];
    integer vec = (UInt(vbase) + offset) MOD vstride;
7
8
    bits(VL) result;
9
10
   for r = 0 to nreg-1
11
        bits(VL) operand1 = Z[(n+r) MOD 32, VL];
        bits(VL) operand2 = Z[m, VL];
12
13
        bits(VL) operand3 = ZAvector[vec, VL];
14
        for e = 0 to elements-1
15
             bits(esize) sum = Elem[operand3, e, esize
16
             for i = 0 to 1
                 integer element1 = UInt(Elem) ere '1,
integer element2 = UInt(Ele operar 2,
17
                                                                         esize DIV 2]);
                                                                         esize DIV 2]);
18
                                                             2
19
                 sum = sum + element1 * el .ent2;
        Elem[result, e, esize] = sum;
ZAvector[vec, VL] = result;
20
21
22
        vec = vec + vstride;
```
D1.1.237 UDOT (2-way, multiple vectors)

Multi-vector unsigned integer dot-product

The instruction operates on two or four ZA single-vector groups.

The unsigned integer dot product instruction computes the dot product of two unsigned 16-bit integer values held in each 32-bit element of the two or four first source vectors and two unsigned 16-bit integer values in the corresponding 32-bit element of the two or four second source vectors. The widened dot product result is destructively added to the corresponding 32-bit element of the two or four ZA single-vector groups. The vector numbers forming the single-vector group within each half or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The VECTOR GROUP symbol is preferred for disasser to prove optional in assembler source code.

This instruction is unpredicated.

It has encodings from 2 classes: Two ZA single-vectors and Fr ZA sing, vectors

Two ZA single-vectors (FEAT_SME2)

1

23

4

5 6

1

2

3

4



- 5 integer m = UInt(Zm:'00'); 6 integer offset = UInt(off3);
- 6 integer offset = UInt(off3); 7 constant integer nreg = 4;

Assembler Symbols

- <Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
- <offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.
- <Zn1> For the two ZA single-vectors variant: is the name of the first scalable vector register of a

multi-vector sequence, encoded as "Zn" times 2.

For the four ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

- <Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.
- <Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.
- <Zm1> For the two ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 2.

For the four ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 4.

- <Zm4> Is the name of the fourth scalable vector register of a multi-v_tor sequence encoded as "Zm" times 4 plus 3.
- <Zm2> Is the name of the second scalable vector register of a ________ sequency encoded as "Zm" times 2 plus 1.

```
CheckStreamingSVEAndZAEnabled();
1
   constant integer VL = CurrentVL;
2
3
   constant integer elements = VL DIV es;
   integer vectors = VL DIV 8;
4
   integer vstride = vectors DIV nreg;
5
6
   bits(32) vbase = X[v, 32];
   integer vec = (UInt(vbase) +
7
                                c fsei
                                       MOD
                                           v.
                                               ide:
   bits(VL) result;
8
9
10
   for r = 0 to nreg-1
       bits(VL) operand1 = 2[n+r, 'L];
11
12
       bits(VL) operand = Z[m+r,
       bits(VL) opera .3 = ZAvector[
13
                                        VL1:
14
       for e = 0 to lements-1
           bits(e ze) sur Elem[operand3, e, esize];
for i     to
15
16
               17
18
19
                          f .ment1 * element2;
                 ı = su
20
           Elem[ esult, e, size] = sum;
21
        Avecto<sup>.</sup>
                    VIJ = result;
22
                +
                   vs....de;
```

D1.1.238 UDOT (4-way, multiple and indexed vector)

Multi-vector unsigned integer dot-product by indexed element

The instruction operates on two or four ZA single-vector groups.

The unsigned integer dot product instruction computes the dot product of four unsigned 8-bit or 16-bit integer values held in each 32-bit or 64-bit element of the two or four first source vectors and four unsigned 8-bit or 16-bit integer values in the corresponding indexed 32-bit or 64-bit element of the second source vector. The widened dot product result is destructively added to the corresponding 32-bit or 64-bit element of the two or four ZA single-vector groups.

The groups within the second source vector are specified using an immediate element index which selects the same group position within each 128-bit vector segment. The index range is from 0 to one less than the number of groups per 128-bit segment, encoded in 1 to 2 bits depending on the simple group. The vector numbers forming the single-vector group within each half or each quarter of the A array selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA ray vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the Z operan onsists of t o or four ZA single-vector groups respectively. The VECTOR GROUP symbol is preferre for disassen. v, but ptional in assembler source code.

This instruction is unpredicated.

ID_AA64SMFR0_EL1.I16I64 indicates whether leng it integeneration is implemented.

It has encodings from 4 classes: Two ZA si gle-vectors of 32-bit elements, Two ZA single-vectors of 64-bit elements, Four ZA single-vectors of 32-bit elements are single-vectors of 64-bit elements

Two ZA single-vectors of 32-bit element (FEAT_SME2)

UDOT ZA.S[< >, < fs>{, VGx2}], { <Zn1>.B-<Zn2>.B }, <Zm>.B[<index>]

```
1
  if !Have
                then. WDEFI
2
  integr v = U
               it('01
                      R۰
  cons int inte esize
                         32;
3
4
                  ···'0');
  int. `r n =
  5
           fset = UInt(off3);
6
  integer
  integer 1. \circle x = UInt(i2);
7
```

```
8 constant in ver nreg = 2;
```

Two ZA single-vectors of 64-bit elements (FEAT_SME_I16I64)



```
UDOT ZA.D[<Wv>, <offs>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H[<index>]
```

```
1 if !(HaveSME2() && HaveSMEI16164()) then UNDEFINED;
2 integer v = UInt('010':Rv);
3 constant integer esize = 64;
4 integer n = UInt(Zn:'0');
5 integer m = UInt('0':Zm);
```

Chapter D1. SME instructions D1.1. SME and SME2 data-processing instructions

- 6 integer offset = UInt(off3);
- 7 integer index = UInt(i1); 8 constant integer nreg = 2;
 - Four ZA single-vectors of 32-bit elements (FEAT_SME2)



- <Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.
- <index> For the four ZA single-vectors of 32-bit elements and two ZA single-vectors of 32-bit

elements variant: is the element index, in the range 0 to 3, encoded in the "i2" field.

For the four ZA single-vectors of 64-bit elements and two ZA single-vectors of 64-bit elements variant: is the element index, in the range 0 to 1, encoded in the "i1" field.

```
CheckStreamingSVEAndZAEnabled();
1
   constant integer VL = CurrentVL;
2
   constant integer elements = VL DIV esize;
3
4
   integer vectors = VL DIV 8;
   integer vstride = vectors DIV nreq;
5
6
   integer eltspersegment = 128 DIV esize;
7
   bits(32) vbase = X[v, 32];
   integer vec = (UInt(vbase) + offset) MOD vstride;
8
9
   bits(VL) result;
10
   for r = 0 to nreg-1
11
12
        bits(VL) operand1 = Z[n+r, VL];
13
        bits(VL) operand2 = Z[m, VL];
        bits(VL) operand3 = ZAvector[vec, VL];
14
15
        for e = 0 to elements-1
            bits(esize) sum = Elem[operand3, e, esize
16
17
            integer segmentbase = e - (e MOD eltspe
                                                       egment)
18
            integer s = segmentbase + index;
19
            for i = 0 to 3
                                                     11,
20
                integer element1 = UInt(Elem[
                                                                     esize DIV 4]);
21
                integer element2 = UInt(Ele<sup>-</sup> operat
                                                     2.
                                                        4
                                                                    esize DIV 4]);
22
                sum = sum + element1 * el ent2;
23
            Elem[result, e, esize] = sum;
        ZAvector[vec, VL] = result;
24
25
        vec = vec + vstride;
```

D1.1.239 UDOT (4-way, multiple and single vector)

Multi-vector unsigned integer dot-product by vector

The instruction operates on two or four ZA single-vector groups.

The unsigned integer dot product instruction computes the dot product of four unsigned 8-bit or 16-bit integer values held in each 32-bit or 64-bit element of the two or four first source vectors and four unsigned 8-bit or 16-bit integer values in the corresponding 32-bit or 64-bit element of the second source vector. The widened dot product result is destructively added to the corresponding 32-bit or 64-bit element of the two or four ZA single-vector groups. The vector numbers forming the single-vector group within each half or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand control of two or four ZA single-vector groups respectively. The VECTOR GROUP symbol is preferred for disact mbly, but period in assembler source code.

This instruction is unpredicated.

ID_AA64SMFR0_EL1.I16I64 indicates whether the 16-bit ...eger variant is valuented.

It has encodings from 2 classes: Two ZA single-vectors a. Four ' A single-vectors

Two ZA single-vectors (FEAT_SME2)

 31
 30
 29
 23
 22
 21
 20

 1
 1
 0
 0
 0
 1
 0
 sz
 1
 0

 Zm
 0
 ...
 1
 0
 1
 0
 off3

```
UDOT ZA.<T>[<Wv>, < _fs, VGx2}], _ Zn1>.<Tb>-<Zn2>.<Tb> }, <Zm>.<Tb>
```

```
if !HaveSME2() ther NDEFINED;
1
2
   if sz == '1' && ' veSMEI16I64() t. UNDEFINED;
   integer v = UIr 010':P ';
3
   constant inte( · esize
integer n = UIn, 'n)
4
                              32
                                 << UInt(sz);
5
                     ~n)
                         :Zm):
6
   integer m
                UInt(
   integer IIS = U1. 'off
7
         c inte :r nreg
8
   const
```

```
Four 's' sie-vecus
(FEAT_, 'E2)
```



UDOT ZA.<T>[<Wv>, <offs>{, VGx4}], { <Zn1>.<Tb>-<Zn4>.<Tb> }, <Zm>.<Tb>

```
if !HaveSME2() then UNDEFINED;
1
  if sz == '1' && !HaveSMEI16I64() then UNDEFINED;
2
  integer v = UInt('010':Rv);
3
4
  constant integer esize = 32 << UInt(sz);</pre>
5
   integer n = UInt(Zn);
  integer m = UInt('0':Zm);
6
7
  integer offset = UInt(off3);
  constant integer nreg = 4;
8
```

Assembler Symbols

<T> Is the size specifier, encoded in "sz":

SZ	< T >
0	S
1	D

- <Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
- <offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.
- <Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn".
- <Tb> Is the size specifier, encoded in "sz":

SZ	< Tb >
0	В
1	Н

- <Zn4> Is the name of the fourth scalable vector register of a *r*...ti-vector sequ. ~e. coded as "Zn" plus 3 modulo 32.
- <Zn2> Is the name of the second scalable vector register of alti-vector sequence, encoded as "Zn" plus 1 modulo 32.
- <Zm> Is the name of the second source scale is vector gister Z0-, 3, encoded in the "Zm" field.

```
1
    CheckStreamingSVEAndZAEnabled
2
    constant integer VL = Curre /L;
    constant integer elements 'L DIV esiz
 3
    integer vectors = VL DI 3;
 4
 5
    integer vstride = vectors DIV
                                          rg;
    bits(32) vbase = X[- 32];
 6
    integer vec = (UI vbase) + offse MOD vstride;
7
 8
    bits(VL) result
9
10
    for r = 0 to nr
                         1
        bits(VI) opei = Z[(r / MOD 32, VL];
bit ___, peran = Z[ VL];
11
12
         b<sup>:</sup> s(VL) erands
13
                                 vector[vec, VL];
14
          sr e = f co elemen s-1
15
             bit
                            m = Elem[operand3, e, esize];
16
                  i = 0 to 3
                  integer element1 = UInt(Elem[operand1, 4 * e + i, esize DIV 4]);
integer element2 = UInt(Elem[operand2, 4 * e + i, esize DIV 4]);
17
18
                   m = sum + element1 * element2;
19
20
             Elem[result, e, esize] = sum;
21
         ZAvector[vec, VL] = result;
22
         vec = vec + vstride;
```

D1.1.240 UDOT (4-way, multiple vectors)

Multi-vector unsigned integer dot-product

The instruction operates on two or four ZA single-vector groups.

The unsigned integer dot product instruction computes the dot product of four unsigned 8-bit or 16-bit integer values held in each 32-bit or 64-bit element of the two or four first source vectors and four unsigned 8-bit or 16-bit integer values in the corresponding 32-bit or 64-bit element of the two or four second source vectors. The widened dot product result is destructively added to the corresponding 32-bit or 64-bit element of the two or four ZA single-vector groups. The vector numbers forming the single-vector group within each half or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand control of two or four ZA single-vector groups respectively. The VECTOR GROUP symbol is preferred for disacombly, but ptional in assembler source code.

This instruction is unpredicated.

ID_AA64SMFR0_EL1.I16I64 indicates whether the 16-bit ...eger variant is variant is ...ented.

It has encodings from 2 classes: Two ZA single-vectors a. Four ' A single-vectors

Two ZA single-vectors (FEAT_SME2)



```
4 constant integer esize = 32 << UInt(sz);
5 integer n = UInt(Zn:'00');
```

```
5 integer n = UInt(Zn:'00');
6 integer = UInt(Zn:'00');
```

```
6 integer m = UInt(Zm:'00');
7 integer offset = UInt(off3)
```

```
7 integer offset = UInt(off3);
8 constant integer nreg = 4;
```

Assembler Symbols

<T> Is the size specifier, encoded in "sz":

SZ	< T >
0	S
1	D

- <Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
- <offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.
- <Zn1> For the two ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.

For the four ZA single-vectors variant: is the name of the first scale of a multi-vector sequence, encoded as "Zn" times 4.

<Tb> Is the size specifier, encoded in "sz":

SZ	< Tb >
0	В
1	Н

- <Zn4> Is the name of the fourth scalable vector region for a multiple vector quence, encoded as "Zn" times 4 plus 3.
- <Zn2> Is the name of the second scalable ve or regimetrivector sequence, encoded as "Zn" times 2 plus 1.
- <Zm1> For the two ZA single-vect's variation is the time of the first scalable vector register of a multi-vector sequence, e coded as "Zn_times 2.

For the four ZA sin -vec. s variant: is the name of the first scalable vector register of a multi-vector sequence, encode is "Zm" times 4.

- <Zm4> Is the name the fourth scalable actor register of a multi-vector sequence, encoded as "Zm" tim 4 plus 3
- $\langle Zm2 \rangle$ Is the name γ second able vector register of a multi-vector sequence, encoded as "7 les 2 μ 1.

Op ation

```
.ingSVEAnoZAEnabled();
 1
   Check
   constan. nteger VL = CurrentVL;
constant . eger elements = VL DIV esize;
 2
 3
 4
   integer vect s = VL DIV 8;
   integer vstride = vectors DIV nreg;
 5
   bits(32) vbase = X[v, 32];
 6
   integer vec = (UInt(vbase) + offset) MOD vstride;
 7
8
   bits(VL) result;
 0
10
   for r = 0 to nreg-1
11
        bits(VL) operand1 = Z[n+r, VL];
12
        bits(VL) operand2 = Z[m+r, VL];
        bits(VL) operand3 = ZAvector[vec, VL];
13
14
        for e = 0 to elements-1
15
            bits(esize) sum = Elem[operand3, e, esize];
16
            for i = 0 to 3
17
                 integer element1 = UInt(Elem[operand1, 4 * e + i, esize DIV 4]);
18
                 integer element2 = UInt(Elem[operand2, 4 * e + i, esize DIV 4]);
19
                 sum = sum + element1 * element2;
20
            Elem[result, e, esize] = sum;
21
        ZAvector[vec, VL] = result;
22
        vec = vec + vstride;
```

D1.1.241 UMAX (multiple and single vector)

Multi-vector unsigned maximum by vector

Determine the unsigned maximum of elements of the second source vector and the corresponding elements of the two or four first source vectors and destructively place the results in the corresponding elements of the two or four first source vectors.

This instruction is unpredicated.

It has encodings from 2 classes: Two registers and Four registers

Two registers (FEAT_SME2)



Assembler Sy bols

<Zdn1> For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2.

For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4.

<T> Is the size specifier, encoded in "size":

size	<t></t>
00	В
01	Н
10	S
11	D

<Zdn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as

"Zdn" times 4 plus 3.

- <Zdn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2 plus 1.
 - <Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

```
1
   CheckStreamingSVEEnabled();
   constant integer VL = CurrentVL;
2
3
   constant integer elements = VL DIV esize;
4
   array [0..3] of bits(VL) results;
5
6
   for r = 0 to nreg-1
        bits(VL) operand1 = Z[dn+r, VL];
bits(VL) operand2 = Z[m, VL];
7
8
9
        for e = 0 to elements-1
            integer element1 = Int(Elem[operand1, e, esize],
10
                                                                    signed);
            integer element2 = Int(Elem[operand2, e, esize],
11
                                                                  ù
                                                                     'gned);
12
            integer res = Max(element1, element2);
13
            Elem[results[r], e, esize] = res<esize-1:0>
14
   for r = 0 to nreg-1
15
16
    Z[dn+r, VL] = results[r];
```

D1.1.242 UMAX (multiple vectors)

Multi-vector unsigned maximum

Determine the unsigned maximum of elements of the two or four second source vectors and the corresponding elements of the two or four first source vectors and destructively place the results in the corresponding elements of the two or four first source vectors.

This instruction is unpredicated.

It has encodings from 2 classes: Two registers and Four registers

Two registers (FEAT_SME2)



Assembler Symbols

<Zdn1> For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2.

For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4.

<T> Is the size specifier, encoded in "size":

size	<t></t>
00	В
01	Н
10	S
11	D

- <Zdn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4 plus 3.
- <Zdn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2 plus 1.
- <Zm1> For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 2.

For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 4.

- <Zm4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zm" times 4 plus 3.
- <Zm2> Is the name of the second scalable vector register of a multi-vector ence, encoded as "Zm" times 2 plus 1.

```
1
   CheckStreamingSVEEnabled();
   constant integer VL = CurrentVL;
2
3
   constant integer elements = VL DIV esize;
4
   array [0..3] of bits(VL) results;
5
6
   for r = 0 to nreg-1
       bits(VL) operand1 = Z[dn+r, VL];
7
       bits(VL) operand2 = Z[m+r, VL];
8
9
       for e = 0 to elements-1
10
                                              nd
                                                      es___, unsigned);
            integer element1 = Int(Elem[ope
                                                   e,
11
            integer element2 = Int(E1 [opera
                                                 2, e, esize], unsigned);
12
            integer res = Max(eler nt1,
                                          lemen
13
            Elem[results[r], e,
                                  .ize] =
                                             s<esiz
                                                    1:0>;
14
15
   for r = 0 to nreg-1
16
       Z[dn+r, VL] = resul .s[r];
```

D1.1.243 UMIN (multiple and single vector)

Multi-vector unsigned minimum by vector

Determine the unsigned minimum of elements of the second source vector and the corresponding elements of the two or four first source vectors and destructively place the results in the corresponding elements of the two or four first source vectors.

This instruction is unpredicated.

It has encodings from 2 classes: Two registers and Four registers

Two registers (FEAT_SME2)



Assembler Sy bols

<Zdn1> For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2.

For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4.

<T> Is the size specifier, encoded in "size":

size	<t></t>
00	В
01	Н
10	S
11	D

<Zdn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as

"Zdn" times 4 plus 3.

- <Zdn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2 plus 1.
 - <Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

```
1
   CheckStreamingSVEEnabled();
   constant integer VL = CurrentVL;
2
3
   constant integer elements = VL DIV esize;
4
   array [0..3] of bits(VL) results;
5
6
   for r = 0 to nreg-1
        bits(VL) operand1 = Z[dn+r, VL];
bits(VL) operand2 = Z[m, VL];
7
8
9
        for e = 0 to elements-1
            integer element1 = Int(Elem[operand1, e, esize],
10
                                                                    signed);
            integer element2 = Int(Elem[operand2, e, esize],
11
                                                                  ù
                                                                     'gned);
12
            integer res = Min(element1, element2);
13
            Elem[results[r], e, esize] = res<esize-1:0>
14
   for r = 0 to nreg-1
15
16
    Z[dn+r, VL] = results[r];
```

D1.1.244 UMIN (multiple vectors)

Multi-vector unsigned minimum

Determine the unsigned minimum of elements of the two or four second source vectors and the corresponding elements of the two or four first source vectors and destructively place the results in the corresponding elements of the two or four first source vectors.

This instruction is unpredicated.

It has encodings from 2 classes: Two registers and Four registers

Two registers (FEAT_SME2)



Assembler Symbols

<Zdn1> For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2.

For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4.

<T> Is the size specifier, encoded in "size":

size	<t></t>
00	В
01	Н
10	S
11	D

- <Zdn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4 plus 3.
- <Zdn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2 plus 1.
- <Zm1> For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 2.

For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 4.

- <Zm4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zm" times 4 plus 3.
- <Zm2> Is the name of the second scalable vector register of a multi-vector ence, encoded as "Zm" times 2 plus 1.

```
1
   CheckStreamingSVEEnabled();
   constant integer VL = CurrentVL;
2
3
   constant integer elements = VL DIV esize;
4
   array [0..3] of bits(VL) results;
5
6
   for r = 0 to nreg-1
       bits(VL) operand1 = Z[dn+r, VL];
7
       bits(VL) operand2 = Z[m+r, VL];
8
9
       for e = 0 to elements-1
10
                                              nd
                                                      es___, unsigned);
            integer element1 = Int(Elem[ope
                                                   e,
11
            integer element2 = Int(E1 [opera
                                                 2, e, esize], unsigned);
            integer res = Min(eler nt1,
12
                                          lemen
13
            Elem[results[r], e,
                                  .ize] =
                                             s<esiz
                                                    1:0>;
14
15
   for r = 0 to nreg-1
16
       Z[dn+r, VL] = resul .s[r];
```

D1.1.245 UMLAL (multiple and indexed vector)

Multi-vector unsigned integer multiply-add long by indexed element

The instruction operates on one, two, or four ZA double-vector groups.

This unsigned integer multiply-add long instruction multiplies each unsigned 16-bit element in the one, two, or four first source vectors with each unsigned 16-bit indexed element of the second source vector, widens each product to 32-bits and destructively adds these values to the corresponding 32-bit elements of the one, two, or four ZA double-vector groups.

The elements within the second source vector are specified using an immediate element index which selects the same element position within each 128-bit vector segment. The index range is from 0 to 7, encoded in 3 bits. The lowest of the two consecutive vector numbers forming the double-vector group within all, each half, or each quarter of the ZA array are selected by the sum of the vector select register mediate offset, modulo all, half, or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the \land operand consists of two or four ZA double-vector groups respectively. The VECTOR GROUP symbol is $_{\rm F}$ ferred for isassembly, but optional in assembler source code.

This instruction is unpredicated.

It has encodings from 3 classes: One ZA double-vector, Two double-vectors and Four ZA double-vectors

One ZA double-vector (FEAT_SME2)



UMLAL ZA.S[<Wv>, <offsf>:<< [sl>], <Zn>.H, <Zm>.H[<index>]

```
if !HaveSME2() t<sup>1</sup> . UNDEFINED;
1
2
  constant integr esize = 32;
  integer v = U1 ('010' .v);
3
4
   integer n = UInt
                      יר '
                         •Zm):
5
  integer m
                VInt (
6
  intege offs.
                   = UIn. off
                                 (0')
   inter _ index = UInt(1
7
                              131);
  cor ant int /er nreg = 1;
8
```

```
Two ZA uble-vectors
(FEAT_SN. ?)
```



UMLAL ZA.S[<Wv>, <offsf>:<offsl>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H[<index>]

```
1 if !HaveSME2() then UNDEFINED;
```

```
2 constant integer esize = 32;
```

```
3 integer v = UInt('010':Rv);
```

```
4 integer n = UInt(Zn:'0');
```

```
5 integer m = UInt('0':Zm);
```

```
6 integer offset = UInt(off2:'0');
```

```
7 integer index = UInt(i3h:i3l);
8 constant integer nreg = 2;
```

Four ZA double-vectors (FEAT_SME2) 0 0 0 1 0 0 Zm Rv Zn 1 1 0 1 i3h 1 UMLAL ZA.S[<Wv>, <offsf>:<offsl>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H[<index>] 1 if !HaveSME2() then UNDEFINED; 2 constant integer esize = 32; integer v = UInt('010':Rv); 3 4 integer n = UInt(Zn:'00'); integer m = UInt('0':Zm); 5 integer offset = UInt(off2:'0'); 6 integer index = UInt(i3h:i3l); constant **integer** nreg = 4; **Assembler Symbols** / field. $\langle Wv \rangle$ Is the 32-bit name of the vector select register W8.³ 11, encc d in the For the one ZA double-vector variant: is the vector electron offset, r inting to first of two <offsf> consecutive vectors, encoded as "off3" field s 2. For the four ZA double-vectors and two A double vectors At: is the vector select offset, pointing to first of two consecutive v 'ors, enc int as "off2" field times 2. <offsl> For the one ZA double-vector riant: . e vector select offset, pointing to last of two consecutive vectors, encoder'ns "o. "" fiela res 2 plus 1. For the four ZA double- tors and two \double-vectors variant: is the vector select offset, pointing to last of tw _on_ utive vectors needed as "off2" field times 2 plus 1. Is the name of the first source _____able vector register, encoded in the "Zn" field. $\langle Zn \rangle$ $\langle Zn1 \rangle$ For the two double-vectors val. at: is the name of the first scalable vector register of a multi-vec c sequer , encoded as "Zn" times 2. For the four <u>v</u> ouble-vec rs variant: is the name of the first scalable vector register of a tor sec, ince, i loded as "Zn" times 4. m <Zn4>is the nr le of the full in scalable vector register of a multi-vector sequence, encoded as "Zn" times ls of name of the second scalable vector register of a multi-vector sequence, encoded as <Zn2>"Zn nes 2 plus 1. Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field. $\langle Zm \rangle$ <index> Is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields. Operation

```
CheckStreamingSVEAndZAEnabled();
1
2
   constant integer VL = CurrentVL;
   constant integer elements = VL DIV esize;
3
4
   integer vectors = VL DIV 8;
   integer vstride = vectors DIV nreg;
5
   integer eltspersegment = 128 DIV esize;
6
   bits(32) vbase = X[v, 32];
7
8
   integer vec = (UInt(vbase) + offset) MOD vstride;
   bits(VL) result;
9
10
   vec = vec - (vec MOD 2);
11
12 for r = 0 to nreg-1
```

Chapter D1. SME instructions D1.1. SME and SME2 data-processing instructions

```
bits(VL) operand1 = Z[n+r, VL];
13
14
        bits(VL) operand2 = Z[m, VL];
15
        for i = 0 to 1
16
             bits(VL) operand3 = ZAvector[vec + i, VL];
17
             for e = 0 to elements-1
18
                  integer segmentbase = e - (e MOD eltspersegment);
                 integer s = 2 * segmentbase + index;
19
                 integer element1 = UInt(Elem[operand1, 2 * e + i, esize DIV 2]);
integer element2 = UInt(Elem[operand2, s, esize DIV 2]);
20
21
22
                 bits(esize) product = (element1 * element2)<esize-1:0>;
23
                 Elem[result, e, esize] = Elem[operand3, e, esize] + product;
24
             ZAvector[vec + i, VL] = result;
25
        vec = vec + vstride;
```

D1.1.246 UMLAL (multiple and single vector)

Multi-vector unsigned integer multiply-add long by vector

The instruction operates on one, two, or four ZA double-vector groups.

This unsigned integer multiply-add long instruction multiplies each unsigned 16-bit element in the one, two, or four first source vectors with each unsigned 16-bit element in the second source vector, widens each product to 32-bits and destructively adds these values to the corresponding 32-bit elements of the one, two, or four ZA double-vector groups. The lowest of the two consecutive vector numbers forming the double-vector group within all, each half, or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo all, half, or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA double-vector groups respectively. The VECTOR GROUP symbol is provide for disassembly, but optional in assembler source code.

This instruction is unpredicated.

It has encodings from 3 classes: One ZA double-vector, Two 7, double-vectors and our ZA double-vectors

One ZA double-vector (FEAT_SME2)



UMLAL ZA.S[<Wv>, <offsf>:< ffsl . <Zn H, <Zm>.H

```
1 if !HaveSME2() then UNDEF
```

2 constant integer esize - 32;

- 3 integer v = UInt('010':xv);
- 4 integer n = UInt(Zn'
- 5 integer m = UInt(' :Zm);
- 6 integer offset > Int(off3:'0');
- constant **inte** nreg



UMLAL ZA.S[<Wv>, <offsf>:<offsl>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H

- 1 if !HaveSME2() then UNDEFINED;
- 2 constant integer esize = 32;
- 3 integer v = UInt('010':Rv);
- 4 integer n = UInt(Zn);
- 5 integer m = UInt('0':Zm);
- 6 integer offset = UInt(off2:'0');
 7 constant integer preg = 2:
- constant integer nreg = 2;

Four ZA double-vectors (FEAT_SME2)



UMLAL ZA.S[<Wv>, <offsf>:<offsl>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H

- 1 if !HaveSME2() then UNDEFINED;
- 2 constant integer esize = 32;
- 3 integer v = UInt('010':Rv);
- 4 integer n = UInt(Zn);
- 5 integer m = UInt('0':Zm);
- 6 integer offset = UInt(off2:'0');
- 7 constant **integer** nreg = 4;

Assembler Symbols

- <Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.

For the four ZA double-vectors and two ZA double-vectors variables is the vector elect offset, pointing to first of two consecutive vectors, encoded as " .f2" field imes 2.

<offsl> For the one ZA double-vector variant: is the vect select offset, put is to last of two consecutive vectors, encoded as "off3" field times plus 1

For the four ZA double-vectors and two ZA double-vector variant: the vector select offset, pointing to last of two consecutive vector encoded as "to 2" for times 2 plus 1.

- <Zn> Is the name of the first source scalab' vector rester, encoded in the "Zn" field.
- $\langle Zn1 \rangle$ Is the name of the first scalable vector region of a multi-vector sequence, encoded as "Zn".
- <Zn4> Is the name of the fourth sc?' ble ver regis of a multi-vector sequence, encoded as "Zn" plus 3 modulo 32.
- <Zn2> Is the name of the sound plable vector gister of a multi-vector sequence, encoded as "Zn" plus 1 modplo 32.
- <Zm> Is the name c .ne second source s 'able vector register Z0-Z15, encoded in the "Zm" field.

```
CheckStre qSVEA 7AEnabl ();
constar int, rVL Cur .tVL;
const .t inte :r eleme = VL DIV esize;
 1
2
 3
    int er vect s = VL DIV 8;
inte. VS rate Cors D
 4
                           .ors DIV nreg;
 5
    bits(3. pase = X[v, 32];
integer = (UInt(vbase) + offset) MOD vstride;
6
 7
    bits(VL) r lt;
vec = vec - vec MOD 2);
 8
9
10
11
    for r = 0 to nreg-1
         bits(VL) operand1 = Z[(n+r) MOD 32, VL];
12
13
         bits(VL) operand2 = Z[m, VL];
14
         for i = 0 to 1
15
             bits(VL) operand3 = ZAvector[vec + i, VL];
16
              for e = 0 to elements-1
                  integer element1 = UInt(Elem[operand1, 2 * e + i, esize DIV 2]);
17
18
                  integer element2 = UInt(Elem[operand2, 2 * e + i, esize DIV 2]);
19
                  bits(esize) product = (element1 * element2)<esize-1:0>;
20
                  Elem[result, e, esize] = Elem[operand3, e, esize] + product;
21
             ZAvector[vec + i, VL] = result;
22
        vec = vec + vstride;
```

D1.1.247 UMLAL (multiple vectors)

Multi-vector unsigned integer multiply-add long

The instruction operates on two or four ZA double-vector groups.

This unsigned integer multiply-add long instruction multiplies each unsigned 16-bit element in the two or four first source vectors with each unsigned 16-bit element in the two or four second source vectors, widens each product to 32-bits and destructively adds these values to the corresponding 32-bit elements of the two or four ZA double-vector groups. The lowest of the two consecutive vector numbers forming the double-vector group within each half or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA double-vector groups respectively. The VECTOR GROUP symbol is project of disassembly, but optional in assembler source code.

This instruction is unpredicated.

It has encodings from 2 classes: Two ZA double-vectors and F at ZA dou 2-vector

Two ZA double-vectors (FEAT_SME2)



Assembler Symbols

integer offset = UInt(off2:'0');

constant integer nreg = 4;

<Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.

<offsf> Is the vector select offset, pointing to first of two consecutive vectors, encoded as "off2" field times 2.

6 7

Chapter D1. SME instructions

D1.1. SME and SME2 data-processing instructions

- <offsl> Is the vector select offset, pointing to last of two consecutive vectors, encoded as "off2" field times 2 plus 1.
- <Zn1> For the two ZA double-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.

For the four ZA double-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

- <Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.
- <Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.
- <Zm1> For the two ZA double-vectors variant: is the name of the first scale¹ vector register of a multi-vector sequence, encoded as "Zm" times 2.

For the four ZA double-vectors variant: is the name of the fin. calable vecto. egister of a multi-vector sequence, encoded as "Zm" times 4.

- <Zm4> Is the name of the fourth scalable vector register of multi-vector rue c, encoded as "Zm" times 4 plus 3.
- <Zm2> Is the name of the second scalable vector register o. _____iti-vecto__equence, encoded as "Zm" times 2 plus 1.

```
1
   CheckStreamingSVEAndZAEnabled();
2
   constant integer VL = CurrentVL;
   constant integer elements = V DIV
3
                                            ze;
4
   integer vectors = VL DIV 8;
   integer vstride = vectors
5
                                  1 nreq;
6
   bits(32) vbase = X[v, 3',
   integer vec = (UInt(vbase) + o. et) MOD vstride;
7
   bits(VL) result;
8
9
                       5 2);
   vec = vec - (vec)
10
   for r = 0 to i g-1
11
       bits(VL) op nd^2 = Z[n+r].
bits(V^T) oper Z = Z[m+r]
12
                            Z[n+r. VL];
13
        bits(VT ) oper
                                    VL];
14
        for
                  to 1
            bits( ) oper
                            > = ZAvector[vec + i, VL];
15
16
            for < = 0 to e1 ments-1
17
                          lement1 = UInt(Elem[operand1, 2 * e + i, esize DIV 2]);
                integer element2 = UInt(Elem[operand2, 2 * e + i, esize DIV 2]);
18
19
                bits(esize) product = (element1 * element2)<esize-1:0>;
20
                Elem[result, e, esize] = Elem[operand3, e, esize] + product;
21
            ZAve or[vec + i, VL] = result;
22
        vec = vec + vstride;
```

D1.1.248 UMLALL (multiple and indexed vector)

Multi-vector unsigned integer multiply-add long long by indexed element

The instruction operates on one, two, or four ZA quad-vector groups.

This unsigned integer multiply-add long long instruction multiplies each unsigned 8-bit or 16-bit element in the one, two, or four first source vectors with each unsigned 8-bit or 16-bit indexed element of second source vector, widens each product to 32-bits or 64-bits and destructively adds these values to the corresponding 32-bit or 64-bit elements of the one, two, or four ZA quad-vector groups.

The elements within the second source vector are specified using an immediate element index which selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 3 to 4 bits depending on the size of the element. The lowest of the four consecutive vector numbers forming the quad-vector group within all half, or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, not all half, or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the 7 operation consists of voor four ZA quad-vector groups respectively. The VECTOR GROUP symbol is preferre for disassen. v, but ptional in assembler source code.

This instruction is unpredicated.

ID_AA64SMFR0_EL1.I16I64 indicates whether leng it integination is implemented.

It has encodings from 6 classes: One ZA quad-vector of 2-bit elements, One ZA quad-vector of 64-bit elements, Two ZA quad-vectors of 32-bit elements, Tv ZA quad-vectors of 64-bit elements, Four ZA quad-vectors of 64-bit elements

One ZA quad-vector of 32-bit e' nents (FEAT_SME2)



UMLALL ZA.S[<Wv, offsf>: ffsl>], <Zn>.B, <Zm>.B[<index>]

```
if !H eSME2 ( then
1
                             NED.
2
  con
       ant int er esize
                            32:
3
  inte
        τv
                        :Rv);
4
               UInt(Zn);
  intege
             = UInt('0':Zm);
5
  integer
  integer on et = UInt(off2:'00');
6
  integer inde = UInt(i4h:i4l);
7
8
  constant integer nreg = 1;
```

One ZA quad-vector of 64-bit elements

(FEAT_SME_I16I64)



UMLALL ZA.D[<Wv>, <offsf>:<offsl>], <Zn>.H, <Zm>.H[<index>]

if !(HaveSME2() && HaveSMEI16I64()) then UNDEFINED;

```
2 constant integer esize = 64;
```

```
3 integer v = UInt('010':Rv);
```

```
4 integer n = UInt(Zn);
```

1

Chapter D1. SME instructions D1.1. SME and SME2 data-processing instructions

- integer m = UInt('0':Zm); 5
- integer offset = UInt(off2:'00'); 6
- integer index = UInt(i3h:i3l); 7
- 8 constant integer nreg = 1;

Two ZA quad-vectors of 32-bit elements (FEAT_SME2)



UMLALL ZA.S[<Wv>, <offsf>:<offsl>{, VGx2}], { <Zn1>.B-<Zn2>.B }, <Zm>.B[<index>]

- if !HaveSME2() then UNDEFINED; 1
- 2 constant integer esize = 32;
- 3 integer v = UInt('010':Rv);
- integer n = UInt(Zn:'0'); 4
- integer m = UInt('0':Zm); 5
- 6 integer offset = UInt(o1:'00'); integer index = UInt(i4h:i4l);
- 7
- 8 constant integer nreg = 2;

Two ZA quad-vectors of 64-bit elements (FEAT SME I16I64)



UMLALL ZA.D[<Wv>, <off >> offsl>{, Vc }], { <Zn1>.H-<Zn2>.H }, <Zm>.H[<index>]

1

- constant integer < _ze = 64; integer v = UIn⁺ 010':R^x); integer n = U⁺ (Zn:'0' 2
- 3
- 4
- 5
- 6 7
- integer m = UIh. '0' ...);
 integer offset = _ (o1:'00'
 integer = UIh. 'i3h:i',
 constat inte ;r nreg 8

Fou 'A qv f 32-bit elements (FEA1_ Æ2) 0 0 0 0 1 1 0 Zm 1 0 0 1 1 Rv 0 i4h 0 0 1 0 υJ Ls

UMLALL ZA.S[<Wv>, <offsf>:<offsl>{, VGx4}], { <Zn1>.B-<Zn4>.B }, <Zm>.B[<index>]

- 1 if !HaveSME2() then UNDEFINED;
- 2 constant integer esize = 32;
- 3 integer v = UInt('010':Rv);
- 4 integer n = UInt(Zn:'00');
- 5 integer m = UInt('0':Zm);
- 6 integer offset = UInt(o1:'00'); 7
- integer index = UInt(i4h:i4l); 8

constant integer nreg = 4;

Four ZA quad-vectors of 64-bit elements (FEAT_SME_I16I64)

01

i4l



UMLALL ZA.D[<Wv>, <offsf>:<offsl>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H[<index>]

- if !(HaveSME2() && HaveSMEI16I64()) then UNDEFINED;
- 2 constant integer esize = 64;
- 3 integer v = UInt('010':Rv); 4
- integer n = UInt(Zn:'00');
- 5 integer m = UInt('0':Zm);
- 6 integer offset = UInt(o1:'00'); integer index = UInt(i3h:i3l);
- 7 8 constant integer nreg = 4;

Assembler Symbols

<Wv> Is the 32-bit name of the vector select register W8-W11, encode in the "Rv" t ld.

For the one ZA quad-vector of 32-bit elements and or $\angle A$ quad-vector of f bit elements <offsf> variant: is the vector select offset, pointing to first four corecutive .ors, encoded as "off2" field times 4.

> four. quad-y .ors of 64-bit elements, For the four ZA quad-vectors of 32-bit eler two ZA quad-vectors of 32-bit elements a two. A quad- vector of 64-bit elements variant: is the vector select offset, pointing to st of four consecutive vectors, encoded as "o1" field times 4.

For the one ZA quad-vector of 32 it elen. 's and one ZA quad-vector of 64-bit elements <offsl> variant: is the vector selec' ifset, porting to st of four consecutive vectors, encoded as "off2" field times 4 plus

> For the four ZA qu. -vecto. of 32-bit elements, four ZA quad-vectors of 64-bit elements, two ZA quad-vectors of 32-bit ments and two ZA quad-vectors of 64-bit elements variant: is the vector ect offset, pointing last of four consecutive vectors, encoded as "o1" field times 4 ply 5.

- Is the name ^cth *inst source scalable vector register, encoded in the "Zn" field.* $\langle Zn \rangle$
- <Zn1> F inc 'o ZA ad-ve is of 32-bit elements and two ZA quad-vectors of 64-bit elements √ariant: the name the first scalable vector register of a multi-vector sequence, encoded

he four ZA quad-vectors of 32-bit elements and four ZA quad-vectors of 64-bit elements var. t: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn imes 4.

- <Zn4>Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.
- <Zn2>Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.
- <Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.
- <index> For the four ZA quad-vectors of 32-bit elements, one ZA quad-vector of 32-bit elements and two ZA quad-vectors of 32-bit elements variant: is the element index, in the range 0 to 15, encoded in the "i4h:i4l" fields.

For the four ZA quad-vectors of 64-bit elements, one ZA quad-vector of 64-bit elements and two ZA quad-vectors of 64-bit elements variant: is the element index, in the range 0 to 7, encoded in the "i3h:i31" fields.

```
CheckStreamingSVEAndZAEnabled();
 1
    constant integer VL = CurrentVL;
 2
3
   constant integer elements = VL DIV esize;
4 integer vectors = VL DIV 8;
    integer vstride = vectors DIV nreg;
5
   integer eltspersegment = 128 DIV esize;
6
7
    bits(32) vbase = X[v, 32];
    integer vec = (UInt(vbase) + offset) MOD vstride;
8
9
    bits(VL) result;
   vec = vec - (vec MOD 4);
10
11
12
    for r = 0 to nreg-1
        bits(VL) operand1 = Z[n+r, VL];
13
14
        bits(VL) operand2 = Z[m, VL];
15
         for i = 0 to 3
             bits(VL) operand3 = ZAvector[vec + i, VL];
16
17
             for e = 0 to elements-1
18
                  integer segmentbase = e - (e MOD eltspersegme. `;
19
                  integer s = 4 * segmentbase + index;
                 integer element1 = UInt(Elem[operand1, 4 e + i, size i
integer element2 = UInt(Elem[operand2, , esize DIV ');
bits(esize) product = (element1 * el ent2)<e ze-1:0</pre>
20
                                                                              rize DIV
                                                                                         ·]);
21
22
23
                  Elem[result, e, esize] = Elem[operal <sup>3</sup>, e,
                                                                    _ze] + product;
24
             ZAvector[vec + i, VL] = result;
25
        vec = vec + vstride;
```

D1.1.249 UMLALL (multiple and single vector)

Multi-vector unsigned integer multiply-add long long by vector

The instruction operates on one, two, or four ZA quad-vector groups.

This unsigned integer multiply-add long long instruction multiplies each unsigned 8-bit or 16-bit element in the one, two, or four first source vectors with each unsigned 8-bit or 16-bit element in the second source vector, widens each product to 32-bits or 64-bits and destructively adds these values to the corresponding 32-bit or 64-bit elements of the one, two, or four ZA quad-vector groups. The lowest of the four consecutive vector numbers forming the quad-vector group within all, each half, or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo all, half, or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA quad-vector groups respectively. The VECTOR GROUP symbol is preferred for disasser to prove optional in assembler source code.

This instruction is unpredicated.

ID_AA64SMFR0_EL1.I16I64 indicates whether the 16-bit int _er varian. implem_ted.

It has encodings from 3 classes: One ZA quad-vector, Tv ZA quad ectors a our ZA quad-vectors

One ZA quad-vector (FEAT_SME2)



UMLALL ZA.<T>[<Wv>, <of t>:<offsl> <Zn>.<Tb>, <Zm>.<Tb>

```
if !HaveSME2() then UNL_FINED;
1
                                           hen UNDEFINED;
2
   if sz == '1' && !Hay CMEI16I64()
3
   constant integer < _ze = 32 << UIn sz);</pre>
                       010':R');
4
   integer v = UIn+
5
   integer n = U<sup>T</sup> .(Zn);
   integer m = UIh
integer offset =
6
                        '0'
                              .0 ;
7
                           .(off2:
8
   constan+
                    er ni
                             = 1;
```

Tw 4A quad ectors (FEA SN .2)



UMLALL ZA.<T>[<Wv>, <offsf>:<offsl>{, VGx2}], { <Zn1>.<Tb>-<Zn2>.<Tb> }, <Zm>.<Tb>

```
1
  if !HaveSME2() then UNDEFINED;
  if sz == '1' && !HaveSMEI16I64() then UNDEFINED;
2
3
  constant integer esize = 32 << UInt(sz);</pre>
  integer v = UInt('010':Rv);
4
5
  integer n = UInt(Zn);
  integer m = UInt('0':Zm);
6
  integer offset = UInt(o1:'00');
7
  constant integer nreg = 2;
8
```

Four ZA quad-vectors (FEAT_SME2)



UMLALL ZA.<T>[<Wv>, <offsf>:<offsl>{, VGx4}], { <Zn1>.<Tb>-<Zn4>.<Tb> }, <Zm>.<Tb>

```
1 if !HaveSME2() then UNDEFINED;
```

- 2 if sz == '1' && !HaveSMEI16I64() then UNDEFINED;
- 3 constant integer esize = 32 << UInt(sz);</pre>
- 4 integer v = UInt('010':Rv);
- 5 integer n = UInt(Zn);
- 6 integer m = UInt('0':Zm);
- 7 integer offset = UInt(o1:'00');
- 8 constant **integer** nreg = 4;

Assembler Symbols

<T> Is the size specifier, encoded in "sz":

SZ	<t></t>
0	S
1	D

- <Wv> Is the 32-bit name of the vector select regimer v, W11, ordgime in the "Rv" field.
- $\langle offsf \rangle$ For the one ZA quad-vector variant is the vector select outset, pointing to first of four consecutive vectors, encoded as "off2 vectors 4.

For the four ZA quad-vectors and vo ZA ad-vectors variant: is the vector select offset, pointing to first of four conjuctive v tors, ended as "o1" field times 4.

<offsl> For the one ZA quad out variant: is evector select offset, pointing to last of four consecutive vectors, encoded s "off2" field times 4 plus 3.

For the four Z quad-vectors an wo ZA quad-vectors variant: is the vector select offset, pointing to 'st of four consecutive vectors, encoded as "o1" field times 4 plus 3.

- <Zn> Is the nan. If the 1st source scalable vector register, encoded in the "Zn" field.
- <Zn1> Is "me on first so uble vector register of a multi-vector sequence, encoded as "Zn".
- <Tb> is the size specifier coded in "sz":



- <Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" plus 3 modulo 32.
- <Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" plus 1 modulo 32.
- <Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

```
1 CheckStreamingSVEAndZAEnabled();
2 constant integer VL = CurrentVL;
3 constant integer elements = VL DIV esize;
4 integer vectors = VL DIV 8;
5 integer vstride = vectors DIV nreg;
6 bits(32) vbase = X[v, 32];
7 integer vec = (UInt(vbase) + offset) MOD vstride;
```

```
8 bits(VL) result;
9
    vec = vec - (vec MOD 4);
10
11
    for r = 0 to nreg-1
        bits(VL) operand1 = Z[(n+r) MOD 32, VL];
12
13
        bits(VL) operand2 = Z[m, VL];
        for i = 0 to 3
14
15
             bits(VL) operand3 = ZAvector[vec + i, VL];
             for e = 0 to elements-1
16
                 integer element1 = UInt(Elem[operand1, 4 * e + i, esize DIV 4]);
17
                 integer element2 = UInt(Elem[operand2, 4 * e + i, esize DIV 4]);
18
                 bits(esize) product = (element1 * element2)<esize-1:0>;
Elem[result, e, esize] = Elem[operand3, e, esize] + product;
19
20
21
             ZAvector[vec + i, VL] = result;
22
        vec = vec + vstride;
```

D1.1.250 UMLALL (multiple vectors)

Multi-vector unsigned integer multiply-add long long

The instruction operates on two or four ZA quad-vector groups.

This unsigned integer multiply-add long long instruction multiplies each unsigned 8-bit or 16-bit element in the two or four first source vectors with each unsigned 8-bit or 16-bit element in the one, two, or four second source vectors, widens each product to 32-bits or 64-bits and destructively adds these values to the corresponding 32-bit or 64-bit elements of the two or four ZA quad-vector groups. The lowest of the four consecutive vector numbers forming the quad-vector group within each half or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA quad-vector groups respectively. The VECTOR GROUP symbol is preferred for disasser to prove optional in assembler source code.

This instruction is unpredicated.

ID_AA64SMFR0_EL1.I16I64 indicates whether the 16-bit int _er varian. implem_ited.

It has encodings from 2 classes: Two ZA quad-vectors an our ZA ad-vector

Two ZA quad-vectors (FEAT_SME2)

12

3 4

5

6

7

8



↔<Zm1>.<Tb>-<Zm4>.<Tb> }

```
1 if !HaveSME2() then UNDEFINED;
2 if sz == '1' && !HaveSMEI16I64() then UNDEFINED;
3 constant integer esize = 32 << UInt(sz);
4 integer v = UInt('010':Rv);
5 integer n = UInt(Zn:'00');
6 integer m = UInt(Zm:'00');
7 integer offset = UInt(o1:'00');
8 constant integer nreg = 4;
```

Assembler Symbols

<T> Is the size specifier, encoded in "sz":

SZ	<t></t>
0	S
1	D

- <Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
- <offsf> Is the vector select offset, pointing to first of four consecutive vectors, encoded as "o1" field times 4.
- <offsl> Is the vector select offset, pointing to last of four consecutive vectors, encoded as "o1" field times 4 plus 3.
- <Zn1> For the two ZA quad-vectors variant: is the name of the first smable vector register of a multi-vector sequence, encoded as "Zn" times 2.

For the four ZA quad-vectors variant: is the name of the lirst sc. ble vector egister of a multi-vector sequence, encoded as "Zn" times 4.

<Tb> Is the size specifier, encoded in "sz":

SZ	<tb></tb>
0	В
1	Н

- <Zn4> Is the name of the fourth scalable vector in the rector in the rector is a multi-vector sequence, encoded as "Zn" times 4 plus 3.
- <Zn2> Is the name of the secone scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.
- <Zm1> For the two ZA vad-vectors riant: is the name of the first scalable vector register of a multi-vector s valuence, encoded a. 'Zm" times 2.

For the fr ZA que vectors variant: is the name of the first scalable vector register of a multi-vector equation equation of the scalable vector register of a multi-vector equation of the scalable vector register of a multi-vector equation of the scalable vector register of a multi-vector equation of the scalable vector register of a multi-vector equation of the scalable vector register of a multi-vector equation of the scalable vector register of a multi-vector equation of the scalable vector register of a multi-vector equation of the scalable vector equation of the s

- <Zm4> Is _____ me on ____ fourt' _scalable vector register of a multi-vector sequence, encoded as Zm" tin s 4 plus
- <Zm2> Is the times 2 plus 1.

```
CheckStreamingSVEAndZAEnabled();
1
2
   constant integer VL = CurrentVL;
   constant integer elements = VL DIV esize;
3
   integer vectors = VL DIV 8;
4
   integer vstride = vectors DIV nreg;
5
6
   bits(32) vbase = X[v, 32];
7
   integer vec = (UInt(vbase) + offset) MOD vstride;
8
   bits(VL) result;
9
   vec = vec - (vec MOD 4);
10
   for r = 0 to nreg-1
11
12
       bits(VL) operand1 = Z[n+r, VL];
       bits(VL) operand2 = Z[m+r, VL];
13
14
       for i = 0 to 3
           bits(VL) operand3 = ZAvector[vec + i, VL];
15
16
           for e = 0 to elements-1
17
                integer element1 = UInt(Elem[operand1, 4 * e + i, esize DIV 4]);
```

Chapter D1. SME instructions D1.1. SME and SME2 data-processing instructions

18	<pre>integer element2 = UInt(Elem[operand2, 4 * e + i, esize DIV 4]);</pre>
19	<pre>bits(esize) product = (element1 * element2)<esize-1:0>;</esize-1:0></pre>
20	<pre>Elem[result, e, esize] = Elem[operand3, e, esize] + product;</pre>
21	ZAvector[vec + i, VL] = result;
22	vec = vec + vstride;



Chapter D1. SME instructions D1.1. SME and SME2 data-processing instructions

D1.1.251 UMLSL (multiple and indexed vector)

Multi-vector unsigned integer multiply-subtract long by indexed element

The instruction operates on one, two, or four ZA double-vector groups.

This unsigned integer multiply-subtract long instruction multiplies each unsigned 16-bit element in the one, two, or four first source vectors with each unsigned 16-bit indexed element of the second source vector, widens each product to 32-bits and destructively subtracts these values from the corresponding 32-bit elements of the one, two, or four ZA double-vector groups.

The elements within the second source vector are specified using an immediate element index which selects the same element position within each 128-bit vector segment. The index range is from 0 to 7, encoded in 3 bits. The lowest of the two consecutive vector numbers forming the double-vector group within all, each half, or each quarter of the ZA array are selected by the sum of the vector select register mediate offset, modulo all, half, or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the \land operand consists of two or four ZA double-vector groups respectively. The VECTOR GROUP symbol is $_{\rm F}$ ferred for isassembly, but optional in assembler source code.

This instruction is unpredicated.

It has encodings from 3 classes: One ZA double-vector, Two double-vectors and Four ZA double-vectors

One ZA double-vector (FEAT_SME2)



```
if !HaveSME2() t<sup>1</sup> . UNDEFINED;
1
2
  constant integr esize = 32;
  integer v = U1 ('010' .v);
3
4
   integer n = UInt
                      יר '
                        •Zm):
5
  integer m
                VInt (
6
  intege offs.
                   = UIn. off
                                 (0')
   inter _ index = UInt(1
7
                              131);
  cor ant int /er nreg = 1;
8
```

```
Two ZA uble-vectors
(FEAT_SN. ?)
```



UMLSL ZA.S[<Wv>, <offsf>:<offsl>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H[<index>]

```
1 if !HaveSME2() then UNDEFINED;
```

```
2 constant integer esize = 32;
```

```
3 integer v = UInt('010':Rv);
```

```
4 integer n = UInt(Zn:'0');
5 integer m = UInt('0':Zm);
```

```
5 integer m = UInt('0':Zm);
6 integer offset = UInt(off2:'0')
```

```
6 integer offset = UInt(off2:'0');
7 integer index = UInt(i3h:i3l);
```

```
8 constant integer nreg = 2;
```

Four ZA double-vectors (FEAT_SME2) 0 0 0 1 Zm 0 0 Rv Zn 1 1 0 1 i3h 1 UMLSL ZA.S[<Wv>, <offsf>:<offsl>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H[<index>] 1 if !HaveSME2() then UNDEFINED; 2 constant integer esize = 32; integer v = UInt('010':Rv); 3 4 integer n = UInt(Zn:'00'); integer m = UInt('0':Zm); 5 integer offset = UInt(off2:'0'); 6 integer index = UInt(i3h:i3l); constant **integer** nreg = 4; **Assembler Symbols** / field. $\langle Wv \rangle$ Is the 32-bit name of the vector select register W8.³ 11, encc d in the For the one ZA double-vector variant: is the vector electron offset, r inting to first of two <offsf> consecutive vectors, encoded as "off3" field <u>s</u> 2. At: is the vector select offset, For the four ZA double-vectors and two A double vectors pointing to first of two consecutive v 'ors, enc int as "off2" field times 2. <offsl> For the one ZA double-vector riant: . e vector select offset, pointing to last of two consecutive vectors, encoder'ns "o. "" fiela res 2 plus 1. For the four ZA double- tors and two \double-vectors variant: is the vector select offset, pointing to last of tw _on. "utive vectors nooded as "off2" field times 2 plus 1. Is the name of the first source _____able vector register, encoded in the "Zn" field. $\langle Zn \rangle$ $\langle Zn1 \rangle$ For the two double-vectors val. at: is the name of the first scalable vector register of a multi-vec c sequer , encoded as "Zn" times 2. For the four <u>v</u> ouble-vec rs variant: is the name of the first scalable vector register of a tor sec, ince, i loded as "Zn" times 4. m <Zn4>is the nr le of the full in scalable vector register of a multi-vector sequence, encoded as "Zn" times ls of name of the second scalable vector register of a multi-vector sequence, encoded as <Zn2>"Zn nes 2 plus 1. Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field. $\langle Zm \rangle$ <index> Is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields. Operation

```
CheckStreamingSVEAndZAEnabled();
1
2
   constant integer VL = CurrentVL;
   constant integer elements = VL DIV esize;
3
4
   integer vectors = VL DIV 8;
   integer vstride = vectors DIV nreg;
5
   integer eltspersegment = 128 DIV esize;
6
   bits(32) vbase = X[v, 32];
7
   integer vec = (UInt(vbase) + offset) MOD vstride;
8
   bits(VL) result;
9
10
   vec = vec - (vec MOD 2);
11
12 for r = 0 to nreg-1
```
Chapter D1. SME instructions D1.1. SME and SME2 data-processing instructions

```
bits(VL) operand1 = Z[n+r, VL];
13
14
        bits(VL) operand2 = Z[m, VL];
15
        for i = 0 to 1
16
             bits(VL) operand3 = ZAvector[vec + i, VL];
17
             for e = 0 to elements-1
18
                  integer segmentbase = e - (e MOD eltspersegment);
                 integer s = 2 * segmentbase + index;
19
                 integer element1 = UInt(Elem[operand1, 2 * e + i, esize DIV 2]);
integer element2 = UInt(Elem[operand2, s, esize DIV 2]);
20
21
22
                 bits(esize) product = (element1 * element2)<esize-1:0>;
23
                 Elem[result, e, esize] = Elem[operand3, e, esize] - product;
24
             ZAvector[vec + i, VL] = result;
25
        vec = vec + vstride;
```

D1.1.252 UMLSL (multiple and single vector)

Multi-vector unsigned integer multiply-subtract long by vector

The instruction operates on one, two, or four ZA double-vector groups.

This unsigned integer multiply-subtract long instruction multiplies each unsigned 16-bit element in the one, two, or four first source vectors with each unsigned 16-bit element in the second source vector, widens each product to 32-bits and destructively subtracts these values from the corresponding 32-bit elements of the one, two, or four ZA double-vector groups. The lowest of the two consecutive vector numbers forming the double-vector group within all, each half, or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo all, half, or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA double-vector groups respectively. The VECTOR GROUP symbol is provide for disassembly, but optional in assembler source code.

This instruction is unpredicated.

It has encodings from 3 classes: One ZA double-vector, Two 7, double- tors and our ZA double-vectors

One ZA double-vector (FEAT_SME2)



```
UMLSL ZA.S[<Wv>, <offsf>:< ffsl . <Zn H, <Zm>.H
```

```
1 if !HaveSME2() then UNDEF
```

```
2 constant integer esize > J2;
```

```
3 integer v = UInt('010':xv);
```

- 4 integer n = UInt(Zn)
- 5 integer m = UInt(' :Zm);
- 6 integer offset = JInt(off3:'0');
- constant **inte** nreg



UMLSL ZA.S[<Wv>, <offsf>:<offsl>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H

- 1 if !HaveSME2() then UNDEFINED;
- 2 constant **integer** esize = 32;
- 3 integer v = UInt('010':Rv);
- 4 integer n = UInt(Zn);
- 5 integer m = UInt('0':Zm);
- 6 integer offset = UInt(off2:'0');
- 7 constant integer nreg = 2;

Four ZA double-vectors (FEAT_SME2)



UMLSL ZA.S[<Wv>, <offsf>:<offsl>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H

- 1 if !HaveSME2() then UNDEFINED;
- 2 constant integer esize = 32;
- 3 integer v = UInt('010':Rv);
- 4 integer n = UInt(Zn);
- 5 integer m = UInt('0':Zm);
- 6 integer offset = UInt(off2:'0');
 7
- 7 constant **integer** nreg = 4;

Assembler Symbols

- <Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.

For the four ZA double-vectors and two ZA double-vectors variables is the vector elect offset, pointing to first of two consecutive vectors, encoded as " .f2" field imes 2.

<offsl> For the one ZA double-vector variant: is the vect select offset, put is to last of two consecutive vectors, encoded as "off3" field times plus 1

For the four ZA double-vectors and two ZA double-vector variant: the vector select offset, pointing to last of two consecutive vector encode as "c ?" find times 2 plus 1.

- <Zn> Is the name of the first source scalab' vector rester, encoded in the "Zn" field.
- $\langle Zn1 \rangle$ Is the name of the first scalable vector region of a multi-vector sequence, encoded as "Zn".
- <Zn4> Is the name of the fourth sc?' ble ver regis of a multi-vector sequence, encoded as "Zn" plus 3 modulo 32.
- <Zn2> Is the name of the sound plable vector gister of a multi-vector sequence, encoded as "Zn" plus 1 modplo 32.
- <Zm> Is the name c .ne second source s 'able vector register Z0-Z15, encoded in the "Zm" field.

```
CheckStre qSVEA 7AEnabl ();
constar int, rVL Cur .tVL;
const .t inte :r eleme = VL DIV esize;
 1
2
 3
    int er vect s = VL DIV 8;
inte. VS rate Cors D
 4
                           .ors DIV nreg;
 5
    bits(3. pase = X[v, 32];
integer = (UInt(vbase) + offset) MOD vstride;
6
 7
    bits(VL) r lt;
vec = vec - vec MOD 2);
 8
9
10
11
    for r = 0 to nreg-1
         bits(VL) operand1 = Z[(n+r) MOD 32, VL];
12
13
         bits(VL) operand2 = Z[m, VL];
14
         for i = 0 to 1
15
             bits(VL) operand3 = ZAvector[vec + i, VL];
16
              for e = 0 to elements-1
                  integer element1 = UInt(Elem[operand1, 2 * e + i, esize DIV 2]);
17
18
                  integer element2 = UInt(Elem[operand2, 2 * e + i, esize DIV 2]);
19
                  bits(esize) product = (element1 * element2)<esize-1:0>;
20
                  Elem[result, e, esize] = Elem[operand3, e, esize] - product;
21
             ZAvector[vec + i, VL] = result;
22
        vec = vec + vstride;
```

D1.1.253 UMLSL (multiple vectors)

Multi-vector unsigned integer multiply-subtract long

The instruction operates on two or four ZA double-vector groups.

This unsigned integer multiply-subtract long instruction multiplies each unsigned 16-bit element in the two or four first source vectors with each unsigned 16-bit element in the two or four second source vectors, widens each product to 32-bits and destructively subtracts these values from the corresponding 32-bit elements of the two or four ZA double-vector groups. The lowest of the two consecutive vector numbers forming the double-vector group within each half or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA double-vector groups respectively. The VECTOR GROUP symbol is project of disassembly, but optional in assembler source code.

This instruction is unpredicated.

It has encodings from 2 classes: Two ZA double-vectors and F at ZA dou. 2-vector

Two ZA double-vectors (FEAT_SME2)



```
6 integer offset = UInt(off2:'0');
```

```
7 constant integer nreg = 4;
```

Assembler Symbols

<Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.

<offsf> Is the vector select offset, pointing to first of two consecutive vectors, encoded as "off2" field times 2.

Chapter D1. SME instructions

D1.1. SME and SME2 data-processing instructions

- <offsl> Is the vector select offset, pointing to last of two consecutive vectors, encoded as "off2" field times 2 plus 1.
- <Zn1> For the two ZA double-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.

For the four ZA double-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

- <Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.
- <Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.
- <Zm1> For the two ZA double-vectors variant: is the name of the first scale¹ vector register of a multi-vector sequence, encoded as "Zm" times 2.

For the four ZA double-vectors variant: is the name of the fin. calable vecto. egister of a multi-vector sequence, encoded as "Zm" times 4.

- <Zm4> Is the name of the fourth scalable vector register of multi-vector rue c, encoded as "Zm" times 4 plus 3.
- <Zm2> Is the name of the second scalable vector register o. _____iti-vecto__equence, encoded as "Zm" times 2 plus 1.

```
1
   CheckStreamingSVEAndZAEnabled();
2
   constant integer VL = CurrentVL;
   constant integer elements = V DIV
                                            ze;
3
4
   integer vectors = VL DIV 8;
   integer vstride = vectors
5
                                  1 nreq;
6
   bits(32) vbase = X[v, 3',
   integer vec = (UInt(vbase) + o. vet) MOD vstride;
7
   bits(VL) result;
8
9
                       5 2);
   vec = vec - (vec)
10
   for r = 0 to i g-1
11
       bits(VL) op nd^2 = Z[n+r].
bits(V^T) oper Z = Z[m+r]
12
                            Z[n+r. VL];
13
        bits(VT ) oper
                                    VL];
14
        for
                  to 1
            bits( ) oper
                            > = ZAvector[vec + i, VL];
15
16
            for r = 0 to el ments-1
                          lement1 = UInt(Elem[operand1, 2 * e + i, esize DIV 2]);
17
                integer element2 = UInt(Elem[operand2, 2 * e + i, esize DIV 2]);
18
19
                bits(esize) product = (element1 * element2)<esize-1:0>;
20
                Elem[result, e, esize] = Elem[operand3, e, esize] - product;
21
            ZAve or[vec + i, VL] = result;
22
        vec = vec + vstride;
```

D1.1.254 UMLSLL (multiple and indexed vector)

Multi-vector unsigned integer multiply-subtract long long by indexed element

The instruction operates on one, two, or four ZA quad-vector groups.

This unsigned integer multiply-subtract long long instruction multiplies each unsigned 8-bit or 16-bit element in the one, two, or four first source vectors with each unsigned 8-bit or 16-bit indexed element of second source vector, widens each product to 32-bits or 64-bits and destructively subtracts these values from the corresponding 32-bit or 64-bit elements of the one, two, or four ZA quad-vector groups.

The elements within the second source vector are specified using an immediate element index which selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 3 to 4 bits depending on the size of the element. The lowest of the four consecutive vector numbers forming the quad-vector group within a¹¹ balf, or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, n⁻¹ulo all, half, or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the 7 operation consists of voor four ZA quad-vector groups respectively. The VECTOR GROUP symbol is preferre for disassen. v, but ptional in assembler source code.

This instruction is unpredicated.

ID_AA64SMFR0_EL1.I16I64 indicates whether leng it integination is implemented.

It has encodings from 6 classes: One ZA quad-vector of 2-bit elements, One ZA quad-vector of 64-bit elements, Two ZA quad-vectors of 32-bit elements, Tv ZA quad-vectors of 64-bit elements, Four ZA quad-vectors of 64-bit elements

One ZA quad-vector of 32-bit e' ...ents (FEAT_SME2)



UMLSLL ZA.S[<Wv, offsf>: ffsl>], <Zn>.B, <Zm>.B[<index>]

```
if !H eSME2 ( then
1
                             NED.
2
  con
       ant int er esize
                            32:
3
  inte
        ~ v
                        :Rv);
4
               UInt(Zn);
  intege
             = UInt('0':Zm);
5
  integer
  integer on et = UInt(off2:'00');
6
  integer inde = UInt(i4h:i4l);
7
8
  constant integer nreg = 1;
```

One ZA quad-vector of 64-bit elements (FEAT SME 116164)



UMLSLL ZA.D[<Wv>, <offsf>:<offsl>], <Zn>.H, <Zm>.H[<index>]

1 if !(HaveSME2() && HaveSMEI16I64()) then UNDEFINED;

```
2 constant integer esize = 64;
```

```
3 integer v = UInt('010':Rv);
```

```
4 integer n = UInt(Zn);
```

Chapter D1. SME instructions D1.1. SME and SME2 data-processing instructions

- integer m = UInt('0':Zm); 5
- 6 integer offset = UInt(off2:'00');
- integer index = UInt(i3h:i3l); 7
- 8 constant integer nreg = 1;

Two ZA quad-vectors of 32-bit elements (FEAT_SME2)



UMLSLL ZA.S[<Wv>, <offsf>:<offsl>{, VGx2}], { <Zn1>.B-<Zn2>.B }, <Zm>.B[<index>]

- if !HaveSME2() then UNDEFINED; 1
- 2 constant integer esize = 32;
- 3 integer v = UInt('010':Rv);
- 4 integer n = UInt(Zn:'0'); 5
- integer m = UInt('0':Zm); 6
- integer offset = UInt(o1:'00'); integer index = UInt(i4h:i4l); 7
- 8 constant integer nreg = 2;

Two ZA quad-vectors of 64-bit elements (FEAT SME I16I64)



UMLSLL ZA.D[<Wv>, <off >> offsl>{, Vc }], { <Zn1>.H-<Zn2>.H }, <Zm>.H[<index>]

1

- constant integer < _ze = 64; integer v = UIn⁺ 010':R^x); integer n = U⁺ (Zn:'0' 2
- 3
- 4
- 5
- 6 7
- integer m = UIh. '0' ...);
 integer offset = _ (o1:'00'
 integer = UIh. 'i3h:i',
 constat inte ;r nreg 8

Fou 'A qv f 32-bit elements (FEA1_ .£2) 0 0 0 0 1 1 0 Zm 1 1 0 0 1 1 Rv 0 i4h 0 0 υJ

UMLSLL ZA.S[<Wv>, <offsf>:<offsl>{, VGx4}], { <Zn1>.B-<Zn4>.B }, <Zm>.B[<index>]

- 1 if !HaveSME2() then UNDEFINED;
- 2 constant integer esize = 32;
- 3 integer v = UInt('010':Rv);
- 4 integer n = UInt(Zn:'00');
- 5 integer m = UInt('0':Zm);
- 6 integer offset = UInt(o1:'00'); 7
- integer index = UInt(i4h:i4l);
- 8 constant integer nreg = 4;

Four ZA quad-vectors of 64-bit elements (FEAT_SME_I16I64)

1

Ls

i4l

01



UMLSLL ZA.D[<Wv>, <offsf>:<offsl>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H[<index>]

- if !(HaveSME2() && HaveSMEI16I64()) then UNDEFINED;
- 2 constant integer esize = 64;
- 3 integer v = UInt('010':Rv); 4
- integer n = UInt(Zn:'00');
- 5 integer m = UInt('0':Zm);
- 6 integer offset = UInt(o1:'00');
- integer index = UInt(i3h:i3l); 7
- 8 constant integer nreg = 4;

Assembler Symbols

<Wv> Is the 32-bit name of the vector select register W8-W11, encode in the "Rv" t ld.

For the one ZA quad-vector of 32-bit elements and or $\angle A$ quad-vector of f bit elements <offsf> variant: is the vector select offset, pointing to first four corecutive .ors, encoded as "off2" field times 4.

> four. quad-y cors of 64-bit elements, For the four ZA quad-vectors of 32-bit eler two ZA quad-vectors of 32-bit elements a two. A quad- vector of 64-bit elements variant: is the vector select offset, pointing to st of four consecutive vectors, encoded as "o1" field times 4.

For the one ZA quad-vector of 32 it elen. 's and one ZA quad-vector of 64-bit elements <offsl> variant: is the vector selec' ifset, porting to st of four consecutive vectors, encoded as "off2" field times 4 plus

> For the four ZA qu. -vecto. of 32-bit elements, four ZA quad-vectors of 64-bit elements, two ZA quad-vectors of 32-bit ments and two ZA quad-vectors of 64-bit elements variant: is the vector ect offset, pointing last of four consecutive vectors, encoded as "o1" field times 4 ply 5.

- Is the name ^cth *inst source scalable vector register, encoded in the "Zn" field.* $\langle Zn \rangle$
- <Zn1> F inc 'o ZA ad-ve is of 32-bit elements and two ZA quad-vectors of 64-bit elements √ariant: the name the first scalable vector register of a multi-vector sequence, encoded

he four ZA quad-vectors of 32-bit elements and four ZA quad-vectors of 64-bit elements var. t: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn imes 4.

- <Zn4>Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.
- Is the name of the second scalable vector register of a multi-vector sequence, encoded as <Zn2>"Zn" times 2 plus 1.
- <Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.
- <index> For the four ZA quad-vectors of 32-bit elements, one ZA quad-vector of 32-bit elements and two ZA quad-vectors of 32-bit elements variant: is the element index, in the range 0 to 15, encoded in the "i4h:i4l" fields.

For the four ZA quad-vectors of 64-bit elements, one ZA quad-vector of 64-bit elements and two ZA quad-vectors of 64-bit elements variant: is the element index, in the range 0 to 7, encoded in the "i3h:i31" fields.

```
CheckStreamingSVEAndZAEnabled();
 1
    constant integer VL = CurrentVL;
 2
3
   constant integer elements = VL DIV esize;
4 integer vectors = VL DIV 8;
    integer vstride = vectors DIV nreg;
5
   integer eltspersegment = 128 DIV esize;
6
7
    bits(32) vbase = X[v, 32];
    integer vec = (UInt(vbase) + offset) MOD vstride;
8
9
    bits(VL) result;
   vec = vec - (vec MOD 4);
10
11
12
    for r = 0 to nreg-1
        bits(VL) operand1 = Z[n+r, VL];
13
14
        bits(VL) operand2 = Z[m, VL];
15
         for i = 0 to 3
             bits(VL) operand3 = ZAvector[vec + i, VL];
16
17
             for e = 0 to elements-1
18
                  integer segmentbase = e - (e MOD eltspersegme. `;
19
                  integer s = 4 * segmentbase + index;
                 integer element1 = UInt(Elem[operand1, 4 e + i, size
integer element2 = UInt(Elem[operand2, , esize DIV ');
bits(esize) product = (element1 * el ent2)<e ze-1:0</pre>
20
                                                                              rize DIV
                                                                                        ·]);
21
22
23
                  Elem[result, e, esize] = Elem[operal <sup>2</sup>, e,
                                                                    _ze] - product;
24
             ZAvector[vec + i, VL] = result;
25
        vec = vec + vstride;
```

D1.1.255 UMLSLL (multiple and single vector)

Multi-vector unsigned integer multiply-subtract long long by vector

The instruction operates on one, two, or four ZA quad-vector groups.

This unsigned integer multiply-subtract long long instruction multiplies each unsigned 8-bit or 16-bit element in the one, two, or four first source vectors with each unsigned 8-bit or 16-bit element in the second source vector, widens each product to 32-bits or 64-bits and destructively subtracts these values from the corresponding 32-bit or 64-bit elements of the one, two, or four ZA quad-vector groups. The lowest of the four consecutive vector numbers forming the quad-vector group within all, each half, or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo all, half, or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA quad-vector groups respectively. The VECTOR GROUP symbol is preferred for disasser to prove optional in assembler source code.

This instruction is unpredicated.

ID_AA64SMFR0_EL1.I16I64 indicates whether the 16-bit int _er varian. implem_ted.

It has encodings from 3 classes: One ZA quad-vector, Tv ZA quad ectors a our ZA quad-vectors

One ZA quad-vector (FEAT_SME2)



UMLSLL ZA.<T>[<Wv>, <of t>:<offsl> <Zn>.<Tb>, <Zm>.<Tb>

```
if !HaveSME2() then UNL_FINED;
1
                                           hen UNDEFINED;
2
   if sz == '1' && !Hay CMEI16I64()
3
   constant integer /
                         .ze = 32 << UIn 3z);</pre>
                       010':R');
4
   integer v = UIn+
5
   integer n = U<sup>T</sup> .(Zn);
   integer m = UIh
integer offset =
6
                        0'
                              .0 ;
7
                           .(off2:
8
   constan+
                    er ni
                             = 1;
```

Tw 4A quad ectors (FEA SN .2)



UMLSLL ZA.<T>[<Wv>, <offsf>:<offsl>{, VGx2}], { <Zn1>.<Tb>-<Zn2>.<Tb> }, <Zm>.<Tb>

```
1
  if !HaveSME2() then UNDEFINED;
  if sz == '1' && !HaveSMEI16I64() then UNDEFINED;
2
3
  constant integer esize = 32 << UInt(sz);</pre>
  integer v = UInt('010':Rv);
4
5
  integer n = UInt(Zn);
  integer m = UInt('0':Zm);
6
  integer offset = UInt(o1:'00');
7
  constant integer nreg = 2;
8
```

Four ZA quad-vectors (FEAT_SME2)



UMLSLL ZA.<T>[<Wv>, <offsf>:<offsl>{, VGx4}], { <Zn1>.<Tb>-<Zn4>.<Tb> }, <Zm>.<Tb>

```
1 if !HaveSME2() then UNDEFINED;
```

- 2 if sz == '1' && !HaveSMEI16I64() then UNDEFINED;
- 3 constant integer esize = 32 << UInt(sz);</pre>
- 4 integer v = UInt('010':Rv);
- 5 integer n = UInt(Zn);
- 6 integer m = UInt('0':Zm);
- 7 integer offset = UInt(o1:'00');
- 8 constant **integer** nreg = 4;

Assembler Symbols

<T> Is the size specifier, encoded in "sz":

SZ	<t></t>
0	S
1	D

- <Wv> Is the 32-bit name of the vector select regimer v, W11, ordgiment with the "Rv" field.
- $\langle offsf \rangle$ For the one ZA quad-vector variant is the vector select outset, pointing to first of four consecutive vectors, encoded as "off2 vectors 4.

For the four ZA quad-vectors and vo ZA ad-vectors variant: is the vector select offset, pointing to first of four conjuctive v tors, ended as "o1" field times 4.

<offsl> For the one ZA quad our variant: is evector select offset, pointing to last of four consecutive vectors, encode of "off2" field times 4 plus 3.

For the four Z quad-vectors an wo ZA quad-vectors variant: is the vector select offset, pointing to 'st of four consecutive vectors, encoded as "o1" field times 4 plus 3.

- <Zn> Is the nan. If the 1st source scalable vector register, encoded in the "Zn" field.
- <Zn1> Is "me on first so uble vector register of a multi-vector sequence, encoded as "Zn".
- <Tb> is the size specifier coded in "sz":



- <Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" plus 3 modulo 32.
- <Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" plus 1 modulo 32.
- <Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

```
1 CheckStreamingSVEAndZAEnabled();
2 constant integer VL = CurrentVL;
3 constant integer elements = VL DIV esize;
4 integer vectors = VL DIV 8;
5 integer vstride = vectors DIV nreg;
6 bits(32) vbase = X[v, 32];
7 integer vec = (UInt(vbase) + offset) MOD vstride;
```

```
8 bits(VL) result;
9
    vec = vec - (vec MOD 4);
10
11
    for r = 0 to nreg-1
        bits(VL) operand1 = Z[(n+r) MOD 32, VL];
12
13
        bits(VL) operand2 = Z[m, VL];
        for i = 0 to 3
14
15
             bits(VL) operand3 = ZAvector[vec + i, VL];
             for e = 0 to elements-1
16
                 integer element1 = UInt(Elem[operand1, 4 * e + i, esize DIV 4]);
17
                 integer element2 = UInt(Elem[operand2, 4 * e + i, esize DIV 4]);
18
                 bits(esize) product = (element1 * element2)<esize-1:0>;
Elem[result, e, esize] = Elem[operand3, e, esize] - product;
19
20
21
             ZAvector[vec + i, VL] = result;
22
        vec = vec + vstride;
```

D1.1.256 UMLSLL (multiple vectors)

Multi-vector unsigned integer multiply-subtract long long

The instruction operates on two or four ZA quad-vector groups.

This unsigned integer multiply-subtract long long instruction multiplies each unsigned 8-bit or 16-bit element in the two or four first source vectors with each unsigned 8-bit or 16-bit element in the one, two, or four second source vectors, widens each product to 32-bits or 64-bits and destructively subtracts these values from the corresponding 32-bit or 64-bit elements of the two or four ZA quad-vector groups. The lowest of the four consecutive vector numbers forming the quad-vector group within each half or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA quad-vector groups respectively. The VECTOR GROUP symbol is preferred for disasser "t optional in assembler source code.

This instruction is unpredicated.

ID_AA64SMFR0_EL1.I16I64 indicates whether the 16-bit int er varian, implem ited.

It has encodings from 2 classes: Two ZA quad-vectors an our ZA ad-vectors

Two ZA quad-vectors (FEAT_SME2)

1 2

5

7



```
if !HaveSME2() then UNDEFINED;
  if sz == '1' && !HaveSMEI16I64() then UNDEFINED;
2
3
   constant integer esize = 32 << UInt(sz);</pre>
4
  integer v = UInt('010':Rv);
5
  integer n = UInt(Zn:'00');
  integer m = UInt(Zm:'00');
6
  integer offset = UInt(o1:'00');
7
```

```
8 constant integer nreg = 4;
```

Assembler Symbols

<T> Is the size specifier, encoded in "sz":

SZ	<t></t>
0	S
1	D

- <Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
- <offsf> Is the vector select offset, pointing to first of four consecutive vectors, encoded as "o1" field times 4.
- <offsl> Is the vector select offset, pointing to last of four consecutive vectors, encoded as "o1" field times 4 plus 3.
- <Zn1> For the two ZA quad-vectors variant: is the name of the first smable vector register of a multi-vector sequence, encoded as "Zn" times 2.

For the four ZA quad-vectors variant: is the name of the lirst sc. ble vector egister of a multi-vector sequence, encoded as "Zn" times 4.

<Tb> Is the size specifier, encoded in "sz":

SZ	<tb></tb>
0	В
1	Н

- <Zn4> Is the name of the fourth scalable vector in the rector in the rector is a multi-vector sequence, encoded as "Zn" times 4 plus 3.
- <Zn2> Is the name of the secone scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.
- <Zm1> For the two ZA vad-vectors riant: is the name of the first scalable vector register of a multi-vector s valuence, encoded a. 'Zm" times 2.

For the fr ZA que vectors variant: is the name of the first scalable vector register of a multi-vector equation equation of the scalable vector register of a multi-vector equation of the scalable vector register of a multi-vector equation of the scalable vector register of a multi-vector equation of the scalable vector register of a multi-vector equation of the scalable vector register of a multi-vector equation of the scalable vector register of a multi-vector equation of the scalable vector register of a multi-vector equation of the scalable vector equation of the s

- <Zm4> Is _____ me on ____ fourt' _scalable vector register of a multi-vector sequence, encoded as Zm" tin s 4 plus
- <Zm2> Is the times 2 plus 1.

```
CheckStreamingSVEAndZAEnabled();
1
2
   constant integer VL = CurrentVL;
   constant integer elements = VL DIV esize;
3
   integer vectors = VL DIV 8;
4
   integer vstride = vectors DIV nreg;
5
6
   bits(32) vbase = X[v, 32];
7
   integer vec = (UInt(vbase) + offset) MOD vstride;
8
   bits(VL) result;
9
   vec = vec - (vec MOD 4);
10
   for r = 0 to nreg-1
11
12
       bits(VL) operand1 = Z[n+r, VL];
       bits(VL) operand2 = Z[m+r, VL];
13
14
       for i = 0 to 3
           bits(VL) operand3 = ZAvector[vec + i, VL];
15
16
           for e = 0 to elements-1
17
                integer element1 = UInt(Elem[operand1, 4 * e + i, esize DIV 4]);
```

Chapter D1. SME instructions D1.1. SME and SME2 data-processing instructions

18	<pre>integer element2 = UInt(Elem[operand2, 4 * e + i, esize DIV 4]);</pre>
19	<pre>bits(esize) product = (element1 * element2)<esize-1:0>;</esize-1:0></pre>
20	<pre>Elem[result, e, esize] = Elem[operand3, e, esize] - product;</pre>
21	<pre>ZAvector[vec + i, VL] = result;</pre>
22	<pre>vec = vec + vstride;</pre>



D1.1.257 UMOPA (2-way)

Unsigned integer sum of outer products and accumulate

This instruction works with a 32-bit element ZA tile.

The unsigned integer sum of outer products and accumulate instructions multiply the sub-matrix in the first source vector by the sub-matrix in the second source vector. The first source holds $SVL_S \times 2$ sub-matrix of unsigned 16-bit integer values, and the second source holds $2 \times SVL_S$ sub-matrix of unsigned 16-bit integer values.

Each source vector is independently predicated by a corresponding governing predicate. When a 16-bit source element is inactive, it is treated as having the value 0.

The resulting $SVL_S \times SVL_S$ widened 32-bit integer sum of outer products is then destructively added to the 32-bit integer destination tile. This is equivalent to performing a 2-way dot product and accumulate to each of the destination tile elements.

Each 32-bit container of the first source vector holds 2 consecutive (umn element) f each row of a $SVL_S \times 2$ sub-matrix, and each 32-bit container of the second source vector bolds _____nsecutive rc elements of each column of a $2 \times SVL_S$ sub-matrix.

SME2 (FEAT SME2)

0 0 0 0 0 Pm 7n 7Ada 1 1 0 Pn 0 1 0 Ls Lu0

UMOPA <7Ada>.S. <Pn>/M. 7m>.H7.n>

```
1
  if !HaveSME2() then UNDEF
```

- 2 constant **integer** esize
- 3 integer a = UInt(Pn);
- integer b = UInt(Pm' 4
- integer n = UInt(/ /; 5
- 6 integer m = UIn⁺ _m);
- integer da = V (t (ZAda) 7
- boolean sub_op 8 TAL! UE :
- boolean unsigned

Asser her Syr ols

- <ZAda> 's the ZA tile ZA0-ZA3, encoded in the "ZAda" field.
 - Is name of the first governing scalable predicate register P0-P7, encoded in the "Pn" field. <Pn>
 - <Pm> Is the . ne of the second governing scalable predicate register P0-P7, encoded in the "Pm" field.
 - $\langle Zn \rangle$ Is the name of the first source scalable vector register, encoded in the "Zn" field.
 - <Zm>Is the name of the second source scalable vector register, encoded in the "Zm" field.

```
1
   CheckStreamingSVEAndZAEnabled();
2
   constant integer VL = CurrentVL;
3
   constant integer PL = VL DIV 8;
Δ
   constant integer dim = VL DIV esize;
5
   bits(PL) mask1 = P[a, PL];
6
   bits(PL) mask2 = P[b, PL];
   bits(VL) operand1 = Z[n, VL];
7
   bits(VL) operand2 = Z[m, VL];
8
9
   bits(dim*dim*esize) operand3 = ZAtile[da, esize, dim*dim*esize];
10
   bits(dim*dim*esize) result;
```

```
Chapter D1. SME instructions
D1.1. SME and SME2 data-processing instructions
```

```
integer prod;
11
12
13
   for row = 0 to dim-1
14
      for col = 0 to dim-1
15
          bits(esize) sum = Elem[operand3, row*dim+col, esize];
16
          for k = 0 to 1
              if ActivePredicateElement(mask1, 2*row + k, esize DIV 2) &&
17
18
                     ActivePredicateElement(mask2, 2*col + k, esize DIV 2) then
                 19
20
21
                  if sub_op then prod = -prod;
22
                  sum = sum + prod;
23
24
          Elem[result, row*dim+col, esize] = sum;
25
26
   ZAtile[da, esize, dim*dim*esize] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of
 - The values of the data supplied in any of its o rand registers when s governing predicate registers contain the same value for each execution.
 - The values of the NZCV flags.
- The response of this instruction to asyn fromous eleptions de s not vary based on:
 - The values of the data supplied in a. of .s operand registers when its governing predicate registers contain the same value for each vecution
 - The values of the NZC' dags.

D1.1.258 UMOPA (4-way)

Unsigned integer sum of outer products and accumulate

The 8-bit integer variant works with a 32-bit element ZA tile.

The 16-bit integer variant works with a 64-bit element ZA tile.

The unsigned integer sum of outer products and accumulate instructions multiply the sub-matrix in the first source vector by the sub-matrix in the second source vector. In case of the 8-bit integer variant, the first source holds $SVL_S \times 4$ sub-matrix of unsigned 8-bit integer values, and the second source holds $4 \times SVL_S$ sub-matrix of unsigned 8-bit integer values. In case of the 16-bit integer variant, the first source holds $SVL_D \times 4$ sub-matrix of unsigned 16-bit integer values, and the second source holds $4 \times SVL_S \times 4$ sub-matrix of unsigned 16-bit integer values.

Each source vector is independently predicated by a corresponding governing predicate. When an 8-bit source element in case of 8-bit integer variant or a 16-bit source element in case 10-c. Pteger variant is Inactive, it is treated as having the value 0.

The resulting $SVL_S \times SVL_S$ widened 32-bit integer or $SVL_D \times SVL_D$ where d64-bit in ger sum of outer products is then destructively added to the 32-bit integer or 64-bit integer destination ile, resp. dively for 8-bit integer and 16-bit integer instruction variants. This is equivalent to perfer hing a 4-way derived and accumulate to each of the destination tile elements.

In case of the 8-bit integer variant, each 32-bit container on dirst sour vector holds 4 consecutive column elements of each row of a $SVL_S \times 4$ sub-matrix and the h 32-b. Fonto der of the second source vector holds 4 consecutive row elements of each column of a f SVL_S sub-matrix. The second source vector holds 4 consecutive row of the first source vector holds 4 consecutive row elements of each column of a f SVL_S sub-matrix. The second source vector holds 4 consecutive row elements of each column of a f SVL_S sub-matrix. The second source vector holds 4 consecutive row elements of each column of a $4 \times SVL_D$ sub-matrix.

It has encodings from 2 classes by bit and 64-bit



```
UMOPA <ZAda>.D, <Pn>/M, <Pm>/M, <Zn>.H, <Zm>.H
1
   if !HaveSMEI16I64() then UNDEFINED;
2
   constant integer esize = 64;
  integer a = UInt(Pn);
3
4
   integer b = UInt(Pm);
5
   integer n = UInt(Zn);
6 integer m = UInt(Zm);
   integer da = UInt(ZAda);
7
8
   boolean sub_op = FALSE;
   boolean op1_unsigned = TRUE;
9
10 boolean op2_unsigned = TRUE;
   Assembler Symbols
```

- <ZAda> For the 32-bit variant: is the name of the ZA tile ZA0-ZA3, encoded in the "ZAda" field.
 - For the 64-bit variant: is the name of the ZA tile ZA0-ZA7, enclued in the Ada" field.
 - <Pn> Is the name of the first governing scalable predicate register P0-. encoded in t "Pn" field.
 - <Pm> Is the name of the second governing scalable predicate _gister P0-1 encod _ in the "Pm" field.
 - <Zn> Is the name of the first source scalable vector regis. enc .ed in the "Zn" field.
 - <Zm> Is the name of the second source scalable y registe 'ncode' in the "Zm" field.

Operation

```
1
   CheckStreamingSVEAndZAEnabled();
2
   constant integer VL = CurrentVL;
3
   constant integer PL = VL DIV
   constant integer dim = VL D
4
                                 esize;
5
   bits(PL) mask1 = P[a, PL]
   bits(PL) mask2 = P[b, P'
6
   bits(VL) operand1 = Z[n, VL];
7
8
   bits(VL) operand2 = [m, VL];
   bits(dim*dim*esize operand3 = ZAt >[da, esize, dim*dim*esize];
9
10
   bits(dim*dim*es' .) resu't;
11
   integer prod
12
13
   for row = 0 to d_{\perp}
14
        for
                 0 to
                        m-1
15
           bits ( ize)
                              _lem[operand3, row*dim+col, esize];
                   0 to 3
16
            for }
17
                       wePredicateElement(mask1, 4*row + k, esize DIV 4) &&
18
                        ActivePredicateElement(mask2, 4*col + k, esize DIV 4) then
19
                    prod = (Int(Elem[operand1, 4*row + k, esize DIV 4], op1_unsigned) *
                            Int(Elem[operand2, 4*col + k, esize DIV 4], op2_unsigned));
20
21
                    if sub_op then prod = -prod;
22
                    sum = sum + prod;
23
24
            Elem[result, row*dim+col, esize] = sum;
25
26
   ZAtile[da, esize, dim*dim*esize] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
 - The values of the data supplied in any of its operand registers when its governing predicate registers contain the same value for each execution.
 - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:

- The values of the data supplied in any of its operand registers when its governing predicate registers contain the same value for each execution.
- The values of the NZCV flags.



D1.1.259 UMOPS (2-way)

Unsigned integer sum of outer products and subtract

This instruction works with a 32-bit element ZA tile.

The unsigned integer sum of outer products and subtract instructions multiply the sub-matrix in the first source vector by the sub-matrix in the second source vector. The first source holds $SVL_S \times 2$ sub-matrix of unsigned 16-bit integer values, and the second source holds $2 \times SVL_S$ sub-matrix of unsigned 16-bit integer values.

Each source vector is independently predicated by a corresponding governing predicate. When a 16-bit source element is inactive, it is treated as having the value 0.

The resulting $SVL_S \times SVL_S$ widened 32-bit integer sum of outer products is then destructively subtracted from the 32-bit integer destination tile. This is equivalent to performing a 2-way dot product and subtract from each of the destination tile elements.

Each 32-bit container of the first source vector holds 2 consecutive summer of each row of a $SVL_S \times 2$ sub-matrix, and each 32-bit container of the second source vector holds 2 onsecutive root elements of each column of a $2 \times SVL_S$ sub-matrix.

SME2 (FEAT_SME2)

0 0 0 0 0 Pm 7n 7Ada 1 1 0 Pn 1 1 0 Ĺs Lu0

UMOPS <ZAda>.S, <Pn>/M, <P >/M, <Zn>.h. <Zm>.H

```
1 if !HaveSME2() then UNDEF
```

- 2 constant **integer** esize = 32
- 3 integer a = UInt(Pn);
- 4 integer b = UInt (Pm'
- 5 integer n = UInt(5 /;
- 6 integer m = UIn⁺ am);
- 7 integer da = r (t (ZAda
- 8 boolean sub_op TRUT
 9 boolean unsigned AUE;
- 9 boolean unsigned

Asser her Syr ols

- <ZAda> Is the ZA tile ZA0-ZA3, encoded in the "ZAda" field.
 - <Pn> Is name of the first governing scalable predicate register P0-P7, encoded in the "Pn" field.
 - <Pm> Is the . ne of the second governing scalable predicate register P0-P7, encoded in the "Pm" field.
 - <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
 - <Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

```
1
   CheckStreamingSVEAndZAEnabled();
2
   constant integer VL = CurrentVL;
3
   constant integer PL = VL DIV 8;
Δ
   constant integer dim = VL DIV esize;
5
   bits(PL) mask1 = P[a, PL];
6
   bits(PL) mask2 = P[b, PL];
   bits(VL) operand1 = Z[n, VL];
7
   bits(VL) operand2 = Z[m, VL];
8
9
   bits(dim*dim*esize) operand3 = ZAtile[da, esize, dim*dim*esize];
10
   bits(dim*dim*esize) result;
```

```
Chapter D1. SME instructions
D1.1. SME and SME2 data-processing instructions
```

```
integer prod;
11
12
13
   for row = 0 to dim-1
14
      for col = 0 to dim-1
15
          bits(esize) sum = Elem[operand3, row*dim+col, esize];
16
          for k = 0 to 1
              if ActivePredicateElement(mask1, 2*row + k, esize DIV 2) &&
17
18
                     ActivePredicateElement(mask2, 2*col + k, esize DIV 2) then
                 19
20
21
                  if sub_op then prod = -prod;
22
                  sum = sum + prod;
23
24
          Elem[result, row*dim+col, esize] = sum;
25
26
   ZAtile[da, esize, dim*dim*esize] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of
 - The values of the data supplied in any of its o rand registers when s governing predicate registers contain the same value for each execution.
 - The values of the NZCV flags.
- The response of this instruction to asyn fromous eleptions de s not vary based on:
 - The values of the data supplied in a. of .s operand registers when its governing predicate registers contain the same value for each vecution
 - The values of the NZC' dags.

D1.1.260 UMOPS (4-way)

Unsigned integer sum of outer products and subtract

The 8-bit integer variant works with a 32-bit element ZA tile.

The 16-bit integer variant works with a 64-bit element ZA tile.

The unsigned integer sum of outer products and subtract instructions multiply the sub-matrix in the first source vector by the sub-matrix in the second source vector. In case of the 8-bit integer variant, the first source holds $SVL_S \times 4$ sub-matrix of unsigned 8-bit integer values, and the second source holds $4 \times SVL_S$ sub-matrix of unsigned 8-bit integer values. In case of the 16-bit integer variant, the first source holds $SVL_D \times 4$ sub-matrix of unsigned 16-bit integer values.

Each source vector is independently predicated by a corresponding governing predicate. When an 8-bit source element in case of 8-bit integer variant or a 16-bit source element in case 10-c. Pteger variant is Inactive, it is treated as having the value 0.

The resulting $SVL_S \times SVL_S$ widened 32-bit integer or $SVL_D \times SVL_D$ where d64-bit in ger sum of outer products is then destructively subtracted from the 32-bit integer or 64 at integer stination dile, respectively for 8-bit integer and 16-bit integer instruction variants. This is equivalent to perform, a vary dot product and subtract from each of the destination tile elements.

In case of the 8-bit integer variant, each 32-bit container of first sour vector holds 4 consecutive column elements of each row of a $SVL_S \times 4$ sub-matrix and the 32-bit container of the second source vector holds 4 consecutive row elements of each column of a $4 \times SVL_S$ sub-matrix. The second source vector holds 4 consecutive row of a $SVL_D \times 4$ sub-matrix, and each 64-bit container of the second source vector holds 4 consecutive row elements of each column of a $4 \times SVL_S$ sub-matrix.

ID_AA64SMFR0_EL1.I16I64 *i* ________ cates whe. r the 1______ bit integer variant is implemented.

It has encodings from 2 classes by bit and 64-bit



```
UMOPS <ZAda>.D, <Pn>/M, <Pm>/M, <Zn>.H, <Zm>.H
1
   if !HaveSMEI16I64() then UNDEFINED;
2
   constant integer esize = 64;
  integer a = UInt(Pn);
3
4
   integer b = UInt(Pm);
5
   integer n = UInt(Zn);
  integer m = UInt(Zm);
6
   integer da = UInt(ZAda);
7
8
   boolean sub_op = TRUE;
   boolean op1_unsigned = TRUE;
9
10 boolean op2_unsigned = TRUE;
   Assembler Symbols
```

- <ZAda> For the 32-bit variant: is the name of the ZA tile ZA0-ZA3, encoded in the "ZAda" field.
 - For the 64-bit variant: is the name of the ZA tile ZA0-ZA7, enclued in the Ada" field.
 - <Pn> Is the name of the first governing scalable predicate register P0-. encoded in t "Pn" field.
 - <Pm> Is the name of the second governing scalable predicate _gister P0-, encod in the "Pm" field.
 - <Zn> Is the name of the first source scalable vector regis. enc .ed in the "Zn" field.
 - <Zm> Is the name of the second source scalable y registe 'ncode' in the "Zm" field.

Operation

```
1
   CheckStreamingSVEAndZAEnabled();
2
   constant integer VL = CurrentVL;
3
   constant integer PL = VL DIV
   constant integer dim = VL D
4
                                 esize;
5
   bits(PL) mask1 = P[a, PL]
   bits(PL) mask2 = P[b, P'
6
7
   bits(VL) operand1 = Z[n, VL];
8
   bits(VL) operand2 = [m, VL];
   bits(dim*dim*esize operand3 = ZAt >[da, esize, dim*dim*esize];
9
10
   bits(dim*dim*es' .) resu't;
11
   integer prod
12
13
   for row = 0 to d_{\perp}
14
        for
                 0 to
                        m-1
15
            bits ( ize)
                              _lem[operand3, row*dim+col, esize];
                   = 0 to 3
16
            for }
17
                        wePredicateElement(mask1, 4*row + k, esize DIV 4) &&
18
                        ActivePredicateElement(mask2, 4*col + k, esize DIV 4) then
19
                    prod = (Int(Elem[operand1, 4*row + k, esize DIV 4], op1_unsigned) *
                            Int(Elem[operand2, 4*col + k, esize DIV 4], op2_unsigned));
20
21
                    if sub_op then prod = -prod;
22
                    sum = sum + prod;
23
24
            Elem[result, row*dim+col, esize] = sum;
25
26
   ZAtile[da, esize, dim*dim*esize] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
 - The values of the data supplied in any of its operand registers when its governing predicate registers contain the same value for each execution.
 - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:

- The values of the data supplied in any of its operand registers when its governing predicate registers contain the same value for each execution.
- The values of the NZCV flags.



D1.1.261 UQCVT (two registers)

Multi-vector unsigned saturating extract narrow

Saturate the unsigned integer value in each element of the two source vectors to half the orginal source element width, and place the results in the half-width destination elements.

This instruction is unpredicated.

SME2 (FEAT_SME2)



UQCVT <Zd>.H, { <Zn1>.S-<Zn2>.S }

```
1 if !HaveSME2() then UNDEFINED;
```

- 2 constant integer esize = 16;
- 3 integer n = UInt(Zn:'0');
- 4 integer d = UInt(Zd);

Assembler Symbols

- <Zd> Is the name of the destination scalab¹ vector revister, encoded in the "Zd" field.
- <Zn1> Is the name of the first scalable vector noise of a multi-vector sequence, encoded as "Zn" times 2.
- <Zn2> Is the name of the second scalable velor register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.

```
CheckStreamingSV _______...abled();
1
  constant integ VL = C lentVL;
2
  constant integ elem _s = VL DIV (2 * esize);
3
4
  bits(VL) result;
5
  6
7
8
9
10
           .em[result, r*elements + e, esize] = UnsignedSat(element, esize);
11
  Z[d, VL] = sult;
12
```

D1.1.262 UQCVT (four registers)

Multi-vector unsigned saturating extract narrow

Saturate the unsigned integer value in each element of the four source vectors to quarter the orginal source element width, and place the results in the quarter-width destination elements.

This instruction is unpredicated.

SME2 (FEAT_SME2)

 31
 24
 23
 22
 21

 1
 1
 0
 0
 0
 0
 1
 Sz
 0
 1
 1 0 0 1 1 1 1 1 0 0 Zn 7d 0 0 1 Lu

```
UQCVT <Zd>.<T>, { <Zn1>.<Tb>-<Zn4>.<Tb> }
```

```
1 if !HaveSME2() then UNDEFINED;
```

```
2 constant integer esize = 8 << UInt(sz);</pre>
```

```
3 integer n = UInt(Zn:'00');
```

```
4 integer d = UInt(Zd);
```

Assembler Symbols

SZ

1

- <Zd> Is the name of the destination scalab¹ vector revister, encoded in the "Zd" field.
- <T> Is the size specifier, encoded in "<z":

SZ	<t></t>
0	В
1	Н

- <Zn1> Is the name of t first scalable register of a multi-vector sequence, encoded as "Zn" times 4.
- <Tb> Is the siz pecifier needed in "sz":

<<u>Tb</u>

<Zn4> is a name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" time 4 plus 3.

```
CheckStreamingSVEEnabled();
1
2
   constant integer VL = CurrentVL;
3
   constant integer elements = VL DIV (4 * esize);
4
   bits(VL) result;
5
6
   for r = 0 to 3
       bits(VL) operand = Z[n+r, VL];
7
8
       for e = 0 to elements-1
9
           integer element = UInt(Elem[operand, e, 4 * esize]);
10
           Elem[result, r*elements + e, esize] = UnsignedSat(element, esize);
11
12 Z[d, VL] = result;
```

D1.1.263 UQCVTN

Multi-vector unsigned saturating extract narrow and interleave

Saturate the unsigned integer value in each element of the four source vectors to quarter the orginal source element width, and place the four-way interleaved results in the quarter-width destination elements.

This instruction is unpredicated.

SME2 (FEAT_SME2)

 31
 24
 23
 22
 21

 1
 1
 0
 0
 0
 0
 1
 Sz
 0
 1
 1 0 0 1 1 1 1 1 0 0 Zn 1 7d 0 1 Ν_ Lu

UQCVTN <Zd>.<T>, { <Zn1>.<Tb>-<Zn4>.<Tb> }

```
1 if !HaveSME2() then UNDEFINED;
```

```
2 constant integer esize = 8 << UInt(sz);</pre>
```

```
3 integer n = UInt(Zn:'00');
```

```
4 integer d = UInt(Zd);
```

Assembler Symbols

SZ

1

- <Zd> Is the name of the destination scalab¹ vector revister, encoded in the "Zd" field.
- <T> Is the size specifier, encoded in "<z":

SZ	<t></t>
0	В
1	Н
Ţ	п

- <Zn1> Is the name of t first scalable stor register of a multi-vector sequence, encoded as "Zn" times 4.
- <Tb> Is the siz pecifier needed in "sz":

<<u>Tb</u>

<Zn4> e name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" tim, 4 plus 3.

```
CheckStreamingSVEEnabled();
1
2
    constant integer VL = CurrentVL;
3
    constant integer elements = VL DIV (4 * esize);
4
   bits(VL) result;
5
6
    for e = 0 to elements-1
        for i = 0 to 3
7
8
             bits(VL) operand = Z[n+i, VL];
             integer element = UInt(Elem[operand, e, 4 * esize]);
Elem[result, 4*e + i, esize] = UnsignedSat(element, esize);
9
10
11
12 Z[d, VL] = result;
```

D1.1.264 UQRSHR (two registers)

Multi-vector unsigned saturating rounding shift right narrow by immediate

Shift right by an immediate value, the unsigned integer value in each element of the two source vectors and place the rounded results in the half-width destination elements. Each result element is saturated to the half-width N-bit element's unsigned integer range 0 to (2^N) -1. The immediate shift amount is an unsigned value in the range 1 to 16.

This instruction is unpredicated.

SME2 (FEAT_SME2)



- Assembler Symbols
- $\langle Zd \rangle$ Is the name of the destination sc 'able v. register, encoded in the "Zd" field.
- <Zn1> Is the name of the first scale, e vector register of a multi-vector sequence, encoded as "Zn" times 2.
- <Zn2> Is the name of the sound lable vector, gister of a multi-vector sequence, encoded as "Zn" times 2 plus 1.
- <const> Is the immer '.e shift amount, in u range 1 to 16, encoded in the "imm4" field.

```
CheckStr TSVEL bled();
constation interver VL our ntVL;
1
2
3
    cons' nt inte ar eleme.
                                  = VL DIV (2 * esize);
4
    bit VL) res
    integ rr id_cons. 1 << (shift-1);</pre>
5
6
7
    for r = 0
                 o 1
        bits(VL operand = Z[n+r, VL];
for e = . to elements-1
8
9
10
             bits(2 * esize) element = Elem[operand, e, 2 * esize];
11
             integer res = (UInt(element) + round_const) >> shift;
12
             Elem[result, r*elements + e, esize] = UnsignedSat(res, esize);
13
14
   Z[d, VL] = result;
```

D1.1.265 UQRSHR (four registers)

Multi-vector unsigned saturating rounding shift right narrow by immediate

Shift right by an immediate value, the unsigned integer value in each element of the four source vectors and place the rounded results in the quarter-width destination elements. Each result element is saturated to the quarter-width N-bit element's unsigned integer range 0 to (2^N) -1. The immediate shift amount is an unsigned value in the range 1 to number of bits per source element.

This instruction is unpredicated.

SME2 (FEAT_SME2)

1

2

4

5

6 7

9



Assembler Symbols

<Zd $>$	Is the name of the d	.cinatic	calable vec	etcr register,	encoded in the	"Zd" field.
---------	----------------------	----------	-------------	----------------	----------------	-------------

Is the size specier, encoded in <T> ze":

- tsize $\mathbf{f} \geq$ 00 SERVF 01 R
- <Zn1> Is the the first scalable vector register of a multi-vector sequence, encoded as "Zn" 4
- < Tb >Is the 'ize specifier, encoded in "tsize":

tsize	<tb></tb>
00	RESERVED
01	S
1x	D

- <Zn4>Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.
- <const> Is the immediate shift amount, in the range 1 to number of bits per source element, encoded in "tsize:imm5".

```
CheckStreamingSVEEnabled();
1
```

```
2
  constant integer VL = CurrentVL;
```

```
constant integer elements = VL DIV (4 * esize);
3
```

Chapter D1. SME instructions D1.1. SME and SME2 data-processing instructions

```
4 bits(VL) result;
5
   integer round_const = 1 << (shift-1);</pre>
6
7
   for r = 0 to 3
       bits(VL) operand = Z[n+r, VL];
8
9
       for e = 0 to elements-1
10
           bits(4 * esize) element = Elem[operand, e, 4 * esize];
            integer res = (UInt(element) + round_const) >> shift;
11
12
           Elem[result, r*elements + e, esize] = UnsignedSat(res, esize);
13
14 Z[d, VL] = result;
```

D1.1.266 UQRSHRN

1

2

7

9

Multi-vector unsigned saturating rounding shift right narrow by immediate and interleave

Shift right by an immediate value, the unsigned integer value in each element of the four source vectors and place the four-way interleaved rounded results in the quarter-width destination elements. Each result element is saturated to the quarter-width N-bit element's unsigned integer range 0 to (2^N) -1. The immediate shift amount is an unsigned value in the range 1 to number of bits per source element.

This instruction is unpredicated.

SME2 (FEAT_SME2)



Assembler Symbols

<Zd $>$	Is the name of the d	.cinatic	calable vec	etcr register,	encoded in the	"Zd" field.
---------	----------------------	----------	-------------	----------------	----------------	-------------

Is the size specier, encoded in <T> ze":

- tsize $\mathbf{f} \geq$ 00 SERVF 01 R
- <Zn1> Is the the first scalable vector register of a multi-vector sequence, encoded as "Zn" 4
- < Tb >Is the ize specifier, encoded in "tsize":

tsize	<tb></tb>
00	RESERVED
01	S
1x	D

- <Zn4>Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.
- <const> Is the immediate shift amount, in the range 1 to number of bits per source element, encoded in "tsize:imm5".

```
CheckStreamingSVEEnabled();
1
```

```
2
  constant integer VL = CurrentVL;
```

```
constant integer elements = VL DIV (4 * esize);
3
```

Chapter D1. SME instructions D1.1. SME and SME2 data-processing instructions

```
4 bits(VL) result;
5
   integer round_const = 1 << (shift-1);</pre>
6
7
    for e = 0 to elements-1
8
        for i = 0 to 3
            bits(VL) operand = Z[n+i, VL];
bits(4 * esize) element = Elem[operand, e, 4 * esize];
9
10
             integer res = (UInt(element) + round_const) >> shift;
11
12
             Elem[result, 4*e + i, esize] = UnsignedSat(res, esize);
13
14 Z[d, VL] = result;
```

D1.1.267 URSHL (multiple and single vector)

Multi-vector unsigned rounding shift left by vector

Shift active unsigned elements of the two or four first source vectors by corresponding elements of the second source vector and destructively place the rounded results in the corresponding elements of the two or four first source vectors. A positive shift amount performs a left shift, otherwise a right shift by the negated shift amount is performed.

This instruction is unpredicated.

It has encodings from 2 classes: Two registers and Four registers

```
Two registers
   (FEAT_SME2)
                 0
                    0
                       0
                          0
                             0
                               1
                                  size
                                        1
                                           0
                                                 Ζm
                                                           0
                                                              1
                                                                 0
                                                                       0
                                                                            0
                                                                               0
                                                                                           Zdr
                                                                                                  1
                                                                                                   URSHL { <Zdn1>.<T>-<Zdn2>.<T> }, { <Zdn1>
                                                                                (7m) <T
                                                              <70
1
   if !HaveSME2() then UNDEFINED;
2
   constant integer esize = 8 << UInt(size)
3
   integer dn = UInt(Zdn:'0');
   integer m = UInt('0':Zm);
4
5
   constant integer nreg = 2;
   Four registers
   (FEAT SME2)
                                  s.
                                                                                         7dn
               1
                 0
                    0
                       0
                          0
                               1
                                       1
                                          0
                                                           0
                                                              1
                                                                 0
                                                                       0
                                                                            0
                                                                               0
                                                                                 0
                                                                                                  1
   URSHL
            { < 7.dn
                              Zdn4>
                                             { <Zdn1>.<T>-<Zdn4>.<T> }, <Zm>.<T>
                                     \langle T \rangle \}.
1
   if !Hav
                    then 'DEF'
                                  ; <u>۵</u>.
2
           c inte r esiz
                                 << UInt(size);
   const
   int er dn = Int(Zdn:'\0');
3
4
   inte
            m
                          ~m);
               .teger nreg = 4;
5
   consta
```

Assembler **b**, **b**ols

<Zdn1> For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2.

For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4.

<T> Is the size specifier, encoded in "size":

size	< T >
00	В
01	Η
10	S
11	D

<Zdn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as

"Zdn" times 4 plus 3.

- <Zdn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2 plus 1.
 - <Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

```
1
    CheckStreamingSVEEnabled();
   constant integer VL = CurrentVL;
2
3
    constant integer elements = VL DIV esize;
4
    array [0..3] of bits(VL) results;
5
6
    for r = 0 to nreg-1
        bits(VL) operand1 = Z[dn+r, VL];
bits(VL) operand2 = Z[m, VL];
7
8
9
        for e = 0 to elements-1
10
             integer element = UInt(Elem[operand1, e, esize])
11
             integer shift = ShiftSat(SInt(Elem[operand2, e, es.
                                                                        ), esize)
12
             integer res;
             if shift >= 0 then
13
14
                 res = element << shift;</pre>
15
             else
16
                 shift = -shift;
17
                 res = (\text{element} + (1 << (\text{shift} - 1)))
                                                                 .ft;
             Elem[results[r], e, esize] = res
18
                                                          :0>;
19
20
    for r = 0 to nreg-1
21
        Z[dn+r, VL] = results[r];
```

D1.1.268 URSHL (multiple vectors)

Multi-vector unsigned rounding shift left

Shift active unsigned elements of the two or four first source vectors by corresponding elements of the two or four second source vectors and destructively place the rounded results in the corresponding elements of the two or four first source vectors. A positive shift amount performs a left shift, otherwise a right shift by the negated shift amount is performed.

This instruction is unpredicated.

It has encodings from 2 classes: Two registers and Four registers



Assembler Symbols

<Zdn1> For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2.

For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4.

<T> Is the size specifier, encoded in "size":

size	< T >
00	В
01	Η
10	S
11	D
- <Zdn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4 plus 3.
- <Zdn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2 plus 1.
- <Zm1> For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 2.

For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 4.

- <Zm4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zm" times 4 plus 3.
- <Zm2> Is the name of the second scalable vector register of a multi-vector ence, encoded as "Zm" times 2 plus 1.

```
1
   CheckStreamingSVEEnabled();
   constant integer VL = CurrentVL;
2
3
   constant integer elements = VL DIV esize;
4
   array [0..3] of bits(VL) results;
5
   for r = 0 to nreg-1
6
       bits(VL) operand1 = Z[dn+r, VL];
7
8
       bits(VL) operand2 = Z[m+r, VL];
9
       for e = 0 to elements-1
10
           integer element = UInt(Elem[or
                                          ar , e, es_ ]);
11
           12
           integer res;
13
           if shift >= 0 then
14
               res = element
                               shift;
15
           else
16
               shift = -sh.ft;
           res = (e oment + (1
Elem[resul [r], e, esize]
17
                                     (shift - 1))) >> shift;
18
                                      res<esize-1:0>;
19
   for r = 0 to r = -1
20
21
       Z[dn+r, VL]
                    res
                        cs[r];
```

D1.1.269 USDOT (multiple and indexed vector)

Multi-vector unsigned by signed integer dot-product by indexed element

The instruction operates on two or four ZA single-vector groups.

The unsigned by signed integer dot product instruction computes the dot product of four unsigned 8-bit integer values held in each 32-bit element of the two or four first source vectors and four signed 8-bit integer values in the corresponding indexed 32-bit element of the second source vector. The widened dot product result is destructively added to the corresponding 32-bit element of the two or four ZA single-vector groups.

The groups within the second source vector are specified using an immediate element index which selects the same group position within each 128-bit vector segment. The index range is from 0 to 3, encoded in 2 bits. The vector numbers forming the single-vector group within each half or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the the of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA oper disconsists of voor four ZA single-vector groups respectively. The VECTOR GROUP symbol is preferred for disconsists of the vector on a sembler source code.

This instruction is unpredicated.

It has encodings from 2 classes: Two ZA single-vectors a. Four' A single-vectors

Two ZA single-vectors (FEAT_SME2)



USDOT ZA.S[<Wv>, <offs>{, VGx4}], { <Zn1>.B-<Zn4>.B }, <Zm>.B[<index>]

```
1 if !HaveSME2() then UNDEFINED;
```

```
2 integer v = UInt('010':Rv);
```

```
3 constant integer esize = 32;
4 integer n = UInt(Zn:'00');
```

```
4 integer n = UInt(Zn:'00');
5 integer m = UInt('0':Zm);
```

```
6 integer offset = UInt(off3);
```

```
7 integer index = UInt(i2);
```

```
8 constant integer nreg = 4;
```

Assembler Symbols

- <Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
- <offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.
- <Zn1> For the two ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.

For the four ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

- <Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.
- <Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.
- <Zm> Is the name of the second source scalable vector register Z0-Z['], encoded in "Zm" field.

<index> Is the element index, in the range 0 to 3, encoded in the "?" field

```
CheckStreamingSVEAndZAEnabled();
 1
 2
   constant integer VL = CurrentVL;
   constant integer elements = VL DIV esize;
 3
4
   integer vectors = VL DIV 8;
    integer vstride = vectors DIV nreg;
 5
   integer eltspersegment = 128 DIV esiz
6
7
   bits(32) vbase = X[v, 32];
 8
   integer vec = (UInt(vbase) + offs
                                             MOD
                                                     ride;
9
   bits(VL) result;
10
11
   for r = 0 to nreg-1
                              , nh
12
        bits(VL) operand1 =
                                      VL];
        bits(VL) operand2 = Z[m,
13
14
        bits(VL) operanc' = ZAvector
                                          ec, VL];
15
        for e = 0 to \epsilon .ments-1
            bits(esi ) sum = Elem[oper.d3, e, esize];
intege segment se = e - (e MOD eltspersegment);
16
17
             intege.
18
                       = s _mentbase + index;
19
             for i =
                   `tege. lemen* = UInt(Elem[operand1, 4 * e + i, esize DIV 4]);
20
                 1 :eger m 2 = SInt(Elem[operand2, 4 * s + i, esize DIV 4]);
21
                             .1ement1 * element2;
                 s n = sum
22
                       '' e, esize] = sum;
'___ = result;
23
             Elei
24
             c' ⊥[vec,
25
              vec + vstride;
        vec
```

D1.1.270 USDOT (multiple and single vector)

Multi-vector unsigned by signed integer dot-product by vector

The instruction operates on two or four ZA single-vector groups.

The unsigned by signed integer dot product instruction computes the dot product of four unsigned 8-bit integer values held in each 32-bit element of the two or four first source vectors and four signed 8-bit integer values in the corresponding 32-bit element of the second source vector. The widened dot product result is destructively added to the corresponding 32-bit element of the two or four ZA single-vector groups. The vector numbers forming the single-vector group within each half or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The VECTOR GROUP symbol is preferred for disasser to prior optional in assembler source code.

This instruction is unpredicated.

It has encodings from 2 classes: Two ZA single-vectors and Fr ZA sing, vectors

Two ZA single-vectors (FEAT_SME2)



USDOT ZA.S[<Wv>, <offs>{, '3x2}] { <Zn .B-<Zn2>.B }, <Zm>.B

```
1 if !HaveSME2() then UNDEF
```

- 2 integer v = UInt('010';" /;
- 3 constant integer esize = 32;
- 4 integer n = UInt(Zn)
- 5 integer m = UInt(' :Zm);
- 6 integer offset JInt(of 3);
 7
- 7 constant **inte**: nreg



1 0 0 1 0 0 1 1 Zm 0 Rv 1 0 1 Zn 0 1 off3

USDOT ZA.S[<Wv>, <offs>{, VGx4}], { <Zn1>.B-<Zn4>.B }, <Zm>.B

```
1 if !HaveSME2() then UNDEFINED;
```

```
2 integer v = UInt('010':Rv);
```

```
3 constant integer esize = 32;
```

```
4 integer n = UInt(Zn);
```

```
5 integer m = UInt('0':Zm);
6 integer offset = UInt(off3);
```

```
o integer offset = Ulnt(off3);
7 constant integer nreg = 4;
```

Assembler Symbols

- <Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
- <offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.
- <Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn".

off3

0 1 Lu

- <Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" plus 3 modulo 32.
- <Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" plus 1 modulo 32.
- <Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

```
1
    CheckStreamingSVEAndZAEnabled();
    constant integer VL = CurrentVL;
2
 3
    constant integer elements = VL DIV esize;
4
   integer vectors = VL DIV 8;
    integer vstride = vectors DIV nreg;
 5
 6
   bits(32) vbase = X[v, 32];
    integer vec = (UInt(vbase) + offset) MOD vstride;
7
8
    bits(VL) result;
9
10
   for r = 0 to nreg-1
11
        bits(VL) operand1 = Z[(n+r) MOD 32, VL];
        bits(VL) operand2 = Z[m, VL];
12
13
        bits(VL) operand3 = ZAvector[vec, VL];
14
        for e = 0 to elements-1
15
             bits(esize) sum = Elem[operand3, e, esize
16
             for i = 0 to 3
                 integer element1 = UInt(Elem / Grow '1,
integer element2 = SInt(Ele operar 2,
17
                                                                          esize DIV 4]);
18
                                                                         esize DIV 4]);
                                                             4
19
                 sum = sum + element1 * e]
                                               .ent2;
        Elem[result, e, esize] = sum;
ZAvector[vec, VL] = result;
20
21
22
        vec = vec + vstride;
```

D1.1.271 USDOT (multiple vectors)

Multi-vector unsigned by signed integer dot-product

The instruction operates on two or four ZA single-vector groups.

The unsigned by signed integer dot product instruction computes the dot product of four unsigned 8-bit integer values held in each 32-bit element of the two or four first source vectors and four signed 8-bit integer values in the corresponding 32-bit element of the two or four second source vectors. The widened dot product result is destructively added to corresponding 32-bit element of the two or four ZA single-vector groups. The vector numbers forming the single-vector group within each half or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The VECTOR GROUP symbol is preferred for disasser to prior optional in assembler source code.

This instruction is unpredicated.

It has encodings from 2 classes: Two ZA single-vectors and Fr ZA sing, vectors

Two ZA single-vectors (FEAT_SME2)

0 0 0 0 Zn off3 0 1 0 0 0 1 1 7n Rν 0 1

USDOT ZA.S[<Wv>, <offs>{, VGx2 { <_>.B-<Zn2>.B }, { <Zm1>.B-<Zm2>.B }

- 1 if !HaveSME2() then UNDEFIN .;
- 2 integer v = UInt('010':R^{*}
- 3 constant integer esize 32;
- 4 integer n = UInt(Zn:'0');
- 5 integer m = UInt(Zr 0');
- 6 integer offset = .nt(off3); 7 constant integr nreg =

Four ZA single-vec





USDOT ZA.S. Wv>, <offs>{, VGx4}], { <Zn1>.B-<Zn4>.B }, { <Zm1>.B-<Zm4>.B }

```
1 if !HaveSME2() then UNDEFINED;
2 integer v = UInt('010':Rv);
```

- 3 constant integer esize = 32;
- 4 integer n = UInt(Zn:'00');
- 5 integer m = UInt(Zm:'00');
- 6 integer offset = UInt(off3);
- 7 constant **integer** nreg = 4;

Assembler Symbols

- <Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
- <offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.
- <Zn1> For the two ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.

For the four ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

- <Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.
- <Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.
- <Zm1> For the two ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 2.

For the four ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 4.

- <Zm4> Is the name of the fourth scalable vector register of a multi-vector spence, encoded as "Zm" times 4 plus 3.
- <Zm2> Is the name of the second scalable vector register of a multi-ctor sequence encoded as "Zm" times 2 plus 1.

```
CheckStreamingSVEAndZAEnabled();
1
 2
    constant integer VL = CurrentVL;
    constant integer elements = VL DIV esize;
3
4
    integer vectors = VL DIV 8;
 5
    integer vstride = vectors DIV nreg;
    bits(32) vbase = X[v, 32];
 6
    integer vec = (UInt(vbase) + offsot) Mc
 7
                                                            ide;
8
    bits(VL) result;
9
10
    for r = 0 to nreg-1
         bits(VL) operand1 = Z' r, VL];
bits(VL) operand2 = _m, VL];
bits(VL) operand3 = ZAvect [vec, VL];
11
12
13
         for e = 0 to ele ents-1
14
15
              bits(esize sum = Elem[ope nd3, e, esize];
                         to 3
16
              for i =
                   ir .ger elc .nt1 = UInt(Elem[operand1, 4 * e + i, esize DIV 4]);
17
                   in er c ment2 = SInt(Elem[operand2, 4 * e + i, esize DIV 4]);
sum + elem 1 * element2;
18
19
           esult =, es' =] = sum;
cctor[ +c, VL, - sult;
cc = vec + vstride
20
21
         7.A
22
```

D1.1.272 USMLALL (multiple and indexed vector)

Multi-vector unsigned by signed integer multiply-add long long by indexed element

The instruction operates on one, two, or four ZA quad-vector groups.

This unsigned by signed integer multiply-add long long instruction multiplies each unsigned 8-bit element in the one, two, or four first source vectors with each signed 8-bit indexed element of the second source vector, widens each product to 32-bits and destructively adds these values to the corresponding 32-bit elements of the one, two, or four ZA quad-vector groups.

The elements within the second source vector are specified using an immediate element index which selects the same element position within each 128-bit vector segment. The element index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 4 bits. The lowest of the four consecutive vector numbers forming the quad-vector group within all, each half, or each quarter of the 7 are selected by the sum of the vector select register and immediate offset, modulo all, half, or quarter ' _ number 'ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA opcond consists of yoor four ZA quad-vector groups respectively. The VECTOR GROUP symbol is preferred f disass obly, but or conal in assembler source code.

This instruction is unpredicated.

It has encodings from 3 classes: One ZA quad-vector Two 2 quad-vector and Four ZA quad-vectors

```
One ZA quad-vector
(FEAT_SME2)
```



```
if !HaveSME2() t<sup>1</sup> . UNDEFINED;
1
2
  constant integr esize = 32;
  integer v = U1 ('010' .v);
3
4
   integer n = UInt
                     יר
                        ·Zm);
5
  integer m
               VInt (
                               00');
6
  intege Jffs.
                   = UIn. off
   inter _ index = UInt(1
7
                             _41);
  cor ant int /er nreg = 1;
8
```

```
Two ZA ad-vectors
(FEAT_SN. ?)
```



USMLALL ZA.S[<Wv>, <offsf>:<offsl>{, VGx2}], { <Zn1>.B-<Zn2>.B }, <Zm>.B[<index>]

```
1 if !HaveSME2() then UNDEFINED;
```

```
2 constant integer esize = 32;
```

```
3 integer v = UInt('010':Rv);
4 integer n = UInt(Zn:'0');
```

```
5 integer m = UInt('0':Zm);
```

```
6 integer m = OINt('0':2m);
6 integer offset = UInt(o1:'00');
```

```
7 integer index = UInt(i4h:i4l);
```

```
8 constant integer nreg = 2;
```

Four ZA quad-vectors (FEAT_SME2) 1 0 0 0 0 0 1 0 Zm Rv i4h Zn 0 0 1 1 0 i4l USMLALL ZA.S[<Wv>, <offsf>:<offsl>{, VGx4}], { <Zn1>.B-<Zn4>.B }, <Zm>.B[<index>] 1 if !HaveSME2() then UNDEFINED; 2 constant integer esize = 32; integer v = UInt('010':Rv); 3 4 integer n = UInt(Zn:'00'); integer m = UInt('0':Zm); 5 integer offset = UInt(o1:'00'); 6 integer index = UInt(i4h:i4l); constant **integer** nreg = 4; **Assembler Symbols** / field. $\langle Wv \rangle$ Is the 32-bit name of the vector select register W8.³ 11, encc d in the <offsf> For the one ZA quad-vector variant: is the vector le offset, printing to first of four consecutive vectors, encoded as "off2" field s 4. For the four ZA quad-vectors and two A quad- ectors va *c*: is the vector select offset, pointing to first of four consecutive vertors, en vert <offsl> For the one ZA quad-vector v. ant: is vector select offset, pointing to last of four consecutive vectors, encoder' as "o " field res 4 plus 3. For the four ZA quad-v ors and two \quad vectors variant: is the vector select offset, pointing to last of for con. outive vector. ncoded as "o1" field times 4 plus 3. Is the name of the first source _____able vector register, encoded in the "Zn" field. $\langle Zn \rangle$

<Zn1> For the two A quad-vectors varia, c is the name of the first scalable vector register of a multi-vec a sequer , encoded as "Zn" times 2.

For the four $^{\prime}$ und-vect $^{\prime}$ variant: is the name of the first scalable vector register of a m $^{\prime}$ tor sec nce, ϵ oded as "Zn" times 4.

- <Zn4> Is the name of the full in scalable vector register of a multi-vector sequence, encoded as "Zn" times r
- <Zn2> ls e name of the second scalable vector register of a multi-vector sequence, encoded as "Zn nes 2 plus 1.
- <Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

<index> Is the element index, in the range 0 to 15, encoded in the "i4h:i4l" fields.

```
CheckStreamingSVEAndZAEnabled();
1
   constant integer VL = CurrentVL;
2
   constant integer elements = VL DIV esize;
3
4
   integer vectors = VL DIV 8;
   integer vstride = vectors DIV nreg;
5
   integer eltspersegment = 128 DIV esize;
6
   bits(32) vbase = X[v, 32];
7
   integer vec = (UInt(vbase) + offset) MOD vstride;
8
   bits(VL) result;
9
10
   vec = vec - (vec MOD 4);
11
12 for r = 0 to nreg-1
```

```
bits(VL) operand1 = Z[n+r, VL];
13
14
         bits(VL) operand2 = Z[m, VL];
15
          for i = 0 to 3
16
              bits(VL) operand3 = ZAvector[vec + i, VL];
17
               for e = 0 to elements-1
18
                    integer segmentbase = e - (e MOD eltspersegment);
                    integer s = 4 * segmentbase + index;
19
                   integer element1 = UInt(Elem[operand1, 4 * e + i, esize DIV 4]);
integer element2 = SInt(Elem[operand2, s, esize DIV 4]);
bits(esize) product = (element1 * element2)<esize-1:0>;
20
21
22
23
                   Elem[result, e, esize] = Elem[operand3, e, esize] + product;
24
               ZAvector[vec + i, VL] = result;
25
         vec = vec + vstride;
```

D1.1.273 USMLALL (multiple and single vector)

Multi-vector unsigned by signed integer multiply-add long long by vector

The instruction operates on one, two, or four ZA quad-vector groups.

This unsigned by signed integer multiply-add long long instruction multiplies each unsigned 8-bit element in the one, two, or four first source vectors with each signed 8-bit element in the second source vector, widens each product to 32-bits and destructively adds these values to the corresponding 32-bit elements of the one, two, or four ZA quad-vector groups. The lowest of the four consecutive vector numbers forming the quad-vector group within all, each half, or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo all, half, or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA quad-vector groups respectively. The VECTOR GROUP symbol is preferred for disasser to prove optional in assembler source code.

This instruction is unpredicated.

It has encodings from 3 classes: One ZA quad-vector, Two Z' quad-vector and Fo ZA quad-vectors

One ZA quad-vector (FEAT_SME2)



USMLALL ZA.S[<Wv>, <offsf>:< ffsl . <Zn B, <Zm>.B

```
1 if !HaveSME2() then UNDEF
```

- 2 constant **integer** esize >2;
- 3 integer v = UInt('010':xv);
- 4 integer n = UInt(Zn'
- 5 integer m = UInt(' :Zm);
- 6 integer offset = JInt(off2:'00');

'n.

7 constant **inte** nreg



USMLALL ZA.S[<Wv>, <offsf>:<offsl>{, VGx2}], { <Zn1>.B-<Zn2>.B }, <Zm>.B

- 1 if !HaveSME2() then UNDEFINED;
- 2 constant integer esize = 32;
- 3 integer v = UInt('010':Rv);
- 4 integer n = UInt(Zn);
- 5 integer m = UInt('0':Zm);
- 6 integer offset = UInt(o1:'00');
- 7 constant integer nreg = 2;

Four ZA quad-vectors (FEAT_SME2)



USMLALL ZA.S[<Wv>, <offsf>:<offsl>{, VGx4}], { <Zn1>.B-<Zn4>.B }, <Zm>.B

- 1 if !HaveSME2() then UNDEFINED;
- 2 constant integer esize = 32;
- 3 integer v = UInt('010':Rv);
- 4 integer n = UInt(Zn);
- 5 integer m = UInt('0':Zm);
- 6 integer offset = UInt(o1:'00');
 7 constant integer preg = 4:
- 7 constant **integer** nreg = 4;

Assembler Symbols

- <Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
- <offsf> For the one ZA quad-vector variant: is the vector select offset. to first of four consecutive vectors, encoded as "off2" field times 4.

For the four ZA quad-vectors and two ZA quad-vectors varial is the vector lect offset, pointing to first of four consecutive vectors, encoded as '1" field mes 4.

<offsl> For the one ZA quad-vector variant: is the vector select of the point of the last of four consecutive vectors, encoded as "off2" field times the plus 3

For the four ZA quad-vectors and two ZA *curd*-vector variant: *i* ne vector select offset, pointing to last of four consecutive vector , encured as 'curd and 'games 4 plus 3.

- <Zn> Is the name of the first source scalab' vector rester. encoded in the "Zn" field.
- $\langle Zn1 \rangle$ Is the name of the first scalable \sim ctor region of a multi-vector sequence, encoded as "Zn".
- <Zn4> Is the name of the fourth sc?' ble ver regis of a multi-vector sequence, encoded as "Zn" plus 3 modulo 32.
- <Zn2> Is the name of the sond plable vector gister of a multi-vector sequence, encoded as "Zn" plus 1 modplo 32.
- <Zm> Is the name one second source so "able vector register Z0-Z15, encoded in the "Zm" field.

```
CheckStre qSVEA 7AEnabl ();
constar int, rVL Cur .tVL;
const .t inte :r eleme = VL DIV esize;
 1
2
 3
    int er vect s = VL DIV 8;
inte. VS rate Cors D
4
                           .ors DIV nreg;
 5
    bits(3. pase = X[v, 32];
integer = (UInt(vbase) + offset) MOD vstride;
6
 7
    bits(VL) r lt;
vec = vec - vec MOD 4);
 8
9
10
11
    for r = 0 to nreg-1
         bits(VL) operand1 = Z[(n+r) MOD 32, VL];
12
13
         bits(VL) operand2 = Z[m, VL];
14
         for i = 0 to 3
15
             bits(VL) operand3 = ZAvector[vec + i, VL];
16
              for e = 0 to elements-1
                  integer element1 = UInt(Elem[operand1, 4 * e + i, esize DIV 4]);
17
18
                  integer element2 = SInt(Elem[operand2, 4 * e + i, esize DIV 4]);
19
                  bits(esize) product = (element1 * element2)<esize-1:0>;
20
                  Elem[result, e, esize] = Elem[operand3, e, esize] + product;
21
             ZAvector[vec + i, VL] = result;
22
        vec = vec + vstride;
```

D1.1.274 USMLALL (multiple vectors)

Multi-vector unsigned by signed integer multiply-add long long

The instruction operates on two or four ZA quad-vector groups.

This unsigned by signed integer multiply-add long long instruction multiplies each unsigned 8-bit element in the two or four first source vectors with each signed 8-bit element in the two or four second source vectors, widens each product to 32-bits and destructively adds these values to the corresponding 32-bit elements of the two or four ZA quad-vector groups. The lowest of the four consecutive vector numbers forming the quad-vector group within each half or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA quad-vector groups respectively. The VECTOR GROUP symbol is preferred for disasser to prove optional in assembler source code.

This instruction is unpredicated.

It has encodings from 2 classes: Two ZA quad-vectors and For ZA quad- ctors

```
Two ZA quad-vectors (FEAT_SME2)
```



```
USMLALL ZA.S[<Wv>, <offsf>:< ffsl . VGx. . { <Zn1>.B-<Zn2>.B }, { <Zm1>.B-<Zm2>.B }
```

```
1 if !HaveSME2() then UNDEF
```

```
2 constant integer esize s2;
3 integer y = UInt('010'.sy):
```

```
3 integer v = UInt('010':xv);
4 integer n = UInt(Zn )');
```

```
5 integer m = UInt(" .'0');
```

```
6 integer offset - JInt(o1 '00');
```

7 constant **inte** nreg

```
Four ZA vecto
(FEAT ME2
```



USMLALL ZA.S[<Wv>, <offsf>:<offsl>{, VGx4}], { <Zn1>.B-<Zn4>.B }, { <Zm1>.B-<Zm4>.B }

```
1 if !HaveSME2() then UNDEFINED;
```

```
2 constant integer esize = 32;
```

```
3 integer v = UInt('010':Rv);
```

```
4 integer n = UInt(Zn:'00');
```

```
5 integer m = UInt(Zm:'00');
```

```
6 integer offset = UInt(o1:'00');
```

```
7 constant integer nreg = 4;
```

Assembler Symbols

- <Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
- <offsf> Is the vector select offset, pointing to first of four consecutive vectors, encoded as "o1" field times 4.

Chapter D1. SME instructions

D1.1. SME and SME2 data-processing instructions

- <offsl> Is the vector select offset, pointing to last of four consecutive vectors, encoded as "o1" field times 4 plus 3.
- <Zn1> For the two ZA quad-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.

For the four ZA quad-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

- <Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.
- <Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.
- <Zm1> For the two ZA quad-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 2.

For the four ZA quad-vectors variant: is the name of the firs `alable vector `gister of a multi-vector sequence, encoded as "Zm" times 4.

- <Zm4> Is the name of the fourth scalable vector register of multi-vector rue c, encoded as "Zm" times 4 plus 3.
- <Zm2> Is the name of the second scalable vector register o. _____iti-vecto__equence, encoded as "Zm" times 2 plus 1.

```
1
   CheckStreamingSVEAndZAEnabled();
2
   constant integer VL = CurrentVL;
   constant integer elements = V DIV
3
                                            ze;
4
   integer vectors = VL DIV 8;
   integer vstride = vectors
5
                                  1 nreq;
6
   bits(32) vbase = X[v, 3'
   integer vec = (UInt(vbase) + o. et) MOD vstride;
7
   bits(VL) result;
8
9
   vec = vec - (vec)
                       J 4);
10
   for r = 0 to g-1
11
       bits(VL) op and = Z[n+r. VL];
bits(VT) oper Z = Z[m+ VL];
12
13
        bits(VT ) oper
14
        for
                  to 3
            bits( ) oper
                            > = ZAvector[vec + i, VL];
15
16
            for < = 0 to e1 ments-1
17
                          lement1 = UInt(Elem[operand1, 4 * e + i, esize DIV 4]);
                integer element2 = SInt(Elem[operand2, 4 * e + i, esize DIV 4]);
18
19
                bits(esize) product = (element1 * element2)<esize-1:0>;
20
                Elem[result, e, esize] = Elem[operand3, e, esize] + product;
21
            ZAve or[vec + i, VL] = result;
22
        vec = vec + vstride;
```

D1.1.275 USMOPA

Unsigned by signed integer sum of outer products and accumulate

The 8-bit integer variant works with a 32-bit element ZA tile.

The 16-bit integer variant works with a 64-bit element ZA tile.

The unsigned by signed integer sum of outer products and accumulate instructions multiply the sub-matrix in the first source vector by the sub-matrix in the second source vector. In case of the 8-bit integer variant, the first source holds $SVL_S \times 4$ sub-matrix of unsigned 8-bit integer values, and the second source holds $4 \times SVL_S$ sub-matrix of signed 8-bit integer values. In case of the 16-bit integer variant, the first source holds $SVL_D \times 4$ sub-matrix of unsigned 16-bit integer values, and the second source holds $4 \times SVL_D \times 4$ sub-matrix of unsigned 16-bit integer values.

Each source vector is independently predicated by a corresponding governing predicate. When an 8-bit source element in case of 8-bit integer variant or a 16-bit source element in case 10-c. Pteger variant is Inactive, it is treated as having the value 0.

The resulting $SVL_S \times SVL_S$ widened 32-bit integer or $SVL_D \times SVL_D$ where d64-bit in ger sum of outer products is then destructively added to the 32-bit integer or 64-bit integer destination ile, resp. dively for 8-bit integer and 16-bit integer instruction variants. This is equivalent to perfer aing a 4-way do recalculate to each of the destination tile elements.

In case of the 8-bit integer variant, each 32-bit container of first sour vector holds 4 consecutive column elements of each row of a $SVL_S \times 4$ sub-matrix and the h 32-b. contrast of the second source vector holds 4 consecutive row elements of each column of a f SVL_S sub-matrix. The second source vector holds 4 consecutive row of the first source vector holds 4 consecutive row of a $SVL_D \times 4$ sub-matrix, and each 64-bit container of the second source vector holds 4 consecutive row elements of each column of a $4 \times SVL_S$ sub-matrix.

It has encodings from 2 classics, bit and 64-bit



```
USMOPA <ZAda>.D, <Pn>/M, <Pm>/M, <Zn>.H, <Zm>.H
1
   if !HaveSMEI16I64() then UNDEFINED;
2
   constant integer esize = 64;
  integer a = UInt(Pn);
3
4
   integer b = UInt(Pm);
5
   integer n = UInt(Zn);
  integer m = UInt(Zm);
6
   integer da = UInt(ZAda);
7
8
   boolean sub_op = FALSE;
   boolean op1_unsigned = TRUE;
9
10 boolean op2_unsigned = FALSE;
   Assembler Symbols
```

- <ZAda> For the 32-bit variant: is the name of the ZA tile ZA0-ZA3, encoded in the "ZAda" field.
 - For the 64-bit variant: is the name of the ZA tile ZA0-ZA7, enclued in the Ada" field.
 - <Pn> Is the name of the first governing scalable predicate register P0-. encoded in t "Pn" field.
 - <Pm> Is the name of the second governing scalable predicate _gister P0-, encod in the "Pm" field.
 - <Zn> Is the name of the first source scalable vector regis. enc .ed in the "Zn" field.
 - <Zm> Is the name of the second source scalable y registe 'ncode' in the "Zm" field.

Operation

```
1
   CheckStreamingSVEAndZAEnabled();
2
   constant integer VL = CurrentVL;
3
   constant integer PL = VL DIV
   constant integer dim = VL D
4
                                 esize;
5
   bits(PL) mask1 = P[a, PL]
   bits(PL) mask2 = P[b, P'
6
7
   bits(VL) operand1 = Z[n, VL];
8 bits(VL) operand2 = [m, VL];
   bits(dim*dim*esize operand3 = ZAt >[da, esize, dim*dim*esize];
9
10
   bits(dim*dim*es' .) resu't;
11
   integer prod
12
13
   for row = 0 to d_{\perp}
14
        for
                 0 to
                        m-1
15
            bits ( ize)
                              _lem[operand3, row*dim+col, esize];
                   = 0 to 3
16
            for }
17
                        wePredicateElement(mask1, 4*row + k, esize DIV 4) &&
18
                        ActivePredicateElement(mask2, 4*col + k, esize DIV 4) then
19
                    prod = (Int(Elem[operand1, 4*row + k, esize DIV 4], op1_unsigned) *
                            Int(Elem[operand2, 4*col + k, esize DIV 4], op2_unsigned));
20
21
                    if sub_op then prod = -prod;
22
                    sum = sum + prod;
23
24
            Elem[result, row*dim+col, esize] = sum;
25
26
   ZAtile[da, esize, dim*dim*esize] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
 - The values of the data supplied in any of its operand registers when its governing predicate registers contain the same value for each execution.
 - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:

- The values of the data supplied in any of its operand registers when its governing predicate registers contain the same value for each execution.
- The values of the NZCV flags.



D1.1.276 USMOPS

Unsigned by signed integer sum of outer products and subtract

The 8-bit integer variant works with a 32-bit element ZA tile.

The 16-bit integer variant works with a 64-bit element ZA tile.

The unsigned by signed integer sum of outer products and subtract instructions multiply the sub-matrix in the first source vector by the sub-matrix in the second source vector. In case of the 8-bit integer variant, the first source holds $SVL_S \times 4$ sub-matrix of unsigned 8-bit integer values, and the second source holds $4 \times SVL_S$ sub-matrix of signed 8-bit integer values. In case of the 16-bit integer variant, the first source holds $SVL_D \times 4$ sub-matrix of unsigned 16-bit integer values, and the second source holds $4 \times SVL_D \times 4$ sub-matrix of unsigned 16-bit integer values.

Each source vector is independently predicated by a corresponding governing predicate. When an 8-bit source element in case of 8-bit integer variant or a 16-bit source element in case 10-c. reger variant is Inactive, it is treated as having the value 0.

The resulting $SVL_S \times SVL_S$ widened 32-bit integer or $SVL_D \times SVL_D$ where d64-bit in ger sum of outer products is then destructively subtracted from the 32-bit integer or 64 at integer stination dile, respectively for 8-bit integer and 16-bit integer instruction variants. This is equivalent to perform, a vary dot product and subtract from each of the destination tile elements.

In case of the 8-bit integer variant, each 32-bit container of first sour vector holds 4 consecutive column elements of each row of a $SVL_S \times 4$ sub-matrix and the 32-bit container of the second source vector holds 4 consecutive row elements of each column of a $4 \times SVL_S$ sub-matrix. The second source vector holds 4 consecutive row of a $SVL_D \times 4$ sub-matrix, and each 64-bit container of the second source vector holds 4 consecutive row elements of each column of a $4 \times SVL_S$ sub-matrix.

It has encodings from 2 classics, bit and 64-bit



```
USMOPS <ZAda>.D, <Pn>/M, <Pm>/M, <Zn>.H, <Zm>.H
1
   if !HaveSMEI16I64() then UNDEFINED;
2
   constant integer esize = 64;
  integer a = UInt(Pn);
3
4
   integer b = UInt(Pm);
5
   integer n = UInt(Zn);
6 integer m = UInt(Zm);
   integer da = UInt(ZAda);
7
8
   boolean sub_op = TRUE;
   boolean op1_unsigned = TRUE;
9
10 boolean op2_unsigned = FALSE;
   Assembler Symbols
```

- <ZAda> For the 32-bit variant: is the name of the ZA tile ZA0-ZA3, encoded in the "ZAda" field.
 - For the 64-bit variant: is the name of the ZA tile ZA0-ZA7, enclued in the $\langle Pn \rangle$ Is the name of the first governing scalable predicate register P0-. encoded in t "Pn" field.

Ada" field.

- Is the name of the second governing scalable predicate _gister P0-1 encod _in the "Pm" <Pm>
- field.
- Is the name of the first source scalable vector regis. enc. ed in the "Zn" field. $\langle Zn \rangle$
- <Zm> Is the name of the second source scalable y registe 'ncoder' if the "Zm" field.

Operation

```
1
   CheckStreamingSVEAndZAEnabled();
2
   constant integer VL = CurrentVL;
3
   constant integer PL = VL DIV
   constant integer dim = VL D
4
                                 esize;
5
   bits(PL) mask1 = P[a, PL]
   bits(PL) mask2 = P[b, P'
6
   bits(VL) operand1 = Z[n, VL];
7
8 bits(VL) operand2 = [m, VL];
   bits(dim*dim*esize operand3 = ZAt >[da, esize, dim*dim*esize];
9
10
   bits(dim*dim*es' .) resu't;
11
   integer prod
12
13
   for row = 0 to d_{\perp}
14
        for
                 0 to
                        m-1
15
           bits ( ize)
                              _lem[operand3, row*dim+col, esize];
                   0 to 3
16
            for }
17
                       wePredicateElement(mask1, 4*row + k, esize DIV 4) &&
18
                        ActivePredicateElement(mask2, 4*col + k, esize DIV 4) then
19
                    prod = (Int(Elem[operand1, 4*row + k, esize DIV 4], op1_unsigned) *
                            Int(Elem[operand2, 4*col + k, esize DIV 4], op2_unsigned));
20
21
                    if sub_op then prod = -prod;
22
                    sum = sum + prod;
23
24
            Elem[result, row*dim+col, esize] = sum;
25
26
   ZAtile[da, esize, dim*dim*esize] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
 - The values of the data supplied in any of its operand registers when its governing predicate registers contain the same value for each execution.
 - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:

- The values of the data supplied in any of its operand registers when its governing predicate registers contain the same value for each execution.
- The values of the NZCV flags.



D1.1.277 USVDOT

Multi-vector unsigned by signed integer vertical dot-product by indexed element

The instruction operates on four ZA single-vector groups.

The unsigned by signed integer vertical dot product instruction computes the vertical dot product of corresponding unsigned 8-bit elements from the four first source vectors and four signed 8-bit integer values in the corresponding indexed 32-bit element of the second source vector. The widened dot product result is destructively added to the corresponding 32-bit element of four ZA single-vector groups.

The groups within the second source vector are specified using an immediate element index which selects the same group position within each 128-bit vector segment. The index range is from 0 to 3, encoded in 2 bits.

The vector numbers forming the single-vector group within each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo quarter the number of Δt_{c} , ray vectors.

The VECTOR GROUP symbol VGx4 indicates that the ZA operand constst of four Z₂ single-vector groups. The VECTOR GROUP symbol is preferred for disassembly, but optional in a mbler source code.

This instruction is unpredicated.

SME2 (FEAT_SME2)

0 0 0 0 0 1 1 0 1 0 **7π** Rv 0 1 0 1 off3

USVDOT ZA.S[<Wv>, <offs>{, Gx4}], <Zn1 B-<Zn4>.B }, <Zm>.B[<index>]

```
1 if !HaveSME2() then UNDF M
2 integer v = UInt('010'. v);
3 constant integer esi 2 = 32;
4 integer n = UInt(7 00');
5 integer m = UInt J':Zm);
6 integer offset UInt(c 3);
```

```
7 integer index "Int( );
```

Assemb' bols

```
<Wv> Is the 3' bit name of the vector select register W8-W11, encoded in the "Rv" field.
```

- $\langle offs \rangle$ t⁺ vector second offset, in the range 0 to 7, encoded in the "off3" field.
- <Zn1> Is us name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times
- <Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.
- <Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

<index> Is the element index, in the range 0 to 3, encoded in the "i2" field.

```
CheckStreamingSVEAndZAEnabled();
1
  constant integer VL = CurrentVL;
2
3
  constant integer elements = VL DIV esize;
4
  integer vectors = VL DIV 8;
  integer vstride = vectors DIV 4;
5
6
  integer eltspersegment = 128 DIV esize;
7
  bits(32) vbase = X[v, 32];
  integer vec = (UInt(vbase) + offset) MOD vstride;
8
```

```
Chapter D1. SME instructions
D1.1. SME and SME2 data-processing instructions
```

```
9 bits(VL) operand2 = Z[m, VL];
10 bits(VL) result;
11
12
    for r = 0 to 3
13
        bits(VL) operand3 = ZAvector[vec, VL];
14
         for e = 0 to elements-1
15
            integer segmentbase = e - (e MOD eltspersegment);
             integer s = segmentbase + index;
16
17
             bits(esize) sum = Elem[operand3, e, esize];
             for i = 0 to 3
18
19
                 bits(VL) operand1 = Z[n+i, VL];
                 integer element1 = UInt(Elem[operand1, 4 * e + r, esize DIV 4]);
integer element2 = SInt(Elem[operand2, 4 * s + i, esize DIV 4]);
20
21
22
                 sum = sum + element1 * element2;
23
             Elem[result, e, esize] = sum;
         ZAvector[vec, VL] = result;
24
25
        vec = vec + vstride;
```

D1.1.278 UUNPK

Unpack and zero-extend multi-vector elements

Unpack elements from one or two source vectors and then zero-extend them to place in elements of twice their size within the two or four destination vectors.

This instruction is unpredicated.

It has encodings from 2 classes: Two registers and Four registers

Two registers (FEAT_SME2)



Assembler Symbols

<Zd1> For the two registers variant: is the name of the first destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 2.

For the four registers variant: is the name of the first destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 4.

<T> Is the size specifier, encoded in "size":

size	<t></t>
00	RESERVED
01	Н
10	S
11	D

- <Zd4> Is the name of the fourth destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 4 plus 3.
- <Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.
- <Zd2> Is the name of the second destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 2 plus 1.
- <Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.
- <Tb> Is the size specifier, encoded in "size":

size	< Tb >
00	RESERVED
01	В
10	Н
11	S

<Zn2> Is the name of the second scalable vector register of *r*_nulti-vector requence encoded as "Zn" times 2 plus 1.

```
CheckStreamingSVEEnabled();
1
2
   constant integer VL = CurrentVL;
3
   constant integer elements = VL DIV es
                                            .e;
4
   constant integer hsize = esize DIV 2;
5
   constant integer sreg = nreg DIV
   array [0..3] of bits(VL) results;
6
7
8
   for r = 0 to sreg-1
9
       bits(VL) operand = Z
                                   VL];
10
       for i = 0 to 1
11
            for e = 0 to elements-
               bits(hs e) element
12
                                        'lem[operand, i*elements + e, hsize];
13
                Elem[ sults[2*r+i], e, size] = Extend(element, esize, unsigned);
14
   for r = 0 to :
15
                    ∙g−1
16
       Z[d+r, VL]
                      res
                          cs[r];
```

D1.1.279 UVDOT (2-way)

Multi-vector unsigned integer vertical dot-product by indexed element

The instruction operates on two ZA single-vector groups.

The unsigned integer vertical dot product instruction computes the vertical dot product of the corresponding two unsigned 16-bit integer values held in the two first source vectors and two unsigned 16-bit integer values in the corresponding indexed 32-bit element of the second source vector. The widened dot product results are destructively added to the corresponding 32-bit element of two ZA single-vector groups.

The groups within the second source vector are specified using an immediate element index which selects the same group position within each 128-bit vector segment. The index range is from 0 to 3, encoded in 2 bits.

The vector numbers forming the single-vector group within each half of the 7A array are selected by the sum of the vector select register and immediate offset, modulo half the number c = A are vectors.

The VECTOR GROUP symbol VGx2 indicates that the ZA operand c sists of two Z₄ single-vector groups. The VECTOR GROUP symbol is preferred for disassembly, but optional in a mbler source code.

This instruction is unpredicated.

SME2 (FEAT_SME2)

0 0 0 0 0 1 1 0 1 0 1 **7π** Rv 7n 1 1 0 off3

UVDOT ZA.S[<Wv>, <offs>{, Gx2}], <Zn1 H-<Zn2>.H }, <Zm>.H[<index>]

```
1 if !HaveSME2() then UNDF M
2 integer v = UInt('010'. v);
3 constant integer esi c = 32;
4 integer n = UInt(7 0');
5 integer m = UInt o':Zm);
6 integer offset UInt(c 3);
```

```
7 integer index "Int( );
```

Assemb' bols

- <Wv> Is the 3' bit name of the vector select register W8-W11, encoded in the "Rv" field.
- $\langle offs \rangle$ t⁺ vector second offset, in the range 0 to 7, encoded in the "off3" field.
- <Zn1> Is u name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times.
- <Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.
- <Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

<index> Is the element index, in the range 0 to 3, encoded in the "i2" field.

```
CheckStreamingSVEAndZAEnabled();
1
  constant integer VL = CurrentVL;
2
3
  constant integer elements = VL DIV esize;
4
  integer vectors = VL DIV 8;
  integer vstride = vectors DIV 2;
5
6
  integer eltspersegment = 128 DIV esize;
7
  bits(32) vbase = X[v, 32];
  integer vec = (UInt(vbase) + offset) MOD vstride;
8
```

```
Chapter D1. SME instructions
D1.1. SME and SME2 data-processing instructions
```

```
9 bits(VL) operand2 = Z[m, VL];
10 bits(VL) result;
11
12
    for r = 0 to 1
13
        bits(VL) operand3 = ZAvector[vec, VL];
14
         for e = 0 to elements-1
15
            integer segmentbase = e - (e MOD eltspersegment);
             integer s = segmentbase + index;
16
17
             bits(esize) sum = Elem[operand3, e, esize];
18
             for i = 0 to 1
19
                 bits(VL) operand1 = Z[n+i, VL];
                 integer element1 = UInt(Elem[operand1, 2 * e + r, esize DIV 2]);
integer element2 = UInt(Elem[operand2, 2 * s + i, esize DIV 2]);
20
21
22
                 sum = sum + element1 * element2;
23
             Elem[result, e, esize] = sum;
         ZAvector[vec, VL] = result;
24
25
        vec = vec + vstride;
```

D1.1.280 UVDOT (4-way)

Multi-vector unsigned integer vertical dot-product by indexed element

The instruction operates on four ZA single-vector groups.

The unsigned integer vertical dot product instruction computes the vertical dot product of the corresponding four unsigned 8-bit or 16-bit integer values held in the four first source vectors and four unsigned 8-bit or 16-bit integer values in the corresponding indexed 32-bit or 64-bit element of the second source vector. The widened dot product results are destructively added to the corresponding 32-bit or 64-bit element of the four ZA single-vector groups.

The groups within the second source vector are specified using an immediate element index which selects the same group position within each 128-bit vector segment. The index range is from 0 to one less than the number of groups per 128-bit segment, encoded in 1 to 2 bits depending on the size of the group.

The vector numbers forming the single-vector group within each quarter $o^{c} = z_{c}$, and are selected by the sum of the vector select register and immediate offset, modulo quarter the nur ser of ZA a. vectors.

The VECTOR GROUP symbol VGx4 indicates that the ZA operand con. 's of four ZA ingle-vector groups. The VECTOR GROUP symbol is preferred for disassembly, but optir ... in asse. 1er sour code.

This instruction is unpredicated.

ID_AA64SMFR0_EL1.I16I64 indicates whether the 16-but 'er', variant i implemented.

It has encodings from 2 classes: 32-bit and 64-bi

32-bit (FEAT SME2)

1 2 3

5

6

7

```
13
                                                                        11
                  0
                     0
                        0
                           0
                              0
                                                            1
                                                                Rv
                                                                                 7n
                                                                                          1
                                                                                            1
                                                                                                    off3
                                 1
                                    0
                                          ß
                                                    7m
                                                                     0
                                                                         i2
                                                                                       Θ
                                                                                               0
                                                                                             Lu
                            offs>{, VGx.
   UVDOT ZA.S[<Wv>
                                                { <Zn1>.B-<Zn4>.B }, <Zm>.B[<index>]
  if !HaveSME2() nen UND INED;
integer v = UI. ''010 Rv);
   constant integer
                            e = 32;
                  Tnt(Z. '00');
4
   integer
   intege in = t it('0'
   inte r offse = UInt(
                                 .3);
   int ar ind
                       ™nt (i2):
   64-bit
   (FEAT_SM.
                  U16I64)
                  0
                     0
                        0
                           0
                              0
                                 1
                                    1
                                       1
                                          0
                                                    Zm
                                                                Rv
                                                                                                    off3
                                             1
                                                                     0
                                                                           i1
                                                                                             1
                                                            1
                                                                        1
```

```
UVDOT ZA.D[<Wv>, <offs>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H[<index>]
```

```
1
  if !(HaveSME2() && HaveSMEI16I64()) then UNDEFINED;
```

```
2
  integer v = UInt('010':Rv);
```

```
3
  constant integer esize = 64;
```

```
4
  integer n = UInt(Zn:'00');
5
  integer m = UInt('0':Zm);
```

```
6
```

```
integer offset = UInt(off3);
  integer index = UInt(i1);
7
```

Assembler Symbols

- <Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.
- <offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.
- <Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.
- <Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.
- <Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

<index> For the 32-bit variant: is the element index, in the range 0 to 3, encoded in the "i2" field.

For the 64-bit variant: is the element index, in the range 0 to 1, encoded in the "i1" field.

```
1
   CheckStreamingSVEAndZAEnabled();
   constant integer VL = CurrentVL;
2
3
   constant integer elements = VL DIV esize;
4
   integer vectors = VL DIV 8;
   integer vstride = vectors DIV 4;
5
6
   integer eltspersegment = 128 DIV esize;
 7
   bits(32) vbase = X[v, 32];
 8
   integer vec = (UInt(vbase) + offset) MOD
                                                stri
9
   bits(VL) operand2 = Z[m, VL];
10
   bits(VL) result;
11
12
    for r = 0 to 3
13
        bits(VL) operand3 = ZAvector[v
                                            VL];
14
        for e = 0 to elements-1
                                               eltsp rsegment);
15
            integer segmentbas
                                   e - (e M
16
                                 se + index;
            integer s = segmr
                               Ele operand3,
17
            bits(esize) sur
                                               e. esize];
            for i = 0 to 3
18
                bits(VI operand1 = 2 +i, VL];
inter . element1 = UInt lem[operand1, 4 * e + r, esize DIV 4]);
19
20
21
                int ger eler it2 = UInt(Elem[operand2, 4 * s + i, esize DIV 4]);
22
                su = sum element1 * element2;
23
            Elem[res
                           ., esize' = sum;
                           = res
24
        ZAvec
                 vec.
25
        vec
                    vstı 'ə;
              VA.
```

D1.1.281 UZP (four registers)

Concatenate elements from four vectors

Concatenate every fourth element from each of the four source vectors and place them in the corresponding elements of the four destination vectors.

This instruction is unpredicated.

It has encodings from 2 classes: 8-bit to 64-bit elements and 128-bit element

8-bit to 64-bit elements (FEAT_SME2)



- <Zd4> Is the name of the fourth destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 4 plus 3.
- $\label{eq:2n1} <\!\! \text{Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn"} \\ times 4.$
- <Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.

```
1 CheckStreamingSVEEnabled();
 2 constant integer VL = CurrentVL;
 3
    if VL < esize * 4 then UNDEFINED;</pre>
    constant integer quads = VL DIV (esize * 4);
 4
 5 bits(VL) result0;
 6 bits(VL) result1;
 7 bits(VL) result2;
 8 bits(VL) result3;
 9
10
    for r = 0 to 3
11
         bits(VL) operand = Z[n+r, VL];
12
          integer base = r * quads;
13
          for q = 0 to quads-1
14
               Elem[result0, base+q, esize] = Elem[operand, 4*q+0, esize];
              Elem[result1, base+q, esize] = Elem[operand, 4*q+1, esize];
Elem[result2, base+q, esize] = Elem[operand, 4*q+2, esize];
Elem[result3, base+q, esize] = Elem[operand, 4*q+3, esize]
15
16
17
18
19
    Z[d+0, VL] = result0;
20 Z[d+1, VL] = result1;
21 Z[d+2, VL] = result2;
22 Z[d+3, VL] = result3;
```

D1.1.282 UZP (two registers)

Concatenate elements from two vectors

Concatenate every second element from each of the first and second source vectors and place them in the corresponding elements of the two destination vectors.

This instruction is unpredicated.

It has encodings from 2 classes: 8-bit to 64-bit elements and 128-bit element

8-bit to 64-bit elements (FEAT_SME2)



Is the size specifier, encoded in "size":

<t></t>
В
Η
S
D

- <Zd2> Is the name of the second destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 2 plus 1.
- $\langle Zn \rangle$ Is the name of the first source scalable vector register, encoded in the "Zn" field.
- Is the name of the second source scalable vector register, encoded in the "Zm" field. <Zm>

```
1 CheckStreamingSVEEnabled();
2 constant integer VL = CurrentVL;
    if VL < esize * 2 then UNDEFINED;
constant integer pairs = VL DIV (esize * 2);
 3
4
 5 bits(VL) result0;
6 bits(VL) result1;
7
8
     for r = 0 to 1
          integer base = r * pairs;
bits(VL) operand = if r == 0 then Z[n, VL] else Z[m, VL];
9
10
11
          for p = 0 to pairs-1
                Elem[result0, base+p, esize] = Elem[operand, 2*p+0, esize];
Elem[result1, base+p, esize] = Elem[operand, 2*p+1, esize];
12
13
14
15 Z[d+0, VL] = result0;
16 Z[d+1, VL] = result1;
```

D1.1.283 WHILEGE

While decrementing signed scalar greater than or equal to scalar (predicate-as-counter)

Generate a predicate for a group of two or four vectors that starting from the highest numbered element of the group is true while the decrementing value of the first, signed scalar operand is greater than or equal to the second scalar operand and false thereafter down to the lowest numbered element of the group.

If the second scalar operand is equal to the minimum signed integer value then a condition which includes an equality test can never fail and the result will be an all-true predicate.

The full width of the scalar operands is significant for the purposes of comparison, and the full width first operand is decremented by one for each destination predicate element, irrespective of the predicate result element size.

The predicate result is placed in the predicate destination register using the predicate-as-counter encoding. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate suit, a the V flag to zero.

SME2 (FEAT_SME2)



WHILEGE <PNd>.<T>, <Xn>, <Xm>, <v]

- 1 if !HaveSME2() then UNDEFINED;
- 2 constant integer esize = 8 << UIL 'size)</pre>
- 3 constant **integer** rsize = 64;
- 4 integer n = UInt(Rn);
- 5 integer m = UInt(Rm);
- 6 integer d = UInt('1':PNd'
- 7 **boolean** unsigned = FAL
- 8 **boolean** invert = TRU
- 9 SVECmp op = Cmp_GF
- 10 integer width = < < UInt(vl);</pre>

Assembler Symb.

<PNd> Is ... be of the desting on scalable predicate register PN8-PN15, with predicate-as-counter incoding encoded are "PNd" field.

<t> Is the</t>		"for, encoded in "size":
_ ب	<t></t>	
_0 _	В	-
01	Н	
10	S	
11	D	
		-

- <Xn> Is the 64-bit name of the first source general-purpose register, encoded in the "Rn" field.
- <Xm> Is the 64-bit name of the second source general-purpose register, encoded in the "Rm" field.
- <vl> Is the vl specifier, encoded in "vl":

vl	<vl></vl>
0	VLx2
1	VLx4

```
1 CheckStreamingSVEEnabled();
2 constant integer VL = CurrentVL;
3
   constant integer PL = VL DIV 8;
4
   constant integer elements = width * (VL DIV esize);
5 bits(rsize) operand1 = X[n, rsize];
6
   bits(rsize) operand2 = X[m, rsize];
7
   bits(PL) result;
8
   boolean last = TRUE;
9
   integer count = 0;
10
11
   for e = elements-1 downto 0
12
       boolean cond;
13
       case op of
14
           when Cmp_GT cond = (Int(operand1, unsigned) > Int(operand2, unsigned));
15
           when Cmp_GE cond = (Int(operand1, unsigned) >= Int(operand2, unsigned));
16
17
       last = last && cond;
18
       if last then count = count + 1;
19
       operand1 = operand1 - 1;
20
21
   result = EncodePredCount(esize, elements, count, inve, PL)
22
   PSTATE.<N,Z,C,V> = PredCountTest(elements, count, ir .rt);
23
   P[d, PL] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
 - The values of the data supplied in an f registers.
 - The values of the NZCV rags.
- The response of this instr ion to asynci nous exceptions does not vary based on:
 - The values of the .ata su, lied in any on its registers.
 - The values cone NZCV flag.

D1.1.284 WHILEGT

While decrementing signed scalar greater than scalar (predicate-as-counter)

Generate a predicate for a group of two or four vectors that starting from the highest numbered element of the group is true while the decrementing value of the first, signed scalar operand is greater than the second scalar operand and false thereafter down to the lowest numbered element of the group.

The full width of the scalar operands is significant for the purposes of comparison, and the full width first operand is decremented by one for each destination predicate element, irrespective of the predicate result element size.

The predicate result is placed in the predicate destination register using the predicate-as-counter encoding. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

0

U_

Rr

Rm

SME2 (FEAT_SME2)

```
WHILEGT <PNd>.<T>, <Xn>, <Xm>, <vl>
```

```
if !HaveSME2() then UNDEFINED;
1
2
  constant integer esize = 8 << UInt(siz
3
  constant integer rsize = 64;
```

```
4
  integer n = UInt(Rn);
5
  integer m = UInt(Rm);
```

```
integer d = UInt('1':PNd);
6
```

0 1 0 0 1 0 1 size

7 boolean unsigned = FALSE;

```
boolean invert = TRUE;
8
```

```
9
   SVECmp op = Cmp_GT;
```

```
10
   integer width = 2 << U'
                              (vl
```

Assembler Symbols

Is the nary of the deviation scalable predicate register PN8-PN15, with predicate-as-counter <PNd> encoding, $\operatorname{od}_{\mathcal{E}}$ in the "PNd" field.



- Is the 64-bit name of the first source general-purpose register, encoded in the "Rn" field. <Xn>
- < Xm >Is the 64-bit name of the second source general-purpose register, encoded in the "Rm" field.
 - Is the vl specifier, encoded in "vl": $\langle vl \rangle$

	-
vl	<vl></vl>
0	VLx2
1	VLx4

- 1 CheckStreamingSVEEnabled();
- constant integer VL = CurrentVL; constant integer PL = VL DIV 8; 2
- 3

```
4
   constant integer elements = width * (VL DIV esize);
5
   bits(rsize) operand1 = X[n, rsize];
   bits(rsize) operand2 = X[m, rsize];
 6
   bits(PL) result;
7
8
   boolean last = TRUE;
9
   integer count = 0;
10
11
   for e = elements-1 downto 0
12
        boolean cond;
        case op of
13
14
            when Cmp_GT cond = (Int(operand1, unsigned) > Int(operand2, unsigned));
15
            when Cmp_GE cond = (Int(operand1, unsigned) >= Int(operand2, unsigned));
16
17
        last = last && cond;
        if last then count = count + 1;
operand1 = operand1 - 1;
18
19
20
21
   result = EncodePredCount(esize, elements, count, invert, PL)
22
   PSTATE.<N,Z,C,V> = PredCountTest(elements, count, invert)
23
   P[d, PL] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent or
 - The values of the data supplied in any of its resters.
 - The values of the NZCV flags.
- The response of this instruction to vnchro as exceptions does not vary based on:
 - The values of the data s _plied in _v of in. _gisters.
 - The values of the N \sim flags.
D1.1.285 WHILEHI

While decrementing unsigned scalar higher than scalar (predicate-as-counter)

Generate a predicate for a group of two or four vectors that starting from the highest numbered element of the group is true while the decrementing value of the first, unsigned scalar operand is higher than the second scalar operand and false thereafter down to the lowest numbered element of the group.

The full width of the scalar operands is significant for the purposes of comparison, and the full width first operand is decremented by one for each destination predicate element, irrespective of the predicate result element size.

The predicate result is placed in the predicate destination register using the predicate-as-counter encoding. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

0

Rr

Rm

SME2 (FEAT_SME2)

WHILEHI <PNd>.<T>, <Xn>, <Xm>, <vl>

```
if !HaveSME2() then UNDEFINED;
1
2
  constant integer esize = 8 << UInt(siz
```

- 3 constant integer rsize = 64;
- 4 integer n = UInt(Rn);
- 5 integer m = UInt(Rm);

0 1 0 0 1 0 1 size

- integer d = UInt('1':PNd); 6
- 7 boolean unsigned = TRUE; boolean invert = TRUE; 8
- 9 SVECmp op = Cmp_GT;
- 10 integer width = 2 << U7 (vl

Assembler Symbols

Is the nary of the deviation scalable predicate register PN8-PN15, with predicate-as-counter <PNd> encoding, $\operatorname{od}_{\mathcal{E}}$ in the "PNd" field.



- Is the 64-bit name of the first source general-purpose register, encoded in the "Rn" field. <Xn>
- < Xm >Is the 64-bit name of the second source general-purpose register, encoded in the "Rm" field.
 - Is the vl specifier, encoded in "vl": $\langle vl \rangle$

vl	<vl></vl>
0	VLx2
1	VLx4

- 1 CheckStreamingSVEEnabled();
- constant integer VL = CurrentVL; constant integer PL = VL DIV 8; 2
- 3

```
4
   constant integer elements = width * (VL DIV esize);
5
   bits(rsize) operand1 = X[n, rsize];
   bits(rsize) operand2 = X[m, rsize];
 6
   bits(PL) result;
7
8
   boolean last = TRUE;
9
   integer count = 0;
10
11
   for e = elements-1 downto 0
12
        boolean cond;
        case op of
13
14
            when Cmp_GT cond = (Int(operand1, unsigned) > Int(operand2, unsigned));
15
            when Cmp_GE cond = (Int(operand1, unsigned) >= Int(operand2, unsigned));
16
17
        last = last && cond;
        if last then count = count + 1;
operand1 = operand1 - 1;
18
19
20
21
   result = EncodePredCount(esize, elements, count, invert, PL)
22
   PSTATE.<N,Z,C,V> = PredCountTest(elements, count, invert)
23
   P[d, PL] = result;
```

Operational information

- The execution time of this instruction is independent or
 - The values of the data supplied in any of its resters.
 - The values of the NZCV flags.
- The response of this instruction to vnchro as exceptions does not vary based on:
 - The values of the data s _plied in _v of in. _gisters.
 - The values of the N \sim flags.

D1.1.286 WHILEHS

While decrementing unsigned scalar higher or same as scalar (predicate-as-counter)

Generate a predicate for a group of two or four vectors that starting from the highest numbered element of the group is true while the decrementing value of the first, unsigned scalar operand is higher or same as the second scalar operand and false thereafter down to the lowest numbered element of the group.

If the second scalar operand is equal to the minimum unsigned integer value then a condition which includes an equality test can never fail and the result will be an all-true predicate.

The full width of the scalar operands is significant for the purposes of comparison, and the full width first operand is decremented by one for each destination predicate element, irrespective of the predicate result element size.

The predicate result is placed in the predicate destination register using the predicate-as-counter encoding. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate __uu, ... the V flag to zero.

SME2 (FEAT_SME2)



WHILEHS <PNd>.<T>, <Xn>, < Xm >. < v

- 1 if !HaveSME2() then UNDEFINED;
- constant integer esize = 8 << UIA 2 (size)
- 3 constant integer rsize = 64; 4
- integer n = UInt(Rn);
- integer m = UInt(Rm); 5
- integer d = UInt('1':PNd' 6 **boolean** unsigned = TRUI 7
- 8
- **boolean** invert = TRU 9 SVECmp op = Cmp_GF
- 10 integer width = < UInt(vl);

Assembler Symb.

<PNd> Is ... be of the destine on scalable predicate register PN8-PN15, with predicate-as-counter

<t> Is the</t>		Gor, encoded in "size":
ى 🔻	<t></t>	
	В	-
01	Н	
10	S	
11	D	
		-

- Is the 64-bit name of the first source general-purpose register, encoded in the "Rn" field. <Xn>
- < Xm >Is the 64-bit name of the second source general-purpose register, encoded in the "Rm" field.
- Is the vl specifier, encoded in "vl": $\langle vl \rangle$

vl	<vl></vl>
0	VLx2
1	VLx4

```
1 CheckStreamingSVEEnabled();
2 constant integer VL = CurrentVL;
3
   constant integer PL = VL DIV 8;
4
   constant integer elements = width * (VL DIV esize);
5 bits(rsize) operand1 = X[n, rsize];
6
   bits(rsize) operand2 = X[m, rsize];
7
   bits(PL) result;
8
   boolean last = TRUE;
9
   integer count = 0;
10
11
   for e = elements-1 downto 0
12
       boolean cond;
13
       case op of
14
           when Cmp_GT cond = (Int(operand1, unsigned) > Int(operand2, unsigned));
15
           when Cmp_GE cond = (Int(operand1, unsigned) >= Int(operand2, unsigned));
16
17
       last = last && cond;
18
       if last then count = count + 1;
19
       operand1 = operand1 - 1;
20
21
   result = EncodePredCount(esize, elements, count, inver, PL)
22
   PSTATE.<N,Z,C,V> = PredCountTest(elements, count, ir .rt);
23
   P[d, PL] = result;
```

- The execution time of this instruction is independent of:
 - The values of the data supplied in an f registers.
 - The values of the NZCV rags.
- The response of this instr ion to asynci nous exceptions does not vary based on:
 - The values of the .ata su, lied in any on its registers.
 - The values cone NZCV flag.

D1.1.287 WHILELE

While incrementing signed scalar less than or equal to scalar (predicate-as-counter)

Generate a predicate for a group of two or four vectors that starting from the lowest numbered element of the group is true while the incrementing value of the first, signed scalar operand is less than or equal to the second scalar operand and false thereafter up to the highest numbered element of the group.

If the second scalar operand is equal to the maximum signed integer value then a condition which includes an equality test can never fail and the result will be an all-true predicate.

The full width of the scalar operands is significant for the purposes of comparison, and the full width first operand is incremented by one for each destination predicate element, irrespective of the predicate result element size.

The predicate result is placed in the predicate destination register using the predicate-as-counter encoding. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate suit, a the V flag to zero.

SME2 (FEAT_SME2)



WHILELE <PNd>.<T>, <Xn>, <Xm>, <v]

```
1 if !HaveSME2() then UNDEFINED;
```

- 2 constant integer esize = 8 << UIL 'size)</pre>
- 3 constant **integer** rsize = 64;
- 4 integer n = UInt(Rn);
- 5 integer m = UInt(Rm);
- 6 integer d = UInt('1':PNd'
- 7 **boolean** unsigned = FAL
- 8 boolean invert = FAI E;
- 9 SVECmp op = Cmp_LF
- 10 integer width = / << UInt(vl);</pre>

Assembler Symb.

<PNd> Is ... be of the desting on scalable predicate register PN8-PN15, with predicate-as-counter incoding encoded are "PNd" field.

<T $>$ Is the		for, encoded in "size":
ى 🔻	<t></t>	
	В	-
01	Н	
10	S	
11	D	
		-

- <Xn> Is the 64-bit name of the first source general-purpose register, encoded in the "Rn" field.
- <Xm> Is the 64-bit name of the second source general-purpose register, encoded in the "Rm" field.
- <vl> Is the vl specifier, encoded in "vl":

vl	<vl></vl>
0	VLx2
1	VLx4

```
1 CheckStreamingSVEEnabled();
2 constant integer VL = CurrentVL;
3
   constant integer PL = VL DIV 8;
4
   constant integer elements = width * (VL DIV esize);
5 bits(rsize) operand1 = X[n, rsize];
6
   bits(rsize) operand2 = X[m, rsize];
7
   bits(PL) result;
8
   boolean last = TRUE;
9
   integer count = 0;
10
11
   for e = 0 to elements-1
12
       boolean cond;
13
       case op of
14
           when Cmp_LT cond = (Int(operand1, unsigned) < Int(operand2, unsigned));</pre>
15
           when Cmp_LE cond = (Int(operand1, unsigned) <= Int(operand2, unsigned));</pre>
16
17
       last = last && cond;
18
       if last then count = count + 1;
19
       operand1 = operand1 + 1;
20
21
   result = EncodePredCount(esize, elements, count, inver, PL)
22
   PSTATE.<N,Z,C,V> = PredCountTest(elements, count, ir .rt);
23
   P[d, PL] = result;
```

- The execution time of this instruction is independent of:
 - The values of the data supplied in an f registers.
 - The values of the NZCV rags.
- The response of this instr ion to asynci nous exceptions does not vary based on:
 - The values of the .ata su, lied in any on its registers.
 - The values cone NZCV flag.

D1.1.288 WHILELO

While incrementing unsigned scalar lower than scalar (predicate-as-counter)

Generate a predicate for a group of two or four vectors that starting from the lowest numbered element of the group is true while the incrementing value of the first, unsigned scalar operand is lower than the second scalar operand and false thereafter up to the highest numbered element of the group.

The full width of the scalar operands is significant for the purposes of comparison, and the full width first operand is incremented by one for each destination predicate element, irrespective of the predicate result element size.

The predicate result is placed in the predicate destination register using the predicate-as-counter encoding. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

0

Rr

0

SME2 (FEAT_SME2)

U_

Rm

```
WHILELO <PNd>.<T>, <Xn>, <Xm>, <vl>
```

```
if !HaveSME2() then UNDEFINED;
1
2
  constant integer esize = 8 << UInt(siz
```

3 constant integer rsize = 64;

0 1 0 0 1 0 1 size

- 4 integer n = UInt(Rn); 5 integer m = UInt(Rm);
- integer d = UInt('1':PNd); 6
- 7 boolean unsigned = TRUE;
- boolean invert = FALSE; 8
- 9 SVECmp op = Cmp_LT;
- 10 integer width = 2 << U' (v1)

Assembler Symbols

Is the nary of the deviation scalable predicate register PN8-PN15, with predicate-as-counter <PNd> encoding, $\operatorname{od}_{\mathcal{E}}$ in the "PNd" field.



- Is the 64-bit name of the first source general-purpose register, encoded in the "Rn" field. <Xn>
- < Xm >Is the 64-bit name of the second source general-purpose register, encoded in the "Rm" field.
 - Is the vl specifier, encoded in "vl": $\langle vl \rangle$

	\1/
0	VLx2
1	VLx4

- 1 CheckStreamingSVEEnabled();
- constant integer VL = CurrentVL; constant integer PL = VL DIV 8; 2
- 3

```
4
   constant integer elements = width * (VL DIV esize);
5
   bits(rsize) operand1 = X[n, rsize];
   bits(rsize) operand2 = X[m, rsize];
6
   bits(PL) result;
7
8
   boolean last = TRUE;
9
   integer count = 0;
10
11
   for e = 0 to elements-1
12
       boolean cond;
        case op of
13
14
            when Cmp_LT cond = (Int(operand1, unsigned) < Int(operand2, unsigned));</pre>
15
            when Cmp_LE cond = (Int(operand1, unsigned) <= Int(operand2, unsigned));</pre>
16
17
        last = last && cond;
18
        if last then count = count + 1;
19
        operand1 = operand1 + 1;
20
21
   result = EncodePredCount(esize, elements, count, invert, PL)
22
   PSTATE.<N,Z,C,V> = PredCountTest(elements, count, invert)
23
   P[d, PL] = result;
```

- The execution time of this instruction is independent or
 - The values of the data supplied in any of its resters.
 - The values of the NZCV flags.
- The response of this instruction to vnchro as exceptions does not vary based on:
 - The values of the data s _plied in _v of in. _gisters.
 - The values of the N \sim flags.

D1.1.289 WHILELS

While incrementing unsigned scalar lower or same as scalar (predicate-as-counter)

Generate a predicate for a group of two or four vectors that starting from the lowest numbered element of the group is true while the incrementing value of the first, unsigned scalar operand is lower or same as the second scalar operand and false thereafter up to the highest numbered element of the group.

If the second scalar operand is equal to the maximum unsigned integer value then a condition which includes an equality test can never fail and the result will be an all-true predicate.

The full width of the scalar operands is significant for the purposes of comparison, and the full width first operand is incremented by one for each destination predicate element, irrespective of the predicate result element size.

The predicate result is placed in the predicate destination register using the predicate-as-counter encoding. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate __uu, ... the V flag to zero.

SME2 (FEAT_SME2)



WHILELS <PNd>.<T>, <Xn>, < Xm >. < v

```
1
  if !HaveSME2() then UNDEFINED;
```

- constant integer esize = 8 << UI 2 (size)
- 3 constant integer rsize = 64;
- 4 integer n = UInt(Rn);
- integer m = UInt(Rm); 5
- integer d = UInt('1':PNd' 6
- **boolean** unsigned = TRUI 7 8
- boolean invert = FAI E; 9
- SVECmp op = Cmp_LF 10
- integer width = < UInt(vl);

Assembler Symb.

<PNd> Is ... be of the destine on scalable predicate register PN8-PN15, with predicate-as-counter ncodin encode. .e "PNd" field.

<t> Is the</t>	ne	Gor, encoded in "size":
	∠ <t></t>	,
	В	
0	1 Н	
1	0 S	
1	1 D	

- Is the 64-bit name of the first source general-purpose register, encoded in the "Rn" field. <Xn>
- < Xm >Is the 64-bit name of the second source general-purpose register, encoded in the "Rm" field.
- Is the vl specifier, encoded in "vl": $\langle vl \rangle$

vl	<vl></vl>
0	VLx2
1	VLx4

```
1 CheckStreamingSVEEnabled();
2 constant integer VL = CurrentVL;
3
   constant integer PL = VL DIV 8;
4
   constant integer elements = width * (VL DIV esize);
5 bits(rsize) operand1 = X[n, rsize];
6
   bits(rsize) operand2 = X[m, rsize];
7
   bits(PL) result;
8
   boolean last = TRUE;
9
   integer count = 0;
10
11
   for e = 0 to elements-1
12
       boolean cond;
13
       case op of
14
           when Cmp_LT cond = (Int(operand1, unsigned) < Int(operand2, unsigned));</pre>
15
           when Cmp_LE cond = (Int(operand1, unsigned) <= Int(operand2, unsigned));</pre>
16
17
       last = last && cond;
18
       if last then count = count + 1;
19
       operand1 = operand1 + 1;
20
21
   result = EncodePredCount(esize, elements, count, inver, PL)
22
   PSTATE.<N,Z,C,V> = PredCountTest(elements, count, ir .rt);
23
   P[d, PL] = result;
```

- The execution time of this instruction is independent of:
 - The values of the data supplied in an f registers.
 - The values of the NZCV rags.
- The response of this instr ion to asynci nous exceptions does not vary based on:
 - The values of the .ata su, lied in any on its registers.
 - The values cone NZCV flag.

D1.1.290 WHILELT

While incrementing signed scalar less than scalar (predicate-as-counter)

Generate a predicate for a group of two or four vectors that starting from the lowest numbered element of the group is true while the incrementing value of the first, signed scalar operand is less than the second scalar operand and false thereafter up to the highest numbered element of the group.

The full width of the scalar operands is significant for the purposes of comparison, and the full width first operand is incremented by one for each destination predicate element, irrespective of the predicate result element size.

The predicate result is placed in the predicate destination register using the predicate-as-counter encoding. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

0

U_

Rr

0

Rm

SME2 (FEAT_SME2)

```
WHILELT <PNd>.<T>, <Xn>, <Xm>, <vl>
```

```
if !HaveSME2() then UNDEFINED;
1
2
  constant integer esize = 8 << UInt(siz
3
  constant integer rsize = 64;
```

- 4 integer n = UInt(Rn);
- 5 integer m = UInt(Rm); 6

0 1 0 0 1 0 1 size

- integer d = UInt('1':PNd); 7
- boolean unsigned = FALSE; boolean invert = FALSE; 8
- 9
- SVECmp op = Cmp_LT; 10 integer width = 2 << U7 (vl

Assembler Symbols

Is the nary of the deviation scalable predicate register PN8-PN15, with predicate-as-counter <PNd> encoding, $\operatorname{od}_{\mathcal{E}}$ in the "PNd" field.



- Is the 64-bit name of the first source general-purpose register, encoded in the "Rn" field. <Xn>
- < Xm >Is the 64-bit name of the second source general-purpose register, encoded in the "Rm" field.
 - Is the vl specifier, encoded in "vl": $\langle vl \rangle$

	\1/
0	VLx2
1	VLx4

- 1 CheckStreamingSVEEnabled();
- constant integer VL = CurrentVL; constant integer PL = VL DIV 8; 2
- 3

```
4
   constant integer elements = width * (VL DIV esize);
5
   bits(rsize) operand1 = X[n, rsize];
   bits(rsize) operand2 = X[m, rsize];
6
   bits(PL) result;
7
8
   boolean last = TRUE;
9
   integer count = 0;
10
11
   for e = 0 to elements-1
12
       boolean cond;
        case op of
13
14
            when Cmp_LT cond = (Int(operand1, unsigned) < Int(operand2, unsigned));</pre>
15
            when Cmp_LE cond = (Int(operand1, unsigned) <= Int(operand2, unsigned));</pre>
16
17
        last = last && cond;
18
        if last then count = count + 1;
19
        operand1 = operand1 + 1;
20
21
   result = EncodePredCount(esize, elements, count, invert, PL)
22
   PSTATE.<N,Z,C,V> = PredCountTest(elements, count, invert)
23
   P[d, PL] = result;
```

- The execution time of this instruction is independent or
 - The values of the data supplied in any of its resters.
 - The values of the NZCV flags.
- The response of this instruction to vnchro as exceptions does not vary based on:
 - The values of the data s _plied in _v of in. _gisters.
 - The values of the N \sim flags.

D1.1.291 ZERO (tile)

Zero a list of 64-bit element ZA tiles

Zeroes all bytes within each of the up to eight listed 64-bit element tiles named ZA0.D to ZA7.D, leaving the other 64-bit element tiles unmodified.

This instruction does not require the PE to be in Streaming SVE mode, and it is expected that this instruction will not experience a significant slowdown due to contention with other PEs that are executing in Streaming SVE mode.

For programmer convenience an assembler must also accept the names of 32-bit, 16-bit, and 8-bit element tiles which are converted into the corresponding set of 64-bit element tiles.

In accordance with the architecturally defined mapping between different element size tiles:

- Zeroing the 8-bit element tile name ZA0.B, or the entire array name equivalent to zeroing all eight 64-bit element tiles named ZA0.D to ZA7.D.
- Zeroing the 16-bit element tile name ZA0.H is equivalent to 2 ping 64-bit element tiles named ZA0.D, ZA2.D, ZA4.D, and ZA6.D.
- Zeroing the 16-bit element tile name ZA1.H is equivalent to zoroing the element tiles named ZA1.D, ZA3.D, ZA5.D, and ZA7.D.
- Zeroing the 32-bit element tile name ZA0.S is equivale. zeroing / -bit element tiles named ZA0.D and ZA4.D.
- Zeroing the 32-bit element tile name 7 1.S is eq. /alent to ze. oing 64-bit element tiles named ZA1.D and ZA5.D.
- Zeroing the 32-bit element tile nan. ⁷A2.S. ⁹quivalent to zeroing 64-bit element tiles named ZA2.D and ZA6.D.
- Zeroing the 32-bit element in name ZA3. In equivalent to zeroing 64-bit element tiles named ZA3.D and ZA7.D.

The preferred disasse only of this insumation uses the shortest list of tile names that represent the encoded immediate mask.

For example:

- An *j* liate w chence is 64-bit element tiles ZA0.D, ZA1.D, ZA4.D, and ZA5.D is disassembled as $\{7, 0.S, 2, 1.S\}$.
- In immediate the encodes 64-bit element tiles ZA0.D, ZA2.D, ZA4.D, and ZA6.D is disassembled as {2 0 1}.
- An an ves immediate is disassembled as {ZA}.
- An all-zeros immediate is disassembled as an empty list { }.

SME (FEAT_SME)

 31
 10
 9
 8
 7
 0

 1
 1
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0</td

ZERO { (mask >)

```
if !HaveSME() then UNDEFINED;
```

```
2 bits(8) mask = imm8;
```

```
3 constant integer esize = 64;
```

Assembler Symbols

Is a list of up to eight 64-bit element tile names separated by commas, encoded in the "imm8" <mask> field.

Operation

5

7

9

```
CheckSMEAndZAEnabled();
1
2
   constant integer SVL = CurrentSVL;
3
   constant integer dim = SVL DIV esize;
4
   bits(dim*dim*esize) result = Zeros(dim*dim*esize);
   if HaveTME() && TSTATE.depth > 0 then
6
       FailTransaction(TMFailure_ERR, FALSE);
8
   for i = 0 to 7
      if mask<i> == '1' then ZAtile[i, esize, dim*dim*esize] =
10
```

D1.1.292 ZERO (ZT0)

1

Zero ZT0

Zero all bytes of the ZT0 register.

This instruction does not require the PE to be in Streaming SVE mode, and it is expected that this instruction will not experience a significant slowdown due to contention with other PEs that are executing in Streaming SVE mode.

SME2 (FEAT_SME2)

<u>31 30 29 20 19 10 9 4 3 2 1 0</u>
ZERO { ZTO }
if !HaveSME2() then UNDEFINED;
Operation
· · · · · · · · ·
CheckSMEEnabled();
CheckSMEZTUEnabled();
if HaveTME() is TSTATE depth > 0 then
FailTransaction (TMFailure ERR, F2, F):
ZT0[512] = Zeros(512);

D1.1.293 ZIP (four registers)

Interleave elements from four vectors

Place the four-way interleaved elements from the four source vectors in the corresponding elements of the four destination vectors.

This instruction is unpredicated.

It has encodings from 2 classes: 8-bit to 64-bit elements and 128-bit element

8-bit to 64-bit elements (FEAT_SME2)



size	$\overline{}$	eneoue
00	В	
0.1	TT	

01 H 10 S 11 D

- <Zd4> Is the name of the fourth destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 4 plus 3.
- <Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.
- <Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.

```
1 CheckStreamingSVEEnabled();
 2 constant integer VL = CurrentVL;
 3
     if VL < esize * 4 then UNDEFINED;</pre>
    constant integer quads = VL DIV (esize * 4);
 4
 5 bits(VL) operand0 = Z[n, VL];
    bits(VL) operand1 = Z[n+1, VL];
bits(VL) operand2 = Z[n+2, VL];
 6
 7
 8 bits(VL) operand3 = Z[n+3, VL];
9
     bits(VL) result;
10
11
     for r = 0 to 3
12
           integer base = r * quads;
13
           for q = 0 to quads-1
                Elem[result, 4*q+0, esize] = Elem[operand0, base+q, esize];
Elem[result, 4*q+1, esize] = Elem[operand1, base+q, esize];
Elem[result, 4*q+2, esize] = Elem[operand2, base+q, esize];
Elem[result, 4*q+3, esize] = Elem[operand3, base+q, e<sup>c</sup>
14
15
16
17
18
           Z[d+r, VL] = result;
```

D1.1.294 ZIP (two registers)

Interleave elements from two vectors

Place the two-way interleaved elements from the first and second source vectors in the corresponding elements of the two destination vectors.

This instruction is unpredicated.

It has encodings from 2 classes: 8-bit to 64-bit elements and 128-bit element

8-bit to 64-bit elements (FEAT_SME2)



<T> Is the size specifier, encoded in "size":

	1
size	<t></t>
00	В
01	Н
10	S
11	D

- <Zd2> Is the name of the second destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 2 plus 1.
- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
- <Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

```
1 CheckStreamingSVEEnabled();
2 constant integer VL = CurrentVL;
   if VL < esize * 2 then UNDEFINED;
constant integer pairs = VL DIV (esize * 2);
3
4
5 bits(VL) operand0 = Z[n, VL];
6 bits(VL) operand1 = Z[m, VL];
7 bits(VL) result;
8
9
    for r = 0 to 1
         integer base = r * pairs;
10
         for p = 0 to pairs-1
11
              Elem[result, 2*p+0, esize] = Elem[operand0, base+p, esize];
Elem[result, 2*p+1, esize] = Elem[operand1, base+p, esize];
12
13
14
         Z[d+r, VL] = result;
```



D1.2 SVE2 data-processing instructions

The following SVE2 instructions are added by the SME or SME2 architecture, and are available when the PE is in Streaming SVE mode.

D1.2.1 BFMLSLB (vectors)

BFloat16 floating-point multiply-subtract long from single-precision (bottom)

This BFloat16 floating-point multiply-subtract long instruction widens the even-numbered BFloat16 elements in the first source vector and the corresponding elements in the second source vector to single-precision format and then destructively multiplies and subtracts these values without intermediate rounding from the single-precision elements of the destination vector that overlap with the corresponding P _____ 'ements in the source vectors. This instruction is unpredicated.

SVE2 (FEAT_SME2)



BFMLSLB <Zda>.S, <Zn>.H, <Zm>.H

- if !HaveSME2() then UNDEFINED; 1
- 2 integer n = UInt(Zn);
- integer m = UInt(Zm); 3
- integer da = UInt(Zda); 4 5

```
boolean op1_neg = TRUE;
```

Assembler Symbols

<Zda> Is the name of the thir ' source and destination scalable vector register, encoded in the "Zda" field.

Is the same on e first so we scalable vector register, encoded in the "Zn" field. $\langle Zn \rangle$

Id source scalable vector register, encoded in the "Zm" field. <Zm>s the na e of the

Oper. 'on

```
CheckSVL. bled();
1
2
   constant 1. •ger VL = CurrentVL;
3
   constant int _er PL = VL DIV 8;
   constant integer elements = VL DIV 32;
4
5
   bits(VL) operand1 = Z[n, VL];
6
   bits(VL) operand2 = Z[m, VL];
7
   bits(VL) operand3 = Z[da, VL];
8
   bits(VL) result;
9
10
   for e = 0 to elements-1
       bits(16) element1 = Elem[operand1, 2 * e + 0, 16];
11
12
       bits(16) element2 = Elem[operand2, 2 * e + 0, 16];
13
       bits(32) element3 = Elem[operand3, e, 32];
14
       if op1_neg then element1 = BFNeg(element1);
       Elem[result, e, 32] = BFMulAddH(element3, element1, element2, FPCR[]);
15
16
   Z[da, VL] = result;
17
```

Operational information

This instruction might be immediately preceded in program order by a MOVPREX instruction. The MOVPREX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPREX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

D1.2.2 BFMLSLB (indexed)

BFloat16 floating-point multiply-subtract long from single-precision (bottom, indexed)

This BFloat16 floating-point multiply-subtract long instruction widens the even-numbered BFloat16 elements in the first source vector and the indexed element from the corresponding 128-bit segment in the second source vector to single-precision format and then destructively multiplies and subtracts these values without intermediate rounding from the single-precision elements of the destination vector that overlap with the corresponding BFloat16 elements in the first source vector. This instruction is unpredicated.

SVE2 (FEAT_SME2)



2 integer n = UInt(Zn); 3 integer m = UInt(Zm);

1

- 4 integer da = UInt(Zda);
- 5 integer index = UInt(i3h:i3l);
- 6 boolean op1_neg = TRUE;

Assembler Symbols

- <Zda> Is the name of the third source and estinate scalable vector register, encoded in the "Zda" field.
- <Zn> Is the name of the first on the scalable vert register, encoded in the "Zn" field.
- <Zm> Is the name of the second sou. scalable vector register Z0-Z7, encoded in the "Zm" field.
- <imm> Is the immed in the range 2 to 7, encoded in the "i3h:i3l" fields.

Operation

```
CheckSVEF led(),
1
   constar int or VL
const it inte or PL =
                           Cury .tVL;
2
3
                              JIV 8;
   cor ant int ,er elements = VL DIV 32;
4
5
   cons it i' -y--
                      persegment = 128 DIV 32;
             perand1 = Z[n, VL];
6
   bits(V
   bits(VL) perand2 = Z[m, VL];
7
8
   bits(VL) o_ rand3 = Z[da, VL];
9
   bits(VL) res .t;
10
11
   for e = 0 to elements-1
12
        integer segmentbase = e - (e MOD eltspersegment);
13
        integer s = 2 * segmentbase + index;
14
        bits(16) element1 = Elem[operand1, 2 * e + 0, 16];
        bits(16) element2 = Elem[operand2, s, 16];
15
        bits(32) element3 = Elem[operand3, e, 32];
16
17
        if op1_neg then element1 = BFNeg(element1);
        Elem[result, e, 32] = BFMulAddH(element3, element1, element2, FPCR[]);
18
19
20
   Z[da, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.



D1.2.3 BFMLSLT (vectors)

BFloat16 floating-point multiply-subtract long from single-precision (top)

This BFloat16 floating-point multiply-subtract long instruction widens the odd-numbered BFloat16 elements in the first source vector and the corresponding elements in the second source vector to single-precision format and then destructively multiplies and subtracts these values without intermediate rounding from the single-precision elements of the destination vector that overlap with the corresponding BFloat16 elements in the source vectors. This instruction is unpredicated.

SVE2 (FEAT_SME2)

1

2

3 4

5



- <Zda> Is the name of the third source an destina. I scalable vector register, encoded in the "Zda" field.
- $\langle Zn \rangle$ Is the name of the first the scalable v for register, encoded in the "Zn" field.
- <Zm> Is the name of the second sole e scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabl
1
                    );
   constant intege. 'L
2
                            CurrentV :
   constant integer = VL DI
constant integr = vL DI
3
                           VI. DTV
4
                                 VL DIV 32;
   bits(' ) oper ld1 = Z
5
                               ы];
   bit /L) ope' nd2 = Z[m, VL];
6
   bits (,) or
7
                          [da, VL];
              sult;
8
   bits ()
9
   for e = 0
10
                 elements-1
       bits(16) lement1 = Elem[operand1, 2 * e + 1, 16];
11
12
       bits(16) element2 = Elem[operand2, 2 * e + 1, 16];
13
       bits(32) element3 = Elem[operand3, e, 32];
14
        if op1_neg then element1 = BFNeg(element1);
15
        Elem[result, e, 32] = BFMulAddH(element3, element1, element2, FPCR[]);
16
   Z[da, VL] = result;
17
```

Operational information

This instruction might be immediately preceded in program order by a MOVPREX instruction. The MOVPREX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPREX and this instruction is UNPREDICTABLE:

- The MOVPREX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.

• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

D1.2.4 BFMLSLT (indexed)

BFloat16 floating-point multiply-subtract long from single-precision (top, indexed)

This BFloat16 floating-point multiply-subtract long instruction widens the odd-numbered BFloat16 elements in the first source vector and the indexed element from the corresponding 128-bit segment in the second source vector to single-precision format and then destructively multiplies and subtracts these values without intermediate rounding from the single-precision elements of the destination vector that overlap with the corresponding BFloat16 elements in the first source vector. This instruction is unpredicated.

SVE2 (FEAT_SME2)



- 2 integer n = UInt(Zn);
- 3 integer m = UInt(Zm);

1

- 4 integer da = UInt(Zda);
- 5 integer index = UInt(i3h:i3l);
- 6 boolean op1_neg = TRUE;

Assembler Symbols

- <Zda> Is the name of the third source and estinate scalable vector register, encoded in the "Zda" field.
- <Zn> Is the name of the first on the scalable vector register, encoded in the "Zn" field.
- <Zm> Is the name of the second sou. scalable vector register Z0-Z7, encoded in the "Zm" field.
- <imm> Is the immed in the range 2 to 7, encoded in the "i3h:i3l" fields.

Operation

```
CheckSVEF led(),
1
   constar int or VL
const it inte or PL =
                           Cury .tVL;
2
3
                              JIV 8;
   cor ant int ,er elements = VL DIV 32;
4
5
   cons it i' -y--
                      persegment = 128 DIV 32;
             perand1 = Z[n, VL];
6
   bits(V
   bits(VL) perand2 = Z[m, VL];
7
8
   bits(VL) o_ rand3 = Z[da, VL];
9
   bits(VL) res .t;
10
11
   for e = 0 to elements-1
12
        integer segmentbase = e - (e MOD eltspersegment);
13
        integer s = 2 * segmentbase + index;
14
        bits(16) element1 = Elem[operand1, 2 * e + 1, 16];
        bits(16) element2 = Elem[operand2, s, 16];
15
        bits(32) element3 = Elem[operand3, e, 32];
16
17
        if op1_neg then element1 = BFNeg(element1);
        Elem[result, e, 32] = BFMulAddH(element3, element1, element2, FPCR[]);
18
19
20
   Z[da, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.



D1.2.5 FCLAMP

Floating-point clamp to minimum/maximum number

Clamp each floating-point element in the destination vector to between the floating-point minimum value in the corresponding element of the first source vector and the floating-point 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. If at least one element value contributing to a result is numeric and the others are either numeric or a quiet NaN, then the result is the numeric value. This instruction is unpredicated.

SVE2 (FEAT_SME2)



<Zd> Is the name of the destination sca. e_{vec} register, encoded in the "Zd" field.

<T> Is the size specifier, enco d in "size'.

size	<t></t>
00	RESERVEL
01	Н
10	
11	D

<Zn> Is ... ne of the first share scalable vector register, encoded in the "Zn" field.

<Zm> is the n: le of the s, and source scalable vector register, encoded in the "Zm" field.

Opera.

```
CheckSVEE. led();
1
2
   constant in. er VL = CurrentVL;
   constant integer PL = VL DIV 8;
3
4
   constant integer elements = VL DIV esize;
5
   bits(VL) result;
6
   bits(VL) operand1 = Z[n, VL];
7
   bits(VL) operand2 = Z[m, VL];
   bits(VL) operand3 = Z[d, VL];
8
9
10
   for e = 0 to elements-1
       bits(esize) element1 = Elem[operand1, e, esize];
11
12
       bits(esize) element2 = Elem[operand2, e, esize];
13
        bits(esize) element3 = Elem[operand3, e, esize];
14
       Elem[result, e, esize] = FPMinNum(FPMaxNum(element1, element3, FPCR[]), element2,
            \hookrightarrow FPCR[]);
   Z[d, VL] = result;
15
```

Operational information

This instruction might be immediately preceded in program order by a MOVPREX instruction. The MOVPREX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPREX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

D1.2.6 FDOT (vectors)

12

3 4 Half-precision floating-point dot product

This instruction computes the fused sum-of-products of a pair of half-precision floating-point values held in each 32-bit element of the first source and second source vectors, without intermediate rounding, and then destructively adds the single-precision sum-of-products to the corresponding single-precision element of the destination vector.

This instruction is unpredicated.

SVE2 (FEAT_SME2)



Assembler Symbols

- <Zda> Is the name of the third source and durination for the vector register, encoded in the "Zda" field.
- <Zn> Is the name of the first source scale. "vector "gister, encoded in the "Zn" field.
- <Zm> Is the name of the secor source scalar vector register, encoded in the "Zm" field.

Operation

```
1
    CheckSVEEnabled()
    constant intege L = CurrentVL;
 2
3
    constant inter . PL = V
                              DIV 8;
 4
    constant integ. eler .ts = VL DIV 32;
 5
    bits(VL) operand.
                          .[n, VL]
 6
    bits(VL)
                  nd2
                           '[m, V'
 7
    bits(V)
              oper d3 =
                                 /L];
    bits .L) resy t;
 8
 9
10
            0 s elements-1
    for e
        bit. 6) elt1_a = Elem[operand1, 2 * e + 0, 16];
11
                  elt1_b = Elem[operand1, 2 * e + 1, 16];
12
        bits
        bits(16) lt2_a = Elem[operand2, 2 * e + 0, 16];
bits(16) elt2_b = Elem[operand2, 2 * e + 1, 16];
13
14
        bits(32) sum = Elem[operand3, e, 32];
15
16
17
        sum = FPDotAdd(sum, elt1_a, elt1_b, elt2_a, elt2_b, FPCR[]);
18
        Elem[result, e, 32] = sum;
19
20
    Z[da, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPREX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.

• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

D1.2.7 FDOT (indexed)

Half-precision floating-point indexed dot product

This instruction computes the fused sum-of-products of a pair of half-precision floating-point values held in each 32-bit element of the first source vector and a pair of half-precision floating-point values in an indexed 32-bit element of the second source vector, without intermediate rounding, and then destructively adds the single-precision sum-of-products to the corresponding single-precision element of the destination vector.

The half-precision floating-point pairs within the second source vector are specified using an immediate index which selects the same pair position within each 128-bit vector segment. The index range is from 0 to 3.

This instruction is unpredicated.

SVE2 (FEAT_SME2) 10 1 1 0 0 1 0 0 0 i2 Zm 0 0 Zr Zda FDOT <Zda>.S, <Zn>.H, <Zm>.H[<imm>] if !HaveSME2() then UNDEFINED; 2 integer n = UInt(Zn); integer m = UInt(Zm); 3 integer da = UInt(Zda); 5 integer index = UInt(i2); Assembler Symbols

- $\langle Zda \rangle$ Is the name of the third rce and destin. ion scalable vector register, encoded in the "Zda" field.
- < Zn >Is the name of *t* first source suble vector register, encoded in the "Zn" field.
- $\langle Zm \rangle$ Is the name the sec nd source scalable vector register Z0-Z7, encoded in the "Zm" field.

<imm> Is the imn, "ate" lex, in the range 0 to 3, encoded in the "i2" field.

Operat[;] .

1

```
Che JVEEnab d();
1
2
   cons
         nt ir
                          CurrentVL;
             steger PL = VL DIV 8;
3
   consta
   constant rteger elements = VL DIV 32;
4
5
   constant > >ger eltspersegment = 128 DIV 32;
6
   bits(VL) ope nd1 = Z[n, VL];
7
   bits(VL) operand2 = Z[m, VL];
8
   bits(VL) operand3 = Z[da, VL];
9
   bits(VL) result;
10
11
   for e = 0 to elements-1
        integer segmentbase = e - (e MOD eltspersegment);
12
        integer s = segmentbase + index;
13
14
       bits(16) elt1_a = Elem[operand1, 2 * e + 0, 16];
15
       bits(16) elt1_b = Elem[operand1, 2 * e + 1, 16];
16
       bits(16) elt2_a = Elem[operand2, 2 * s + 0, 16];
17
       bits(16) elt2_b = Elem[operand2, 2 * s + 1, 16];
18
       bits(32) sum = Elem[operand3, e, 32];
19
20
        sum = FPDotAdd(sum, elt1_a, elt1_b, elt2_a, elt2_b, FPCR[]);
21
        Elem[result, e, 32] = sum;
22
23
   Z[da, VL] = result;
```

This instruction might be immediately preceded in program order by a MOVPREX instruction. The MOVPREX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPREX and this instruction is UNPREDICTABLE:

- The MOVPREX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

D1.2.8 PFALSE

Set all predicate elements to false

Set all elements in the destination predicate to false.

For programmer convenience, an assembler must also accept predicate-as-counter register name for the destination predicate register.



PFALSE <Pd>.B

```
1 if !HaveSVE() && !HaveSME() then UNDEFINED;
2 integer d = UInt(Pd);
```

Assembler Symbols

 $\langle Pd \rangle$ Is the name of the destination scalable predicate r ster, encluded in the d field.

Operation

```
1 CheckSVEEnabled();
```

```
2 constant integer VL = CurrentVL;
```

```
3 constant integer PL = VL DIV 8;
4 P[d, PL] = Zeros(PL);
```

Operational information

If FEAT_SVE2 is implement **SEAT_SME** mplemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
 - The value of the data supplied any of its registers.
 - The v. es of the ZCV flags.
- The response of as instruct in to asynchronous exceptions does not vary based on:

The v ues of v. d a supplied in any of its registers.

- The NZCV flags.

D1.2.9 PSEL

Predicate select between predicate register or all-false

If the indexed element of the second source predicate is true, place the contents of the first source predicate register into the destination predicate register, otherwise set the destination predicate to all-false. The indexed element is determined by the sum of a general-purpose index register and an immediate, modulo the number of elements. Does not set the condition flags.

For programmer convenience, an assembler must also accept predicate-as-counter register names for the destination predicate register and the first source predicate register.

SVE2 (FEAT_SME)



Assembler Sym 4s

```
<Pd> Is the name destinat: A scalable predicate register, encoded in the "Pd" field.
```

<Pn> / the na e of the "rst surce scalable predicate register, encoded in the "Pn" field.

<Pm. Is the *r* - read the second source scalable predicate register, encoded in the "Pm" field.

	-	
ts.	tszl	<t></t>
0	00C	RESERVED
Х	xx1	В
Х	x10	Н
Х	100	S
1	000	D

- <Wv> Is the 32-bit name of the vector select register W12-W15, encoded in the "Rv" field.
- <imm> Is the element index, in the range 0 to one less than the number of vector elements in a 128-bit vector register, encoded in "i1:tszh:tszl".

```
1 CheckSVEEnabled();
```

```
2 constant integer VL = CurrentVL;
```

```
3 constant integer PL = VL DIV 8;
```

```
constant integer elements = VL DIV esize;
4
5
  bits(PL) operand1 = P[n, PL];
6
   bits(PL) operand2 = P[m, PL];
   bits(32) idx = X[v, 32];
7
8
   integer element = (UInt(idx) + imm) MOD elements;
9
   bits(PL) result;
10
11
   if ActivePredicateElement(operand2, element, esize) then
12
       result = operand1;
13
   else
14
       result = Zeros(PL);
15
16 P[d, PL] = result;
```

- The execution time of this instruction is independent of:
 - The values of the data supplied in any of its registers.
 - The values of the NZCV flags.
- The response of this instruction to asynchronous exc tions les not vary based on:
 - The values of the data supplied in any of gister.
 - The values of the NZCV flags.
D1.2.10 REVD

Reverse 64-bit doublewords in elements (predicated)

Reverse the order of 64-bit doublewords within each active element of the source vector, and place the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified.

SVE2 (FEAT_SME)



<Zd> Is the name of the destination scalable _tor _gisu, _ded in the "Zd" field.

<Pg> Is the name of the governing scale 'e register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source _alable vec. regist encoded in the "Zn" field.

Operation

```
CheckSVEEnabled();
1
                         = CurrentVL;
2
   constant integer .
3
   constant integr PL = V<sup>T</sup> )IV 8;
   constant integ elem s = VL DIV esize;
bits(PL) mask = q L];
4
5
               rand =
                         f AnyAct veElement(mask, esize) then Z[n, VL] else Zeros(VL);
6
   bits(VL)
                     = Z .
7
   bits(VI res. t
                            VT.
8
9
          = 0 to lements-1
   for
10
                          Element(mask, e, esize) then
           Act
11
              cs(esize) element = Elem[operand, e, esize];
12
               m[result, e, esize] = Reverse(element, swsize);
13
   Z[d, VL] = r Jult;
14
```

Operational information

- The execution time of this instruction is independent of:
 - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
 - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
 - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.

- The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPREX instruction. The MOVPREX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPREX and this instruction is UNPREDICTABLE:

- The MOVPREX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

D1.2.11 SCLAMP

1 2

3

4

5

<T>

Signed clamp to minimum/maximum vector

Clamp each signed element in the destination vector to between the signed minimum value in the corresponding element of the first source vector and the signed maximum value in the corresponding element of the second source vector and destructively write the results in the corresponding elements of the destination vector. This instruction is unpredicated.

SVE2 (FEAT_SME)



Is the s	ize spec	ifier, encoded 'n "s.
size	<t></t>	
00	В	
01	Н	
10	S	
11	D	

<Zn> Is the name `th Arst source scalable vector register, encoded in the "Zn" field.

< Zm > Jr ne n the of the second source scalable vector register, encoded in the "Zm" field.

Op____tion

```
1
   CheckS
              abled();
              teger VL = CurrentVL;
2
   constant
3
   constant i. >ger elements = VL DIV esize;
   bits(VL) ope.and1 = Z[n, VL];
4
   bits(VL) operand2 = Z[m, VL];
5
   bits(VL) operand3 = Z[d, VL];
6
   bits(VL) result;
7
8
9
   for e = 0 to elements-1
10
       integer element1 = SInt(Elem[operand1, e, esize]);
11
       integer element2 = SInt(Elem[operand2, e, esize]);
       integer element3 = SInt(Elem[operand3, e, esize]);
12
13
       integer res = Min(Max(element1, element3), element2);
14
       Elem[result, e, esize] = res<esize-1:0>;
15
16
   Z[d, VL] = result;
```

Operational information

- The execution time of this instruction is independent of:
 - The values of the data supplied in any of its registers.
 - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
 - The values of the data supplied in any of its registers.
 - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination regist as this insu-tion.
- The destination register must not refer to architectural register stal referenced by any other source operand register of this instruction.

D1.2.12 SDOT (2-way, vectors)

Signed integer dot product

The signed integer dot product instruction computes the dot product of a group of two signed 16-bit integer values held in each 32-bit element of the first source vector multiplied by a group of two signed 16-bit integer values in the corresponding 32-bit element of the second source vector, and then destructively adds the widened dot product to the corresponding 32-bit element of the destination vector.

This instruction is unpredicated.

SVE2 (FEAT_SME2)

1

2

3 4

5



- <Zda> Is the name of the third source and 'estinath scalable vector register, encoded in the "Zda" field.
- <Zn> Is the name of the first on the scalable vector register, encoded in the "Zn" field.
- <Zm> Is the name of the second sou. scalable vector register, encoded in the "Zm" field.

Operation

```
1
   CheckSVEEnable \;
2
   constant intege1
                          Current
                         = VL DJ 3;
   constant ger F
3
Δ
   constar int. :r ele nts VL DIV esize;
5
   bits' \Box) oper d1 = Z_{L}
                             /L];
   bit VL) ope .nd2 = Z[m, VL];
6
   bits ) c rando
bits(V1 esult;
7
                         [da, VL];
8
   bits(V.
9
10
   for e = 0
                elements-1
       bits(esi.e) res = Elem[operand3, e, esize];
11
12
        for i = 0 to 1
13
           integer element1 = SInt(Elem[operand1, 2 * e + i, esize DIV 2]);
14
           integer element2 = SInt(Elem[operand2, 2 * e + i, esize DIV 2]);
15
            res = res + element1 * element2;
16
       Elem[result, e, esize] = res;
17
   Z[da, VL] = result;
18
```

Operational information

This instruction might be immediately preceded in program order by a MOVPREX instruction. The MOVPREX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPREX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.

Chapter D1. SME instructions D1.2. SVE2 data-processing instructions

• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

D1.2.13 SDOT (2-way, indexed)

Signed integer indexed dot product

The signed integer indexed dot product instruction computes the dot product of a group of two signed 16-bit integer values held in each 32-bit element of the first source vector multiplied by a group of two signed 16-bit integer values in an indexed 32-bit element of the second source vector, and then destructively adds the widened dot product to the corresponding 32-bit element of the destination vector.

The groups within the second source vector are specified using an immediate index which selects the same group position within each 128-bit vector segment. The index range is from 0 to 3. This instruction is unpredicated.

SVE2 (FEAT_SME2)

2

3

4

5

6



Assembler Symbols

- <Zda> Is the name of the third vrce and dest. vion scalable vector register, encoded in the "Zda" field.
- <Zn> Is the name of t¹ first source s 'able vector register, encoded in the "Zn" field.
- <Zm> Is the name the second source schable vector register Z0-Z7, encoded in the "Zm" field.
- <imm> Is the imm 'iate i .ex of a pair of two 16-bit elements within each 128-bit vector segment, in the range , s, encode .n the "i2" field.

Oper .ion

```
1
   Chec VEEn
             .teger VL = CurrentVL;
2
   consta
   constant nteger PL = VL DIV 8;
3
4
   constant > >ger elements = VL DIV esize;
5
   constant in  )er eltspersegment = 128 DIV esize;
6
   bits(VL) operand1 = Z[n, VL];
7
   bits(VL) operand2 = Z[m, VL];
   bits(VL) operand3 = Z[da, VL];
8
9
   bits(VL) result;
10
11
   for e = 0 to elements-1
12
        integer segmentbase = e - (e MOD eltspersegment);
13
        integer s = seqmentbase + index;
14
       bits(esize) res = Elem[operand3, e, esize];
15
        for i = 0 to 1
            integer element1 = SInt(Elem[operand1, 2 * e + i, esize DIV 2]);
16
17
            integer element2 = SInt(Elem[operand2, 2 * s + i, esize DIV 2]);
18
            res = res + element1 * element2;
19
        Elem[result, e, esize] = res;
20
21
   Z[da, VL] = result;
```

This instruction might be immediately preceded in program order by a MOVPREX instruction. The MOVPREX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPREX and this instruction is UNPREDICTABLE:

- The MOVPREX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

D1.2.14 SQCVTN

1 2

3

4

Signed saturating extract narrow and interleave

Saturate the signed integer value in each element of the group of two source vectors to half the orginal source element width, and place the two-way interleaved results in the half-width destination elements.

This instruction is unpredicated.

SVE2 (FEAT_SME2)



Assembler Symbols

- $\langle Zd \rangle$ Is the name of the destination scalab. vector r \sim oncoded in the "Zd" field.
- $\langle Zn1 \rangle$ Is the name of the first scalable for regime of a multi-vector sequence, encoded as "Zn" times 2.
- <Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.

Operation

```
1
   CheckSVEEnabled
   constant inter VL = constant intege. ~L
2
                             rrentVL;
                         VL DIV
3
   constant '---ger
                                     DIV (2 * esize);
4
                         ments =
5
   bits(VI' res t;
6
          = 0 to lements-1
7
   for
8
9
              .s(VL) operand = Z[n+i, VL];
              teger element = SInt(Elem[operand, e, 2 * esize]);
10
11
            En
                 [result, 2*e + i, esize] = SignedSat(element, esize);
12
13
   Z[d, VL] = result;
```

Operational information

- The execution time of this instruction is independent of:
 - The values of the data supplied in any of its registers.
 - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
 - The values of the data supplied in any of its registers.
 - The values of the NZCV flags.

D1.2.15 SQCVTUN

1

2

4

Signed saturating unsigned extract narrow and interleave

Saturate the signed integer value in each element of the group of two source vectors to unsigned integer value that is half the orginal source element width, and place the two-way interleaved results in the half-width destination elements.

This instruction is unpredicated.

SVE2 (FEAT_SME2)



- Is the name of the destination scalable to egister, called in the "Zd" field. $\langle Zd \rangle$
- $\langle Zn1 \rangle$ Is the name of the first scala¹ e ve. r regis. of a multi-vector sequence, encoded as "Zn" times 2.
- Is the name of the se no valable vector gister of a multi-vector sequence, encoded as <Zn2>"Zn" times 2 plus 1.

Operation

```
CheckSVEEnabl );
1
   constant intege. 'L
2
                          Currenty;
3
   constant ;---qer
                         VI. DIV
   constar' inc er el
                               VL DIV (2 * esize);
4
                         nts
5
   bits(' .) resu .;
6
7
   for
           0
             . 0 to 1
8
       fe
9
             ←s(VL) operand = Z[n+i, VL];
10
           in. yer element = SInt(Elem[operand, e, 2 * esize]);
           Elen esult, 2*e + i, esize] = UnsignedSat(element, esize);
11
12
13
   Z[d, VL] = result;
```

Operational information

- The execution time of this instruction is independent of:
 - The values of the data supplied in any of its registers.
 - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
 - The values of the data supplied in any of its registers.
 - The values of the NZCV flags.

D1.2.16 SQRSHRN

1

23

4

5

Signed saturating rounding shift right narrow by immediate and interleave

Shift right by an immediate value, the signed integer value in each element of the group of two source vectors and place the two-way interleaved rounded results in the half-width destination elements. Each result element is saturated to the half-width N-bit element's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$. The immediate shift amount is an unsigned value in the range 1 to 16.

This instruction is unpredicated.

SVE2 (FEAT_SME2)



- <Zd> Is the name of the destination scale 'e vector egister, encoded in the "Zd" field.
- <Zn1> Is the name of the first sector vister of a multi-vector sequence, encoded as "Zn" times 2.
- $\langle Zn2 \rangle$ Is the name of the second scale vector register of a multi-vector sequence, encoded as "Zn" times 2r is 1.
- <const> Is the imr just shif amount, in the range 1 to 16, encoded in the "imm4" field.

Operation

```
1
    Checks .Enabl l();
   cons int int or VL = C rentVL;
2
3
   con. ont in' = VL DIV 8;
              ceger eroments = VL DIV (2 * esize);
esult;
4
   const.
5
   bits(VL)
   integer i _____const = 1 << (shift-1);</pre>
6
7
   for e = 0 to elements-1
8
9
        for i = 0 to 1
            bits(VL) operand = Z[n+i, VL];
bits(2 * esize) element = Elem[operand, e, 2 * esize];
10
11
12
            integer res = (SInt(element) + round_const) >> shift;
13
             Elem[result, 2*e + i, esize] = SignedSat(res, esize);
14
```

```
15 Z[d, VL] = result;
```

Operational information

- The execution time of this instruction is independent of:
 - The values of the data supplied in any of its registers.
 - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
 - The values of the data supplied in any of its registers.
 - The values of the NZCV flags.



D1.2.17 SQRSHRUN

1

23

4

5

Signed saturating rounding shift right unsigned narrow by immediate and interleave

Shift right by an immediate value, the signed integer value in each element of the group of two source vectors and place the two-way interleaved rounded results in the half-width destination elements. Each result element is saturated to the half-width N-bit element's unsigned integer range 0 to (2^N) -1. The immediate shift amount is an unsigned value in the range 1 to 16.

This instruction is unpredicated.

SVE2 (FEAT_SME2)



- <Zd> Is the name of the destination scale 'e vector egister, encoded in the "Zd" field.
- <Zn1> Is the name of the first set able vector vister of a multi-vector sequence, encoded as "Zn" times 2.
- $\langle Zn2 \rangle$ Is the name of the second scale vector register of a multi-vector sequence, encoded as "Zn" times 2r is 1.
- <const> Is the imr just shif amount, in the range 1 to 16, encoded in the "imm4" field.

Operation

```
1
    Checks .Enabl l();
   cons int int or VL = C rentVL;
2
3
   con. ont in' = VL DIV 8;
              ceger eroments = VL DIV (2 * esize);
esult;
4
   const.
5
   bits(VL)
   integer i _____const = 1 << (shift-1);</pre>
6
7
   for e = 0 to elements-1
8
9
        for i = 0 to 1
            bits(VL) operand = Z[n+i, VL];
bits(2 * esize) element = Elem[operand, e, 2 * esize];
10
11
12
            integer res = (SInt(element) + round_const) >> shift;
13
             Elem[result, 2*e + i, esize] = UnsignedSat(res, esize);
14
```

```
15 Z[d, VL] = result;
```

Operational information

- The execution time of this instruction is independent of:
 - The values of the data supplied in any of its registers.
 - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
 - The values of the data supplied in any of its registers.
 - The values of the NZCV flags.



D1.2.18 UCLAMP

1 2

3

4

5

<T>

Unsigned clamp to minimum/maximum vector

Clamp each unsigned element in the destination vector to between the unsigned minimum value in the corresponding element of the first source vector and the unsigned maximum value in the corresponding element of the second source vector and destructively write the results in the corresponding elements of the destination vector. This instruction is unpredicated.

SVE2 (FEAT_SME)



Is the size specifier, encoded in "s.			
size	<t></t>		
00	В		
01	Н		
10	S		
11	D		

<Zn> Is the name `th Arst source scalable vector register, encoded in the "Zn" field.

< Zm > Jr ne n ue of n secor source scalable vector register, encoded in the "Zm" field.

Op____tion

```
1
   CheckS
              abled();
              teger VL = CurrentVL;
2
   constant
3
   constant i. >ger elements = VL DIV esize;
   bits(VL) ope.and1 = Z[n, VL];
4
   bits(VL) operand2 = Z[m, VL];
5
   bits(VL) operand3 = Z[d, VL];
6
   bits(VL) result;
7
8
9
   for e = 0 to elements-1
10
       integer element1 = UInt(Elem[operand1, e, esize]);
11
       integer element2 = UInt(Elem[operand2, e, esize]);
       integer element3 = UInt(Elem[operand3, e, esize]);
12
13
       integer res = Min(Max(element1, element3), element2);
14
       Elem[result, e, esize] = res<esize-1:0>;
15
16
   Z[d, VL] = result;
```

Operational information

- The execution time of this instruction is independent of:
 - The values of the data supplied in any of its registers.
 - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
 - The values of the data supplied in any of its registers.
 - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination regist as this insu-tion.
- The destination register must not refer to architectural register stal referenced by any other source operand register of this instruction.

D1.2.19 UDOT (2-way, vectors)

Unsigned integer dot product

The unsigned integer dot product instruction computes the dot product of a group of two unsigned 16-bit integer values held in each 32-bit element of the first source vector multiplied by a group of two unsigned 16-bit integer values in the corresponding 32-bit element of the second source vector, and then destructively adds the widened dot product to the corresponding 32-bit element of the destination vector.

This instruction is unpredicated.

SVE2 (FEAT_SME2)

1

2

3 4

5



- <Zda> Is the name of the third source and 'estinath' scalable vector register, encoded in the "Zda" field.
- <Zn> Is the name of the first a rescalable ver register, encoded in the "Zn" field.
- <Zm> Is the name of the second sou. scalable vector register, encoded in the "Zm" field.

Operation

```
1
   CheckSVEEnable \;
2
   constant intege1
                          Current
                         = VL DJ 3;
   constant ger F
3
Δ
   constar int. :r ele nts VL DIV esize;
5
   bits' \Box) oper d1 = Z_{L}
                             /L];
   bit VL) ope .nd2 = Z[m, VL];
6
   bits ) c rando
bits(V1 esult;
7
                         [da, VL];
8
   bits(V.
9
10
   for e = 0
                elements-1
       bits(esi.e) res = Elem[operand3, e, esize];
11
12
        for i = 0 to 1
13
           integer element1 = UInt(Elem[operand1, 2 * e + i, esize DIV 2]);
14
           integer element2 = UInt(Elem[operand2, 2 * e + i, esize DIV 2]);
15
            res = res + element1 * element2;
16
       Elem[result, e, esize] = res;
17
   Z[da, VL] = result;
18
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.

Chapter D1. SME instructions D1.2. SVE2 data-processing instructions

• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

D1.2.20 UDOT (2-way, indexed)

Unsigned integer indexed dot product

The unsigned integer indexed dot product instruction computes the dot product of a group of two unsigned 16-bit integer values held in each 32-bit element of the first source vector multiplied by a group of two unsigned 16-bit integer values in an indexed 32-bit element of the second source vector, and then destructively adds the widened dot product to the corresponding 32-bit element of the destination vector.

The groups within the second source vector are specified using an immediate index which selects the same group position within each 128-bit vector segment. The index range is from 0 to 3. This instruction is unpredicated.

SVE2 (FEAT_SME2)

2

3

4

5

6



Assembler Symbols

- <Zda> Is the name of the third irce and dest. tion scalable vector register, encoded in the "Zda" field.
- <Zn> Is the name of t¹ first source s 'able vector register, encoded in the "Zn" field.
- <Zm> Is the name the second source schable vector register Z0-Z7, encoded in the "Zm" field.
- <imm> Is the imm 'iate', ex of a pair of two 16-bit elements within each 128-bit vector segment, in the range , s, encode ... the "i2" field.

Oper .ion

```
1
   Chec VEEn
             .teger VL = CurrentVL;
2
   consta
   constant nteger PL = VL DIV 8;
3
4
   constant > >ger elements = VL DIV esize;
5
   constant in  )er eltspersegment = 128 DIV esize;
6
   bits(VL) operand1 = Z[n, VL];
7
   bits(VL) operand2 = Z[m, VL];
   bits(VL) operand3 = Z[da, VL];
8
9
   bits(VL) result;
10
11
   for e = 0 to elements-1
12
        integer segmentbase = e - (e MOD eltspersegment);
13
        integer s = seqmentbase + index;
14
       bits(esize) res = Elem[operand3, e, esize];
15
        for i = 0 to 1
            integer element1 = UInt(Elem[operand1, 2 * e + i, esize DIV 2]);
16
17
            integer element2 = UInt(Elem[operand2, 2 * s + i, esize DIV 2]);
18
            res = res + element1 * element2;
19
        Elem[result, e, esize] = res;
20
21
   Z[da, VL] = result;
```

This instruction might be immediately preceded in program order by a MOVPREX instruction. The MOVPREX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPREX and this instruction is UNPREDICTABLE:

- The MOVPREX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

D1.2.21 UQCVTN

1 2

3

4

Unsigned saturating extract narrow and interleave

Saturate the unsigned integer value in each element of the group of two source vectors to half the orginal source element width, and place the two-way interleaved results in the half-width destination elements.

This instruction is unpredicated.

SVE2 (FEAT_SME2)



Assembler Symbols

- $\langle Zd \rangle$ Is the name of the destination scalab. vector r \sim oncoded in the "Zd" field.
- <Zn1> Is the name of the first scalable ror re_E or of a multi-vector sequence, encoded as "Zn" times 2.
- <Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.

Operation

```
1
   CheckSVEEnabled
   constant inter VL = constant intege. ~L
2
                             rrentVL;
                         VL DIV
3
   constant ;---eger
4
                                     DIV (2 * esize);
                         ments =
5
   bits(VI' res t;
6
          = 0 to lements-1
7
   for
8
9
              .s(VL) operand = Z[n+i, VL];
              teger element = UInt(Elem[operand, e, 2 * esize]);
10
                 [result, 2*e + i, esize] = UnsignedSat(element, esize);
11
            En
12
13
   Z[d, VL] = result;
```

Operational information

- The execution time of this instruction is independent of:
 - The values of the data supplied in any of its registers.
 - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
 - The values of the data supplied in any of its registers.
 - The values of the NZCV flags.

D1.2.22 UQRSHRN

1

23

4

5

Unsigned saturating rounding shift right narrow by immediate and interleave

Shift right by an immediate value, the unsigned integer value in each element of the group of two source vectors and place the two-way interleaved rounded results in the half-width destination elements. Each result element is saturated to the half-width N-bit element's unsigned integer range 0 to (2^N) -1. The immediate shift amount is an unsigned value in the range 1 to 16.

This instruction is unpredicated.

SVE2 (FEAT_SME2)



- <Zd> Is the name of the destination scale 'e vector egister, encoded in the "Zd" field.
- <Zn1> Is the name of the first set able vector vister of a multi-vector sequence, encoded as "Zn" times 2.
- $\langle Zn2 \rangle$ Is the name of the second scale vector register of a multi-vector sequence, encoded as "Zn" times 2r is 1.
- <const> Is the imr juice shif amount, in the range 1 to 16, encoded in the "imm4" field.

Operation

```
1
    Checks .Enabl l();
   cons int int or VL = C rentVL;
2
3
   con. ont in' = VL DIV 8;
              ceger erc.ents = VL DIV (2 * esize);
esult;
4
   const.
5
   bits(VL)
   integer i _____const = 1 << (shift-1);</pre>
6
7
   for e = 0 to elements-1
8
9
        for i = 0 to 1
            bits(VL) operand = Z[n+i, VL];
bits(2 * esize) element = Elem[operand, e, 2 * esize];
10
11
12
            integer res = (UInt(element) + round_const) >> shift;
13
             Elem[result, 2*e + i, esize] = UnsignedSat(res, esize);
14
```

```
15 Z[d, VL] = result;
```

Operational information

- The execution time of this instruction is independent of:
 - The values of the data supplied in any of its registers.
 - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
 - The values of the data supplied in any of its registers.
 - The values of the NZCV flags.



D1.2.23 WHILEGE (predicate pair)

While decrementing signed scalar greater than or equal to scalar (pair of predicates)

Generate a pair of predicates that starting from the highest numbered element of the pair is true while the decrementing value of the first, signed scalar operand is greater than or equal to the second scalar operand and false thereafter down to the lowest numbered element of the pair.

If the second scalar operand is equal to the minimum signed integer value then a condition which includes an equality test can never fail and the result will be an all-true predicate.

The full width of the scalar operands is significant for the purposes of comparison, and the full width first operand is decremented by one for each destination predicate element, irrespective of the predicate result element size. The first general-purpose source register is not itself updated.

The lower-numbered elements are placed in the first predicate destination register. Sets the FIRST $_{,,}$ NONE (2, 'LAST (C) condition flags based on the predicate result, and the V flag to zero.

SVE2 (FEAT_SME2)



Xm>

```
WHILEGE { <Pd1>.<T>, <Pd2>.<T> }, <
```

```
1 if !HaveSME2() then UNDEFINED
```

- 2 constant **integer** esize = 8 UInt(si.
- 3 constant **integer** rsize
- 4 **integer** n = UInt(Rn);
- 5 integer m = UInt(Rm);
- 6 integer d0 = UInt(P 0');
- 7 integer d1 = UInt a:'1');
 8 boolean unsigner = FALSF
- 8 boolean unsigne = 9 SVECmp op = Cr GE;

Assemble nbols

<Pd1> .s the na e of the . Justination scalable predicate register, encoded as "Pd" times 2.

<t> '</t>	s the		r, encode	d in "size":
	ં ર	<t></t>		
_	0	В		
	01	Н		
	10	S		
	11	D		
-				

- <Pd2> Is the name of the second destination scalable predicate register, encoded as "Pd" times 2 plus 1.
- <Xn> Is the 64-bit name of the first source general-purpose register, encoded in the "Rn" field.
- <Xm> Is the 64-bit name of the second source general-purpose register, encoded in the "Rm" field.

```
1 CheckSVEEnabled();
2 constant integer VL = CurrentVL;
3 constant integer PL = VL DIV 8;
4 constant integer elements = VL DIV esize;
```

```
5 bits(PL*2) mask = Ones(PL*2);
6 bits(rsize) operand1 = X[n, rsize];
   bits(rsize) operand2 = X[m, rsize];
7
8 bits(PL*2) result;
9 boolean last = TRUE;
10
   constant integer psize = esize DIV 8;
11
12
   for e = (elements * 2) - 1 downto 0
13
       boolean cond;
14
       case op of
15
           when Cmp_GT cond = (Int(operand1, unsigned) > Int(operand2, unsigned));
            when Cmp_GE cond = (Int(operand1, unsigned) >= Int(operand2, unsigned));
16
17
18
       last = last && cond;
       bit pbit = if last then '1' else '0';
19
20
       Elem[result, e, psize] = ZeroExtend(pbit, psize);
21
       operand1 = operand1 - 1;
22
23
   PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
   P[d0, PL] = result<PL-1:0>;
24
25 P[d1, PL] = result<PL*2-1:PL>;
```

- The execution time of this instruction is ind ______t of:
 - The values of the data supplied in f y of its r sters.
 - The values of the NZCV flags
- The response of this instruction to a schron exceptions does not vary based on:
 - The values of the dat applied in a of its ugisters.
 - The values of the $\langle ZCV \rangle$ gs.

D1.2.24 WHILEGT (predicate pair)

While decrementing signed scalar greater than scalar (pair of predicates)

Generate a pair of predicates that starting from the highest numbered element of the pair is true while the decrementing value of the first, signed scalar operand is greater than the second scalar operand and false thereafter down to the lowest numbered element of the pair.

The full width of the scalar operands is significant for the purposes of comparison, and the full width first operand is decremented by one for each destination predicate element, irrespective of the predicate result element size. The first general-purpose source register is not itself updated.

The lower-numbered elements are placed in the first predicate destination register, and the higher-numbered elements in the second predicate destination register. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

SVE2 (FEAT_SME2)



WHILEGT { <Pd1>.<T>, <Pd2>.<T> }, <

```
1 if !HaveSME2() then UNDEFINED;
```

- 2 constant **integer** esize = 8 << UIn* (size
- 3 constant integer rsize = 64;
- 4 integer n = UInt(Rn);
- 5 integer m = UInt(Rm);
- 6 integer d0 = UInt(Pd:'0')
- 7 integer d1 = UInt(Pd:'1'
- 8 boolean unsigned = FALS_;
- 9 SVECmp op = Cmp_GT;

Assembler Sym⁺ .s

<Pd1> Is the name fthe arst destination scalable predicate register, encoded as "Pd" times 2.



- <Pd2> Is the name of the second destination scalable predicate register, encoded as "Pd" times 2 plus 1.
- <Xn> Is the 64-bit name of the first source general-purpose register, encoded in the "Rn" field.
- <Xm> Is the 64-bit name of the second source general-purpose register, encoded in the "Rm" field.

```
1 CheckSVEEnabled();
2 constant integer VL = CurrentVL;
3 constant integer PL = VL DIV 8;
4 constant integer elements = VL DIV esize;
5 bits(PL*2) mask = Ones(PL*2);
6 bits(rsize) operand1 = X[n, rsize];
7 bits(rsize) operand2 = X[m, rsize];
```

```
bits(PL*2) result;
 8
 9
    boolean last = TRUE;
10
    constant integer psize = esize DIV 8;
11
12
    for e = (elements \star 2) - 1 downto 0
13
         boolean cond;
14
         case op of
               when Cmp_GT cond = (Int(operand1, unsigned) > Int(operand2, unsigned));
when Cmp_GE cond = (Int(operand1, unsigned) >= Int(operand2, unsigned));
15
16
17
18
         last = last && cond;
         bit pbit = if last then '1' else '0';
Elem[result, e, psize] = ZeroExtend(pbit, psize);
19
20
21
         operand1 = operand1 - 1;
22
23
    PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
    P[d0, PL] = result<PL-1:0>;
24
25 P[d1, PL] = result<PL*2-1:PL>;
```

- The execution time of this instruction is independer of:
 - The values of the data supplied in any of its regist.
 - The values of the NZCV flags.
- The response of this instruction to asy. pronous contions does not vary based on:
 - The values of the data suppli in any is registers.
 - The values of the NZCV .ags.

D1.2.25 WHILEHI (predicate pair)

While decrementing unsigned scalar higher than scalar (pair of predicates)

Generate a pair of predicates that starting from the highest numbered element of the pair is true while the decrementing value of the first, unsigned scalar operand is higher than the second scalar operand and false thereafter down to the lowest numbered element of the pair.

The full width of the scalar operands is significant for the purposes of comparison, and the full width first operand is decremented by one for each destination predicate element, irrespective of the predicate result element size. The first general-purpose source register is not itself updated.

The lower-numbered elements are placed in the first predicate destination register, and the higher-numbered elements in the second predicate destination register. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

SVE2 (FEAT_SME2)



WHILEHI { <Pd1>.<T>, <Pd2>.<T> }, <

```
1 if !HaveSME2() then UNDEFINED;
```

- 2 constant **integer** esize = 8 << UIn* (size
- 3 constant integer rsize = 64;
- 4 integer n = UInt(Rn);
- 5 integer m = UInt(Rm);
- 6 integer d0 = UInt(Pd:'0')
- 7 integer d1 = UInt(Pd:'1'
- 8 **boolean** unsigned = TRUE
- 9 SVECmp op = Cmp_GT;

Assembler Sym⁺ .s

<Pd1> Is the name fthe arst destination scalable predicate register, encoded as "Pd" times 2.



- <Pd2> Is the name of the second destination scalable predicate register, encoded as "Pd" times 2 plus 1.
- <Xn> Is the 64-bit name of the first source general-purpose register, encoded in the "Rn" field.
- <Xm> Is the 64-bit name of the second source general-purpose register, encoded in the "Rm" field.

```
1 CheckSVEEnabled();
2 constant integer VL = CurrentVL;
3 constant integer PL = VL DIV 8;
4 constant integer elements = VL DIV esize;
5 bits(PL*2) mask = Ones(PL*2);
6 bits(rsize) operand1 = X[n, rsize];
7 bits(rsize) operand2 = X[m, rsize];
```

```
bits(PL*2) result;
 8
9
    boolean last = TRUE;
10
    constant integer psize = esize DIV 8;
11
12
    for e = (elements \star 2) - 1 downto 0
13
        boolean cond;
14
        case op of
15
             when Cmp_GT cond = (Int(operand1, unsigned) > Int(operand2, unsigned));
             when Cmp_GE cond = (Int(operand1, unsigned) >= Int(operand2, unsigned));
16
17
18
        last = last && cond;
        bit pbit = if last then '1' else '0';
Elem[result, e, psize] = ZeroExtend(pbit, psize);
19
20
21
        operand1 = operand1 - 1;
22
23
   PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
   P[d0, PL] = result<PL-1:0>;
24
25 P[d1, PL] = result<PL*2-1:PL>;
```

- The execution time of this instruction is independer of:
 - The values of the data supplied in any of its regist.
 - The values of the NZCV flags.
- The response of this instruction to asy, pronous contions does not vary based on:
 - The values of the data suppli in any is registers.
 - The values of the NZCV .ags.

D1.2.26 WHILEHS (predicate pair)

While decrementing unsigned scalar higher or same as scalar (pair of predicates)

Generate a pair of predicates that starting from the highest numbered element of the pair is true while the decrementing value of the first, unsigned scalar operand is higher or same as the second scalar operand and false thereafter down to the lowest numbered element of the pair.

If the second scalar operand is equal to the minimum unsigned integer value then a condition which includes an equality test can never fail and the result will be an all-true predicate.

The full width of the scalar operands is significant for the purposes of comparison, and the full width first operand is decremented by one for each destination predicate element, irrespective of the predicate result element size. The first general-purpose source register is not itself updated.

The lower-numbered elements are placed in the first predicate destine in region, and the higher-numbered elements in the second predicate destination register. Sets the FIRST (), NONE (2 'LAST (C) condition flags based on the predicate result, and the V flag to zero.

SVE2 (FEAT_SME2)



Xm>

```
WHILEHS { <Pd1>.<T>, <Pd2>.<T>
```

```
if !HaveSME2() then UNDEFINED
1
```

- 2 constant integer esize = 8 UInt (si
- 3 constant **integer** rsize
- 4 integer n = UInt(Rn);
- integer m = UInt(Rm); 5
- integer d0 = UInt(P 0'): 6
- integer d1 = UInt d:'1'); 7 TRUE;
- 8 boolean unsigne
- SVECmp op = Cr GE;

Assemble mbols

Justination scalable predicate register, encoded as "Pd" times 2. $\langle Pd1 \rangle$ is the name of the .

<t></t>	¶s the		r, encoded in "size":
	ē	<t></t>	
	0、	В	
	01	Н	
	10	S	
	11	D	

- <Pd2> Is the name of the second destination scalable predicate register, encoded as "Pd" times 2 plus 1.
- Is the 64-bit name of the first source general-purpose register, encoded in the "Rn" field. <Xn>
- < Xm >Is the 64-bit name of the second source general-purpose register, encoded in the "Rm" field.

```
CheckSVEEnabled();
1
  constant integer VL = CurrentVL;
2
3
  constant integer PL = VL DIV 8;
4
  constant integer elements = VL DIV esize;
```

```
5 bits(PL*2) mask = Ones(PL*2);
6 bits(rsize) operand1 = X[n, rsize];
   bits(rsize) operand2 = X[m, rsize];
7
8 bits(PL*2) result;
9 boolean last = TRUE;
10
   constant integer psize = esize DIV 8;
11
12
   for e = (elements * 2) - 1 downto 0
13
       boolean cond;
14
       case op of
15
           when Cmp_GT cond = (Int(operand1, unsigned) > Int(operand2, unsigned));
            when Cmp_GE cond = (Int(operand1, unsigned) >= Int(operand2, unsigned));
16
17
18
       last = last && cond;
       bit pbit = if last then '1' else '0';
19
20
       Elem[result, e, psize] = ZeroExtend(pbit, psize);
21
       operand1 = operand1 - 1;
22
23
   PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
   P[d0, PL] = result<PL-1:0>;
24
25 P[d1, PL] = result<PL*2-1:PL>;
```

- The execution time of this instruction is ind ______t of:
 - The values of the data supplied in f y of its r sters.
 - The values of the NZCV flags
- The response of this instruction to a schron exceptions does not vary based on:
 - The values of the dat applied in a of its ugisters.
 - The values of the $\langle ZCV \rangle$ gs.

D1.2.27 WHILELE (predicate pair)

While incrementing signed scalar less than or equal to scalar (pair of predicates)

Generate a pair of predicates that starting from the lowest numbered element of the pair is true while the incrementing value of the first, signed scalar operand is less than or equal to the second scalar operand and false thereafter up to the highest numbered element of the pair.

If the second scalar operand is equal to the maximum signed integer value then a condition which includes an equality test can never fail and the result will be an all-true predicate.

The full width of the scalar operands is significant for the purposes of comparison, and the full width first operand is incremented by one for each destination predicate element, irrespective of the predicate result element size. The first general-purpose source register is not itself updated.

The lower-numbered elements are placed in the first predicate destine in region, and the higher-numbered elements in the second predicate destination register. Sets the FIRST (), NONE (2 'LAST (C) condition flags based on the predicate result, and the V flag to zero.

SVE2 (FEAT_SME2)



Xm>

WHILELE { <Pd1>.<T>, <Pd2>.<T>

```
if !HaveSME2() then UNDEFINED
1
```

- 2 constant integer esize = 8 UInt (si
- 3 constant **integer** rsize
- 4 integer n = UInt(Rn);
- integer m = UInt(Rm); 5
- **integer** d0 = UInt(P 0'); 6
- integer d1 = UInt a:'1'); 7 FALSE
- 8 boolean unsigne
- SVECmp op = Cr LE;

Assemble mbols

<Pd1> s the na e of the. Justination scalable predicate register, encoded as "Pd" times 2.

$\langle T \rangle$ Is the , encoded in "size": <T> e 0 В 01 Η 10 S 11 D

- <Pd2> Is the name of the second destination scalable predicate register, encoded as "Pd" times 2 plus 1.
- Is the 64-bit name of the first source general-purpose register, encoded in the "Rn" field. <Xn>
- < Xm >Is the 64-bit name of the second source general-purpose register, encoded in the "Rm" field.

```
CheckSVEEnabled();
1
  constant integer VL = CurrentVL;
2
3
  constant integer PL = VL DIV 8;
4
  constant integer elements = VL DIV esize;
```

```
5 bits(PL*2) mask = Ones(PL*2);
6 bits(rsize) operand1 = X[n, rsize];
   bits(rsize) operand2 = X[m, rsize];
7
8 bits(PL*2) result;
9 boolean last = TRUE;
10
   constant integer psize = esize DIV 8;
11
12
   for e = 0 to (elements \star 2) -1
13
       boolean cond;
        case op of
14
15
            when Cmp_LT cond = (Int(operand1, unsigned) < Int(operand2, unsigned));</pre>
            when Cmp_LE cond = (Int(operand1, unsigned) <= Int(operand2, unsigned));</pre>
16
17
18
        last = last && cond;
        bit pbit = if last then '1' else '0';
19
20
        Elem[result, e, psize] = ZeroExtend(pbit, psize);
21
        operand1 = operand1 + 1;
22
23
   PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
   P[d0, PL] = result<PL-1:0>;
24
25 P[d1, PL] = result<PL*2-1:PL>;
```

- The execution time of this instruction is ind ______t of:
 - The values of the data supplied in f y of its r sters.
 - The values of the NZCV flags
- The response of this instruction to a schron exceptions does not vary based on:
 - The values of the dat applied in a of its ugisters.
 - The values of the $\langle ZCV \rangle$ gs.

D1.2.28 WHILELO (predicate pair)

While incrementing unsigned scalar lower than scalar (pair of predicates)

Generate a pair of predicates that starting from the lowest numbered element of the pair is true while the incrementing value of the first, unsigned scalar operand is lower than the second scalar operand and false thereafter up to the highest numbered element of the pair.

The full width of the scalar operands is significant for the purposes of comparison, and the full width first operand is incremented by one for each destination predicate element, irrespective of the predicate result element size. The first general-purpose source register is not itself updated.

The lower-numbered elements are placed in the first predicate destination register, and the higher-numbered elements in the second predicate destination register. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

SVE2 (FEAT_SME2)



WHILELO { <Pd1>.<T>, <Pd2>.<T> }, <

```
1 if !HaveSME2() then UNDEFINED;
```

- 2 constant integer esize = 8 << UIn* (size</pre>
- 3 constant integer rsize = 64;
- 4 integer n = UInt(Rn);
- 5 integer m = UInt(Rm);
- 6 integer d0 = UInt(Pd:'0')
- 7 integer d1 = UInt(Pd:'1'
- 8 **boolean** unsigned = TRUL
- 9 SVECmp op = Cmp_LT;

Assembler Sym⁺ .s

<Pd1> Is the name fthe arst destination scalable predicate register, encoded as "Pd" times 2.



- <Pd2> Is the name of the second destination scalable predicate register, encoded as "Pd" times 2 plus 1.
- <Xn> Is the 64-bit name of the first source general-purpose register, encoded in the "Rn" field.
- <Xm> Is the 64-bit name of the second source general-purpose register, encoded in the "Rm" field.

```
1 CheckSVEEnabled();
2 constant integer VL = CurrentVL;
3 constant integer PL = VL DIV 8;
4 constant integer elements = VL DIV esize;
5 bits(PL*2) mask = Ones(PL*2);
6 bits(rsize) operand1 = X[n, rsize];
7 bits(rsize) operand2 = X[m, rsize];
```

```
bits(PL*2) result;
 8
9
    boolean last = TRUE;
10
    constant integer psize = esize DIV 8;
11
12
    for e = 0 to (elements \star 2) -1
13
        boolean cond;
14
        case op of
15
             when Cmp_LT cond = (Int(operand1, unsigned) < Int(operand2, unsigned));</pre>
             when Cmp_LE cond = (Int(operand1, unsigned) <= Int(operand2, unsigned));</pre>
16
17
18
        last = last && cond;
        bit pbit = if last then '1' else '0';
Elem[result, e, psize] = ZeroExtend(pbit, psize);
19
20
21
        operand1 = operand1 + 1;
22
23
   PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
   P[d0, PL] = result<PL-1:0>;
24
25 P[d1, PL] = result<PL*2-1:PL>;
```

- The execution time of this instruction is independer of:
 - The values of the data supplied in any of its regist.
 - The values of the NZCV flags.
- The response of this instruction to asy, pronous contions does not vary based on:
 - The values of the data suppli in any is registers.
 - The values of the NZCV .ags.

D1.2.29 WHILELS (predicate pair)

While incrementing unsigned scalar lower or same as scalar (pair of predicates)

Generate a pair of predicates that starting from the lowest numbered element of the pair is true while the incrementing value of the first, unsigned scalar operand is lower or same as the second scalar operand and false thereafter up to the highest numbered element of the pair.

If the second scalar operand is equal to the maximum unsigned integer value then a condition which includes an equality test can never fail and the result will be an all-true predicate.

The full width of the scalar operands is significant for the purposes of comparison, and the full width first operand is incremented by one for each destination predicate element, irrespective of the predicate result element size. The first general-purpose source register is not itself updated.

The lower-numbered elements are placed in the first predicate destination register. Sets the FIRST $_{,,}$ NONE (2, 'LAST (C) condition flags based on the predicate result, and the V flag to zero.

SVE2 (FEAT_SME2)



Xm>

WHILELS { <Pd1>.<T>, <Pd2>.<T> }, <

```
1 if !HaveSME2() then UNDEFINED
```

- 2 constant **integer** esize = 8 UInt(si.
- 3 constant **integer** rsize
- 4 integer n = UInt(Rn);
- 5 integer m = UInt(Rm);
- 6 integer d0 = UInt(P 0');
- 7 integer d1 = UInt a:'1');
 8 boolean unsigner = TRUE;
- 8 boolean unsigne = 1
 9 SVECmp op = Cr LE;
- y sveenp op c. Le

Assemble vbols

<Pd1> .s the na e of the . Justination scalable predicate register, encoded as "Pd" times 2.

<t></t>	¶s th∈		z, enc	oded in	"size":
	ē	<t></t>			
	0、	В			
	01	Н			
	10	S			
	11	D			
			•		

- <Pd2> Is the name of the second destination scalable predicate register, encoded as "Pd" times 2 plus 1.
- <Xn> Is the 64-bit name of the first source general-purpose register, encoded in the "Rn" field.
- <Xm> Is the 64-bit name of the second source general-purpose register, encoded in the "Rm" field.

```
1 CheckSVEEnabled();
2 constant integer VL = CurrentVL;
3 constant integer PL = VL DIV 8;
4 constant integer elements = VL DIV esize;
```
```
5 bits(PL*2) mask = Ones(PL*2);
6 bits(rsize) operand1 = X[n, rsize];
   bits(rsize) operand2 = X[m, rsize];
7
8 bits(PL*2) result;
9 boolean last = TRUE;
10
   constant integer psize = esize DIV 8;
11
12
   for e = 0 to (elements \star 2) -1
13
       boolean cond;
        case op of
14
15
            when Cmp_LT cond = (Int(operand1, unsigned) < Int(operand2, unsigned));</pre>
            when Cmp_LE cond = (Int(operand1, unsigned) <= Int(operand2, unsigned));</pre>
16
17
18
        last = last && cond;
        bit pbit = if last then '1' else '0';
19
20
        Elem[result, e, psize] = ZeroExtend(pbit, psize);
21
        operand1 = operand1 + 1;
22
23
   PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
   P[d0, PL] = result<PL-1:0>;
24
25 P[d1, PL] = result<PL*2-1:PL>;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is ind ______t of:
 - The values of the data supplied in f y of its r sters.
 - The values of the NZCV flags
- The response of this instruction to a schron exceptions does not vary based on:
 - The values of the dat applied in a of its ugisters.
 - The values of the $\langle ZCV \rangle$ gs.

D1.2.30 WHILELT (predicate pair)

While incrementing signed scalar less than scalar (pair of predicates)

Generate a pair of predicates that starting from the lowest numbered element of the pair is true while the incrementing value of the first, signed scalar operand is less than the second scalar operand and false thereafter up to the highest numbered element of the pair.

The full width of the scalar operands is significant for the purposes of comparison, and the full width first operand is incremented by one for each destination predicate element, irrespective of the predicate result element size. The first general-purpose source register is not itself updated.

The lower-numbered elements are placed in the first predicate destination register, and the higher-numbered elements in the second predicate destination register. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

SVE2 (FEAT_SME2)



WHILELT { <Pd1>.<T>, <Pd2>.<T> }, <

```
1 if !HaveSME2() then UNDEFINED;
```

- 2 constant **integer** esize = 8 << UIn* (size
- 3 constant integer rsize = 64;
- 4 integer n = UInt(Rn);
- 5 integer m = UInt(Rm);
- 6 integer d0 = UInt(Pd:'0')
- 7 integer dl = UInt(Pd:'1'
- 8 boolean unsigned = FALS_;
- 9 SVECmp op = Cmp_LT;

Assembler Sym⁺ .s

<Pd1> Is the name fthe arst destination scalable predicate register, encoded as "Pd" times 2.



- <Pd2> Is the name of the second destination scalable predicate register, encoded as "Pd" times 2 plus 1.
- <Xn> Is the 64-bit name of the first source general-purpose register, encoded in the "Rn" field.
- <Xm> Is the 64-bit name of the second source general-purpose register, encoded in the "Rm" field.

Operation

```
1 CheckSVEEnabled();
2 constant integer VL = CurrentVL;
3 constant integer PL = VL DIV 8;
4 constant integer elements = VL DIV esize;
5 bits(PL*2) mask = Ones(PL*2);
6 bits(rsize) operand1 = X[n, rsize];
7 bits(rsize) operand2 = X[m, rsize];
```

er

```
bits(PL*2) result;
 8
9
    boolean last = TRUE;
10
    constant integer psize = esize DIV 8;
11
12
    for e = 0 to (elements \star 2) -1
13
        boolean cond;
14
        case op of
15
             when Cmp_LT cond = (Int(operand1, unsigned) < Int(operand2, unsigned));</pre>
             when Cmp_LE cond = (Int(operand1, unsigned) <= Int(operand2, unsigned));</pre>
16
17
18
        last = last && cond;
        bit pbit = if last then '1' else '0';
Elem[result, e, psize] = ZeroExtend(pbit, psize);
19
20
21
        operand1 = operand1 + 1;
22
23
   PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
   P[d0, PL] = result<PL-1:0>;
24
25 P[d1, PL] = result<PL*2-1:PL>;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independer of:
 - The values of the data supplied in any of its regist.
 - The values of the NZCV flags.
- The response of this instruction to asy, pronous contions does not vary based on:
 - The values of the data suppli in any is registers.
 - The values of the NZCV .ags.

Chapter D1. SME instructions D1.3. Base A64 instructions

D1.3 Base A64 instructions

The following Base A64 instructions are added or modified by the SME or SME2 architecture.

D1.3.1 MSR (immediate)

Move immediate value to Special Register 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.ALV
- If FEAT_SME is implemented, PSTATE A and F FATE.Zr

This instruction is used by the aliases SMSTA T, ar Since

```
MSR <pstatefield>. "<imm>
```

```
if op1 == '000' • op2 = '000' then SEE "CFINV";
if op1 == '000' && op2 = '001' then SEE "XAFLAG"
 1
                                 '001' then SEE "XAFLAG";
 2
    if op1 == '000'
3
                              == '010' then SEE "AXFLAG";
                         0
 4
                    vsten. ress( J', op1, '0100', CRm, op2, '11111', '0');
5
    AArch64 ....
6
    bits( min_E
7
    boo' an need ecure = F. LSE;
8
9
    case
        wher. 00x'
10
11
            m. EL = EL1;
        when '0.
12
            min_EL = EL1;
13
14
        when '011'
15
            min_EL = ELO;
16
        when '100'
17
            min_EL = EL2;
        when '101'
18
19
             if !HaveVirtHostExt() then
20
                 UNDEFINED;
21
             min_EL = EL2;
22
        when '110'
23
            min_EL = EL3;
24
        when '111'
25
            min_EL = EL1;
26
             need secure = TRUE;
27
28
    if (UInt(PSTATE.EL) < UInt(min_EL) || (need_secure && CurrentSecurityState() != SS_Secure))
         \rightarrowthen
29
        UNDEFINED;
```

Chapter D1. SME instructions D1.3. Base A64 instructions

```
30
31
    PSTATEField field;
32
     case opl:op2 of
33
          when '000 011'
               if !HaveUAOExt() then UNDEFINED;
34
35
               field = PSTATEField_UAO;
          when '000 100'
36
37
               if !HavePANExt() then UNDEFINED;
38
               field = PSTATEField_PAN;
          when '000 101' field = PSTATEField_SP;
39
40
          when '001 000'
41
               if CRm == '000x' then
                     if !HaveFeatNMI() then UNDEFINED;
42
43
                     field = PSTATEField_ALLINT;
44
                else
45
                     UNDEFINED;
          when '011 010'
46
47
               if !HaveDITExt() then UNDEFINED;
48
               field = PSTATEField_DIT;
49
          when '011 011'
50
               case CRm of
                     when '001x'
51
                          if !HaveSME() then UNDEFINED;
52
53
                          field = PSTATEField_SVCRSM;
54
                     when '010x'
55
                          if !HaveSME() then UNDEFINED;
56
                          field = PSTATEField_SVCRZ<sup>p</sup>
57
                     when '011x'
58
                          if !HaveSME() then UNP INED;
59
                          field = PSTATEField_
                                                         'RSMZA;
60
                     otherwise
61
                          UNDEFINED;
          when '011 100'
62
               if !HaveMTEExt() the JNDEFIN
63
          field = PSTATEFie' fCO;
when '011 110' field' r 'ATEField_DA. 'Set;
when '011 111' fiel. = PS. 'EField_DAIrClr;
64
65
66
67
          when '011 001'
               if !HaveSS' Ext() then UNL
68
                                                     'INED:
          field = 'ATEField_SSBS;
otherwise DEFINED
69
70
71
         Check that an _____n64 MSR' & access to the DAIF flags is permitted

PSTAT' == EL '& fie' IN {PSTATEField_DAIFSet, PSTATEField_DAIFClr} then

if !ELUS igAArc. '(F ) && ((EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') ||

⇔SCT R_EL1.UM. = '0')) then

if F '- 'bled() && !ELUSingAArch32(EL2) && HCR_EL2.TGE == '1' then
72
     // Check that an
73
     if PSTAT
74
75
76
                     AArcno. ystemAccessTrap(EL2, 0x18);
77
                  .se
78
                     AArch64.SystemAccessTrap(EL1, 0x18);
```

Assembler Symbols

<pstatefield> Is a PSTATE field name. For the MSR instruction, this is encoded in "op1:op2:CRm":

op1	op2	CRm	<pstatefield></pstatefield>	Architectural Feature
000	00x	XXXX	SEE PSTATE	-
000	010	XXXX	SEE PSTATE	_
000	011	XXXX	UAO	FEAT_UAO
000	100	XXXX	PAN	FEAT_PAN
000	101	XXXX	SPSel	_
000	11x	XXXX	RESERVED	_
001	000	000x	ALLINT	FEAT_NMI
001	000	001x	RESERVED	_
001	000	01xx	RESERVED	_
001	000	1xxx	RESERVED	_
001	001	XXXX	RESERVED	_
001	01x	XXXX	RESERVED	-
001	1xx	XXXX	RESERVED	-
010	XXX	XXXX	RESERVED	_
011	000	XXXX	RESERVED	-
011	001	XXXX	SSBS	FEAT_S BS
011	010	XXXX	DIT	FEA'_DIT
011	011	000x	RESERVED	-
011	011	001x	SVCRSM	1 \T_S' .
011	011	010x	SVCRZA	FEr. ME
011	011	011x	SVCRSMZA	TEAT_ ME
011	011	1xxx	RESERVED	—
011	100	XXXX	TCO	MTE
011	101	XXXX	RESFRVED	_
011	110	XXXX	DAI1 rt	-
011	111	XXXX	AIFC	-
1xx	XXX	XXXX	RESERVEL	_

For the SMSTART .d S. TOP aliases, 'is is encoded in "CRm<2:1>", where 0b01 specifies SVCRSM, 0b10 specifies SVCRZA, and 0b11 specifies SVCRSMZA.

<imm> Is a 4-bit uns' ned immediate, in a range 0 to 15, encoded in the "CRm" field. Restricted to the ran 0 to 1, c coded in "CRm<0>", when <pstatefield> is ALLINT, SVCRSM, SVCRSM, `, or c CRZA.

Alias Cr and 3

Alias	Is preferred when
SMSTART	op1 == '011' && CRm == '0xx1' && op2 == '011'
SMSTOP	op1 == '011' && CRm == '0xx0' && op2 == '011'

Operation

```
case field of
1
2
       when PSTATEField_SSBS
3
           PSTATE.SSBS = CRm<0>;
4
       when PSTATEField_SP
5
           PSTATE.SP = CRm<0>;
6
       when PSTATEField_DAIFSet
7
           PSTATE.D = PSTATE.D OR CRm<3>;
            PSTATE.A = PSTATE.A OR CRm<2>;
8
9
           PSTATE.I = PSTATE.I OR CRm<1>;
10
           PSTATE.F = PSTATE.F OR CRm<0>;
11
       when PSTATEField_DAIFClr
12
           PSTATE.D = PSTATE.D AND NOT(CRm<3>);
13
            PSTATE.A = PSTATE.A AND NOT(CRm<2>);
14
            PSTATE.I = PSTATE.I AND NOT(CRm<1>);
```

```
PSTATE.F = PSTATE.F AND NOT(CRm<0>);
15
16
        when PSTATEField_PAN
17
            PSTATE.PAN = CRm<0>;
18
        when PSTATEField_UAO
19
           PSTATE.UAO = CRm<0>;
20
        when PSTATEField_DIT
           PSTATE.DIT = CRm<0>;
21
22
        when PSTATEField_TCO
23
            PSTATE.TCO = CRm<0>;
24
        when PSTATEField_ALLINT
25
            if (PSTATE.EL == EL1 && IsHCRXEL2Enabled() && HCRX_EL2.TALLINT == '1' && CRm<0> ==
                \hookrightarrow '1') then
26
                AArch64.SystemAccessTrap(EL2, 0x18);
27
            PSTATE.ALLINT = CRm<0>;
28
        when PSTATEField_SVCRSM
29
            CheckSMEAccess();
30
            SetPSTATE_SM(CRm<0>);
31
        when PSTATEField_SVCRZA
32
            CheckSMEAccess();
33
            SetPSTATE_ZA(CRm<0>);
34
        when PSTATEField_SVCRSMZA
35
            CheckSMEAccess();
36
            SetPSTATE_SM(CRm<0>);
37
            SetPSTATE_ZA(CRm<0>);
```

D1.3.2 RPRFM

Range Prefetch Memory 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, but because these signals are only hints, the instruction cannot cause a synchronous Data Abort exception and is guaranteed not to access Device memory. It is valid for the PE to treat this instruction as a NOP.

An RPRFM instruction specifies the type of accesses and range of addresses using the following parameters:

- 'Type', in the <rprfop> operand opcode bits, specifies whether the prefetched bata will be accessed by load or store instructions.
- 'Policy', in the <rprfop> operand opcode bits, specifies ther, data is like y to be reused or if it is a streaming, non-temporal prefetch. If a streaming prefetc' is specified, in the reuseDistance' parameter is ignored.
- 'BaseAddress', in the 64-bit base register, holds the in. 'al' ock addre for the accesses.
- 'ReuseDistance', in the metadata register bit 105:0, indic, the aximum number of bytes to be accessed by this PE before executing the next RPF instruction that spaces the same range. This includes the total number of bytes inside and outside of the range time be accessed by the same PE. This parameter can be used to influence cache eviction and replacing policies, in order to retain the data in the most optimal levels of the memory hierarchy after each press. In offware cannot easily determine the amount of other memory that will be accessed, these busing powers in the range 512MiB (0b0001) to 32KiB (0b1111).
- 'Stride', in the metada a registal 'bits[59:38], is a signed, two's complement integer encoding of the number of bytes to advance the block addres. "fter 'Length' bytes have been accessed, in the range -2MiB to +2MiB-1B. A negative value indicates that the back address is advanced in a descending direction.
- 'Count', in e metad a register bits[37:22], is an unsigned integer encoding of the number of blocks of data to be accessed vir a 1, representing the range 1 to 65536 blocks. If 'Count' is 0, then the 'Stride' parameter is ignored on a single lock of contiguous bytes from 'BaseAddress' to ('BaseAddress' + 'Length' 1) is describe.
- Length' is metadata register bits[21:0], is a signed, two's complement integer encoding of the number of contributions of the starting from the current block address, without changing the block address, in the starting from the current block address 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 a 'Type' parameter of PST (prefetch for store) should be specified.

Integer (FEAT_RPRFM)



RPRFM (<rprfop>|#<imm6>), <Xm>, [<Xn|SP>]

- 1 bits(6) operation = option<2>:option<0>:S:Rt<2:0>;
- 2 integer n = UInt(Rn);
- 3 integer m = UInt(Rm);

Assembler Symbols

```
<rprfop> Is the range prefetch operation, defined as <type><policy>. .ype> is onc
```

PLD

Prefetch for load, encoded in the "Rt<0>" f .d as 0.

PST

Prefetch for store, encoded in the " $P^{t} < 0 >$ " as 1.

<policy> is one of:

KEEP

Retained or temporal refetch. data that is expected to be kept in caches to be ressed ore than once, encoded in the "option 2>:optic <0>:S <<2:1>" fields as 0b00000.

STRM

S^{*} aming or non-te. oral prefetch, for data that is xpected be accessed once and not reused, encoded in the option<2>:option<0>:S:Rt<2:1>" fields as $\sqrt{2}$, 10.

- F. out, encod. s of c "option<2>:option<0>:S:Rt<2:0>" fields, use <imm6>.
- <ibr/>simm6 Is the rigge prefetch operation encoding as an immediate, in the range 0 to 63, encoded in "cition <0>:S:Rt<2:0>". This syntax is only for encodings that are not is esentable using <rpre>rprfop>.
 - <Xm> Is the 1-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

```
1
   bits(64) address = if n == 31 then SP[] else X[n, 64];
   bits(64) metadata = X[m, 64];
2
3
   integer stride = SInt(metadata<59:38>);
4
   integer count = UInt(metadata<37:22>) + 1;
5
   integer length = SInt(metadata<21:0>);
6
   integer reuse;
7
   if metadata<63:60> == '0000' then
8
9
       reuse = -1; // Not known
10
   else
11
       reuse = 32768 << (15 - UInt(metadata<63:60>));
```

Chapter D1. SME instructions D1.3. Base A64 instructions

12
13 Hint_RangePrefetch(address, length, stride, count, reuse, operation);



D1.3.3 SMSTART

Enables access to Streaming SVE mode and SME architectural state.

0

0

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:

1 0 1 0 1 0 1

- 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.

0 0

1 0

<u></u>Ск.

1 0

Lop1

1 1

1 1

1

1





SMSTART {<option>}

is equivalent to

MSR <pstatefield>, #1

and is always the preferred disass ...bly.

Assembler Symbols



<pstatefield> 's a P ' name. For the MSR instruction, this is encoded in "op1:op2:CRm":

op1	op2	CRm	<pstatefield></pstatefield>	Architectural Feature
000	00x	XXXX	SEE PSTATE	_
000	010	XXXX	SEE PSTATE	_
000	011	XXXX	UAO	FEAT UAO
000	100	XXXX	PAN	FEAT PAN
000	101	XXXX	SPSel	_
000	11x	XXXX	RESERVED	_
001	000	000x	ALLINT	FEAT_NMI
001	000	001x	RESERVED	_
001	000	01xx	RESERVED	_
001	000	1xxx	RESERVED	_
001	001	XXXX	RESERVED	_
001	01x	XXXX	RESERVED	-
001	1xx	XXXX	RESERVED	-
010	XXX	XXXX	RESERVED	-
011	000	XXXX	RESERVED	-
011	001	XXXX	SSBS	FEAT_S BS
011	010	XXXX	DIT	FEA ^r _DIT
011	011	000x	RESERVED	-
011	011	001x	SVCRSM	⊥ `T_S' .
011	011	010x	SVCRZA	FEr. ME
011	011	011x	SVCRSMZA	TEAT_ YE
011	011	lxxx	RESERVED	-
011	100	XXXX	TCO	MTE
011	101	XXXX	RESFRVED	_
011	110	XXXX	DAI1 rt	-
011	111	XXXX	AIFC	-
1xx	XXX	XXXX	RESERVEL	_

For the SMSTART J. J. S. TOP aliases, 'his is encoded in "CRm<2:1>", where 0b01 specifies SVCRSM, 0b10 spc. 'es SVCRZA, and 0b11 specifies SVCRSMZA.

Operation

The description of **SR** (amediate) gives the operational pseudocode for this instruction.

D1.3.4 SMSTOP

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.

1 1

1 1

0

1

1





```
SMSTOP {<option>}
```

is equivalent to

MSR <pstatefield>, #0

and is always the preferred disass ...bly.

Assembler Symbols



<pstatefield> 's a P ' name. For the MSR instruction, this is encoded in "op1:op2:CRm":

op1	op2	CRm	<pstatefield></pstatefield>	Architectural Feature
000	00x	XXXX	SEE PSTATE	_
000	010	XXXX	SEE PSTATE	_
000	011	XXXX	UAO	FEAT UAO
000	100	XXXX	PAN	FEAT PAN
000	101	XXXX	SPSel	_
000	11x	XXXX	RESERVED	_
001	000	000x	ALLINT	FEAT_NMI
001	000	001x	RESERVED	_
001	000	01xx	RESERVED	_
001	000	1xxx	RESERVED	_
001	001	XXXX	RESERVED	_
001	01x	XXXX	RESERVED	-
001	1xx	XXXX	RESERVED	-
010	XXX	XXXX	RESERVED	-
011	000	XXXX	RESERVED	-
011	001	XXXX	SSBS	FEAT_S BS
011	010	XXXX	DIT	FEA ^r _DIT
011	011	000x	RESERVED	-
011	011	001x	SVCRSM	⊥ `T_S' .
011	011	010x	SVCRZA	FEr. ME
011	011	011x	SVCRSMZA	TEAT_ YE
011	011	lxxx	RESERVED	-
011	100	XXXX	TCO	MTE
011	101	XXXX	RESFRVED	_
011	110	XXXX	DAI1 rt	-
011	111	XXXX	AIFC	-
1xx	XXX	XXXX	RESERVEL	_

For the SMSTART of S. TOP aliases, 'his is encoded in "CRm<2:1>", where 0b01 specifies SVCRSM, 0b10 specifies SVCRZA, and 0b11 specifies SVCRSMZA.

Operation

The description of **SR** (amediate) gives the operational pseudocode for this instruction.

Part E Appendices

Chapter E1 Instructions affected by SME

The behavior of some non-JME in. tcions is affected when SME is implemented and the PE is in *Streaming SVE mode*.

This section lists c acted instructions by the type of effect, with a description of the changes. It is a reference summary of infect ation t⁺ can be viewed in more detail in *Arm[®] A64 Instruction Set Architecture, for A-profile architecture* [3].

E1.1 Illegal instructions in Streaming SVE mode

E1.1.1 Illegal Advanced SIMD instructions

The instruction encoding tables in this section are provided as an aid to understanding, and are consistent with the A64 ISA in Armv8.8-A and Armv9.3-A, but will require correction if subsequent versions of the A64 ISA add new instructions which overlap with these encodings.

AArch64 Advanced SIMD instructions with encodings that match the following patterns are *illegal* when the PE is in *Streaming SVE mode* and FEAT_SME_FA64 is not implemented or not enabled at the current Exception level:

A64 Encoding Pattern	Encoding Block
0x00 110x xxxx xxxx xxxx xxxx xxxx	Advanced SIMDructure load ore
0xx0 111x xxxx xxxx xxxx xxxx xxxx xxxx	Advanced S ^T MD , tor operatio
01x1 111x xxxx xxxx xxxx xxxx xxxx xxxx	Advance SIMD sing 1 teme operations
1100 1110 xxxx xxxx xxxx xxxx xxxx	Adv ced SIV cryptogra _r ny extensions

With the exception of certain vector to GPR *i*² .ger mc e instructions, and some single-element floating-point instructions that match the following pattern and which execute normally when the PE is in *Streaming SVE mode*:

A64 Encoding Pattern	Instructions or Instruction Class
0x00 1110 0000 0001 0010 11xx xxx	SMOV W Xd,Vn.B[0]
0x00 1110 0000 0010 0010 11xx xxxx xxx.	SMOV W Xd,Vn.H[0]
0100 1110 0000 0100 0010 ' x xxxx xxxx	SMOV Xd,Vn.S[0]
0000 1110 0000 0001 0° . 11xx x .x xxxx	UMOV Wd,Vn.B[0]
0000 1110 0000 0010 0011 xxxx xy .	UMOV Wd, Vn.H[0]
0000 1110 0000 100 t 11 11x. 'xx xxxx	UMOV Wd,Vn.S[0]
0100 1110 C 0 1000 11 11xx xxxx xxxx	UMOV Xd,Vn.D[0]
0101 1110 xx1x x 11x1 11xx xxxx xxxx	FMULX/FRECPS/FRSQRTS (scalar)
0101 1110 x10x xxx	FMULX/FRECPS/FRSQRTS (scalar, FP16)
01x1 1110 1x10 0001 11x1 10xx xxxx xxxx	FRECPE/FRSQRTE/FRECPX (scalar)
01x1 1110 1111 1001 11x1 10xx xxxx xxxx	FRECPE/FRSQRTE/FRECPX (scalar, FP16)

For the avoidance of doubt, A64 scalar floating-point instructions which match following encoding patterns remain *legal* when the PE is in *Streaming SVE mode*:

A64 Encoding Pattern	Instructions or Instruction Class
x001 111x xxxx xxxx xxxx xxxx xxxx	Scalar floating-point operations
xx10 110x xxxx xxxx xxxx xxxx xxxx	Load/store pair of FP registers
xx01 1100 xxxx xxxx xxxx xxxx xxxx xxxx	Load FP register (PC-relative literal)

A64 Encoding Pattern	Instructions or Instruction Class
xx11 1100 xx0x xxxx xxxx xxxx xxxx xxxx	Load/store FP register (unscaled imm)
xx11 1100 xx1x xxxx xxxx xxxx xxx10	Load/store FP register (register offset)
xx11 1101 xxxx xxxx xxxx xxxx xxxx	Load/store FP register (scaled imm)

With the exception of the following floating-point operation which is *illegal* when the PE is in *Streaming SVE mode*:

A64 Encoding Pattern	Instructions or Instruction Class	
0001 1110 0111 1110 0000 00xx xxxx xxxx	FJCVTZS	

E1.1.1.1 Vector instructions

This section lists by name those A64 Advanced SIMD instance ages in which all encoding variants are *illegal* when the PE is in *Streaming SVE mode* and FEAT **SME_FA** is not implemented or not enabled at the current Exception level.

The Advanced SIMD instructions described the folloing pages, and their aliases, are affected in this way:

- ABS: Absolute value (vector).
- ADD (vector): Add (vector).
- ADDHN, ADDHN2: Add r _rning h. ` Narro
- ADDP (scalar): Add Pair elements (sc. *).
- ADDP (vector): Add ^{*} .rwis ^{*} vector).
- ADDV: Add acros Vector.
- AESD: AES sir c round decryption
- AESE: AES ______gle rour 1 encryption.
- AESIMC: 'S inver mix columns.
- AESMC: AE
- AN^{Γ} or): Levise A^N (vector).
- F AX: B Clear a. Y R.
- 3FCVTN 8FCVTN2. Floating-point convert from single-precision to BFloat16 format (vector).
- DC (by example): BFloat16 floating-point dot product (vector, by element).
- BF. T (vector): BFloat16 floating-point dot product (vector).
- BFML UB, BFMLALT (by element): BFloat16 floating-point widening multiply-add long (by element).
- BFMLA, 3, BFMLALT (vector): BFloat16 floating-point widening multiply-add long (vector).
- BFMMLA: BFloat16 floating-point matrix multiply-accumulate into 2x2 matrix.
- BIC (vector, immediate): Bitwise bit Clear (vector, immediate).
- BIC (vector, register): Bitwise bit Clear (vector, register).
- BIF: Bitwise Insert if False.
- BIT: Bitwise Insert if True.
- BSL: Bitwise Select.
- CLS (vector): Count Leading Sign bits (vector).
- CLZ (vector): Count Leading Zero bits (vector).
- CMEQ (register): Compare bitwise Equal (vector).
- CMEQ (zero): Compare bitwise Equal to zero (vector).
- CMGE (register): Compare signed Greater than or Equal (vector).
- CMGE (zero): Compare signed Greater than or Equal to zero (vector).
- CMGT (register): Compare signed Greater than (vector).
- CMGT (zero): Compare signed Greater than zero (vector).

- CMHI (register): Compare unsigned Higher (vector).
- CMHS (register): Compare unsigned Higher or Same (vector).
- CMLE (zero): Compare signed Less than or Equal to zero (vector).
- CMLT (zero): Compare signed Less than zero (vector).
- CMTST: Compare bitwise Test bits nonzero (vector).
- CNT: Population Count per byte.
- DUP (element): Duplicate vector element to vector or scalar.
- DUP (general): Duplicate general-purpose register to vector.
- EOR (vector): Bitwise Exclusive OR (vector).
- EOR3: Three-way Exclusive OR.
- EXT: Extract vector from pair of vectors.
- FABD: Floating-point Absolute Difference.
- FABS (vector): Floating-point Absolute value (vector).
- FACGE: Floating-point Absolute Compare Greater than or Equal (v 101).
- FACGT: Floating-point Absolute Compare Greater than (vector)
- FADD (vector): Floating-point Add (vector).
- FADDP (scalar): Floating-point Add Pair of elements (sca').
- FADDP (vector): Floating-point Add Pairwise (vector)
- FCADD: Floating-point Complex Add.
- FCMEQ (register): Floating-point Compare Equal (-tor).
- FCMEQ (zero): Floating-point Compare Equal to zero .or).
- FCMGE (register): Floating-point Compare sites than "qual sector).
- FCMGE (zero): Floating-point Compare preater t in or Equ. zero (vector).
- FCMGT (register): Floating-point Col. "re Gre ------- (vector).
- FCMGT (zero): Floating-point Compare
- FCMLA: Floating-point Complex . Itiply . rumulate.
- FCMLA (by element): Float g-point mplex fultiply Accumulate (by element).
- FCMLE (zero): Floating- int Compare, is than or Equal to zero (vector).
- FCMLT (zero): Floati po. Compare Le. han zero (vector).
- FCVTAS (vector): Floating-po. Convert to Signed integer, rounding to nearest with ties to Away (vector).
- FCVTAU (vector floating-point vert to Unsigned integer, rounding to nearest with ties to Away (vector).
- FCVTL, FCV L2: Florting-point Culturer to higher precision Long (vector).
- FCVTMS -ctor): F' ating-point Convert to Signed integer, rounding toward Minus infinity (vector).
- FCVTMU (v. or relating- int Convert to Unsigned integer, rounding toward Minus infinity (vector).
- FCV⁻ FCV⁻ Floatir point Convert to lower precision Narrow (vector).
 F⁻ TNS ector): be aground to Signed integer, rounding to nearest with ties to even (vector).
- CVTNU vector): Fluating-point Convert to Unsigned integer, rounding to nearest with ties to even (vector). VT www. Ploating-point Convert to Signed integer, rounding toward Plus infinity (vector).
- PU (vector): Floating-point Convert to Unsigned integer, rounding toward Plus infinity (vector). • FC • FCV. N, FCVTXN2: Floating-point Convert to lower precision Narrow, rounding to odd (vector).
- FCVT2. vector, fixed-point): Floating-point Convert to Signed fixed-point, rounding toward Zero (vector).
- FCVTZS (vector, integer): Floating-point Convert to Signed integer, rounding toward Zero (vector).
- FCVTZU (vector, fixed-point): Floating-point Convert to Unsigned fixed-point, rounding toward Zero (vector).
- FCVTZU (vector, integer): Floating-point Convert to Unsigned integer, rounding toward Zero (vector).
- FDIV (vector): Floating-point Divide (vector).
- FJCVTZS: Floating-point Javascript Convert to Signed fixed-point, rounding toward Zero.
- FMAX (vector): Floating-point Maximum (vector).
- FMAXNM (vector): Floating-point Maximum Number (vector).
- FMAXNMP (scalar): Floating-point Maximum Number of Pair of elements (scalar).
- FMAXNMP (vector): Floating-point Maximum Number Pairwise (vector).
- FMAXNMV: Floating-point Maximum Number across Vector.
- FMAXP (scalar): Floating-point Maximum of Pair of elements (scalar).
- FMAXP (vector): Floating-point Maximum Pairwise (vector).
- FMAXV: Floating-point Maximum across Vector.

- FMIN (vector): Floating-point minimum (vector).
- FMINNM (vector): Floating-point Minimum Number (vector).
- FMINNMP (scalar): Floating-point Minimum Number of Pair of elements (scalar).
- FMINNMP (vector): Floating-point Minimum Number Pairwise (vector).
- FMINNMV: Floating-point Minimum Number across Vector.
- FMINP (scalar): Floating-point Minimum of Pair of elements (scalar).
- FMINP (vector): Floating-point Minimum Pairwise (vector).
- FMINV: Floating-point Minimum across Vector.
- FMLA (by element): Floating-point fused Multiply-Add to accumulator (by element).
- FMLA (vector): Floating-point fused Multiply-Add to accumulator (vector).
- FMLAL, FMLAL2 (by element): Floating-point fused Multiply-Add Long to accumulator (by element).
- FMLAL, FMLAL2 (vector): Floating-point fused Multiply-Add Long to accumulator (vector).
- FMLS (by element): Floating-point fused Multiply-Subtract from accumulator (by element).
- FMLS (vector): Floating-point fused Multiply-Subtract from accur ______ rtor).
- FMLSL, FMLSL2 (by element): Floating-point fused Multiply-Srift act Long frequencies accumulator (by element).
- FMLSL, FMLSL2 (vector): Floating-point fused Multiply-Subi rt Long from a cumulator (vector).
- FMOV (vector, immediate): Floating-point move immedia (vector
- FMUL (by element): Floating-point Multiply (by element).
- FMUL (vector): Floating-point Multiply (vector).
- FMULX (by element): Floating-point Multiply exte. rd (by lement).
- FNEG (vector): Floating-point Negate (vector)
- FRINT32X (vector): Floating-point Round 32 'Integ. using arrent rounding mode (vector).
- FRINT32Z (vector): Floating-point Rov to 32-b Integer L d Zero (vector).
- FRINT64X (vector): Floating-point R and to 64 and to 6
- FRINT64Z (vector): Floating-poirt Roun r-bit Integer toward Zero (vector).
- FRINTA (vector): Floating-point K nd to L gral, to nearest with ties to Away (vector).
- FRINTI (vector): Floating-p ... it Roun. 'o Intes ' using current rounding mode (vector).
- FRINTM (vector): Floati point Round Integral, toward Minus infinity (vector).
- FRINTN (vector): Floring-1 int Round to Legral, to nearest with ties to even (vector).
- FRINTP (vector): Floating-pol. Round to Integral, toward Plus infinity (vector).
- FRINTX (vector' rloating-point . und to Integral exact, using current rounding mode (vector).
- FRINTZ (vec .): Floating-point Round to Integral, toward Zero (vector).
- FSQRT (v or): Flor ag-point Square Root (vector).
- FSUB (vecto. F¹ ...ing-poir Subtract (vector).
- INS (ent): Prt vector element from another vector element.
- I^r (gene 1): Inse. ^{re} or element from general-purpose register.
- LD1 (mu' ple structures): Load multiple single-element structures to one, two, three, or four registers.
- LL . Load one single-element structure and Replicate to all lanes (of one register).
- LD2 Ultiple structures): Load multiple 2-element structures to two registers.
- LD2 (sin. e structure): Load single 2-element structure to one lane of two registers.
- LD2R: Load single 2-element structure and Replicate to all lanes of two registers.
- LD3 (multiple structures): Load multiple 3-element structures to three registers.
- LD3 (single structure): Load single 3-element structure to one lane of three registers).
- LD3R: Load single 3-element structure and Replicate to all lanes of three registers.
- LD4 (multiple structures): Load multiple 4-element structures to four registers.
- LD4 (single structure): Load single 4-element structure to one lane of four registers.
- LD4R: Load single 4-element structure and Replicate to all lanes of four registers.
- MLA (by element): Multiply-Add to accumulator (vector, by element).
- MLA (vector): Multiply-Add to accumulator (vector).
- MLS (by element): Multiply-Subtract from accumulator (vector, by element).
- MLS (vector): Multiply-Subtract from accumulator (vector).
- MOVI: Move Immediate (vector).
- MUL (by element): Multiply (vector, by element).
- MUL (vector): Multiply (vector).

- MVNI: Move inverted Immediate (vector).
- NEG (vector): Negate (vector).
- NOT: Bitwise NOT (vector).
- ORN (vector): Bitwise inclusive OR NOT (vector).
- ORR (vector, immediate): Bitwise inclusive OR (vector, immediate).
- ORR (vector, register): Bitwise inclusive OR (vector, register).
- PMUL: Polynomial Multiply.
- PMULL, PMULL2: Polynomial Multiply Long.
- RADDHN, RADDHN2: Rounding Add returning High Narrow.
- RAX1: Rotate and Exclusive OR.
- RBIT (vector): Reverse Bit order (vector).
- REV16 (vector): Reverse elements in 16-bit halfwords (vector).
- REV32 (vector): Reverse elements in 32-bit words (vector).
- REV64: Reverse elements in 64-bit doublewords (vector).
- RSHRN, RSHRN2: Rounding Shift Right Narrow (immediate).
- RSUBHN, RSUBHN2: Rounding Subtract returning High Nari
- SABA: Signed Absolute difference and Accumulate.
- SABAL, SABAL2: Signed Absolute difference and Ac inulate Lone
- SABD: Signed Absolute Difference.
- SABDL, SABDL2: Signed Absolute Difference Lo.
- SADALP: Signed Add and Accumulate Long Pairwise.
- SADDL, SADDL2: Signed Add Long (vec' .).
- SADDLP: Signed Add Long Pairwise.
- SADDLV: Signed Add Long across Veror.
- SADDW, SADDW2: Signed Add Wide.
- SCVTF (vector, fixed-point): Sign Gxed-p vt Convert to Floating-point (vector).
- SCVTF (vector, integer): Signal integer Conve. O Floating-point (vector).
- SDOT (by element): Dot ' oduct signed 'thmetic (vector, by element).
- SDOT (vector): Dot P Juc, `rned arithme. (vector).
- SHA1C: SHA1 hash update (cr. se).
- SHA1H: SHA1^{-/} .ed rotate.
- SHA1M: SH' hash ur date (majorn).
- SHA1P: S' 1 hash date (parity).
- SHA1SU0: $\sum \sqrt{1}$ _nedule v^{-1} ate 0.
- SHA 1. SHA schedul apdate 1. S' A2561 SHA25, he update (part 1).
- JHA256¹ .: SHA256 ...ash update (part 2).
- 4A25 200. 256 schedule update 0.
- 56SU1: SHA256 schedule update 1. • Sh.
- SHA. [°]H: SHA512 Hash update part 1.
- SHA512 2: SHA512 Hash update part 2.
- SHA512SU0: SHA512 Schedule Update 0.
- SHA512SU1: SHA512 Schedule Update 1.
- SHADD: Signed Halving Add.
- SHL: Shift Left (immediate).
- SHLL, SHLL2: Shift Left Long (by element size).
- SHRN, SHRN2: Shift Right Narrow (immediate).
- SHSUB: Signed Halving Subtract.
- SLI: Shift Left and Insert (immediate).
- SM3PARTW1: SM3PARTW1.
- SM3PARTW2: SM3PARTW2.
- SM3SS1: SM3SS1.
- SM3TT1A: SM3TT1A.
- SM3TT1B: SM3TT1B.
- SM3TT2A: SM3TT2A.

- SM3TT2B: SM3TT2B.
- SM4E: SM4 Encode.
- SM4EKEY: SM4 Key.
- SMAX: Signed Maximum (vector).
- SMAXP: Signed Maximum Pairwise.
- SMAXV: Signed Maximum across Vector.
- SMIN: Signed Minimum (vector).
- SMINP: Signed Minimum Pairwise.
- SMINV: Signed Minimum across Vector.
- SMLAL, SMLAL2 (by element): Signed Multiply-Add Long (vector, by element).
- SMLAL, SMLAL2 (vector): Signed Multiply-Add Long (vector).
- SMLSL, SMLSL2 (by element): Signed Multiply-Subtract Long (vector, by element).
- SMLSL, SMLSL2 (vector): Signed Multiply-Subtract Long (vector).
- SMMLA (vector): Signed 8-bit integer matrix multiply-accumulate ccus,
- SMULL, SMULL2 (by element): Signed Multiply Long (vector y element).
- SMULL, SMULL2 (vector): Signed Multiply Long (vector).
- SQABS: Signed saturating Absolute value.
- SQADD: Signed saturating Add.
- SQDMLAL, SQDMLAL2 (by element): Signed sationing Dov Ling Mu V-Add Long (by element).
- SQDMLAL, SQDMLAL2 (vector): Signed saturati. Doub' .g Multiply-Add Long.
- SQDMLSL, SQDMLSL2 (by element): Signed satural. Joubling ' ultiply-Subtract Long (by element).
- SQDMLSL, SQDMLSL2 (vector): Signed and g Douing M appy-Subtract Long.
- SQDMULH (by element): Signed satur? ...g Dout ag Multi, ...eturning High half (by element).
- SQDMULL, SQDMULL2 (by element): reasturating Doubling Multiply Long (by element).
- SQDMULL, SQDMULL2 (vector, *`igned variation* variation of the sector variable of the sec
- SQNEG: Signed saturating [▶] gate.
- SQRDMLAH (by eleme Signed Satu ting Kounding Doubling Multiply Accumulate returning High Half (by
- element).
- SQRDMLAH (* _tor): Signed S., *ating Rounding Doubling Multiply Accumulate returning High Half (vector).
- SQRDML (by ele ent): Signed Saturating Rounding Doubling Multiply Subtract returning High Half (by element).
- SQP SH (v, vr): Sig a Saturating Rounding Doubling Multiply Subtract returning High Half (vector).
- St xDML H (by c. ne): Signed saturating Rounding Doubling Multiply returning High half (by element).
- JQRDM¹ JH (vector) Signed saturating Rounding Doubling Multiply returning High half.
- St. HRN, SQRSHRN2: Signed saturating Rounded Shift Right Narrow (immediate).
- SQR. RUN, SQRSHRUN2: Signed saturating Rounded Shift Right Unsigned Narrow (immediate).
- SQSHL mmediate): Signed saturating Shift Left (immediate).
- SQSHL (register): Signed saturating Shift Left (register).
- SQSHLU: Signed saturating Shift Left Unsigned (immediate).
- SQSHRN, SQSHRN2: Signed saturating Shift Right Narrow (immediate).
- SQSHRUN, SQSHRUN2: Signed saturating Shift Right Unsigned Narrow (immediate).
- SQSUB: Signed saturating Subtract.
- SQXTN, SQXTN2: Signed saturating extract Narrow.
- SQXTUN, SQXTUN2: Signed saturating extract Unsigned Narrow.
- SRHADD: Signed Rounding Halving Add.
- SRI: Shift Right and Insert (immediate).
- SRSHL: Signed Rounding Shift Left (register).
- SRSHR: Signed Rounding Shift Right (immediate).
- SRSRA: Signed Rounding Shift Right and Accumulate (immediate).
- SSHL: Signed Shift Left (register).
- SSHLL, SSHLL2: Signed Shift Left Long (immediate).

- SSHR: Signed Shift Right (immediate).
- SSRA: Signed Shift Right and Accumulate (immediate).
- SSUBL, SSUBL2: Signed Subtract Long.
- SSUBW, SSUBW2: Signed Subtract Wide.
- ST1 (multiple structures): Store multiple single-element structures from one, two, three, or four registers.
- ST1 (single structure): Store a single-element structure from one lane of one register.
- ST2 (multiple structures): Store multiple 2-element structures from two registers.
- ST2 (single structure): Store single 2-element structure from one lane of two registers.
- ST3 (multiple structures): Store multiple 3-element structures from three registers.
- ST3 (single structure): Store single 3-element structure from one lane of three registers.
- ST4 (multiple structures): Store multiple 4-element structures from four registers.
- ST4 (single structure): Store single 4-element structure from one lane of four registers.
- SUB (vector): Subtract (vector).
- SUBHN, SUBHN2: Subtract returning High Narrow.
- SUDOT (by element): Dot product with signed and unsigned int jers (vector, element).
- SUQADD: Signed saturating Accumulate of Unsigned value.
- TBL: Table vector Lookup.
- TBX: Table vector lookup extension.
- TRN1: Transpose vectors (primary).
- TRN2: Transpose vectors (secondary).
- UABA: Unsigned Absolute difference and Accumulate
- UABAL, UABAL2: Unsigned Absolute diff .enc nd Ac mulz Long.
- UABD: Unsigned Absolute Difference (ctor).
- UABDL, UABDL2: Unsigned Absolu Differer
- UADALP: Unsigned Add and Accumulat Ver & Pairwise.
- UADDL, UADDL2: Unsigned Aacong (v or).
- UADDLP: Unsigned Add L g Pairw
- UADDLV: Unsigned sum ong across V. or.
- UADDW, UADDW2: ' .isig 1 Add Wide.
- UCVTF (vector, fixed-point): igned fixed-point Convert to Floating-point (vector).
- UCVTF (vector .leger): Unsigne integer Convert to Floating-point (vector).
- UDOT (by el_lent): Dot Product un igned arithmetic (vector, by element).
- UDOT (ve v): Dot * oduct unsigned arithmetic (vector).
- UHADD: Un nr Aalving / ld.
- UHS Unsig. 'Halvir subtract.
- JMAXP Jnsigned N. aximum Pairwise.
- MAY . Una Maximum across Vector.
- Un : Unsigned Minimum (vector).
- UMh Unsigned Minimum Pairwise.
- UMINV. Insigned Minimum across Vector.
- UMLAL, UMLAL2 (by element): Unsigned Multiply-Add Long (vector, by element).
- UMLAL, UMLAL2 (vector): Unsigned Multiply-Add Long (vector).
- UMLSL, UMLSL2 (by element): Unsigned Multiply-Subtract Long (vector, by element).
- UMLSL, UMLSL2 (vector): Unsigned Multiply-Subtract Long (vector).
- UMMLA (vector): Unsigned 8-bit integer matrix multiply-accumulate (vector).
- UMULL, UMULL2 (by element): Unsigned Multiply Long (vector, by element).
- UMULL, UMULL2 (vector): Unsigned Multiply long (vector).
- UQADD: Unsigned saturating Add.
- UQRSHL: Unsigned saturating Rounding Shift Left (register).
- UQRSHRN, UQRSHRN2: Unsigned saturating Rounded Shift Right Narrow (immediate).
- UQSHL (immediate): Unsigned saturating Shift Left (immediate).
- UQSHL (register): Unsigned saturating Shift Left (register).
- UQSHRN, UQSHRN2: Unsigned saturating Shift Right Narrow (immediate).
- UQSUB: Unsigned saturating Subtract.

- UQXTN, UQXTN2: Unsigned saturating extract Narrow.
- URECPE: Unsigned Reciprocal Estimate.
- URHADD: Unsigned Rounding Halving Add.
- URSHL: Unsigned Rounding Shift Left (register).
- URSHR: Unsigned Rounding Shift Right (immediate).
- URSQRTE: Unsigned Reciprocal Square Root Estimate.
- URSRA: Unsigned Rounding Shift Right and Accumulate (immediate).
- USDOT (by element): Dot Product with unsigned and signed integers (vector, by element).
- USDOT (vector): Dot Product with unsigned and signed integers (vector).
- USHL: Unsigned Shift Left (register).
- USHLL, USHLL2: Unsigned Shift Left Long (immediate).
- USHR: Unsigned Shift Right (immediate).
- USMMLA (vector): Unsigned and signed 8-bit integer matrix multiply-accumulate (vector).
- USQADD: Unsigned saturating Accumulate of Signed value.
- USRA: Unsigned Shift Right and Accumulate (immediate).
- USUBL, USUBL2: Unsigned Subtract Long.
- USUBW, USUBW2: Unsigned Subtract Wide.
- UZP1: Unzip vectors (primary).
- UZP2: Unzip vectors (secondary).
- XAR: Exclusive OR and Rotate.
- XTN, XTN2: Extract Narrow.
- ZIP1: Zip vectors (primary).
- ZIP2: Zip vectors (secondary).

If execution of an illegal Advanced SIMD instance of the probability of the probability

E1.1.1.2 Single-elem ... instruction.

This section lists by nar those A64 Ac need SIMD instruction pages in which only the SIMD "Vector" encoding variants can be *illeg* when the PE is in 5. *uning SVE mode*, but in which the single-element "Scalar" encoding variants are alway *legal* in *ceaning SVE mode*.

The Vector encoding of Advanced IMD instructions described in the following pages are affected in this way:

- FN JLX. 'oating pint' altiply extended.
- **xECPE**: loating-p. . Reciprocal Estimate.
- F. 7 X: Floating-point Reciprocal Exponent.¹
- FRS PTE: Floating-point Reciprocal Square Root Estimate.
- FRSQN 7: Floating-point Reciprocal Square Root Step.

E1.1.1.3 Element move to general register

The following Advanced SIMD instructions and their aliases can only be *illegal* when the PE is in *Streaming SVE mode* if their immediate vector element index is greater than zero. They are always *legal* in *Streaming SVE mode* when their element index is zero:

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

The 64-bit to top half of 128-bit and Top half of 128-bit to 64-bit variants from the following instruction page are part of the scalar floating-point instruction set and therefore execute normally when the PE is in *Streaming SVE mode*:

¹FRECPX is an exception in that it only has a single-element form.

• FMOV (general): Floating-point Move to or from general-purpose register without conversion.

E1.1.2 Illegal SVE instructions

Allocated SVE and SVE2 instructions with encodings that match the following patterns are *illegal* when the PE is in *Streaming SVE mode* and FEAT_SME_FA64 is not implemented or not enabled at the current Exception level:

A64 Encoding Pattern	SVE Instructions or Instruction Class
0000 0100 xx1x xxxx 1010 xxxx xxxx xxxx	SVE address generation
0000 0100 xx1x xxxx 1011 0xxx xxxx xxxx	SVE floating-point trig select coefficient
0000 0100 xx1x xxxx 1011 10xx xxxx xxxx	SVE floating-point exponent accest or
0000 0101 xx10 0001 100x xxxx xxxx xxxx	SVE compress active el ents
0000 0101 101x xxxx 000x xxxx xxxx xxxx	SVE permute vectoregmen.
0010 0101 xx01 1000 1111 000x xxx0 xxxx	SVE predicate ad from FR (p. "ed)
0010 0101 xx01 1001 1111 0000 0000 xxxx	SVE predicate ad f in FFR (unpredicated)
0010 0101 xx10 1000 1001 000x xxx0 0000	SVE - ite fro predic
0010 0101 xx10 1100 1001 0000 0000 0000	S' L FFR ir ialise
0100 0101 000x xxxx 0110 1xxx xxxx xxxx	Ph. J.B. JULLI (oit result)
0100 0101 xx0x xxxx 1001 10xx xxxx xxx	SVE n. ver matrix multiply accumulate
0100 0101 xx0x xxxx 1011 xxxx xxxx .xx	S 72 bitwe permute
0100 0101 xx1x xxxx 100x xxxx y .x xx	SVE2 .ring processing
0100 0101 xx1x xxxx 1010 00 . xxxx xxxx	SVE2 histogram generation (segment)
0100 0101 xx1x xxxx 110 xxxx xx x xxxx	SVE2 histogram computation (vector)
0100 0101 xx10 0000 111 3x0° J00x xxx*	SVE2 crypto unary operations
0100 0101 xx1x y 11 0x xxxx xxx	SVE2 crypto constructive binary operations
0100 0101 xy 001x 1 10 0xxxx xxxx	SVE2 crypto destructive binary operations
0110 0100 xx1 xx 1110 v xxxx xxxx	SVE floating point matrix multiply accumulate
0110 0101 xx0x x. 0000 11xx xxxx xxxx	FTSMUL
0110 0101 xx01 0xxx _000 00xx xxxx xxxx	SVE floating-point trig multiply accumulate coefficient
0110 0101 xx01 10xx 001x xxxx xxxx xxxx	SVE floating-point serial reduction (predicated)
1000 010x xx0x xxxx 0xxx xxxx xxxx xxxx	SVE 32-bit gather load byte (scalar plus 32-bit unscaled offsets)
1000 010x x00x xxxx 10xx xxxx xxxx xxxx	SVE2 32-bit gather non-temporal load (scalar plus 32-bit unscaled offsets)
1000 010x x00x xxxx 111x xxxx xxx0 xxxx	SVE 32-bit gather prefetch (vector plus immediate)
1000 0100 0x1x xxxx 0xxx xxxx xxx0 xxxx	SVE 32-bit gather prefetch (scalar plus 32-bit scaled offsets)
1000 010x x01x xxxx 1xxx xxxx xxxx xxxx	SVE 32-bit gather load (vector plus immediate)
1000 0100 1x1x xxxx 0xxx xxxx xxxx xxxx	SVE 32-bit gather load halfwords (scalar plus 32-bit scaled offsets)
1000 0101 0x1x xxxx 0xxx xxxx xxxx xxxx	SVE 32-bit gather load words (scalar plus 32-bit scaled offsets)

A64 Encoding Pattern	SVE Instructions or Instruction Class
1010 0100 001x xxxx 000x xxxx xxxx xxxx	LDIROB (scalar plus scalar)
1010 0100 101x xxxx 000x xxxx xxxx xxxx	LD1ROH (scalar plus scalar)
1010 0101 001x xxxx 000x xxxx xxxx xxxx	LD1ROW (scalar plus scalar)
1010 0101 101x xxxx 000x xxxx xxxx xxxx	LD1ROD (scalar plus scalar)
1010 0100 0010 xxxx 001x xxxx xxxx xxxx	LD1ROB (scalar plus immediate)
1010 0100 1010 xxxx 001x xxxx xxxx xxxx	LD1ROH (scalar plus immediate)
1010 0101 0010 xxxx 001x xxxx xxxx xxxx	LD1ROW (scalar plus immediate)
1010 0101 1010 xxxx 001x xxxx xxxx xxxx	LD1ROD (scalar plus immedia)
1010 010x xxxx xxxx 011x xxxx xxxx xxxx	SVE contiguous first-fav oad (scalar p. scalar)
1010 010x xxx1 xxxx 101x xxxx xxxx xxxx	SVE contiguous nor 'ault ic ' (scalar ply immediate)
1100 010x xx0x xxxx 0xxx xxxx xxxx xxxx	SVE 64-bit gat ¹ load (scalar p. un cked 32-bit unscaled offsets)
1100 010x x00x xxxx 1x0x xxxx xxxx xxxx	SVE2 64-bit ther not temporal load (scalar plus unpacked 32-bit unscaled offsets)
1100 010x x00x xxxx 111x xxxx xxx0 xxxx	SVF .4-bit ther p. '>tc ¹ vector plus immediate)
1100 010x xx1x xxxx 0xxx xxxx xxxx xxxx	/E 64-bi load (scalar plus 32-bit unpacked scaled offsets)
1100 0100 0x1x xxxx 0xxx xxxx xxx0 xxxx	SV oit gather prefetch (scalar plus unpacked 32-bit scaled offsets)
1100 010x x01x xxxx 1xxx xxxx xxxx .xx	S 764-bi, ather load (vector plus immediate)
1100 010x x10x xxxx 1xxx xxxx y .x x.	SVE bit gather load (scalar plus 64-bit unscaled offsets)
1100 010x x11x xxxx 1xxx xx xxxx xxxx	SVE 64-bit gather load (scalar plus 64-bit scaled offsets)
1100 0100 011x xxxx 1xx xxxx xx 0 xxxx	SVE 64-bit gather prefetch (scalar plus 64-bit scaled offsets)
1110 010x x00x xxxx 001 'XXY' .xxx xxxx	SVE2 64-bit scatter non-temporal store (vector plus scalar)
1110 010x x10x y 1x x, xxxx .xx	SVE2 32-bit scatter non-temporal store (vector plus scalar)
1110 010x xy . xxxx 1 Jx xxxxx xxxx	SVE scatter store with 32-bit offset
1110 010x xxx xx 101x xxxx xxxx	SVE scatter store with 64-bit offset
1110 010x xxxx xx 101x xxxx xxxx xxxx	SVE scatter store with immediate offset

The following SVE and SVE2 instructions and their aliases are affected:

- ADR: Compute vector address.
- AESD: AES single round decryption.
- AESE: AES single round encryption.
- AESIMC: AES inverse mix columns.
- AESMC: AES mix columns.
- BDEP: Scatter lower bits into positions selected by bitmask
- BEXT: Gather lower bits from positions selected by bitmask.
- BFMMLA: BFloat16 floating-point matrix multiply-accumulate.
- BGRP: Group bits to right or left as selected by bitmask.
- COMPACT: Shuffle active elements of vector to the right and fill with zero.
- FADDA: Floating-point add strictly-ordered reduction, accumulating in scalar.
- FEXPA: Floating-point exponential accelerator.

- FMMLA: Floating-point matrix multiply-accumulate.
- FTMAD: Floating-point trigonometric multiply-add coefficient.
- FTSMUL: Floating-point trigonometric starting value.
- FTSSEL: Floating-point trigonometric select coefficient.
- HISTCNT: Count matching elements in vector.
- HISTSEG: Count matchine elements in vector segments.
- LD1B (scalar plus vector): Gather load unsigned bytes to vector (vector index).
- LD1B (vector plus immediate): Gather load unsigned bytes to vector (immediate index).
- LD1D (scalar plus vector): Gather load doublewords to vector (vector index).
- LD1D (vector plus immediate): Gather load doublewords to vector (immediate index).
- LD1H (scalar plus vector): Gather load unsigned halfwords to vector (vector index).
- LD1H (vector plus immediate): Gather load unsigned halfwords to vector (immediate index).
- LD1ROB (scalar plus immediate): Contiguous load and replicate thirty-two bytes (immediate index).
- LD1ROB (scalar plus scalar): Contiguous load and replicate thirty- Juy, (scalar index).
- LD1ROD (scalar plus immediate): Contiguous load and replicat our doubles "ds (immediate index).
- LD1ROD (scalar plus scalar): Contiguous load and replicate for 4 oublewords (alar index).
- LD1ROH (scalar plus immediate): Contiguous load and relicate steen halfwords (immediate index).
- LD1ROH (scalar plus scalar): Contiguous load and rep¹ ate sixteen h. 'word' scalar index).
- LD1ROW (scalar plus immediate): Contiguous load d replice eight s (immediate index).
- LD1ROW (scalar plus scalar): Contiguous load and plicate tight words (scalar index).
- LD1SB (scalar plus vector): Gather load signed bytes L _____tor (vect___index).
- LD1SB (vector plus immediate): Gather lo^c sig. 1 bytes vect (immediate index).
- LD1SH (scalar plus vector): Gather load gned ha words to or (vector index).
- LD1SH (vector plus immediate): Gath load sig 11-1fwords to vector (immediate index).
- LD1SW (scalar plus vector): Gath r load r words to vector (vector index).
- LD1SW (vector plus immediate): her loa, igned words to vector (immediate index).
- LD1W (scalar plus vector): *C* .ther loc unsign, words to vector (vector index).
- LD1W (vector plus immediate index): Gather 1 unsigned words to vector (immediate index).
- LDFF1B (scalar plus ...ar) "ontiguous lo. first-fault unsigned bytes to vector (scalar index).
- LDFF1B (scalar plus vector): her load first-fault unsigned bytes to vector (vector index).
- LDFF1B (vector lus immediate). In ther load first-fault unsigned bytes to vector (immediate index).
- LDFF1D (sc², plus scalar): Contig. ous load first-fault doublewords to vector (scalar index).
- LDFF1D (Jar plus cctor): Gather load first-fault doublewords to vector (vector index).
- LDFF1D (ve. rr. s immed; 'e): Gather load first-fault doublewords to vector (immediate index).
- LDF scalar 'us scale'. Contiguous load first-fault unsigned halfwords to vector (scalar index).
- *V* AF1H calar pr. VF or): Gather load first-fault unsigned halfwords to vector (vector index).
- DFF1H /ector plus ...nmediate): Gather load first-fault unsigned halfwords to vector (immediate index).
- >FF'
 Just scalar): Contiguous load first-fault signed bytes to vector (scalar index).
- LL (SB (scalar plus vector): Gather load first-fault signed bytes to vector (vector index).
- LDF1 ^{TB} (vector plus immediate): Gather load first-fault signed bytes to vector (immediate index).
- LDFF15 (scalar plus scalar): Contiguous load first-fault signed halfwords to vector (scalar index).
- LDFF1SH (scalar plus vector): Gather load first-fault signed halfwords to vector (vector index).
- LDFF1SH (vector plus immediate): Gather load first-fault signed halfwords to vector (immediate index).
- LDFF1SW (scalar plus scalar): Contiguous load first-fault signed words to vector (scalar index).
- LDFF1SW (scalar plus vector): Gather load first-fault signed words to vector (vector index).
- LDFF1SW (vector plus immediate): Gather load first-fault signed words to vector (immediate index).
- LDFF1W (scalar plus scalar): Contiguous load first-fault unsigned words to vector (scalar index).
- LDFF1W (scalar plus vector): Gather load first-fault unsigned words to vector (vector index).
- LDFF1W (vector plus immediate): Gather load first-fault unsigned words to vector (immediate index).
- LDNF1B: Contiguous load non-fault unsigned bytes to vector (immediate index).
- LDNF1D: Contiguous load non-fault doublewords to vector (immediate index).
- LDNF1H: Contiguous load non-fault unsigned halfwords to vector (immediate index).
- LDNF1SB: Contiguous load non-fault signed bytes to vector (immediate index).
- LDNF1SH: Contiguous load non-fault signed halfwords to vector (immediate index).
- LDNF1SW: Contiguous load non-fault signed words to vector (immediate index).

- LDNF1W: Contiguous load non-fault unsigned words to vector (immediate index).
- LDNT1B (vector plus scalar): Gather load non-temporal unsigned bytes.
- LDNT1D (vector plus scalar): Gather load non-temporal unsigned doublewords.
- LDNT1H (vector plus scalar): Gather load non-temporal unsigned halfwords.
- LDNT1SB: Gather load non-temporal signed bytes.
- LDNT1SH: Gather load non-temporal signed halfwords.
- LDNT1SW: Gather load non-temporal signed words.
- LDNT1W (vector plus scalar): Gather load non-temporal unsigned words.
- MATCH: Detect any matching elements, setting the condition flags.
- NMATCH: Detect no matching elements, setting the condition flags.
- PMULLB: Polynomial multiply long (bottom) [128b result only].
- PMULLT: Polynomial multiply long (top) [128b result only].
- PRFB (scalar plus vector): Gather prefetch bytes (scalar plus vector).
- PRFB (vector plus immediate): Gather prefetch bytes (vector plus juneum
- PRFD (scalar plus vector): Gather prefetch doublewords (scalar us vector).
- PRFD (vector plus immediate): Gather prefetch doublewords (v * or plus imme ate).
- PRFH (scalar plus vector): Gather prefetch halfwords (sca' plus vtor).
- PRFH (vector plus immediate): Gather prefetch halfwo (vector plus immediate). .e).
- PRFW (scalar plus vector): Gather prefetch words (_____ar plus _____ctor).
- PRFW (vector plus immediate): Gather prefetch wo. (vec* plus immediate).
- RAX1: Bitwise rotate left by 1 and exclusive OR.
- RDFFR (unpredicated): Read the first-fault _gist
- RDFFR, RDFFRS (predicated): Return / cdicate (succesfu) aded elements.
- SETFFR: Initialise the first-fault regist to all tr
- SM4E: SM4 encryption and decryption.
- SM4EKEY: SM4 key updates.
- SMMLA: Signed integer m? .x multı, '-accuı. 'ate.
- ST1B (scalar plus vector) catter store L_rs from a vector (vector index).
- ST1B (vector plus imr Jia, Scatter store res from a vector (immediate index).
- ST1D (scalar plus vector): Scale store doublewords from a vector (vector index).
- ST1D (vector pl- immediate): Sc or store doublewords from a vector (immediate index).
- ST1H (scalar _us vector): Scatter stule halfwords from a vector (vector index).
- ST1H (vec plus in .ediate): Scatter store halfwords from a vector (immediate index).
- ST1W (scala. ¹u⁻ ector): S⁻ ter store words from a vector (vector index).
- ST1^v ctor p. immed[:] e): Scatter store words from a vector (immediate index).
- S^r T1B ector p. s^r ar): Scatter store non-temporal bytes.
- JTNT1D /ector plus .calar): Scatter store non-temporal doublewords.
- S'1 1W (vector plus scalar): Scatter store non-temporal words.
- TRN. TRN2 (vectors, quadwords): Interleave even or odd quadwords from two vectors.
- UMML Unsigned integer matrix multiply-accumulate.
- USMMLA: Unsigned by signed integer matrix multiply-accumulate.
- UZP1, UZP2 (vectors, quadwords): Concatenate even or odd quadwords from two vectors.
- WRFFR: Write the first-fault register.
- ZIP1, ZIP2 (vectors, quadwords): Interleave quadwords from two half vectors.

If execution of an illegal SVE or SVE2 instruction is attempted when the PE is in *Streaming SVE mode*, and SVE instructions are not configured to trap, this will cause an SME exception to be taken, as defined by rule R_{PLYVH} in C1.2.1 *Exception priorities*.

Chapter E1. Instructions affected by SME E1.2. Unimplemented SVE instructions

E1.2 Unimplemented SVE instructions

If execution of any SVE or SVE2 instruction is attempted when the PE is not in *Streaming SVE mode* and FEAT_SVE or FEAT_SVE2 is not implemented by the PE, and the instructions are not configured to trap, this will cause an SME exception to be taken, as defined by rule R_{PLYVH} in C1.2.1 *Exception priorities*.

E1.3 Reduced performance in Streaming SVE mode

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, might be significantly delayed if the PE is in Streaming SVE mode and FEAT_SME_FA64 is not implemented or not enabled at the current Exception level.

The following subsections list the instructions that are affected by this change.

E1.3.1 Scalar floating-point instructions

The following scalar floating-point instructions are affected.

- FCCMP: Floating-point Conditional quiet Compare (scalar).
- FCCMPE: Floating-point Conditional signaling Compare (scal
- FCMP: Floating-point quiet Compare (scalar).
- FCMPE: Floating-point signaling Compare (scalar).
- FCVTAS (scalar): Floating-point Convert to Signed in ger, rounding to exact with ties to Away (scalar).
- FCVTAU (scalar): Floating-point Convert to Unsign integer Junding to Learest with ties to Away (scalar).
- • FCVTMU (scalar): Floating-point Convert to
- FCVTNS (scalar): Floating-point Convert J Signe integer, by ing to nearest with ties to even (scalar).
- FCVTNU (scalar): Floating-point Cong. to Unside integer, counding to nearest with ties to even (scalar).
- FCVTPS (scalar): Floating-point Conve. o Si cumer rounding toward Plus infinity (scalar).
- FCVTPU (scalar): Floating-point powert insigned integer, rounding toward Plus infinity (scalar).
 FCVTZS (scalar, fixed-point) Float point powert to Signed fixed-point, rounding toward Zero (scalar).
- FCVTZS (scalar, integer): Jating-pon Conver D Signed integer, rounding toward Zero (scalar).
- FCVTZU (scalar, fixed a t): Floating int Convert to Unsigned fixed-point, rounding toward Zero (scalar).
- FCVTZU (scalar teger): Float. -point Convert to Unsigned integer, rounding toward Zero (scalar).

This only applies to ne variants of the following scalar floating-point instructions that write to a general-purpose register:

• FMOV (genera loating-r _____nt Move to or from general-purpose register without conversion.

E1.3.2 SVE ' structi is

The folloging SVE instructions are affected.

- ANDS dicates): Bitwise AND predicates.
- BICS (predicates): Bitwise clear predicates.
- BRKAS: Break after first true condition.
- BRKBS: Break before first true condition.
- BRKNS: Propagate break to next partition.
- BRKPAS: Break after first true condition, propagating from previous partition.
- BRKPBS: Break before first true condition, propagating from previous partition.
- CLASTA (scalar): Conditionally extract element after last to general-purpose register.
- CLASTB (scalar): Conditionally extract last element to general-purpose register.
- CMP<cc> (immediate): Compare vector to immediate.
- CMP<cc> (vectors): Compare vectors.
- CMP<cc> (wide elements): Compare vector to 64-bit wide elements.
- CNTP: Set scalar to count of true predicate elements.
- DECP (scalar): Decrement scalar by count of true predicate elements.
- EORS (predicates): Bitwise exclusive OR predicates.

- FAC<cc>: Floating-point absolute compare vectors.
- FCM<cc> (vectors): Floating-point compare vectors.
- FCM<cc> (zero): Floating-point compare vector with zero.
- INCP (scalar): Increment scalar by count of true predicate elements.
- LASTA (scalar): Extract element after last to general-purpose register.
- LASTB (scalar): Extract last element to general-purpose register.
- NANDS: Bitwise NAND predicates.
- NORS: Bitwise NOR predicates.
- ORNS (predicates): Bitwise inclusive OR inverted predicate.
- ORRS (predicates): Bitwise inclusive OR predicate.
- PFIRST: Set the first active predicate element to true.
- PNEXT: Find next active predicate.
- PTEST: Set condition flags for predicate.
- PTRUES: Initialise predicate from named constraint.
- SQDECP (scalar): Signed saturating decrement scalar by count or up predical elements.
- SQINCP (scalar): Signed saturating increment scalar by count corrue predicate ements.
- UQDECP (scalar): Unsigned saturating decrement scalar ¹ count true predicte elements.
- UQINCP (scalar): Unsigned saturating increment scalar y count of u. pred: .te elements.

Chapter E2 SME Shared pseudocode

This section provides the full inform ion for shared pseudocode functions added or modified by SME or SME2.

This content is from t' **2022-12** version. *Arm® A64 Instruction Set Architecture, for A-profile architecture* [3], which contains the cfinitive letails of the pseudocode.

Chapter E2. SME Shared pseudocode E2.1. Pseudocode functions

E2.1 Pseudocode functions

E2.1.1 AArch64.CheckFPAdvSIMDEnabled

```
1 // AArch64.CheckFPAdvSIMDEnabled()
2
  3
4 AArch64.CheckFPAdvSIMDEnabled()
5
     AArch64.CheckFPEnabled();
      // Check for illegal use of Advanced
6
7
     // SIMD in Streaming SVE Mode
     if HaveSME() && PSTATE.SM == '1' && !IsFullA64Enabled() then
8
9
         SMEAccessTrap(SMEExceptionType_Streaming, PSTATE.EL);
```

E2.1.2 BFDotAdd

```
1 // BFDotAdd()
 2
     // ========
 3 // BFloat16 2-way dot-product and add to single-pre-
                                                                                      .sion
 4
    // result = addend + op1_a*op2_a + op1_b*op2_b
 5

        6
        bits(32)
        BFDotAdd(bits(32)
        addend,
        bits(16)
        op1_a
        bit
        16)
        op1 b,

        7
        bits(16)
        op2_a,
        bits(16)
        op2_b,
        F
        Type fpr_in)

 8
            FPCRType fpcr = fpcr_in;
 9
10
           bits(32) prod;
11
12
           bits(32) result;
            if !HaveEBF16() || fpcr.EBF = 0' th // Standard BFloat16 behaviors
13
                 prod = FPAdd_BF16(BFM_H(op1_.op2_.BFMulH(op1_b, op2_b));
14
15
                  result = FPAdd_BF16 ddend, p1 ');
16
            else
                                                                               / Extended BFloat16 behaviors
                 boolean isbfloar o = PUE;
17
                 boolean fpexc = FALSE; // Do not generate floating-point exceptions
fpcr.DN = ' , Generate default NaN values
prod = FPr c(op1_a, op1_b, 2_a, op2_b, fpcr, isbfloat16, fpexc);
result _ PAdd(a dend, prod, fpcr, fpexc);
18
19
20
21
22
23
```

return resu.

E2.1.3 BFNeg

```
1
    // ==-
2
3
   bits(16) _ 'eg(bits(16) op)
boolean nor_altfp = TRUE; // Honor alternate handling
4
5
6
        return BFNeg(op, honor_altfp);
7
   // BFNeg()
8
9
   // ===
10
   bits(16) BFNeg(bits(16) op, boolean honor_altfp)
11
12
13
        if honor_altfp && !UsingAArch32() && HaveAltFP() then
14
            FPCRType fpcr = FPCR[];
15
            if fpcr.AH == '1' then
16
                boolean fpexc = FALSE;
17
                boolean isbfloat16 = TRUE;
                (fptype, -, -) = FPUnpackBase(op, fpcr, fpexc, isbfloat16);
18
19
                if fptype IN {FPType_SNaN, FPType_QNaN} then
20
21
                                       // When fpcr.AH=1, sign of NaN has no consequence
                    return op;
22
23
   return NOT(op<15>) : op<14:0>;
```

E2.1.4 CheckFPAdvSIMDEnabled64

E2.1.5 CheckNonStreamingSVEEnabled

```
1 // CheckNonStreamingSVEEnabled()
2
  // ===
  // Checks for traps on SVE instructions that are not legal in streaming mode.
3
4
5
  CheckNonStreamingSVEEnabled()
6
      CheckSVEEnabled();
7
       if HaveSME() && PSTATE.SM == '1' && !IsFullA64Enab ed()
8
                                                                  en
           SMEAccessTrap(SMEExceptionType_Streaming, PS .fE.EL);
9
```

E2.1.6 CheckSMEAccess

```
1 // CheckSMEAccess()
 2
     // =
 3 // Check that access to SME System resisters i
                                                                                     enabled.
 4
 5
      CheckSMEAccess()
 6
             boolean disabled;
            boolean disabled;
// Check if access disable in _CR_EL
if PSTATE.EL IN {EL0, E', && !Is. 'ost() 'en
    // Check SME at E' TL1
    case CPACR_EL1.5 LN
        when 'x0' alsablea TRUE;
        when '0' disablea TRUE;
        when '0' disabled = STATE.EL == EL0;
        when 'r' disabled = F_SE;
    if disa' id then SMEAccessTrap(SMEExceptionType_AccessTrap, EL1);
 7
 8
 9
10
11
12
13
14
15
            if PSTATE.E TN ' _0, EL1, EL2} && EL2Enabled() then
if HaveVi __stExt() __ HCR_EL2.E2H == '1' then
Chec _SME a' _LL2
C :e CPTF_ T.' _SMEN of
when 'x __disabled = TRUE;
16
17
18
19
20
                                 '01' disabled = PSTATE.EL == ELO && HCR_EL2.TGE == '1';
when '11' disabled = FALSE;
21
22
23
                           if disabled then SMEAccessTrap(SMEExceptionType_AccessTrap, EL2);
24
                    è.
25
                            CPTR_EL2.TSM == '1' then SMEAccessTrap(SMEExceptionType_AccessTrap, EL2);
26
             // Check if access disabled in CPTR_EL3
27
28
             if HaveEL(EL3) then
29
                    if CPTR_EL3.ESM == '0' then SMEAccessTrap(SMEExceptionType_AccessTrap, EL3);
```

E2.1.7 CheckSMEAndZAEnabled

Chapter E2. SME Shared pseudocode E2.1. Pseudocode functions

E2.1.8 CheckSMEEnabled

```
// CheckSMEEnabled()
 1
2 // =============
3
 4
    CheckSMEEnabled()
5
        boolean disabled;
         // Check if access disabled in CPACR_EL1
 6
 7
        if PSTATE.EL IN {EL0, EL1} && !IsInHost() then
8
             // Check SME at EL0/EL1
 9
             case CPACR_EL1.SMEN of
                 when 'x0' disabled = TRUE;
when '01' disabled = PSTATE.EL == EL0;
10
11
                  when '11' disabled = FALSE;
12
13
             if disabled then SMEAccessTrap(SMEExceptionType_AccessTrap, EL1);
14
15
             // Check SIMD&FP at ELO/EL1
16
             case CPACR_EL1.FPEN of
17
                  when 'x0' disabled = TRUE;
                  when '01' disabled = PSTATE.EL == EL0;
18
                  when '11' disabled = FALSE;
19
20
             if disabled then AArch64.AdvSIMDFPAccessTr (EL1);
21
22
        if PSTATE.EL IN {ELO, EL1, EL2} && EL2Enabl. () the
23
             if HaveVirtHostExt() && HCR_EL2.E2H == '1 hr
24
                  // Check SME at EL2
25
                  case CPTR_EL2.SMEN of
26
                      when 'x0' disabled = TF _;
                      when '01' disabled = `TATE.EI == ELO && nCR_EL2.TGE == '1';
when '11' disabled = 1 SE;
27
28
                                                     MEExceptionType_AccessTrap, EL2);
29
                  if disabled then SME/ ressTi
30
                  // Check SIMD&FP _ EL2
31
                  case CPTR_EL2.F _N of
32
                      when 'x0' abled = TRb
when 'c dis led = PSTA1 .EL == EL0 && HCR_EL2.TGE == '1';
33
34
                      when '11' disa. d = FALSE;
35
36
                  if disa .ed then AArc '4.AdvSIMDFPAccessTrap(EL2);
37
             else
                  if IR_EL2 `M == '1' then SMEAccessTrap(SMEExceptionType_AccessTrap, EL2);
i. PTR_EV IFP == '1' then AArch64.AdvSIMDFPAccessTrap(EL2);
38
39
40
41
         // Cb
                 if ac
                           ss disa<sup>1</sup> .d in CPTR_EL3
        if .vet. EL3) . n
if CP _EL3.E.
42
                                '0' then SMEAccessTrap(SMEExceptionType_AccessTrap, EL3);
43
             if C' R EL3.TFP == '1' then AArch64.AdvSIMDFPAccessTrap(EL3);
44
```

E2.1.9 CheckSh. <T0Enabled

```
// CheckSMEZ_JEnabled()
1
2
   // Checks for ZTO enabled.
3
4
5
   CheckSMEZT0Enabled()
        // Check if ZA and ZTO are inactive in \ensuremath{\mathsf{PSTATE}}
6
       if PSTATE.ZA == '0' then
7
8
            SMEAccessTrap(SMEExceptionType_InactiveZA, PSTATE.EL);
9
        // Check if ELO/EL1 accesses to ZTO are disabled in SMCR_EL1
10
11
       if PSTATE.EL IN {EL0, EL1} && !IsInHost() then
            if SMCR_EL1.EZT0 == '0' then
12
13
               SMEAccessTrap(SMEExceptionType_InaccessibleZT0, EL1);
14
15
        // Check if EL0/EL1/EL2 accesses to ZT0 are disabled in SMCR_EL2
       if PSTATE.EL IN {EL0, EL1, EL2} && EL2Enabled() then
16
            if SMCR_EL2.EZT0 == '0' then
17
18
               SMEAccessTrap(SMEExceptionType_InaccessibleZTO, EL2);
```

Chapter E2. SME Shared pseudocode E2.1. Pseudocode functions

```
19
20 // Check if all accesses to ZTO are disabled in SMCR_EL3
21 if HaveEL(EL3) then
22 if SMCR_EL3.EZTO == '0' then
23 SMEAccessTrap(SMEExceptionType_InaccessibleZTO, EL3);
```

E2.1.10 CheckStreamingSVEAndZAEnabled

E2.1.11 CheckStreamingSVEEnabled

E2.1.12 CounterToPredicate

```
1 // CounterToPredicate()
2
   3
4 bits(width) CounterT Predicate( $ $(16) pred, integer width)
      integer count;
5
6
       integer esiz
       integer el ents;
constant i. ger = CurrentVL;
constant int. PL = VL .V 8;
7
8
9
       10
11
12
                  + = pred<15> == '1';
13
       rolean
14
       ass . width == PL || width == PL*2 || width == PL*3 || width == PL*4;
15
16
       if IsZe (pred<3:0>) then
17
18
           retuin Zeros(width);
19
20
       case pred<3:0> of
21
           when 'xxx1'
22
              count = UInt(pred<maxbit:1>);
23
              esize = 8;
           when 'xx10'
24
25
              count = UInt(pred<maxbit:2>);
26
               esize = 16;
27
           when 'x100'
28
              count = UInt(pred<maxbit:3>);
29
               esize = 32;
           when '1000'
30
31
              count = UInt(pred<maxbit:4>);
32
               esize = 64;
33
34
      elements = (VL * 4) DIV esize;
35
   result = Zeros(PL*4);
```
```
36 constant integer psize = esize DIV 8;
37 for e = 0 to elements-1
38 bit pbit = if e < count then '1' else '0';
39 if invert then
40 pbit = NOT(pbit);
41 Elem[result, e, psize] = ZeroExtend(pbit, psize);
42
43 return result<width-1:0>;
```

E2.1.13 CurrentNSVL

```
1 // CurrentNSVL - non-assignment form
2
   3 // Non-Streaming VL
4
5
   integer CurrentNSVL
6
       integer v1;
7
8
       if PSTATE.EL == EL1 || (PSTATE.EL == EL0 && !IsInHost()
                                                             then
9
           vl = UInt(ZCR_EL1.LEN);
10
       if PSTATE.EL == EL2 || (PSTATE.EL == EL0 && J nHost() then
11
12
           vl = UInt(ZCR_EL2.LEN);
       elsif PSTATE.EL IN {EL0, EL1} && EL2Enabled() her
13
14
           vl = Min(vl, UInt(ZCR_EL2.LEN));
15
       if PSTATE.EL == EL3 then
16
17
           vl = UInt(ZCR_EL3.LEN);
18
       elsif HaveEL(EL3) then
19
           vl = Min(vl, UInt(ZCR_EL3 LEN)),
20
21
       vl = (vl + 1) * 128;
       vl = ImplementedSVEVect
22
                              Length(v.
23
24
       return vl;
```

E2.1.14 CurrentSVL

```
// CurrentSVL non-as ignment form
1
2
   // ==
3 // Stream' SVL
4
   inter . Curre .SVL
5
6
        nteger :
7
       it
8
             ATE.EL == EL1 || (PSTATE.EL == EL0 && !IsInHost()) then
9
              = UInt(SMCR_EL1.LEN);
10
       if PSTA1. EL == EL2 || (PSTATE.EL == EL0 && IsInHost()) then
11
           vl = UInt(SMCR_EL2.LEN);
12
13
       elsif PSTATE.EL IN {EL0, EL1} && EL2Enabled() then
14
           vl = Min(vl, UInt(SMCR_EL2.LEN));
15
16
       if PSTATE.EL == EL3 then
           vl = UInt(SMCR_EL3.LEN);
17
18
       elsif HaveEL(EL3) then
           vl = Min(vl, UInt(SMCR_EL3.LEN));
19
20
21
       vl = (vl + 1) * 128;
22
       vl = ImplementedSMEVectorLength(vl);
23
24
       return vl;
```

E2.1.15 CurrentVL

```
1 // CurrentVL - non-assignment form
3
  integer CurrentVL
4
5
    return if HaveSME() && PSTATE.SM == '1' then CurrentSVL else CurrentNSVL;
```

E2.1.16 EncodePredCount

```
1 // EncodePredCount()
 2
    3
4 bits(width) EncodePredCount(integer esize, integer elements,
5
                                   integer count_in, boolean invert_in, integer width)
        integer count = count_in;
6
7
        boolean invert = invert_in;
        constant integer PL = CurrentVL DIV 8;
8
9
        assert width == PL;
        assert esize IN {8, 16, 32, 64};
10
11
        assert count >=0 && count <= elements;</pre>
12
        bits(16) pred;
13
14
        if count == 0 then
15
             return Zeros(width);
16
17
        if invert then
18
            count = elements - count;
        elsif count == elements then
19
20
            count = 0;
21
             invert = TRUE;
22
        bit inv = (if invert then '1' se
23
24
        case esize of
25
             when 8 pred = inv count<13.</pre>
             when 16 pred = in count<12:0, '10';
when 32 pred = v : punt<11:0> '100';
when 64 pred = inv : c pt<10:0> : '1000';
26
27
28
29
30
```

return ZeroEx' .nd(pred, width)

E2.1.17 FPAdd ZA

```
// FPAd _ZAL
1
2
   // =
  // 1culate op1+op2 for SME2 ZA-targeting instructions.
3
4
5
  bits(N
            Add_ZA(bits(N) op1, bits(N) op2, FPCRType fpcr_in)
      FPCk pe fpcr = fpcr_in;
6
       boolea. fpexc = FALSE;
fpcr.DN '1';
7
                                    // Do not generate floating-point exceptions
8
                                   // Generate default NaN values
       return FPAdd(op1, op2, fpcr, fpexc);
9
```

E2.1.18 FPDot

```
1 // FPDot()
2 // ==
3
   // Calculates single-precision result of 2-way 16-bit floating-point dot-product
4
   // with a single rounding.
   // The 'fpcr' argument supplies the FPCR control bits and 'isbfloat16'
5
6
   // determines whether input operands are BFloat16 or half-precision type.
   // and 'fpexc' controls the generation of floating-point exceptions.
7
8
9
   bits(N) FPDot(bits(N DIV 2) op1_a, bits(N DIV 2) op1_b, bits(N DIV 2) op2_a,
        bits(N DIV 2) op2_b, FPCRType fpcr, boolean isbfloat16)
boolean fpexc = TRUE; // Generate floating-point exceptions
10
11
12
   return FPDot(op1_a, op1_b, op2_a, op2_b, fpcr, isbfloat16, fpexc);
```

```
13
    bits(N) FPDot(bits(N DIV 2) op1_a, bits(N DIV 2) op1_b, bits(N DIV 2) op2_a,
14
                    bits (N DIV 2) op2_b, FPCRType fpcr_in, boolean isbfloat16, boolean fpexc)
15
         FPCRType fpcr = fpcr_in;
16
17
18
         assert N == 32;
         bits(N) result;
19
20
         boolean done;
21
         fpcr.AHP = '0';
                                       // Ignore alternative half-precision option
22
         rounding = FPRoundingMode(fpcr);
23
24
         (type1_a,sign1_a,value1_a) = FPUnpackBase(op1_a, fpcr, fpexc, isbfloat16);
25
         (type1_b,sign1_b,value1_b) = FPUnpackBase(op1_b, fpcr, fpexc, isbfloat16);
         (type2_a, sign2_a, value2_a) = FPUnpackBase(op2_a, fpcr, fpexc, isbfloat16);
26
27
         (type2_b,sign2_b,value2_b) = FPUnpackBase(op2_b, fpcr, fpexc, isbfloat16);
28
        infl_a = (typel_a == FPType_Infinity); zerol_a = (typel_a vpe_Zero);
infl_b = (typel_b == FPType_Infinity); zerol_b = (typel == FP1, Zero);
inf2_a = (type2_a == FPType_Infinity); zero2_a = (type _a == FPType_Zero);
inf2_b = (type2_b == FPType_Infinity); zero2_b = (type _b == FPType_Zero);
29
30
31
32
33
                                                                            pe2_b,
34
         (done,result) = FPProcessNaNs4(type1_a, type1_b _ype2_a,
35
                                             op1_a, op1_b, < __a, op2_b, 1 r, pexc);
36
37
         if (((inf1_a && zero2_a) || (zero1_a && inf2_ )) &/
38
             ((inf1_b && zero2_b) || (zero1_b && inf2_b \ .nen
39
             result = FPDefaultNaN(fpcr, N);
             if fpexc then FPProcessExceptior .PExc_ valia
40
                                                                          er);
41
42
         if !done then
             // Determine sign and type procets all have of it does not cause an Invalid
43
44
              // Operation.
             signPa = sign1_a EOR s gn2_
signPb = sign1_b EOP ign2_b,
45
46
             infPa = infl_a || f2_a;
infPb = infl_b | 1 b;
zeroPa = zerol_ || ze 2_a;
47
48
49
50
             zeroPb = zer 1_b || zerc
                                           b;
51
              // Non S' N-generated Inval. Operation cases are multiplies of zero
52
             // by finity d additions of opposite-signed infinities.
invalit = / nfl_a && zero2_a) || (zero1_a && inf2_a) ||
53
54

    zero2 || (zero1_b && inf2_b) || (infPa && infPb && signPa !=

55
                  (inf.
                       ↔. rnPb)).
56
57
             if ir alidop t.
                  if rpe.. chen FPProcessException(FPExc_InvalidOp, fpcr);
58
59
60
               ther cases involving infinities produce an infinity of the same sign.
61
62
             els (infPa && signPa == '0') || (infPb && signPb == '0') then
                  result = FPInfinity('0', N);
63
             elsif (infPa && signPa == '1') || (infPb && signPb == '1') then
64
65
                  result = FPInfinity('1', N);
66
67
              // Cases where the result is exactly zero and its sign is not determined by the
68
             // rounding mode are additions of same-signed zeros.
69
             elsif zeroPa && zeroPb && signPa == signPb then
70
                  result = FPZero(signPa, N);
71
             \ensuremath{//} Otherwise calculate fused sum of products and round it.
72
73
             else
74
                  result_value = (value1_a * value2_a) + (value1_b * value2_b);
75
                  if result_value == 0.0 then // Sign of exact zero result depends on rounding
                       ∽mode
                       result_sign = if rounding == FPRounding_NEGINF then '1' else '0';
76
77
                       result = FPZero(result_sign, N);
78
                  else
79
                       result = FPRound(result_value, fpcr, rounding, fpexc, N);
```

80
81 return result;

E2.1.19 FPDotAdd

```
1 // FPDotAdd()
2
   // =======
3 // Half-precision 2-way dot-product and add to single-precision.
4
   bits(N) FPDotAdd(bits(N) addend, bits(N DIV 2) op1_a, bits(N DIV 2) op1_b,
5
                    bits(N DIV 2) op2_a, bits(N DIV 2) op2_b, FPCRType fpcr)
6
7
       assert N == 32;
8
9
       bits(N) prod;
10
      boolean isbfloat16 = FALSE;
       boolean fpexc = TRUE; // Generate floating-point excf
11
12
       prod = FPDot(op1_a, op1_b, op2_a, op2_b, fpcr, isbfloat , fpexc),
13
       result = FPAdd(addend, prod, fpcr, fpexc);
14
15
       return result;
```

E2.1.20 FPDotAdd_ZA

```
1 // FPDotAdd_ZA()
 2 // ==============
    // Half-precision 2-way dot-product and add to
                                                                         ingle-p.
3
                                                                                          sion
 4
    // for SME ZA-targeting instructions
 5

      6
      bits(N) FPDotAdd_ZA(bits(N) adder
      bits
      IV 2) op1_a, bits(N DIV 2) op1_b,

      7
      bits(N DIV 2)
      2_a, b
      s(N DIV 2) op2_b, FPCRType fpcr_in)

8
          FPCRType fpcr = fpcr_in;
9
          assert N == 32;
10
          bits(N) prod;
11
         boolean isbfloat16 = FALSE;
12
         boolean fpexc = ALSE; //
fper.DN = '1'
                                               // not generate floating-point exceptions
// Generate default NaN values
13
14
         prod = FPDc op1_a, p1_b, op2_a, op2_b, fpcr, isbfloat16, fpexc);
result = r dd(adc d, prod, fpcr, fpexc);
15
16
17
18
         retur sult,
```

E2.1.21 FPM^r .Add_7

```
1 // FPMu
            d_ZA()
2 // ======
              ____
  // Calculat addend + op1*op2 with a single rounding for SME ZA-targeting
3
4 // instructions.
5
   bits(N) FPMulAdd_ZA(bits(N) addend, bits(N) op1, bits(N) op2, FPCRType fpcr_in)
6
      FPCRType fpcr = fpcr_in;
7
       boolean fpexc = FALSE;
8
                                   // Do not generate floating-point exceptions
9
       fpcr.DN = '1';
                                  // Generate default NaN values
10
       return FPMulAdd(addend, op1, op2, fpcr, fpexc);
```

E2.1.22 FPMulAddH_ZA

8 fpcr.DN = '1'; // Generate default NaN values 9 return FPMulAddH(addend, op1, op2, fpcr, fpexc);

E2.1.23 FPProcessDenorms4

```
1 // FPProcessDenorms4()
2
  3 // Handles denormal input in case of single-precision or double-precision
4 // when using alternative floating-point mode.
5
6 FPProcessDenorms4(FPType type1, FPType type2, FPType type3, FPType type4, integer N,
       \hookrightarrow FPCRType fpcr)
7
       boolean altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
       if altfp && N != 16 && (type1 == FPType_Denormal || type2 == FPType_Denormal ||
8
9
           type3 == FPType_Denormal || type4 == FPType_Denormal) them
           FPProcessException(FPExc_InputDenorm, fpcr);
10
```

E2.1.24 FPProcessNaNs4

```
// FPProcessNaNs4()
 1
 2
    // ==
 3 // The boolean part of the return value says where a has been found and 4 // processed. The bits(N) part is only relevant in this and so plies the
 5
    // result of the operation.
 6
    11
    // The 'fpcr' argument supplies FPCR c _crol bi s.
// Status information is updated dir _ly in t > FPSR where appropriate.
// The 'fpexc' controls the generation _f f' _crms____ c exceptions.
 7
 8
 9
                                                                         c exceptions.
10
     (boolean, bits(N)) FPProcessNaNs4( Type t 21, FPType type2, FPType type3, FPType type4,
11
                                                 bi (N DIV ) op1, bits (N DIV 2) op2, bits (N DIV 2) op3,
bits DIV 2 op4, FPCRType fpcr, boolean fpexc)
12
13
14
15
          assert N == 32:
16
17
          bits(N) result.
18
          boolean done ·
          // The FPCF ff contr : does not affect these checks
if type1 = FPType NaN then
19
20
          done = 1 E: result > FPConvertNaN(FPProcessNaN(type1, op1, fpcr, fpexc), N);
elsif be2 == PType_S' + then
tone TRUE, rest = FPConvertNaN(FPProcessNaN(type2, op2, fpcr, fpexc), N);
21
22
23
           .if typ : == FP1 __SNaN then
done __TRUE; result = FPConvertNaN(FPProcessNaN(type3, op3, fpcr, fpexc), N);
24
25
             if per
                 ne = TRUE; result = FPConvertNaN(FPProcessNaN(type4, op4, fpcr, fpexc), N);
26
27
28
          elsi. ype1 == FPType_QNaN then
29
               do. = TRUE; result = FPConvertNaN(FPProcessNaN(type1, op1, fpcr, fpexc), N);
30
          elsif ty 2 == FPType_QNaN then
31
               done = TRUE; result = FPConvertNaN(FPProcessNaN(type2, op2, fpcr, fpexc), N);
32
          elsif type3 == FPType_QNaN then
33
              done = TRUE; result = FPConvertNaN(FPProcessNaN(type3, op3, fpcr, fpexc), N);
34
          elsif type4 == FPType_QNaN then
35
              done = TRUE; result = FPConvertNaN(FPProcessNaN(type4, op4, fpcr, fpexc), N);
36
          else
               done = FALSE; result = Zeros(N); // 'Don't care' result
37
38
39
          return (done, result);
```

E2.1.25 FPSub_ZA

```
5 bits(N) FPSub_ZA(bits(N) op1, bits(N) op2, FPCRType fpcr_in)
6 FPCRType fpcr = fpcr_in;
7 boolean fpexc = FALSE; // Do not generate floating-point exceptions
8 fpcr.DN = '1'; // Generate default NaN values
9 return FPSub(op1, op2, fpcr, fpexc);
```

E2.1.26 HaveEBF16

E2.1.27 HaveSME

E2.1.28 HaveSME2

E2.1.29 HaveSMEF64F64

E2.1.30 HaveSi 16164

E2.1.31 ImplementedSMEVectorLength

```
9
       assert 128 <= nbits && nbits <= 2048 && Align(nbits, 128) == nbits;
10
11
       // Search for a supported power-of-two VL less than or equal to nbits
12
       while nbits > 128 do
13
           if IsPow2(nbits) && SupportedPowerTwoSVL(nbits) then return nbits;
14
           nbits = nbits - 128;
15
16
       // Return the smallest supported power-of-two VL
17
       nbits = 128;
       while nbits < maxbits do
18
19
           if SupportedPowerTwoSVL(nbits) then return nbits;
20
           nbits = nbits \star 2;
21
22
       // The only option is maxbits
23
     return maxbits;
```

E2.1.32 InStreamingMode

E2.1.33 IsFullA64Enabled

```
1 // IsFullA64Enabled()
3 // Returns TRUE is full A64 is er bled 1
                                                 reaming mode and FALSE othersise.
4
5 boolean IsFullA64Enabled()
        if !HaveSMEFullA64() t<sup>y</sup> . return F. SE;
6
7
       // Check if full SV disa ed in SMCR_11
if PSTATE.EL IN 'ELO, EL1} !IsInHost() then
   // Check fv = SVE at ELO
   if SMCR_F ..FA64 == '0' the return FALSE;
8
9
10
11
12
       13
14
15
16
         Check | full . _ uisabled in SMCR_EL3
f HaveF (EL3) then
17
18
            19
20
        retu. TRUE;
21
```

E2.1.34 IsMerging

E2.1.35 IsOriginalSVEEnabled

```
4
   // exception level and FALSE otherwise.
5
 6
   boolean IsOriginalSVEEnabled(bits(2) el)
        boolean disabled:
7
8
        if ELUsingAArch32(el) then
9
            return FALSE;
10
11
        // Check if access disabled in CPACR_EL1
12
        if el IN {EL0, EL1} && !IsInHost() then
13
             // Check SVE at EL0/EL1
             case CPACR_EL1.ZEN of
14
15
                 when 'x0' disabled = TRUE;
                 when '01' disabled = el == EL0;
16
                 when '11' disabled = FALSE;
17
18
             if disabled then return FALSE;
19
20
        // Check if access disabled in CPTR_EL2
21
        if el IN {EL0, EL1, EL2} && EL2Enabled() then
22
             if HaveVirtHostExt() && HCR_EL2.E2H == '1' then
23
                 case CPTR_EL2.ZEN of
24
                     when 'x0' disabled = TRUE;
                     when '01' disabled = el == ELO && H<sup>c</sup>_EL2.TGE
when '11' disabled = FALSE;
25
26
27
                 if disabled then return FALSE;
28
             else
29
                 if CPTR_EL2.TZ == '1' then return FAL
30
31
        // Check if access disabled in CPTR
32
        if HaveEL(EL3) then
             if CPTR_EL3.EZ == '0' then re rn FAI
33
34
35
        return TRUE;
```

11:

E2.1.36 IsSMEEnabled

```
1
   // IsSMEEnabled()
 2
   // ================
   // Returns TRUE if access to SME i stionality is enabled at the target
3
4
   // exception le
                        and F'LSE otherwise.
5
   boolean IsSMEEn led its(2) el)
 6
        boolear disab ,
if - ... AArch (el)
7
8
                                / .en
9
            retur FALSE;
10
                           s disabled in CPACR_EL1
11
           Cher
12
            1 .N {ELO, EL1} && !IsInHost() then
13
                Check SME at EL0/EL1
14
                 CPACR_EL1.SMEN of
            Ċ.
                  en 'x0' disabled = TRUE;
15
                 when '01' disabled = el == EL0;
when '11' disabled = FALSE;
16
17
            if disabled then return FALSE;
18
19
20
        // Check if access disabled in CPTR_EL2
21
        if el IN {EL0, EL1, EL2} && EL2Enabled() then
22
            if HaveVirtHostExt() && HCR_EL2.E2H == '1' then
23
                 case CPTR_EL2.SMEN of
24
                     when 'x0' disabled = TRUE;
25
                     when '01' disabled = el == ELO && HCR_EL2.TGE == '1';
26
                     when '11' disabled = FALSE;
27
                 if disabled then return FALSE;
28
            else
29
                 if CPTR_EL2.TSM == '1' then return FALSE;
30
31
        // Check if access disabled in CPTR_EL3
        if HaveEL(EL3) then
32
            if CPTR_EL3.ESM == '0' then return FALSE;
33
```

34 35 return TRUE;

E2.1.37 IsSVEEnabled

```
1 // IsSVEEnabled()
2
  3 // Returns TRUE if access to SVE registers is enabled at the target exception
4 // level and FALSE otherwise.
6 boolean IsSVEEnabled(bits(2) el)
      if HaveSME() && PSTATE.SM == '1' then
7
8
          return IsSMEEnabled(el);
       elsif HaveSVE() then
9
10
          return IsOriginalSVEEnabled(el);
11
       else
12
          return FALSE;
```

E2.1.38 Lookup

1 **bits**(512) _ZTO;

E2.1.39 MaybeZeroSVEUppers

```
1 // MaybeZeroSVEUppers()
2 // ==
3
4 MaybeZeroSVEUppers(bits(2) target 1)
5
       boolean lower_enabled;
6
       if UInt(target_el) <= ot(PSTATE.L</pre>
                                              || !_sSVEEnabled(target_el) then
7
8
           return;
9
       if target_el == ...3 then
    if EL2Enab d() then
10
11
               lowe _enabled = IsFPEna. .ed(EL2);
12
13
            else
       14
15
16
17
18
19
            elsr
20
                iower___.oled = IsFPEnabled(EL0);
21
       els
22
            a srt target_el == EL1 && !ELUsingAArch32(EL1);
23
            low enabled = IsFPEnabled(EL0);
24
25
       if lower_enabled then
26
            constant integer VL = if IsSVEEnabled(PSTATE.EL) then CurrentVL else 128;
27
            constant integer PL = VL DIV 8;
28
            for n = 0 to 31
29
               if ConstrainUnpredictableBool(Unpredictable_SVEZEROUPPER) then
30
                    _Z[n] = ZeroExtend(_Z[n] < VL-1:0>, MAX_VL);
            for n = 0 to 15
31
32
               if ConstrainUnpredictableBool(Unpredictable_SVEZEROUPPER) then
33
                    _P[n] = ZeroExtend(_P[n]<PL-1:0>, MAX_PL);
34
            if ConstrainUnpredictableBool(Unpredictable_SVEZEROUPPER) then
                _FFR = ZeroExtend(_FFR<PL-1:0>, MAX_PL);
35
            if HaveSME() && PSTATE.ZA == '1' then
36
37
               constant integer SVL = CurrentSVL;
                constant integer accessiblevecs = SVL DIV 8;
38
39
                constant integer allvecs = MaxImplementedSVL() DIV 8;
40
41
                for n = 0 to accessiblevecs - 1
```

Copyright © 2022-2023 Arm Limited or its affiliates. All rights reserved. Non-confidential

42	if ConstrainUnpredictableBool(Unpredictable_SMEZEROUPPER)	then
43	<pre>_ZA[n] = ZeroExtend(_ZA[n]<svl-1:0>, MAX_VL);</svl-1:0></pre>	
44	for n = accessiblevecs to allvecs - 1	
45	if ConstrainUnpredictableBool(Unpredictable_SMEZEROUPPER)	then
46	_ZA[n] = Zeros(MAX_VL);	

E2.1.40 PredCountTest

```
1 // PredCountTest()
2
   // =====
3
4 bits(4) PredCountTest(integer elements, integer count, boolean invert)
5
      bit n, z, c, v;
       z = (if count == 0 then '1' else '0');
6
                                                          // none active
7
      if !invert then
                                                          // fj
         n = (if count != 0 then '1' else '0');
8
           c = (if count == elements then '0' else '1');
                                                          // f last a
9
                                                                         ive
10
       else
                                                          // rst active
11
         n = (if count == elements then '1' else '0');
           c = (if count != 0 then '0' else '1');
                                                          // N. last act
12
13
       v = '0';
14
15
       return n:z:c:v;
```

E2.1.41 ResetSMEState

E2.1.42 ResetSVEState

E2.1.43 SetPSTATE_SM

E2.1.44 SetPSTATE_SVCR

- 4 SetPSTATE_SVCR(bits(32) svcr)
 5 SetPSTATE_SM(svcr<0>);
- 6 SetPSTATE_ZA(svcr<1>);

E2.1.45 SetPSTATE_ZA

E2.1.46 SMEAccessTrap

```
1 // SMEAccessTrap()
   // ============
2
3 // Trapped access to SME registers due to CPACR_EL1, _TR_EL2, _r CPTR,
                                                                                   .3
4
5
   SMEAccessTrap(SMEExceptionType etype, bits(2) ta _____i
       bits(2) target_el = target_el_in;
 6
        assert UInt(target_el) >= UInt(PSTATE.EL);
7
8
        if target_el == EL0 then
            target_el = EL1;
 9
10
        boolean route_to_el2;
        route_to_el2 = PSTATE.EL == EL0 < target 1 == EL1 & EL2Enabled() & HCR_EL2.TGE ==
11
            \hookrightarrow '1';
12
13
        exception = ExceptionSyndrome(. reptic SMEAccessTrap);
14
        bits(64) preferred_excep* _n_ret > = Th. TnstrAddr(64);
15
        vect_offset = 0x0;
16
17
        case etype of
18
            when SMEExcertionType_A PssTrap
            except ...syndrome<2. = '000';
when SMEF ceptionType_Strea ng
    exc .tion.sy lrome<2:0> = '001';
when S Except nType_NotStreaming
19
20
21
22
            23
24
25
26
27
                  .ception.syndrome<2:0> = '100';
28
              ce_to_el2 then
29
        it
30
               ch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
31
        else
32
            AArc. J4.TakeException(target_el, exception, preferred_exception_return, vect_offset);
```

E2.1.47 System

E2.1.48 ZAhslice

```
6
       integer tiles = esize DIV 8;
7
        assert tile >= 0 && tile < tiles;</pre>
 8
        integer slices = CurrentSVL DIV esize;
        assert slice >= 0 && slice < slices;</pre>
9
10
11
        return ZAvector[tile + slice * tiles, width];
12
13
   // ZAhslice[] - assignment form
14
   // =
15
   ZAhslice[integer tile, integer esize, integer slice, integer width] = bits(width) value
16
17
        assert esize IN {8, 16, 32, 64, 128};
18
        integer tiles = esize DIV 8;
        assert tile >= 0 && tile < tiles;</pre>
19
20
        integer slices = CurrentSVL DIV esize;
21
        assert slice >= 0 && slice < slices;</pre>
22
23
     ZAvector[tile + slice * tiles, width] = value;
```

E2.1.49 ZAslice

```
1
   // ZAslice[] - non-assignment form
   // ==
2
3
                                                           an verti 1, integer slice, integer
4 bits(width) ZAslice[integer tile, integer esize, b.
        -→widthl
5
       bits(width) result;
6
7
       if vertical then
8
           result = ZAvslice[tile, erize,
                                                   width];
9
       else
10
            result = ZAhslice[til/_esi.
                                            slic
                                                   width];
11
12
       return result;
13
   // ZAslice[] - assignment form
14
15
   // ==
16
   ZAslice[integer .le, in eger esize, boolean vertical,
17
       integr slice, iteger width] = bits(width) value
if vertical hen
18
19
            ZAvslice, s, esize slice, width] = value;
20
21
        elsr
22
            ZAhsl :e[tile e ____e, slice, width] = value;
```

E2.1.50 ZAtile

```
3
4 bits(width) ZAtile[integer tile, integer esize, integer width]
      constant integer SVL = CurrentSVL;
5
6
      integer slices = SVL DIV esize;
      assert width == SVL * slices;
7
8
      bits(width) result;
9
10
      for slice = 0 to slices-1
11
          Elem[result, slice, SVL] = ZAhslice[tile, esize, slice, SVL];
12
13
      return result;
14
15
  // ZAtile[] - assignment form
16
   17
18
   ZAtile[integer tile, integer esize, integer width] = bits(width) value
19
      constant integer SVL = CurrentSVL;
20
      integer slices = SVL DIV esize;
```

Copyright © 2022-2023 Arm Limited or its affiliates. All rights reserved. Non-confidential

```
21 assert width == SVL * slices;
22
23 for slice = 0 to slices-1
24 ZAhslice[tile, esize, slice, SVL] = Elem[value, slice, SVL];
```

E2.1.51 ZAvector

```
1 // ZAvector[] - non-assignment form
2 // ==
3
4 bits(width) ZAvector[integer index, integer width]
5
       assert width == CurrentSVL;
       assert index >= 0 && index < (width DIV 8);</pre>
6
7
8
       return _ZA[index]<width-1:0>;
9
10 // ZAvector[] - assignment form
   11
12
13 ZAvector[integer index, integer width] = bits(width) lue
14
      assert width == CurrentSVL;
15
       assert index >= 0 && index < (width DIV 8);</pre>
16
17
       if ConstrainUnpredictableBool(Unpredictable_> ZEP PPER) then
           _ZA[index] = ZeroExtend(value, MAX_V<sup>I</sup>);
18
19
       else
20
           _ZA[index]<width-1:0> = value;
```

E2.1.52 ZAvslice

```
1 // ZAvslice[] - non-assignmer form
2
   // =====
 3
   bits(width) ZAvslice[ir _ger ]e, integer size, integer slice, integer width]
4
       integer slices = CurrentSVL TV esize;
bits(width) reg t;
5
6
7
        for s = 0 + slices-
bits(v. 'th) hs ce = ZAhslice[tile, esize, s, width];
8
9
10
11
        ret .n r ult;
12
13
   // vslice[ - assignment form
// = ----
14
15
16
17
   ZAvslice 'teger tile, integer esize, integer slice, integer width] = bits(width) value
18
        intege. lices = CurrentSVL DIV esize;
19
        for s = 0 to slices-1
20
21
            bits(width) hslice = ZAhslice[tile, esize, s, width];
22
            Elem[hslice, slice, esize] = Elem[value, s, esize];
23
            ZAhslice[tile, esize, s, width] = hslice;
```

E2.1.53 ZT0

10		
11	<pre>ZT0[integer width] = bits(width)</pre>	value
12	assert width == 512;	
13	_ZTO <width-1:0> = value;</width-1:0>	



Chapter E3 System registers affected by SME

This section provides the full inform i on for System registers added or modified by SME or SME2.

This content is from 2022-12 version of Arm[®] Architecture Registers, for A-profile architecture [2], which contains the definite version of the register information.

E3.1 SME-Specific System registers

System registers that are added to support SME architecture.

E3.1.1 ID_AA64SMFR0_EL1, SME Feature ID register 0

The ID_AA64SMFR0_EL1 characteristics are:

Purpose

Provides information about the implemented features of the AArch64 Scalable Matrix Extension.

The fields in this register do not follow the standard ID scheme. See Alternative ID scheme used for $ID_AA64SMFR0_EL1$.

Configuration

Prior to the introduction of the features described by this register, this register was unnamed and reserved, RES0 from EL1, EL2, and EL3.

Attributes

ID_AA64SMFR0_EL1 is a 64-bit register.

Field descriptions

The ID_AA64SMFR0_EL1 bit assignments are:



FA64, bit [63]

Indicates support for encution of the 1 A64 instruction set when the PE is in Streaming SVE mode. Defined values are:

FA64	Meaning
0d0	Only those A64 instructions defined as being legal can be executed in Streaming SVE mode.
0b1	All implemented A64 instructions are legal for execution in Streaming SVE mode, when enabled by SMCR_EL1.FA64, SMCR_EL2.FA64, and SMCR_EL3.FA64.
	FA64 0b0 0b1

FEAT_SME_FA64 implements the functionality identified by the value 0b1.

Bits [62:60]

Reserved, RESO.

SMEver, bits [59:56]

When ID_AA64PFR1_EL1.SME != 0b0000:

Indicates support for SME instructions when FEAT_SME is implemented. Defined values are:

SMEver	Meaning
00000	The mandatory SME instructions are implemented.
0b0001	As 0b0000, and adds the mandatory SME2 instructions.

All other values are reserved.

If FEAT_SME is implemented and FEAT_SME2 is not implemented, the only permitted value is 0b0000.

If FEAT_SME2 is implemented the only permitted value is 0b0001.

Otherwise:

res0

116l64, bits [55:52]

Indicates SME support for instructions that accumulate int ________4-bit int _______, er elemen______ in the ZA array. Defined values are:

1	1161	lv v .ıg
	0.000	Istructions that accumulate into 64-bit integer elements in the ZA array are not implemented.
	0. 11	The variants of the ADDHA, ADDVA, SMOPA, SMOPS, SUMOPA, SUMOPS, UMOPA, UMOPS, USMOPA, and USMOPS instructions that accumulate into 64-bit integer tiles are implemented. When FEAT_SME2 is implemented, the variants of the ADD, ADDA, SDOT, SMLALL, SMLSLL, SUB, SUBA, SVDOT, UDOT, UMLALL, UMLSLL, and UVDOT instructions that accumulate into 64-bit integer elements in ZA array vectors are implemented.

All other v. es are reserved.

FEAT_SME_1.oI64 implements the functionality identified by the value 0b1111.

The only permitted values are 0b0000 and 0b1111.

Bits [51:49]

Reserved, RESO.

F64F64, bit [48]

Indicates SME support for instructions that accumulate into FP64 double-precision floating-point elements in the ZA array. Defined values are:

F64F64	Meaning
000	Instructions that accumulate into double-precision floating-point elements in the ZA array are not implemented.
0b1	The variants of the FMOPA and FMOPS instructions that accumulate into double-precision tiles are implemented. When FEAT_SME2 is implemented, the variants of the FADD, FMLA, FMLS, and FSUB instructions that accumulate into double-precision elements in ZA array vectors are imple

FEAT_SME_F64F64 implements the functionality identified by 'he val, 0b1.

116132, bits [47:44]

Indicates SME2 support for instructions that accumulate bit ther products into 32-bit integer tiles. Defined values are:

<u></u>	Mening
	Instructions that accumulate 16-bit outer products into 32-bit integer tiles are not implemented.
b0101	The SMOPA (2-way), SMOPS (2-way), UMOPA (2-way), and UMOPS (2-way) instructions that accumulate 16-bit outer products into 32-bit integer tiles are implemented.

All other are re rved.

If FF __SME is implemented, the only permitted value is 0b0101. Otherwise, the only permitted value is 0b0 '0.

Bits [4、 /]

Reserved, RE

18132, bits [39:36]

Indicates SME support for instructions that accumulate 8-bit integer outer products into 32-bit integer tiles. Defined values are:

18132	Meaning
060000	Instructions that accumulate 8-bit outer products into 32-bit tiles are not implemented.
0b1111	The SMOPA, SMOPS, SUMOPA, SUMOPS, UMOPA, UMOPS, USMOPA, and USMOPS instructions that accumulate 8-bit outer products into 32-bit tiles are implemented.

All other values are reserved.

If FEAT_SME is implemented, the only permitted value is 0b1111.

F16F32, bit [35]

Indicates SME support for instructions that accumulate FP16 half-precision floating-point outer products into FP32 single-precision floating-point tiles. Defined values are:

F16F32	Meaning	
060	Instructions that accumulate half-precision outer products into single-precision tiles are not implement	
0b1	The F' \rightarrow PA and μ 'OPS instructions that acct. Plate half-prection outer products into ingle- μ cision tiles e implemented.	

If FEAT_SME is implemented, the only permitted value . ^{\bl.}

B16F32, bit [34]

Indicates SME support for instructions that .ccumula BFloat Suter products into FP32 single-precision floating-point tiles. Defined values are:

L TF32	Meaning
^о ь0	Instructions that accumulate BFloat16 outer products into single-precision tiles are not implemented.
0b1	The BFMOPA and BFMOPS instructions that accumulate BFloat16 outer products into single-precision tiles are implemented.

If $F' T_SMF$, implemented, the only permitted value is 0b1.

BI3215. 1 . [33]

Indicates SN. support for instructions that accumulate thirty-two 1-bit binary outer products into 32-bit integer tiles. Defined values are:

BI32I32	Meaning
060	Instructions that accumulate 1-bit binary outer products into 32-bit integer tiles are not implemented.
0b1	The BMOPA and BMOPS instructions that accumulate 1-bit binary outer products into 32-bit integer tiles are implemented.

If FEAT_SME2 is implemented, the only permitted value is 0b1. Otherwise, the only permitted value is 0b0.

F32F32, bit [32]

Indicates SME support for instructions that accumulate FP32 single-precision floating-point outer products into single-precision floating-point tiles. Defined values are:

F32F32	Meaning	
0b0	Instructions that accumulate single-precision outer products into single-precision tiles are no implemented.	
0b1	The FMOP ^A and FMOPS instructions that accume ¹ le sing ¹ precision tile. re implemented.	

If FEAT_SME is implemented, the only permitted value is 0[°].

Bits [31:0]

Reserved, RESO.

Accessing ID_AA64SMFR0_EL1

This register is read-only and can be accessed fix $\exists L1$ and higher.

This register is only accessible from the Ar. h64 sta

Accesses to this register use t¹ 'lowing encours in the System register encoding space:

MRS <Xt>, ID_AA64SMFr40_EL1

	0q <i>ı</i>	op1	CRn	CRm	op2		
	0b1	0b000	0b0000	0b0100	0b101		
1	if P. TE						
2	it eatureImplemen	ted(FEAT_IDST) the	en				
3	. EL2Enabled()	&& HCR_EL2.TGE ==	'1' then				
4	Arch64.Syst	emAccessTrap(EL2,	0x18);				
5	else						
6	AArch64.Syst	emAccessTrap(EL1,	0x18);				
7	else						
8	UNDEFINED;						
9	elsif PSTATE.EL == EL1 t	hen					
10	if EL2Enabled() && H	CR_EL2.TID3 == '1'	' then				
11	AArch64.SystemAc	cessTrap(EL2, 0x18	3);				
12	else						
13	$X[t, 64] = ID_AA$	64SMFR0_EL1;					
14	elsif PSTATE.EL == EL2 then						
15	$X[t, 64] = ID_AA64SM$	FR0_EL1;					
16	elsif PSTATE.EL == EL3 t	hen					
17	$X[t, 64] = ID_AA64SM$	FR0_EL1;					

E3.1.2 MPAMSM_EL1, MPAM Streaming Mode Register

The MPAMSM_EL1 characteristics are:

Purpose

Holds information to generate MPAM labels for memory requests that are:

- Issued due to the execution of SME load and store instructions.
- Issued when the PE is in Streaming SVE mode due to the execution of SVE and SIMD&FP load and store instructions and SVE prefetch instructions.

If an implementation uses a shared SMCU, then the MPAM labels in this register have precedence over the labels in MPAM0_EL1, MPAM1_EL1, MPAM2_EL2, and MPAM3_EL3.

If an implementation includes an SMCU that is not shared with other Point is IMPLEMENTATION DEFINED whether the MPAM labels in this register have precedent over the bels in MPAM0_EL1, MPAM1_EL1, MPAM2_EL2, and MPAM3_EL3.

The MPAM labels in this register are only used if MPAM¹ 3L1.M MEN is 1

For memory requests issued from EL0, the MPAM PA⁺ 1D in this regis. is atual and mapped into a physical PARTID when all of the following are true

- EL2 is implemented and enabled in the current Sec y state, ar ACR_EL2.{E2H, TGE} is not {1, 1}.
- The MPAM virtualization option is inplement d and Mi iHCR_EL2.EL0_VPMEN is 1.

For memory requests issued from EL1, the MPA reasonable into a physical PARTID when all of the viowing true:

- EL2 is implemented an __nabled . the cu. t Security state.
- The MPAM virtualizes on option is not emented and MPAMHCR_EL2.EL1_VPMEN is 1.

Configuration

This register is ______sent only when. "EAT_MPAM is implemented and FEAT_SME is implemented. Otherwise, d; ______ct accesses to MPAM_M_EL1 are UNDEFINED.

Attributes

MP[^] _EL1 a 64-bⁱ gister.

Fig descr stions

The MP. SM_EL1 bit assignments are:



Bits [63:48]

Reserved, RESO.

PMG_D, bits [47:40]

Performance monitoring group property for PARTID_D.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [39:32]

Reserved, RESO.

PARTID_D, bits [31:16]

Partition ID for requests issued due to the execution at any Exception level of SME load and store instructions and, when the PE is in Streaming SVE mode, SVE and SIMD&FP load and store instructions and SVE prefetch instructions.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [15:0]

Reserved, RESO.

Accessing MPAMSM_EL1

None of the fields in this register are permitted to be cache in a TLF

Accesses to this register use the following encodings in the _st__register__coding space:

MRS <Xt>, MPAMSM_EL1

op0	0r 1	CRn	CRm	op2	
0b11	06000	0ь1010	0b0101	0b011	

```
1
    if PSTATE.EL == EL0 +hen
2
        UNDEFINED;
 3
    elsif PSTATE.EL
                         EL1 then
        if HaveEL(F ) && MF 13_EL3.TRAPLOWER ==
    if Hal d() &F _DSCR.SDD == '1' then
 4
                                                        '1' then
5
6
                  UNDL
7
8
                     cch64. ster ccessTrap(EL3, 0x18);
            if EL2 labled()
9
                                MPAM2_EL2.EnMPAMSM == '0' then
             AArc',4_SystemAccessTrap(EL2, 0x18);
10
11
    c, 64] = MPAMSM_EL1;
elsif PS TE.EL == EL2 then
12
13
14
         if Hav
                   (EL3) && MPAM3_EL3.TRAPLOWER == '1' then
15
             if h. ted() && EDSCR.SDD == '1' then
16
                  UNDEFINED;
17
             else
18
                  AArch64.SystemAccessTrap(EL3, 0x18);
19
         else
20
             X[t, 64] = MPAMSM\_EL1;
21
    elsif PSTATE.EL == EL3 then
22
        X[t, 64] = MPAMSM_EL1;
```

MSR MPAMSM_EL1, <Xt>

ор0	op1	CRn	CRm	op2
0b11	0b000	0b1010	0b0101	0b011

```
if PSTATE.EL == EL0 then
1
2
       UNDEFINED;
3
   elsif PSTATE.EL == EL1 then
       if HaveEL(EL3) && MPAM3_EL3.TRAPLOWER == '1' then
4
5
            if Halted() && EDSCR.SDD == '1' then
6
               UNDEFINED;
7
            else
8
                AArch64.SystemAccessTrap(EL3, 0x18);
9
       elsif EL2Enabled() && MPAM2_EL2.EnMPAMSM == '0' then
10
           AArch64.SystemAccessTrap(EL2, 0x18);
11
       else
12
           MPAMSM\_EL1 = X[t, 64];
   elsif PSTATE.EL == EL2 then
13
14
       if HaveEL(EL3) && MPAM3_EL3.TRAPLOWER == '1' then
            if Halted() && EDSCR.SDD == '1' then
15
16
                UNDEFINED;
17
            else
18
                AArch64.SystemAccessTrap(EL3, 0x18);
19
       else
20
           MPAMSM\_EL1 = X[t, 64];
21
   elsif PSTATE.EL == EL3 then
22
    MPAMSM\_EL1 = X[t, 64];
```

E3.1.3 SMCR_EL1, SME Control Register (EL1)

The SMCR_EL1 characteristics are:

Purpose

This register controls aspects of Streaming SVE that are visible at Exception levels EL1 and EL0.

Configuration

This register has no effect if the PE is not in Streaming SVE mode.

When HCR_EL2.{E2H, TGE} == $\{1, 1\}$ and EL2 is enabled in the current Security state, this register has no effect on execution at EL0 and EL1.

This register is present only when FEAT_SME is implemented. Otherwise, direct accesses to SMCR_EL1 are UNDEFINED.

Attributes

SMCR_EL1 is a 64-bit register.

Field descriptions

The SMCR_EL1 bit assignments are:



Bits [63:32]

Reserved, RESO.

FA64, bit [31]

When FF ME_ 54 is im mented:

Contr s wheth execution an A64 instruction is considered legal when the PE is in Streaming SVE mode.

FA64	Meaning
0b0	This control does not cause any instruction to be treated as legal in Streaming SVE mode.
0b1	This control causes all implemented A64 instructions to be treated as legal in Streaming SVE mode at EL1 and EL0, if they are treated as legal at more privileged Exception levels in the current Security state.

Arm recommends that portable SME software should not rely on this optional feature, and that operating systems should provide a means to test for compliance with this recommendation.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

res0

EZT0, bit [30]

When FEAT_SME2 is implemented:

Traps execution at EL1 and EL0 of the LDR, LUTI2, LUTI4, MOVT, STR, and ZERO instructions that access the ZT0 register to EL1, or to EL2 when EL2 is implemented and enabled in the current Security state and HCR_EL2.TGE is 1.

The exception is reported using ESR_EL1.EC or ESR_EL2.EC value 0x1D, with an ISS code of 0x0000004, at a lower priority than a trap due to PSTATE.SM or PSTATE.ZA.



Changes to this field only affect whether instructions the access Z'_1 are trapped. They do not affect the contents of ZT0, which remain valid so long as PSTA. ZA is

The reset behavior of this field is:

• On a Warm reset, this field sets to an phitecu. Ily UNKNOWN value.

Otherwise:

res0

Bits [29:9]

Reserved, RESO.

Bits [8'

Res. ed, R/

LEN, br. 3:0]

Requests an E. ctive Streaming SVE vector length (SVL) at EL1 of (LEN+1)*128 bits. This field also defines the Effective Streaming SVE vector length at EL0 when EL2 is not implemented, or EL2 is not enabled in the current Security state, or HCR_EL2.{E2H,TGE} is not {1,1}.

The Streaming SVE vector length can be any power of two from 128 bits to 2048 bits inclusive. An implementation can support any subset of the architecturally permitted lengths.

When the PE is in Streaming SVE mode, the Effective SVE vector length (VL) is equal to SVL.

When FEAT_SVE is implemented, and the PE is not in Streaming SVE mode, VL is equal to the Effective Non-streaming SVE vector length. See ZCR_EL1.

For all purposes other than returning the result of a direct read of SMCR_EL1, the PE selects the Effective Streaming SVE vector length by performing checks in the following order:

1. If the requested length is less than the minimum implemented Streaming SVE vector length, then the Effective length is the minimum implemented Streaming SVE vector length.

- 2. If EL2 is implemented and enabled in the current Security state, and the requested length is greater than the Effective length at EL2, then the Effective length at EL2 is used.
- 3. If EL3 is implemented and the requested length is greater than the Effective length at EL3, then the Effective length at EL3 is used.
- 4. Otherwise, the Effective length is the highest supported Streaming SVE vector length that is less than or equal to the requested length.

An indirect read of SMCR_EL1.LEN appears to occur in program order relative to a direct write of the same register, without the need for explicit synchronization.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing SMCR_EL1

When HCR_EL2.E2H is 1, without explicit synchronization, access from EL3 using the mnemonic SMCR_EL1 or SMCR_EL12 are not guaranteed to be ordered with respect to accesses the mnemonic.

Accesses to this register use the following encodings in the system register encoding space:

MRS <Xt>, SMCR_EL1

op0	op1	CRn	CRm	op2	
0b11	05 70	0b0001	0b0010	0b110	

```
if PSTATE.EL == ELO then
 1
2
        UNDEFINED;
3
    elsif PSTATE.EL == EL1 chen
4
        if Halted() && ' veEL(EL3) &.
                                        CDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3
            \hookrightarrowtrap pr rity when SDD =
                                          '1'" && CPTR_EL3.ESM == '0' then
 5
            UNDEFIN
        elsif CPA EL1.SM == 'x0' then
AArch64 'st AccessTr p(EL1, 0x1D);
 6
 7
 8
        elsif "'2Enab. () && HC _EL2.E2H == '0' && CPTR_EL2.TSM == '1' then
              ALC 4.Sys ACCF (rap(EL2, 0x1D);
 9
        e' if EL2 labled(, ' HCR_EL2.E2H == '1'
10
                                                    && CPTR_EL2.SMEN == 'x0' then
            AArc<sup>1</sup> 4.SystemA cessTrap(EL2, 0x1D);
11
12
          sif⊦
                         && CPTR_EL3.ESM == '0'
                                                    then
13
               Halted() && EDSCR.SDD == '1' then
14
                UNDEFINED;
15
            e.
                 rch64.SystemAccessTrap(EL3, 0x1D);
16
17
        elsif EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '111' then
18
            X[t, 64] = NVMem[0x1F0];
19
        else
20
            X[t, 64] = SMCR\_EL1;
21
    elsif PSTATE.EL == EL2 then
        if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3
22
            →trap priority when SDD == '1'" && CPTR_EL3.ESM == '0' then
23
            UNDEFINED;
        elsif HCR_EL2.E2H == '0' && CPTR_EL2.TSM == '1' then
24
25
            AArch64.SystemAccessTrap(EL2, 0x1D);
        elsif HCR_EL2.E2H == '1' && CPTR_EL2.SMEN == 'x0' then
26
27
            AArch64.SystemAccessTrap(EL2, 0x1D);
28
        elsif HaveEL(EL3) && CPTR_EL3.ESM == '0' then
29
            if Halted() && EDSCR.SDD == '1' then
30
                UNDEFINED;
31
            else
32
                AArch64.SystemAccessTrap(EL3, 0x1D);
33
        elsif HCR_EL2.E2H == '1' then
```

MSR SMCR_EL1, <Xt>

		op0	op1	CRn	CRm	op2
		0b11	0b000	0b0001	210	0b110
1	if PSTATE.EL =:	= ELO then				
2	ONDEFINED;	T FT1 +b	en			
4	if Haltod() EE Havefi	(FI3) 22 FDSCR SD	$D == \frac{1}{2} \frac{1}{2}$	IN THE STATT	N DEFINED "FL3
т	unaitea(⊖tran	priority w	then SDD == '1'" &	בי באר רפט ג	= 101 then	
5	UNDEFT	NED:	11011 0000 1 4	• 0111 10.110		
6	elsif CPAC	R EL1.SMEN	== 'x0' then			
7	AArch6	4.SystemAcc	essTrap(EL1, 0x'	(j		
8	elsif EL2E:	nabled() &&	HCR_EL2.E2H	0' && CPTR_EL	A == '1' then	
9	AArch6	4.SystemAcc	essTrap(EL2, x1D);		
10	elsif EL2E:	nabled() &&	HCR_EL2.E2H '	1'. . 2.SI	MEN == $' \times 0'$ then	
11	AArch6	4.SystemAcc	essTrap 72, Ox			
12	elsif Have	EL(EL3) &&	CPTR_FL3. M ==	then		
13	if Hal	ted() && ED	SCR JD == 'th	er.		
14	UN	DEFINED;				
15	else			1.5.)		
10	AA	rcn64.Syr :	MACC SITAP(ELS,	XID);	~~	
17	UVMom	nadieu) œœ Oviec – vi	+ 6/1.	, NV > 1111 CH	en	
19	else	OXII) - X[C, 04],			
20	SMCR F	y = x[t]	11:			
21	elsif PSTATE.L	== EL ² .n	en			
22	<pre>if Halted(</pre>) F eEL	(EL3) S& EDSCR.SD	D == '1' && bool	ean IMPLEMENTATIO	ON_DEFINED "EL3
	gr 2	pr. ity w	he odd == '1'" &	& CPTR_EL3.ESM =	= '0' then	
23	JNDE.	VED;				
24	if HCR,	L2.E2H	0' && CPTR_EL2.T	SM == '1' then		
25	AArc',	4 SvstemAcc	essTrap(EL2, 0x1D);		
26	if	==	'1' && CPTR_EL2.SI	MEN == $' \times 0'$ then		
27		4.SystemAcc	essTrap(EL2, 0x1D);		
28	elsi. 'ave.	EL(EL3) &&	CPTR_EL3.ESM == '	0' then		
29	11 11	ted() && ED Deeined.	SCR.SDD == '1' th	en		
30	olco	DEFINED;				
32	erse 22	rch64 Sveta	maccessTran(FL3	0v1D).		
33	elsif HCR	EL2.E2H ==	'1' then	OXID),		
34	SMCR E	L2 = X[t, 6]	41;			
35	else					
36	SMCR_E	L1 = X[t, 6]	4];			
37	elsif PSTATE.E	L == EL3 th	en			
38	if CPTR_EL	3.ESM == '0	' then			
39	AArch6	4.SystemAcc	essTrap(EL3, 0x1D);		
40	else					
41	SMCR_E	L1 = X[t, 6]	4];			

MRS <Xt>, SMCR_EL12

ор0	op1	CRn	CRm	op2
0b11	0b101	0b0001	0b0010	0b110



MSR SMCR_EL. <X

4

	op0	op1	CRn	CRm	op2	
V	0b11	0b101	0b0001	0b0010	0b110	

```
1
   if PSTATE.EL == ELO then
2
      UNDEFINED;
3
   elsif PSTATE.EL == EL1 then
4
       if EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '101' then
5
          NVMem[0x1F0] = X[t, 64];
       elsif EL2Enabled() && HCR_EL2.NV == '1' then
6
7
          AArch64.SystemAccessTrap(EL2, 0x18);
8
       else
9
          UNDEFINED;
10
   elsif PSTATE.EL == EL2 then
      if HCR_EL2.E2H == '1' then
11
          if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED
12
              13
              UNDEFINED;
14
          elsif CPTR_EL2.SMEN == 'x0' then
15
             AArch64.SystemAccessTrap(EL2, 0x1D);
16
          elsif HaveEL(EL3) && CPTR_EL3.ESM == '0' then
17
              if Halted() && EDSCR.SDD == '1' then
```

Copyright © 2022-2023 Arm Limited or its affiliates. All rights reserved. Non-confidential

18 UNDEFINED; 19 else 20 AArch64.SystemAccessTrap(EL3, 0x1D); 21 else 22 $SMCR_EL1 = X[t, 64];$ 23 else UNDEFINED; 24 25 elsif PSTATE.EL == EL3 then if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then
 if CPTR_EL3.ESM == '0' then 26 27 28 AArch64.SystemAccessTrap(EL3, 0x1D); 29 else 30 $SMCR_EL1 = X[t, 64];$ 31 else 32 UNDEFINED;



E3.1.4 SMCR_EL2, SME Control Register (EL2)

The SMCR_EL2 characteristics are:

Purpose

This register controls aspects of Streaming SVE that are visible at Exception levels EL2, EL1, and EL0.

Configuration

This register has no effect if the PE is not in Streaming SVE mode, or if EL2 is not enabled in the current Security state.

If EL2 is not implemented, this register is RESO from EL3.

This register is present only when FEAT_SME is implemented. Otherwise, direct accesses to SMCR_EL2 are UNDEFINED.

Attributes

SMCR_EL2 is a 64-bit register.

Field descriptions

The SMCR_EL2 bit assignments are:



Bits [63:32]

Reserved, RESO.

FA64, bit [31]

When FF ME_ 54 is im mented:

Contr s wheth execution an A64 instruction is considered legal when the PE is in Streaming SVE mode.

FA64	Meaning
060	This control does not cause any instruction to be treated as legal in Streaming SVE mode.
0b1	This control causes all implemented A64 instructions to be treated as legal in Streaming SVE mode at EL2, if they are treated as legal at EL3.

Arm recommends that portable SME software should not rely on this optional feature, and that operating systems should provide a means to test for compliance with this recommendation.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

res0

EZT0, bit [30]

When FEAT_SME2 is implemented:

Traps execution at EL2, EL1, and EL0 of the LDR, LUTI2, LUTI4, MOVT, STR, and ZERO instructions that access the ZT0 register to EL2, when EL2 is enabled in the current Security state.

The exception is reported using ESR_EL2.EC value 0x1D, with an ISS code of 0x0000004, at a lower priority than a trap due to PSTATE.SM or PSTATE.ZA.

EZT0	Meaning
0b0	This co .ol caus, execution of these instructions at EL2,,1, and EL0 to be trapped.
0b1	This could does not a use execution of any instruction be traded.

Changes to this field only affect whether instructions that $a_{x} = 10$ are trajed. They do not affect the contents of ZTO, which remain valid so long as PSTATE.Z

The reset behavior of this field is:

ΨI.

• On a Warm reset, this field resets to an a ite arally Una town value.

Otherwise:

res0

Bits [29:9]

Reserved, RESO.

Bits [8:4]

Reserved

LEN Jits [3:0

Reques \pm Effective Streaming SVE vector length (SVL) at EL2 of (LEN+1)*128 bits. This field also defines the Effective \pm aming SVE vector length at EL0 when EL2 is implemented and enabled in the current Security state, and HCR_EL \pm [E2H,TGE] is {1,1}.

The Streaming SVE vector length can be any power of two from 128 bits to 2048 bits inclusive. An implementation can support any subset of the architecturally permitted lengths.

When the PE is in Streaming SVE mode, the Effective SVE vector length (VL) is equal to SVL.

When FEAT_SVE is implemented, and the PE is not in Streaming SVE mode, VL is equal to the Effective Non-streaming SVE vector length. See ZCR_EL2.

For all purposes other than returning the result of a direct read of SMCR_EL2, the PE selects the Effective Streaming SVE vector length by performing checks in the following order:

- 1. If the requested length is less than the minimum implemented Streaming SVE vector length, then the Effective length is the minimum implemented Streaming SVE vector length.
- 2. If EL3 is implemented and the requested length is greater than the Effective length at EL3, then the Effective length at EL3 is used.

3. Otherwise, the Effective length is the highest supported Streaming SVE vector length that is less than or equal to the requested length.

An indirect read of SMCR_EL2.LEN appears to occur in program order relative to a direct write of the same register, without the need for explicit synchronization.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing SMCR_EL2

When HCR_EL2.E2H is 1, without explicit synchronization, access from EL2 using the mnemonic SMCR_EL2 or SMCR_EL1 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Accesses to this register use the following encodings in the System register encoding space:

MRS <Xt>, SMCR_EL2

		op0	op1	C .	L P	op2
		0b11	0b100	01 71	`b0010	0b110
1	if PSTATE.EL =	= ELO then				
2	UNDEFINED;					
3	elsif PSTATE.E	L == EL1 th	en			
4	if EL2Enab	led() && HCI	R_EL2 NV '1' L	n		
5	AArch6	4.SystemAcc	ess .p(EL2_ ^x18) ,		
6	else					
7	UNDEF'I	NED;				
8	elsif PSTATE.E	L == EL2	en			
9	if Halted() && HaveEL	(EL3) EDSCR.SDI	D == '1' && bool	ean IMPLEMENTATIC	N_DEFINED "EL3
10	⇔trap	pric ty w	hen SDD '1'" &	& CPTR_EL3.ESM =	= '0' then	
10	UNDEF 1	NEL				
11	elsif HCR_	FE2H ==	0' && CPTR_EL2.T	SM == '1' then		
12	AArcht	Syster .co	essirap(ELZ, UXID);		
13	eisii HCK_		CPIR_ELZ.SI	MEN == 'XU' then		
14		4.5) EMACCO	C D ET3 ECM !	i 1 +hon		
15	if Ha	od() c	CP SDD 11! +h			
17		DEEINED.	Der.SDD 1 Ch	e11		
18	el					
19	AA	rch64 System	MAccessTran(EL3.	0x1D).		
20	else	101101.090000		onid),		
21	X) 6	4] = SMCR E	L2;			
22	elsif PSTATE	L == EL3 th	en			
23	if CPTR EL	3.ESM == '0	' then			
24	AArch6	4.SystemAcc	essTrap(EL3, 0x1D);		
25	else	-	± · · ·			
26	X[t, 6	4] = SMCR_E	L2;			

MSR SMCR_EL2, <Xt>

ор0	op1	CRn	CRm	op2
0b11	0b100	0b0001	0b0010	0b110

1 if PSTATE.EL == ELO then

2 UNDEFINED;

```
elsif PSTATE.EL == EL1 then
3
       if EL2Enabled() && HCR_EL2.NV == '1' then
4
5
           AArch64.SystemAccessTrap(EL2, 0x18);
6
       else
7
           UNDEFINED;
8
   elsif PSTATE.EL == EL2 then
       if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3
9
           10
           UNDEFINED:
       elsif HCR_EL2.E2H == '0' && CPTR_EL2.TSM == '1' then
11
12
           AArch64.SystemAccessTrap(EL2, 0x1D);
       elsif HCR_EL2.E2H == '1' && CPTR_EL2.SMEN == 'x0' then
13
14
           AArch64.SystemAccessTrap(EL2, 0x1D);
       elsif HaveEL(EL3) && CPTR_EL3.ESM == '0' then
15
           if Halted() && EDSCR.SDD == '1' then
16
17
               UNDEFINED;
18
           else
19
              AArch64.SystemAccessTrap(EL3, 0x1D);
20
       else
           SMCR\_EL2 = X[t, 64];
21
22
   elsif PSTATE.EL == EL3 then
23
       if CPTR_EL3.ESM == '0' then
24
          AArch64.SystemAccessTrap(EL3, 0x1D);
25
       else
26
         SMCR\_EL2 = X[t, 64];
```

MRS <Xt>, SMCR_EL1

op0	01 1	CRn	CRm	op2	
0b11	0b00v	060001	0b0010	0b110	

```
if PSTATE.EL == ELO then
 1
2
        UNDEFINED:
3
    elsif PSTATE.EL = LL1 then
4
        if Halted() & HaveF'(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3
             ⇔tra_ priorit when SDD == '1'" && CPTR_EL3.ESM == '0' then
 5
             UNDEFIN.
        UNDEFIN.

elsif CPACR_E. SMEN == .0' then

Arc 4.Sys PACCE frap(EL1, 0x1D);

e'if EL2 labled(, ' HCR_EL2.E2H == '0' && CPTR_EL2.TSM == '1' then
6
7
 8
            AArc<sup>1</sup> 4.SystemA cessTrap(EL2, 0x1D);
9
                        && HCR_EL2.E2H == '1' && CPTR_EL2.SMEN == 'x0' then
10
           sif F
              .ch64.SystemAccessTrap(EL2, 0x1D);
11
12
        els: HaveEL(EL3) && CPTR_EL3.ESM == '0' then
             i. alted() && EDSCR.SDD == '1' then
13
                  'DEFINED;
14
15
             else
16
                AArch64.SystemAccessTrap(EL3, 0x1D);
        elsif EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '111' then
17
18
            X[t, 64] = NVMem[0x1F0];
19
        else
            X[t, 64] = SMCR\_EL1;
20
21
    elsif PSTATE.EL == EL2 then
        if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3
22
             →trap priority when SDD == '1'" && CPTR_EL3.ESM == '0' then
23
             UNDEFINED:
        elsif HCR_EL2.E2H == '0' && CPTR_EL2.TSM == '1' then
24
25
             AArch64.SystemAccessTrap(EL2, 0x1D);
        elsif HCR_EL2.E2H == '1' && CPTR_EL2.SMEN == 'x0' then
26
27
             AArch64.SystemAccessTrap(EL2, 0x1D);
28
        elsif HaveEL(EL3) && CPTR_EL3.ESM == '0' then
29
            if Halted() && EDSCR.SDD == '1' then
30
                 UNDEFINED;
31
             else
```

```
Chapter E3. System registers affected by SME E3.1. SME-Specific System registers
```

```
AArch64.SystemAccessTrap(EL3, 0x1D);
elsif HCR_EL2.E2H == '1' then
32
33
34
            X[t, 64] = SMCR\_EL2;
35
        else
36
             X[t, 64] = SMCR\_EL1;
37
    elsif PSTATE.EL == EL3 then
        if CPTR_EL3.ESM == '0' then
38
39
             AArch64.SystemAccessTrap(EL3, 0x1D);
40
        else
41
             X[t, 64] = SMCR\_EL1;
```

MSR SMCR_EL1, <Xt>

		ор0	op1	CRn	CDη	op2	
		0b11	0b000	0Ь0001	0b0016	0b110	
1	if PSTATE.EL ==	= ELO then					
2	UNDEFINED;	FI1 +hon					
4	if Halted()) && HaveEL(EL3)	&& EDSCR.SDD ==	'l `& Jolean I	LEMENTATION DE	FINED "EL3	
5	→trap	priority when S	SDD == '1'" &	TR_EL. SM == '	then		
6	UNDEFINED;						
7	AArch64 SystemAccessTrap(EL1 (x1D))						
8	elsif El2Enabled() & HCR EL2.E2H '0' 2.TSM == '1' then						
9	AArch64.SystemAccessTrap 2, 0x						
10	elsif EL2En	nabled() && HCR_	FL2. Y == & &	CPTR_EL2.SMEN =	= 'x0' then		
11	AArch64.SystemAccess ⁺ _p(EL2, ^x1D);						
12	elsif HaveEL(EL3) && CF _EL3.ESM '0' t .m						
13	if Halted() & $\mathbf{E}\mathbf{F}$ SDD == '1' in						
14 15	UNDEFINED;						
16	AA	rch6/ vstemAcce	ss) p(EL3, 0x1D)				
17	elsif EL2En	na ⁺ .d() && HCR	EL2. 2.NV1.NV>	/ == '111' then			
18	NVMem $[0, F0] = Y$, 64]:						
19	else						
20	SMCR_EI	Li V., 64];					
21	l elsif PST EL = 42 then						
22	22 if .ite. && h pel _3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3						
22	⇔tra priori,						
23	UNDF' NED;						
24 25	coh64_SustemProcessTrap(FL2Ox1D):						
25	elsi VCR EL2 E2H == '1' & CPTR EL2 SMEN == ' $\sqrt{0}$ ' then						
27	AA. ``64	4.SvstemAccessTr	ap(EL2, 0x1D);	No Unon			
28	elsif Ha J	EL(EL3) && CPTR_	EL3.ESM == '0' th	ien			
29	if Halt	ted() && EDSCR.S	DD == '1' then				
30	UNI	DEFINED;					
31	else						
32	AA:	rch64.SystemAcce	ssTrap(EL3, 0x1D)	;			
33	elsif HCR_H	EL2.E2H == '1' t	hen				
34 25	SMCR_E	L2 = X[t, 64];					
33 36	eise	1 - Y + 6/1					
37	elsif PSTATE.EL == EL3 then						
38	if CPTR EL	3.ESM == '0' the	n				
39	AArch6	4.SystemAccessTr	ap(EL3, 0x1D);				
40	else	-					
41	SMCR_EI	L1 = X[t, 64];					

E3.1.5 SMCR_EL3, SME Control Register (EL3)

The SMCR_EL3 characteristics are:

Purpose

This register controls aspects of Streaming SVE that are visible at all Exception levels.

Configuration

This register has no effect if the PE is not in Streaming SVE mode.

This register is present only when FEAT_SME is implemented and EL3 is implemented. Otherwise, direct accesses to SMCR_EL3 are UNDEFINED.

Attributes

SMCR_EL3 is a 64-bit register.

Field descriptions

The SMCR_EL3 bit assignments are:



Bits [63:32]

Reserved, RESO.

FA64, bit [31]

When FEAT_SN _FA64 i' nplemented:

Controls whether e cut in of an A 4 instruction is considered legal when the PE is in Streaming SVE mode.

FA64	Meaning
060	This control does not cause any instruction to be treated as legal in Streaming SVE mode.
0b1	This control causes all implemented A64 instructions to be treated as legal in Streaming SVE mode at EL3.

Arm recommends that portable SME software should not rely on this optional feature, and that operating systems should provide a means to test for compliance with this recommendation.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

res0
EZT0, bit [30]

When FEAT_SME2 is implemented:

Traps execution at all Exception levels of the LDR, LUTI2, LUTI4, MOVT, STR, and ZERO instructions that access the ZT0 register to EL3.

The exception is reported using ESR_EL3.EC value 0x1D, with an ISS code of 0x0000004, at a lower priority than a trap due to PSTATE.SM or PSTATE.ZA.

EZT0	Meaning
0b0	This control causes execution of these instruction ¹¹ Exception levels to be trapped.
0b1	This c trol does it. cause execution of any institution to be trap d.

Changes to this field only affect whether instructions that e^{-1} ess ZTO e^{-1} re trap_k e^{-1} ney do not affect the contents of ZTO, which remain valid so long as PSTATE.ZA is 1.

The reset behavior of this field is:

• On a Warm reset, this field resets to an ar litecture y UNKN value.

Otherwise:

res0

Bits [29:9]

Reserved, RESO.

Bits [8:4]

Reserved, RAZ/

LEN, bits [0.0]

Requires an Effictive Streeting SVE vector length (SVL) at EL3 of (LEN+1)*128 bits.

The Sami SVE of length can be any power of two from 128 bits to 2048 bits inclusive. An implementation can support any subset of the architecturally permitted lengths.

When the PE in Streaming SVE mode, the Effective SVE vector length (VL) is equal to SVL.

When FEAT_SVE is implemented, and the PE is not in Streaming SVE mode, VL is equal to the Effective Non-streaming SVE vector length. See ZCR_EL3.

For all purposes other than returning the result of a direct read of SMCR_EL3, the PE selects the Effective Streaming SVE vector length by performing checks in the following order:

- 1. If the requested length is less than the minimum implemented Streaming SVE vector length, then the Effective length is the minimum implemented Streaming SVE vector length.
- 2. Otherwise, the Effective length is the highest supported Streaming SVE vector length that is less than or equal to the requested length.

An indirect read of SMCR_EL3.LEN appears to occur in program order relative to a direct write of the same register, without the need for explicit synchronization.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing SMCR_EL3

Accesses to this register use the following encodings in the System register encoding space:

MRS <Xt>, SMCR_EL3

		ор0	op1	CRn	CRm	op2
		0b11	0b110	0b0001	0b0010	0b110
1	if PSTATE.EL ==	= ELO then				
3	elsif PSTATE.EI	L == EL1 then				
4	UNDEFINED;					
5	elsif PSTATE.EI	L == EL2 then				
6 7	UNDEFINED;	FI2 then				
8	if CPTR EL3	3.ESM == '0' the	n			
9	AArch64	1.SystemAccessTra	ap(EL3, 0x1D):			
10	else					
11	X[t, 64	1] = SMCR_EL3;				
	MSR SMCR_EL	3, <xt></xt>				
		op0	op1	CRn	CRm	op2
		0b11	ъ110	0b0001	0b0010	0b110
1	if PSTATE.EL =	LO the	*			
2	UNDEFINED;					
3	elsif PSTATE.EI	I I then				
5	elsif / ATE.	L == L. th				
6	JEFINED					
7	el: PSTATF	-= EL3 then				
8	▲ CPT _EL:	S.E. = '0' the	n			
9 10	else	.systemAccessira	ар(ггэ, охто);			
11	SMC EI	L3 = X[t, 64];				

E3.1.6 SMIDR_EL1, Streaming Mode Identification Register

The SMIDR_EL1 characteristics are:

Purpose

Provides additional identification mechanisms for scheduling purposes, for a PE that supports Streaming SVE mode.

Configuration

This register is present only when FEAT_SME is implemented. Otherwise, direct accesses to SMIDR_EL1 are UNDEFINED.

Attributes

SMIDR_EL1 is a 64-bit register.

Field descriptions

The SMIDR_EL1 bit assignments are:



Bits [63:32]

Reserved, RESO.

Implementer, bits [31:?4]

The Implementer c e. This field must hold an implementer code that has been assigned by Arm. Assigned codes include the folloging:

Implementer	Meaning
0x00	Reserved for software use.
0x41	Arm Limited.
0x42	Broadcom Corporation.
0x43	Cavium Inc.
0x44	Digital Equipment Corporation.
0x46	Fujitsu Ltd.
0x49	Infineon Technologies AG.
0x4D	Motorola or Freescale Semiconductor Inc.
0x4E	NVIDIA Corporation.
0x50	Applied Micro Circuits Corporation.
0x51	Qualcomm Inc.
0x56	Marvell International Ltd.

Implementer	Meaning
0x69	Intel Corporation.
0xC0	Ampere Computing.

Arm can assign codes that are not published in this manual. All values not assigned by Arm are reserved and must not be used.

It is not required that this value is the same as the value of MIDR_EL1.Implementer.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Revision, bits [23:16]

Revision number for the Streaming Mode Compute Unit (SMC)

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

SMPS, bit [15]

Indicates support for Streaming SVE mode / ccution p ority.

MPS	Meaning
060	Priority control not supported.
0b1	Priority control supported.

Bits [14:12]

Reserved, pro0.

Affini', bits [:0]

The S CU mnuy ... accessing PE.

- A va of zero indicates that the PE's implementation of Streaming SVE mode is not shared with other PEs.
- Otherwis, the value identifies which SMCU is associated with this PE. The Affinity value associated with each SMCU is unique within the system as a whole.

Accessing SMIDR_EL1

Accesses to this register use the following encodings in the System register encoding space:

MRS <Xt>, SMIDR_EL1

ор0	op1	CRn	CRm	op2	
0b11	0b001	0b0000	0b0000	0b110	

```
if PSTATE.EL == ELO then
1
2
        \quad \text{if} \text{ IsFeatureImplemented(FEAT_IDST)} \ \text{then} \\
3
             if EL2Enabled() && HCR_EL2.TGE == '1' then
                AArch64.SystemAccessTrap(EL2, 0x18);
4
5
            else
6
                AArch64.SystemAccessTrap(EL1, 0x18);
7
        else
8
            UNDEFINED;
9
   elsif PSTATE.EL == EL1 then
10
        if EL2Enabled() && HCR_EL2.TID1 == '1' then
11
            AArch64.SystemAccessTrap(EL2, 0x18);
12
        else
13
            X[t, 64] = SMIDR\_EL1;
   elsif PSTATE.EL == EL2 then
14
        X[t, 64] = SMIDR\_EL1;
15
    elsif PSTATE.EL == EL3 then
16
    X[t, 64] = SMIDR\_EL1;
17
```

E3.1.7 SMPRI_EL1, Streaming Mode Priority Register

The SMPRI_EL1 characteristics are:

Purpose

Configures the streaming execution priority for instructions executed on a shared Streaming Mode Compute Unit (SMCU) when the PE is in Streaming SVE mode at any Exception Level.

Configuration

When **SMIDR_EL1.SMPS** is '0', this register is RES0.

This register is present only when FEAT_SME is implemented. Otherwise, direct accesses to SMPRI_EL1 are UNDEFINED.

Attributes

SMPRI_EL1 is a 64-bit register.

Field descriptions

The SMPRI_EL1 bit assignments are:



Bits [63:4]

Reserved, RESO.

Priority, bits [3:0]

Streaming execut a priority alue.

Either this value is variable and a set of the set of t

This value used any of the following are true:

- the curre Exception level is EL3 or EL2.
- The compared on the exception level is EL1 or EL0, if EL2 is either not implemented or not enabled in the current Security stee.

The precise meaning and behavior of each streaming execution priority value is IMPLEMENTATION DEFINED.

In an implementation that shares execution resources between PEs, higher priority values are allocated more processing resource than other PEs configured with lower priority values in the same Priority domain.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing SMPRI_EL1

Accesses to this register use the following encodings in the System register encoding space:

MRS <Xt>, SMPRI_EL1

ор0	op1	CRn	CRm	op2	
0b11	0b000	0b0001	0b0010	0b100	



	оро	орт	CKn	CRM	op2
	Ve	0b000	0b0001	0b0010	0b100
1	if PSTATE.EL = ELO then				
2	UNDEFINED;				
3	elsif PSTATE.EL == EL1 the	n			
4	if Halted() && HaveEL(EL3) && EDSCR.SDI) == '1' && boole	an IMPLEMENTATIO	N_DEFINED "EL3
-	⇔trap priority wh	en SDD == '1'" &	& CPTR_EL3.ESM ==	• '0' then	
5	UNDEFINED;				
6	elsif EL2Enabled() &&	IsFeatureImplemen	nted(FEAT_FGT) &&	(!HaveEL(EL3)	SCR_EL3.FGTEn
_	↔== '1') && HFGWI	'R_EL2.nSMPRI_EL1	== '0' then		
7	AArch64.SystemAcce	ssTrap(EL2, 0x18));		
8	elsif HaveEL(EL3) && C	PTR_EL3.ESM == '()' then		
9	if Halted() && EDS	CR.SDD == '1' the	en		
10	UNDEFINED;				
11	else				
12	AArch64.System	AccessTrap(EL3, (0x18);		
13	else				
14	$SMPRI_EL1 = X[t, 6]$	4];			
15	elsif PSTATE.EL == EL2 the	n			
16	if Halted() && HaveEL(EL3) && EDSCR.SDI) == '1' && boole	an IMPLEMENTATIO	N_DEFINED "EL3
	⇔trap priority wh	en SDD == '1'" &	& CPTR_EL3.ESM ==	• '0' then	

Copyright © 2022-2023 Arm Limited or its affiliates. All rights reserved. Non-confidential

```
17
            UNDEFINED;
18
        elsif HaveEL(EL3) && CPTR_EL3.ESM == '0' then
19
             if Halted() && EDSCR.SDD == '1' then
20
                 UNDEFINED;
21
             else
22
                 AArch64.SystemAccessTrap(EL3, 0x18);
23
        else
24
             SMPRI\_EL1 = X[t, 64];
    elsif PSTATE.EL == EL3 then
   if CPTR_EL3.ESM == '0' then
25
26
27
            AArch64.SystemAccessTrap(EL3, 0x18);
28
        else
29
            SMPRI\_EL1 = X[t, 64];
```

E3.1.8 SMPRIMAP_EL2, Streaming Mode Priority Mapping Register

The SMPRIMAP_EL2 characteristics are:

Purpose

Maps the value in SMPRI_EL1 to a streaming execution priority value for instructions executed at EL1 and EL0 in the same Security states as EL2.

Configuration

When **SMIDR_EL1.SMPS** is '0', this register is RES0.

If EL2 is not implemented, this register is RESO from EL3.

This register is present only when FEAT_SME is implemented. Otherwise, direct accesses to SMPRIMAP_EL2 are UNDEFINED.

Attributes

SMPRIMAP_EL2 is a 64-bit register.

Field descriptions

The SMPRIMAP_EL2 bit assignments are:

63		60 ₁ 5	9	56 I	55		52	, F		48	,	4.			40	39		36	35		32
	P15		P14			P13			P12		F	211		P10			P9			P8	
31		28 2	7	24	23		20	19		16	15	12	111		8	7		4	3		0
	P7		P6			P5						Р3		P2			Ρ1			P0	

When all of the following are ue, • value in SN, **PI_EL1** is mapped to a streaming execution priority using this register:

- The current Ex_{P} tion level is EL1 EL0.
- EL2 is imple .ented ar enabled in the current Security state.
- HCRX_EL. `MPM' is '1'.

Otherwise **PRI** holds t⁺ streaming execution priority value.

P15 ,its [63:/]

This value is the highest streaming execution priority.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

P14, bits [59:56]

Priority Mapping Entry 14. This entry is used when priority mapping is supported and enabled, and the SMPRI_EL1.Priority value is '14'.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

P13, bits [55:52]

Priority Mapping Entry 13. This entry is used when priority mapping is supported and enabled, and the SMPRI_EL1.Priority value is '13'.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

P12, bits [51:48]

Priority Mapping Entry 12. This entry is used when priority mapping is supported and enabled, and the SMPRI_EL1.Priority value is '12'.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOW, alue.

P11, bits [47:44]

Priority Mapping Entry 11. This entry is used when *r* ority m^r ping 1. ^r ported and enabled, and the SMPRI_EL1.Priority value is '11'.

The reset behavior of this field is:

• On a Warm reset, this field resets to an ar litecture y UNKN value.

P10, bits [43:40]

Priority Mapping Entry 10. This entry a used ways priority mapping is supported and enabled, and the SMPRI_EL1.Priority value is '1'.

The reset behavior of this f _____is:

• On a Warm reset ins field resets on architecturally UNKNOWN value.

P9, bits [39:36'

Priority Mapping E. (9. The entry is used when priority mapping is supported and enabled, and the SMPRI ΔL 1.F prity value is (

The set beha or of this field is:

• C. (arm reset, this field resets to an architecturally UNKNOWN value.

P8, bits [35. ۲

Priority Mapping Entry 8. This entry is used when priority mapping is supported and enabled, and the SMPRI_EL1.Priority value is '8'.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

P7, bits [31:28]

Priority Mapping Entry 7. This entry is used when priority mapping is supported and enabled, and the SMPRI_EL1.Priority value is '7'.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

P6, bits [27:24]

Priority Mapping Entry 6. This entry is used when priority mapping is supported and enabled, and the SMPRI_EL1.Priority value is '6'.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

P5, bits [23:20]

Priority Mapping Entry 5. This entry is used when priority mapping is supported and enabled, and the SMPRI_EL1.Priority value is '5'.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOW, alue.

P4, bits [19:16]

Priority Mapping Entry 4. This entry is used when r priving is ported and enabled, and the SMPRI_EL1.Priority value is '4'.

The reset behavior of this field is:

• On a Warm reset, this field resets to an *ar* intecture y UNKN value.

P3, bits [15:12]

Priority Mapping Entry 3. This atry a used with priority mapping is supported and enabled, and the SMPRI_EL1.Priority value is '2'

The reset behavior of this f ____is:

• On a Warm reset ins field resets on architecturally UNKNOWN value.

P2, bits [11:8]

Priority Mapping E. (2. The entry is used when priority mapping is supported and enabled, and the SMPRI ΔL 1.F prity value is (

The set beha or of this field is:

• C. arm reset, this field resets to an architecturally UNKNOWN value.

P1, bits [7:4]

Priority Mapping Entry 1. This entry is used when priority mapping is supported and enabled, and the SMPRI_EL1.Priority value is '1'.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

P0, bits [3:0]

Priority Mapping Entry 0. This entry is used when priority mapping is supported and enabled, and the SMPRI_EL1.Priority value is '0'.

This value is the lowest streaming execution priority.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing SMPRIMAP_EL2

Accesses to this register use the following encodings in the System register encoding space:

MRS <Xt>, SMPRIMAP_EL2

	op0	op1	CRn	CRm	op2
	0b11	0b100	r J001	01 10	0b101
I if PSTAT	TE.EL == ELO then				
2 UNDI	EFINED;				
elsit Pa	STATE.EL == ELI th				
	ELZENADIEC() & HU	K_ELZ. <nvz,nv =="<br">0.11501.</nvz,nv>			
	f FI2Enabled() ff	HCD FI2 '	then		
7	AArch64 SystemAcc	ess n(EL/)x18	.nen		
, B else	a	coo vp (DD2, Mic			
)	UNDEFINED;				
) elsif PS	STATE.EL == EL2	en			
l if H	Halted() && HrveEL	(EL3) EDSCR.SD	D == '1' && boole	an IMPLEMENTATIC	N_DEFINED "EL3
	⇔trap pri⁄ .ty w	hen SDD '1'" 8	& CPTR_EL3.ESM ==	'0' then	
2	UNDEFINED				
3 els:	if HaveF (EL3) &/	CPTR_EL3.ESM == '	0 ' then		
1	if Ha. d() & _D	SCR.SDD == '1' th	en		
5	UNDL NF				
5	e'				
	rch64. ste	cessTrap(EL3,	0x18);		
5 F.6					
	X[t, J4] = SMPRIM	AP_ELZ;			
) eisi 1 it	FIS FCM '0	then			
,	sch64 SystemAcc	essTran(EL3, 0x18	· ·		
- B else	a	COSTINE (DEC, ONIO	//		
4	X[t, J4] = SMPRIM	AP EL2;			
		_ ,			

MSR SMPRIMAP_EL2, <Xt>

орО	op1	CRn	CRm	op2
0b11	0b100	0b0001	0b0010	0b101

```
1 if PSTATE.EL == EL0 then
2 UNDEFINED;
3 elsif PSTATE.EL == EL1 then
4 if EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
5 NVMem[0x1F8] = X[t, 64];
6 elsif EL2Enabled() && HCR_EL2.NV == '1' then
```

```
Chapter E3. System registers affected by SME E3.1. SME-Specific System registers
```

```
7
           AArch64.SystemAccessTrap(EL2, 0x18);
8
       else
9
            UNDEFINED;
10
   elsif PSTATE.EL == EL2 then
       if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3
11

when SDD == '1'" && CPTR_EL3.ESM == '0' then

           UNDEFINED;
12
13
       elsif HaveEL(EL3) && CPTR_EL3.ESM == '0' then
14
            if Halted() && EDSCR.SDD == '1' then
15
               UNDEFINED;
16
            else
17
               AArch64.SystemAccessTrap(EL3, 0x18);
18
       else
19
           SMPRIMAP\_EL2 = X[t, 64];
20
   elsif PSTATE.EL == EL3 then
       if CPTR_EL3.ESM == '0' then
21
22
           AArch64.SystemAccessTrap(EL3, 0x18);
23
       else
24
           SMPRIMAP\_EL2 = X[t, 64];
```

E3.1.9 SVCR, Streaming Vector Control Register

The SVCR characteristics are:

Purpose

Controls Streaming SVE mode and SME behavior.

Configuration

This register is present only when FEAT_SME is implemented. Otherwise, direct accesses to SVCR are UNDEFINED.

Attributes

SVCR is a 64-bit register.

Field descriptions

The SVCR bit assignments are:



Bits [63:2]

Reserved, RESO.

ZA, bit [1]

Enables SME ZA storacy If FEAT_5 52 is implemented, also enables SME2 ZT0 storage.

When this storage is asabled, execution or a instruction which can access it is trapped. The exception is reported using an ESR_F' ..{EC, S' .C} value of {0x1D, 0x3}.

The possible values .s bit are.

ZA	Meaning
060	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, this field resets to Ob0.

SM, bit [0]

Enables 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:



When a write to SVCR.SM changes the valu. If PST/ The following applies:

- When changed from 0 to 1, an en, to Streeting SVE mode is performed.
- When changed from 1 to 0, ar xit find Streating SVE mode is performed.
- All implemented bits of the VE register Z0-Z3. P0-P15, and FFR in the new mode are set to zero.
- FPSR in the new mode $3 \times 900000_{0} \times 0800_{0}09f$, in which all cumulative status bits are set to 1.

Changes to this field do not have an a ct on SME ZA storage or, if implemented, ZT0 storage.

A direct or indirect ad of SM appears occur in program order relative to a direct write of SVCR, and to MSR SVCRSM and X SVCRSY A instructions, without the need for explicit synchronization.

The reset behavior `th' field is:

• Or was reset, 's fie' resets to 0b0.

Ac ssin

SVCR is 1/write and can be accessed from any Exception level.

Accesses to the register use the following encodings in the System register encoding space:

MRS <Xt>, SVCR

орО	op1	CRn	CRm	op2
0b11	0b011	0b0100	0b0010	0b010

1	if PSTATE.EL == EL0 then
2	<pre>if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3</pre>
	↔trap priority when SDD == '1' " && CPTR_EL3.ESM == '0' then
3	UNDEFINED;
4	elsif !(EL2Enabled() && HCR_EL2. <e2h,tge> == '11') && CPACR_EL1.SMEN != '11' then</e2h,tge>
5	if EL2Enabled() && HCR_EL2.TGE == '1' then
6	AArch64.SystemAccessTrap(EL2, 0x1D);



MSR SVCR, <Xt>

ор0	op1	CRn	CRm	op2
0b11	0b011	0b0100	0b0010	0b010

```
6
                AArch64.SystemAccessTrap(EL2, 0x1D);
7
            else
8
                AArch64.SystemAccessTrap(EL1, 0x1D);
9
        elsif EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' && CPTR_EL2.SMEN != '11' then
10
            AArch64.SystemAccessTrap(EL2, 0x1D);
11
        elsif EL2Enabled() & HCR_EL2.E2H == '1' & CPTR_EL2.SMEN == 'x0' then
           AArch64.SystemAccessTrap(EL2, 0x1D);
12
13
        elsif EL2Enabled() && HCR_EL2.E2H == '0' && CPTR_EL2.TSM == '1' then
            AArch64.SystemAccessTrap(EL2, 0x1D);
14
        elsif HaveEL(EL3) && CPTR_EL3.ESM == '0' then
15
            if Halted() && EDSCR.SDD == '1' then
16
17
                UNDEFINED;
18
            else
19
                AArch64.SystemAccessTrap(EL3, 0x1D);
20
        else
21
            SetPSTATE_SVCR(X[t, 32]);
22
   elsif PSTATE.EL == EL1 then
       if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boole / IMPLEM. 'ATION_DEFINED "EL3

→trap priority when SDD == '1'' && CPTR_EL3.ESM / '0' then
23
24
            UNDEFINED;
25
        elsif CPACR_EL1.SMEN == 'x0' then
26
            AArch64.SystemAccessTrap(EL1, 0x1D);
        elsif EL2Enabled() && HCR_EL2.E2H == '0' && CP _EL2.TSM =
27
                                                                              an
28
            AArch64.SystemAccessTrap(EL2, 0x1D);
29
        elsif EL2Enabled() && HCR_EL2.E2H == '1'
                                                   88
                                                        TR_ET .SMEN ==
                                                                        'x0' then
30
            AArch64.SystemAccessTrap(EL2, 0x1D);
31
        elsif HaveEL(EL3) && CPTR_EL3.ESM ==
                                                      ٩n
32
            if Halted() && EDSCR.SDD == '1'
                                              en،
                UNDEFINED;
33
34
            else
35
                AArch64.SystemAccessTrap(E1
                                                0 0);
36
        else
37
            SetPSTATE_SVCR(X[t, 32);
38
   elsif PSTATE.EL == EL2 then
       39
40
            UNDEFINED;
        elsif HCR_EL2.E2 == '0' && TR_EL2.TSM == '1' then
41
           AArch64.Sv emAccessTrap(L . 0x1D);
42
        elsif HCR_EI E2H == '1' && CPT._EL2.SMEN == 'x0' then
AArch6 SystemA essTrap(EL2, 0x1D);
elsif Have. 'EL3) & CPTR_EL3.ESM == '0' then
43
44
45
46
            if Halter & EDSCR .D == '1' then
47
                 UDEFI. );
48
            else
49
                P rch64.Sy. mAccessTrap(EL3, 0x1D);
50
         lse
            Sr .STATE____R(X[t, 32]);
51
            TE.EL == EL3 then
52
    elsif F
53
        if CF EL3.ESM == '0' then
54
            AA1 54.SystemAccessTrap(EL3, 0x1D);
55
        else
56
            SetPSTATE_SVCR(X[t, 32]);
```

MSR SVCRSM, #<imm>

op0	op1	CRn	CRm	op2
0b00	0b011	0b0100	0b001x	0b011

MSR SVCRZA, #<imm>

ор0	op1	CRn	CRm	op2
0b00	0b011	0b0100	0b010x	0b011

MSR SVCRSMZA, #<imm>

opo	op1	CRn	CRm	op2
)b00	0b011	0b0100	0b011x	0b011

E3.1.10 TPIDR2_EL0, EL0 Read/Write Software Thread ID Register 2

The TPIDR2_EL0 characteristics are:

Purpose

Provides a location where SME-aware software executing at EL0 can store thread identifying information, for context management purposes.

The PE makes no use of this register.

Configuration

This register is present only when FEAT_SME is implemented. Otherwise, direct accesses to TPIDR2_EL0 are UNDEFINED.

Attributes

TPIDR2_EL0 is a 64-bit register.

Field descriptions

The TPIDR2_EL0 bit assignments are:

63		32
	Thre ID	
31		0
	T' cdu 12	

Bits [63:0]

Thread identifying information sto. 'by software anning at this Exception level.

The reset behavior of *t*' s field is:

• On a Warm · .et, this f ¹d resets to an architecturally UNKNOWN value.

Accessing TPI. 7._EL0

2

Access to this egister u. ... e following encodings in the System register encoding space:

MRL Xt>,

орО	op1	CRn	CRm	op2	
0b11	0b011	0b1101	0b0000	0b101	

```
if PSTATE.EL == EL0 then
1
2
       if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3
          3
          UNDEFINED;
4
       elsif !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && SCTLR_EL1.EnTP2 == '0' then
5
          if EL2Enabled() && HCR_EL2.TGE == '1' then
6
              AArch64.SystemAccessTrap(EL2, 0x18);
7
          else
              AArch64.SystemAccessTrap(EL1, 0x18);
8
9
       elsif EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' && SCTLR_EL2.EnTP2 == '0' then
10
          AArch64.SystemAccessTrap(EL2, 0x18);
       elsif EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && IsFeatureImplemented(FEAT_FGT) &&
11
          ↔ (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGRTR_EL2.nTPIDR2_EL0 == '0' then
```

```
12
           AArch64.SystemAccessTrap(EL2, 0x18);
13
       elsif HaveEL(EL3) && SCR_EL3.EnTP2 == '0' then
14
            if Halted() && EDSCR.SDD == '1' then
15
               UNDEFINED:
16
            else
17
               AArch64.SystemAccessTrap(EL3, 0x18);
18
       else
            X[t, 64] = TPIDR2\_EL0;
19
20
   elsif PSTATE.EL == EL1 then
       if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3
21
            →trap priority when SDD == '1'" && SCR_EL3.EnTP2 == '0' then
22
            UNDEFINED:
23
        elsif EL2Enabled() & IsFeatureImplemented(FEAT_FGT) & (!HaveEL(EL3) || SCR_EL3.FGTEn
            24
            AArch64.SystemAccessTrap(EL2, 0x18);
25
        elsif HaveEL(EL3) && SCR_EL3.EnTP2 == '0' then
26
            if Halted() && EDSCR.SDD == '1' then
27
               UNDEFINED:
28
            else
29
               AArch64.SystemAccessTrap(EL3, 0x18);
30
       else
31
            X[t, 64] = TPIDR2\_EL0;
   elsif PSTATE.EL == EL2 then
32
                                                      && bor _an IMPL
33
       if Halted() && HaveEL(EL3) && EDSCR.SDD == /
                                                                         LNTATION_DEFINED "EL3
            ↔trap priority when SDD == '1'" && SCR____
                                                       3.En7
                                                              == '0' then
34
           UNDEFINED;
       elsif HaveEL(EL3) && SCR_EL3.EnTP2 ==
35
                                                   hen
36
            if Halted() && EDSCR.SDD == '1'
                                            .en
37
               UNDEFINED;
38
            else
39
               AArch64.SystemAccessTrap(EL
                                              0
                                                 d);
40
        else
41
            X[t, 64] = TPIDR2\_EL0.
42
   elsif PSTATE.EL == EL3 then
43
       X[t, 64] = TPIDR2\_EL0
   MSR TPIDR2 EL0, <Xt>
                   ٠p0
                                                    CRn
                                                                    CRm
                                   op1
                                                                                     op2
                                   0b011
                   0b1
                                                    0b1101
                                                                    0b0000
                                                                                     0b101
   if P. NTE.
1
                         .en
2
       it
             ced() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3
              rap priority when SDD == '1'" && SCR_EL3.EnTP2 == '0' then
3
                TINED;
       elsif !(. _2Enabled() && HCR_EL2.<E2H,TGE> == '11') && SCTLR_EL1.EnTP2 == '0' then
4
            if EL2Enabled() && HCR_EL2.TGE == '1' then
5
               AArch64.SystemAccessTrap(EL2, 0x18);
6
7
            else
               AArch64.SystemAccessTrap(EL1, 0x18);
8
       elsif EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' && SCTLR_EL2.EnTP2 == '0' then
9
            AArch64.SystemAccessTrap(EL2, 0x18);
10
        elsif El2Enabled() && HCR_EL2.<E2H,TGE> != '11' && IsFeatureImplemented(FEAT_FGT) &&
11
            ↔ (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGWTR_EL2.nTPIDR2_EL0 == '0' then
12
            AArch64.SystemAccessTrap(EL2, 0x18);
13
        elsif HaveEL(EL3) && SCR_EL3.EnTP2 == '0' then
14
           if Halted() && EDSCR.SDD == '1' then
15
               UNDEFINED;
16
            else
17
                AArch64.SystemAccessTrap(EL3, 0x18);
```

18

19

20

else

TPIDR2_ELO = X[t, 64];

elsif PSTATE.EL == EL1 then

```
if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & Boolean IMPLEMENTATION_DEFINED "EL3
21
           22
           UNDEFINED;
23
       elsif EL2Enabled() & IsFeatureImplemented(FEAT_FGT) & (!HaveEL(EL3) || SCR_EL3.FGTEn
           ↔== '1') && HFGWTR_EL2.nTPIDR2_EL0 == '0' then
24
           AArch64.SystemAccessTrap(EL2, 0x18);
25
       elsif HaveEL(EL3) && SCR_EL3.EnTP2 == '0' then
26
           if Halted() && EDSCR.SDD == '1' then
27
              UNDEFINED;
28
           else
29
              AArch64.SystemAccessTrap(EL3, 0x18);
30
       else
31
           TPIDR2\_EL0 = X[t, 64];
32
   elsif PSTATE.EL == EL2 then
33
       if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3
           →trap priority when SDD == '1'" && SCR_EL3.EnTP2 == '0' then
34
           UNDEFINED;
35
       elsif HaveEL(EL3) && SCR_EL3.EnTP2 == '0' then
36
           if Halted() && EDSCR.SDD == '1' then
37
              UNDEFINED;
38
           else
39
               AArch64.SystemAccessTrap(EL3, 0x18);
40
       else
41
           TPIDR2_ELO = X[t, 64];
42
   elsif PSTATE.EL == EL3 then
       TPIDR2\_EL0 = X[t, 64];
43
```

E3.1.11 EDHSR, External Debug Halting Syndrome Register

The EDHSR characteristics are:

Purpose

Holds syndrome information for a debug event.

Configuration

EDHSR is in the Core power domain

EDHSR is in the Core power domain.

This register is present only when FEAT_Debugv8p2 is implemented and an implementation implements EDHSR. Otherwise, direct accesses to EDHSR are RES0.

Attributes

EDHSR is a 64-bit register.

Field descriptions

The EDHSR bit assignments are:



Bits [63:24]

Reserved, RESO.

WPT, bits [23:1

Watchpoint number. In EDH C.WPTV is 1, holds the index of a watchpoint that triggered the Watchpoint debug erant.

The set beha or of this field is:

• C. Arm reset, this field resets to an architecturally UNKNOWN value.

WPTV, bit [.

Watchpoint number valid.

WPTV	Meaning
0b0	EDHSR.WPT field is not valid, and holds an UNKNOWN value.
0b1	EDHSR.WPT field is valid, and holds the number of a watchpoint that triggered the Watchpoint debug event.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

WPF, bit [16]

Watchpoint match might be False.

WPF	Meaning	Applies
0b0	The watchpoint matched the original access or set of contiguous accesses.	
0b1	The watchpoint matched an access or set of contiguous accesses where the lowest accessed address was rounded down to the multiple of 16 bytes and the highest access, address was rounded up to the pearest multiple of 16 bytes minus 1, but the way point might not have matched the signal addition of the access or set of core guous accesses.	When FEAT_SME is implemented or FEAT_SVE is implemented

The reset behavior of this field is:

• On a Warm reset, this field resets to an *chitectur* ly UNKNC I value.

FnP, bit [15]

EDWAR not Precise.

IP	Meaning	Applies
060	h • EDWAR is valid, it holds the virtual address of an access or sequence of contiguous accesses that triggered the Watchpoint debug event.	
1b1	If the EDWAR is valid, it holds any virtual address within the smallest implemented translation granule that contains the virtual address of an access or set of contiguous accesses that triggered the Watchpoint debug event.	When FEAT_SME is implemented or FEAT_SVE is implemented

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [14:11]

Reserved, RESO.

FnV, bit [10]

EDWAR not Valid.

FnV	Meaning	Applies
060	EDWAR is valid.	
0b1	EDWAR is not valid, and holds an UNKNOWN value.	When FEAT_SME is implemented or FEAT_SVE is implemented

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [9:0]

Reserved, RESO.

Accessing EDHSR

Accesses to this register use the following encodings in the xterre debug interface:

EDHSR can be accessed through the external debr

Sompo.	Offset	Instance
Deb.	0x038	EDHSR

This interface is accessible folle

• When DoubleLoc' tatus(), or !Isc 'ePowered() or OSLockStatus() access to this register returns an ERROR.

rface.

• Otherwise acc \Box to this register is κ .

Chapter E3. System registers affected by SME E3.2. Changes to existing System registers

E3.2 Changes to existing System registers

System registers that are updated with additional fields, values, or description changes, to support SME functionality.

E3.2.1 CPACR_EL1, Architectural Feature Access Control Register

The CPACR_EL1 characteristics are:

Purpose

Controls access to trace, SME, Streaming SVE, SVE, and Advanced SIMD and floating-point functionality.

Configuration

When EL2 is implemented and enabled in the current Security state and HCR_EL2.{E2H, TGE} == $\{1, 1\}$, the fields in this register have no effect on execution at EL0 and EL1. In this case, the controls provided by CPTR_EL2 are used.

AArch64 system register CPACR_EL1 bits [31:0] are architectural poed to AArch32 system register CPACR[31:0].

Attributes

CPACR_EL1 is a 64-bit register.

Field descriptions

The CPACR_EL1 bit assignments are:



Bits [63:29]

Reserved, RESO.

TTA, bit [28]

Traps EL and L1 Sy, m registers from both Execution states to EL1, or to L2 when is implemented in the current Security state and HCR_EL2.TGE is 1, as follows:

- AA⁺ ... cesses to trace registers are trapped, reported using ESR_ELx.EC value 0x18.
- In A vch32 state, MRC and MCR accesses to trace registers are trapped, reported using ESR_ELx.EC value 0x05.
- In AArch32 state, MRRC and MCRR accesses to trace registers are trapped, reported using ESR_ELx.EC value 0x0C.

TTA	Meaning
0b0	This control does not cause any instructions to be trapped.
0b1	This control causes EL0 and EL1 System register accesses to all implemented trace registers to be trapped.

• The ETMv4 architecture and ETE do not permit EL0 to access the trace registers. If the trace unit implements FEAT_ETMv4 or FEAT_ETE, EL0 accesses to the trace registers are UNDEFINED, and any resulting exception

- is higher priority than an exception that would be generated because the value of CPACR_EL1.TTA is 1.
- The Arm architecture does not provide traps on trace register accesses through the optional memory-mapped interface.

System register accesses to the trace registers can have side-effects. When a System register access is trapped, any side-effects that are normally associated with the access do not occur before the exception is taken.

If System register access to the trace functionality is not implemented, this bit is RESO.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [27:26]

Reserved, RESO.

SMEN, bits [25:24]

When FEAT_SME is implemented:

Traps execution at EL1 and EL0 of SME instructions, SVF astructic when 7 __SVE is not implemented or the PE is in Streaming SVE mode, and instructions that din *ly acc s the SVCR or SMCR_EL1 System registers to EL1, or to EL2 when EL2 is implemented and enabled in arrent Se aty state and HCR_EL2.TGE is 1.

When instructions that directly access the SVC' Syste registe re upped with reference to this control, the MSR SVCRSM, MSR SVCRZA, and MSR SVCRSMZ' instruction are also to upped.

The exception is reported using ESR_E¹ x.EC ¹U² a 0x1D, with an ISS code of 0x0000000.

This field does not affect whether Steam. SVE c "ME register values are valid.

A trap taken as a result of CPAC EL1.SMEN s precidence over a trap taken as a result of CPACR_EL1.FPEN.

SMEN	Meaning
0b00	This control causes execution of these instructions at EL1 and EL0 to be trapped.
0b01	This control causes execution of these instructions at EL0 to be trapped, but does not cause execution of any instructions at EL1 to be trapped.
0b10	This control causes execution of these instructions at EL1 and EL0 to be trapped.
0b11	This control does not cause execution of any instructions to be trapped.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

res0

Bits [23:22]

Reserved, RESO.

FPEN, bits [21:20]

Traps execution at EL1 and EL0 of instructions that access the Advanced SIMD and floating-point registers from both Execution states to EL1, reported using ESR_ELx.EC value 0x07, or to EL2 reported using ESR_ELx.EC value 0x00 when EL2 is implemented and enabled in the current Security state and HCR_EL2.TGE is 1, as follows:

- In AArch64 state, accesses to FPCR, FPSR, any of the SIMD and floating-point registers V0-V31, including their views as D0-D31 registers or S0-31 registers.
- In AArch32 state, FPSCR, and any of the SIMD and floating-point registers Q0-15, including their views as D0-D31 registers or S0-31 registers.

Traps execution at EL1 and EL0 of SME and SVE instructions to EL1. to EL2 en EL2 is implemented and enabled for the current Security state and HCR_EL2.TGE is 1. The acception is reported using ESR_ELx.EC value 0x07.

A trap taken as a result of CPACR_EL1.SMEN has precedence over a trap taken as a sult of CPACR_EL1.FPEN.

A trap taken as a result of CPACR_EL1.ZEN has preced the over *r* ap taken a result of CPACR_EL1.FPEN.

rn. T	reani [,] ,
060	This control causes execution of these
01	This control causes execution of these instructions at EL0 to be trapped, but does not cause execution of any instructions at EL1 to be trapped.
0b10	This control causes execution of these instructions at EL1 and EL0 to be trapped.
0b11	This control does not cause execution of any instructions to be trapped.

Writ to MVF J, MVFR1, and MVFR2 from EL1 or higher are CONSTRAINED UNPREDICTABLE and whether these cess can be ped by this control depends on implemented CONSTRAINED UNPREDICTABLE behavior.

- Attents to write to the FPSID count as use of the registers for accesses from EL1 or higher.
- Access from EL0 to FPSID, MVFR0, MVFR1, MVFR2, and FPEXC are UNDEFINED, and any resulting exception is higher priority than an exception that would be generated because the value of CPACR_EL1.FPEN is not 0b11.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [19:18]

Reserved, RESO.

ZEN, bits [17:16]

When FEAT_SVE is implemented:

Chapter E3. System registers affected by SME E3.2. Changes to existing System registers

Traps execution at EL1 and EL0 of SVE instructions when the PE is not in Streaming SVE mode, and instructions that directly access the ZCR_EL1 System register to EL1, or to EL2 when EL2 is implemented and enabled in the current Security state and HCR_EL2.TGE is 1.

The exception is reported using ESR_ELx.EC value 0x19.

A trap taken as a result of CPACR_EL1.ZEN has precedence over a trap taken as a result of CPACR_EL1.FPEN.

ZEN	Meaning
0b00	This control causes execution of these instructions at EL1 and EL0 to be trapped.
0b01	This control causes execution of these instruction $a_{\rm exc} = 0$ to be trapped, but does not cause electron of v instructions at EL1 to be trap _F = 1.
0b10	nis con. I causes e cution of these instruction. EL ¹ nd EL0 to be trapped.
0b11	Th' control do, a not cause execution of any struction of be trapped.

The reset behavior of this field is:

• On a Warm reset, this field resets to an arc 'te urally UNKNOWN value.

Otherwise:

res0

Bits [15:0]

Reserved, RESO.

Accessing C. \CP _L1

When $H = E_E$ E2H is without explicit synchronization, access from EL3 using the mnemonic CPACR_EL1 or CP CR_EL are not granteed to be ordered with respect to accesses using the other mnemonic.

Acce. s to is region ase the following encodings in the System register encoding space:

MRS <Xi. CPACR_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0001	0b0000	0b010

```
1
  if PSTATE.EL == EL0 then
2
       UNDEFINED;
3
   elsif PSTATE.EL == EL1 then
4
       if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & Boolean IMPLEMENTATION_DEFINED "EL3
           →trap priority when SDD == '1'" && CPTR_EL3.TCPAC == '1' then
5
           UNDEFINED;
       elsif EL2Enabled() && CPTR_EL2.TCPAC == '1' then
6
           AArch64.SystemAccessTrap(EL2, 0x18);
7
8
       elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn
           \hookrightarrow == '1') && HFGRTR_EL2.CPACR_EL1 == '1' then
9
           AArch64.SystemAccessTrap(EL2, 0x18);
```

Chapter E3. System registers affected by SME E3.2. Changes to existing System registers

```
elsif HaveEL(EL3) && CPTR_EL3.TCPAC == '1' then
10
11
            if Halted() && EDSCR.SDD == '1' then
12
                 UNDEFINED;
13
            else
14
                AArch64.SystemAccessTrap(EL3, 0x18);
15
        elsif EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '111' then
            X[t, 64] = NVMem[0x100];
16
17
        else
   X[t, 64] = CPACR_EL1;
elsif PSTATE.EL == EL2 then
18
19
20
        if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & Boolean IMPLEMENTATION_DEFINED "EL3
            ↔ trap priority when SDD == '1'" && CPTR_EL3.TCPAC == '1' then
21
            UNDEFINED;
22
        elsif HaveEL(EL3) && CPTR_EL3.TCPAC == '1' then
23
            if Halted() && EDSCR.SDD == '1' then
24
                UNDEFINED;
25
            else
26
                AArch64.SystemAccessTrap(EL3, 0x18);
27
        elsif HCR_EL2.E2H == '1' then
            X[t, 64] = CPTR\_EL2;
28
29
        else
    X[t, 64] = CPACR_EL1;
elsif PSTATE.EL == EL3 then
30
31
32
        X[t, 64] = CPACR\_EL1;
```

MSR CPACR_EL1, <Xt>

	op0	op1		CRm	op2
	0b11	Овс	060001	0b0000	0b010
$\frac{1}{2}$	<pre>if PSTATE.EL == EL0 UNDEFINED:</pre>	the			
3	elsif PSTATE.EL == /	then			
4	<pre>if Halted() & .</pre>	iaveEL(EL3) && . CR.	SDD == '1' && boole	ean IMPLEMENTATIO	N DEFINED "EL3
	⇔trap 10	city hen SDD == '1'"	&& CPTR EL3.TCPAC	== '1' then	
5	UNDEF D;				
6	elsif EL2En. 'eq	& CPTP EL2.TCPAC	== '1' then		
7	AA~~h64.5	emAccess ap(EL2, 0x)	18);		
8	els' LL nable	&& FeatureImple	mented(FEAT_FGT) &	🖌 (!HaveEL(EL3) 丨	SCR_EL3.FGTEn
	∽== _') &	R_EL2.CPACR_EL	1 == '1' then		
9	AArc ¹ 4.Syst	emA cessTrap(EL2, 0x	18);		
10	sif F	&& CPTR_EL3.TCPAC	== '1' then		
11	<pre> Halted()</pre>	&& EDSCR.SDD == '1'	then		
12	UNDEFINI	ID;			
13	er				
14	rch64	SystemAccessTrap(EL3	, 0x18);		
15	<pre>elsif EL2Enabled() && HCR_EL2.<nv2,nv1,nv> == '111' then</nv2,nv1,nv></pre>				
16	NVMem[0x100] = X[t, 64];				
17	else				
18	CPACR_ELI =	X[t, 64];			
19	elsif PSTATE.EL == I	L2 then			
20	II Halted() && H	IAVELL(EL3) & EDSCR.	SUD == 'I' && DOOLO	ean IMPLEMENIAIIC	N_DEFINED "EL3
21	UNDEFINED.	ricy when SDD == .1	&& CPIR_ELS.ICPAC	== ·1· then	
21	olcif Hower (FI		!1! then		
22	if Halted()	££ FDSCR SDD == '1' '	⊥ Chen then		
24			chen		
25	else				
26	AArch64	SystemAccessTrap(EL3	. 0x18):		
27	elsif HCR EL2.E2	2H == '1' then			
28	CPTR_EL2 = 2	<[t, 64];			
29	else				
30	CPACR_EL1 =	X[t, 64];			
31	elsif PSTATE.EL == H	L3 then			

Copyright © 2022-2023 Arm Limited or its affiliates. All rights reserved. Non-confidential

32 CPACR_EL1 = X[t, 64];

MRS <Xt>, CPACR_EL12

op0	op1	CRn	CRm	op2
0b11	0b101	0b0001	0b0000	0b010

```
if PSTATE.EL == EL0 then
 1
2
        UNDEFINED;
3
   elsif PSTATE.EL == EL1 then
4
        if EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '101' then
5
            X[t, 64] = NVMem[0x100];
 6
        elsif EL2Enabled() && HCR_EL2.NV == '1' then
 7
            AArch64.SystemAccessTrap(EL2, 0x18);
8
        else
 9
            UNDEFINED;
   elsif PSTATE.EL == EL2 then
   if HCR_EL2.E2H == '1' then
10
11
            if Halted() && HaveEL(EL3) && EDSCR.SDD
                                                           '1'
                                                                 boolean 1MPLEMENTATION_DEFINED
12
                 \hookrightarrow"EL3 trap priority when SDD == '1'"
                                                              IR_EL3.T YAC == '1' then
13
                UNDEFINED;
14
            elsif HaveEL(EL3) && CPTR_EL3.TC
                                                   1' th.
                                               1' the
15
                 if Halted() && EDSCR.SDD =-
16
                     UNDEFINED;
17
                 else
18
                     AArch64.SystemAcc ssTrap
                                                   , 0x18);
19
            else
20
                X[t, 64] = CPACR \_1;
21
        else
22
            UNDEFINED;
    elsif PSTATE.EL == EL3 .en
23
        if EL2Enabled() <& !ELUsing rch32(EL2) && HCR_EL2.E2H == '1' then
24
25
                         ACR_EL1;
            X[t, 64] =
26
        else
27
            UNDEFIN ,;
```

MSR CPACR_EL'. .t>

	op1	CRn	CRm	op2	
0b11	0b101	0b0001	0b0000	0b010	

```
1
   if PSTATE.EL == EL0 then
2
       UNDEFINED;
3
   elsif PSTATE.EL == EL1 then
4
       if EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '101' then
           NVMem[0x100] = X[t, 64];
5
       elsif EL2Enabled() && HCR_EL2.NV == '1' then
6
7
           AArch64.SystemAccessTrap(EL2, 0x18);
8
       else
9
           UNDEFINED;
10
   elsif PSTATE.EL == EL2 then
       if HCR_EL2.E2H == '1' then
11
12
            if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED
                ↔"EL3 trap priority when SDD == '1'" && CPTR_EL3.TCPAC == '1' then
13
                UNDEFINED;
14
            elsif HaveEL(EL3) && CPTR_EL3.TCPAC == '1' then
15
                if Halted() && EDSCR.SDD == '1' then
16
                    UNDEFINED;
```

Copyright © 2022-2023 Arm Limited or its affiliates. All rights reserved. Non-confidential

Chapter E3. System registers affected by SME E3.2. Changes to existing System registers

17	else
18	AArch64.SystemAccessTrap(EL3, 0x18);
19	else
20	$CPACR_EL1 = X[t, 64];$
21	else
22	UNDEFINED;
23	elsif PSTATE.EL == EL3 then
24	<pre>if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then</pre>
25	$CPACR_EL1 = X[t, 64];$
26	else
27	UNDEFINED;

E3.2.2 CPTR_EL2, Architectural Feature Trap Register (EL2)

The CPTR_EL2 characteristics are:

Purpose

Controls trapping to EL2 of accesses to CPACR, CPACR_EL1, trace, Activity Monitor, SME, Streaming SVE, SVE, and Advanced SIMD and floating-point functionality.

Configuration

If EL2 is not implemented, this register is RESO from EL3.

This register has no effect if EL2 is not enabled in the current Security state.

AArch64 system register CPTR_EL2 bits [31:0] are architecturally mapped to AArch32 system register HCPTR[31:0].

Attributes

CPTR_EL2 is a 64-bit register.

Field descriptions

The CPTR_EL2 bit assignments are:

When FEAT_VHE is implemented and HC^r_EL2.E 1 == 1.



Bits [63:32]

Reserved, RESO.

TCPAC Jit ['s

In A ch64 ccesses to CPACR_EL1 from EL1 to EL2, when EL2 is enabled in the current Security state. 1 cception is reported using ESR_ELx.EC value 0x18.

In AArch32 ate, traps accesses to CPACR from EL1 to EL2, when EL2 is enabled in the current Security state. The exception \therefore reported using ESR_ELx.EC value 0x03.

TCPAC	Meaning
0b0	This control does not cause any instructions to be trapped.
0b1	EL1 accesses to CPACR_EL1 and CPACR are trapped to EL2, when EL2 is enabled in the current Security state.

When HCR_EL2.TGE is 1, this control does not cause any instructions to be trapped.

CPACR_EL1 and CPACR are not accessible at EL0.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

TAM, bit [30]

When FEAT_AMUv1 is implemented:

Trap Activity Monitor access. Traps EL1 and EL0 accesses to all Activity Monitor registers to EL2, as follows:

- In AArch64 state, accesses to the following registers are trapped to EL2, reported using ESR_ELx.EC value 0x18:
 - AMUSERENR_EL0, AMCFGR_EL0, AMCGCR_EL0, AMCNTENCLR0_EL0, AMCNTENCLR1_EL0, AMCNTENSET0_EL0, AMCNTENSET1_EL0, AMCR_EL0, AMEVCNTR0<n>_EL0, AMEVCNTR1<n>_EL0, AMEVCNTR1<n]
- In AArch32 state, MRC or MCR accesses to the following regist are trapped > EL2 and reported using ESR_ELx.EC value 0x03:
 - AMUSERENR, AMCFGR, AMCGCR, AMC 'ENCL'), AM, 'ENCLR1, AMCNTENSET0, AMCNTENSET1, AMCR, AMEVTYPER0<n. and A' EVTYPER1<n>.
- In AArch32 state, MRRC or MCRR accesses `MEV、 TR0<n> .id AMEVCNTR1<n>, are trapped to EL2, reported using ESR_ELx.EC value 0 .4.

- M	Meaning
Öx	Accesses from EL1 and EL0 to Activity Monitor registers are not trapped.
Jb1	Accesses from EL1 and EL0 to Activity Monitor registers are trapped to EL2, when EL2 is enabled in the current Security state.

The reset behavior field is:

• O' A War, reset, fif resets to an architecturally UNKNOWN value.

Otl wise:

res0

Bit [29]

Reserved, RESO.

TTA, bit [28]

Traps System register accesses to all implemented trace registers from both Execution states to EL2, when EL2 is enabled in the current Security state, as follows:

- In AArch64 state, accesses to trace registers with op0=2, op1=1, and CRn<0b1000 are trapped to EL2, reported using EC syndrome value 0x18.
- In AArch32 state, MRC or MCR accesses to trace registers with cpnum=14, opc1=1, and CRn<0b1000 are trapped to EL2, reported using EC syndrome value 0x05.

Meaning
This control does not cause any instructions to be trapped.
Any attempt at EL0, EL1 or EL2, to execute a System register access to an implemented trace register is trapped to EL2, when EL2 is enabled in the current Security state, unless HCR_EL2.TGE is 0 and it is trapped by CPACR.NSTRCDIS or CPACR_EL1.TTA. When HCR_EL2.TGE is 1, any attempt at EL0 or EL2 to execute a System register access to an implemered access to an implemered access to an implemered access in the current Security

The ETMv4 architecture and ETE do not permit EL0 to *z* less the *t* lice reg. If the trace unit implements FEAT_ETMv4 or ETE, EL0 accesses to the trace register. The UNT_FINED, and any resulting exception is higher priority than an exception that would be generated because unit are of CPT_LEL2.TTA is 1.

EL2 does not provide traps on trace register accesses the 1gh the stice. Memory-mapped interface.

System register accesses to the trace register an have the effects. When a System register access is trapped, any side-effects that are normally associated with the access do not excur before the exception is taken.

If System register access to the trace functionality a pot supported, this bit is RESO.

The reset behavior of this field :

• On a Warm reset, this eld re. 's to an archive turally UNKNOWN value.

Bits [27:26]

Reserved, RESO.

SMEN, bi+~ "25:24)

Whe *c*'EAT_S IE is imp. .ented:

Traps <u>cr</u> on at EL2, L1, and EL0 of SME instructions, SVE instructions when FEAT_SVE is not implemented or the PL in Streaming SVE mode, and instructions that directly access the SVCR, SMCR_EL1, or SMCR_EL2 System reg. rs to EL2, when EL2 is enabled in the current Security state.

When instructions that directly access the SVCR System register are trapped with reference to this control, the MSR SVCRSM, MSR SVCRZA, and MSR SVCRSMZA instructions are also trapped.

The exception is reported using ESR_EL2.EC value of 0x1D, with an ISS code of 0x0000000.

This field does not affect whether Streaming SVE or SME register values are valid.

A trap taken as a result of CPTR_EL2.SMEN has precedence over a trap taken as a result of CPTR_EL2.FPEN.

SMEN	Meaning
0b00	This control causes execution of these instructions at EL2, EL1, and EL0 to be trapped.

SMEN	Meaning
0b01	When HCR_EL2.TGE is 0, this control does not cause execution of any instructions to be trapped. When HCR_EL2.TGE is 1, this control causes execution of these instructions at EL0 to be trapped, but does not cause execution of any instructions at EL2 to be trapped.
0b10	This control causes execution of these instructions at EL2, EL1, and EL0 to be trapped.
0b11	This control does not cause execution of any instructions to be trapped.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally U' .NOWN va.

Otherwise:

res0

Bits [23:22]

Reserved, RESO.

FPEN, bits [21:20]

Traps execution at EL2, EL1, $e \in EL0$ of instructions that access the Advanced SIMD and floating-point registers from both Execution states EL_2 when EL2 is called in the current Security state. The exception is reported using ESR_ELx.EC value 0x07.

Traps execution at F'_{-} , EL1, and EL0 of 1E and SVE instructions to EL2, when EL2 is enabled in the current Security state. T^{*k*} exceptions reported using ESR_ELx.EC value 0x07.

A trap taken as a real to CPTR_F' 2.SMEN has precedence over a trap taken as a result of CPTR_EL2.FPEN.

A trap tran as result CPT _EL2.ZEN has precedence over a trap taken as a result of CPTR_EL2.FPEN.

FPEN	Meaning
0600	This control causes execution of these instructions at EL2, EL1, and EL0 to be trapped.
0b01	 When HCR_EL2.TGE is 0, this control does not cause execution of any instructions to be trapped. When HCR_EL2.TGE is 1, this control causes execution of these instructions at EL0 to be trapped, but does not cause execution of any instructions at EL2 to be trapped.
0b10	This control causes execution of these instructions at EL2, EL1, and EL0 to be trapped.
0b11	This control does not cause execution of any instructions to be trapped.

Writes to MVFR0, MVFR1, and MVFR2 from EL1 or higher are CONSTRAINED UNPREDICTABLE and whether
these accesses can be trapped by this control depends on implemented CONSTRAINED UNPREDICTABLE behavior.

- Attempts to write to the FPSID count as use of the registers for accesses from EL1 or higher.
- Accesses from EL0 to FPSID, MVFR0, MVFR1, MVFR2, and FPEXC are UNDEFINED, and any resulting exception is higher priority than an exception that would be generated because the value of CPTR_EL2.FPEN is not 0b11.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [19:18]

Reserved, RESO.

ZEN, bits [17:16]

When FEAT_SVE is implemented:

Traps execution at EL2, EL1, and EL0 of SVE instructions y en the F 's not in reaming SVE mode, and instructions that directly access the ZCR_EL1 or ZCR_EL2 ystem registers E¹, when EL2 is enabled in the current Security state.

The exception is reported using ESR_ELx.EC value 0x19.

A trap taken as a result of CPTR_EL2.ZEN has .ecede e over "a" .ken as a result of CPTR_EL2.FPEN.

ĿN	Meaning
0.)	This control causes execution of these instructions at EL2, EL1, and EL0 to be trapped.
JD01	 When HCR_EL2.TGE is 0, this control does not cause execution of any instructions to be trapped. When HCR_EL2.TGE is 1, this control causes execution of these instructions at EL0 to be trapped, but does not cause execution of any instructions at EL2 to be trapped.
0b10	This control causes execution of these instructions at EL2, EL1, and EL0 to be trapped.
0b11	This control does not cause execution of any instructions to be trapped.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

res0

Chapter E3. System registers affected by SME E3.2. Changes to existing System registers

Bits [15:0]

Reserved, RESO.

Otherwise:



This format applies in all Armv8.0 implementations.

Bits [63:32]

Reserved, RESO.

TCPAC, bit [31]

In AArch64 state, traps accesses to CPACR_EL1 ... L1 to 2, where L2 is enabled in the current Security state. The exception is reported using ESR_EJ ... C val 0x18.

In AArch32 state, traps accesses to CPACR m EL ben EL2 is enabled in the current Security state. The exception is reported using ESR_F x.EC v Jx03.

ТСі АС	Meaning
060	This control does not cause any instructions to be trapped.
0b1	 EL1 accesses to the following registers are trapped to EL2, when EL2 is enabled in the current Security state: CPACR_EL1. CPACR.

When HCk_ L2.TGE is 1, this control does not cause any instructions to be trapped.

CPACR_EL1 and CPACR are not accessible at EL0.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

TAM, bit [30]

When FEAT_AMUv1 is implemented:

Trap Activity Monitor access. Traps EL1 and EL0 accesses to all Activity Monitor registers to EL2, as follows:

• In AArch64 state, accesses to the following registers are trapped to EL2, reported using ESR_ELx.EC value 0x18:

– AMUSERENR_EL0,	AMCFGR_EL0,	AMCGCR_EL0,	AMCNTE	NCLR0_EL0,
AMCNTENCLR1_EL0,	AMCNTENSET0_EI	L0, AMCNTENSET1	_EL0,	AMCR_EL0,

AMEVCNTR0<n>_EL0,AMEVCNTR1<n>_EL0,AMEVTYPER0<n>_EL0,andAMEVTYPER1<n>_EL0.

- In AArch32 state, MCR or MRC accesses to the following registers are trapped to EL2 and reported using ESR_ELx.EC value 0x03:
 - AMUSERENR, AMCFGR, AMCGCR, AMCNTENCLR0, AMCNTENCLR1, AMCNTENSET0, AMCNTENSET1, AMCR, AMEVTYPER0<n>, and AMEVTYPER1<n>.
- In AArch32 state, MCRR or MRRC accesses to AMEVCNTR0<n> and AMEVCNTR1<n>, are trapped to EL2, reported using ESR_ELx.EC value 0x04.

Meaning
Accesses on a and EL0 to Activity Monitor regist of are not traged.
Acces from EL1 a EL0 to Activity Monitor

The reset behavior of this field is:

• On a Warm reset, this field resets to an a* intecturary UNKN value.

Otherwise:

res0

Bits [29:21]

Reserved, RESO.

TTA, bit [20]

Traps System re_{ξ} for accrete sto all implemented trace registers from both Execution states to EL2, when EL2 is enabled in the curre Γ curity stat as follows:

- In Arch, ¹ state, ¹ ces's to trace registers with op0=2, op1=1, and CRn<0b1000 are trapped to EL2, ported v ng EC sy, ome value 0x18.
- . AA n32 state, ARC or MCR accesses to trace registers with cpnum=14, opc1=1, and CRn<0b1000 are trap. 4 to EL2, reported using EC syndrome value 0x05.

ТТА	Meaning
0b0	This control does not cause any instructions to be trapped.
0b1	Any attempt at EL0, EL1, or EL2, to execute a System register access to an implemented trace register is trapped to EL2, when EL2 is enabled in the current Security state, unless it is trapped by one of the following controls: • CPACR_EL1.TTA. • CPACR.TRCDIS.

- The ETMv4 architecture does not permit EL0 to access the trace registers. If the trace unit implements FEAT_ETMv4, EL0 accesses to the trace registers are UNDEFINED, and any resulting exception is higher priority than an exception that would be generated because the value of CPTR_EL2.TTA is 1.
- EL2 does not provide traps on trace register accesses through the optional memory-mapped interface.

System register accesses to the trace registers can have side-effects. When a System register access is trapped, any side-effects that are normally associated with the access do not occur before the exception is taken.

If System register access to the trace functionality is not supported, this bit is RESO.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [19:14]

Reserved, RESO.

Bit [13]

Reserved, RES1.

TSM, bit [12]

When FEAT_SME is implemented:

Traps execution at EL2, EL1, and EL0 of SMF instruction, SVE insul ctions when FEAT_SVE is not implemented or the PE is in Streaming SVE mode, and insult tions in the SVCR, SMCR_EL1, or SMCR_EL2 System registers to EL2, when EL2 is children in the current Security state.

When instructions that directly ac ss the 'CR Sy m register are trapped with reference to this control, the MSR SVCRSM, MSR SVCRSM, and SVCRSMZA, tructions are also trapped.

The exception is reported v ing ES. EL2.EC value of 0x1D, with an ISS code of 0x0000000.

This field does not aff whether Strea. ng SVE or SME register values are valid.

A trap taken as a *i* ult of CF 'R_EL2.TSM has precedence over a trap taken as a result of CPTR_EL2.TFP.

TSM	Meaning
060	This control does not cause execution of any instructions to be trapped.
0b1	This control causes execution of these instructions at EL2, EL1, and EL0 to be trapped.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

res1

Bit [11]

Reserved, RESO.

TFP, bit [10]

Traps execution of instructions which access the Advanced SIMD and floating-point functionality, from both

Execution states to EL2, when EL2 is enabled in the current Security state, as follows:

- In AArch64 state, accesses to the following registers are trapped to EL2, reported using ESR_ELx.EC value 0x07:
 - FPCR, FPSR, FPEXC32_EL2, any of the SIMD and floating-point registers V0-V31, including their views as D0-D31 registers or S0-31 registers.
- In AArch32 state, accesses to the following registers are trapped to EL2, reported using ESR_ELx.EC value 0x07:
 - MVFR0, MVFR1, MVFR2, FPSCR, FPEXC, and any of the SIMD and floating-point registers Q0-15, including their views as D0-D31 registers or S0-31 registers. For the purposes of this trap, the architecture defines a VMSR access to FPSID from EL1 or higher as an access to a SIMD and floating-point register. Otherwise, permitted VMSR accesses to FPSID are ignored.

Traps execution at the same Exception levels of SME and SVE instructions to EL2, when EL2 is enabled in the current Security state. The exception is reported using ESR_ELx.EC values of the exception.

A trap taken as a result of CPTR_EL2.TSM has precedence over a tr taken as a res t of CPTR_EL2.TFP.

A trap taken as a result of CPTR_EL2.TZ has precedence over *c* ap tak as a result *i* CPTR_EL2.TFP.



FPEXC32_EL2 is not accessib1 rom EL0 us. AArc.104.

FPSID, MVFR0, MVFR1, ...d FP1 C are not acc .ssible from EL0 using AArch32.

The reset behavior of t' s field is:

• On a Warm - .et, this f 'd resets to an architecturally UNKNOWN value.

Bit [9]

Reser J, RES1

TZ, '[8]

When FL. SVE is implemented:

Traps executio. at EL2, EL1, and EL0 of SVE instructions when the PE is not in Streaming SVE mode, and instructions that directly access the ZCR_EL2 or ZCR_EL1 System registers to EL2, when EL2 is enabled in the current Security state.

The exception is reported using ESR_ELx.EC value 0x19.

A trap taken as a result of CPTR_EL2.TZ has precedence over a trap taken as a result of CPTR_EL2.TFP.

TZ	Meaning
0b0	This control does not cause execution of any instructions to be trapped.
0b1	This control causes execution of these instructions at EL2, EL1, and EL0 to be trapped.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

res1

Bits [7:0]

Reserved, RES1.

Accessing CPTR_EL2

Accesses to this register use the following encodings in the System register encoding space:

MRS <Xt>, CPTR_EL2

	орО	op1	CRr	CRm	op2
	0b11	0b100	<u>`0001</u>	066.501	0b010
if PSTATE.E	L == ELO then				
onderin alcif pstat	E FI. == FI.1 +h				
if EL2E	nabled() && HC	'R EL2.NV == '1			
AAr	ch64.SvstemAcc	essTrap() . Ox)			
else					
UND	EFINED;				
elsif PSTAT	E.EL == EL2 th				
if Halt ∽t UND	ed() && Have [,] rap priority w EFINED;	(EL && EDSCR.S. when s == '1'" a	== '1' && boolea && CPTR_EL3.TCPAC =	an IMPLEMENTATIO == '1' then	N_DEFINED "EL3
elsif H	aveEL(5) &&	CPTR_EL3. PAC ==	= '1' then		
if	Halt () && E	SCR.SDD == 1' th	hen		
	Ur (FINED;				
eis	e		010) -		
olar	AArch Syste	emac ssirap(ELS,	UX18);		
V[+	1 = C + Z				
els: PSTAT	F = EL3	en			
101M1	FL2:				
5, 0					

```
MSR CP. `EL2, <Xt>
```

орО	op1	CRn	CRm	op2
0b11	0b100	0b0001	0b0001	0b010

```
1
  if PSTATE.EL == EL0 then
2
      UNDEFINED;
3
   elsif PSTATE.EL == EL1 then
4
      if EL2Enabled() && HCR_EL2.NV == '1' then
          AArch64.SystemAccessTrap(EL2, 0x18);
5
6
       else
          UNDEFINED;
7
  elsif PSTATE.EL == EL2 then
8
9
      if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3
          →trap priority when SDD == '1'" && CPTR_EL3.TCPAC == '1' then
```

Chapter E3. System registers affected by SME E3.2. Changes to existing System registers

```
10
           UNDEFINED;
        elsif HaveEL(EL3) && CPTR_EL3.TCPAC == '1' then
11
12
             if Halted() && EDSCR.SDD == '1' then
                 UNDEFINED;
13
14
             else
15
                 AArch64.SystemAccessTrap(EL3, 0x18);
16
        else
    CPTR_EL2 = X[t, 64];
elsif PSTATE.EL == EL3 then
17
18
19
        CPTR\_EL2 = X[t, 64];
```

MRS <Xt>, CPACR_EL1

		op0	op1	CRn	CDη	op2
		0b11	0b000	0b0001	0Ь000с	0b010
1	if PSTATE.EL ==	ELO then				
2	ONDEFINED;	FI1 +b	~			
4	if Halted()	ELI CH && HaveEL	(EL3) && EDSCR SDF) == '1 '& vol	an TOLEMENTATI	ON DEFINED "EL3
-	→t.rap	priority w	nhen SDD == '1'" &	TR EL CPAC	== /' then	on_burindb blo
5	UNDEFIN	IED;				
6	elsif EL2En	abled() &&	CPTR_EL2.TCP/ ==	'1' :hen		
7	AArch64	.SystemAcc	essTrap(EL2, x18)	;		
8	elsif EL2En	abled() &&	IsFeatureImp men	1 . (E LI) &	🖌 (!HaveEL(EL3)	SCR_EL3.FGTEn
	↔== '1	') && HFGR	TR_EL2. \CR_EL	\cdot '1' then		
9	AArch64	.SystemAcc	essTrap(L. 0x1b			
10	elsif HaveE	L(EL3) &&	CPTT LL3.TC. ~ ==	then		
11	if Halt	ed() && ED	SC .SDD == '1 'he	en 🔪		
12	UNE	EFINED;				
13	else					
14	AAr	ch64.°vste	mAcces rap(EL3, 0	X18);		
15	eisii elzen Vit ca	able () & &	HCR_ELZ NVZ,NVI,	NV> == '111' th	en	
10	ALL, 04		UX100];			
18	V [+)	= CPZ	FT.1 •			
19	elsif PSTATE EI	F th	en			
20	if Ha	& aveEL	(E' && EDSCR.SDE	== '1' && bool	an IMPLEMENTATI	ON DEFINED "EL3
	→tr.	prio. V W	" SDD == '1'" &8	CPTR EL3.TCPAC	== '1' then	
21	UNDEF	iED;				
22	1sif Ha .F	L(EL3) &u	CPTR_EL3.TCPAC ==	'1' then		
23	if are	L ED	SCR.SDD == '1' the	n		
24	UNE	EFINED;				
25	'e					
26	٦Ar	ch64.Syste	mAccessTrap(EL3, C	x18);		
27	elsif HCE	L2.E2H ==	'1' then			
28	X[t, 64] = CPTR_E	L2;			
29	else					
30	X[t, 64	J = CPACR_	EL1;			
31	elsif PSTATE.EL	. == EL3 th	en			
32	X[t, 64] =	CPACK_EL1;				

MSR CPACR_EL1, <Xt>

ор0	op1	CRn	CRm	op2
0b11	0b000	0b0001	0b0000	0b010

1 if PSTATE.EL == EL0 then

```
2
       UNDEFINED;
3
   elsif PSTATE.EL == EL1 then
4
        if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & Boolean IMPLEMENTATION_DEFINED "EL3
            ↔ trap priority when SDD == '1'" && CPTR_EL3.TCPAC == '1' then
5
            UNDEFINED;
6
        elsif EL2Enabled() && CPTR_EL2.TCPAC == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
7
8
        elsif EL2Enabled() & IsFeatureImplemented(FEAT_FGT) & (!HaveEL(EL3) || SCR_EL3.FGTEn
            ↔== '1') && HFGWTR_EL2.CPACR_EL1 == '1' then
9
            AArch64.SystemAccessTrap(EL2, 0x18);
10
        elsif HaveEL(EL3) && CPTR_EL3.TCPAC == '1' then
            if Halted() && EDSCR.SDD == '1' then
11
12
                UNDEFINED;
13
            else
14
                AArch64.SystemAccessTrap(EL3, 0x18);
15
        elsif EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '111' then
           NVMem[0x100] = X[t, 64];
16
17
        else
18
            CPACR\_EL1 = X[t, 64];
   elsif PSTATE.EL == EL2 then
19
20
        if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && pole.
                                                                    TMPLEMENT
                                                                               ION_DEFINED "EL3
            →trap priority when SDD == '1'" && CPTR_EL<sup>^</sup> .CPAC =
                                                                         ther
21
            UNDEFINED:
22
        elsif HaveEL(EL3) && CPTR_EL3.TCPAC == '1' + .n
23
            if Halted() && EDSCR.SDD == '1' then
24
                UNDEFINED;
25
            else
26
                AArch64.SystemAccessTrap(EL<sup>2</sup> 0x18)
        elsif HCR_EL2.E2H == '1' then
27
28
           CPTR\_EL2 = X[t, 64];
29
        else
30
            CPACR\_EL1 = X[t, 64];
   elsif PSTATE.EL == EL3 then
31
32
       CPACR\_EL1 = X[t, 64];
```

E3.2.3 CPTR_EL3, Architectural Feature Trap Register (EL3)

The CPTR_EL3 characteristics are:

Purpose

Controls trapping to EL3 of accesses to CPACR, CPACR_EL1, HCPTR, CPTR_EL2, trace, Activity Monitor, SME, Streaming SVE, SVE, and Advanced SIMD and floating-point functionality.

Configuration

This register is present only when EL3 is implemented. Otherwise, direct accesses to CPTR_EL3 are UNDEFINED.

Attributes

CPTR_EL3 is a 64-bit register.

Field descriptions

The CPTR_EL3 bit assignments are:



Bits [63:32]

Reserved, RESO.

TCPAC, bit [31]

- EL2 accesses <u>PTR_EL</u> reported using ESR_ELx.EC value 0x18, or HCPTR, reported using ESP_____ EC va. 0x03
- F 2 and F 1 acces. \therefore CPACR_EL1 reported using ESR_ELx.EC value 0x18, or CPACR reported using \therefore SR_EL \Rightarrow C value 0x03.

When \checkmark _EL3.TCPAC is:

TCPAC	Meaning
060	This control does not cause any instructions to be trapped.
0b1	EL2 accesses to the CPTR_EL2 or HCPTR, and EL2 and EL1 accesses to the CPACR_EL1 or CPACR, are trapped to EL3, unless they are trapped by CPTR_EL2.TCPAC.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

TAM, bit [30]

When FEAT_AMUv1 is implemented:

Trap Activity Monitor access. Traps EL2, EL1, and EL0 accesses to all Activity Monitor registers to EL3.

Accesses to the Activity Monitors registers are trapped as follows:

- In AArch64 state, the following registers are trapped to EL3 and reported with ESR_ELx.EC value 0x18:
 - AMUSERENR_EL0, AMCFGR_EL0, AMCGCR_EL0, AMCNTENCLR0_EL0, AMCNTENCLR1_EL0, AMCNTENSET0_EL0, AMCNTENSET1_EL0, AMCR_EL0, AMEVCNTR0<n>_EL0, AMEVCNTR1<n>_EL0, AMEVTYPER0<n>_EL0, and AMEVTYPER1<n>_EL0.
- In AArch32 state, accesses with MRC or MCR to the following regist ported with ESR_ELx.EC value 0x03:
 - AMUSERENR, AMCFGR, AMCGCR, AMCNTENCL. AMCNTEN LR1, AMCNTENSET0, AMCNTENSET1, AMCR, AMEVTYPER0<n>, and MEV. PER1<n>.
- In AArch32 state, accesses with MRRC or MCRR to the following re_{ϵ} ter reported with ESR_ELx.EC value 0x04:
 - AMEVCNTR0<n>, AMEVCNTR1<n>.

TAP	Meing
00.	Accesses from EL2, EL1, and EL0 to Activity Monitor registers are not trapped.
0b1	Accesses from EL2, EL1, and EL0 to Activity Monitor registers are trapped to EL3.

The reset behavior conis field is:

• On a Warn. set, th: ...eld resets to an architecturally UNKNOWN value.

Otherwise[•]

res0

2: ר Bits

Reserved, K

TTA, bit [20]

Traps System register accesses. Accesses to the trace registers, from all Exception levels, any Security state, and both Execution states are trapped to EL3 as follows:

- In AArch64 state, Trace registers with op0=2, op1=1, and CRn<0b1000 are trapped to EL3 and reported using EC syndrome value 0x18.
- In AArch32 state, accesses using MCR or MRC to the Trace registers with cpnum=14, opc1=1, and CRn<0b1000 are reported using EC syndrome value 0x05.

TTA	Meaning
0b0	This control does not cause any instructions to be trapped.

TTA	Meaning
0b1	Any System register access to the trace registers is trapped to EL3, unless it is trapped by CPACR.TRCDIS, CPACR_EL1.TTA, or CPTR_EL2.TTA.

If System register access to trace functionality is not supported, this bit is RESO.

The ETMv4 architecture and ETE do not permit EL0 to access the trace registers. If the trace unit implements FEAT_ETMv4 or FEAT_ETE, EL0 accesses to the trace registers are UNDEFINED, and any resulting exception is higher priority than this trap exception.

EL3 does not provide traps on trace register accesses through the Meme _-mappe_ terface.

System register accesses to the trace registers can have side-effects. Von a System register access is trapped, no side-effects occur before the exception is taken, see 'Configurab' instruction controls'

The reset behavior of this field is:

• On a Warm reset, this field resets to an architectural. UNKN AN value.

Bits [19:13]

Reserved, RESO.

ESM, bit [12]

When FEAT_SME is implemen

Traps execution of SME ir _uc_ns, SVE ins. _ctions when FEAT_SVE is not implemented or the PE is in Streaming SVE mode, and inst__ctions that directly access the SMCR_EL1, SMCR_EL2, SMCR_EL3, SMPRI_EL1, SMPRI_AP_EL2, or Sv__R System registers, from all Exception levels and any Security state, to EL3.

When instruction. that discuss the SVCR System register are trapped with reference to this control, the MSR SVCRSM. MSR SV. A, and MSF VCRSMZA instructions are also trapped.

When f ect a cases to M^* d_EL1 and SMPRIMAP_EL2 are trapped, the exception is reported using an ESP $\pm L3.EC^*$ lue of 0x1c. Otherwise, the exception is reported using an ESR_EL3.EC value of 0x1D, with an ISS c^* e of f and c

This field, bes not affect whether Streaming SVE or SME register values are valid.

A trap taken a result of CPTR_EL3.ESM has precedence over a trap taken as a result of CPTR_EL3.TFP.

ESM	Meaning
0b0	This control causes execution of these instructions at all Exception levels to be trapped.
0b1	This control does not cause execution of any instructions to be trapped.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

res0

Bit [11]

Reserved, RESO.

TFP, bit [10]

Traps execution of instructions which access the Advanced SIMD and floating-point functionality, from all Exception levels, any Security state, and both Execution states, to EL3.

This includes the following registers, all reported using ESR_ELx.EC value 0x07:

- FPCR, FPSR, FPEXC32_EL2, and any of the SIMD and floating-point registers V0-V31, including their views as D0-D31 registers or S0-S31 registers.
- MVFR0, MVFR1, MVFR2, FPSCR, FPEXC, and any of the SIV and the ing-point registers Q0-Q15, including their views as D0-D31 registers or S0-S31 registers.
- VMSR accesses to FPSID.

Permitted VMSR accesses to FPSID are ignored, but for the provided of the optime chitecture defines a VMSR access to the FPSID from EL1 or higher as an access to a $^{\circ}$ \mathcal{AD} and $^{\circ}$ ating- $_{\rm F}$ register.

Traps execution at all Exception levels of SME and SVE instatic to EL3 from any Security state. The exception is reported using ESR_ELx.EC value 0x07.

A trap taken as a result of CPTR_EL3.ESM h^r precede :e over a taken as a result of CPTR_EL3.TFP.

A trap taken as a result of CPTR_EL3.EZ has reced approximately approxim

Defined values are:

TFP	Meaning
060	This control does not cause execution of any instructions to be trapped.
0b1	This control causes execution of these instructions at all Exception levels to be trapped.

FPEV 32_EL2 s not acc sle from EL0 using AArch64.

FPS1. MV ..., and FPEXC are not accessible from EL0 using AArch32.

The reset vavior of this field is:

• On a Wa. A reset, this field resets to an architecturally UNKNOWN value.

Bit [9]

Reserved, RESO.

EZ, bit [8]

When FEAT_SVE is implemented:

Traps execution of SVE instructions when the PE is not in Streaming SVE mode, and instructions that directly access the ZCR_EL3, ZCR_EL2, or ZCR_EL1 System registers, from all Exception levels and any Security state, to EL3.

The exception is reported using ESR_ELx.EC value 0x19.

A trap taken as a result of CPTR_EL3.EZ has precedence over a trap taken as a result of CPTR_EL3.TFP.

EZ	Meaning
0b0	This control causes execution of these instructions at all Exception levels to be trapped.
0b1	This control does not cause execution of any instructions to be trapped.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

res0

Bits [7:0]

Reserved, RESO.

Accessing CPTR_EL3

Accesses to this register use the following encoding space: "he System register are needing space:

MRS <Xt>, CPTR_EL3

	op0	op1	CRn	CRm	op2
	0b11	0b110	0b0001	0b0001	0b010
1 if PSTATE.E 2 UNDEFIN	L == F' then HED:				
3 elsif PSTAT	E = EL1	en			
4 UNDEFIN	IED;				
5 elsif PST ^A	T = L2 the	en			
0 UND IN 7 eleif STAT		2			
8 1t, 64	CPTR ELS	,11			
MSR C. í	EL3. <xt></xt>				
	,				
	op0	op1	CRn	CRm	op2
	0b11	0b110	0b0001	0b0001	0b010

```
if PSTATE.EL == ELO then
1
       UNDEFINED;
2
3
   elsif PSTATE.EL == EL1 then
4
      UNDEFINED;
5
   elsif PSTATE.EL == EL2 then
6
       UNDEFINED;
7
   elsif PSTATE.EL == EL3 then
     CPTR\_EL3 = X[t, 64];
8
```

E3.2.4 FAR_EL1, Fault Address Register (EL1)

The FAR_EL1 characteristics are:

Purpose

Holds the faulting Virtual Address for all synchronous Instruction Abort exceptions, Data Abort exceptions, PC alignment fault exceptions and Watchpoint exceptions that are taken to EL1.

Configuration

AArch64 system register FAR_EL1 bits [31:0] are architecturally mapped to AArch32 system register DFAR31:0.

AArch64 system register FAR_EL1 bits [63:32] are architecturally mapped to AArch32 system register IFAR31:0.

Attributes

FAR_EL1 is a 64-bit register.

Field descriptions

The FAR_EL1 bit assignments are:



Bits [63:0]

Faulting Virtual Address for synchrone exceptions taken to EL1. Exceptions that set the FAR_EL1 are Instruction Aborts (EC 0x20 or 0.1), Data Aborts (-0x24 or 0x25), PC alignment faults (EC 0x22), and Watchpoints (EC 0x34 or 0x35). ES _EL1.EC holds the EC value for the exception.

For a synchronou $\exists xt \\ d a bort if the VA that generated the abort was from an address range for which TCR_ELx <math>\exists x \\ d b \\$

For synchron we External abort other than a synchronous External abort on a translation table walk, this field is valid v_{i} $SR_{EL1...}$ is 0, and FAR_EL1 is UNKNOWN if ESR_EL1.FnV is 1.

If a memory fault that sets FAR_EL1, other than a Tag Check Fault, is generated from a data cache maintenance or other DC instruction, this field holds the address specified in the register argument of the instruction.

On an exception due to a Tag Check Fault caused by a data cache maintenance or other DC instruction, the address held in FAR_EL1 is IMPLEMENTATION DEFINED as one of the following:

- The lowest address that gave rise to the fault.
- The address specified in the register argument of the instruction as generated by MMU faults caused by DC ZVA.

If the exception that updates FAR_EL1 is taken from an Exception level using AArch32, the top 32 bits are all zero, unless both of the following apply, in which case the top 32 bits of FAR_ELx are 0x00000001:

- The faulting address was generated by a load or store instruction that sequentially incremented from address 0xFFFFFFF. Such a load or store instruction is CONSTRAINED UNPREDICTABLE.
- The implementation treats such incrementing as setting bit[32] of the virtual address to 1.

When the PE sets ESR_EL1.{ISV,FnP} to $\{0,1\}$ on taking a Data Abort exception, or sets ESR_EL1.{FnV,FnP} to $\{0,1\}$ on taking a Watchpoint exception, the PE sets FAR_EL1 to any address within the naturally-aligned fault granule that contains the virtual address of the memory access that generated the Data Abort exception or Watchpoint exception.

The naturally-aligned fault granule is one of:

- When ESR_EL1.DFSC is 0b010001, indicating a Synchronous Tag Check fault, it is a 16-byte tag granule.
- When ESR_EL1.DFSC is 0b11010x, indicating an IMPLEMENTATION DEFINED fault, it is an IMPLEMENTA-TION DEFINED granule.
- Otherwise, it is the smallest implemented translation granule.

When FEAT_MOPS is implemented, the value in FAR_EL1 on a synchronous exception from any of the Memory Copy and Memory Set instructions represents the first element that has not been copied or set, and is determined as follows:

- For a Data Abort generated by the MMU, the value is within the address range of the relevant translation granule, aligned to the size of the relevant translation granule of the value are UNKNOWN, where 2^n is the relevant constant of the value size in bytes. For the purpose of calculating the relevant translation granule, if the MMU is considered to a stage of translation, then the current translation granule size is equal to 2^{64} for stage 1. Indicate the relevant translation granule is:
 - For MMU faults generated at stage 1, the pent sta, 1 trans¹ on granule.
 - For MMU faults generated at stage the siller of the rent stage 1 translation granule and the current stage 2 translation granule
 - If FEAT_RME is implemented, for a mchr sous construction ort generated as the result of a GPF, the smallest of the current stage 1 translation granule and the configured granule size in GPCCR_F_3.Pc
- For a Data Abort generate by a Tag Che. failure, the value is the lowest address that failed the Tag Check within the block size of the 1 d or store.
- For a Watchpoint ception, the 'ue is an address range of the size defined by the DCZID_EL0.BS field. This address de a not need to be the 'ement with a watchpoint, but can be some earlier element.
- Otherwise, e value ' the lowest address in the block size of the load or store.

For a Data Abort exc. on or Wa' point exception, if address tagging is enabled for the address accessed by the data accession, aused becching, then this field includes the tag. For more information about address tagging, see 'A dress tag ing'.

For a "nchr .ous_____leck Fault abort, bits[63:60] are UNKNOWN.

Execution. EL0 makes FAR_EL1 become UNKNOWN.

The address he in this field is an address accessed by the instruction fetch or data access that caused the exception that actually gave rise to the instruction or data abort. It is the lower address that gave rise to the fault that is reported. Where different faults from different addresses arise from the same instruction, such as for an instruction that loads or stores an unaligned address that crosses a page boundary, the architecture does not prioritize which fault is reported.

For all other exceptions taken to EL1, FAR_EL1 is UNKNOWN.

FAR_EL1 is made UNKNOWN on an exception return from EL1.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing FAR_EL1

When HCR_EL2.E2H is 1, without explicit synchronization, access from EL3 using the mnemonic FAR_EL1 or FAR_EL12 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Accesses to this register use the following encodings in the System register encoding space:

MRS <Xt>, FAR_EL1

ор0	op1	CRn	CRm	op2
0b11	0b000	0b0110	0b0000	0b000



MRS <Xt>, FAR_EL12

ор0	op1	CRn	CRm	op2
0b11	0b101	0b0110	0b0000	0b000

```
1 if PSTATE.EL == ELO then
2
       UNDEFINED;
3
   elsif PSTATE.EL == EL1 then
       if EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '101' then
4
5
           X[t, 64] = NVMem[0x220];
       elsif EL2Enabled() && HCR_EL2.NV == '1' then
6
7
           AArch64.SystemAccessTrap(EL2, 0x18);
8
       else
9
           UNDEFINED;
10
   elsif PSTATE.EL == EL2 then
       if HCR_EL2.E2H == '1' then
11
12
           X[t, 64] = FAR\_EL1;
13
       else
14
           UNDEFINED;
15
   elsif PSTATE.EL == EL3 then
16
       if EL2Enabled() && !ELUsingAArch32(EL2) && HCR
                                                           .E2H ==
                                                                        then
           X[t, 64] = FAR\_EL1;
17
18
       else
19
           UNDEFINED;
```

```
MSR FAR_EL12, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	101	0b0110	0b0000	0b000

```
1
   if PSTATE.EL -
                    TL0 t'
2
       UNDEFINED;
                         11 ther
3
   elsif PST EL ==
       if L2Enc led() HC'_LL2.<NV2,NV1,NV> == ':
NVMem Jx220] = C, 64];
lsif EI_mabled() && HCR_EL2.NV == '1' then
4
                          HC' LL2.<NV2,NV1,NV> == '101' then
5
6
7
          8
        el.
9
             EFINED;
   elsif PSTA. EL == EL2 then
10
        if HCR_E.Z.E2H == '1' then
11
12
            FAR\_EL1 = X[t, 64];
13
        else
            UNDEFINED;
14
15
   elsif PSTATE.EL == EL3 then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then
16
17
            FAR\_EL1 = X[t, 64];
18
        else
19
           UNDEFINED;
```

MRS <Xt>, FAR_EL2

ор0	op1	CRn	CRm	op2
0b11	0b100	0b0110	0Ь0000	0b000

Chapter E3. System registers affected by SME E3.2. Changes to existing System registers

```
if PSTATE.EL == ELO then
1
2
       UNDEFINED;
3
   elsif PSTATE.EL == EL1 then
4
       if EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
5
           X[t, 64] = FAR\_EL1;
6
       elsif EL2Enabled() && HCR_EL2.NV == '1' then
           AArch64.SystemAccessTrap(EL2, 0x18);
7
8
        else
           UNDEFINED;
9
   elsif PSTATE.EL == EL2 then
10
11
       X[t, 64] = FAR\_EL2;
   elsif PSTATE.EL == EL3 then
12
13
       X[t, 64] = FAR\_EL2;
```

MSR FAR_EL2, <Xt>

op0	op1	CRn	CRm	op2
)b11	0b100	0b01	960000	0b000

```
if PSTATE.EL == EL0 then
1
2
        UNDEFINED;
3
   elsif PSTATE.EL == EL1 then
        if EL2Enabled() && HCR_EL2.<NV2,NV == '11'</pre>
4
                                                        then
        FAR_EL1 = X[t, 64];
elsif EL2Enabled() && HCR_EL2.NV
5
                                                11
6
                                                      en.
7
            AArch64.SystemAccessTrap 2, 0x
8
        else
9
            UNDEFINED;
10
   elsif PSTATE.EL == EL2 the
11
       FAR_EL2 = X[t, 64];
   elsif pstate.el == el3 .en
12
13
    FAR\_EL2 = X[t, 6^];
```

E3.2.5 FAR_EL2, Fault Address Register (EL2)

The FAR_EL2 characteristics are:

Purpose

Holds the faulting Virtual Address for all synchronous Instruction Abort exceptions, Data Abort exceptions, PC alignment fault exceptions and Watchpoint exceptions that are taken to EL2.

Configuration

If EL2 is not implemented, this register is RESO from EL3.

This register has no effect if EL2 is not enabled in the current Security state.

AArch64 system register FAR_EL2 bits [31:0] are architecturally mapped to AArch32 system register HDFAR[31:0].

AArch64 system register FAR_EL2 bits [63:32] are architectura mapped to A. ch32 system register HIFAR[31:0].

When EL2 is implemented, AArch64 system register F⁷ _EL2 bits [_ ?] are rehitecturally mapped to AArch32 system register DFAR31:0.

When EL2 is implemented, AArch64 system register 1 2 2 bits [6 32] are architecturally mapped to AArch32 system register IFAR31:0.

Attributes

FAR_EL2 is a 64-bit register.

Field descriptions

The FAR_EL2 bit assignmer a.



[0:ر / Bits

Faulth. Vir. al Address for synchronous exceptions taken to EL2. Exceptions that set the FAR_EL2 are Instruction Aborts (L 0x20 or 0x21), Data Aborts (EC 0x24 or 0x25), PC alignment faults (EC 0x22), and Watchpoints (EC 0x34 or 0x5). ESR_EL2.EC holds the EC value for the exception.

For a synchronous External abort, if the VA that generated the abort was from an address range for which $TCR_ELx.TBI\{<0|1>\} == 1$ for the translation regime in use when the abort was generated, then the top eight bits of FAR_EL2 are UNKNOWN.

For a synchronous External abort other than a synchronous External abort on a translation table walk, this field is valid only if ESR_EL2.FnV is 0, and FAR_EL2 is UNKNOWN if ESR_EL2.FnV is 1.

If a memory fault that sets FAR_EL2, other than a Tag Check Fault, is generated from a data cache maintenance or other DC instruction, this field holds the address specified in the register argument of the instruction.

On an exception due to a Tag Check Fault caused by a data cache maintenance or other DC instruction, the address held in FAR_EL2 is IMPLEMENTATION DEFINED as one of the following:

- The lowest address that gave rise to the fault.
- The address specified in the register argument of the instruction as generated by MMU faults caused by DC ZVA.

If the exception that updates FAR_EL2 is taken from an Exception level using AArch32, the top 32 bits are all zero, unless both of the following apply, in which case the top 32 bits of FAR_ELx are 0x00000001:

- The faulting address was generated by a load or store instruction that sequentially incremented from address 0xFFFFFFF. Such a load or store instruction is CONSTRAINED UNPREDICTABLE.
- The implementation treats such incrementing as setting bit[32] of the virtual address to 1.

When the PE sets ESR_EL2.{ISV,FnP} to $\{0,1\}$ on taking a Data Abort exception, or sets ESR_EL2.{FnV,FnP} to $\{0,1\}$ on taking a Watchpoint exception, the PE sets FAR_EL2 to any address within the naturally-aligned fault granule that contains the virtual address of the memory access that generated the Data Abort exception or Watchpoint exception.

The naturally-aligned fault granule is one of:

- When ESR_EL2.DFSC is 0b010001, indicating a Synchronous Tag Check fault, it is a 16-byte tag granule.
- When ESR_EL2.DFSC is 0b11010x, indicating an IMPLEMENTATIC DEF. D fault, it is an IMPLEMENTA-TION DEFINED granule.
- Otherwise, it is the smallest implemented translation granule.

When FEAT_MOPS is implemented, the value in FAR_EL2 or synchron. s exception from any of the Memory Copy and Memory Set instructions represents the first element and has not been or all or set, and is determined as follows:

- For a Data Abort generated by the MMU, the value is in the ad ess range of the relevant translation granule, aligned to the size of the relevant translation n gran is of t' address that generated the Data Abort. Bits[(n-1):0] of the value are UNKNOWN where 2 is the relevant translation granule size in bytes. For the purpose of calculating the relevant translation granule size is equivited to the size of translation, then the current translation granule size is equivited to the size of translation granule size is equivited. The MMU is disabled for a stage of translation, then the current translation granule size is equivited to the size of the size of the translation granule is:
 - For MMU faults genered at stage the cu. nt stage 1 translation granule.
 - For MMU faults g ted at stage 2 he smaller of the current stage 1 translation granule and the current stage 2 translation ranule.
 - If FEAT_RM⁺ is implement for a synchronous data abort generated as the result of a GPF, the smallest of the current stage 1 translation granule and the configured granule i.ze in GP CR_EL3.PGS.
- For a Data A. to related by Tag Check failure, the value is the lowest address that failed the Tag Check with block re of the pad or store.
- Jor a Wat point explored on the value is an address range of the size defined by the DCZID_EL0.BS field. This add to be the element with a watchpoint, but can be some earlier element.
- O. wise, the value is the lowest address in the block size of the load or store.

For a Data A rt exception or Watchpoint exception, if address tagging is enabled for the address accessed by the data access that caused the exception, then this field includes the tag. For more information about address tagging, see 'Address tagging'.

For a synchronous Tag Check Fault abort, bits[63:60] are UNKNOWN.

Execution at EL1 or EL0 makes FAR_EL2 become UNKNOWN.

The address held in this field is an address accessed by the instruction fetch or data access that caused the exception that actually gave rise to the instruction or data abort. It is the lower address that gave rise to the fault that is reported. Where different faults from different addresses arise from the same instruction, such as for an instruction that loads or stores an unaligned address that crosses a page boundary, the architecture does not prioritize which fault is reported.

For all other exceptions taken to EL2, FAR_EL2 is UNKNOWN.

FAR_EL2 is made UNKNOWN on an exception return from EL2.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing FAR_EL2

When HCR_EL2.E2H is 1, without explicit synchronization, access from EL2 using the mnemonic FAR_EL2 or FAR_EL1 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Accesses to this register use the following encodings in the System register encoding space:

MRS <Xt>, FAR_EL2

		op0	op1	CRn		op2	
		0b11	0b100	0b0110	0Ь0000	0Ь000	
1	if PSTATE.EL =	= ELO then					
2	UNDEFINED;						
3	elsif PSTATE.E	L == EL1 the	en				
4	if EL2Enab	led() && HCH	R_EL2. <nv2,nv> ==</nv2,nv>	'11' the			
5	X[t, 6	4] = FAR_EL	L;				
6	elsif EL2E:	nabled() &&	HCR_EL2.NV == 1	' the			
7	AArch6	4.SystemAcce	essTrap(EL2, x18));			
8	else	2	<u> </u>				
9	UNDEFI	NED:					
10	elsif PSTATE.E	, . == EL2 the	en				
11	X[t, 64] =	FAR EL2:					
12	Aleif DOTATE F	I FI3 +b					
13	X[t, 64] =	FAR_EL2;					
	MSR FAR EL2,	<xt></xt>					

<u>n0</u>	op1	CRn	CRm	op2	
0b1.	0b100	0b0110	060000	0b000	
if PS TE == nen nen					
UNL NED;					
elsif PS1 E.EL == EL1 then					
<pre>if EL2_ bled() && HCR_EL2.<nv2,nv> == '11' then</nv2,nv></pre>					
$FAR_{LL1} = X[t, 64];$					
elsif EL2Enabled() &&	HCR_EL2.NV == '1	' then			
AArch64.SystemAcce	ssTrap(EL2, 0x18);			
else					
UNDEFINED;					
elsif PSTATE.EL == EL2 the	n				
$FAR_EL2 = X[t, 64];$					
elsif PSTATE.EL == EL3 the	n				
$FAR_EL2 = X[t, 64];$					
	<pre>>>0 0bl. 0bl. if PS: TE: == En. Aen UNL :NED; elsif PS: F.EL == EL1 the if EL2L: bled() && HCR FAR_LL1 = X[t, 64] elsif EL2Enabled() && AArch64.SystemAcce else UNDEFINED; elsif PSTATE.EL == EL2 the FAR_EL2 = X[t, 64]; elsif PSTATE.EL == EL3 the FAR_EL2 = X[t, 64];</pre>	<pre>n0 op1 Obl. Obl00 if PS_TELL == ELnen UNENED; elsif PS_E.EL == EL1 then if EL2E_bled() && HCR_EL2.<nv2,nv> == FAR_LL1 = X[t, 64]; elsif EL2Enabled() && HCR_EL2.NV == '1 AArch64.SystemAccessTrap(EL2, 0x18 else UNDEFINED; elsif PSTATE.EL == EL2 then FAR_EL2 = X[t, 64]; elsif PSTATE.EL == EL3 then FAR_EL2 = X[t, 64];</nv2,nv></pre>	<pre>n0 op1 CRn Ob1. Ob100 Ob0110 if FS_TELLS == ELL_Men UNE_MED; elsif PS1 F.EL == EL1 then if EL2E bled() && HCR_EL2.<nv2,nv> == '11' then FAR_LL1 = X[t, 64]; elsif EL2Enabled() && HCR_EL2.NV == '1' then AArch64.SystemAccessTrap(EL2, 0x18); else UNDEFINED; elsif PSTATE.EL == EL2 then FAR_EL2 = X[t, 64]; elsif PSTATE.EL == EL3 then FAR_EL2 = X[t, 64];</nv2,nv></pre>	p0 op1 CRn CRm Ob1. Ob100 Ob0110 Ob0000 if PLUTELS == ELLINEN UNLYNED; elsif PS1 F.EL == EL1 then if EL2L bled() && HCR_EL2. <nv2,nv> == '11' then FAR_LL1 = X[t, 64]; elsif EL2Enabled() && HCR_EL2.NV == '1' then AArch64.SystemAccessTrap(EL2, 0x18); else UNDEFINED; elsif PSTATE.EL == EL2 then FAR_EL2 = X[t, 64]; elsif PSTATE.EL == EL3 then FAR_EL2 = X[t, 64];</nv2,nv>	

MRS <Xt>, FAR_EL1

ор0	op1	CRn	CRm	op2
0b11	0b000	0b0110	0b0000	0b000

Chapter E3. System registers affected by SME E3.2. Changes to existing System registers

```
if PSTATE.EL == EL0 then
1
2
       UNDEFINED;
3
   elsif PSTATE.EL == EL1 then
4
       if EL2Enabled() && HCR_EL2.TRVM == '1' then
5
           AArch64.SystemAccessTrap(EL2, 0x18);
6
       elsif EL2Enabled() & IsFeatureImplemented(FEAT_FGT) & (!HaveEL(EL3) || SCR_EL3.FGTEn
           ↔== '1') && HFGRTR_EL2.FAR_EL1 == '1' then
7
           AArch64.SystemAccessTrap(EL2, 0x18);
8
       elsif EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '111' then
9
           X[t, 64] = NVMem[0x220];
10
       else
           X[t, 64] = FAR\_EL1;
11
12
   elsif PSTATE.EL == EL2 then
      if HCR_EL2.E2H == '1' then
13
14
           X[t, 64] = FAR\_EL2;
15
       else
           X[t, 64] = FAR_EL1;
16
17
   elsif PSTATE.EL == EL3 then
18
       X[t, 64] = FAR\_EL1;
```

MSR FAR_EL1, <Xt>

op0	op1	· · ·	CRm	op2	
0b11	0b000)b011t	0b0000	0b000	

```
1
   if PSTATE.EL == ELO then
       UNDEFINED;
2
3
   elsif PSTATE.EL == EL1 then
4
        if EL2Enabled() && HCR _2.TVM ==
                                                then.
        5
6
        AArch64.Sy = mAccessTrap ( 2, 0x18);
elsif EL2Enab d() && HCR_EL2. 2, NV1, NV> == '111' then
7
8
9
           NVMem[ .20] = Y ., 64];
10
        else
   Else
   FAR_EL1 X' 64];
elsif PST' EL = 12 then
   if .R_L E2H '1' .ien
      FAR_E I = X[t, ];
   lse
11
12
13
14
15
         1se
           FA' ____ 64];
16
17
   elsif .
             .EL == EL3 then
        FAR_. ^{1} = X[t, 64];
18
```

E3.2.6 FAR_EL3, Fault Address Register (EL3)

The FAR_EL3 characteristics are:

Purpose

Holds the faulting Virtual Address for all synchronous Instruction Abort exceptions, Data Abort exceptions and PC alignment fault exceptions that are taken to EL3.

Configuration

This register is present only when EL3 is implemented. Otherwise, direct accesses to FAR_EL3 are UNDEFINED.

Attributes

FAR_EL3 is a 64-bit register.

Field descriptions

The FAR_EL3 bit assignments are:

63		32
	Faulting Virtual Address for synch we exception taken to EL3	
31		0
	Faulting Virtual Addreg for syn ironousptions taken to EL3	

Bits [63:0]

Faulting Virtual Address for sync c onous exce_F c ins tak, to EL3. Exceptions that set the FAR_EL3 are Instruction Aborts (EC 0x20 or 0x21), P a borts (EC 0x2 or 0x25), and PC alignment faults (EC 0x22). ESR_EL3.EC holds the EC value for the exception.

For a synchronous F ernal abort, if u. VA that generated the abort was from an address range for which TCR_ELx.TBI{ $<^{\circ} >$ } == 1 for the translation regime in use when the abort was generated, then the top eight bits of FAR_EL3 are NKNOV.

For a synchronous E. nal abort ner than a synchronous External abort on a translation table walk, this field is valid or ' if E. _EL3.. V is , and FAR_EL3 is UNKNOWN if ESR_EL3.FnV is 1.

If e = mory fe, that sets FAR_EL3, other than a Tag Check Fault, is generated from a data cache maintenance or other \cap in field holds the address specified in the register argument of the instruction.

On an exc. ion due to a Tag Check Fault caused by a data cache maintenance or other DC instruction, the address held in FAR___3 is IMPLEMENTATION DEFINED as one of the following:

- The lowest address that gave rise to the fault.
- The address specified in the register argument of the instruction as generated by MMU faults caused by DC ZVA.

If the exception that updates FAR_EL3 is taken from an Exception level using AArch32, the top 32 bits are all zero, unless both of the following apply, in which case the top 32 bits of FAR_ELx are 0x00000001:

- The faulting address was generated by a load or store instruction that sequentially incremented from address 0xFFFFFFF. Such a load or store instruction is CONSTRAINED UNPREDICTABLE.
- The implementation treats such incrementing as setting bit[32] of the virtual address to 1.

When the PE sets ESR_EL3.{ISV,FnP} to $\{0,1\}$ on taking a Data Abort exception, the PE sets FAR_EL3 to any address within the naturally-aligned fault granule that contains the virtual address of the memory access that generated the Data Abort exception.

The naturally-aligned fault granule is one of:

- When ESR_EL3.DFSC is 0b010001, indicating a Synchronous Tag Check fault, it is a 16-byte tag granule.
- When ESR_EL3.DFSC is 0b11010x, indicating an IMPLEMENTATION DEFINED fault, it is an IMPLEMENTATION DEFINED granule.
- Otherwise, it is the smallest implemented translation granule.

When FEAT_MOPS is implemented, the value in FAR_EL3 on a synchronous exception from any of the Memory Copy and Memory Set instructions represents the first element that has not been copied or set, and is determined as follows:

- For a Data Abort generated by the MMU, the value is within the address range of the relevant translation granule, aligned to the size of the relevant translation granule of the address that generated the Data Abort. Bits[(n-1):0] of the value are UNKNOWN, where 2ⁿ is the relevant translation granule size in bytes. For the purpose of calculating the relevant translation granule, if the MMU is disabled for a stage of translation, then the current translation granule size is equal to 2⁶⁴ for stage 1, and the PARange for stage 2. The relevant translation granule is:
 - For MMU faults generated at stage 1, the current stage 1 tr slation granu
 - For MMU faults generated at stage 2, the smaller of the current stage 1 t nslation granule and the current stage 2 translation granule.
 - If FEAT_RME is implemented, for a synchronous a abort general, as c result of a GPF, the smallest of the current stage 1 translation granule, the green stage 2 transle in granule and the configured granule size in GPCCR_EL3.PGS.
- For a Data Abort generated by a Tag Check in the variable is the owest address that failed the Tag Check within the block size of the load or store
- Otherwise, the value is the lowest addre. in the local of the load or store.

For a Data Abort exception, if address ta_i ing is bled for the address accessed by the data access that caused the exception, then this field incluies the ta_{ϵ} For model information about address tagging, see 'Address tagging'.

For a synchronous Tag Check . 't abort, bits[t. 50] are UNKNOWN.

Execution at EL2, EL1, or EL0 make FAR_EL3 become UNKNOWN.

The address held in this register is an access accessed by the instruction fetch or data access that caused the exception that accelly gave lise to the instruction or data abort. It is the lowest address that gave rise to the fault that is report 1 Wb to different faults from different addresses arise from the same instruction, such as for an instruction that is not or stores in unaligned address that crosses a page boundary, the architecture does not prioritize time fault is porte

For 1 other ey eptions tak 1 to EL3, FAR_EL3 is UNKNOWN.

FAR_1 3' made UNKNOWN on an exception return from EL3.

The reset by vior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing FAR_EL3

Accesses to this register use the following encodings in the System register encoding space:

MRS <Xt>, FAR_EL3

ор0	op1	CRn	CRm	op2
0b11	0b110	0b0110	0b0000	0b000

1 if PSTATE.EL == ELO then

Chapter E3. System registers affected by SME E3.2. Changes to existing System registers

```
2 UNDEFINED;
3 elsif PSTATE.EL == EL1 then
4 UNDEFINED;
5 elsif PSTATE.EL == EL2 then
6 UNDEFINED;
7 elsif PSTATE.EL == EL3 then
8 X[t, 64] = FAR_EL3;
```

MSR FAR_EL3, <Xt>

1

6 7 8

	ор0	op1	CRn	CRm	op2
	0b11	0b110	0b0110	06000	0b000
<pre>if PSTATE.EL = UNDEFINED; elsif PSTATE.E UNDEFINED; elsif PSTATE.E UNDEFINED; elsif PSTATE.E</pre>	= ELO then L == EL1 then L == EL2 then L == EL3 then				

E3.2.7 FPCR, Floating-point Control Register

The FPCR characteristics are:

Purpose

Controls floating-point behavior.

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 apped to A. h32 system register FPSCR[12:8].

Attributes

FPCR is a 64-bit register.

Field descriptions

The FPCR bit assignments are:



Alternative h	^c -precision	control bit.
1 meetinaan ve m	precision	control on.

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.

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 FABS, FMAX*, FMIN*, and FNEG instructions, and a default N er returned as a result of these instructions.		

The value of this bit controls both scalar and Advanced SIMD soating-poil. writhme c.

The reset behavior of this field is:

FZ, bit [24]

Flushing denormalized numbers to zero contre it

Fz.	Meaning
0	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.

The specified rounding mode is used by both scalar and Advanced SP \supset floating-p, t instructions.

If FPCR.AH is 1, then the following instructions use Round to Noarest orde regardle of the value of this bit:

- The FRECPE, FRECPS, FRECPX, FRSQRTE, and FP 2RTS instruc ns.
- The BFCVT, BFCVTN, BFCVTN2, BFCVTNT, E 1LALP .nd BFMLAT instructions.

The reset behavior of this field is:

• On a Warm reset, this field resets to an ar frecture y UNK w value.

Stride, bits [21:20]

This field has no function in AArch¹ stat. and no. are values are ignored during execution in AArch64 state. This field is included only for *c* text saving a. restoration of the AArch32 FPSCR.Stride field. The reset behavior of this f. d is:

• On a Warm reset inis field resets it in architecturally UNKNOWN value.

FZ16, bit [19]

When FF TP16 ... mplemer d:

Flush' g denor alized nu. is to zero control bit on half-precision data-processing instructions.

FZ16	Meaning
060	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.

The value of this bit applies to both scalar and Advanced SIMD floating-point half-precision calculations.

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:

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 v

IDE, bit [15]

Input Denormal floating-point exception trap enable.



update the FPSR.IDC bit.

When the PE is in Streeping SVE mode and FEAT_SME_FA64 is not implemented or not enabled, the value of FPCR.IDE is treated is 0 for all purposes there than a direct read or write of the FPCR.

The Effective va. of this at controls both scalar and vector floating-point arithmetic.

If the implementation we not support this exception, this bit is RAZ/WI.

The re c behaver of this ' is:

In a W this field resets to an architecturally UNKNOWN value.

Bit [14]

Reserved, RESU.

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.

Chapter E3. System registers affected by SME E3.2. Changes to existing System registers

EBF	Meaning
060	 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 addition with intermediate rounding of all rounds and rounds.
0b1	 These is fructions us the extended BFloat16 behaviors. Support and four IEEE 754 rounding modes selected by the FPCR.RMode contain.
2	 C Lionally, 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.

Otherwise:

res0

IXE, bit [12]

Inexact floating-point exception trap enable.

IXE	Meaning
000	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-pc , arithmetic

If the implementation does not support this exception, this bit is RAZ VI.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally _NKNOW value.

UFE, bit [11]

Underflow floating-point exception trap enable

	Meaning
0b	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.

Where ne PE is a Stream. SVE mode, and FEAT_SME_FA64 is not implemented or not enabled, the value of FPC UFE is a los 0 for all purposes other than a direct read or write of the FPCR.

The Effe ve value of this bit controls both scalar and vector floating-point arithmetic.

If the implementation does not support this exception, this bit is RAZ/WI.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

OFE, bit [10]

Overflow floating-point exception trap enable.

OFE	Meaning
0b0	Untrapped exception handling selected. If the floating-point exception occurs, the FPSR.OFC bit is set to 1.

OFE	Meaning
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.

If the implementation does not support this exception, this bit is RAZ/WI.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOV value.

DZE, bit [9]

Divide by Zero floating-point exception trap enable.

JL.	reani.
060	Untrapped exception handling selected. If theoating-point exception occurs, the FPSR.DZC bit is set to 1.
0Ъ.	Trapped exception handling selected. If the floating-point exception occurs, the PE does not update the FPSR.DZC bit.

When the PE is in _reaming `VE mode, and FEAT_SME_FA64 is not implemented or not enabled, the value of FPCR.DZE is tr_____d as 0 ____r all purposes other than a direct read or write of the FPCR.

The Effective value of this bit cover ols both scalar and vector floating-point arithmetic.

If the '.plemer tion doc' support this exception, this bit is RAZ/WI.

The set bet in field is:

• On Varm reset, this field resets to an architecturally UNKNOWN value.

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.

The Effective value of this bit controls both scalar and vector floating-point arithmetic.

If the implementation does not support this exception, this bit is RAZ/WI.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [7:3]

Reserved, RESO.

NEP, bit [2]

When FEAT_AFP is implemented:

Controls how the output elements other than the lowest element of the vertice and for Advanced SIMD scalar instructions.

NEP	' caning
060	Summer instructions.

NEP	Meaning
Db1	 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.</dd></sd></hd> For 3-input versions of the FMADD, FMSUB, FNMADD, and FNMSUB instructions, the <ha>, <sa>, or <da> register.</da></sa></ha> For ^ in t scalar versions of the FACGE, FGT, F 'EQ (register), FCMGE register), and `CMGT (register) tructions, th <hm>, <sm>, or <dm> reg. ?r.</dm></sm></hm> For 2- nutalar versions of the FABD, FADD (slar), FDIV (scalar), FMAX (scal⁻¹), FMAXNM (scalar), FMIN (scalar), FMINN (scalar), FMUL (by element), FMUL (scalar), FMUL (by element), FMUL (scalar), FMULX (by element), FMUL (scalar), FMECPS, FRSQRTS, and FSUB (scalar) instructions, the <hn>, <sn>, or <dn> register.</dn></sn></hn> For 1-input scalar versions of the following instructions, the <hd>, <sd>, or <dd> register:</dd></sd></hd> The (vector) versions of the FCVTAS, FCVTAU, FCVTNS, FCVTMU, FCVTNS, FCVTMU, FCVTPS, and FCVTPU instructions. The (vector, fixed-point) and (vector, integer) versions of the FABS, FNEG, FRINT32X, FRINT32Z, FRINT4X, FRINT1, FRINT32X, FRINT32Z, FRINT4X, FRINT1, FRINT32X, FRINT32Z, FRINT4, FRINT1, FRINT32X, FRINT32X, FRINT32Z, FRINT4, FRINT1, FRINT3X, FRINT32X, FRINT32Z, FRINT4, FRINT4,

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:

res0

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 tip: and other exceptions and a range of other behaviors. The value of the FPCR.AH bit selects between these nodels.

For more information on the FPCR.AH bit, see 'Flushing deno. Plized number to zero', Floating- point exceptions and exception traps and the pseudocode of the floating point protections.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architectural. UNKN NN value.

Otherwise:

res0

FIZ, bit [0]

When FEAT_AFP is implemente

Flush Inputs to Zero. Control other single-precision, double-precision and BFloat16 input operands that are denormalized numbers are ushed 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:

res0

Accessing FPCR

Accesses to this register use the following encodings in the System register encoding space:

MRS <Xt>, FPCR

op0	op1	CRn	CRm	ор2	
0b11	0b011	0b0100	0b0100	0b000	

```
1
   if PSTATE.EL == EL0 then
       if Halted() & HaveEL(EL3) & EDSCR.SDD == '1' & boolean INFIGURATION_DEFINED "EL3
2
            →trap priority when SDD == '1'" && CPTR_EL3.TFP ==
                                                                     tne.
3
            UNDEFINED:
4
       elsif !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && PACR_EL1.FPL != '11' then
5
           if EL2Enabled() && HCR_EL2.TGE == '1' then
6
               AArch64.SystemAccessTrap(EL2, 0x00);
7
            else
8
               AArch64.SystemAccessTrap(EL1, 0x07);
                                                      ' && C .R_EL2.FPL .. != '11' then
9
       elsif EL2Enabled() && HCR_EL2.<E2H,TGE> == 
            AArch64.SystemAccessTrap(EL2, 0x07);
10
       elsif EL2Enabled() && HCR_EL2.E2H == '1' '& CP'
                                                          L2.FPEN -
                                                                      'x0' then
11
           AArch64.SystemAccessTrap(EL2, 0x^{0}
12
                                             1' && CPTR_EL
13
       elsif EL2Enabled() && HCR_EL2.E2H /
                                                                 == '1' then
14
           AArch64.SystemAccessTrap(EL2, x07);
       elsif HaveEL(EL3) && CPTR_EL3.TFP
15
                                             1
            if Halted() && EDSCR.SDD = '1'
16
17
               UNDEFINED:
18
            else
19
               AArch64.System AssTrap(L
                                              0x0
20
       else
21
            X[t, 64] = FPCP
22
   elsif PSTATE.EL == EL1 chen
       23
24
           UNDEFIN
       elsif CPA EL1.FP' == 'x0' then
AArch64 'st AccessTr'p(EL1, 0x07);
elsif F12Enab. () && HC EL2.E2H != '1' && CPTR_EL2.TFP == '1' then
25
26
27
             ALC 4.Sys ACCF frap(EL2, 0x07);
28
        e if EL2 abled . - ACR_EL2.E2H == '1' && CPTR_EL2.FPEN == 'x0' then
29
           AArc<sup>1</sup> 4.SystemA cessTrap(EL2, 0x07);
30
31
          sif F
                        && CPTR_EL3.TFP == '1' then
              Halted() && EDSCR.SDD == '1' then
32
33
               UNDEFINED;
           e. rch64.SystemAccessTrap(EL3, 0x07);
34
35
36
       else
37
           X[t, 64] = FPCR;
   elsif PSTATE.EL == EL2 then
38
39
       if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3
            →trap priority when SDD == '1'" && CPTR_EL3.TFP == '1' then
40
            UNDEFINED:
41
       elsif HCR_EL2.E2H == '0' && CPTR_EL2.TFP == '1' then
42
           AArch64.SystemAccessTrap(EL2, 0x07);
        elsif HCR_EL2.E2H == '1' && CPTR_EL2.FPEN == 'x0' then
43
44
           AArch64.SystemAccessTrap(EL2, 0x07);
       elsif HaveEL(EL3) && CPTR_EL3.TFP == '1' then
45
46
            if Halted() && EDSCR.SDD == '1' then
47
               UNDEFINED;
48
            else
49
               AArch64.SystemAccessTrap(EL3, 0x07);
50
       else
51
            X[t, 64] = FPCR;
52
   elsif PSTATE.EL == EL3 then
```
Chapter E3. System registers affected by SME E3.2. Changes to existing System registers

MSR FPCR, <Xt>

орО	op1	CRn	CRm	op2
0b11	0b011	0b0100	0b0100	0b000

```
if PSTATE.EL == ELO then
 1
         if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolear MPLL TATION_DEFINED "EL3
2
              ↔trap priority when SDD == '1'" && CPTR_EL3.TFP => 1' then
3
             UNDEFINED:
4
         elsif !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CR_EL1.FPEI != '11' then
 5
             if EL2Enabled() && HCR_EL2.TGE == '1' then
                 AArch64.SystemAccessTrap(EL2, 0x00);
6
7
             else
8
                 AArch64.SystemAccessTrap(EL1, 0x07)
         elsif EL2Enabled() && HCR_EL2.<E2H,TGE> == '1
                                                                &F .PTR_EL2 TPEN != '11' then
9
10
             AArch64.SystemAccessTrap(EL2, 0x07) ·
         elsif EL2Enabled() && HCR_EL2.E2H ==
                                                        . CPTR_ 2.FPF == 'x0' then
11
        AArch64.SystemAccessTrap(EL2, 0 7);
elsif EL2Enabled() && HCR_EL2.E2P = '1' & CPTR_EL2.FP == '1' then
AArch64.SystemAccessTrap(EL2, 07);
elsif HaveEL(EL3) && CPTR_EL3 TFP = ' then
12
13
14
15
             if Halted() && EDSCR.SDD -
16
                                              11 t.
17
                 UNDEFINED;
18
             else
19
                 AArch64.Syste essTrap(EL_ 0x07);
20
         else
21
             FPCR = X[t, 64];
    22
23
24
        elsif CPACR_ 11. EN == 'x^' then
   A^moh64.S_ smAccess ap(EL1, 0x07);
els' EL nable. && R_EL2.E2H != '1' && CPTR_EL2.TFP == '1' then
   AArch .Syste. ssTrap(EL2, 0x07);
isif EL' nabled() & HCR_EL2.E2H == '1' && CPTR_EL2.FPEN == 'x0' then
   AArch .Syste. Strap(CPAC) = COTO

25
26
27
28
29
30
             AAr
                           mAccessTrap(EL2, 0x07);
31
         e
              f laveEL(EL3) && CPTR_EL3.TFP == '1' then
32
              Halted() && EDSCR.SDD == '1' then
33
                 UNDEFINED;
34
             else
35
                 AArch64.SystemAccessTrap(EL3, 0x07);
36
         else
37
            FPCR = X[t, 64];
38
    elsif PSTATE.EL == EL2 then
39
         if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3
             →trap priority when SDD == '1'" && CPTR_EL3.TFP == '1' then
40
             UNDEFINED;
41
         elsif HCR_EL2.E2H == '0' && CPTR_EL2.TFP == '1' then
42
             AArch64.SystemAccessTrap(EL2, 0x07);
43
         elsif HCR_EL2.E2H == '1' && CPTR_EL2.FPEN == 'x0' then
44
             AArch64.SystemAccessTrap(EL2, 0x07);
         elsif HaveEL(EL3) && CPTR_EL3.TFP == '1' then
45
             if Halted() && EDSCR.SDD == '1' then
46
47
                 UNDEFINED:
48
             else
49
                  AArch64.SystemAccessTrap(EL3, 0x07);
50
         else
51
            FPCR = X[t, 64];
```

Copyright © 2022-2023 Arm Limited or its affiliates. All rights reserved. Non-confidential

Chapter E3. System registers affected by SME E3.2. Changes to existing System registers



E3.2.8 HCRX_EL2, Extended Hypervisor Configuration Register

The HCRX_EL2 characteristics are:

Purpose

Provides configuration controls for virtualization, including defining whether various operations are trapped to EL2.

Configuration

If EL2 is not implemented, this register is RESO from EL3.

The bits in this register behave as if they are 0 for all purposes other than direct reads of the register if:

- EL2 is not enabled in the current Security state.
- SCR_EL3.HXEn is 0.

This register is present only when FEAT_HCX is implemented. Otherwine direct accesses to HCRX_EL2 are UNDEFINED.

Attributes

HCRX_EL2 is a 64-bit register.

Field descriptions

The HCRX_EL2 bit assignments are:





SCTLR2_EL1 Enable. In AArch64 state, accesses to SCTLR2_EL1 are trapped to EL2 and reported using EC syndrome value 0x18.

SCTLR2En	Meaning
040	Accesses to SCTLR2_EL1 at EL1 are trapped to EL2, unless the access generates a higher priority exception. The value in SCTLR2_EL1 is treated as 0.
0b1	This control does not cause any instructions to be trapped.

- On a Warm reset:
 - When EL3 is not implemented, this field resets to Ob0.
 - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

res0

TCR2En, bit [14]

When FEAT_TCR2 is implemented:

TCR2_EL1 Enable. In AArch64 state, accesses to TCR2_EL1 are trapped to EL2 and reported using EC syndrome value 0x18.

TCR2En	Mean. 7
0Ъ0	Accesses to 'CR? L1 at EL1 are trapped to EL? unless the ccess generates a higher rority exception. The value in TCR2_EL1 is reated as
0b1	n atrol does not cause any instructions to be u apped.

The reset behavior of this field is:

- On a Warm reset:
 - When EL3 is not aple. rted, this field esets to 0b0.
 - Otherwise, this field resets an architecturally UNKNOWN value.

Otherwise:

res0

Bits [13:-

Res ved, RES'

MSC⊾ + [11]

When FEA. MOPS is implemented:

Memory Set and Memory Copy instructions Enable. Enables execution of the CPY*, SETG*, SETP*, SETM*, and SETE* instructions at EL1 or EL0.

MSCEn	Meaning
0b0	Execution of the Memory Copy and Memory Set instructions is UNDEFINED at EL1 or EL0.
0b1	This control does not cause any instructions to be UNDEFINED.

This bit behaves as if it is 1 if any of the following are true:

• EL2 is not implemented or enabled.

• The value of HCR_EL2.{E2H, TGE} is {1, 1}.

The reset behavior of this field is:

- On a Warm reset:
 - When EL3 is not implemented, this field resets to 0b0.
 - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

res0

MCE2, bit [10]

When FEAT_MOPS is implemented:

Controls Memory Copy and Memory Set exceptions generated as part of dempting presecute the Memory Copy and Memory Set instructions from EL1.

MCE2	Meaning
0b0	M .nory Copy and Memory Set exceptions senerated
0b1	Copy and Memory Set exceptions generated from EL1 are taken to EL2.

When the value of HCR_EL2. {E2H TGL is $\{1, n\}$ this control does not affect any exceptions due to the higher priority SCTLR_EL2. MSCEn co_ol.

The reset behavior of this fie' is.

- On a Warm reset:
 - When EL3 not implemented, 'is field resets to Ob0.
 - Otherw[:], this fie¹ resets to an architecturally UNKNOWN value.

Otherwise:

res0

CN N, bit [?

When F. T_CMOW is implemented:

Controls caci. naintenance instruction permission for the following instructions executed at EL1 or EL0.

- IC IVAU, DC CIVAC, DC CIGDVAC and DC CIGVAC.
- ICIMVAU, DCCIMVAC.

CMOW	Meaning
060	These instructions executed at EL1 or EL0 with stage 2 read permission, but without stage 2 write permission do not generate a stage 2 permission fault.

CMOW	Meaning
0Ь1	These instructions executed at EL1 or EL0, if enabled as a result of SCTLR_EL1.UCI==1, with stage 2 read permission, but without stage 2 write permission generate a stage 2 permission fault.

For this control, stage 2 has write permission if S2AP[1] is 1 or DBM is 1 in the stage 2 descriptor. The instructions do not cause an update to the dirty state.

This bit is permitted to be cached in a TLB.

The reset behavior of this field is:

- On a Warm reset:
 - When EL3 is not implemented, this field resets to Ob
 - Otherwise, this field resets to an architecturally UN NOWN value

Otherwise:

res0

VFNMI, bit [8]

When FEAT_NMI is implemented:

Virtual FIQ Interrupt with Superprivity. Lobles standing of virtual FIQ interrupts with Superprivity.

VFNMI	Meaning
060	When HCR_EL2.VF is 1, a signaled pending virtual FIQ interrupt does not have Superpriority.
0b1	When HCR_EL2.VF is 1, a signaled pending virtual FIQ interrupt has Superpriority.

When $HCR_F' \neq VF$ is 0, this bit has no effect.

The res ' navior of this field is:

• On a v rm reset:

- Whe. EL3 is not implemented, this field resets to 0b0.

- Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

res0

VINMI, bit [7]

When FEAT_NMI is implemented:

Virtual IRQ Interrupt with Superpriority. Enables signaling of virtual IRQ interrupts with Superpriority.

VINMI	Meaning
0b0	When HCR_EL2.VI is 1, a signaled pending virtual IRQ interrupt does not have Superpriority.
0b1	When HCR_EL2.VI is 1, a signaled pending virtual IRQ interrupt has Superpriority.

When HCR_EL2.VI is 0, this bit has no effect.

The reset behavior of this field is:

- On a Warm reset:
 - When EL3 is not implemented, this field resets to 0b0.

Otherwise:

res0

TALLINT, bit [6]

When FEAT_NMI is implemented:

Traps the following writes at EL1 using AAr _____4 to EL _____when EL____s implemented and enabled:

- MSR (register) writes of ALLINT
- MSR (immediate) writes of ALL1. \with a \ue of 1.

"ALLINT	Meaning
0b0	This control does not cause any instructions to be trapped.
0b1	The specified MSR accesses at EL1 using AArch64 are trapped to EL2.

The *r* et behav r of this _ . . is:

a V minus

vhen EL3 is not implemented, this field resets to 0b0.

- C prwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

res0

SMPME, bit [5]

When FEAT_SME is implemented:

Streaming Mode Priority Mapping Enable.

Controls mapping of the value of SMPRI_EL1.Priority for streaming execution priority at EL0 or EL1.

SMPME	Meaning
000	The effective priority value is taken from SMPRI_EL1.Priority.

SMPME	Meaning
0b1	 The effective priority value is: When the current Exception level is EL2 or EL3, the value of SMPRI_EL1.Priority When the current Exception level is EL0 or EL1, the value of the SMPRIMAP_EL2 field corresponding to the value of SMPRI_EL1.Priority.

When **SMIDR_EL1**.SMPS is '0', this field is RES0.

The reset behavior of this field is:

- On a Warm reset:
 - When EL3 is not implemented, this field resets to OF
 - Otherwise, this field resets to an architecturally U NOWN value.

Otherwise:

res0

FGTnXS, bit [4]

When FEAT_XS is implemented:

Determines if the fine-grained trap in HFG. P_EL2 in the apply to each of the TLBI maintenance instructions that are accessible at EL1 also apply in the corresponding TLBI maintenance instructions with the nXS qualifier.

	FGTnXS	Meaning
	060	The fine-grained trap in the HFGITR_EL2 that applies to a TLBI maintenance instruction at EL1 also applies to the corresponding TLBI instruction with the nXS qualifier at EL1.
	0b1	The fine-grained trap in the HFGITR_EL2 that applies to a TLBI maintenance instruction at EL1 does not apply to the corresponding TLBI instruction with the nXS qualifier at EL1.

The reset behavior of this field is:

- On a Warm reset:
 - When EL3 is not implemented, this field resets to Ob0.
 - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

res0

FnXS, bit [3]

When FEAT_XS is implemented:

Determines the behavior of TLBI instructions affected by the XS attribute.

This control bit also determines whether an AArch64 DSB instruction behaves as a DSB instruction with an nXS qualifier when executed at EL0 and EL1.

FnXS	Meaning
0b0	This control does not have any effect on the behavior of the TLBI maintenance instructions.
0b1	A TLBI maintenance instruction without the nXS qualifier executed at EL1 behaves in the same way as the corresponding TLBI maintenance instruction with the nXS qualifier. An AArch64 DSB instruction executed at EL1 or EL0 behaves and a same way as the corresponding DS1 instruction with the nXS

This bit is permitted to be cached in a TLB.

The reset behavior of this field is:

- On a Warm reset:
 - When EL3 is not implemented, this / id reset to 0b0.
 - Otherwise, this field resets to an a pitecture / UNKNOWN value.

Otherwise:

res0

EnASR, bit [2]

When FEAT_LS64_V implementea.

When HCR_EL2 / .2H, TGF \ != {1, 1}, traps execution of an ST64BV instruction at EL0 or EL1 to EL2.

EnASR	Meaning
060	Execution of an ST64BV instruction at EL0 is trapped to EL2 if the execution is not trapped by SCTLR_EL1.EnASR. Execution of an ST64BV instruction at EL1 is trapped to EL2.
0b1	This control does not cause any instructions to be trapped.

A trap of an ST64BV instruction is reported using an ESR_ELx.EC value of 0x0A, with an ISS code of 0x0000000. The reset behavior of this field is:

- On a Warm reset:
 - When EL3 is not implemented, this field resets to Ob0.
 - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

res0

EnALS, bit [1]

When FEAT_LS64 is implemented:

When HCR_EL2.{E2H, TGE} $!= \{1, 1\}$, traps execution of an LD64B or ST64B instruction at EL0 or EL1 to EL2.

EnALS	Meaning
060	Execution of an LD64B or ST64B instruction at EL0 is trapped to EL2 if the execution is not trapped by SCTLR_EL1.EnALS. Execution of an LD64B or ST64B instruction at EL1 is transformed by SCTLR_EL2.
0b1	This introl does not ause any instructions to be trajed.

A trap of an LD64B or ST64B instruction is reported usir an ESR 2x.EC v. of 0x0A, with an ISS code of 0x0000002.

The reset behavior of this field is:

- On a Warm reset:
 - When EL3 is not implemented, th. 'reld re
 - Otherwise, this field resets to n arch. Ally UNKNOWN value.

Otherwise:

res0

EnAS0, bit [0]

When FEAT_LS6 _ACCDATA is implem_nted:

When HCR_EL2. $\] H, \] JE \} = \{1, 1\}$, traps execution of an ST64BV0 instruction at EL0 or EL1 to EL2.

EnAS0	Meaning
060	Execution of an ST64BV0 instruction at EL0 is trapped to EL2 if the execution is not trapped by SCTLR_EL1.EnAS0. Execution of an ST64BV0 instruction at EL1 is trapped to EL2.
0b1	This control does not cause any instructions to be trapped.

A trap of an ST64BV0 instruction is reported using an ESR_ELx.EC value of 0x0A, with an ISS code of 0x0000001. The reset behavior of this field is:

- On a Warm reset:
 - When EL3 is not implemented, this field resets to Ob0.
 - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

res0

Accessing HCRX_EL2

Accesses to this register use the following encodings in the System register encoding space:

MRS <Xt>, HCRX_EL2

ор0	op1	CRn	CRm	op2
0b11	0b100	0b0001	0b0010	0b010



MSR HCRX_EL2, <' >

		6,	op1	CRn	CRm	op2	
		0b11	0b100	0b0001	0b0010	0b010	
1	if PSTA. EL =	= ELO then					
2	UNDEF. 7D;						
3	elsif PSTAL	L == EL1 then					
4	if EL2Enat)led() && HCR_E	EL2. <nv2,nv> ==</nv2,nv>	'11' then			
5	NVMem [0xA0] = X[t, 6]	54];				
6	elsif EL2E	nabled() && HC	CR_EL2.NV == '1'	then			
7	AArch6	4.SystemAccess	sTrap(EL2, 0x18)	;			
8	else						
9	UNDEFI	NED;					
10	elsif PSTATE.E	L == EL2 then					
11	if Halted) && HaveEL(EI	3) && EDSCR.SDE) == '1' && boole	an IMPLEMENTATIO	N_DEFINED "EL3	
	⇔trap	> priority wher	n SDD == '1'" &&	SCR_EL3.HXEn ==	'0' then		
12	UNDEFI	NED;					
13	elsif Have	EL(EL3) && SCF	R_EL3.HXEn == '0	' then			
14	if Hal	.ted() && EDSCR	R.SDD == '1' the	en			
15	UN	IDEFINED;					
16	else						
17	AP	rch64.SystemAc	cessTrap(EL3, ()x18);			
18	else						
19	HCRX_E	L2 = X[t, 64];					
20	elsif PSTATE.E	L == EL3 then					

Copyright © 2022-2023 Arm Limited or its affiliates. All rights reserved. Non-confidential

Chapter E3. System registers affected by SME E3.2. Changes to existing System registers

21 HCRX_EL2 = X[t, 64];



E3.2.9 HFGRTR_EL2, Hypervisor Fine-Grained Read Trap Register

The HFGRTR_EL2 characteristics are:

Purpose

Provides controls for traps of MRS and MRC reads of System registers.

Configuration

This register is present only when FEAT_FGT is implemented. Otherwise, direct accesses to HFGRTR_EL2 are UNDEFINED.

Attributes

HFGRTR_EL2 is a 64-bit register.

Field descriptions

The HFGRTR_EL2 bit assignments are:



Bits [63:56]

Reserved, RESO.

nTPIDR2_EL0, bit [55]

When FEAT_SME is implemented:

Trap MRS reads of TPIDR2_EL0 at EL1 and EL0 using AArch64 to EL2.

nTPIDR2_EL0	Meaning
0ъ0	If EL2 is implemented and enabled in the current Security state, HCR_EL2.{E2H, TGE} != {1, 1}, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of TPIDR2_EL0 at EL1 and EL0 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.
)b1	MRS reads of TPIDR2_EL0 are not trapped by this mechanism.

The reset behavior of this field is:

• On a Warm reset, this field resets to Ob0.

Otherwise:

res0

nSMPRI_EL1, bit [54]

When FEAT_SME is implemented:

Trap MRS reads of SMPRI_EL1 at EL1 . g AA. 64 to EL2.

RI_EL1	Meaning
	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of SMPRI_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read
	MRS reads of SMPRI_EL1 are not trapped by this mechanism.
	'RI_EL1

The reset behavior of this field is:

• On a Warm reset, this field resets to Ob0.

Otherwise:

res0

Bits [53:51]

Reserved, RESO.

nACCDATA_EL1, bit [50]

When FEAT_LS64_ACCDATA is implemented:

Trap ${\tt MRS}$ reads of ACCDATA_EL1 at EL1 using AArch64 to EL2.

nACCDATA_EL1	Meaning
060	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of ACCDATA_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.
0b1	MRS reads of ACCDATA_EL1 are not trapped by this mechanism.

• On a Warm reset, this field resets to Ob0.

Otherwise:

res0

ERXADDR_EL1, bit [49]

When FEAT_RAS is implemented:

Trap MRS reads of ERXADDR_EL1 at FL1 us. A' cn64 to _ 2.

Ek. \DDk_ 'L1	Meaning
0b0	MRS reads of ERXADDR_EL1 are not trapped by this mechanism.
	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of ERXADDR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

Accessing th. Geld has the following behavior:

- This field is permitted to be RESO if all of the following are true:
 - ERRSELR_EL1 and all ERX* registers are implemented as UNDEFINED or RAZ/WI.
 - ERRIDR_EL1.NUM is zero.

The reset behavior of this field is:

- On a Warm reset:
 - When EL3 is not implemented, this field resets to Ob0.
 - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

res0

ERXPFGCDN_EL1, bit [48]

When FEAT_RASv1p1 is implemented:

Trap MRS reads of ERXPFGCDN_EL1 at EL1 using AArch64 to EL2.

	ERXPFGCDN_EL1	Meaning
	060	MRS reads of ERXPFGCDN_EL1 are not trapped by this mechanism.
Accessing this field has the following	0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS read of ER2. FGCDN_EL1 at EL1 using AAr 54 are trapped of EL2 and reported with EC sy frome value (18, unless the read generated higher provide exception.
 This field is permitted to be RET ERRSELR_EL1 and all EI ERRIDR_EL1.NUM is zer 	s0 if all of the conving a RX* regist is are in leme ro.	. true: onte UNDEFINED or RAZ/WI.
The reset behavior of this field is:		
 On a Warm reset: When EL3 is not imr' net Otherwise, this fie' re. 	nted, this ¹ d reseas to 0b to an archit, ¹ 1rally UNK	0. NOWN value.
Otherwise:	Ť	

res0

ERXPFGCTL_E⊾ bit .7]

When F AT_ Sv1p1 im mented:

Tra TRS reads if FRXPFGCTL_EL1 at EL1 using AArch64 to EL2.

ERXPFGCTL_EL1	Meaning
060	MRS reads of ERXPFGCTL_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of ERXPFGCTL_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

Accessing this field has the following behavior:

• This field is permitted to be RESO if all of the following are true:

- ERRSELR_EL1 and all ERX* registers are implemented as UNDEFINED or RAZ/WI.
- ERRIDR_EL1.NUM is zero.

- On a Warm reset:
 - When EL3 is not implemented, this field resets to 0b0.
 - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

res0

ERXPFGF_EL1, bit [46]

When FEAT_RAS is implemented:

Trap MRS reads of ERXPFGF_EL1 at EL1 using AArch64 to EL2.

ERXPFGF_EL1 Meaning

060	✓ reads of ERXPFGF_EL1 are not trapped by
	lis mech sm.
0b1	FL2 implemented and enabled in the current
	Sec. Aty state, and either EL3 is not
	implemented or SCR_EL3.FGTEn == 1, then
	MRS reads of ERXPFGF_EL1 at EL1 using
	AArch64 are trapped to EL2 and reported with
	EC syndrome value 0x18, unless the read
	generates a higher priority exception.

Accessing this field by the following be, vior:

- This field i ermitted be RES0 if all of the following are true:
 - ERRSE. F' and all FRX* registers are implemented as UNDEFINED or RAZ/WI.
 - F^{DDI}DR_ (.NUM i .ero.

The react behave r of this ' is:

• `n a W

nen EL3 is not implemented, this field resets to 0b0.

'rerwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

res0

ERXMISCn_EL1, bit [45]

When FEAT_RAS is implemented:

Trap MRS reads of multiple System registers. Enables a trap on MRS reads at EL1 using AArch64 of any of the following AArch64 System registers to EL2:

- ERXMISC0_EL1.
- ERXMISC1_EL1.
- ERXMISC2_EL1.
- ERXMISC3_EL1.

ERXMISCn_EL1	Meaning
060	MRS reads of the specified System registers are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads at EL1 using AArch64 of any of the specified System registers are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

Accessing this field has the following behavior:

- This field is permitted to be RESO if all of the following are true.
 - ERRSELR_EL1 and all ERX* registers are implemeed as UN FINED of KAZ/WI.
 - ERRIDR_EL1.NUM is zero.

The reset behavior of this field is:

- On a Warm reset:
 - When EL3 is not implemented, this fit a rese to 0b0.
 - Otherwise, this field resets to an ar .tectural' UNKNOW value.

Otherwise:

res0

ERXSTATUS_EL1, bit [44]

When FEAT_RAS is in lemented:

Trap MRS reads of F' STATUS_EL1 at L. J using AArch64 to EL2.

ERXSTATU	S_EL1 Meaning
0b0	MRS reads of ERXSTATUS_EL1 are not trappe by this mechanism.
0b1	If EL2 is implemented and enabled in the curre Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of ERXSTATUS_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

Accessing this field has the following behavior:

- This field is permitted to be RESO if all of the following are true:
 - ERRSELR_EL1 and all ERX* registers are implemented as UNDEFINED or RAZ/WI.
 - ERRIDR_EL1.NUM is zero.

The reset behavior of this field is:

- On a Warm reset:
 - When EL3 is not implemented, this field resets to 0b0.

- Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

res0

ERXCTLR_EL1, bit [43]

When FEAT_RAS is implemented:

Trap MRS reads of ERXCTLR_EL1 at EL1 using AArch64 to EL2.

ERXCTLR_EL1	Meaning
0d0	MRS read TLR_EL1 are not trapped by this minanism.
0b1	If EL, primplementer und enabled in the current recurity the, and either EL3 is not implementer or SC CEL3.FGTEn == 1, then MRS cads of E. CTLR_EL1 at EL1 using Carch64 are trapped to EL2 and reported with EC syndmine value 0x18, unless the read other of a higher priority exception.

Accessing this field has the following b vavior:

- This field is permitted to be $F \rightarrow 0$ if a $\neg f$ the h $\neg wing are true:$
 - ERRSELR_EL1 and ERX* regi. 's are h. plemented as UNDEFINED or RAZ/WI.
 - ERRIDR_EL1.NV 1. ro.

The reset behavior of th[;] field is:

- On a Warm re .:
 - When 3 is not aplemented, this field resets to 0b0.
 - Otherw, thi leld reset to an architecturally UNKNOWN value.

Otherwis

RESC

ERXF. 7 .1, bit [42]

When FEA1_ `4S is implemented:

Trap ${\tt MRS}$ reads of ERXFR_EL1 at EL1 using AArch64 to EL2.

ERXFR_EL1	Meaning
0b0	MRS reads of ERXFR_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of ERXFR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

Accessing this field has the following behavior:

- This field is permitted to be RESO if all of the following are true:
 - ERRSELR_EL1 and all ERX* registers are implemented as UNDEFINED or RAZ/WI.
 - ERRIDR_EL1.NUM is zero.

The reset behavior of this field is:

- On a Warm reset:
 - When EL3 is not implemented, this field resets to Ob0.
 - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

res0

ERRSELR_EL1, bit [41]

When FEAT_RAS is implemented:

Trap MRS reads of ERRSELR_EL1 at EL1 using AArch64 to 7 .2.

ERRSELR_EL1	eaning
0ъ0	re as of ERRSELR_EL1 are not trapped by this mechanism.
0b1	A EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of ERRSELR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

Accessing this fier 'as' following behavior:

- Thi ... 's pern. ed to ' RESO if all of the following are true:
 - ERR: LR_EL all ERX* registers are implemented as UNDEFINED or RAZ/WI.
 - ERF OR EL1.NUM is zero.

The res. ' navior of this field is:

- On a v rm reset:
 - When EL3 is not implemented, this field resets to Ob0.
 - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

res0

ERRIDR_EL1, bit [40]

When FEAT_RAS is implemented:

Trap MRS reads of ERRIDR_EL1 at EL1 using AArch64 to EL2.

ERRIDR_EL1	Meaning
0d0	MRS reads of ERRIDR_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of ERRIDR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

Accessing this field has the following behavior:

- This field is permitted to be RESO if all of the following are true.
 - ERRSELR_EL1 and all ERX* registers are impleme ed as UN FINED of KAZ/WI.
 - ERRIDR_EL1.NUM is zero.

The reset behavior of this field is:

- On a Warm reset:
 - When EL3 is not implemented, this fi _ rese to 0b0.
 - Otherwise, this field resets to an ar _tectural' UNKNOW value.

Otherwise:

res0

ICC_IGRPENn_EL1, bit [39"

When FEAT_GICv3 is `mplementea.

Trap MRS reads of IC _IGRPEN<n>_EL1 . EL1 using AArch64 to EL2.

	C_IGRPENn_EL1	Meaning
	060	MRS reads of ICC_IGRPEN <n>_EL1 are not trapped by this mechanism.</n>
	0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of ICC_IGRPEN <n>_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.</n>

The reset behavior of this field is:

• On a Warm reset, this field resets to Ob0.

Otherwise:

res0

VBAR_EL1, bit [38]

Trap MRS reads of VBAR_EL1 at EL1 using AArch64 to EL2.

VBAR_EL1	Meaning
0b0	MRS reads of VBAR_EL1 are not trapped by this mechanism.
0Ъ1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of VBAR_EL1 at EL1 using AArch64 are trapped and reported with EC syndre i.e value $0x^2$ unless the read generates a higher priority exception.

The reset behavior of this field is:

• On a Warm reset, this field resets to Ob0.

TTBR1_EL1, bit [37]

Trap MRS reads of TTBR1_EL1 at EL1 using Arch6

TTBR. 71	.1 Meaning
060	MRS reads of TTBR1_EL1 are not trapped by this mechanism.
061	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of TTBR1_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset by vior of this field is:

• On a Warm reset, this field resets to Ob0.

TTBR0_EL1, bit [36]

Trap MRS reads of TTBR0_EL1 at EL1 using AArch64 to EL2.

TTBR0_EL1	Meaning
000	MRS reads of TTBR0_EL1 are not trapped by this mechanism.

TTBR0_EL1	Meaning
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of TTBR0_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

• On a Warm reset, this field resets to Ob0.

TPIDR_EL0, bit [35]

Trap MRS reads of TPIDR_EL0 at EL1 and EL0 using AAr .04 and MRC rads c .1PIDRURW at EL0 using AArch32 when EL1 is using AArch64 to EL2.



The reset behavior of this field is:

• On a Warm reset, this field resets to Ob0.

TPIDRRO_EL0, bit [34]

Trap MRS reads of TPIDRRO_EL0 at EL1 and EL0 using AArch64 and MRC reads of TPIDRURO at EL0 using AArch32 when EL1 is using AArch64 to EL2.

Meaning
MRS reads of TPIDRRO_EL0 at EL1 and EL0 using AArch64 and MRC reads of TPIDRURO at EL0 using AArch32 are not trapped by this mechanism.
 If EL2 is implemented and enabled in the curren Security state, HCR_EL2.{E2H, TGE} != {1, 1}, EL1 is using AArch64, and either EL3 is no implemented or SCR_EL3.FGTEn == 1, then, unless the read generates a higher priority exception: MRC LL STPIDRRO_EL0 at EL1 and L0 using A ch64 are trapped to EL2 and reported with EC syndrome value U 8. MRC ands of 7 IDRURO at EL0 using AArch. trapped to EL2 and reported with EC syndrome value 0x03

• On a Warm reset, this field resets 0b0.

TPIDR_EL1, bit [33]

Trap MRS reads of TPIDR_F _1 at 1 using AArc. 54 to EL2.

	FPIDR_EL1	Meaning
	0d0	MRS reads of TPIDR_EL1 are not trapped by this mechanism.
C	Db1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of TPIDR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

• On a Warm reset, this field resets to Ob0.

TCR_EL1, bit [32]

Trap MRS reads of any of the following registers at EL1 using AArch64 to EL2.

- TCR_EL1.
- TCR2_EL1, if FEAT_TCR2 is implemented.

Chapter E3. System registers affected by SME E3.2. Changes to existing System registers

TCR_EL1	Meaning
000	MRS reads of the specified registers are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of the specified registers at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

• On a Warm reset, this field resets to Ob0.

SCXTNUM_EL0, bit [31]

When FEAT_CSV2_2 is implemented or FEAT_CSV2_1, `is j _lementer'

Trap MRS reads of SCXTNUM_EL0 at EL1 and F' g AA. 64 to F

SC. NUM	Meaning
	MRS reads of SCXTNUM_EL0 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, HCR_EL2.{E2H, TGE} != {1, 1}, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of SCXTNUM_EL0 at EL1 and EL0 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The res ¹ .avior of this field is:

> • On a rm reset, this field resets to 0b0.

Otherwise:

res0

SCXTNUM_EL1, bit [30]

When FEAT_CSV2_2 is implemented or FEAT_CSV2_1p2 is implemented:

Trap MRS reads of SCXTNUM_EL1 at EL1 using AArch64 to EL2.

SCXTNUM_EL1	Meaning	
-------------	---------	--

0b0

MRS reads of SCXTNUM_EL1 are not trapped by this mechanism.

SCXTNUM_EL1	Meaning
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of SCXTNUM_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

• On a Warm reset, this field resets to Ob0.

Otherwise:

res0

SCTLR_EL1, bit [29]

- SCTLR_EL1.
- SCTLR2_EL1, if FEAT_SCTLR2 is in .emented

SC1. 7_EL1	Meaning
02	MRS reads of the specified registers are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of the specified registers at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The rese. havior of this field is:

• On a W. reset, this field resets to Ob0.

REVIDR_EL1, bit [28]

Trap MRS reads of REVIDR_EL1 at EL1 using AArch64 to EL2.

REVIDR_EL1	Meaning	
060	MRS reads of REVIDR_EL1 are not trapped by this mechanism.	

REVIDR_EL1	Meaning
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of REVIDR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

• On a Warm reset, this field resets to Ob0.

PAR_EL1, bit [27]

Trap MRS reads of PAR_EL1 at EL1 using AArch64 to EL2.

PAR_EL.	eaning
00.	re is of PAR_EL1 are not trapped by this met anism.
0b1	EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of PAR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior ^e th[;] reld is:

• On ... reset, is field esets to 0b0.

MF 'R_EL1, ,† [26]

Trap MRS nds of MPIDR_EL1 at EL1 using AArch64 to EL2.

MPIDR_EL1	Meaning
0b0	MRS reads of MPIDR_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of MPIDR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

• On a Warm reset, this field resets to Ob0.

MIDR_EL1, bit [25]

Trap MRS reads of MIDR_EL1 at EL1 using AArch64 to EL2.

MIDR_EL1	Meaning
0b0	MRS reads of MIDR_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemer $CR_EL3.FGTEn == 1$, then MRS ress of MID1 FL1 at EL1 using AArch64 are upped to EL2 as reported with EC vindro. value 0x18 inless the read generates a higher p. rity exc_ption.

The reset behavior of this field is:

• On a Warm reset, this field resets to 0b0.

MAIR_EL1, bit [24]

Trap ${\tt MRS}$ reads of MAIR_EL1 at EI 1 usi. AArcı. ' to EL2.

N. VR_EL1	Meaning
060	MRS reads of MAIR_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of MAIR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.
	м. IR_EL1 0b0 0b1

The reset behavior of this field is:

• On a Warm reset, this field resets to Ob0.

LORSA_EL1, bit [23]

When FEAT_LOR is implemented:

Trap MRS reads of LORSA_EL1 at EL1 using AArch64 to EL2.

LORSA_EL1	Meaning	

0b0

MRS reads of LORSA_EL1 are not trapped by this mechanism.

LORSA_EL1	Meaning
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of LORSA_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

• On a Warm reset, this field resets to Ob0.

Otherwise:

res0

LORN_EL1, bit [22]

When FEAT_LOR is implemented:

℃ <u>_</u> EL	1 Aleaning
060	MRS reads of LORN_EL1 are not trapped by this mechanism.
061	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of LORN_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The ret the avior of une field is:

• On a 'arm reset, this field resets to 0b0.

Otherwise:

res0

LORID_EL1, bit [21]

When FEAT_LOR is implemented:

Trap MRS reads of LORID_EL1 at EL1 using AArch64 to EL2.

LORID_EL1	Meaning
060	MRS reads of LORID_EL1 are not trapped by this mechanism.

LORID_EL1	Meaning
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of LORID_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

• On a Warm reset, this field resets to Ob0.

Otherwise:

res0

LOREA_EL1, bit [20]

When FEAT_LOR is implemented:

Trap MRS reads of LOREA_EL1 at EL1 using A .ch64 EL2.

L VF _EL1	feaning
b0	MRS reads of LOREA_EL1 are not trapped by this mechanism.
Obl	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of LOREA_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The ret by avior of all field is:

• On a 'arm reset, this field resets to Ob0.

Otherwise:

res0

LORC_EL1, bit [19]

When FEAT_LOR is implemented:

Trap MRS reads of LORC_EL1 at EL1 using AArch64 to EL2.

LORC_EL1	Meaning
0b0	MRS reads of LORC_EL1 are not trapped by this

mechanism.

LORC_EL1	Meaning
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of LORC_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

• On a Warm reset, this field resets to Ob0.

Otherwise:

res0

ISR_EL1, bit [18]

Trap MRS reads of ISR_EL1 at EL1 using AArch64 to EL2.

ISR_F 1	Me,ing
	MRS reads of ISR_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of ISR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The set behav or of this i. a is:

FAR_EL1, `*[17]

Trap MRS reads of FAR_EL1 at EL1 using AArch64 to EL2.

FAR_EL1	Meaning
0Ь0	MRS reads of FAR_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of FAR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

• On a Warm reset, this field resets to Ob0.

ESR_EL1, bit [16]

Trap MRS reads of ESR_EL1 at EL1 using AArch64 to EL2.

ESR_EL1	Meaning
0b0	MRS reads of ESR_EL1 are not trapped by this mechanism.
0Ъ1	If EL2 is implemented and enabled in the current Security rate, an ither EL3 is not implemented or SC. EL3.FGTEn == 1, then MRS R. 's of ESR_EL at EL1 using AArch64 re trapped to EL2 as reported with EC syndrome verse 0° s, unless the read generates a bill reprior. Exception.

The reset behavior of this field is:

• On a Warm reset, this field resets to ON

DCZID_EL0, bit [15]

Trap MRS reads of DCZID_EL0 &L1 and EL using Arch64 to EL2.

DCZID_EL0	Meaning
000	MRS reads of DCZID_EL0 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, HCR_EL2.{E2H, TGE} != {1, 1}, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of DCZID_EL0 at EL1 and EL0 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

• On a Warm reset, this field resets to Ob0.

CTR_EL0, bit [14]

Trap MRS reads of CTR_EL0 at EL1 and EL0 using AArch64 to EL2.

CTR_EL0	Meaning
060	MRS reads of CTR_EL0 are not trapped by this mechanism.

CTR_EL0	Meaning
0b1	If EL2 is implemented and enabled in the current
	Security state, HCR_EL2.{E2H, TGE} != {1,
	1}, and either EL3 is not implemented or
	SCR_EL3.FGTEn == 1, then MRS reads of
	CTR_EL0 at EL1 and EL0 using AArch64 are
	trapped to EL2 and reported with EC syndrome
	value 0x18, unless the read generates a higher
	priority exception.

• On a Warm reset, this field resets to Ob0.

CSSELR_EL1, bit [13]

Trap MRS reads of CSSELR_EL1 at EL1 using AArch64 tr .L2.



The reset bar vior on s field is

• C . a Wari reset, th. C .d resets to 0b0.

CPAL E', DIT [1-]

Trap MRS re. of CPACR_EL1 at EL1 using AArch64 to EL2.

CPACR_EL1	Meaning
0b0	MRS reads of CPACR_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of CPACR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

• On a Warm reset, this field resets to Ob0.

CONTEXTIDR_EL1, bit [11]

Trap MRS reads of CONTEXTIDR_EL1 at EL1 using AArch64 to EL2.

Meaning
MRS reads of CONTEXTIDR_EL1 are not trapped by this mechanism.
If EL2 is implemented and enabled in the current Security date, and ither EL3 is not implemented or SC. EL3.FGTEn == 1, then MRS rules of CONTE TIDR_EL1 at EL1 using Arch6-, re trapped to EL2 and reported with EC syndron, value Jx18, unless the read generates a high a priority exception.

CLIDR_EL1, bit [10]

The

Trap MRS reads of CLIDR_EL1 *EL1* using rch64 *EL2*.

	CLIDR_EL1	Meaning
	060	MRS reads of CLIDR_EL1 are not trapped by this mechanism.
	0Ъ1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of CLIDR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.
$\mathbf{\overline{v}}$		

The reset behavior of this field is:

• On a Warm reset, this field resets to Ob0.

CCSIDR_EL1, bit [9]

Trap MRS reads of CCSIDR_EL1 at EL1 using AArch64 to EL2.

CCSIDR_EL1	Meaning
0d0	MRS reads of CCSIDR_EL1 are not trapped by
	this mechanism.

CCSIDR_EL1	Meaning
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of CCSIDR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

• On a Warm reset, this field resets to Ob0.

APIBKey, bit [8]

When FEAT_PAuth is implemented:

Trap MRS reads of multiple System registers. Enables a p on MP reads at E 1 using AArch64 of any of the following AArch64 System registers to EL2:

- APIBKeyHi_EL1.
- APIBKeyLo_EL1.

_		
	ıBKey	Meaning
0	060	MRS reads of the System registers listed above are not trapped by this mechanism.
	b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads at EL1 using AArch64 of any of the System registers listed above are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The res ' navior of this field is:

• On a . rm reset, this field resets to Ob0.

Otherwise:

res0

APIAKey, bit [7]

When FEAT_PAuth is implemented:

Trap MRS reads of multiple System registers. Enables a trap on MRS reads at EL1 using AArch64 of any of the following AArch64 System registers to EL2:

- APIAKeyHi_EL1.
- APIAKeyLo_EL1.

APIAKey	Meaning
0b0	MRS reads of the System registers listed above are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads at EL1 using AArch64 of any of the System registers listed above are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

• On a Warm reset, this field resets to Ob0.

Otherwise:

res0

APGAKey, bit [6]

When FEAT_PAuth is implemented:

Trap MRS reads of multiple System registers. following AArch64 System registers to 2:

- APGAKeyHi_EL1.
- APGAKeyLo_EL1.

ab' a trap ... are reads at EL1 using AArch64 of any of the

	APGAKey	Meaning
$\langle \rangle$	0b0	MRS reads of the System registers listed above are not trapped by this mechanism.
	0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads at EL1 using AArch64 of any of the System registers listed above are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

• On a Warm reset, this field resets to Ob0.

Otherwise:

res0

APDBKey, bit [5]

When FEAT_PAuth is implemented:

Trap MRS reads of multiple System registers. Enables a trap on MRS reads at EL1 using AArch64 of any of the following AArch64 System registers to EL2:
- APDBKeyHi_EL1.
- APDBKeyLo_EL1.

APDBKey	Meaning	
0b0	MRS reads of the System registers listed above a not trapped by this mechanism.	
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads at EL1 using AArch64 of any of the System registers listed above are trapped to EL2 and reported	

• On a Warm reset, this field resets to 0b0.

Otherwise:

res0

APDAKey, bit [4]

When FEAT_PAuth is implementer'

Trap MRS reads of multiple Sys' in registers. following AArch64 System 2018 to EL2:

- APDAKeyHi_EL¹
- APDAKeyLo_F 1.

APDAKey	Meaning
0b0	MRS reads of the System registers listed above are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3 EGTEn == 1, then MRS reads at
	EL1 using AArch64 of any of the System
	registers listed above are trapped to EL2 and
	the read generates a higher priority exception.

vables . trap on MRS reads at EL1 using AArch64 of any of the

The reset behavior of this field is:

• On a Warm reset, this field resets to Ob0.

Otherwise:

res0

AMAIR_EL1, bit [3]

Trap MRS reads of AMAIR_EL1 at EL1 using AArch64 to EL2.

	AMAIR_EL1	Meaning
	0d0	MRS reads of AMAIR_EL1 are not trapped by this mechanism.
	0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of AMAIR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.
he reset behavior of this field is:		
• On a Warm reset, this field resets to a	060.	
Frap MRS reads of AIDR_EL1 at EL1 using	g AArch64 to E	
rap MRS reads of AIDR_EL1 at EL1 using	g AArch64 to E	N. Y. ng
YDR_ELT, DR [2]	g AArch64 to Ex $10R_i \frac{1}{10}$	N
Trap MRS reads of AIDR_EL1 at EL1 using	g AArch64 to E.	 N. Ing I.RS reads of AIDR_EL1 are not trapped by this mechanism. If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of AIDR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates
Trap MRS reads of AIDR_EL1 at EL1 using	g AArch64 to Ex IDR_1 1 ``0 Ob1	 Nng RS reads of AIDR_EL1 are not trapped by this mechanism. If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of AIDR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.
Trap MRS reads of AIDR_EL1 at EL1 using	g AArch64 to Ex 	 N. ag As reads of AIDR_EL1 are not trapped by this mechanism. If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of AIDR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.
Trap MRS reads of AIDR_EL1 at EL1 using	g AArch64 to Er .(DR_l ,1 \bar{1} 0b1	N. 19 N. 19 Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of AIDR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

AFSR1_ 1, bit [1]

Trap ${\tt MRS}$ reads of AFSR1_EL1 at EL1 using AArch64 to EL2.

AFSR1_EL1	Meaning
0b0	MRS reads of AFSR1_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of AFSR1_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

• On a Warm reset, this field resets to Ob0.

AFSR0_EL1, bit [0]

Trap MRS reads of AFSR0_EL1 at EL1 using AArch64 to EL2.

AFSR0_EL1	Meaning
0b0	MRS reads of AFSR0_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security rate, an ither EL3 is not implemented or SC. EL3.FGTEn == 1, then MRS R 's of AFSRO_ L1 at EL1 using Arch6- re trapped to EL2 and reported with EC syndron, value $Jx18$, unless the read generates a high priority exception.

The reset behavior of this field is:

• On a Warm reset, this field resets to ON

Accessing HFGRTR_EL2

Accesses to this register use the lowing encyings in the System register encoding space:

MRS <Xt>, HFGRTR_EL(

	op1	CRn	CRm	op2	
<u>0</u> . •	0b100	0b0001	0b0001	0b100	

1	if TATE.E' TO then
2	U FF _D;
3	elsif & ATE.EL == EL1 then
4	<pre>if EL nabled() && HCR_EL2.<nv2,nv> == '11' then</nv2,nv></pre>
5	X[t 64] = NVMem[0x1B8];
6	<pre>elsif EL_Enabled() && HCR_EL2.NV == '1' then</pre>
7	AArch64.SystemAccessTrap(EL2, 0x18);
8	else
9	UNDEFINED;
10	elsif PSTATE.EL == EL2 then
11	<pre>if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3</pre>
	↔trap priority when SDD == '1'" && SCR_EL3.FGTEn == '0' then
12	UNDEFINED;
13	elsif HaveEL(EL3) && SCR_EL3.FGTEn == '0' then
14	<pre>if Halted() && EDSCR.SDD == '1' then</pre>
15	UNDEFINED;
16	else
17	AArch64.SystemAccessTrap(EL3, 0x18);
18	else
19	$X[t, 64] = HFGRTR_EL2;$
20	elsif PSTATE.EL == EL3 then
21	$X[t, 64] = HFGRTR_EL2;$

MSR HFGRTR_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b0001	0b0001	0b100

```
1 if PSTATE.EL == ELO then
2
       UNDEFINED;
3
   elsif PSTATE.EL == EL1 then
       if EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
4
5
           NVMem[0x1B8] = X[t, 64];
       elsif EL2Enabled() && HCR_EL2.NV == '1' then
6
            AArch64.SystemAccessTrap(EL2, 0x18);
7
8
       else
9
            UNDEFINED;
   elsif PSTATE.EL == EL2 then
10
       if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && polea.
                                                                    MPLEMENT
11
                                                                               ION_DEFINED "EL3
            ↔ trap priority when SDD == '1'" && SCR_EL3 JTEn ==
                                                                       then
12
            UNDEFINED;
       elsif HaveEL(EL3) && SCR_EL3.FGTEn == '0' t' .
13
            if Halted() && EDSCR.SDD == '1' then
14
15
               UNDEFINED;
16
            else
17
                AArch64.SystemAccessTrap(EL<sup>2</sup> 0x18)
18
       else
            HFGRTR\_EL2 = X[t, 64];
19
20
   elsif PSTATE.EL == EL3 then
21
    HFGRTR\_EL2 = X[t, 64];
```

E3.2.10 HFGWTR_EL2, Hypervisor Fine-Grained Write Trap Register

The HFGWTR_EL2 characteristics are:

Purpose

Provides controls for traps of MSR and MCR writes of System registers.

Configuration

This register is present only when FEAT_FGT is implemented. Otherwise, direct accesses to HFGWTR_EL2 are UNDEFINED.

Attributes

HFGWTR_EL2 is a 64-bit register.

Field descriptions

The HFGWTR_EL2 bit assignments are:



Bits [63:56]

Reserved, RESO.

nTPIDR2_EL0, bit [55]

When FEAT_SME is implemented:

Trap MSR writes of TPIDR2_EL0 at EL1 and EL0 using AArch64 to EL2.

nTPIDR2_EL0	Meaning
0Ъ0	If EL2 is implemented and enabled in the curren Security state, HCR_EL2.{E2H, TGE} != {1, 1}, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of TPIDR2_EL0 at EL1 and EL0 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generate a higher priority exception.
0b1	MSR writes of TPIDR2_EL0 are not trapped by this mechanism.

The reset behavior of this field is:

- On a Warm reset, this field resets to ${\tt 0b0}.$

Otherwise:

res0

nSMPRI_EL1, bit [54]

When FEAT_SME is implemented:

Trap MSR writes of SMPRI_EL1 at EL1 ing AA 164 to EL2.

nSix	'RI_EL1	Meaning
060		If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of SMPRI_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.
0b1		MSR writes of SMPRI_EL1 are not trapped by this mechanism.

The reset behavior of this field is:

• On a Warm reset, this field resets to Ob0.

Otherwise:

res0

Bits [53:51]

Reserved, RESO.

nACCDATA_EL1, bit [50]

When FEAT_LS64_ACCDATA is implemented:

Trap ${\tt MSR}$ writes of ACCDATA_EL1 at EL1 using AArch64 to EL2.

nACCDATA_EL1	Meaning
060	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of ACCDATA_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.
0b1	MSR writes of ACCDATA_EL1 are not trapped by this mechanism.

• On a Warm reset, this field resets to Ob0.

Otherwise:

res0

ERXADDR_EL1, bit [49]

When FEAT_RAS is implemented:

Trap MSR writes of ERXADDR_EL1 at EL1 u. A ... ch64 w _ .2.

Ek_\DDk_`'\1	Meaning
060	MSR writes of ERXADDR_EL1 are not trapped by this mechanism.
	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of ERXADDR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

Accessing th. Geld has the following behavior:

- This field is permitted to be RESO if all of the following are true:
 - ERRSELR_EL1 and all ERX* registers are implemented as UNDEFINED or RAZ/WI.
 - ERRIDR_EL1.NUM is zero.

The reset behavior of this field is:

- On a Warm reset:
 - When EL3 is not implemented, this field resets to Ob0.
 - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

res0

ERXPFGCDN_EL1, bit [48]

When FEAT_RASv1p1 is implemented:

Trap MSR writes of ERXPFGCDN_EL1 at EL1 using AArch64 to EL2.

	ERXPFGCDN_EL1	Meaning
	060	MSR writes of ERXPFGCDN_EL1 are not trapped by this mechanism.
Accessing this field has the following	0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR write of EK. FGCDN_EL1 at EL1 using AAr 54 are trappe. 5 EL2 and reported with EC sy frome value (18, unless the write generate. higher provide exception.
 This field is permitted to be RES ERRSELR_EL1 and all EF ERRIDR_EL1.NUM is zer 	SO if all of the solution of t	true:
The reset behavior of this field is:		
 On a Warm reset: When EL3 is not imr' .net Otherwise, this fie' re. 	nted, this ¹ d resets to 0be to an archite Prally UNKI). NOWN value.
Otherwise:		

res0

ERXPFGCTL_E⊾ bit .7]

When F AT_ Sv1p1 imr mented:

Tra_ ISR write of FRXPFGCTL_EL1 at EL1 using AArch64 to EL2.

ERXPFGCTL_EL1	Meaning	
0b0	MSR writes of ERXPFGCTL_EL1 are not trapped by this mechanism.	
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of ERXPFGCTL_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.	

Accessing this field has the following behavior:

• This field is permitted to be RESO if all of the following are true:

- ERRSELR_EL1 and all ERX* registers are implemented as UNDEFINED or RAZ/WI.
- ERRIDR_EL1.NUM is zero.

- On a Warm reset:
 - When EL3 is not implemented, this field resets to 0b0.
 - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

res0

Bit [46]

Reserved, RESO.

ERXMISCn_EL1, bit [45]

When FEAT_RAS is implemented:

Trap MSR writes of multiple System registers. Enables a tr on MSR rites at using AArch64 of any of the following AArch64 System registers to EL2:

- ERXMISC0_EL1.
- ERXMISC1_EL1.
- ERXMISC2_EL1.
- ERXMISC3_EL1.



Accessing 's field has the following behavior:

- This field is permitted to be RESO if all of the following are true:
 - ERRSELR_EL1 and all ERX* registers are implemented as UNDEFINED or RAZ/WI.
 - ERRIDR_EL1.NUM is zero.

The reset behavior of this field is:

- On a Warm reset:
 - When EL3 is not implemented, this field resets to Ob0.
 - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

res0

ERXSTATUS_EL1, bit [44]

When FEAT_RAS is implemented:

Trap MSR writes of ERXSTATUS_EL1 at EL1 using AArch64 to EL2.

ERXSTATUS_EL1	Meaning
0d0	MSR writes of ERXSTATUS_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of ERXSTATUS_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates priority exception.

Accessing this field has the following behavior:

- This field is permitted to be RES0 if all of the followin are true:
- ERRSELR_EL1 and all ERX* registers are im emented UNDEFL 3 or RAZ/WI.
 - ERRIDR_EL1.NUM is zero.

The reset behavior of this field is:

- On a Warm reset:
 - When EL3 is not implemented, th. eld re
 - Otherwise, this field resets to arch. arally UNKNOWN value.

Otherwise:

res0

ERXCTLR_EL1, bit [4~]

When FEAT_RA' s impler nted:

Trap MSR writes on RXC LR_EL1 at EL1 using AArch64 to EL2.

ERXCTLR_EL1	Meaning	
060	MSR writes of ERXCTLR_EL1 are not trapped by this mechanism.	
0b1	If EL2 is implemented and enabled in the curren Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of ERXCTLR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.	

Accessing this field has the following behavior:

- This field is permitted to be RESO if all of the following are true:
 - ERRSELR_EL1 and all ERX* registers are implemented as UNDEFINED or RAZ/WI.
 - ERRIDR_EL1.NUM is zero.

The reset behavior of this field is:

• On a Warm reset:

- When EL3 is not implemented, this field resets to Ob0.
- Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

res0

Bit [42]

Reserved, RESO.

ERRSELR_EL1, bit [41]

When FEAT_RAS is implemented:

Trap MSR writes of ERRSELR_EL1 at EL1 using AArch64 to EL2.

ERRSELR_EL1	Meaning
060	MSP rites of L XSELR_EL1 are not trapped ' this mec' anism.
0b1	[•] EL2; inplemented and enabled in the current S [•] y state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then .SR writes of ERRSELR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

Accessing this field has the following to revior:

- This field is ______ mitted t be RESO if all of the following are true:
 - ERRS. R_EL' .nd all ERX* registers are implemented as UNDEFINED or RAZ/WI.
 - ERRIDK_ 'T .NUM is ' .o.

The rest sehal r of th. Geld' ...

On a War reset:

- V en ELS ... ot implemented, this field resets to 0b0.
- therwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

res0

Bit [40]

Reserved, RESO.

ICC_IGRPENn_EL1, bit [39]

When FEAT_GICv3 is implemented:

Trap MSR writes of ICC_IGRPEN<n>_EL1 at EL1 using AArch64 to EL2.

ICC_IGRPENn_EL1	Meaning MSR writes of ICC_IGRPEN <n>_EL1 are not trapped by this mechanism.</n>	
060		
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of ICC_IGRPEN <n>_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.</n>	

• On a Warm reset, this field resets to Ob0.

Otherwise:

res0

VBAR_EL1, bit [38]

Trap MSR writes of VBAR_EL1 at EL1 using Arch64 t EL2.

V. R_EL1	Meaning
b0	MSR writes of VBAR_EL1 are not trapped by this mechanism.
Obl	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of VBAR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The rese havior of this field is:

• On a W n reset, this field resets to 0b0.

TTBR1_EL1, bit [37]

Trap MSR writes of TTBR1_EL1 at EL1 using AArch64 to EL2.

TTBR1_EL1	Meaning	
060	MSR writes of TTBR1_EL1 are not trapped by this mechanism.	

TTBR1_EL1	Meaning
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of TTBR1_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

• On a Warm reset, this field resets to Ob0.

TTBR0_EL1, bit [36]

Trap MSR writes of TTBR0_EL1 at EL1 using AArch64 to EV

TTDDA EL	
	w les of TTBR0_EL1 are not trapped by
O.	EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of TTBR0_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior ^c th[;] reld is:

• On ... reser, is field esets to 0b0.

Trap MSR rites of TPIDR_EL0 at EL1 and EL0 using AArch64 and MCR writes of TPIDRURW at EL0 using AArch32 w. PEL1 is using AArch64 to EL2.

TPIDR_EL0	Meaning
0Ъ0	MSR writes of TPIDR_EL0 at EL1 and EL0 using AArch64 and MCR writes of TPIDRURW at EL0 using AArch32 are not trapped by this mechanism.

	TPIDR_EL0	Meaning
	0b1	 If EL2 is implemented and enabled in the current Security state, HCR_EL2.{E2H, TGE} != {1, 1}, EL1 is using AArch64, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then, unless the write generates a higher priority exception: MSR writes of TPIDR_EL0 at EL1 and EL0 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18. MCR for of TPIDRURW at EL0 using A crch32 a, rapped to EL2 and reported with EC syndrometed syndrometed with EC syndrometed syndromet
he reset behavior of this field is:		
• On a Warm reset, this field resets to Ob	o.	
PIDERO ELO bit 1241		

TPIDRRO_EL0, bit [34]

√ ch64 to LL2. Trap MSR writes of TPIDRRO_EL0 at EV1 usin.

TP, RRO_, 0	Meaning
060	MSR writes of TPIDRRO_EL0 are not trapped by this mechanism.
	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of TPIDRRO_EL0 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset beh. 'or of this field is:

• On a Warm reset, this field resets to Ob0.

TPIDR_EL1, bit [33]

Trap MSR writes of TPIDR_EL1 at EL1 using AArch64 to EL2.

TPIDR_EL1	Meaning
0b0	MSR writes of TPIDR_EL1 are not trapped by this mechanism.

TPIDR_EL1	Meaning
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of TPIDR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

• On a Warm reset, this field resets to Ob0.

TCR_EL1, bit [32]

Trap MSR writes of any of the following registers at EL1 using Arch64 to .

- TCR_EL1.
- TCR2_EL1, if FEAT_TCR2 is implemented.

ſ	CR_1 .1	Nng
31		ASR writes of the specified registers are not trapped by this mechanism.
Ok	51	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of the specified registers at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The react behave r of this ' is:

• 'n a W 'his field resets to Ob0.

SCXTNU. EL0, bit [31]

When FEAT_CSV2_2 is implemented or FEAT_CSV2_1p2 is implemented:

Trap MSR writes of SCXTNUM_EL0 at EL1 and EL0 using AArch64 to EL2.

SCXTNUM_EL0	Meaning
040	MSR writes of SCXTNUM_EL0 are not trapped by this mechanism.

SCXTNUM_EL0	Meaning
0b1	If EL2 is implemented and enabled in the current Security state, HCR_EL2.{E2H, TGE} != {1, 1}, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of SCXTNUM_EL0 at EL1 and EL0 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception

• On a Warm reset, this field resets to Ob0.

Otherwise:

res0

SCXTNUM_EL1, bit [30]

When FEAT_CSV2_2 is implemented or FEAT ... 1p2 is volen ced:

Trap MSR writes of SCXTNUM_EL1 at EL1 ing AAr 64 to EL2.

CXTN_`{_EL1	Meaning
060	MSR writes of SCXTNUM_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of SCXTNUM_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The rese. havior of this field is:

• On a W. reset, this field resets to Ob0.

Otherwise:

res0

SCTLR_EL1, bit [29]

Trap MSR writes of any of the following registers at EL1 using AArch64 to EL2.

• SCTLR_EL1.

• SCTLR2_EL1, if FEAT_SCTLR2 is implemented.

SCTLR_EL1	Meaning
060	MSR writes of the specified registers are not trapped by this mechanism.

SCTLR_EL1	Meaning
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of the specified registers at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:

• On a Warm reset, this field resets to Ob0.

Bit [28]

Reserved, RESO.

PAR_EL1, bit [27]

Trap MSR writes of PAR_EL1 at EL1 using AArch6 EL2.

PAR (1	Meaning
	MSR writes of PAR_EL1 are not trapped by this mechanism.
51	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of PAR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The set beh? or of this field is:

• C. (arm reset, this field resets to 0b0.

Bits [26:25]

Reserved, RESO.

MAIR_EL1, bit [24]

Trap ${\tt MSR}$ writes of MAIR_EL1 at EL1 using AArch64 to EL2.

MAIR_EL1	Meaning
0b0	MSR writes of MAIR_EL1 are not trapped by this
	mechanism.

MAIR_EL1	Meaning
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of MAIR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

• On a Warm reset, this field resets to Ob0.

LORSA_EL1, bit [23]

When FEAT_LOR is implemented:

Trap MSR writes of LORSA_EL1 at EL1 using AArch64 CL2.

50 MSR V	vrites of LORSA EL1 are not trapped by
	nechanism.
0b1 If EL Secu imple MSR V AArc EC s gene	2 is implemented and enabled in the current rity state, and either EL3 is not emented or SCR_EL3.FGTEn == 1, then vrites of LORSA_EL1 at EL1 using th64 are trapped to EL2 and reported with yndrome value $0x18$, unless the write rates a higher priority exception.

The reset or of ... field :

• n a War reset, this eld resets to Ob0.

Other `~e?

res0

LORN_EL1, bit [22]

When FEAT_LOR is implemented:

Trap ${\tt MSR}$ writes of LORN_EL1 at EL1 using AArch64 to EL2.

LORN_EL1	Meaning
060	MSR writes of LORN_EL1 are not trapped by this mechanism.

LORN_EL1	Meaning
0b1	If EL2 is implemented and enabled in the current
	Security state, and either EL3 is not
	implemented or SCR_EL3.FGTEn == 1, then
	MSR writes of LORN_EL1 at EL1 using
	AArch64 are trapped to EL2 and reported with
	EC syndrome value 0x18, unless the write
	generates a higher priority exception.

• On a Warm reset, this field resets to Ob0.

Otherwise:

res0

Bit [21]

Reserved, RESO.

LOREA_EL1, bit [20]

When FEAT_LOR is implemented:

Trap MSR writes of LOREA_EL1 at EL1 ing Ar. h64 to EL2.

LL FA_EL1	Meaning
060	MSR writes of LOREA_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of LOREA_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:

• On a Warm reset, this field resets to Ob0.

Otherwise:

res0

LORC_EL1, bit [19]

When FEAT_LOR is implemented:

Trap MSR writes of LORC_EL1 at EL1 using AArch64 to EL2.

LORC_EL1	Meaning
0Ъ0	MSR writes of LORC_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of LORC_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

• On a Warm reset, this field resets to Ob0.

Otherwise:

res0

Bit [18]

Reserved, RESO.

FAR_EL1, bit [17]

Trap MSR writes of FAR_EL1 at EJ asing Arch64 EL2.

h. R_EL1	Meaning
060	MSR writes of FAR_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of FAR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:

• On a Warm reset, this field resets to Ob0.

ESR_EL1, bit [16]

Trap ${\tt MSR}$ writes of ESR_EL1 at EL1 using AArch64 to EL2.

ESR_EL1	Meaning
0b0	MSR writes of ESR_EL1 are not trapped by this mechanism.

ESR_EL1
0b1

The reset behavior of this field is:

• On a Warm reset, this field resets to Ob0.

Bits [15:14]

Reserved, RESO.

CSSELR_EL1, bit [13]

Trap MSR writes of CSSELR_EL1 at EL1 using AA 4 to E.

SELR 1	Meaning
060	MSR writes of CSSELR_EL1 are not trapped by this mechanism.
Obi	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of CSSELR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The set beh? or of this field is:

• C. varm reset, this field resets to 0b0.

CPACR_EL. hit [12]

Trap MSR writes of CPACR_EL1 at EL1 using AArch64 to EL2.

CPACR_EL1	Meaning
0b0	MSR writes of CPACR_EL1 are not trapped by this mechanism.

CPACR_EL1	Meaning
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of CPACR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

• On a Warm reset, this field resets to Ob0.

CONTEXTIDR_EL1, bit [11]

Trap MSR writes of CONTEXTIDR_EL1 at EL1 using AArch to EL2.

0b0 Image: winder software in the second		CONTEXTIDR_EL1 .eaning
0b1 . EL2 is implemented and enabled in the Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == MSR writes of CONTEXTIDR_EL1 at AArch64 are trapped to EL2 and report	not	0b0 w .es of CONTEXTIDR_H trap_ed by this mechanism.
EC syndrome value 0x18, unless the v generates a higher priority exception.	1, then L1 using ted with rite	0b1 . EL2 is implemented and enable Security state, and either EL3 is implemented or SCR_EL3.FGT MSR writes of CONTEXTIDR_F AArch64 are trapped to EL2 an EC syndrome value 0x18, unles generates a higher priority exce

The reset behavio, ^e th[;] leld is:

• On ... reset, is field esets to 0b0.

Bit 10:9]

Reserved 'ESO.

APIBKey, b. '9]

When FEAT_PAuth is implemented:

Trap MSR writes of multiple System registers. Enables a trap on MSR writes at EL1 using AArch64 of any of the following AArch64 System registers to EL2:

- APIBKeyHi_EL1.
- APIBKeyLo_EL1.

APIBKey	Meaning
0b0	MSR writes of the System registers listed above are not trapped by this mechanism.

APIBKey	Meaning
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes at EL1 using AArch64 of any of the System registers listed above are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

• On a Warm reset, this field resets to Ob0.

Otherwise:

res0

APIAKey, bit [7]

When FEAT_PAuth is implemented:

Trap MSR writes of multiple System registers. F toles trap on v tes at EL1 using AArch64 of any of the following AArch64 System registers to EL2:

- APIAKeyHi_EL1.
- APIAKeyLo_EL1.

PIAKey	Meaning
060	MSR writes of the System registers listed above are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes at EL1 using AArch64 of any of the System registers listed above are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:

• On a Warm reset, this field resets to Ob0.

Otherwise:

res0

APGAKey, bit [6]

When FEAT_PAuth is implemented:

Trap MSR writes of multiple System registers. Enables a trap on MSR writes at EL1 using AArch64 of any of the following AArch64 System registers to EL2:

- APGAKeyHi_EL1.
- APGAKeyLo_EL1.

APGAKey	Meaning
0b0	MSR writes of the System registers listed above are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes at EL1 using AArch64 of any of the System registers listed above are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

• On a Warm reset, this field resets to Ob0.

Otherwise:

res0

APDBKey, bit [5]

When FEAT_PAuth is implemented:

Trap MSR writes of multiple System registers. following AArch64 System registers to 2:

- APDBKeyHi_EL1.
- APDBKeyLo_EL1.

ab' a trap ... ASR writes at EL1 using AArch64 of any of the

	APDBKey	Meaning	
	0b0	MSR writes of the System registers listed above are not trapped by this mechanism.	
2	0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes at EL1 using AArch64 of any of the System registers listed above are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.	

The reset behavior of this field is:

• On a Warm reset, this field resets to Ob0.

Otherwise:

res0

APDAKey, bit [4]

When FEAT_PAuth is implemented:

Trap MSR writes of multiple System registers. Enables a trap on MSR writes at EL1 using AArch64 of any of the following AArch64 System registers to EL2:

Chapter E3. System registers affected by SME E3.2. Changes to existing System registers

- APDAKeyHi_EL1.
- APDAKeyLo_EL1.

APDAKey	Meaning		
000	MSR writes of the System registers listed above are not trapped by this mechanism.		
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes at EL1 using AArch64 of any of the System registers listed above are trapped to EL2 and reported to the system value 0x18, unless the write generates thigher priority exception.		

The reset behavior of this field is:

- On a Warm reset, this field resets to ${\tt 0b0}.$

Otherwise:

res0

AMAIR_EL1, bit [3]

Trap MSR writes of AMAIR_EL1 at FL1 . g AA. 64 to EL2.

	M. TR_EL1	Meaning
0	b0	MSR writes of AMAIR_EL1 are not trapped by this mechanism.
	bl	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of AMAIR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:

• On a Warm reset, this field resets to Ob0.

Bit [2]

Reserved, RESO.

AFSR1_EL1, bit [1]

Trap MSR writes of AFSR1_EL1 at EL1 using AArch64 to EL2.

	AFSR1_EL1	Meaning
	000	MSR writes of AFSR1_EL1 are not trapped by this mechanism.
	0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of AFSR1_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.
The reset behavior of this field is:		
• On a Warm reset, this field rese	ets to Ob0.	
AFSR0 EL1. bit [0]		
Trap MSR writes of AFSR0_EL1 at E	L1 using AArch64 to .	2.
	A' SRO_E	N. ng
	0.	SR writes of AFSR0_EL1 are not trapped by this mechanism.
	1	If EL 2 is implemented and enabled in the curren
		Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of AFSR0 EL1 at EL1 using

רח a W∕ • this field resets to Ob0.

Access. 7 HFGWTR_EL2

Accesses to this register use the following encodings in the System register encoding space:

MRS <Xt>, HFGWTR_EL2

орО	op1	CRn	CRm	op2
0b11	0b100	0b0001	0b0001	0b101

```
if PSTATE.EL == ELO then
1
2
      UNDEFINED;
3
   elsif PSTATE.EL == EL1 then
4
       if EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
5
           X[t, 64] = NVMem[0x1C0];
6
       elsif EL2Enabled() && HCR_EL2.NV == '1' then
```

Chapter E3. System registers affected by SME E3.2. Changes to existing System registers



MSR HFGWTR_EL2, <Xt>

		op0	op1	CR	[¬] Rm	op2
		0b11	0b100	7001	060001	0b101
1						
2	INDEFINED.	- ELO CHEN				
2	ONDEFINED;	T FT1 +h				
4	if EL2Enab	led() && HC	'R EL2 <n 2.nv=""></n>	11' then		
5	NVMem [$0 \times 1 C 0 1 = X 0$	+ 641·			
6	elsif EL2E	nabled() &&	HCT _L2.NV = '1	' n		
7	AArch6	4.SystemAcc	er .rap(EL2, 18);		
8	else	-				
9	UNDEFI	NED;				
10	elsif PSTATE.E	L == E ¹ 2 th	ien	*		
11	if Halted() && .veEI	SCR.SD: د (EL3) אל (EL3)	D == '1' && boole	an IMPLEMENTATIC	N_DEFINED "EL3
	⇔trap	p' _rity v	/hen SDD =- 1'" &	& SCR_EL3.FGTEn =	= '0' then	
12	UNDEFI	۲. T				
13	elsif Have	(EL3)	SCR_EL3.FGTEn ==	'0' then		
14	if Hal	te V .ED	SCR. D == '1' the	en		
15	UN.	DEFL 0;				
16	ise					
17	A	rch64.Sy	AccessTrap(EL3,	0x18);		
10	'Ise		641.			
20	oloif 'F F		04],			
20	HEGN EL2	= X[+ 64]				
21	III OW HIL	M[C, 04]	,			

E3.2.11 ID_AA64ISAR1_EL1, AArch64 Instruction Set Attribute Register 1

The ID_AA64ISAR1_EL1 characteristics are:

Purpose

Provides information about the features and instructions implemented in AArch64 state.

For general information about the interpretation of the ID registers, see Principles of the ID scheme for fields in ID registers .

Attributes

ID_AA64ISAR1_EL1 is a 64-bit register.

Field descriptions

The ID_AA64ISAR1_EL1 bit assignments are:



LS64, bits [63:60]

Indicates support for LD64B and ST6^(*) * instrumous, and the ACCDATA_EL1 register. Defined values of this field are:

	US64	Meaning
	000000	The LD64B, ST64B, ST64BV, and ST64BV0 instructions, the ACCDATA_EL1 register, and associated traps are not supported.
	0b0001	The LD64B and ST64B instructions are supported.
	0b0010	The LD64B, ST64B, and ST64BV instructions, and their associated traps are supported.
	0b0011	The LD64B, ST64B, ST64BV, and ST64BV0 instructions, the ACCDATA_EL1 register, and their associated traps are supported.
The second secon		

All other values are reserved.

FEAT_LS64 implements the functionality identified by 0b0001.

FEAT_LS64_V implements the functionality identified by 0b0010.

FEAT_LS64_ACCDATA implements the functionality identified by 0b0011.

From Armv8.7, the permitted values are 0b0000, 0b0001, 0b0010, and 0b0011.

XS, bits [59:56]

Indicates support for the XS attribute, the TLBI and DSB instructions with the nXS qualifier, and the HCRX_EL2.{FGTnXS, FnXS} fields in AArch64 state. Defined values are:

XS	Meaning
000000	The XS attribute, the TLBI and DSB instructions with the nXS qualifier, and the HCRX_EL2.{FGTnXS, FnXS} fields are not supported.
060001	The XS attribute, the TLBI and DSB instructions with the nXS qualifier, and the HCRX_EL2.{FGTnXS, FnXS} fields are supported.

FEAT_XS implements the functionality identified by 0b0001.

From Armv8.7, the only permitted value is 0b0001.

I8MM, bits [55:52]

Indicates support for Advanced SIMD and Floating-point h me x multiplication instructions in AArch64 state. Defined values are:

I8M J	Meaning
0000d	Int8 matrix multiplication instructions are not implemented.
060001	SMMLA, SUDOT, UMMLA, USMMLA, and USDOT instructions are implemented.

All other values ar *c*served.

FEAT_I8MM im, ment he functionality identified by 0b0001.

When $A^{d} \rightarrow d S_{1} \rightarrow d S_{2} \rightarrow E$ are both implemented, this field must return the same value as ID_AA +ZFRt EL1.18. It

Fro. Armv8 ¹¹ only permitted value is 0b0001.

DGH, L (51:48]

Indicates support for the Data Gathering Hint instruction. Defined values are:

DGH	Meaning
060000	Data Gathering Hint is not implemented.
0b0001	Data Gathering Hint is implemented.

All other values are reserved.

FEAT_DGH implements the functionality identified by 0b0001.

From Armv8.0, the permitted values are 0b0000 and 0b0001.

If the DGH instruction has no effect in preventing the merging of memory accesses, the value of this field is 0b0000.

BF16, bits [47:44]

Indicates support for Advanced SIMD and Floating-point BFloat16 instructions in AArch64 state. Defined values are:

BF16	Meaning
060000	BFloat16 instructions are not implemented
0b0001	BFCVT, BFCVTN, BFCVTN2, BFDOT, BFMLALB, BFMLALT, and BFMMLA instructions consimplemented.
0b0010	As 0b ^o 1, but the PCR.EBF field is also sup, ted.

All other values are reserved.

FEAT_BF16 adds the functionality identified by 0b0001.

FEAT_EBF16 adds the functionality identified by

When FEAT_SVE or FEAT_SME is *i*.plement 1, this *i*d must return the same value as ID_AA64ZFR0_EL1.BF16.

10.

From Armv8.6 and Armv9.1, the value 9000 h. t permitted.

SPECRES, bits [43:40]

Indicates support for predice in in Vidation instructions in AArch64 state. Defined values are:

SPECRES	Meaning	
000000	Prediction invalidation instructions are not implemented.	
0b0001	CFP RCTX, DVP RCTX and CPP RCTX instructions are implemented.	

All other v. es are reserved.

FEAT_SPECR_S implements the functionality identified by 0b0001.

From Armv8.5, the value 0b0000 is not permitted.

SB, bits [39:36]

Indicates support for SB instruction in AArch64 state. Defined values are:

SB	Meaning
060000	SB instruction is not implemented.
0b0001	SB instruction is implemented.

FEAT_SB implements the functionality identified by 0b0001.

In Armv8.0, the permitted values are 0b0000 and 0b0001.

From Armv8.5, the only permitted value is 0b0001.

FRINTTS, bits [35:32]

Indicates support for the FRINT32Z, FRINT32X, FRINT64Z, and FRINT64X instructions are implemented. Defined values are:

	FRINTTS	Meaning
	00000	FRINT ² 2, FRI. ³ 2X, FRINT64Z, and FRI ^N o4X instruct. s are not implemented.
	0b0001	FRIN, °Z, FRINT3 , FRINT64Z, and r'RINT64. 'nstruct' is are implemented.
All other values are reserved.		
FEAT_FRINTTS implements the functionality	ir' .aane by Ob	u 11.
From Armv8.5, the only permitted value is C	001.	

GPI, bits [31:28]

Indicates support for an IMPLEN INTATIO DEFIN. algorithm is implemented in the PE for generic code authentication in AArch64 sta⁺ Defined value re:

GPI	Meaning
00000	Generic Authentication using an IMPLEMENTATION DEFINED algorithm is not implemented.
0b0001	Generic Authentication using an IMPLEMENTATION DEFINED algorithm is implemented. This includes the PACGA instruction.

All other values are reserved.

FEAT_PACIMP implements the functionality identified by 0b0001.

From Armv8.3, the permitted values are 0b0000 and 0b0001.

If the value of ID_AA64ISAR1_EL1.GPA is nonzero, or the value of ID_AA64ISAR2_EL1.GPA3 is nonzero, this field must have the value 0b0000.

GPA, bits [27:24]

Indicates whether the QARMA5 algorithm is implemented in the PE for generic code authentication in AArch64 state. Defined values are:

GPA	Meaning
060000	Generic Authentication using the QARMA5 algorithm is not implemented.
0b0001	Generic Authentication using the QARMA5 algorithm is implemented. This includes the PACGA instruction.

FEAT_PACQARMA5 implements the functionality identified by 0b0001.

From Armv8.3, the permitted values are 0b0000 and 0b0001.

If the value of ID_AA64ISAR1_EL1.GPI is nonzero, or the value of V_AA64ISAR2_EL1.GPA3 is nonzero, this field must have the value 0b0000.

LRCPC, bits [23:20]

Indicates support for weaker release consistency, RCpc, e od movil. Defined values are:

LR C	ar' g
٥٢	^P Cpc instructions are not implemented.
`b0001	The no offset LDAPR, LDAPRB, and LDAPRH instructions are implemented.
060010	As 0b0001, and the LDAPR (unscaled immediate) and STLR (unscaled immediate) instructions are implemented.

All other values reserv

FEAT_LRCPC imply into the fr initiality identified by the value 0b0001.

FEAT .XCPC. mplem. . . e functionality identified by the value 0b0010.

Fron Armv8 _____0b0000 is not permitted.

From A. 8.4, the value 0b0001 is not permitted.

FCMA, bits ?:16]

Indicates support for complex number addition and multiplication, where numbers are stored in vectors. Defined values are:

FCMA	Meaning
000000	The FCMLA and FCADD instructions are not implemented.
0b0001	The FCMLA and FCADD instructions are implemented.

All other values are reserved.

FEAT_FCMA implements the functionality identified by the value 0b0001.

In Armv8.0, Armv8.1, and Armv8.2, the only permitted value is 0b0000.

From Armv8.3, if Advanced SIMD or Floating-point is implemented, the only permitted value is 0b0001.

From Armv8.3, if Advanced SIMD or Floating-point is not implemented, the only permitted value is 0b0000.

JSCVT, bits [15:12]

Indicates support for JavaScript conversion from double precision floating point values to integers in AArch64 state. Defined values are:

	JSCVT	Meaning
	00000	The FJC 1ZS II. Action is not implemented.
	0b0001	The CVTZS instrution is implemented.
All other values are reserved.		
FEAT_JSCVT implements the functionality identify	fied by U \9)'
In Armv8.0, Armv8.1, and Armv8.2, the only per	'alue i	- ` 6000 ^

From Armv8.3, if Advanced SIMD or Float; ,-point is nplemente, , the only permitted value is 0b0001.

From Armv8.3, if Advanced SIMD or Floating is not implemented, the only permitted value is 0b0000.

API, bits [11:8]

24

Indicates whether an IMPLEM TION DEFINE algorithm is implemented in the PE for address authentication, in AArch64 state. This app s to a Pointer Authentication instructions other than the PACGA instruction. Defined values are:

	API	Meaning			
	000000	Address Authentication using an IMPLEMENTATION DEFINED algorithm is not implemented.			
	060001	Address Authentication using an IMPLEMENTATION DEFINED algorithm is implemented, with the HaveEnhancedPAC() and HaveEnhancedPAC2() functions returning FALSE.			
	060010	Address Authentication using an IMPLEMENTATION DEFINED algorithm is implemented, with the HaveEnhancedPAC() function returning TRUE, and the HaveEnhancedPAC2() function returning FALSE.			
	0b0011	Address Authentication using an IMPLEMENTATION DEFINED algorithm is implemented, with the HaveEnhancedPAC2() function returning TRUE, and the HaveEnhancedPAC() function returning FALSE.			

API	Meaning
0Ь0100	Address Authentication using an IMPLEMENTATION DEFINED algorithm is implemented, with the HaveEnhancedPAC2() function returning TRUE, the HaveFPAC() function returning TRUE, the HaveFPACCombined() function returning FALSE, and the HaveEnhancedPAC() function returning FALSE.
0Ъ0101	Address Authentication using an IMPLEMENTATION DEFINED algorithm is implemented, in the HaveEnhancedPAC2() function returning Ture, the Mayer Combined function returning

FEAT_PAuth implements the functionality id stified by b0001.

FEAT_EPAC implements the functionality ide. 'fie _y 0b001...

FEAT_PAuth2 implements the functional. identity is 0b0011.

FEAT_FPAC implements the fr dionality ide. 'fied by Jb0100.

FEAT_FPACCOMBINE ir .emei. the functiona .ty identified by 0b0101.

When this field is non- o, FEAT_PAC. 'P is implemented.

In Armv8.3, the p ______nitted v^{___} res are 0b0001, 0b0010, 0b0011, 0b0100, and 0b0101.

From Armv8.6, the err led value are 0b0011, 0b0100, and 0b0101.

If the value of ID_AA64ISAR2_EL1.APA3 is nonzero, or the value of ID_AA64ISAR2_EL1.APA3 is nonzero, this fina must live the value of 00000.

APA hits [7]

Indicates whether the QARMA5 algorithm is implemented in the PE for address authentication, in AArch64 state. This applies will Pointer Authentication instructions other than the PACGA instruction. Defined values are:

APA	Meaning
060000	Address Authentication using the QARMA5 algorithm is not implemented.
060001	Address Authentication using the QARMA5 algorithm is implemented, with the HaveEnhancedPAC() and HaveEnhancedPAC2() functions returning FALSE.
0b0010	Address Authentication using the QARMA5 algorithm is implemented, with the HaveEnhancedPAC() function returning TRUE and the HaveEnhancedPAC2() function returning FALSE.

Copyright © 2022-2023 Arm Limited or its affiliates. All rights reserved. Non-confidential

APA	Meaning
0b0011	Address Authentication using the QARMA5 algorithm is implemented, with the HaveEnhancedPAC2() function returning TRUE, the HaveFPAC() function returning FALSE, the HaveFPACCombined() function returning FALSE, and the HaveEnhancedPAC() function returning FALSE.
0b0100	Address Authentication using the QARMA5 algorithm is implemented, with the HaveEnhancedPAC2() function returning TRUE, the HaveF acCombin. '() function returning FAL ⁻² and the Hav ⁻ nhancedPAC() function rturnin FALSE.
0b0101	Address Automatic Lon using the QARMA5 algo thm is in the commented, with the ViveEnhancedPAC2() function returning TRUE, the HaveFinAC() function returning TRUE, the tweFinACCombined() function returning TK 1, and the HaveEnhancedPAC() function returning FALSE.

FEAT_PAuth implements the an. onality idem. d by 0b0001.

FEAT_EPAC implements the functio. 'ity identified by 0b0010.

FEAT_PAuth2 imp¹ ...ents the functional. identified by 0b0011.

FEAT_FPAC in, ments' functionality identified by 0b0100.

FEAT_FPACCOMB. implem is the functionality identified by 0b0101.

When is field nonzers $r \int PACQARMA5$ is implemented.

In A v8.3, values are 0b0001, 0b0010, 0b0011, 0b0100, and 0b0101.

From A. 8.6, the permitted values are 0b0011, 0b0100, and 0b0101.

If the value o. <code>___AA64ISAR1_EL1.API</code> is nonzero, or the value of ID__AA64ISAR2_EL1.APA3 is nonzero, this field must have the value 0b0000.

DPB, bits [3:0]

Data Persistence writeback. Indicates support for the DC CVAP and DC CVADP instructions in AArch64 state. Defined values are:

DPB	Meaning
060000	DC CVAP not supported.
0b0001	DC CVAP supported.
0b0010	DC CVAP and DC CVADP supported.

FEAT_DPB implements the functionality identified by the value 0b0001.

FEAT_DPB2 implements the functionality identified by the value 0b0010.

In Armv8.2, the permitted values are 0b0001 and 0b0010.

From Armv8.5, the only permitted value is 0b0010.

Accessing ID_AA64ISAR1_EL1

Accesses to this register use the following encodings in the System register encoding space:

MRS <Xt>, ID_AA64ISAR1_EL1

		орО	op1	CRn	CRm	ор2			
		0b11	0b000	0b000′	0b0110	0b001			
1	if PSTATE.EL	== ELO then							
2	<pre>if IsFeat</pre>	ureImplement	ed(FEAT_IDST) the	n					
3	if EL	2Enabled() &	& HCR_EL2.TGE ==	hen					
4	AArch64.SystemAccessTrap(EL 0x18)								
5	5 else								
6	AArch64.SystemAccessTrap(1, 0x1								
7	else								
8	UNDEF	INED;							
9	elsif PSTATE.	EL == EL1 th	en						
10	if EL2Ena	bled() && HC	R_FTID3 = '1'	th					
11	AArch	64.SystemAcc	Trap(EL2, 0. ?);					
12	else								
13	X[t,	$64] = ID_{A.6}$	4ISAK TL1;						
14	elsif PSTATE.	EL == F 2 th	en						
15	X[t, 64]	= ID .64ISA	R1_EL1;						
16	elsif PSTATE.	EL · EL3 t'	en						
17	X[t, 64]	D_AA647 .	R1_EL1;						
E3.2.12 ID_AA64PFR1_EL1, AArch64 Processor Feature Register 1

The ID_AA64PFR1_EL1 characteristics are:

Purpose

Reserved for future expansion of information about implemented PE features in AArch64 state.

For general information about the interpretation of the ID registers, see Principles of the ID scheme for fields in ID registers .

Attributes

ID_AA64PFR1_EL1 is a 64-bit register.

Field descriptions

The ID_AA64PFR1_EL1 bit assignments are:



Bits [63:40]

Reserved, RESO.

NMI, bits [39:36]

Non-maskable Interrupt Indicates su, ort for Non-maskable interrupts. Defined values are:

NMI	Meaning
000000	SCTLR_ELx.{SPINTMASK, NMI} and PSTATE.ALLINT with its associated instructions are not supported.
0b0001	SCTLR_ELx.{SPINTMASK, NMI} and PSTATE.ALLINT with its associated instructions are supported.

All other values are reserved.

FEAT_NMI implements the functionality identified by the value 0b0001.

From Armv8.8, the only permitted value is 0b0001.

CSV2_frac, bits [35:32]

CSV2 fractional field. Defined values are:

CSV2_frac	Meaning	
000040	Either ID_AA64PFR0_EL1.CSV2 is not 0b0001, or the implementation does not disclose whether FEAT_CSV2_1p1 is implemented. FEAT_CSV2_1p2 is not implemented.	
0b0001	FEAT_CSV2_1p1 is implemented, but FEAT_CSV2_1p2 is not implemented.	
0b0010	FEAT_CSV2_1p2 is implemented.	

All other values are reserved.

FEAT_CSV2_1p1 implements the functionality identified by the value _00001.

FEAT_CSV2_1p2 implements the functionality identified by the value '0010.

From Armv8.0, the permitted values are 0b0000, 0b0001, an^a 00010.

The values 0b0001 and 0b0010 are permitted only when ' _AA64F _k0_EL1. v2 is 0b0001.

RNDR_trap, bits [31:28]

Random Number trap to EL3 field. Defined vp' es are:

	Meaning
0000	Trapping of RNDR and RNDRRS to EL3 is not supported.
060001	Trapping of RNDR and RNDRRS to EL3 is supported. SCR_EL3.TRNDR is present.

All other values are inved.

FEAT MG_T AP imp. v .s the functionality identified by the value 0b0001.

SM. hits [?

Scalable A trix Extension. Defined values are:

SME	Meaning
00000	SME architectural state and programmers' model are not implemented.
0b0001	SME architectural state and programmers' model are implemented.
0b0010	As 0b0001, plus the SME2 ZT0 register.

All other values are reserved.

FEAT_SME implements the functionality identified by the value 0b0001.

FEAT_SME2 implements the functionality identified by the value 0b0010.

From Armv9.2, the permitted values are 0b0000, 0b0001, and 0b0010.

If implemented, refer to ID_AA64SMFR0_EL1 and ID_AA64ZFR0_EL1 for information about which SME and SVE instructions are available.

Bits [23:20]

Reserved, RESO.

MPAM_frac, bits [19:16]

Indicates the minor version number of support for the MPAM Extension.

Defined values are:

Meanir
The nor version nuber of the MPAM stensio. 0.
The minor we ice number of the MPAM ext asion is 1.

All other values are reserved.

When combined with the major version num. from *F* ----- *APFR0_EL1.MPAM*, The combined "major.minor" version is:

MPAM	MPAM_frac
0b0000	0Ь0000
0b0000	0b0001
0b0001	0b0000
0b0001	0b0001
	МРАМ 0b0000 0b0000 0b0001 0b0001

For ore infor ation, see the Memory Partitioning and Monitoring (MPAM) Extension .

RAS_. _____ Jits [15:12]

RAS Extens. fractional field. Defined values are:

RAS_frac	Meaning
060000	If ID_AA64PFR0_EL1.RAS == 0b0001, RAS Extension implemented.

RAS_frac	Meaning	
0b0001	If ID_AA64PFR0_EL1.RAS == 0b0001, as	
	0b0000 and adds support for:	
	 Additional ERXMISC<m>_EL1 System</m> 	
	registers.	
	Additional System registers	
	ERXPFGCDN EL1, ERXPFGCTL EL	
	and ERXPFGF EL1, and the	
	SCR EL3.FIEN and HCR EL2.FIEN	
	trap controls, to support the optional RA	
	Common Fault Injection Model Extensio	
	Error reco [*] essed through System register	
	conforr .o RAS, tem Architecture v1.1.	
	whic includes sime lications to	
	ERR< STATUS, an support for the optional	
	AS Tip, tamp and AS Common Fault	
	Injection M 1 F ansions	

All other values are reserved.

FEAT_RASv1p1 implements the functionality Jentified by the value of 00001.

000

This field is valid only if ID_AA64PFR0_EL_PAS

MTE, bits [11:8]

Support for the Memory Taggir Extension. L Ined values are:

MTE	Meaning
060000	Memory Tagging Extension is not implemented.
0b0001	Instruction-only Memory Tagging Extension is implemented.
0b0010	Full Memory Tagging Extension is implemented.
060011	Memory Tagging Extension is implemented with support for asymmetric Tag Check Fault handling.

All other values are reserved.

FEAT_MTE implements the functionality identified by the value 0b0001.

FEAT_MTE2 implements the functionality identified by the value 0b0010.

FEAT_MTE3 implements the functionality identified by the value 0b0011.

In Armv8.5, the permitted values are 0b0000, 0b0001, 0b0010, and 0b0011.

From Armv8.7, the value 0b0010 is not permitted.

SSBS, bits [7:4]

Speculative Store Bypassing controls in AArch64 state. Defined values are:

SSBS	Meaning
060000	AArch64 provides no mechanism to control the use of Speculative Store Bypassing.
0b0001	AArch64 provides the PSTATE.SSBS mechanism to mark regions that are Speculative Store Bypass Safe.
0b0010	As 0b0001, and adds the MSR and MRS instructions to directly read and write the PSTATE.SSBS field.

All other values are reserved.

FEAT_SSBS implements the functionality identified by the value 0t _____1.

FEAT_SSBS2 implements the functionality identified by the $v_2^{\prime} \ge 0b00$.

BT, bits [3:0]

Branch Target Identification mechanism support in AArchton ter . Defined alues are:

BT	M
0000	The Branch Target Identification mechanism is not implemented.
0bu 1	The Branch Target Identification mechanism is implemented.

All other values are *cerved*.

FEAT_BTI imply tents the unctionality identified by the value 0b0001.

From Army⁸ 5, the permitter' alue is 0b0001.

Acce sing _____AAt 7 1_EL1

Acce. s to is regional set the following encodings in the System register encoding space:

MRS <X1. 'D_AA64PFR1_EL1

орО	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0100	0b001

```
if PSTATE.EL == EL0 then
1
        \quad \text{if} \text{ IsFeatureImplemented(FEAT_IDST)} \ \text{then}
2
3
            if EL2Enabled() && HCR_EL2.TGE == '1' then
4
                 AArch64.SystemAccessTrap(EL2, 0x18);
5
             else
6
                 AArch64.SystemAccessTrap(EL1, 0x18);
        else
7
8
             UNDEFINED;
9
   elsif PSTATE.EL == EL1 then
10
        if EL2Enabled() && HCR_EL2.TID3 == '1' then
11
             AArch64.SystemAccessTrap(EL2, 0x18);
```

Copyright © 2022-2023 Arm Limited or its affiliates. All rights reserved. Non-confidential

Chapter E3. System registers affected by SME E3.2. Changes to existing System registers

E3.2.13 ID_AA64ZFR0_EL1, SVE Feature ID register 0

The ID_AA64ZFR0_EL1 characteristics are:

Purpose

Provides additional information about the implemented features of the AArch64 Scalable Vector Extension instruction set, when one or more of FEAT_SVE and FEAT_SME is implemented.

For general information about the interpretation of the ID registers, see Principles of the ID scheme for fields in ID registers .

Configuration

Prior to the introduction of the features described by this register, this register was unnamed and reserved, RESO from EL1, EL2, and EL3.

If FEAT_SME is implemented and FEAT_SVE is not implement , then SVE . tructions can only be executed when the PE is in Streaming SVE mode and the instructions are legal to execute in Streaming SVE mode.

Attributes

ID_AA64ZFR0_EL1 is a 64-bit register.

Field descriptions

The ID_AA64ZFR0_EL1 bit assignments a

63	60	59 5	6 55		² 51		48	47	44 43	40	39		36	35	32
	RES0	F64MM	Γ.	-32MM		RES		I8MM		SM4		RES0		SHA3	
31			23		 20 15		16	15		8	7		4	3	0
	RE	50		BF16	В	ir i	~m		RES0			AES		SVEve	r

Bits [63:60]

Reserved, RESO.

F64MM ... `9:56j

Indicates supplation SVE Fr64 double-precision floating-point matrix multiplication instructions. Defined values are:

F64MM	Meaning
00000	Double-precision matrix multiplication and related SVE instructions are not implemented.
0b0001	Double-precision variant of the FMMLA instruction, and the LD1RO* instructions are implemented. The 128-bit element variants of the SVE TRN1, TRN2, UZP1, UZP2, ZIP1, and ZIP2 instructions are also implemented.

All other values are reserved.

FEAT_F64MM implements the functionality identified by 0b0001.

From Armv8.2, the permitted values are 0b0000 and 0b0001.

Chapter E3. System registers affected by SME E3.2. Changes to existing System registers

When the PE is in Streaming SVE mode and it is not known whether FEAT_SME_FA64 is implemented and enabled, software should not attempt to execute the instructions described by nonzero values of this field, irrespective of the value of this field.

F32MM, bits [55:52]

Indicates support for the SVE FP32 single-precision floating-point matrix multiplication instruction. Defined values are:

Meaning
Single-precision matrix multiplication instruction is not implemented.
Single-r LISION viant of the FMMLA instruction is impleduated.

All other values are reserved.

FEAT_F32MM implements the functionality identified L_ vb0001

From Arm v8.2, the permitted values are 0b0000 and 0b0001

When the PE is in Streaming SVE mode and is not lown w. * FEAT_SME_FA64 is implemented and enabled, software should not attempt to e cute the instructions described by nonzero values of this field, irrespective of the value of this field.

Bits [51:48]

Reserved, RESO.

I8MM, bits [47:44]

Indicates support for VE Int8 matrix mu plication instructions. Defined values are:

I8MM	Meaning
000000	SVE Int8 matrix multiplication instructions are not implemented.
0b0001	SVE SMMLA, SUDOT, UMMLA, USMMLA, and USDOT instructions are implemented.

All other values are reserved.

FEAT_I8MM implements the functionality identified by 0b0001.

When Advanced SIMD and SVE are both implemented, this field must return the same value as ID_AA64ISAR1_EL1.I8MM.

From Armv8.6, the only permitted value is 0b0001.

When the PE is in Streaming SVE mode and it is not known whether FEAT_SME_FA64 is implemented and enabled, software should not attempt to execute the SVE instructions SMMLA, UMMLA, and USMMLA, irrespective of the value of this field.

SM4, bits [43:40]

Indicates support for SVE SM4 instructions. Defined values are:

SM4	Meaning
00000	SVE SM4 instructions are not implemented.
060001	SVE SM4E and SM4EKEY instructions are implemented.

All other values are reserved.

FEAT_SVE_SM4 implements the functionality identified by 0b0001.

When the PE is in Streaming SVE mode and it is not known whether FEAT_SME_FA64 is implemented and enabled, software should not attempt to execute the instructions described by nonzero values of this field, irrespective of the value of this field.

Bits [39:36]

Reserved, RESO.

SHA3, bits [35:32]

Indicates support for the SVE SHA3 instructions. Defined v. are:

SH/	Me, aing
0000م.	SVE SHA3 instructions are not implemented.
0. 01	SVE RAX1 instruction is implemented.

All other values are reserved.

FEAT_SVE_SHA3 . .plements the funct. ality identified by 0b0001.

When the PE is Stream g SVE mode and it is not known whether FEAT_SME_FA64 is implemented and enabled, software y not attend to execute the instructions described by nonzero values of this field, irrespective the value of this field.

Bits / (:24]

Reserv 2.50.

BF16, bits `3:20]

Indicates support for SVE BFloat16 instructions. Defined values are:

BF16	Meaning
0000d0	SVE BFloat16 instructions are not implemented.
0b0001	SVE BFCVT, BFCVTNT, BFDOT, BFMLALB, BFMLALT, and BFMMLA instructions are implemented.
0b0010	As 0b0001, but the FPCR.EBF field is also supported.

All other values are reserved.

FEAT_BF16 adds the functionality identified by 0b0001.

FEAT_EBF16 adds the functionality identified by 0b0010.

This field must return the same value as ID_AA64ISAR1_EL1.BF16.

When the PE is in Streaming SVE mode and it is not known whether FEAT_SME_FA64 is implemented and enabled, software should not attempt to execute the SVE instruction BFMMLA, irrespective of the value of this field.

From Armv8.6 and Armv9.1, the value 0b0000 is not permitted.

BitPerm, bits [19:16]

Indicates support for SVE bit permute instructions. Defined values are:

BitPerm	Mear .g
060000	VE by rmute inst ctions are not
0b0001	SV ^T 3DEP, B f, and BGRP instructions are
	plemente 1.

All other values are reserved.

FEAT_SVE_BitPerm implements the function. 'ty' Intified by Jb0001.

When the PE is in Streaming SVF mode nd it is pt known whether FEAT_SME_FA64 is implemented and enabled, software should not at ...npt to e. pute the instructions described by nonzero values of this field, irrespective of the value of this 1⁻⁴.

Bits [15:8]

Reserved, RESO.

AES, bits [7:4]

Indicates + for S F AES structions. Defined values are:

AES	Meaning
000000	SVE AES* instructions are not implemented.
0b0001	SVE AESE, AESD, AESMC, and AESIMC instructions are implemented.
060010	As 0b0001, plus 64-bit source element variants of SVE PMULLB and PMULLT instructions are implemented.

All other values are reserved.

FEAT_SVE_AES implements the functionality identified by the value 0b0001.

FEAT_SVE_PMULL128 implements the functionality identified by the value 0b0010.

The permitted values are 0b0000 and 0b0010.

When the PE is in Streaming SVE mode and it is not known whether FEAT_SME_FA64 is implemented and

Chapter E3. System registers affected by SME E3.2. Changes to existing System registers

enabled, software should not attempt to execute the instructions described by nonzero values of this field, irrespective of the value of this field.

SVEver, bits [3:0]

Indicates support for SVE instructions when one or more of FEAT_SME and FEAT_SVE is implemented. Defined values are:

		SVEver	Meaning			
		000000	The SVE	instructions are	implemented.	
		0b0001	As 0b0000, and adds the mandatory SVE2 instructions			
All other values are reserved.						
From Armv9. if this register i	s present, the value	0b0000 is n	Jermitted.			
FEAT SVE2 implements the	functionality identif	fied by $\beta = \sqrt{0}$)1 wh the	PE 1 Jt in Str	eaming SVE mode.	
FEAT_SME implements the	functionality identifi	led by Ubuc	hen the F	is in Streami	ing SVE mode.	
Accessing ID AA64ZI	R0 EL1					
	_					
Accesses to this register use t	he following en、 ¹i	ngs	registe	r encoding space	ce:	
MR5 <xi>, ID_AA042FR0_</xi>						
0		CD		CD		
оро	<u>op1</u>	CK		CRM	op2	
0b1	0L 79	0b00	00	0b0100	0b100	
	<i>Y</i>					
if PSTATE.EL == 0 .en	od AT IDST) +h	~ n				
.r E. Enable)	ICR_EL2.TGE ==	'1' then				
A ch64.Sy	AccessTrap(EL2,	0x18);				
else						
a)	<pre>mAccessTrap(EL1,</pre>	0x18);				
DEFINED;						
elsif PSTA EL == EL1 tl	ıen					
if EL2En ,led() && (1	ísFeatureImplemen [.]	ted(FEAT FO	T) !Is7	ero(ID AA647F	780 FL1)	
→boolean IMPLEM	1					
	ENTATION_DEFINED	"ID_AA64ZF1	RO_EL1 trap	ped by HCR_E	L2.TID3") &&	
Arch64 Sustema	ENTATION_DEFINED = '1' then	"ID_AA64ZF1	RO_EL1 trap	pped by HCR_EI	L2.TID3") &&	
AArch64.SystemAcc	ENTATION_DEFINED = '1' then cessTrap(EL2, 0x1	"ID_AA64ZF1 8);	RO_EL1 trap	pped by HCR_E	L2.TID3") &&	
AArch64.SystemAcc else X[t, 64] = ID_AA	ENTATION_DEFINED = '1' then cessTrap(EL2, 0x1 ;4ZFR0_EL1;	"ID_AA64ZF1 8);	RO_EL1 trap	pped by HCR_E	L2.TID3") &&	
AArch64.SystemAcc else X[t, 64] = ID_AA(elsif PSTATE.EL == EL2 tl	ENTATION_DEFINED = '1' then cessTrap(EL2, 0x1 54ZFR0_EL1; ten	"ID_AA64ZF1 8);	RO_EL1 trap	ped by HCR_E	L2.TID3") &&	
AArch64.SystemAcc else X[t, 64] = ID_AA4 elsif PSTATE.EL == EL2 tl X[t, 64] = ID_AA64ZFI	ENTATION_DEFINED = '1' then cessTrap(EL2, 0x1 542FR0_EL1; ten {0_EL1;	"ID_AA64ZF1 8);	RO_EL1 trap	ped by HCR_E	L2.TID3") &&	

16 elsif PSTATE.EL == EL3 then
17 X[t, 64] = ID_AA64ZFR0_EL1;

E3.2.14 MPAM2_EL2, MPAM2 Register (EL2)

The MPAM2_EL2 characteristics are:

Purpose

Holds information to generate MPAM labels for memory requests when executing at EL2.

Configuration

This register has no effect if EL2 is not enabled in the current Security state.

When EL3 is implemented, AArch64 system register MPAM2_EL2 bit [63] is architecturally mapped to AArch64 system register MPAM3_EL3[63].

AArch64 system register MPAM2_EL2 bit [63] is architecturally mapped to AArch64 system register MPAM1_EL1[63].

This register is present only when FEAT_MPAM is implemented. Otherwine direct accesses to MPAM2_EL2 are UNDEFINED.

Attributes

MPAM2_EL2 is a 64-bit register.

Field descriptions

The MPAM2_EL2 bit assignments are:



MPAMEN [63]

MPA r Enable MPAM is abled when MPAMEN == 1. When disabled, all PARTIDs and PMGs are output as then fault corresponding ID space.

MPAMEN	Meaning
060	The default PARTID and default PMG are output in MPAM information from all Exception levels.
0b1	MPAM information is output based on the MPAMn_ELx register for ELn according to the MPAM configuration.

If EL3 is not implemented, this field is read/write.

If EL3 is implemented, this field is read-only and reads the current value of the read/write MPAM3_EL3.MPAMEN bit.

The reset behavior of this field is:

• On a Warm reset, this field resets to Ob0.

Accessing this field has the following behavior:

- **RW** if !HaveEL(EL3)
- Otherwise, access to this field is RO

Bits [62:59]

Reserved, RESO.

TIDR, bit [58]

When (FEAT_MPAMv0p1 is implemented or FEAT_MPAMv1p1 is implemented) and MPAMIDR_EL1.HAS_TIDR == 1:

TIDR traps accesses to MPAMIDR_EL1 from EL1 to EL2.

TIDR	Меа тд
0b0	This contrations recause any instructions to be traped.
0b1	τ μ accesses to MPAMIDR_EL1 from EL1

MPAMHCR_EL2.TRAP_MPAMIDR_EL1 = 1 also ______MIDR_EL1 accesses from EL1 to EL2. If either TIDR or TRAP_MPAMIDR_EL1 are 1 accesses ______trapped.

The reset behavior of this field is:

• On a Warm reset, this fie' resets to an arc 'ecturally UNKNOWN value.

Otherwise:

res0

Bit [57]

Reserved

ALT' _HFC, t [56]

Hierarchica. pree of alternative PARTID space controls. When MPAM3_EL3.ALTSP_HEN is 0, ALTSP controls in MPAM2_EL_ have no effect. When MPAM3_EL3.ALTSP_HEN is 1, this bit selects whether the PARTIDs in MPAM1_EL1 and MPAM0_EL1 are in the primary (0) or alternative (1) PARTID space for the security state.

ALTSP_HFC	Meaning
060	When MPAM3_EL3.ALTSP_HEN is 1, the PARTID space of MPAM1_EL1.PARTID_I, MPAM1_EL1.PARTID_D, MPAM0_EL1.PARTID_I, and MPAM0_EL1.PARTID_D are in the primary PARTID space for the Security state.

0b1 When MPAM3_EL3.ALTSP_HEN is 1, the PARTID space of MPAM1_EL1.PARTID_I, MPAM1_EL1.PARTID_D, MPAM0_EL1.PARTID_I, and MPAM0_EL1.PARTID_D are in the alternat PARTID space for the Security state.	, tive

This control has no effect when MPAM3_EL3.ALTSP_HEN is 0.

For more information, see 'Alternative PARTID spaces and selection' in Arm® Architecture Reference Manual Supplement, Memory System Resource Partitioning and Monitoring (MP _____, Armv8-A (ARM DDI 0598).

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNV OWN 'ue.

Otherwise:

res0

ALTSP_EL2, bit [55]

When FEAT_RME is implemented and M. MIDR / 1 HAS ALTSP == 1:

Select alternative PARTID space for PATIDs IN AM2_EL2 when MPAM3_EL3.ALTSP_HEN is 1.

A ^T SP_Ł <i>L</i> 2	Meaning
060	When MPAM3_EL3.ALTSP_HEN is 1, selects the primary PARTID space for MPAM2_EL2.PARTID_I and MPAM2_EL2.PARTID_D.
0b1	When MPAM3_EL3.ALTSP_HEN is 1, selects the alternative PARTID space for MPAM2_EL2.PARTID_I and MPAM2_EL2.PARTID_D.

For more in tration, see 'Alternative PARTID spaces and selection' in Arm® Architecture Reference Manual Supplement, N. mory System Resource Partitioning and Monitoring (MPAM), for Armv8-A (ARM DDI 0598).

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

res0

ALTSP_FRCD, bit [54]

When FEAT_RME is implemented and MPAMIDR_EL1.HAS_ALTSP == 1:

Alternative PARTID forced for PARTIDs in this register.

ALTSP_FRCD	Meaning
060	The PARTIDs in this register are using the primary PARTID space.
0b1	The PARTIDs in this register are using the alternative PARTID space.

This bit indicates that a higher Exception level has forced the PARTIDs in this register to use the alternative PARTID space defined for the current Security state. In EL2, it is also 1 when MPAM2_EL2.ALTSP_EL2 is 1.

For more information, see 'Alternative PARTID spaces and selection' in Arm® Architecture Reference Manual Supplement, Memory System Resource Partitioning and Monitoring (MPA^M) for Armv8-A (ARM DDI 0598).

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOW alue.

Access to this field is **RO**.

Otherwise:

res0

Bits [53:51]

Reserved, RESO.

EnMPAMSM, bit [50]

When FEAT_SME is implem

Traps execution at EL1 of instruction that directly access the MPAMSM_EL1 register to EL2. The exception is reported using ESR_F^{r} . EC value $0x_1$

EnMPAMSM	Meaning
0b0	This control causes execution of these instructions at EL1 to be trapped.
0b1	This control does not cause execution of any instructions to be trapped.

This field has no effect on accesses to MPAMSM_EL1 from EL2 or EL3.

d:

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

res0

TRAPMPAM0EL1, bit [49]

Trap accesses from EL1 to the MPAM0_EL1 register trap to EL2.

TRAPMPAM0EL1	Meaning
060	Accesses to MPAM0_EL1 from EL1 are not trapped.
0b1	Accesses to MPAM0_EL1 from EL1 are trapped to EL2.

The reset behavior of this field is:

- On a Warm reset:
 - When EL3 is not implemented, this field resets to Ob1.
 - When EL3 is implemented, this field resets to an architecturally UNIVOWN value.

TRAPMPAM1EL1, bit [48]

Trap accesses from EL1 to the MPAM1_EL1 register trap to E^V



The reset behavior of this field

- On a Warm reset:
 - When EL3 is st implement, this field resets to Ob1.
 - When EL² s implemented, this old resets to an architecturally UNKNOWN value.

PMG_D, bits [4. '7]

Perform .cen utorin, rour or data accesses.

The set beha or of this field is:

• C arm reset, mis field resets to an architecturally UNKNOWN value.

PMG_I, bits '9:32]

Performance monitoring group for instruction accesses.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

PARTID_D, bits [31:16]

Partition ID for data accesses, including load and store accesses, made from EL2.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

PARTID_I, bits [15:0]

Partition ID for instruction accesses made from EL2.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing MPAM2_EL2

None of the fields in this register are permitted to be cached in a TLB.

Accesses to this register use the following encodings in the System register encoding space:

MRS <Xt>, MPAM2_EL2

op0	op1	CRn	CRm	op2
0b11	0b100	0b1_0	'h010-	0b000

```
if PSTATE.EL == ELO then
 1
2
         UNDEFINED;
3
    elsif PSTATE.EL == EL1 then
4
         if EL2Enabled() && HCR_EL2.NV ==
                                                     then
5
              if HaveEL(EL3) && MPAM3_EL3.1. LOWE
                                                                     <u>'``n</u>
6
                   if Halted() && EDSCR. DD ==
                                                         Lhen
 7
                       UNDEFINED;
8
                  else
9
                       AArch64.Sys
                                      .nAccessT.
                                                    (EL3,
                                                             :18);
10
              else
11
                  AArch64.Sys .nAc.
                                         sTrap(EL2,
                                                     :18);
12
         else
              UNDEFINED;
13
                           LL2 then
14
    elsif PSTATE.EL =
         if HaveEL(F' / && MP'M3_EL3.TRAPLOWER == '1' then
    if Hat 'd() && JSCR.SDD == '1' then
15
16
17
                  UND.
                         TNF
18
              el co
                     rch64 vstem cessTrap(EL3, 0x18);
19
20
         e
21
                    4] = MPAN__EL2;
             Xít,
22
    els.
         PSTAT
                              then
23
                   = MPAMz_EL2;
         X<sub>1</sub>
```

MSR MPAn. EL2, <Xt>

орО	op1	CRn	CRm	op2
0b11	0b100	0b1010	0b0101	0b000

```
if PSTATE.EL == ELO then
1
       UNDEFINED;
2
3
   elsif PSTATE.EL == EL1 then
4
       if EL2Enabled() && HCR_EL2.NV == '1' then
5
           if HaveEL(EL3) && MPAM3_EL3.TRAPLOWER == '1' then
               if Halted() && EDSCR.SDD == '1' then
6
7
                   UNDEFINED;
8
               else
9
                   AArch64.SystemAccessTrap(EL3, 0x18);
```

Copyright © 2022-2023 Arm Limited or its affiliates. All rights reserved. Non-confidential

Chapter E3. System registers affected by SME E3.2. Changes to existing System registers

```
10
             else
11
                 AArch64.SystemAccessTrap(EL2, 0x18);
12
        else
13
            UNDEFINED;
14
    elsif PSTATE.EL == EL2 then
15
        if HaveEL(EL3) && MPAM3_EL3.TRAPLOWER == '1' then
            if Halted() && EDSCR.SDD == '1' then
16
17
                 UNDEFINED;
18
             else
19
                 AArch64.SystemAccessTrap(EL3, 0x18);
20
        else
21
    MPAM2_EL2 = X[t, 64];
elsif PSTATE.EL == EL3 then
22
23
      MPAM2\_EL2 = X[t, 64];
```

MRS <Xt>, MPAM1_EL1

op0	op1	CRn	CRm	op2	
0b11	0b000	0b* 10	010	0b000	

```
1
    if PSTATE.EL == ELO then
 2
         UNDEFINED;
 3
     elsif PSTATE.EL == EL1 then
          if HaveEL(EL3) && MPAM3_EL3.TRAP<sup>*</sup> VER ==
 4
                                                                       +hen
               if Halted() && EDSCR.SDD == '1 'her
 5
 6
                     UNDEFINED;
 7
                else
 8
                    AArch64.SystemAc _sTrap ~3, 0x );
          elsif EL2Enabled() && M2_EL2.Tk MPAM1k I == '1' then
 9
          AArch64.SystemAcc rap(EL2, 0x ';
elsif EL2Enabled() + HCK 12.<NV2,NV1,NV> == '111' then
10
11
12
               X[t, 64] = N^{v}Mem[0x900]
13
          else
    else
    X[t, 64] MPAM1_EL1;
elsif PSTATE.E' = EL2 + .n
    if HaveEL(3) && AM3_EL3.TRAPLOWER == '1' then
        if Halte ) . EDSCR.c D == '1' then
        'WDEF1 D;
14
15
16
17
18
19
                .1se
           A :ch64.Sy #AccessTrap(EL3, 0x18);

1sif HC /EL2.E2H == '1' then
20
21
               X/ ..., M2_EL2;
22
23
          el.
    elsif PSTA EL == EL3 then
X[t, 64] MPAM1_EL1;
24
25
26
```

MSR MPAM1_EL1, <Xt>

орО	op1	CRn	CRm	op2
0b11	0b000	0b1010	0b0101	0b000

```
1 if PSTATE.EL == EL0 then
2 UNDEFINED;
3 elsif PSTATE.EL == EL1 then
4 if HaveEL(EL3) && MPAM3_EL3.TRAPLOWER == '1' then
5 if Halted() && EDSCR.SDD == '1' then
6 UNDEFINED;
```

Chapter E3. System registers affected by SME E3.2. Changes to existing System registers

```
7
            else
8
                AArch64.SystemAccessTrap(EL3, 0x18);
9
        elsif EL2Enabled() && MPAM2_EL2.TRAPMPAM1EL1 == '1' then
10
            AArch64.SystemAccessTrap(EL2, 0x18);
        elsif EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '111' then
11
12
            NVMem[0x900] = X[t, 64];
13
        else
14
            MPAM1\_EL1 = X[t, 64];
15
   elsif PSTATE.EL == EL2 then
        if HaveEL(EL3) && MPAM3_EL3.TRAPLOWER == '1' then
16
17
            if Halted() && EDSCR.SDD == '1' then
18
                UNDEFINED;
19
            else
20
                AArch64.SystemAccessTrap(EL3, 0x18);
        elsif HCR_EL2.E2H == '1' then
    MPAM2_EL2 = X[t, 64];
21
22
23
        else
24
            MPAM1\_EL1 = X[t, 64];
25
   elsif PSTATE.EL == EL3 then
      MPAM1\_EL1 = X[t, 64];
26
```

E3.2.15 SCR_EL3, Secure Configuration Register

The SCR_EL3 characteristics are:

Purpose

Defines the configuration of the current Security state. It specifies:

- The Security state of EL0, EL1, and EL2. The Security state is Secure, Non-secure, or Realm.
- The Execution state at lower Exception levels.
- Whether IRQ, FIQ, SError interrupts, and External abort exceptions are taken to EL3.
- Whether various operations are trapped to EL3.

Configuration

This register is present only when EL3 is implemented. Otherwise. CCCesses to SCR_EL3 are UNDEFINED.

Attributes

SCR_EL3 is a 64-bit register.

Field descriptions

The SCR_EL3 bit assignments are:



Bit [63]

Reserved, RESO.

NSE, bit [62]

When FEAT_RME is implemented

NSE, bit [62]

This field, evaluated with SCR_EL3.NS, selects the Security state of EL2 and lower Exception levels.

For a description of the values derived by evaluating NS and NSE together, see SCR_EL3.NS.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise

NSE, bit [62]

Reserved, RESO, and the Effective value of this bit is 0b0.

Bits [61:50]

Reserved, RESO.

MECEn, bit [49]

When FEAT_MEC is implemented:

Enables access to the following EL2 MECID registers, from EL2:

- MECID_P0_EL2.
- MECID_A0_EL2
- MECID_P1_EL2
- MECID_A1_EL2
- VMECID_P_EL2
- VMECID_A_EL2

Accesses to these registers are trapped and a orted us' ron ESR_EL3.EC value of 0x18.

प्CEn	Meaning
060	Accesses from EL2 to a listed MECID register are trapped to EL3. The value of a listed EL2 MECID register is treated as 0 for all purposes other than direct reads or writes to the register from EL3.
0b1	This control does not cause any instructions to be trapped.

¹ field is: The et beh

• On Varm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

res0

GPF, bit [48]

When FEAT_RME is implemented:

Controls the reporting of Granule protection faults at EL0, EL1 and EL2.

GPF	Meaning
0b0	This control does not cause exceptions to be
	routed from EL0, EL1 or EL2 to EL3.

GPF	Meaning
0b1	GPFs at EL0, EL1 and EL2 are routed to EL3 and reported as Granule Protection Check exceptions.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

res0

Bits [47:45]

Reserved, RESO.

SCTLR2En, bit [44]

When FEAT_SCTLR2 is implemented:

SCTLR2_ELx register trap control. Enables acce ... TLR2_TL1 ar SCTLR2_EL2 registers.

SCTY	Meaning
060	EL1 and EL2 accesses to SCTLR2_EL1 and SCTLR2_EL2 registers are disabled, and trapped to EL3. The values in these registers are treated as 0.
0b1	This control does not cause any instructions to be trapped.

an ESR _____3.EC value of 0x18. Traps are remoted us.

Traps c not ta n if the nigher priority exception generated by the access.

[•] field is: The vet be

• On Varm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

res0

TCR2En, bit [43]

When FEAT_TCR2 is implemented:

TCR2_ELx register trap control. Enables access to TCR2_EL1 and TCR2_EL2 registers.

TCR2En	Meaning
060	EL1 and EL2 accesses to TCR2_EL1 and TCR2_EL2 registers are disabled, and trapped to EL3. The values in these registers are treated as 0.

Chapter E3. System registers affected by SME E3.2. Changes to existing System registers

TCR2En	Meaning	
0b1	This control does not cause any instructions to be trapped.	

Traps are reported using an ESR_EL3.EC value of 0x18.

Traps are not taken if there is a higher priority exception generated by the access.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

res0

Bit [42]

Reserved, RESO.

EnTP2, bit [41]

When FEAT_SME is implemented:

Traps instructions executed at EL2, EL1, a. EL0 the traps TPIDR2_EL0 to EL3. The exception is reported using ESR_ELx.EC value 0x18.

Ел 72	Meaning
0dt	This control causes execution of these instructions at EL2, EL1, and EL0 to be trapped.
0b1	This control does not cause execution of any instructions to be trapped.

The rest sehal r of th. Geld

On a War reset, this rield resets to an architecturally UNKNOWN value.

Otherw

res0

TRNDR, bit [40]

When FEAT_RNG_TRAP is implemented:

Controls trapping of reads of RNDR and RNDRRS. The exception is reported using ESR_ELx.EC value 0x18.

Chapter E3. System registers affected by SME E3.2. Changes to existing System registers



HXEn	Meaning
0b0	Accesses at EL2 to HCRX_EL2 are trapped to EL3. Indirect reads of HCRX_EL2 return 0.
0b1	This control does not cause any instructions to be trapped.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

res0

ADEn, bit [37]

When FEAT_LS64_ACCDATA is implemented:

Enables access to the ACCDATA_EL1 register at EL1 and EL2.

ADEn	Meaning
0b0	Accesses to ACCDATA_EL1 at EL1 and EL2 are trapped to EL3, unless the accesses are trapped to EL2 by the EL2 fine-grained trap.
0b1	This control does not cause accesses to ACCDATA CL1 to be trapped.

If the HFGWTR_EL2.nACCDATA_EL1 or HFGRTR_EL2.nACCDA1. EL1 traps are mabled, they take priority over this trap.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturall, 'NK' WN valu^

Otherwise:

res0

EnAS0, bit [36]

When FEAT_LS64_ACCDATA ; mpleme. 4:

Traps execution of an ST64P J. truction at EL EL1, or EL2 to EL3.

	EnAS0	Meaning
	0Ъ0	EL0 execution of an ST64BV0 instruction is trapped to EL3, unless it is trapped to EL1 by SCTLR_EL1.EnAS0, or to EL2 by either HCRX_EL2.EnAS0 or SCTLR_EL2.EnAS0. EL1 execution of an ST64BV0 instruction is trapped to EL3, unless it is trapped to EL2 by HCRX_EL2.EnAS0. EL2 execution of an ST64BV0 instruction is trapped to EL3.
	0b1	This control does not cause any instructions to be trapped.

A trap of an ST64BV0 instruction is reported using an ESR_ELx.EC value of 0x0A, with an ISS code of 0x0000001. The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

res0

AMVOFFEN, bit [35]

When FEAT_AMUv1p1 is implemented:

Activity Monitors Virtual Offsets Enable.

	AMVOFFEN	Meaning
	060	Accesses to AMEVCNTVOFF0 <n>_EL2 and AMEVCNTVOFF1<n>_EL2 at EL2 are trapped to EL3. Indirect reads of the virtual offset registers are zero.</n></n>
	0b1	Accesses to AMEVCNTVOFF0 <n>_EL2 and AMEVC (TVO). <n>_EL2 are not affected by this f .d.</n></n>
The most behavior of this field in		
The reset behavior of this field is:		
• On a Warm reset, this field resets to	an architectura. UN	KN vN value.
Otherwise:		
res0		

When FEAT_TME is implemente

Enables access to the TSTART COMMIT, T. ST and TCANCEL instructions at EL0, EL1 and EL2.

TME	Meaning
060	EL0, EL1 and EL2 accesses to TSTART, TCOMMIT, TTEST and TCANCEL instructions are UNDEFINED.
0b1	This control does not cause any instruction to be UNDEFINED.

The reset by vior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

TME, bit [34]

res0

TWEDEL, bits [33:30]

When FEAT_TWED is implemented:

TWE Delay. A 4-bit unsigned number that, when SCR_EL3.TWEDEn is 1, encodes the minimum delay in taking a trap of WFE* caused by SCR EL3.TWE as $2^{(TWEDEL + 8)}$ cycles.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

res0

TWEDEn, bit [29]

When FEAT_TWED is implemented:

TWE Delay Enable. Enables a configurable delayed trap of the WFE* instruction caused by SCR_EL3.TWE. Traps are reported using an ESR_ELx.EC value of 0x01.

	TWEDEn	Meaning
	060	The delation of the trap is IMPLY LENTATION FFINED.
	0b1	The a v for taking trap is at least the .umber vcles def ed in SCR EL3 /ED/
The reset behavior of this field is:		
• On a Warm reset, this field resets to an ar	tectur: y UN	KN value.
Otherwise:		
RES0		
ECVEn, bit [28]		
When FEAT_ECV is implementea.		
ECV Enable. Enable ccess to the CN1 OFF_El	L2 register.	
	ECVEn	Meaning
	0ದ0	EL2 accesses to CNTPOFF_EL2 are trapped to EL3, and the value of CNTPOFF_EL2 is treated as 0 for all purposes other than direct reads or writes to the register from EL3.
	0b1	EL2 accesses to CNTPOFF_EL2 are not trapped to EL3 by this mechanism.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

res0

FGTEn, bit [27]

When FEAT_FGT is implemented:

Fine-Grained Traps Enable. When EL2 is implemented, enables the traps to EL2 controlled by HAFGRTR_EL2, HDFGRTR_EL2, HDFGRTR_EL2, HFGITR_EL2, and HFGWTR_EL2, and controls access to

those registers.

If EL2 is not implemented but EL3 is implemented, FEAT_FGT implements the MDCR_EL3.TDCC traps.

	FGTEn	Meaning
	060	EL2 accesses to HAFGRTR_EL2, HDFGRTR_EL2, HDFGWTR_EL2, HFGRTR_EL2, HFGITR_EL2 and HFGWTR_EL2 registers are trapped to EL3, and the traps to EL2 controlled by those registers are disabled.
	0Ь1	EL2 accesses to HAFGRTR_EL2, HDFGP K_EL2, 'DFGWTR_EL2, HFC rR_EL2, HF TR_EL2 and HFG, 'R_EL2 regises reare not trapped to EL3 y this Inhanism.
Traps caused by accesses to the fine-grained its associated ISS.	l trap registers .	rer .ted using an ESR_ELx.EC value of 0x18 and
The reset behavior of this field is:		

The reset behavior of this field is:

• On a Warm reset, this field resets to an . hitec⁺ 'OWN value.

Otherwise:

RESO

ATA, bit [26]

When FEAT_MTE2 implemented:

Allocation Tag A Less. Cor JIs access to Allocation Tags, System registers for Memory tagging, and prevention of Tag checking, a 7L2 L1 and F^{*}.0.

ATA	Meaning
0b0	Access to Allocation Tags is prevented at EL2,
	EL1, and EL0.
	Accesses at EL1 and EL2 to GCR_EL1,
	RGSR_EL1, TFSR_EL1, TFSR_EL2 or
	TFSRE0_EL1 that are not UNDEFINED or
	trapped to a lower Exception level are trapped to
	EL3.
	Accesses at EL2 using MRS or MSR with the
	register name TFSR_EL12 that are not
	UNDEFINED are trapped to EL3.
	Memory accesses at EL2, EL1, and EL0 are no subject to a Tag Check operation.
0b1	This control does not prevent access to
	Allocation Tags at EL2, EL1, and EL0.
	This control does not prevent Tag checking at
	EL2, EL1, and EL0.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

res0

EnSCXT, bit [25]

When FEAT_CSV2_2 is implemented or FEAT_CSV2_1p2 is implemented:

Enables access to the SCXTNUM_EL2, SCXTNUM_EL1, and SCXTNUM_EL0 registers.

at EL0, EL and EL2 to M_EL0, S(TNUM_EL1, or SEL2 resters are trapped to EL3 if
M_EL0, S(TNUM_EL1, or EL2 resters are trapped to EL3 if
• EL2 resters are trapped to EL3 if
i sters are trapped to EES if
or op oy a higher priority
, and le values of these registers are
0
ol does not cause any accesses to be
or register values to be treated as 0

The reset behavior of this field is:

• On a Warm reset, this field sets to an phitecu. Ily UNKNOWN value.

Otherwise:

res0

Bits [24:22]

Reserved, RESO.

FIEN, / [21]

Whe FEAT 'is implemented:

Fault lh. tion enable. Trap accesses to the registers ERXPFGCDN_EL1, ERXPFGCTL_EL1, and ERXPFGF_ 1 from EL1 and EL2 to EL3, reported using an ESR_ELx.EC value of 0x18.

FIEN	Meaning
0b0	Accesses to the specified registers from EL1 and EL2 generate a Trap exception to EL3.
0b1	This control does not cause any instructions to be trapped.

If EL3 is not implemented, the Effective value of SCR_EL3.FIEN is 0b1.

If ERRIDR_EL1.NUM is zero, meaning no error records are implemented, or no error record accessible using System registers is owned by a node that implements the RAS Common Fault Injection Model Extension, then this bit might be RES0.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

res0

NMEA, bit [20]

When FEAT_DoubleFault is implemented:

Non-maskable External Aborts. Controls whether PSTATE.A masks SError exceptions at EL3.

	NMEA	Meaning	
	060	SEr exceptions a not taken at EL3 if PSTA. $A == 1$.	
	0b1	SError exc tions 2 taken at EL3 regardless of the v lue of r T. E.A.	
This field is ignored by the PE and treated as zero	^S CR_1	3.EA = 0.	
The reset behavior of this field is:			
• On a Warm reset, this field resets to 0b0.			
Otherwise:			
RESO			
EASE, bit [19]			
When FEAT_Doul' cault is implemente			
External aborts i SError : errupt vector.			
	EASE	Meaning	

	i i i i i i i i i i i i i i i i i i i
060	Synchronous External abort exceptions taken to EL3 are taken to the appropriate synchronous exception vector offset from VBAR_EL3.
0b1	Synchronous External abort exceptions taken to EL3 are taken to the appropriate SError interrupt vector offset from VBAR_EL3.

The reset behavior of this field is:

• On a Warm reset, this field resets to Ob0.

Otherwise:

res0

EEL2, bit [18]

When FEAT_SEL2 is implemented:

Secure EL2 Enable.

EEL2	Meaning
0d0	All behaviors associated with Secure EL2 are disabled. All registers, including timer registers, defined by FEAT_SEL2 are UNDEFINED, and those timers are disabled.
0b1	All behaviors associated with Secure EL2 are enabled.

When the value of this bit is 1, then:

- When SCR_EL3.NS == 0, the SCR_EL3.RW bit is treated as 1 f all purposes her than reading or writing the register.
- If Secure EL1 is using AArch32, then any of the following operations recuted a Secure EL1, is trapped to Secure EL2, using the EC value of ESR_EL2.EC== C = C:
 - A read or write of the SCR.
 - A read or write of the NSACR.
 - A read or write of the MVBAR.
 - A read or write of the SDCR.
 - Execution of an ATS12NSO** instruction.
- If Secure EL1 is using AArch32, using any once following operations, executed in Secure EL1, is trapped to Secure EL2 using the EC value of EL EL2. = 0x0:
 - Execution of an SR⁵ truction that ~s R13_mon.
 - Execution of an LRS anked register) or MSR (Banked register) instruction that would access SPSR_mon, P13_mon, or 1_mon.

If the Effective value of SCR_EL3.EEL2 0, then these operations executed in Secure EL1 using AArch32 are trapped to EL3.

A Secure only implant ation that bes not implement EL3 but implements EL2, behaves as if SCR_EL3.EEL2 == 1.

This' c is permited to be coned in a TLB.

The it the avior of a field is:

• On a ^varm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

res0

API, bit [17]

When FEAT_SEL2 is implemented and FEAT_PAuth is implemented

API, bit [17]

Controls the use of the following instructions related to Pointer Authentication. Traps are reported using an ESR_ELx.EC value of 0x09:

• PACGA, which is always enabled.

- AUTDA, AUTDB, AUTDZA, AUTDZB, AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZA, AUTIZB, PACDA, PACDB, PACDZA, PACDZB, PACIA, PACIA1716, PACIASP, PACIAZ, PACIB, PACIB1716, PACIBSP, PACIBZ, PACIZA, PACIZB, RETAA, RETAB, BRAA, BRAB, BLRAA, BLRAB, BRAAZ, BRABZ, BLRAAZ, BLRABZ, ERETAA, ERETAB, LDRAA and LDRAB when:
 - In EL0, when HCR_EL2.TGE == 0 or HCR_EL2.E2H == 0, and the associated SCTLR_EL1.En<N><M> == 1.
 - In EL0, when HCR_EL2.TGE == 1 and HCR_EL2.E2H == 1, and the associated SCTLR EL2.En<N><M> == 1.
 - In EL1, when the associated SCTLR_EL1.En<N><M> == 1.
 - In EL2, when the associated SCTLR_EL2.En<N><M> == 1.

API	Meaning
0b0	The e of any instrution related to pointer auther ation in any coeption level except FL3
	when the struction are enabled are trapped to
	EL3 unless, v a grapped to EL2 as a result of
	the CR_EL2. PI bit.
0b1	nis contre loes not cause any instructions to
	e trapr

An instruction is trapped only if Pointer Authorication is enaced for that instruction, for more information, see 'PAC generation and verification keys'.

If FEAT_PAuth is implemented *EL3* is n implended, the system behaves as if this bit is 1.

The reset behavior of this fie' 1s.

• On a Warm reset, *t*' is field reset. an architecturally UNKNOWN value.

When FEAT_SEL⁷ s not implemented a. SEAT_PAuth is implemented

API, bit [17]

Control lie us of instition elated to Pointer Authentication:

-ACGA

LTP, AUTEL, AUTDZA, AUTDZB, AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIB, AUTIB1716, AU BSP, AUTIBZ, AUTIZA, AUTIZB, PACDA, PACDB, PACDZA, PACDZB, PACIA, PACIA1716, PACL P, PACIA, PACIB, PACIB1716, PACIBSP, PACIBZ, PACIZA, PACIZ, RETAA, RETAB, BRAA, BRAB, LARAA, BLRAB, BRAAZ, BRABZ, BLRAAZ, BLRABZ, ERETAA, ERETAB, LDRAA and LDRAB when:

- In Non-secure EL0, when HCR_EL2.TGE == 0 or HCR_EL2.E2H == 0, and the associated SCTLR_EL1.En<N><M>== 1.
- In Non-secure EL0, when HCR_EL2.TGE == 1 and HCR_EL2.E2H == 1, and the associated SCTLR_EL2.En<N><M> == 1.
- In Secure EL0, when the associated SCTLR_EL1.En<N><M> == 1.
- In Secure or Non-secure EL1, when the associated SCTLR_EL1.En<N><M> == 1.
- In EL2, when the associated SCTLR_EL2.En<N><M> == 1.

API	Meaning
000	The use of any instruction related to pointer authentication in any Exception level except EL3 when the instructions are enabled are trapped to EL3 unless they are trapped to EL2 as a result of the HCR_EL2.API bit.
0b1	This control does not cause any instructions to be trapped.

If FEAT_PAuth is implemented but EL3 is not implemented, the system behaves as if this bit is 1.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOV value.

Otherwise:

res0

APK, bit [16]

When FEAT_PAuth is implemented:

Trap registers holding "key" values for Poin Auth Traps accesses to the following registers, using an ESR_ELx.EC value of 0x18, from 7L1 or 1, to EL3 unless they are trapped to EL2 as a result of the HCR_EL2.APK bit or other traps:

- APIAKeyLo_EL1, APIAV , Hi_EL1, A 'BKeyL _EL1, APIBKeyHi_EL1.
- APDAKeyLo_EL1, A JAK, Hi_EL1, APL JKeyLo_EL1, APDBKeyHi_EL1.
- APGAKeyLo_E¹ , and APGAK, ¹i_EL1.

АРК	Meaning	
0d0	Access to the registers holding "key" values for pointer authentication from EL1 or EL2 are trapped to EL3 unless they are trapped to EL2 as a result of the HCR_EL2.APK bit or other traps.	
0b1	This control does not cause any instructions to be trapped.	

For more information, see 'PAC generation and verification keys'.

If FEAT_PAuth is implemented but EL3 is not implemented, the system behaves as if this bit is 1.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

res0

TERR, bit [15]

When FEAT_RAS is implemented:

Trap accesses of error record registers. Enables a trap to EL3 on accesses of error record registers.

TERR	Meaning
000	Accesses of the specified error record registers are not trapped by this mechanism.
0b1	Accesses of the specified error record registers at EL2 and EL1 are trapped to EL3, unless the instruction generates a higher priority exception.

In AArch64 state, the instructions affected by this control are:

- MRS and MSR accesses to ERRSELR_EL1, ERXADDR_EL¹ ERXC. R_EL1, ERXMISC0_EL1, ERXMISC1_EL1, and ERXSTATUS_EL1.
- MRS accesses to ERRIDR_EL1 and ERXFR_EL1.
- If FEAT_RASv1p1 is implemented, MRS and MSR accesser JERXN, C2_EL1 d ERXMISC3_EL1.

In AArch32 state, the instructions affected by this control

- MRC and MCR accesses to ERRSELR, ERXADDR, L XA' JR2, ERYCTLR, ERXCTLR2, ERXMISC0, ERXMISC1, ERXMISC2, ERXMISC3, and FrivSTA1
- MRC accesses to ERRIDR, ERXFR, and F XFR2.
- If FEAT_RASv1p1 is implemented, MR and Mr ... and ERXMISC4, ERXMISC5, ERXMISC6, and ERXMISC7.

Unless the instruction generates a ' gher pi ity exc tion, trapped instructions generate an exception to EL3.

Trapped AArch64 instruction reported usin, FC syndrome value 0x18.

Trapped AArch32 instructions are re, rted using EC syndrome value 0x03.

Accessing this field ⁺ , the following bearior:

- This field *i* ermitted *s* be RESO if all of the following are true:
 - ERRSE. F' and all FRX* registers are implemented as UNDEFINED or RAZ/WI.
 - FTJDR_ (.NUM; .ero.

The react behave r of this is:

• n a W 'is field resets to an architecturally UNKNOWN value.

Otherwis

res0

TLOR, bit [14]

When FEAT_LOR is implemented:

Trap LOR registers. Traps accesses to the LORSA_EL1, LOREA_EL1, LORN_EL1, LORC_EL1, and LORID_EL1 registers from EL1 and EL2 to EL3, unless the access has been trapped to EL2.

TLOR	Meaning
060	This control does not cause any instructions to be trapped.

TLOR	Meaning
0b1	EL1 and EL2 accesses to the LOR registers that are not UNDEFINED are trapped to EL3, unless it is trapped HCR_EL2.TLOR.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

res0

TWE, bit [13]

Traps EL2, EL1, and EL0 execution of WFE instructions to E⁷, from ny Securit state and both Execution states, reported using an ESR_ELx.EC value of 0x01.

When FEAT_WFxT is implemented, this trap also applie of the W^r \perp f instruct \perp n.

. . .	Teani-
0br	This control does not cause any instructions to be trapped.
1	Any attempt to execute a WFE instruction at any Exception level lower than EL3 is trapped to EL3, if the instruction would otherwise have caused the PE to enter a low-power state and it is not trapped by SCTLR.nTWE, HCR.TWE, SCTLR_EL1.nTWE, SCTLR_EL2.nTWE, or HCR_EL2.TWE.

In AArch the a mpted lecution of a conditional WFE instruction is only trapped if the instruction passes its cortain conditional wFE instruction conditional wFE instruction is only trapped if the instruction passes its cortain conditional wFE instruction is only trapped if the instruction passes its cortain conditional wFE instruction is only trapped if the instruction passes its cortain conditional wFE instruction is only trapped if the instruction passes its cortain conditional wFE instruction is only trapped if the instruction passes its cortain conditional wFE instruction is only trapped if the instruction passes its cortain conditional wFE instruction is only trapped if the instruction passes its cortain conditional wFE instruction is only trapped if the instruction passes its cortain conditional wFE instruction is only trapped if the instruction passes its cortain conditional wFE instruction is only trapped if the instruction passes its cortain conditional wFE instruction is only trapped if the instruction passes its cortain conditional wFE instruction is only trapped if the instruction passes its cortain conditional wFE instruction is only trapped if the instruction passes its cortain conditional wFE instruction is only trapped if the instruction passes its cortain conditional wFE instruction conditional wFE instructin conditional wFE instruction conditional wFE instructio

Sinc WFF complete at any time, even without a Wakeup event, the traps on WFE of WFI are not guarant to be taken, even if the WFE or WFI is executed when there is no Wakeup event. The only guarantee is that if the truction does not complete in finite time in the absence of a Wakeup event, the trap will be taken.

For more into mation about when WFE instructions can cause the PE to enter a low-power state, see 'Wait for Event mechanism and Send event'.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

TWI, bit [12]

Traps EL2, EL1, and EL0 execution of WFI instructions to EL3, from any Security state and both Execution states, reported using an ESR_ELx.EC value of 0x01.

When FEAT_WFxT is implemented, this trap also applies to the WFIT instruction.

TWI	Meaning
0b0	This control does not cause any instructions to be trapped.
0b1	Any attempt to execute a WFI instruction at any Exception level lower than EL3 is trapped to EL3, if the instruction would otherwise have caused the PE to enter a low-power state and it is not trapped by SCTLR.nTWI, HCR.TWI, SCTLR_EL1.nTWI, SCTLR_EL2.nTWI, or HCR_EL2.TWI.

In AArch32 state, the attempted execution of a conditional WFI instruct⁷ it is only the ped if the instruction passes its condition code check.

Since a WFE or WFI can complete at any time, even without *c*, akeup *c*, nt, the tr*c* s on WFE of WFI are not guaranteed to be taken, even if the WFE or WFI is executed *c*, en there is no the event. The only guarantee is that if the instruction does not complete in finite time in *t* absence *c* a Wake event, the trap will be taken.

For more information about when WFI instructions ca. Ase the F to enter a low-power state, see 'Wait for Interrupt'.

The reset behavior of this field is:

• On a Warm reset, this field resets to an at ite arally UNIT OWN value.

ST, bit [11]

Traps Secure EL1 accesses to the punter-timer registers to EL3, from AArch64 state only, reported using an ESR_ELx.EC value of 0x18.

ST	Meaning
040	Secure EL1 using AArch64 accesses to the CNTPS_TVAL_EL1, CNTPS_CTL_EL1, and CNTPS_CVAL_EL1 are trapped to EL3 when Secure EL2 is disabled. If Secure EL2 is enabled, the behavior is as if the value of this field was 0b1.
0b1	This control does not cause any instructions to be trapped.

Accesses to the Counter-timer Physical Secure timer registers are always enabled at EL3. These registers are not accessible at EL0.

When FEAT_RME is implemented and Secure state is not implemented, this bit is RESO.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

RW, bit [10]

When EL1 is capable of using AArch32 or EL2 is capable of using AArch32:
Chapter E3. System registers affected by SME E3.2. Changes to existing System registers

Execution state control for lower Exception levels.

RW	Meaning
0b0	Lower levels are all AArch32.
061	 The next lower level is AArch64. If EL2 is present: EL2 is AArch64. EL2 controls EL1 and EL0 behaviors. If EL2 is not present: EL1 is AArch64. EL0 is determined by the Execution state described in the current process state when executing at EL0.

If AArch32 state is supported by the implementation at EL1 CR_1 3.NS == 1 nd AArch32 state is not supported by the implementation at EL2, the Effective value this bit is 1.

-

If AArch32 state is supported by the implementation at F , FEAT , EL2 is im, emented and SCR_EL3.{EEL2, NS} == $\{1, 0\}$, the Effective value of this bit is 1.

This bit is permitted to be cached in a TLB.

The reset behavior of this field is:

• On a Warm reset, this field resets to an ai, it urally UNKNOWN value.

Otherwise:

RAO/WI

SIF, bit [9]

Secure instruction / .ch. When the PE is a Secure state, this bit disables instruction execution from memory marked in the fir stage of unslation as being Non-secure.

SIF	Meaning
060	Secure state instruction execution from memory marked in the first stage of translation as being Non-secure is permitted.
0b1	Secure state instruction execution from memory marked in the first stage of translation as being Non-secure is not permitted.

When FEAT_RME is implemented and Secure state is not implemented, this bit is RESO.

When FEAT PAN3 is implemented, it is IMPLEMENTATION DEFINED whether SCR EL3.SIF is also used to determine instruction access permission for the purpose of PAN.

This bit is permitted to be cached in a TLB.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

HCE, bit [8]

Hypervisor Call instruction enable. Enables HVC instructions at EL3 and, if EL2 is enabled in the current Security state, at EL2 and EL1, in both Execution states, reported using an ESR_ELx.EC value of 0x00.

HCE	Meaning
0d0	HVC instructions are UNDEFINED.
0b1	HVC instructions are enabled at EL3, EL2, and EL1.

HVC instructions are always UNDEFINED at EL0 and, if Secure EL2 is cause of Secure EL1. Any resulting exception is taken from the current Exception level to the current Exception level.

If EL2 is not implemented, this bit is RESO.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architectura' UNKNO A value.

SMD, bit [7]

Secure Monitor Call disable. Disables SMC instruction at EL1 as a above, from any Security state and both Execution states, reported using an ESR_EL. C val

5. T	Meaning
°Ъ0	SMC instructions are enabled at EL3, EL2 and EL1.
0b1	SMC instructions are UNDEFINED.

SMC instructions are ys UNDEF ED at ELO. Any resulting exception is taken from the current Exception level to the current ception by el.

If H' $x_{EL2.T}$ C or HCR. SC traps attempted EL1 execution of SMC instructions to EL2, that trap has priority over s disc.

The reset havior of this field is:

• On a W. reset, this field resets to an architecturally UNKNOWN value.

Bit [6]

Reserved, RESO.

Bits [5:4]

Reserved, RES1.

EA, bit [3]

External Abort and SError interrupt routing.

EA	Meaning
0b0	 When executing at Exception levels below EL3, External aborts and SError interrupts are not taken to EL3. In addition, when executing at EL3: SError interrupts are not taken. External aborts are taken to EL3.
0b1	When executing at any Exception level, External aborts and SError interrupts are taken to EL3.

For more information, see 'Asynchronous exception routing'.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally U NOWN van

FIQ, bit [2]

Physical FIQ Routing.

F	Meaning
0	When executing at Exception levels below EL3, physical FIQ interrupts are not taken to EL3. When executing at EL3, physical FIQ interrupts are not taken.
0b1	When executing at any Exception level, physical FIQ interrupts are taken to EL3.

For more informatio. ... e 'Async' onous exception routing'.

The re behav r of this of as:

In a W this field resets to an architecturally UNKNOWN value.

IRQ, bit

Physical IRQ kouting.

IRQ	Meaning
000	When executing at Exception levels below EL3, physical IRQ interrupts are not taken to EL3. When executing at EL3, physical IRQ interrupts are not taken.
0b1	When executing at any Exception level, physical IRQ interrupts are taken to EL3.

For more information, see 'Asynchronous exception routing'.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

NS, bit [0]

When FEAT_RME is implemented

NS, bit [0]

Non-secure bit. This field is used in combination with SCR_EL3.NSE to select the Security state of EL2 and lower Exception levels.

NSE	NS	Meaning	
0b0	0b0	Secure.	
0b0	0b1	Non-secure.	
0b1	0b0	Reserved.	
0b1	0b1	Realm.	

When Secure state is not implemented, SCR_7_3.NS is ES1 and ______fective value is 1.

The reset behavior of this field is:

• On a Warm reset, this field resets to rarch. turally UNKNOWN value.

Otherwise

NS, bit [0]

Non-secure bit.

NS	Meaning		
000	Indicates that EL0 and EL1 are in Secure state. When FEAT_SEL2 is implemented and SCR_EL3.EEL2 == 1, then EL2 is using AArch64 and in Secure state.		
0b1	Indicates that Exception levels lower than EL3 are in Non-secure state, so memory accesses from those Exception levels cannot access Secure memory.		

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing SCR_EL3

Accesses to this register use the following encodings in the System register encoding space:

MRS <Xt>, SCR_EL3

Chapter E3. System registers affected by SME E3.2. Changes to existing System registers

		op0	op1	CRn	CRm	ор2
		0b11	0b110	0b0001	0b0001	0b000
1 2 2	<pre>if PSTATE.EL == UNDEFINED;</pre>	= ELO then				
3 4 5	elsif PSTATE.EI UNDEFINED;	L == ELI then				
6 7	UNDEFINED; elsif PSTATE.EI	L == EL3 then				
8	X[t, 64] =	SCR_EL3;				
	MSR SCR_EL3,	<xt></xt>				
		ор0	op1	CRn	CRm	op2
		0b11	0b110	° .001	0. 1	0b000
1 2	<pre>if PSTATE.EL == UNDEFINED;</pre>	= ELO then				
3 4	elsif PSTATE.EI UNDEFINED;	L == EL1 then				
5 6 7	elsif PSTATE.EI UNDEFINED;	L == EL2 then	\mathbf{X}			
8	SCR_EL3 = X	L == EL3 then X[t, 64];				

E3.2.16 SCTLR_EL1, System Control Register (EL1)

The SCTLR_EL1 characteristics are:

Purpose

Provides top level control of the system, including its memory system, at EL1 and EL0.

Configuration

AArch64 system register SCTLR_EL1 bits [31:0] are architecturally mapped to AArch32 system register SCTLR[31:0].

Attributes

SCTLR_EL1 is a 64-bit register.

Field descriptions

The SCTLR_EL1 bit assignments are:



TIDCP, bit [63]

When FEAT_TIDCP1 is implemented:

Trap IMPLEMENTATION DEFINED functionality. When HCR_EL2. $\{E2H, TGE\} != \{1, 1\}$, traps EL0 accesses to the encodings reserved for IMPLEMENTATION DEFINED functionality to EL1.

TIDCP	Meaning
000	No instructions accessing the System register or System instruction spaces are trapped by this mechanism.

Chapter E3. System registers affected by SME E3.2. Changes to existing System registers

TIDCP	Meaning
DD1	Meaning Instructions accessing the following System register or System instruction spaces are trapped to EL1 by this mechanism: In AArch64 state, EL0 access to the encodings in the following reserved encoding spaces are trapped and reported using EC syndrome 0x18: IMPLEMENTATION DEFINED System instructions, which are accessed using SYS and SYSL, with CRn == 115}. IMPLE
2	 All coproc==p15, CRn==c9, opc1 == {0-7}, CRm == {c0-c2, c5-c8}, opc2 == {0-7}. All coproc==p15, CRn==c10, opc1 =={0-7}, CRm == {c0, c1, c4, c8}, opc2 == {0.7}.
	opc2 == $\{0-7\}$. - All coproc==p15, CRn==c11, opc1== $\{0-7\}$, CRm == $\{c0-c8, c15\}$ opc2 == $\{0-7\}$.

The rese _ena pr of the field .

Jn a War reset, this leld resets to an architecturally UNKNOWN value.

Other,

•

res0

SPINTMASK, bit [62]

When FEAT_NMI is implemented:

SP Interrupt Mask enable. When SCTLR_EL1.NMI is 1, controls whether PSTATE.SP acts as an interrupt mask, and controls the value of PSTATE.ALLINT on taking an exception to EL1.

SPINTMASK	Meaning
060	Does not cause PSTATE.SP to mask interrupts. PSTATE.ALLINT is set to 1 on taking an exception to EL1.

SPINTMASK	Meaning
0b1	When PSTATE.SP is 1 and execution is at EL1, an IRQ or FIQ interrupt that is targeted to EL1 is masked regardless of any denotion of Superpriority. PSTATE.ALLINT is set to 0 on taking an exception to EL1.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

res0

NMI, bit [61]

When FEAT_NMI is implemented:

Non-maskable Interrupt enable.

NM	Meaning
¹ 70	This control does not affect interrupt masking behavior.
0b1	 This control enables all of the following: The use of the PSTATE.ALLINT interrupt mask. IRQ and FIQ interrupts to have Superpriority as an additional attribute. PSTATE.SP to be used as an interrupt mask.

The rut b avior on und field is:

• On a 'arm reset:

W1. EL2 is not implemented and EL3 is not implemented, this field resets to 0b0.
Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

res0

EnTP2, bit [60]

When FEAT_SME is implemented:

Traps instructions executed at EL0 that access TPIDR2_EL0 to EL1, or to EL2 when EL2 is implemented and enabled for the current Security state and HCR_EL2.TGE is 1. The exception is reported using ESR_ELx.EC value 0x18.

EnTP2	Meaning
0b0	This control causes execution of these instructions at EL0 to be trapped.
0b1	This control does not cause execution of any instructions to be trapped.

If FEAT_VHE is implemented, EL2 is implemented and enabled in the current Security state, and HCR_EL2. {E2H, TGE} == $\{1, 1\}$, this field has no effect on execution at EL0.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN vert

Otherwise:

res0

Bits [59:58]

Reserved, RESO.

EPAN, bit [57]

When FEAT_PAN3 is implemented:

Enhanced Privileged Access Never. Wr. PSTA PAN is 1, determines whether an EL1 data access to a page with stage 1 EL0 instruction acces permises on gene tes a Permission fault as a result of the Privileged Access Never mechanism.

EPAN	Meaning
0b0	No additional Permission faults are generated by this mechanism.
0b1	An EL1 data access to a page with stage 1 EL0 data access permission or stage 1 EL0 instruction access permission generates a Permission fault. Any speculative data accesses that would generate a Permission fault as a result of PSTATE.PAN = 1 if the accesses were not speculative, will not cause an allocation into a cache.

This bit is permitted to be cached in a TLB.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

res0

EnALS, bit [56]

When FEAT_LS64 is implemented:

When HCR_EL2.{E2H, TGE} != {1, 1}, traps execution of an LD64B or ST64B instruction at EL0 to EL1.

EnALS	Meaning
0b0	Execution of an LD64B or ST64B instruction at EL0 is trapped to EL1.
0b1	This control does not cause any instructions to be trapped.

A trap of an LD64B or ST64B instruction is reported using an ESR_ELx.EC value of 0x0A, with an ISS code of 0x0000002.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOV, value.

Otherwise:

res0

EnAS0, bit [55]

When FEAT_LS64_ACCDATA is implement/

When HCR_EL2.{E2H, TGE} $!= \{1, 1\}$, tra_k execution at EL0 to EL1.

L ^S0	Meaning
טמר סטר	Execution of an ST64BV0 instruction at EL0 is trapped to EL1.
0b1	This control does not cause any instructions to be trapped.

A trap of $rac{4}BV0$ tructions reported using an ESR_ELx.EC value of 0x0A, with an ISS code of 0x0000001. The racet behaver of this race is:

• a Y III ICS. Is field resets to an architecturally UNKNOWN value.

Otherwise

res0

EnASR, bit [54]

When FEAT_LS64_V is implemented:

When HCR_EL2.{E2H, TGE} != {1, 1}, traps execution of an ST64BV instruction at EL0 to EL1.

EnASR	Meaning
000	Execution of an ST64BV instruction at EL0 is trapped to EL1.
0b1	This control does not cause any instructions to be trapped.

A trap of an ST64BV instruction is reported using an ESR_ELx.EC value of 0x0A, with an ISS code of 0x0000000.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

res0

TME, bit [53]

When FEAT_TME is implemented:

Enables the Transactional Memory Extension at EL1.

	TME	Mear .g
	060	ny at v to execve a TSTART instruction at EL1 is traped to F 1, unless HCR_EL2.TME or SCR_EL3. Causes TSTART instructions to c UNDEFINED at EL1.
		this cont a does not cause any TSTART true on to be trapped.
The reset behavior of this field is:		
• On a Warm reset, this field reas to a reh	nitec "ally	UNKNOWN value.

Otherwise:

res0

TME0, bit [52]

When FEAT_TM. 's ir .emented.

Enables ⁺ _ `sactic `Mem y Extension at EL0.

TME0	Meaning
0Ъ0	Any attempt to execute a TSTART instruction at EL0 is trapped to EL1, unless HCR_EL2.TME or SCR_EL3.TME causes TSTART instructions to be UNDEFINED at EL0.
0b1	This control does not cause any TSTART instruction to be trapped.

If FEAT_VHE is implemented, EL2 is implemented and enabled in the current Security state, and HCR_EL2.{E2H, TGE} == $\{1, 1\}$, this field has no effect on execution at EL0.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

res0

TMT, bit [51]

When FEAT_TME is implemented:

Forces a trivial implementation of the Transactional Memory Extension at EL1.

	ТМТ	Meaning
	060	This control does not cause any TSTART instruction to fail.
	0b1	When the TSTART instruction is executed at EL1, the transaction fails with a TRIVIAL failure cause
The reset behavior of this field is:		
• On a Warm reset, this field resets to an ar	chitecturally t	^V NOWN va.
Otherwise:		
RES0		
TMT0, bit [50]		\mathbf{V}
When FEAT_TME is implemented:		
Forces a trivial implementation of the Transct	tion. Memory	Extension at EL0.
·		
	тмто	Meaning
	060	This control does not cause any TSTART instruction to fail.
	0b1	When the TSTART instruction is executed at

VV rs mg. Inted, EL2 is implemented and enabled in the current Security state, and HCR_EL2.{E2H, If FL. TGE = (, 1}, this field has no effect on execution at EL0.

EL0, the transaction fails with a TRIVIAL

failure cause.

The reset ben. ior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

res0

TWEDEL, bits [49:46]

When FEAT_TWED is implemented:

TWE Delay. A 4-bit unsigned number that, when SCTLR_EL1.TWEDEn is 1, encodes the minimum delay in taking a trap of WFE* caused by SCTLR_EL1.nTWE as $2^{(TWEDEL + 8)}$ cycles.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

res0

TWEDEn, bit [45]

When FEAT_TWED is implemented:

TWE Delay Enable. Enables a configurable delayed trap of the WFE* instruction caused by SCTLR_EL1.nTWE.

Meaning
The delay for taking the trap is IMPLEMENT ON DEFINED.
The de .y for takin. the trap is at least the nun. r of cycles dei ed in CTLi TL1.TWED

The reset behavior of this field is:

• On a Warm reset, this field resets to an archite ally U. NOWN ver.e.

Otherwise:

res0

DSSBS, bit [44]

When FEAT_SSBS is impler od:

Default PSTATE.SSBS value on Ex. vtion Entry.

DSSBS	Meaning
0b0	PSTATE.SSBS is set to 0 on an exception to EL1.
0b1	PSTATE.SSBS is set to 1 on an exception to EL1.

The reset be. ior of this field is:

• On a Warm reset, this field resets to an IMPLEMENTATION DEFINED value.

Otherwise:

res0

ATA, bit [43]

When FEAT_MTE2 is implemented:

Allocation Tag Access in EL1.

When SCR_EL3.ATA == 1 and HCR_EL2.ATA == 1, controls access to Allocation Tags and Tag Check operations in EL1.

ATA	Meaning
000	Access to Allocation Tags is prevented at EL1. Memory accesses at EL1 are not subject to a Tag Check operation.
0b1	This control does not prevent access to Allocation Tags at EL1. Tag Checked memory accesses at EL1 are subject to a Tag Check operation.

The reset behavior of this field is:

Otherwise:

res0

ATA0, bit [42]

When FEAT_MTE2 is implemented:

Allocation Tag Access in EL0.

When SCR_EL3.ATA == 1, HCR_EL2.A1 $_{\sim}$ = 1, Allocation Tags and Tag Check operations in E₁

TI 2.{E2H, TGE} $!= \{1, 1\}$, controls access to

ATA.	J Meaning
060	Access to Allocation Tags is prevented at EL0. Memory accesses at EL0 are not subject to a Tag Check operation.
0b1	This control does not prevent access to Allocation Tags at EL0. Tag Checked memory accesses at EL0 are subject to a Tag Check operation.

Softwar, ay change this control bit on a context switch.

The reset be... ior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

res0

TCF, bits [41:40]

When FEAT_MTE2 is implemented:

Tag Check Fault in EL1. Controls the effect of Tag Check Faults due to Loads and Stores in EL1.

If FEAT_MTE3 is not implemented, the value 0b11 is reserved.

Chapter E3. System registers affected by SME E3.2. Changes to existing System registers

TCF	Meaning	Applies
0b00	Tag Check Faults have no effect on the PE.	
0b01	Tag Check Faults cause a synchronous exception.	
0b10	Tag Check Faults are asynchronously accumulated.	
0b11	Tag Check Faults cause a synchronous exception on reads, and are asynchronously accumulated on writes.	When FEAT_MTE3 is implemented

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOW, value.

Otherwise:

res0

TCF0, bits [39:38]

When FEAT_MTE2 is implemented:

Tag Check Fault in EL0. When HCR_EL2. { L_{T} , T_{T} , L_{T} , controls the effect of Tag Check Faults due to Loads and Stores in EL0.

If FEAT_MTE3 is not implement , the val. 9b11 h. "served.

Software may change this co it on a contex witch.

	TCF0	aning	Applies
	.00	Tag Check Faults have no effect on the PE.	
	0b01	Tag Check Faults cause a synchronous exception.	
	0.10	Tag Check Faults are asynchronously accumulated.	
	0b11	Tag Check Faults cause a synchronous exception on reads, and are asynchronously accumulated on writes.	When FEAT_MTE3 is implemented

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

res0

ITFSB, bit [37]

When FEAT_MTE2 is implemented:

When synchronous exceptions are not being generated by Tag Check Faults, this field controls whether on exception entry into EL1, all Tag Check Faults due to instructions executed before exception entry, that are reported

asynchronously, are synchronized into TFSRE0_EL1 and TFSR_EL1 registers.

ITFSB	Meaning
0b0	Tag Check Faults are not synchronized on entry to EL1.
0b1	Tag Check Faults are synchronized on entry to EL1.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value

Otherwise:

res0

BT1, bit [36]

When FEAT_BTI is implemented:

PAC Branch Type compatibility at EL1.

B"	Meaning
0	When the PE is executing at EL1, PACIASP and PACIBSP are compatible with PSTATE.BTYPE == 0b11.
0b1	When the PE is executing at EL1, PACIASP and PACIBSP are not compatible with PSTATE.BTYPE == 0b11.

The reset behavior o. s field is

• C a Warn reset, th. ⁶ J resets to an architecturally UNKNOWN value.

Oth. "ise:

res0

BT0, bit [35]

When FEAT_BTI is implemented:

PAC Branch Type compatibility at EL0.

BT0	Meaning
0b0	When the PE is executing at EL0, PACIASP and PACIBSP are compatible with PSTATE.BTYPE == 0b11.
0b1	When the PE is executing at EL0, PACIASP and PACIBSP are not compatible with PSTATE.BTYPE == 0b11.

When the value of HCR_EL2.{E2H, TGE} is {1, 1}, the value of SCTLR_EL1.BT0 has no effect on execution at EL0.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

res0

Bit [34]

Reserved, RESO.

MSCEn, bit [33]

When FEAT_MOPS is implemented and (HCR_EL2.E2H == 0 or $CR_EL2.TGL = 0$):

Memory Copy and Memory Set instructions Enable. Enables encutio. f the Memory Copy and Memory Set instructions at EL0.

MSCŁ	1 caning
0b0 C'	ecut a of the Memory Copy and Memory Set instructions is UNDEFINED at EL0. Ais control does not cause any instructions to be UNDEFINED.

When FEAT_MOPS is implemented and HCR_ λ 2.{E2H, TGE} is {1, 1}, the Effective value of this bit is 0b1. The reset behavior of this field is:

• On a Warm register to architecturally UNKNOWN value.

Otherwise:

res0

CMC , bit [32

When . " _CMOW is implemented:

Controls ca. • maintenance instruction permission for the following instructions executed at EL0.

• IC IVAU, DC CIVAC, DC CIGDVAC and DC CIGVAC.

CMOW	Meaning
060	These instructions executed at EL0 with stage 1 read permission, but without stage 1 write permission, do not generate a stage 1 permission fault.
0b1	If enabled as a result of SCTLR_EL1.UCI==1, these instructions executed at EL0 with stage 1 read permission, but without stage 1 write permission, generate a stage 1 permission fault.

When AArch64.HCR_EL2.{E2H, TGE} is {1, 1}, this bit has no effect on execution at EL0.

For this control, stage 1 has write permission if all of the following apply:

- AP[2] is 0 or DBM is 1 in the stage 1 descriptor.
- Where APTable is in use, APTable[1] is 0 for all levels of the translation table.

This bit is permitted to be cached in a TLB.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

res0

EnIA, bit [31]

When FEAT_PAuth is implemented:

Controls enabling of pointer authentication (using the APIAK __EL1 key) instrum on addresses in the EL1&0 translation regime.

For more information, see System register control of point, ut atication

EnL	Mng
U	vointer authentication (using the APIAKey_EL1 key) of instruction addresses is not enabled.
0b.	Pointer authentication (using the APIAKey_EL1 key) of instruction addresses is enabled.

This field controls the ehavior of the Au PACIA and AuthIA pseudocode functions. Specifically, when the field is 1, AddPACIA r arns a corp of a pointer to which a pointer authentication code has been added, and AuthIA returns an authen patted corp of a pointer. When the field is 0, both of these functions are NOP.

The reset b vior on s field is

• C. a Wari reset, th. C. d resets to an architecturally UNKNOWN value.

Othe 'ise:

res0

EnIB, bit [30]

When FEAT_PAuth is implemented:

Controls enabling of pointer authentication (using the APIBKey_EL1 key) of instruction addresses in the EL1&0 translation regime.

For more information, see System register control of pointer authentication .

EnIB	Meaning
000	Pointer authentication (using the APIBKey_EL1 key) of instruction addresses is not enabled.
0b1	Pointer authentication (using the APIBKey_EL1 key) of instruction addresses is enabled.

This field controls the behavior of the AddPACIB and AuthIB pseudocode functions. Specifically, when the field is 1, AddPACIB returns a copy of a pointer to which a pointer authentication code has been added, and AuthIB returns an authenticated copy of a pointer. When the field is 0, both of these functions are NOP.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

res0

LSMAOE, bit [29]

When FEAT_LSMAOC is implemented:

Load Multiple and Store Multiple Atomicity and Ordering Enable.

	Die.
LSMAOE	Ieanii.
060	For memo. cesses at EL0, A32 and T32
	a store multiple and store multiple can have an aterrupt to en during the sequence memory
	cesse and the memory accesses are not
01	The ordering and interrupt behavior of A32 and
	T32 Load Multiple and Store Multiple at EL0 is as defined for Armv8.0.

This bit is permitted to be c ned h. TLB.

When FEAT_VHE is *i* plemented, an he value of HCR_EL2.{E2H, TGE} is {1,1}, this bit has no effect on execution at EL0.

The reset behavior of this and is:

• On a ^{we} m resc his field sets to an architecturally UNKNOWN value.

Other se:

RES.

nTLSMD, '+ [28]

When FEAT_LSMAOC is implemented:

No Trap Load Multiple and Store Multiple to Device-nGRE/Device-nGnRE/Device-nGnRnE memory.

nTLSMD	Meaning
060	All memory accesses by A32 and T32 Load Multiple and Store Multiple at EL0 that are marked at stage 1 as Device-nGRE/Device-nGnRE/Device-nGnRnE memory are trapped and generate a stage 1 Alignment fault.

nTLSMD	Meaning
0b1	All memory accesses by A32 and T32 Load Multiple and Store Multiple at EL0 that are marked at stage 1 as
	memory are not trapped.

This bit is permitted to be cached in a TLB.

When FEAT_VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1,1}, this bit has no effect on execution at EL0.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOW value.

Otherwise:

res1

EnDA, bit [27]

When FEAT_PAuth is implemented:

Controls enabling of pointer authentication (u. 9 the CL1 key) of instruction addresses in the EL1&0 translation regime.

For more information, see System gister c trol on inter authentication .

۲. nDA	Meaning
000	Pointer authentication (using the APDAKey_EL1 key) of data addresses is not enabled.
0b1	Pointer authentication (using the APDAKey_EL1 key) of data addresses is enabled.

This 1d cor bayior of the AddPACDA and AuthDA pseudocode functions. Specifically, when the field is 1, Ac CDA returns a copy of a pointer to which a pointer authentication code has been added, and AuthDA returns an thenticated copy of a pointer. When the field is 0, both of these functions are NOP.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

res0

UCI, bit [26]

Traps EL0 execution of cache maintenance instructions, to EL1, or to EL2 when it is implemented and enabled for the current Security state and HCR_EL2.TGE is 1, from AArch64 state only, reported using an ESR_ELx.EC value of 0x18.

This applies to DC CVAU, DC CIVAC, DC CVAC, DC CVAP, and IC IVAU.

If FEAT_DPB2 is implemented, this trap also applies to DC CVADP.

If FEAT_MTE is implemented, this trap also applies to DC CIGVAC, DC CIGDVAC, DC CGVAC, DC CGDVAC, DC CGVAP, and DC CGDVAP.

If FEAT_DPB2 and FEAT_MTE are implemented, this trap also applies to DC CGVADP and DC CGDVADP.

UCI	Meaning
0b0	Execution of the specified instructions at EL0 using AArch64 is trapped.
0b1	This control does not cause any instructions to be trapped.

When FEAT_VHE is implemented, and the value of HCR_EL2.{E2H. $\Im E$ } is 1, this bit has no effect on execution at EL0.

If the Point of Coherency is before any level of data cache, it is IM_LEME. `ATION DEF_ED whether the execution of any data or unified cache clean, or clean and invalidate instrained that op_tes by A to the point of coherency can be trapped when the value of this control is 1.

If the Point of Unification is before any level of data cache, N IM LEMENTATION DEFINED whether the execution of any data or unified cache clean by VA to the Point f Unity for instruction can be trapped when the value of this control is 1.

If the Point of Unification is before any level of instruction cache, it is IMPLEMENTATION DEFINED whether the execution of any instruction cache invalidate by ' Δ^{+} and Point of Unification instruction can be trapped when the value of this control is 1.

The reset behavior of this field is

• On a Warm reset, this f. d. rts to an arch. rturally UNKNOWN value.

EE, bit [25]

Endianness of d accesse LEL1, and stage 1 translation table walks in the EL1&0 translation regime.

EE	Meaning
0d0	Explicit data accesses at EL1, and stage 1 translation table walks in the EL1&0 translation regime are little-endian.
0b1	Explicit data accesses at EL1, and stage 1 translation table walks in the EL1&0 translatio regime are big-endian.

If an implementation does not provide Big-endian support at Exception levels higher than EL0, this bit is RESO.

If an implementation does not provide Little-endian support at Exception levels higher than EL0, this bit is RES1. The EE bit is permitted to be cached in a TLB.

When FEAT_VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is $\{1, 1\}$, this bit has no effect on the PE.

The reset behavior of this field is:

• On a Warm reset, this field resets to an IMPLEMENTATION DEFINED value.

E0E, bit [24]

Endianness of data accesses at EL0.

E0E	Meaning
0b0	Explicit data accesses at EL0 are little-endian.
0b1	Explicit data accesses at EL0 are big-endian.

If an implementation only supports Little-endian accesses at EL0, then this bit is RES0. This option is not permitted when SCTLR_EL1.EE is RES1.

If an implementation only supports Big-endian accesses at ELO, then this at is RES. This option is not permitted when SCTLR_EL1.EE is RESO.

This bit has no effect on the endianness of LDTR, LDTRH, LDTRSH, TRSW, TR, and ST⁻ H instructions executed at EL1.

When FEAT_VHE is implemented, and the value of HC $EL2.{F 1, TGE}, 1, 1$, this bit has no effect on execution at EL0.

The reset behavior of this field is:

• On a Warm reset, this field resets to an chitectur ay UNKNO A value.

SPAN, bit [23]

When FEAT_PAN is implement

Set Privileged Access Never nu. ing an except. to EL1.

SPAN	Meaning
060	PSTATE.PAN is set to 1 on taking an exception to EL1.
0b1	The value of PSTATE.PAN is left unchanged on taking an exception to EL1.

The reset . `avior of this field is:

• On a Way a reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

res1

EIS, bit [22]

When FEAT_ExS is implemented:

Exception Entry is Context Synchronizing.

EIS	Meaning
060	The taking of an exception to EL1 is not a context synchronizing event.

Chapter E3. System registers affected by SME E3.2. Changes to existing System registers

EIS	Meaning
0b1	The taking of an exception to EL1 is a context synchronizing event.

When FEAT_VHE is implemented, and the value of HCR_EL2. $\{E2H, TGE\}$ is $\{1,1\}$, this bit has no effect on execution at EL0.

If SCTLR_EL1.EIS is set to 0b0:

- Indirect writes to ESR_EL1, FAR_EL1, SPSR_EL1, ELR_EL1 are synchronized on exception entry to EL1, so that a direct read of the register after exception entry sees the indirectly written value caused by the exception entry.
- Memory transactions, including instruction fetches, from an Exption le 'always use the translation resources associated with that translation regime.
- Exception Catch debug events are synchronous debug events.
- DCPS* and DRPS instructions are context synchronizative events.

The following are not affected by the value of SCTLR_E/ LIS:

- Changes to the PSTATE information on entry to EL1.
- Behavior of accessing the banked copies of the sk point susing the sP register name for loads, stores and data processing instructions.
- Exit from Debug state.

The reset behavior of this field is:

• On a Warm reset, this field registo, archite vally UNKNOWN value.

Otherwise:

RES1

IESB, bit [21]

When FEAT_IL. ' is im _mented:

Implicit Fr Synch vzation e int enable. Possible values are:

IESB	Meaning
0b0	Disabled.
0b1	 An implicit error synchronization event is added: At each exception taken to EL1. Before the operational pseudocode of each ERET instruction executed at EL1.

When the PE is in Debug state, the effect of this field is CONSTRAINED UNPREDICTABLE, and its Effective value might be 0 or 1 regardless of the value of the field. If the Effective value of the field is 1, then an implicit error synchronization event is added after each DCPSX instruction taken to EL1 and before each DRPS instruction executed at EL1, in addition to the other cases where it is added.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

res0

TSCXT, bit [20]

When FEAT_CSV2_2 is implemented or FEAT_CSV2_1p2 is implemented:

Trap EL0 Access to the SCXTNUM_EL0 register, when EL0 is using AArch64.

TSCXT	Meaning
0b0	EL0 access to SCXTNUM_EL0 is not disabled by this mechanism.
0b1	EL0 access to SCXTNUM_EL0 is disabled, causing rexcep to EL1, or to EL2 when it is immemented and abled for the current Secure state and HC _EL2.TGE is 1. ne value f SCXTN M_EL0 is treated as 0.

When FEAT_VHE is implemented, and the value of HC $\$ 7L2 22H, TGF} is {1,1}, this bit has no effect on execution at EL0.

The reset behavior of this field is:

• On a Warm reset, this field resets to an hitect NOWN value.

Otherwise:

res1

WXN, bit [19]

Write permission impress XN (Execute-, ver). For the EL1&0 translation regime, this bit can force all memory regions that are w_{rad} to be treated as XIN.

WXN	Meaning
0b0	This control has no effect on memory access permissions.
0b1	Any region that is writable in the EL1&0 translation regime is forced to XN for accesses from software executing at EL1 or EL0.

This bit applies only when SCTLR_EL1.M bit is set.

The WXN bit is permitted to be cached in a TLB.

When FEAT_VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1, 1}, this bit has no effect on the PE.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

nTWE, bit [18]

Traps EL0 execution of WFE instructions to EL1, or to EL2 when it is implemented and enabled for the current

Security state and HCR_EL2.TGE is 1, from both Execution states, reported using an ESR_ELx.EC value of 0x01.

When FEAT_WFxT is implemented, this trap also applies to the WFET instruction.

nTWE	Meaning
060	Any attempt to execute a WFE instruction at EL0 is trapped, if the instruction would otherwise have caused the PE to enter a low-power state.
0b1	This control does not cause any instructions to be trapped.

In AArch32 state, the attempted execution of a conditional WFE instruction is only proper if the instruction passes its condition code check.

Since a WFE or WFI can complete at any time, even without a V'_{a} keup vent, the tradient on WFE of WFI are not guaranteed to be taken, even if the WFE or WFI is executed which there is no Vakeur vent. The only guarantee is that if the instruction does not complete in finite time in the cosence of a Wak. A vent, the trap will be taken.

When FEAT_VHE is implemented, and the value of HCk_ L2 _2H, TGF} is {1, 1}, this bit has no effect on execution at EL0.

The reset behavior of this field is:

Bit [17]

Reserved, RESO.

nTWI, bit [16]

Traps EL0 execution of WFI instructions EL1, or to EL2 when it is implemented and enabled for the current Security state and a CR_EL GE is 1, from both Execution states, reported using an ESR_ELx.EC value of 0x01.

When FEAT_WFx. ______plement_____, this trap also applies to the WFIT instruction.

nTWI	Meaning
000	Any attempt to execute a WFI instruction at EL0 is trapped, if the instruction would otherwise have caused the PE to enter a low-power state.
0b1	This control does not cause any instructions to be trapped.

In AArch32 state, the attempted execution of a conditional WFI instruction is only trapped if the instruction passes its condition code check.

Since a WFE or WFI can complete at any time, even without a Wakeup event, the traps on WFE of WFI are not guaranteed to be taken, even if the WFE or WFI is executed when there is no Wakeup event. The only guarantee is that if the instruction does not complete in finite time in the absence of a Wakeup event, the trap will be taken.

When FEAT_VHE is implemented, and the value of HCR_EL2. $\{E2H, TGE\}$ is $\{1, 1\}$, this bit has no effect on execution at EL0.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

UCT, bit [15]

Traps EL0 accesses to the CTR_EL0 to EL1, or to EL2 when it is implemented and enabled for the current Security state and HCR_EL2.TGE is 1, from AArch64 state only, reported using an ESR_ELx.EC value of 0x18.

UCT	Meaning
0b0	Accesses to the CTR_EL0 from EL0 using AArch64 are trapped.
0b1	This control does not cause any instructions to be trappe

When FEAT_VHE is implemented, and the value of HCR_EL2 E2H, E2H, E1, 1, 1, this bit has no effect on execution at EL0.

The reset behavior of this field is:

DZE, bit [14]

Traps EL0 execution of DC ZVA instructions to "L1 , to EL2 ... en it is implemented and enabled for the current Security state and HCR_EL2.TGE is 1, m AA ... 64 state only, reported using an ESR_ELx.EC value of 0x18.

If FEAT_MTE is implemented, t¹ trap also plies DC GVA and DC GZVA.

DZE	Meaning
060	Any attempt to execute an instruction that this trap applies to at EL0 using AArch64 is trapped. Reading DCZID_EL0.DZP from EL0 returns 1, indicating that the instructions this trap applies to are not supported.
0b1	This control does not cause any instructions to be trapped.

When FEAT____IE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1, 1}, this bit has no effect on execution at EL0.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

EnDB, bit [13]

4

When FEAT_PAuth is implemented:

Controls enabling of pointer authentication (using the APDBKey_EL1 key) of instruction addresses in the EL1&0 translation regime.

For more information, see System register control of pointer authentication .

EnDB	Meaning
0b0	Pointer authentication (using the APDBKey_EL1 key) of data addresses is not enabled.
0b1	Pointer authentication (using the APDBKey_EL1 key) of data addresses is enabled.

This field controls the behavior of the AddPACDB and AuthDB pseudocode functions. Specifically, when the field is 1, AddPACDB returns a copy of a pointer to which a pointer authentication code has been added, and AuthDB returns an authenticated copy of a pointer. When the field is 0, both of these functions are NOP.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOW alue.

Otherwise:

res0

l, bit [12]

Stage 1 instruction access Cacheability control, fr ... ses at

Meaning

0 and

All instruction access to Stage 1 Normal memory from EL0 and EL1 are Stage 1 Non-cacheable. If the value of SCTLR_EL1.M is 0, instruction accesses from stage 1 of the EL1&0 translation regime are to Normal, Outer Shareable, Inner Non-cacheable, Outer Non-cacheable memory.
Ob1
Ob1
This control has no effect on the Stage 1 Cacheability of instruction access to Stage 1 Normal memory from EL0 and EL1. If the value of SCTLR_EL1.M is 0, instruction accesses from stage 1 of the EL1&0 translation regime are to Normal, Outer Shareable, Inner Normal memory from EL0 and EL1. If the value of SCTLR_EL1.M is 0, instruction accesses from stage 1 of the EL1&0 translation regime are to Normal, Outer Shareable, Inner Write-Through, Outer Write-Through memory.

When the value of the HCR_EL2.DC bit is 1, then instruction access to Normal memory from EL0 and EL1 are Cacheable regardless of the value of the SCTLR_EL1.I bit.

When FEAT_VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is $\{1, 1\}$, this bit has no effect on the PE.

The reset behavior of this field is:

- On a Warm reset:
 - When EL2 is not implemented and EL3 is not implemented, this field resets to 0b0.
 - Otherwise, this field resets to an architecturally UNKNOWN value.

EOS, bit [11]

When FEAT_ExS is implemented:

Chapter E3. System registers affected by SME E3.2. Changes to existing System registers

Exception Exit is Context Synchronizing.

EOS	Meaning
0b0	An exception return from EL1 is not a context synchronizing event
0b1	An exception return from EL1 is a context synchronizing event

When FEAT_VHE is implemented, and the value of HCR_EL2. $\{E2H, TGE\}$ is $\{1,1\}$, this bit has no effect on execution at EL0.

If SCTLR_EL1.EOS is set to 0b0:

- Memory transactions, including instruction fetches, from an Excertion rel always use the translation resources associated with that translation regime.
- Exception Catch debug events are synchronous debug events.
- DCPS* and DRPS instructions are context synchronization vents.

The following are not affected by the value of SCTLR_EL1 JS:

- The indirect write of the PSTATE and PC values . m SPS _EL1 and ELR_EL1 on exception return is synchronized.
- Behavior of accessing the banked copies of the point rusing the SP register name for loads, stores and data processing instructions.
- Exit from Debug state.

The reset behavior of this field is:

• On a Warm reset, this field reases to a. rchitec "ally UNKNOWN value.

Otherwise:

res1

EnRCTX, bit [10]

When FEAT_SP1 'RE' s implemented:

Enable F' ... ss to . follow .g System instructions:

- .FPRCT DVPRC1 and CPPRCTX instructions.
- PF 1X, Dv. CTX and CPP RCTX instructions.

EnRCTX	Meaning
060	EL0 access to these instructions is disabled, and these instructions are trapped to EL1, or to EL2 when it is implemented and enabled for the current Security state and HCR_EL2.TGE is 1.
0b1	EL0 access to these instructions is enabled.

When FEAT_VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is $\{1,1\}$, this bit has no effect on execution at EL0.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

res0

UMA, bit [9]

User Mask Access. Traps EL0 execution of MSR and MRS instructions that access the PSTATE.{D, A, I, F} masks to EL1, or to EL2 when it is implemented and enabled for the current Security state and HCR_EL2.TGE is 1, from AArch64 state only, reported using an ESR_ELx.EC value of 0x18.

	UMA	Meaning
	0d0	Any attempt at ELO using AArch64 to execute an MRS, M. (RELETER), OF MSR (IMMEDIATE) instrue on that acceles the DAIF is trapped.
	0b1	This trol does not use any instructions to e trapp.
When FEAT_VHE is implemented, and the va execution at EL0.	alue of HCk_	$TL2 = 2H, TGF$ is {1, 1}, this bit has no effect on
The reset behavior of this field is:		
• On a Warm reset, this field resets to an	'hitec+	NOWN value.

SED, bit [8]

When EL0 is capable of usin (\rch32:

SETEND instruction disable. Disable SETEND instructions at EL0 using AArch32.

SED	Meaning
000	SETEND instruction execution is enabled at EL0 using AArch32.
0b1	SETEND instructions are UNDEFINED at EL0 using AArch32 and any attempt at EL0 to access a SETEND instruction generates an exception to EL1, or to EL2 when it is implemented and enabled for the current Security state and HCR_EL2.TGE is 1, reported using an ESR_ELx.EC value of 0x00.

If the implementation does not support mixed-endian operation at any Exception level, this bit is RES1.

When FEAT_VHE is implemented, and the value of HCR_EL2. $\{E2H, TGE\}$ is $\{1, 1\}$, this bit has no effect on execution at EL0.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

res1

ITD, bit [7]

When EL0 is capable of using AArch32:

IT Disable. Disables some uses of IT instructions at EL0 using AArch32.

Meaning All IT instruction functionality is enabled at EL0 using AArch32. Any attempt at EL0 using AArch32 to execute any of the following is UNDEFINED and generates an ucception, reported using an ESR_F'EC va. of 0x00, to EL1 or to EL2 wher is implemen. d and enabled for the currer. Security state and HCR_EL2.TGE is 1: • An accodings the IT instruction with hw11, ``!!='` J0. All encogs of the subsequent instruction with the following values for hw • 0b11xxxxxxxxxxx All 32-bit instructions, and the 16-bit instructions B, UDF, SVC, LDM, and STM. • 0b1011xxxxxxxxxx All instructions in 'Miscellaneous 16-bit instructions'. • 0b10100xxxxxxxxxx ADD Rd
 All IT instruction functionality is enabled at EL0 using AArch32. Any attempt at EL0 using AArch32 to execute any of the following is UNDEFINED and generates ar sucception, reported using an ESR_F'EC va. of 0x00, to EL1 or to EL2 when this implement of and enabled for the curret. Security state and HCR_EL2.TGE is 1: A. accodings the IT instruction with hw11. 1!=' .00. All encodings of the subsequent instruction with the following values for hwv. Ob11xxxxxxxxxxxx: All 32-bit instructions B, UDF, SVC, LDM, and STM. Ob1011xxxxxxxxxx: All instructions in 'Miscellaneous 16-bit instructions'.
 Any attempt at EL0 using AArch32 to execute any of the following is UNDEFINED and generates an ucception, reported using an ESR_F'EC va. of 0x00, to EL1 or to EL2 wher c is implement d and enabled for the currer. Security state ind HCR_EL2.TGE is 1: An incodings the IT instruction with hw11. 1!=' 0. All encogs of the subsequent instruction with the following values for hw Ob11xxxxxxxxxxx All 32-bit instructions B, UDF, SVC, LDM, and STM. Ob1011xxxxxxxxxxx All instructions in 'Miscellaneous 16-bit instructions'. Ob10100xxxxxxxxxx ADD Rd
 PC, #imm Ob01001xxxxxxxxx: LDR Rd, [PC, #imm] Ob0100x1xxx1111xxx: ADD Rdn, PC; CMP Rn, PC; MOV Rd, PC; BX PC; BLX PC. Ob010001xx1xxxx111: ADD PC, Rm: CMP PC Rm: MOV PC Rm
This pattern also covers unpredictable cases with BLX Rn.
These instructions are always UNDEFINED, regardless of whether they would pass or fail the condition code check that applies to them as a
result of being in an IT block. It is IMPLEMENTATION DEFINED whether the IT instruction is treated as:
• A 16-bit instruction, that can only be followed by another 16-bit instruction.
• The first half of a 32-bit instruction. This means that, for the situations that are
UNDEFINED, either the second 16-bit instruction or the 32-bit instruction is UNDEFINED. An implementation might vary dynamically as to

If an instruction in an active IT block that would be disabled by this field sets this field to 1 then behavior is CONSTRAINED UNPREDICTABLE. For more information, see Changes to an ITD control by an instruction in an IT block .

ITD is optional, but if it is implemented in the SCTLR_EL1 then it must also be implemented in the SCTLR_EL2, HSCTLR, and SCTLR.

When FEAT_VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1, 1}, this bit has no effect on execution at EL0.

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 ITD, access to this field is RAZ/WI.

Otherwise:

res1

nAA, bit [6]

When FEAT_LSE2 is implemented:

Non-aligned access. This bit controls generation of Alignme alts at EI and EL0 under certain conditions.

The following instructions generate an Alignme fault all byte eigencessed are not within a single 16-byte quantity, aligned to 16 bytes for access:

- LDAPR, LDAPRH, LDAPUR, LDAP, Y LDAPURSH, LDAPURSW, LDAR, LDLAR, LDLARH.
- STLLR, STLLRH, STLR, CLRH, ST. R, and TLURH.

nAA	Meaning
0b0	Unaligned accesses by the specified instructions generate an Alignment fault.
0b1	This control does not generate Alignment faults.

Wh. FEAT 2^{-12} is implemented, and the value of HCR_EL2. {E2H, TGE} is {1, 1}, this bit has no effect on executive 2L0.

The reset Lavior of this field is:

• On a War.n reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

res0

CP15BEN, bit [5]

When EL0 is capable of using AArch32:

System instruction memory barrier enable. Enables accesses to the DMB, DSB, and ISB System instructions in the (coproc==0b1111) encoding space from EL0:

CP15BEN	Meaning
060	EL0 using AArch32: EL0 execution of the CP15DMB, CP15DSB, and CP15ISB instructions is UNDEFINED and generates an exception to EL1, or to EL2 when it is implemented and enabled for the current Security state and HCR_EL2.TGE is 1. The exception is reported using an ESR_ELx.EC value of 0x00.
0b1	EL0 using AArch32: EL0 execution of the CP15DMB, CP15DSB, and CP15ISB instruction of the bled.

CP15BEN is optional, but if it is implemented in the SCTLR EL1 . n it must a \rightarrow be implemented in the SCTLR_EL2, HSCTLR, and SCTLR.

When FEAT_VHE is implemented, and the value of HC^r_ $L2.{E^{<math>r$}}, TGE} (1, 1), this bit has no effect on execution at EL0.

The reset behavior of this field is:

• On a Warm reset, this field resets to an *r* nitecture by UNKNG A value.

When an implementation does not implement 215^r and accord this field is **RAO/WI**.

Otherwise:

res0

SA0, bit [4]

SP Alignment check hable for EL0. Whose to 1, if a load or store instruction executed at EL0 uses the SP as the base address *P*, the SP is not aligned to a 16-byte boundary, then an SP alignment fault exception is generated. For more information, see SP alignment checking'.

When FE^{*} 'HE is plemen' 1, and the value of HCR_EL2.{E2H, TGE} is {1, 1}, this bit has no effect on execution at EL

The set beb field is:

• On. Narm reset, this field resets to an architecturally UNKNOWN value.

SA, bit [3]

SP Alignment check enable. When set to 1, if a load or store instruction executed at EL1 uses the SP as the base address and the SP is not aligned to a 16-byte boundary, then an SP alignment fault exception is generated. For more information, see 'SP alignment checking'.

When FEAT_VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1, 1}, this bit has no effect on the PE.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

C, bit [2]

Stage 1 Cacheability control, for data accesses.

С	Meaning			
060	All data access to Stage 1 Normal memory from EL0 and EL1, and all Normal memory accesses from unified cache to the EL1&0 Stage 1 translation tables, are treated as Stage 1 Non-cacheable.			
0b1	 This control has no effect on the Stage 1 Cacheability of: Data access to Normal memory from EL0 and EL1. 			
	Normal memory accesses to the EL1&0 Street and relation tables.			

When the Effective value of the HCR_EL2.DC bit in the current security statis 1, the PE ignores SCTLR_EL1.C. This means that EL0 and EL1 data accesses to Normal means that C sheaple

When FEAT_VHE is implemented, and the Effective value of $P \times EL2.\{F2H, TGE\}$ is $\{1, 1\}$, this bit has no effect on the PE.

The reset behavior of this field is:

- On a Warm reset:
 - When EL2 is not implemented and Ex not implemented, this field resets to 0b0.
 - Otherwise, this field reset to a. chitec. ally UNKNOWN value.

A, bit [1]

Alignment check enable This is the c ble bit for Alignment fault checking at EL1 and EL0.

A	Meaning
040	Alignment fault checking disabled when executing at EL1 or EL0. Instructions that load or store one or more registers, other than load/store exclusive and load-acquire/store-release, do not check that the address being accessed is aligned to the size of the data element(s) being accessed.
0b1	Alignment fault checking enabled when executing at EL1 or EL0. All instructions that load or store one or more registers have an alignment check that the address being accessed is aligned to the size of the data element(s) being accessed. If this check fails it causes an Alignment fault, which is taken as a Data Abort exception.

Load/store exclusive and load-acquire/store-release instructions have an alignment check regardless of the value of the A bit.

If FEAT_MOPS is implemented, SETG* instructions have an alignment check regardless of the value of the A bit.

When FEAT_VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1, 1}, this bit has no effect on execution at EL0.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

M, bit [0]

MMU enable for EL1&0 stage 1 address translation.

М	Meaning
000	EL1&0 standaress translation disabled. See the CTLR_1 1.I field for the behavior of instruction accesses. Normal memory.
0b1	Ll& age 1 addre translation enabled.

If the Effective value of HCR_EL2. {DC, TGE} in the curve $\frac{1}{2}$ Secve $\frac{1}{2}$ state is not {0, 0} then the PE behaves as if the value of the SCTLR_EL1. M field is 0 for all purposes other in returning the value of a direct read of the field.

When FEAT_VHE is implemented, and the Eff ave v is of $H \ge 1.2$.{E2H, TGE} is {1, 1}, this bit has no effect on the PE.

The reset behavior of this field is:

- On a Warm reset:
 - When EL2 is not implemented and ".3 is in implemented, this field resets to 0b0.
 - Otherwise, this field, ts to an arch, turally UNKNOWN value.

Accessing SCTLF_EL1

When HCR_EL2 2H is 1, hout explicit synchronization, access from EL3 using the mnemonic SCTLR_EL1 or SCTLR_EL12 on transfer to be ordered with respect to accesses using the other mnemonic.

Accesses registing the nowing encodings in the System register encoding space:

MRS Xt>, SC LR_EL

ор0	op1	CRn	CRm	op2
0b11	0b000	0b0001	0b0000	0b000

```
1
   if PSTATE.EL == ELO then
2
       UNDEFINED;
   elsif PSTATE.EL == EL1 then
3
4
       if EL2Enabled() && HCR_EL2.TRVM == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
5
6
       elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn
            ↔== '1') && HFGRTR_EL2.SCTLR_EL1 == '1' then
7
            AArch64.SystemAccessTrap(EL2, 0x18);
8
       elsif EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '111' then
9
           X[t, 64] = NVMem[0x110];
10
       else
            X[t, 64] = SCTLR\_EL1;
11
   elsif PSTATE.EL == EL2 then
12
13
       if HCR_EL2.E2H == '1' then
```
```
14 X[t, 64] = SCTLR_EL2;
15 else
16 X[t, 64] = SCTLR_EL1;
17 elsif PSTATE.EL == EL3 then
18 X[t, 64] = SCTLR_EL1;
```

MSR SCTLR_EL1, <Xt>

1

16

17

18

	op0	op1	CRn	CRm	op2
	0b11	0b000	0b0001	0b0000	0b000
if PSTATE.EL UNDEFINEN elsif PSTATE	== ELO then D; .EL == EL1 th	en			
if EL2Ena AArcl elsif EL2	abled() && HC h64.SystemAcc 2Enabled() &&	R_EL2.TVM == '1' essTrap(EL2, 0x18 IsFeatureImpleme	then); nted(FEA ⁺ _GT) &&	(!h、 >EL(.3)	SCR_EL3.FGTEn
↔== AArcl	'1') && HFGW h64.SystemAcc	TR_EL2.SCTLR_EL1 essTrap(EL2, 0x18	== '1' .ien);		
elsif EL2 NVMer	2Enabled() && m[0x110] = X[HCR_EL2. <nv2,nv1 t, 64];</nv2,nv1 	,NV> == '1' the	n	
else SCTLE	$R_{EL1} = X[t,$	64];			
elsif PSTATE if HCR_EI	.EL == EL2 th L2.E2H == '1'	en then			

```
MRS <Xt>, SCTLR_E' 12
```

elsif PSTATE.EL == EL3 ther

 $SCTLR_EL1 = X[t, 64]$.

 $SCTLR_EL1 = X[t, 64]$

else

	` 0	op1	CRn	CRm	op2
	0611	0b101	0b0001	0b0000	0b000
1	if PS TE _ == ELC .nen				
2	UNL. NED;				
3	elsif PSA T.EL == EL1 then				
4	if EL2L 'bled() && HCR_EL	2. <nv2,nv1,nv< th=""><th>> == '101' then</th><th></th><th></th></nv2,nv1,nv<>	> == '101' then		
5	X[t, 64] = NVMem[0x11]	0];			
6	elsif EL2Enabled() && HCR	_EL2.NV == '1	' then		
7	AArch64.SystemAccessT	rap(EL2, 0x18);		
8	else				
9 10	UNDEFINED;				
10	if ucp FI2 F24 11 the	_			
12	$\frac{11}{2} \operatorname{HCR}_{\text{EL2},\text{EZR}} = 1 \operatorname{CHe}_{\text{I}}$				
13	else				
14	UNDEFINED:				
15	elsif PSTATE.EL == EL3 then				
16	if EL2Enabled() && !ELUsin	ngAArch32 (EL2) && HCR_EL2.E2H =	= '1' then	
17	$X[t, 64] = SCTLR_EL1;$	-			
18	else				
19	UNDEFINED;				

MSR SCTLR_EL12, <Xt>

ор0	op1	CRn	CRm	op2
0b11	0b101	0b0001	0b0000	0b000

th

```
if PSTATE.EL == EL0 then
1
2
       UNDEFINED;
3
   elsif PSTATE.EL == EL1 then
4
       if EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '101' then
           NVMem[0x110] = X[t, 64];
5
       elsif EL2Enabled() && HCR_EL2.NV == '1' then
6
7
           AArch64.SystemAccessTrap(EL2, 0x18);
8
       else
           UNDEFINED;
9
10
   elsif PSTATE.EL == EL2 then
11
       if HCR_EL2.E2H == '1' then
           SCTLR\_EL1 = X[t, 64];
12
13
        else
14
           UNDEFINED;
   elsif PSTATE.EL == EL3 then
15
16
       if EL2Enabled() && !ELUsingAArch32(EL2) && HCP _L2.E2H ==
17
           SCTLR_EL1 = X[t, 64];
18
        else
19
           UNDEFINED;
```

E3.2.17 SCTLR_EL2, System Control Register (EL2)

The SCTLR_EL2 characteristics are:

Purpose

Provides top level control of the system, including its memory system, at EL2.

When FEAT_VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is $\{1, 1\}$, these controls apply also to execution at EL0.

Configuration

If EL2 is not implemented, this register is RESO from EL3.

This register has no effect if EL2 is not enabled in the current Security state.

AArch64 system register SCTLR_EL2 bits [31:0] are architect ally map₁ ¹ to AArch32 system register HSCTLR[31:0].

Attributes

SCTLR_EL2 is a 64-bit register.

Field descriptions

The SCTLR_EL2 bit assignments are:



TIDCP, bit [63]

When FEAT_TIDCP1 is implemented and HCR_EL2.E2H == 1:

Trap IMPLEMENTATION DEFINED functionality. Traps EL0 accesses to the encodings reserved for IMPLEMENTA-TION DEFINED functionality to EL2.

TIDCP	Meaning
060	No instructions accessing the System register or System instruction spaces are trapped by this mechanism.
	 If HCR_EL2.TGE==0, no instructions accessing the System register or System instruction spaces are trapped by this mechanism. If HCR_EL2.TGE==1, instructions accessing the following System register or System instruction spaces are trapped to EL2 by this mechanism: In

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

res0

SPINTMASK, bit [62]

When FEAT_NMI is implemented:

SP Interrupt Mask enable. When SCTLR_EL2.NMI is 1, controls whether PSTATE.SP acts as an interrupt mask, and controls the value of PSTATE.ALLINT on taking an exception to EL2.

SPINTMASK	Meaning
0b0	Does not cause PSTATE.SP to mask interrupts. PSTATE.ALLINT is set to 1 on taking an exception to EL2.
0b1	When PSTATE.SP is 1 and execution is at EL2, an IRQ or FIQ interrupt that is targeted to EL2 is masked regardless of any denotion of Superpriority. PSTATE.ALLINT is set to 0 on taking an

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOW. Talue.

Otherwise:

res0

NMI, bit [61]

When FEAT_NMI is implemented:

Non-maskable Interrupt enable.

Nr.	Meaning
0d.	This control does not affect interrupt masking behavior.
0b1	 This control enables all of the following: The use of the PSTATE.ALLINT interrupt mask. IRQ and FIQ interrupts to have Superpriority as an additional attribute. PSTATE.SP to be used as an interrupt mask.

The reset behavior of this field is:

• On a Warm reset, this field resets to Ob0.

Otherwise:

res0

EnTP2, bit [60]

When FEAT_SME is implemented, HCR_EL2.E2H == 1 and HCR_EL2.TGE == 1

EnTP2, bit [60]

Traps instructions executed at EL0 that access TPIDR2_EL0 to EL2 when EL2 is implemented and enabled for the current Security state. The exception is reported using ESR_ELx.EC value 0x18.

EnTP2	Meaning
060	This control causes execution of these instructions at EL0 to be trapped.
0b1	This control does not cause execution of any instructions to be trapped.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SME is implemented, HCR_EL2.E2H == 1 and HCR_EL2.TGE == 0

0]

IGNORED.

Otherwise:

res0

Bits [59:58]

Reserved, RESO.

EPAN, bit [57]

When FEAT_PAN3 is implement/ HCh. 7L2.E2. == 1 and HCR_EL2.TGE == 1

EPAN, bit [57]

Enhanced Privileged Access Never. en PSTATE.PAN is 1, determines whether an EL2 data access to a page with EL0 instruction cess permission nerates a Permission fault as a result of the Privileged Access Never mechanism.

EPAN	Meaning
0b0	No additional Permission faults are generated by this mechanism.
0b1	An EL2 data access to a page with stage 1 EL0 data access permission or stage 1 EL0 instruction access permission generates a Permission fault. Any speculative data accesses that would generate a Permission fault as a result of PSTATE.PAN = 1 if the accesses were not speculative, will not cause an allocation into a cache.

This bit is permitted to be cached in a TLB.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_PAN3 is implemented, HCR_EL2.E2H == 1 and HCR_EL2.TGE == 0

EPAN, bit [57] IGNORED. Otherwise: RESO EnALS, bit [56]

When FEAT_LS64 is implemented, HCR_EL2.E2H == 1 and HCR_EL2.TGE == 1

EnALS, bit [56]

Traps execution of an LD64B or ST64B instruction at EL0 to EL2.

	EnALS	Me ^r ng
	0ъ0	ELO is trapp. ' to 22 .
	0b1	T ⁺ control does not cause any instructions to c trapped
trap of an LD64B or ST64B instruction is po	orted y log ar	ESR_£Lx.EC value of 0x0A, with an ISS code of

A trap of an LD64B or ST64B instruction is ported v log an ESR_eLx.EC value of 0x0A, with an ISS code of 0x0000002.

The reset behavior of this field is:

• On a Warm reset, this field esets to an a "ritectur aly UNKNOWN value.

When FEAT_LS64 is imp/ cente. HCR_EL2.E. H == 1 and HCR_EL2.TGE == 0

EnALS, bit [56]

IGNORED.

Otherwise ·

res0

En, `?, bit

When FL. TLS64_ACCDATA is implemented, HCR_EL2.E2H == 1 and HCR_EL2.TGE == 1 EnAS0, bit [:]

Traps execution of an ST64BV0 instruction at EL0 to EL2.

EnAS0	Meaning
0b0	Execution of an ST64BV0 instruction at EL0 is trapped to EL2.
0b1	This control does not cause any instructions to be trapped.

A trap of an ST64BV0 instruction is reported using an ESR_ELx.EC value of 0x0A, with an ISS code of 0x0000001. The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_LS64_ACCDATA is implemented, HCR_EL2.E2H == 1 and HCR_EL2.TGE == 0

EnAS0, bit [55]

IGNORED.

Otherwise:

res0

EnASR, bit [54]

When FEAT_LS64_V is implemented, HCR_EL2.E2H == 1 and HCR_EL2.TGE == 1 EnASR, bit [54]

Traps execution of an ST64BV instruction at EL0 to EL2.

EnAS	Mezing
060	.xecution an ST64BV instruction at EL0 is apped EL2.
0b1	The control does not cause any instructions to be trapped.

A trap of an ST64BV instruction *i* eported up g an E ELx.EC value of 0x0A, with an ISS code of 0x0000000. The reset behavior of this fie' 4s.

• On a Warm reset, this field reset of an architecturally UNKNOWN value.

When FEAT_LS64 is implemented, h ?_EL2.E2H == 1 and HCR_EL2.TGE == 0

EnASR, bit [54]

IGNORF

Oth wise:

res0

TME, bit 💽

When FEAT_TME is implemented:

Enables the Transactional Memory Extension at EL2.

TME	Meaning
0b0	Any attempt to execute a TSTART instruction at EL2 is trapped, unless HCR_EL2.TME or SCR_EL3.TME causes TSTART instructions to be UNDEFINED at EL2.
0b1	This control does not cause any TSTART instruction to be trapped.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

res0

TME0, bit [52]

When FEAT_TME is implemented, HCR_EL2.E2H == 1 and HCR_EL2.TGE == 1 TME0, bit [52]

Enables the Transactional Memory Extension at EL0.

	TMF0	Mes on
	060	Any attent to exect a TSTART instruction at EL0 is traph to 22, unless HCR_EL2.TME
		or R_EL3.1 E causes TSTART instructions oe UNDF ^{TI} NED at EL0.
		This cor of does not cause any TSTART
l is:		
ld e stata ar sk	itaa Ilr. T	

The reset behavior of this field is:

• On a Warm reset, this field ______ets to an ______hitecu. _____ly UNKNOWN value.

When FEAT_TME is imple _n. ' HCR_EL2. `H == 1 and HCR_EL2.TGE == 0

TME0, bit [52]

IGNORED.

Otherwise: RESO

TM^r bit [51]

When F. *I_TME* is implemented:

Forces a triv, implementation of the Transactional Memory Extension at EL2.

ТМТ	Meaning
0b0	This control does not cause any TSTART instruction to fail.
0b1	When the TSTART instruction is executed at EL2, the transaction fails with a TRIVIAL failure cause.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

TMT0, bit [50]

When FEAT_TME is implemented, HCR_EL2.E2H == 1 and HCR_EL2.TGE == 1

TMT0, bit [50]

Forces a trivial implementation of the Transactional Memory Extension at EL0.

	ТМТО	Meaning
	060	This control does not cause any TSTART instruction on
	0b1	When the TSTART struction is executed at ELO, transaction is with a TRIVIAL silure c
The reset behavior of this field is:		
• On a Warm reset, this field resets to an arch;	'ly un	'OWN · .ue.
When FEAT_TME is implemented, HCR_FE.	2H = 1 and	$H \cup _EL2.TGE == 0$
TMT0, bit [50]		
IGNORED.		
Otherwise:		
RES0		
TWEDEL, bits [49 5]		
When FEAT_TV, ") is plemented, HCR_EL2.	E2H == 1 ai	nd HCR_EL2.TGE == 1
TWEDE/ [3:0] `bits [4′ +6]		
TW Delay. A +-bit unsigned number that, when takine trar - wine sed by SCTLR_EL2.nTW	SCTLR_EL /E as 2 ^{(TWED}	2.TWEDEn is 1, encodes the minimum delay in $^{EL+8)}$ cycles.

The reset . 'vavior of this field is:

• On a Wa. a reset, this field resets to an architecturally UNKNOWN value.

When FEAT_TWED is implemented, HCR_EL2.E2H == 1 and HCR_EL2.TGE == 0

TWEDEL, bits [3:0] of bits [49:46]

IGNORED.

Otherwise:

TWEDEn, bit [45]

When FEAT_TWED is implemented, HCR_EL2.E2H == 1 and HCR_EL2.TGE == 1 TWEDEn, bit [45]

TWE Delay Enable. Enables a configurable delayed trap of the WFE instruction caused by SCTLR_EL2.nTWE.

	TWEDEn	Meaning
	060	The delay for taking a WFE trap is IMPLEMENT ON DEFINED.
	0b1	The dr .y for takin. WFE trap is at least the nun. r of cycles den ed in <u>CTL</u> , FL2.TWED 2.
The reset behavior of this field is:		
• On a Warm reset, this field resets to an archit	termally U.	NOWN y ^e .e.
When FEAT_TWED is implemented, HCR_F 2	.E2H = 1 a	na. ^c EL2.TGE == 0
TWEDEn, bit [45]		
IGNORED.		
Otherwise:		
RES0		
DSSBS, bit [44]		
When FEAT_S_ S is im mented:		
Default PST^TE.SS1 value on ^r ception Entry.		
	DSSBS	Meaning
	01.0	DETATE SEDS is set to 0 on on execution to

D22B2	Meaning
0b0	PSTATE.SSBS is set to 0 on an exception to EL2.
0b1	PSTATE.SSBS is set to 1 on an exception to EL2.

The reset behavior of this field is:

• On a Warm reset, this field resets to an IMPLEMENTATION DEFINED value.

Otherwise:

res0

ATA, bit [43]

When FEAT_MTE2 is implemented:

Allocation Tag Access in EL2.

When SCR_EL3.ATA is 1, controls access to Allocation Tags and Tag Check operations in EL2.

	ATA	Meaning
	060	Access to Allocation Tags is prevented at EL2. Memory accesses at EL2 are not subject to a Tag Check operation.
	0b1	This control does not prevent access to Allocation Tags at EL2. Tag Checked memory accesses at EL2 are subject to a Tay Check operation.
The reset behavior of this field is:		
• On a Warm reset, this field resets to an	architecturally t	v snown vai
Otherwise:		
RES0		
ATA0, bit [42]		
When FEAT_MTE2 is implemented, HCR_	F' = I	u <i>HCR_EL2.TGE</i> == 1
ATA0, bit [42]		
Allocation Tag Access in EL0		
When SCR_EL3.ATA is 1, controls ress to	o Allocation Tag	s and Tag Check operations in EL0.
	ATA0	Meaning
	060	Access to Allocation Tags is prevented at EL0. Memory accesses at EL0 are not subject to a Tag

Check operation.

Allocation Tags at EL0.

This control does not prevent access to

subject to a Tag Check operation.

Tag Checked memory accesses at EL0 are

Software may change this control bit on a context switch.

The reset behavior of this field is:

6

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_MTE2 is implemented, HCR_EL2.E2H == 1 and HCR_EL2.TGE == 0

ATA0, bit [42]

IGNORED.

Otherwise:

0b1

TCF, bits [41:40]

When FEAT_MTE2 is implemented:

Tag Check Fault in EL2. Controls the effect of Tag Check Faults due to Loads and Stores in EL2.

TCF	Meaning	Applies
0000	Tag Check Faults have no effect on the PE.	
0b01	Tag Check Faults cause a synchronous exception.	
0b10	Tag Check Faults are asynchron siy accumulated.	
0b11	Tag Check Faults cause <i>p</i> syncther nous exception on reads, are are asyncher nously accumulated on wrights.	When FEAT_MTE3 is implemented

If FEAT_MTE3 is not implemented, the value 0b¹ servel

The reset behavior of this field is:

• On a Warm reset, this field resets to an a. vite

Otherwise:

RESO

TCF0, bits [39:38]

When FEAT_MTE^{*} s implemented, Hu_EL2.E2H == 1 and HCR_EL2.TGE == 1 TCF0, bits [1:0 vf bits / .:38]

Tag Chec¹ in EL Contro¹ the effect of Tag Check Faults due to Loads and Stores in EL0.

Meaning	Applies
Tag Check Faults have no effect on the PE.	
Tag Check Faults cause a synchronous exception.	
Tag Check Faults are asynchronously accumulated.	
Tag Check Faults cause a synchronous exception on reads, and are asynchronously accumulated on writes.	When FEAT_MTE3 is implemented
	MeaningTag Check Faults have no effect on the PE.Tag Check Faults cause a synchronous exception.Tag Check Faults are asynchronously accumulated.Tag Check Faults cause a synchronous exception on reads, and are asynchronously accumulated on writes.

If FEAT_MTE3 is not implemented, the value 0b11 is reserved.

Software may change this control bit on a context switch.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_MTE2 is implemented, HCR_EL2.E2H == 1 and HCR_EL2.TGE == 0

TCF0, bits [1:0] of bits [39:38]

IGNORED.

Otherwise:

res0

ITFSB, bit [37]

When FEAT_MTE2 is implemented:

When synchronous exceptions are not being generated by Tag Check Faults, this field controls whether on exception entry into EL2, all Tag Check Faults due to instructions executed from seption entry, that are reported asynchronously, are synchronized into TFSRE0_EL1, TFSR_EL1 and rSR_EL2, isters.



res0

BT, bit [36]

Otherwise:

When FEAT_BT1. 'r .emented

PAC Br cn 1, comp 'bilit at EL2.

When $HCR_F' \subseteq \{F2H, TGE\} == \{1, 1\}$, this bit is named BT1.

ВТ	Meaning
000	When the PE is executing at EL2, PACIASP and PACIBSP are compatible with PSTATE.BTYPE $= 0b11$.
0b1	When the PE is executing at EL2, PACIASP and PACIBSP are not compatible with PSTATE.BTYPE == 0b11.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

res0

BT0, bit [35]

When FEAT_BTI is implemented, HCR_EL2.E2H == 1 and HCR_EL2.TGE == 1

BT0, bit [35]

PAC Branch Type compatibility at EL0.

	вто	Meaning
	060	When the PE is executing at EL0, PACIASP and PACIBSP are compatible with PSTATE.BTYPE == 0b11.
	0b1	When the relies continuing at EL0, PACIASP and PAC's P are not constitute with $PSTA = BTYPE = -11.$
The reset behavior of this field is:		
• On a Warm reset, this field resets to an an	rchitecturally	" .OWN val
When FEAT_BTI is implemented, HCR_EL2	2.7 <i>H</i> == ar	nd h ? F 2.TGE == 0
BT0, bit [35]		
IGNORED.		
Otherwise:		
RESO		
Bit [34]		
Reserved, RESO		
MSCEn, bit [33]		
When LAT_N)PS is i. 1 lented, HCR_H	EL2.E2H == 1	and HCR_EL2.TGE == 1
MS. Tn, bit		

Memory vand Memory Set instructions Enable. Enables execution of the Memory Copy and Memory Set instructions . 7L0.

MSCEn	Meaning
060	Execution of the Memory Copy and Memory Set instructions is UNDEFINED at EL0.
0b1	This control does not cause any instructions to be UNDEFINED.

When FEAT_MOPS is implemented and HCR_EL2. {E2H, TGE} is not {1, 1}, the Effective value of this bit is 0b1.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_MOPS is implemented, HCR_EL2.E2H == 1 and HCR_EL2.TGE == 0

MSCEn, bit [33]

IGNORED.

Otherwise:

res0

CMOW, bit [32]

When FEAT_CMOW is implemented and HCR_EL2.E2H == 1:

Controls cache maintenance instruction permission for the following instructions executed at EL0.

• IC IVAU, DC CIVAC, DC CIGDVAC and DC CIGVAC.

CMOW	/ Jeanh
060	Thes instructive executed at EL0 with stage 1 repermission, but without stage 1 write ermission to not generate a stage 1 permission
0b1	The soled as a result of SCTLR_EL2.UCI==1, these instructions executed at EL0 with stage 1.
	read permission, but without stage 1 write permission, generate a stage 1 permission fault.

When HCR_EL2.TGE is 0 ... is bit is no effect of execution at EL0.

For this control, stage ¹ .as write pern. ⁱon if all of the following apply:

- AP[2] is 0 or BM is 1 in the stage 1 descriptor.
- Where AP le is in se, APTable[1] is 0 for all levels of the translation table.

This bit is promitted e cached a TLB.

The re t behaver of this vis:

In a W this field resets to an architecturally UNKNOWN value.

Otherwi.

res0

EnIA, bit [31]

When FEAT_PAuth is implemented:

Controls enabling of pointer authentication (using the APIAKey_EL1 key) of instruction addresses in the EL2 or EL2&0 translation regime.

For more information, see System register control of pointer authentication .

EnIA	Meaning	
0b0	Pointer authentication (using the APIAKey_EL1 key) of instruction addresses is not enabled.	

EnIA	Meaning	
0b1	Pointer authentication (using the APIAKey_EL1 key) of instruction addresses is enabled.	

This field controls the behavior of the AddPACIA and AuthIA pseudocode functions. Specifically, when the field is 1, AddPACIA returns a copy of a pointer to which a pointer authentication code has been added, and AuthIA returns an authenticated copy of a pointer. When the field is 0, both of these functions are NOP.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

res0

EnIB, bit [30]

When FEAT_PAuth is implemented:

Controls enabling of pointer authentication (using the APIB. ____LL1 key) ____instruction addresses in the EL2 or EL2&0 translation regime.

For more information, see System register cr .rol of pc .ter authen. .ation .

rnIB	Meaning
060	Pointer authentication (using the APIBKey_EL1 key) of instruction addresses is not enabled.
0b1	Pointer authentication (using the APIBKey_EL1 key) of instruction addresses is enabled.

This field controls h_{1} avoid of f_{2} AddPACIB and AuthIB pseudocode functions. Specifically, when the field is 1, Add² h_{2} returning copy f_{2} a pointer to which a pointer authentication code has been added, and AuthIB returning an authenticated control is 0, both of these functions are NOP.

The et be

• On Varm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

res0

LSMAOE, bit [29]

When FEAT_LSMAOC is implemented, HCR_EL2.E2H == 1 and HCR_EL2.TGE == 1

LSMAOE, bit [29]

Load Multiple and Store Multiple Atomicity and Ordering Enable.

LSMAOE	Meaning
060	For all memory accesses at EL0, A32 and T32 Load Multiple and Store Multiple can have an interrupt taken during the sequence memory accesses, and the memory accesses are not required to be ordered.
0b1	The ordering and interrupt behavior of A32 and T32 Load Multiple and Store Multiple at EL0 is as defined for Armv8.0.

This bit is permitted to be cached in a TLB.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNK 'OWN 'ue.

When FEAT_LSMAOC is implemented, HCR_EL2.E2H = 1 and HCR_. ?.T' == 0

LSMAOE, bit [29]

IGNORED.

Otherwise:

res1

nTLSMD, bit [28]

When FEAT_LSMAOC is iv u. nted, HCR_ 2.E2H == 1 and HCR_EL2.TGE == 1

nTLSMD, bit [28]

No Trap Load Mu' ple and Store Multiple to Device-nGRE/Device-nGnRE/Device-nGnRnE memory.

nTLSMD	Meaning
040	All memory accesses by A32 and T32 Load Multiple and Store Multiple at EL0 that are marked at stage 1 as Device-nGRE/Device-nGnRE/Device-nGnRnE memory are trapped and generate a stage 1 Alignment fault.
0b1	All memory accesses by A32 and T32 Load Multiple and Store Multiple at EL0 that are marked at stage 1 as Device-nGRE/Device-nGnRE/Device-nGnRnE memory are not trapped.

This bit is permitted to be cached in a TLB.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_LSMAOC is implemented, HCR_EL2.E2H == 1 and HCR_EL2.TGE == 0

nTLSMD, bit [28]

IGNORED.

Otherwise:

res1

EnDA, bit [27]

When FEAT_PAuth is implemented:

Controls enabling of pointer authentication (using the APDAKey_EL1 key) of instruction addresses in the EL2 or EL2&0 translation regime.

For more information, see System register control of pointer authentication

EnDA	Mea. rg
0b0	Pointer au ticati (using the APDAKey_EL1 key) f data a sses is not enabled.
0b1	F inter authentication (using the APDAKey_EL1 key) of d in addresses is enabled.

This field controls the behavior of the AddPA A and conducted functions. Specifically, when the field is 1, AddPACDA returns a copy of a pointer to the a pointer authentication code has been added, and AuthDA returns an authenticated copy of a pointer. When the field is 0, both of these functions are NOP.

The reset behavior of this field ;

• On a Warm reset, this sld re 's to an archit sturally UNKNOWN value.

Otherwise:

res0

UCI, bit [26]

When CR_EL .E2H = ' .d HCR_EL2.TGE == 1

UCı, it [26'

Traps exection of cache maintenance instructions at EL0 to EL2, from AArch64 state only. This applies to DC CVAU, L CIVAC, DC CVAP, and IC IVAU.

If FEAT_DPB2 is implemented, this trap also applies to DC CVADP.

If FEAT_MTE is implemented, this trap also applies to DC CIGVAC, DC CIGDVAC, DC CGVAC, DC CGDVAC, DC CGVAP, and DC CGDVAP.

If FEAT_DPB2 and FEAT_MTE are implemented, this trap also applies to DC CGVADP and DC CGDVADP.

UCI	Meaning	
000	Any attempt to execute an instruction that this trap applies to at EL0 using AArch64 is trapped to EL2.	
0b1	This control does not cause any instructions to be trapped.	

If the Point of Coherency is before any level of data cache, it is IMPLEMENTATION DEFINED whether the execution of any data or unified cache clean, or clean and invalidate instruction that operates by VA to the point of coherency can be trapped when the value of this control is 1.

If the Point of Unification is before any level of data cache, it is IMPLEMENTATION DEFINED whether the execution of any data or unified cache clean by VA to the Point of Unification instruction can be trapped when the value of this control is 1.

If the Point of Unification is before any level of instruction cache, it is IMPLEMENTATION DEFINED whether the execution of any instruction cache invalidate by VA to the Point of Unification instruction can be trapped when the value of this control is 1.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value

When $HCR_EL2.E2H == 1$ and $HCR_EL2.TGE == 0$

UCI, bit [26]

IGNORED.

Otherwise:

res0

EE, bit [25]

Endianness of data accesses at EL2, stage 1 transform table warks in the EL2 or EL2&0 translation regime, and stage 2 translation table walks in the EL. 0 transform regime.

EE	Meaning
060	Explicit data accesses at EL2, stage 1 translation table walks in the EL2 or EL2&0 translation regime, and stage 2 translation table walks in the EL1&0 translation regime are little-endian.
0b1	Explicit data accesses at EL2, stage 1 translation table walks in the EL2 or EL2&0 translation regime, and stage 2 translation table walks in the EL1&0 translation regime are big-endian.

If an implementation does not provide Big-endian support at Exception levels higher than EL0, this bit is RES0. If an implementation does not provide Little-endian support at Exception levels higher than EL0, this bit is RES1. The EE bit is permitted to be cached in a TLB.

The reset behavior of this field is:

• On a Warm reset, this field resets to an IMPLEMENTATION DEFINED value.

E0E, bit [24]

When HCR_EL2.E2H == 1 and HCR_EL2.TGE == 1 E0E, bit [24]

Endianness of data accesses at EL0.

EOE	Meaning
0d0	Explicit data accesses at EL0 are little-endian.
0b1	Explicit data accesses at EL0 are big-endian.

If an implementation only supports Little-endian accesses at EL0, then this bit is RES0. This option is not permitted when SCTLR_EL1.EE is RES1.

If an implementation only supports Big-endian accesses at EL0, then this bit is RES1. This option is not permitted when SCTLR_EL1.EE is RES0.

This bit has no effect on the endianness of LDTR, LDTRSH, LDTRSH, LDTRSW, STTP and STTRH instructions executed at EL1.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNY .OWN . . .

When $HCR_EL2.E2H == 1$ and $HCR_EL2.TGE == 0$

E0E, bit [24]

IGNORED.

Otherwise:

res0

SPAN, bit [23]

When HCR_EL2.E2H == ' .nd . ``R_EL2.TGL = 1 SPAN, bit [23]

Set Privileged Ac ss Never in taking an exception to EL2.

SPAN	Meaning
060	PSTATE.PAN is set to 1 on taking an exception to EL2.
0b1	The value of PSTATE.PAN is left unchanged on taking an exception to EL2.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

When $HCR_EL2.E2H == 1$ and $HCR_EL2.TGE == 0$

SPAN, bit [23]

IGNORED.

Otherwise:

EIS, bit [22]

When FEAT_ExS is implemented:

Exception entry is a context synchronization event.

EIS	Meaning
0b0	The taking of an exception to EL2 is not a context synchronization event.
0b1	The taking of an exception to EL2 is a context synchron'

If SCTLR_EL2.EIS is set to 0b0:

- Indirect writes to ESR_EL2, FAR_EL2, SPSR_EL2 _LR_EL2, and 'PF^ _EL2 are synchronized on exception entry to EL2, so that a direct read of the _sister aft exceptio. Intry sees the indirectly written value caused by the exception entry.
- Memory transactions, including instruction f bes, find an Excertion level always use the translation resources associated with that translation $r_{\rm cume}$.
- Exception Catch debug events are sync¹ snous de 1g events.
- DCPS* and DRPS instructions are con. * sync' vents.

The following are not affected by the va. of SC R_EL2.EIS:

- Changes to the PSTATE inf nation on htry to 2.
- Behavior of accessing the sked copies of e stack pointer using the SP register name for loads, stores, and data processing instructions.
- Exit from Debug state.

The reset behavior ______ this field is:

• On a Warn, set, th' neld resets to an architecturally UNKNOWN value.

Otherwise

RES1

IESЬ, it Г

When FEA. **'ESB** is implemented:

Implicit Error Synchronization event enable.

IESB	Meaning
0b0	Disabled.
0b1	 An implicit error synchronization event is added: At each exception taken to EL2. Before the operational pseudocode of each ERET instruction executed at EL2.

When the PE is in Debug state, the effect of this field is CONSTRAINED UNPREDICTABLE, and its Effective value

might be 0 or 1 regardless of the value of the field. If the Effective value of the field is 1, then an implicit error synchronization event is added after each DCPSX instruction taken to EL2 and before each DRPS instruction executed at EL2, in addition to the other cases where it is added.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

res0

TSCXT, bit [20]

When (FEAT_CSV2_2 is implemented or FEAT_CSV2_1p2 is implemented), HCR_EL2.E2H == 1 and HCR_EL2.TGE == 1

TSCXT, bit [20]

Trap EL0 Access to the SCXTNUM_EL0 register, when EL0; _.sing Ar. h64.

TSCA	M aning
JUC	^Y .0 ac ₃ s to SCXTNUM_EL0 is not disabled
	b, mechanism.
01-	FL0 access to SCXTNUM EL0 is disabled.
	causing an exception to EL2, and the
	SCXTNUM_EL0 value is treated as 0.

The reset behavior of this f ____is:

• On a Warm reset is field resets an architecturally UNKNOWN value.

When FEAT_CS' _2 is no implemented, FEAT_CSV2_1p2 is not implemented, HCR_EL2.E2H == 1 and HCR_EL2.TGE = 1

Bit [0]

Reg ved, RES

When $" f_CSV2_2$ is implemented or FEAT_CSV2_1p2 is implemented), HCR_EL2.E2H == 1 and HCR_EL. "GE == 0

TSCXT, bit [2]

IGNORED.

Otherwise:

res0

WXN, bit [19]

Write permission implies XN (Execute-never). For the EL2 or EL2&0 translation regime, this bit can force all memory regions that are writable to be treated as XN.

WXN	Meaning
000	This control has no effect on memory access permissions.
0b1	Any region that is writable in the EL2 or EL2&0 translation regime is forced to XN for accesses from software executing at EL2.

This bit applies only when SCTLR_EL2.M bit is set.

The WXN bit is permitted to be cached in a TLB.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOW value.

nTWE, bit [18]

When HCR_EL2.E2H == 1 and HCR_EL2.TGE == 1

nTWE, bit [18]

Traps execution of WFE instructions at EL0 to L2, fro both Ex or on states.

.rwe		Meaning
	OĿ	Any attempt to execute a WFE instruction at EL0 is trapped to EL2, if the instruction would otherwise have caused the PE to enter a low-power state.
	0b1	This control does not cause any instructions to be trapped.

In AArc' 2 sta, the at opter' xecution of a conditional WFE instruction is only trapped if the instruction passes its condition condition condition conditional WFE instruction is only trapped if the instruction passes its condition conditional WFE instruction is only trapped if the instruction passes its conditional weight of the instruction of a conditional weight of the instruction passes its conditional weight of the in

Since WF^{*} of write complete at any time, even without a Wakeup event, the traps on WFE of WFI are not guarante to be taken, even if the WFE or WFI is executed when there is no Wakeup event. The only guarantee is that if the net ruction does not complete in finite time in the absence of a Wakeup event, the trap will be taken.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

When HCR_EL2.E2H == 1 and HCR_EL2.TGE == 0

nTWE, bit [18]

IGNORED.

Otherwise:

res1

Bit [17]

Reserved, RESO.

nTWI, bit [16]

When HCR_EL2.E2H == 1 and HCR_EL2.TGE == 1

nTWI, bit [16]

Traps execution of WFI instructions at EL0 to EL2, from both Execution states.

nTWI	Meaning
060	Any empt to execue a WFI instruction at EL0 trappe EL2, if the estruction would otherwise end of the PE to enter a low-end wer su
0b1	T is control does not cause any instructions to be trappe

In AArch32 state, the attempted execution of ondition of the instruction is only trapped if the instruction passes its condition code check.

Since a WFE or WFI can complete t_{i} any t_{i} , even t_{i} ithout a Wakeup event, the traps on WFE of WFI are not guaranteed to be taken, even if t_{i}^{t} WFE or W₁ is exected when there is no Wakeup event. The only guarantee is that if the instruction does not consider in finite t_{i} be in the absence of a Wakeup event, the trap will be taken.

The reset behavior of th[:] field is:

• On a Warm re ., this field resets to . architecturally UNKNOWN value.

When HCR_EL . "2H = - and $HCR_EL2.TGE = = 0$

nTWI, b'′

IGN KED.

Otherw

RES1

UCT, bit [15]

When HCR_EL2.E2H == 1 *and HCR_EL2.TGE* == 1 *UCT, bit* [15]

Traps EL0 accesses to the CTR_EL0 to EL2, from AArch64 state only.

UCT	Meaning
0b0	Accesses to the CTR_EL0 from EL0 using AArch64 are trapped to EL2.
0b1	This control does not cause any instructions to be trapped.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

When $HCR_EL2.E2H == 1$ and $HCR_EL2.TGE == 0$

UCT, bit [15]

IGNORED.

Otherwise:

res0

DZE, bit [14]

When HCR_EL2.E2H == *1 and HCR_EL2.TGE* == *1 DZE*, *bit* [14]

Traps execution of DC ZVA instructions at EL0 to EL2, from Arch64 sta. only.

If FEAT_MTE is implemented, this trap also applies to Γ GVA ar DC GZV.

Ω

/L	feani [,]
br	Any attempt to execute an instruction that this ap applies to at EL0 using AArch64 is trapped to EL2. Reading DCZID_EL0.DZP from EL0 returns 1, indicating that the instructions that this trap applies to are not supported
b1	This control does not cause any instructions to be trapped.

The reset behavior of this / id is:

• On a ^{w-rm} rest nis field jets to an architecturally UNKNOWN value.

When CR_EI .E2H = ' $.dHCR_EL2.TGE == 0$

DZE, '+ [1

IGNORED.

Otherwise:

res0

EnDB, bit [13]

When FEAT_PAuth is implemented:

Controls enabling of pointer authentication (using the APDBKey_EL1 key) of instruction addresses in the EL2 or EL2&0 translation regime.

For more information, see System register control of pointer authentication .

EnDB	Meaning
0b0	Pointer authentication (using the APDBKey_EL1 key) of data addresses is not enabled.
0b1	Pointer authentication (using the APDBKey_EL1 key) of data addresses is enabled.

This field controls the behavior of the AddPACDB and AuthDB pseudocode functions. Specifically, when the field is 1, AddPACDB returns a copy of a pointer to which a pointer authentication code has been added, and AuthDB returns an authenticated copy of a pointer. When the field is 0, both of these functions are NOP.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOW alue.

Otherwise:

res0

l, bit [12]

Instruction access Cacheability control, for access 2 and, then E^{*I*} is enabled in the current Security state and HCR_EL2.{E2H,TGE} == {1,1}, EL0.

Meaning

All instruction accesses to Normal memory from EL2 are Non-cacheable for all levels of instruction and unified cache. When EL2 is enabled in the current Security state and HCR_EL2. $\{E2H, TGE\} == \{1, 1\}, all$ instruction accesses to Normal memory from EL0 are Non-cacheable for all levels of instruction and unified cache. If SCTLR_EL2.M is 0, instruction accesses from stage 1 of the EL2 or EL2&0 translation regime are to Normal, Outer Shareable, Inner Non-cacheable, Outer Non-cacheable memory. This control has no effect on the Cacheability of 0b1 instruction access to Normal memory from EL2 and, when EL2 is enabled in the current Security state and HCR_EL2.{E2H, TGE} == $\{1, 1\}$, instruction access to Normal memory from EL0. If the value of SCTLR EL2.M is 0, instruction accesses from stage 1 of the EL2 or EL2&0 translation regime are to Normal, Outer Shareable, Inner Write-Through, Outer Write-Through memory.

This bit has no effect on the EL3 translation regime.

When EL2 is disabled in the current Security state or HCR_EL2. $\{E2H, TGE\} = \{1,1\}$, this bit has no effect on the EL1&0 translation regime.

The reset behavior of this field is:

• On a Warm reset, this field resets to Ob0.

EOS, bit [11]

When FEAT_ExS is implemented:

Exception exit is a context synchronization event.

EOS	Meaning
0b0	An exception return from EL2 is not a context synchronized event.
0b1	An exption retuil from EL2 is a context synch onization even

If SCTLR_EL2.EOS is set to 0b0:

- Memory transactions, including instruction fetches, 'rom in Exception level always use the translation resources associated with that translation regime
- Exception Catch debug events are synchror us de g even
- DCPS* and DRPS instructions are cont ... synchry lization ev ...s.

The following are not affected by the value of TV _EL2.L.

- The indirect write of the PST^TE . ¹ PC v. ¹es from SPSR_EL2 and ELR_EL2 on exception return is synchronized.
- Behavior of accessing the pixed copies of the stack pointer using the SP register name for loads, stores, and data processing instructions.
- Exit from Debug state.

The reset behavior ______ nis field is:

• On a Warn. set, th: .eld resets to an architecturally UNKNOWN value.

Otherwise⁻

RES1

EnR Y, / _[10]

When FEA. SPECRES is implemented, HCR_EL2.E2H == 1 and HCR_EL2.TGE == 1 EnRCTX, bit [10]

Enable EL0 access to the following System instructions:

- CFPRCTX, DVPRCTX and CPPRCTX instructions.
- CFP RCTX, DVP RCTX and CPP RCTX instructions.

EnRCTX	Meaning
000	EL0 access to these instructions is disabled, and these instructions are trapped to EL1.
0b1	EL0 access to these instructions is enabled.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

```
When FEAT_SPECRES is implemented, HCR_EL2.E2H == 1 and HCR_EL2.TGE == 0
```

ERRCTX, bit [10] IGNORED. Otherwise: RESO Bit [9] Reserved, RESO. SED, bit [8] When EL0 is capable of using AArch32, HCR_EL2.E2H == and HCk_''.2.TC' == 1 SED, bit [8] SETEND instruction disable. Disables SETEND in ____tions__dL0 usin_AArch32.

SF'	Meaning
יה0	SETEND instruction execution is enabled at EL0 using AArch32.
0b1	SETEND instructions are UNDEFINED at EL0 using AArch32.

If the implementation does not support mixed-endian operation at any Exception level, this bit is RES1. The reset behaviour of this celd is:

• On 2 m rese. his field sets to an architecturally UNKNOWN value.

```
Whey LO can nly use ... .h64, HCR_EL2.E2H == 1 and HCR_EL2.TGE == 1
```

Bit [0,

Reserved, R.

When EL0 is capable of using AArch32, HCR_EL2.E2H == 1 and HCR_EL2.TGE == 0

SED, bit [8]

IGNORED.

Otherwise:

res0

ITD, bit [7]

When EL0 is capable of using AArch32, HCR_EL2.E2H == 1 and HCR_EL2.TGE == 1 ITD, bit [7]

IT Disable. Disables some uses of IT instructions at EL0 using AArch32.

ITD	Meaning
0b0	All IT instruction functionality is enabled at EL0 using AArch32.
0ъ1	 Any attempt at EL0 using AArch32 to execute any of the following is UNDEFINED: All encodings of the IT instruction with hw1[3:0]!=1000. All encodings of the subsequent instruction with the following values for hw1: - 0b11xxxxxxxxxxx All 32-bit
	 tions, and the 16-bit instructions B, UDF, SVC, LDM, and STM. 0b1011xx xxxxxxxx All vstructions in
	 A contraction of the second second
2	 - 0b01001xxxxxxxxxx LDR Rd, [PC, #imm] - 0b0100x1xxx1111xxx: ADD Rdn, PC; CMP Rn, PC; MOV Rd, PC; BX
	PC; BLX PC. – 0b010001xx1xxxx111: ADD PC, Rm; CMP PC, Rm; MOV PC, Rm. This pattern also covers UNPREDICTABLE cases with BLX Pn
	These instructions are always UNDEFINED, regardless of whether they would pass or fail the condition code check that applies to them as a result of being in an IT block. It is IMPLEMENTATION DEFINED whether the IT instruction is tracted as
	 A 16-bit instruction, that can only be followed by another 16-bit instruction. The first half of a 32-bit instruction.
	UNDEFINED, either the second 16-bit instruction or the 32-bit instruction is UNDEFINED. An implementation might vary dynamically as to
	whether IT is treated as a 16-bit instruction or the first half of a 32-bit instruction.

If an instruction in an active IT block that would be disabled by this field sets this field to 1 then behavior is CONSTRAINED UNPREDICTABLE. For more information see Changes to an ITD control by an instruction in an IT block .

ITD is optional, but if it is implemented in the SCTLR_EL2 then it must also be implemented in the SCTLR_EL1, HSCTLR, and SCTLR.

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 ITD, access to this field is RAZ/WI.

When EL0 can only use AArch64, HCR_EL2.E2H == 1 and HCR_EL2.TGE == 1

Bit [0]

Reserved, RES1.

When EL0 is capable of using AArch32, HCR_EL2.E2H == 1 and HCR_EL2.TGE == 0

ITD, bit [7]

IGNORED.

Otherwise:

res0

nAA, bit [6]

When FEAT_LSE2 is implemented:

Non-aligned access. This bit controls generation of Alignma alts under ertain conditions at EL2, and, when EL2 is enabled in the current Security state and $P_{--} = 2$. [E2] TGE⁺ = {1, 1}, EL0.

The following instructions generate an Alignent fault all bytes being accessed are not within a single 16-byte quantity, aligned to 16 bytes for access:

- LDAPR, LDAPRH, LDAPUR, 1 `APUK LDAPURSH, LDAPURSW, LDAR, LDARH, LDLAR, LDLARH.
- STLLR, STLLRH, STL "LRH, STLL and STLURH

nAA	Meaning
0b0	Unaligned accesses by the specified instructions generate an Alignment fault.
0b1	Unaligned accesses by the specified instructions do not generate an Alignment fault.

The rese. havior of this field is:

• On a W. n reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

res0

CP15BEN, bit [5]

When EL0 is capable of using AArch32, HCR_EL2.E2H == 1 and HCR_EL2.TGE == 1

CP15BEN, bit [5]

System instruction memory barrier enable. Enables accesses to the DMB, DSB, and ISB System instructions in the (coproc==0b1111) encoding space from EL0:

CP15BEN	Meaning
060	EL0 using AArch32: EL0 execution of the CP15DMB, CP15DSB, and CP15ISB instructions is UNDEFINED.
0b1	EL0 using AArch32: EL0 execution of the CP15DMB, CP15DSB, and CP15ISB instructions is enabled.

CP15BEN is optional, but if it is implemented in the SCTLR_EL2 then it must also be implemented in the SCTLR_EL1, HSCTLR, and SCTLR.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOV. value.

When ELO can only use AArch64, HCR_EL2.E2H == 1 and ' CR_EL_ "GE == 1

Bit [0]

Reserved, RESO.

When EL0 is capable of using AArch32, HCF $_L2.E$. I == 1 a. $\lor _R_EL2.TGE == 0$

CP15BEN, bit [5]

IGNORED.

Otherwise:

res1

SA0, bit [4]

When HCR_EL^{\uparrow} .2H == $^{\uparrow}$.nd $HCR_EL2.TGE$ == 1:

SP Alignment checking of the for EI. When set to 1, if a load or store instruction executed at EL0 uses the SP as the base of the base of

The jet beb is:

• On Varm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

res1

SA, bit [3]

SP Alignment check enable. When set to 1, if a load or store instruction executed at EL2 uses the SP as the base address and the SP is not aligned to a 16-byte boundary, then an SP alignment fault exception is generated. For more information, see 'SP alignment checking'.

The reset behavior of this field is:

• On a Warm reset, this field resets to an architecturally UNKNOWN value.

C, bit [2]

Data access Cacheability control, for accesses at EL2 and, when EL2 is enabled in the current Security state and HCR_EL2.{E2H, TGE} == $\{1, 1\}$, EL0

	С	Meaning		
0b0		 The following are Non-cacheable for all levels of data and unified cache: Data accesses to Normal memory from EL2. When HCR_EL2.{E2H, TGE} != {1, 1}, Normal memory accesses to the EL2 translation tables. When EL2 is enabled in the current Security state and HCR_EL2.{E2H, TGE} == {1, 1}: accesses to Normal memory from 10. Normalmory accesses to the EL2&0 translation tables. 		
	0b1	 This ontrol. effect on the Cacheability of: Data access to Normal memory from EL2. Whe HCR_EL2.{E2H, TGE} != {1, 1}, N' mal memory accesses to the EL2 ranslation tables. When EL2 is enabled in the current Security state and HCR_EL2.{E2H, TGE} == {1, 1}: Data accesses to Normal memory from EL0. Normal memory accesses to the EL2&0 translation tables. 		

This bit has no eff. op the EL3 translation regime.

When EV to able the cubic of HCR_EL2. $\{E2H, TGE\} = \{1, 1\}$, this bit has no effect on the EV & 0 tran ation res

The . et be¹ ... field is:

• On Varm reset, this field resets to Ob0.

A, bit [1]

Alignment check enable. This is the enable bit for Alignment fault checking at EL2 and, when EL2 is enabled in the current Security state and HCR_EL2.{E2H, TGE} == $\{1, 1\}$, EL0.

Α	Meaning
0b0	Alignment fault checking disabled when
	executing at EL2.
	When EL2 is enabled in the current Security
	state and HCR_EL2.{E2H, TGE} == $\{1, 1\}$,
	alignment fault checking disabled when
	executing at EL0.
	Instructions that load or store one or more
	registers, other than load/store exclusive and
	load-acquire/store-release, do not check that the
	address being accessed is aligned to the size of
	the data electrics) being accessed.
0b1	Align Int fault ch. 'ring enabled when
	execing at EL2.
	When 2 is enabled 1 the current Security
	state and R_EL^2 , $E2H$, TGE == {1, 1},
	alignment fat. cking enabled when
	ev uting at ELU.
	Il instructions that load or store one or more
	gister Jave an alignment check that the
	a being accessed is aligned to the size of
	the data element(s) being accessed. If this check
	ails it causes an Alignment fault, which is taken
K	as a Data Abort exception.

Load/store exclusive and loar seq releas instructions have an alignment check regardless of the value of the A bit.

If FEAT_MOPS is in the emented, SETG^{*} tructions have an alignment check regardless of the value of the A bit. The reset behavior of this final is:

• On a Warm re. As field re is to an architecturally UNKNOWN value.

M, bi* _]

MMU ration EL2. EL2&0 stage 1 address translation.

М	Meaning
0b0	When HCR_EL2.{E2H, TGE} != {1, 1}, EL2 stage 1 address translation disabled. When HCR_EL2.{E2H, TGE} == {1, 1}, EL2&0 stage 1 address translation disabled. See the SCTLR_EL2.I field for the behavior of instruction accesses to Normal memory.
0b1	When HCR_EL2.{E2H, TGE} != {1, 1}, EL2 stage 1 address translation enabled. When HCR_EL2.{E2H, TGE} == {1, 1}, EL2&0 stage 1 address translation enabled.

The reset behavior of this field is:

• On a Warm reset, this field resets to 0b0.

Accessing SCTLR_EL2

When HCR_EL2.E2H is 1, without explicit synchronization, access from EL2 using the mnemonic SCTLR_EL2 or SCTLR_EL1 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Accesses to this register use the following encodings in the System register encoding space:

MRS <Xt>, SCTLR_EL2



MRS <Xt>, SCTLR_EL1

орО	op1	CRn	CRm	op2
0b11	0b000	0b0001	0b0000	0b000

```
1 if PSTATE.EL == ELO then
```

```
2 UNDEFINED;
```

```
3 elsif PSTATE.EL == EL1 then
```

```
4 if EL2Enabled() && HCR_EL2.TRVM == '1' then
```
Chapter E3. System registers affected by SME E3.2. Changes to existing System registers

```
5
           AArch64.SystemAccessTrap(EL2, 0x18);
6
       elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn
            ← == '1') && HFGRTR_EL2.SCTLR_EL1 == '1' then
7
           AArch64.SystemAccessTrap(EL2, 0x18);
8
       elsif EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '111' then
9
           X[t, 64] = NVMem[0x110];
10
       else
11
           X[t, 64] = SCTLR\_EL1;
   elsif PSTATE.EL == EL2 then
12
       if HCR_EL2.E2H == '1' then
13
14
           X[t, 64] = SCTLR\_EL2;
15
       else
16
           X[t, 64] = SCTLR\_EL1;
17
   elsif PSTATE.EL == EL3 then
18
     X[t, 64] = SCTLR\_EL1;
```

MSR SCTLR_EL1, <Xt>

op0	op1	CRn	CRm	op2	
0b11	0b000	r J001	010	0b000	

```
if PSTATE.EL == EL0 then
 1
 2
          UNDEFINED;
 3
     elsif PSTATE.EL == EL1 then
          if EL2Enabled() && HCR_EL2.TVM ==
 4
                                                                t٢
          AArch64.SystemAccessTrap 12, 0x

elsif EL2Enabled() && IsFeptur opleme od(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn

→== '1') && HFGWTR 7 ...SCTL 7L1 == 1' then

AArch64.SystemAccer rap(EL2, 18);
 5
 6
 7
 8
           elsif EL2Enabled() &/ . EL2.<NV2, N NV> == '111' then
 9
                NVMem[0x110] = t,
10
           else
     SCTLR_EL1 = .[t, 64];
elsif PSTATE.EL = EL2 then
11
12
          if HCR_EL2 .H == '1 then
    SCTLR_ 2 = X' 64];
13
14
15
           else
     S P EL1 X[t, 647
elsif P ATE L == L th
16
17
18
              .LR_EL1
                           X[t,
```

E3.2.18 EDDEVID1, External Debug Device ID register 1

The EDDEVID1 characteristics are:

Purpose

Provides extra information for external debuggers about features of the debug implementation.

Attributes

EDDEVID1 is a 32-bit register.

Field descriptions

The EDDEVID1 bit assignments are:

	131		8 7 4 3 0		
	RES0		HSi		
			LPCSROffset		
Bits [31:8]	1				
Reserved, R	res0.				
HSR, bits	[7:4]				
Indicates support for the External Debug Halt tus legister, ASR. Defined values are:					
		H.	Meaning		
		0000đ	EDHSR not implemented, and the PE follows behaviors consistent with all of the EDHSR fields having a zero value.		
		0b0001	EDHSR implemented.		

Wh FEAT_F ougv8p2 is not implemented, the only permitted value is 0b0000.

PCSh、* ,, bits [3:0]

Indicates the ^{*}set applied to PC samples returned by reads of EDPCSR. Permitted values of this field in Armv8 are:

PCSROffset	Meaning
0b0000	EDPCSR not implemented.
0b0010	EDPCSR implemented, and samples have no offset applied and do not sample the instruction set state in AArch32 state.

When FEAT_PCSRv8p2 is implemented, the only permitted value is 0b0000.

FEAT_PCSRv8p2 implements the PC Sample-based Profiling Extension in the Performance Monitors register

space, as indicated by the value of PMU.PMDEVID.PCSample.

Accessing EDDEVID1

Accesses to this register use the following encodings in the external debug interface:

EDDEVID1 can be accessed through the external debug interface:

Component	Offset	Instance
Debug	0xFC4	EDDEVID1

This interface is accessible as follows:

- When FEAT_DoPD is not implemented or IsCorePowered() actors to this region r is RO.
- Otherwise access to this register returns an ERROR.

Chapter E4 Glossary terms

Effective Non-streaming SVE vector ungth

The Non-streaming S'_{-} vector length in t_s at the current Exception level, is an implementation-supported power of two up to the M_{-} amum in plemented Non-streaming SVE vector length, further constrained by ZCR_ELx at the current and L_{-} ber Exc - cion levels.

Effective Streaming SVF vector ugth

The S^t aming /E vecu ¹/₂ th in bits at the current Exception level is an implementation-supported power of two om 128 t to the Maximum implemented Streaming SVE vector length, further constrained by SMCR_ELx at the ¹/₁ re⁻ and m₂. Exception levels.

Effective SVE vector , 9th

The vector length in bits that applies to the execution of SVE instructions at the current Exception level is the Effective Streaming SVE vector length when the PE is in Streaming SVE mode, otherwise it is the Effective Non-streaming SVE vector length.

Illegal

Describes an implemented 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.

Legal

Describes an implemented instruction that can be executed by a PE when <code>PSTATE.SM</code> and <code>PSTATE.ZA</code> are in the required state, unless its execution at the current Exception level is prevented by a configurable trap or enable.

Maximum implemented Non-streaming SVE vector length

The maximum Non-streaming SVE vector length in bits supported by the implementation.

Maximum implemented Streaming SVE vector length

The maximum Streaming SVE vector length in bits supported by the implementation.

NSVL

Effective Non-streaming SVE vector length.

Scalable Matrix Extension

Defines architectural state capable of holding two-dimensional matrix tiles, and a Streaming SVE mode which supports execution of SVE2 instructions with a vector length that matches the tile width, along with instructions that accumulate the outer product of two vectors into a tile, as well as load, store, and move instructions that transfer a vector to or from from a tile row or column.

Scalable Matrix Extension version 2

Extends the Scalable Matrix Extension by adding data-processing in. vctions with n lti-vector operands and a multi-vector predication mechanism, a lookup table feature, a bin y outer voluct instruction, and a range prefetch hint.

SMCU

Streaming Mode Compute Unit.

SME

Scalable Matrix Extension.

SME2

Scalable Matrix Extension versior

Streaming execution

Execution of instructions by a PE w. that PE is in Streaming SVE mode.

Streaming Mode Compute Unit

Where more that one PE states resources for Streaming execution of SVE and SME instructions, those shared resources are called Staming M ¹e Compute Unit (SMCU).

Streaming SVE mod

An ecution r de that su_F ports a substantial subset of the SVE2 instruction set and architectural state with a vector engritering the width of SME tiles, which may be different from the vector length when the PE is not in S. ming SVE.

Streaming SVE register hte

The registers Z0-Z31, P0-P15, and FFR that are accessed by SVE and SME instructions when the PE is in Streaming SVE mode.

SVL

Effective Streaming SVE vector length.

٧L

Effective SVE vector length.

ZA array

A two-dimensional array of $[SVL_B \times SVL_B]$ bytes contained within the ZA storage.

ZA array vector

A vector that is SVL bits in size within the ZA array.

ZA storage

The architectural register state added by SME that is capable of holding two-dimensional matrix tiles.

ZA tile

A square, two-dimensional sub-array of the ZA array.

ZA tile slice

A one-dimensional set of horizontally or vertically contiguous elements within a ZA tile.

ZT0 register

The 512-bit architectural register added by SME2 that consists of up to sixteen 32-bit table entries, each containing an 8-bit, 16-bit, or 32-bit element.

