AI Compute Extensions (ACE) Specification



x86 Ecosystem Advisory Group

Specification Release

Date: 10/22/2026
Version: 1.16.3.1

Notices & Disclaimers

This document contains information on products in the design phase of development. The information in this document, and all product plans and roadmaps are subject to change without notice. Do not finalize a design with this information. The products described may contain design defects or errors known as errata which may cause the product to deviate from published specifications.

Code names are used by Intel and AMD to identify products, technologies, or services that are in development and not publicly available. These are not “commercial” names and not intended to function as trademarks.

Intel or AMD technologies may require enabled hardware, software or service activation. Performance varies by use, configuration, and other factors. Your costs and results may vary. No product or component can be absolutely secure.

This document and the information contained in it are provided “as is”. Intel and AMD disclaim all express and implied warranties, including without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement, as well as any warranty arising from course of performance, course of dealing, or usage in trade. You use this document and the information contained in it at your own risk. Intel and AMD are not required to maintain, update, or support this document. Neither Intel nor AMD will be liable to you under any legal theory for any losses or damages in connection with this document, the information contained in it, or your use of either.

Intel and AMD each retain ownership of their respective content and intellectual property reflected in this document and may make changes to it at any time. Intel and AMD’s joint development and distribution of this document does not alter any existing intellectual property rights or licenses of Intel or AMD.

No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document, with the sole exception that a) you may publish an unmodified copy, provided that the document is published verbatim in its entirety and all notices, disclaimers, copyright statements, and legal notices are retained in full and unaltered. You may create software implementations based on this document and in compliance with the foregoing that are intended to execute on x86 product(s) referenced in this document. No rights are granted to create modifications or derivatives of this document.

You may not use or facilitate the use of this document in connection with any infringement or other legal analysis concerning Intel or AMD products described herein. You agree to grant Intel and AMD a non-exclusive, royalty-free license to any patent claim thereafter drafted which includes subject matter disclosed herein.

If you give Intel or AMD any comments or suggestions related to this document or the information contained in it, Intel and AMD can use them in any way and disclose them to anyone, without payment or other obligations to you. You represent and warrant that you own, or have sufficient rights from the owner of, any such comments or suggestions, and the intellectual property rights in them, to grant the above permission.

This document and the information contained herein may be subject to U.S. and non U.S. export control and sanctions laws and regulations, including the U.S. Export Administration Regulations (EAR) and applicable economic sanctions laws. You are responsible for complying with all such laws and regulations, including obtaining any required licenses or approvals prior to export, re-export, transfer, or release of this information to foreign persons or destinations.

© 2026 Intel Corporation and Advanced Micro Devices, Inc. Intel and other Intel marks are trademarks of Intel Corporation or its subsidiaries. AMD and other AMD marks are trademarks of Advanced Micro Devices, Inc. or its subsidiaries. Other names and brands may be claimed as the property of others.

Table of Contents

1 Revision History 6
2 Introduction 8
2.1 AI Compute Extensions (ACE) Overview 8
2.2 Architecture Components & Baseline 8
2.3 Data Formats 8
2.4 OCP MX Formats – Numeric Definition 8
2.5 Assembler Mnemonic Naming 9
2.6 Rounding 10
3 CPUID Detection 11
3.1 Detection Algorithm 11
4 Instruction Summary 12
4.1 AVX-VNNI-INT8 and AVX-VNNI-INT16 12
4.2 AVX10.2 Subset (AVX10_V1_AUX) 12
4.3 OCP Format Conversions (AVX10_V2_AUX) 12
4.4 ACE Tile Instructions (ACE v1) 12
5 Exception Classes 13
5.1 Exception Classes by Instruction 13
5.2 Exceptions Type E2 14
5.3 Exceptions Type E4 and E4NF of EVEX-Encoded Instructions 15
5.4 Exceptions Type E6 17
5.5 Exceptions Type E7NM 18
5.6 Exceptions Type AMX 19
5.7 Exceptions Type ACE 21
5.8 #UD Equations for EVEX Encoded Instructions 24
5.9 Implementation Specific Alignment Checking for 16, 32 and 64 Byte Accesses 25
6 Instruction Encodings 26
6.1 Instructions Enumerated under AVX10_V1_AUX 26
6.2 Instructions Enumerated under AVX10_V2_AUX 28
6.3 Instructions Enumerated under ACE v1 (Tile) 30
7 AVX-VNNI-INT8 and AVX-VNNI-INT16 32
8 AVX10.2 Subset 33
8.1 Introduction 33
8.2 Convert from FP16 to FP8 33
8.3 Convert FP32 Pair to FP16 35
8.4 Convert FP16 to FP8 with Bias 36

8.5 Convert FP8 E4M3 to FP16 38
8.6 Multiply and Add Bytes (EVEX) 38
8.7 Multiply and Add Words (EVEX) 40
9 OCP Format Conversion Functions 43
9.1 Introduction 43
9.2 Convert from FP32 to FP8 43
9.3 Convert from FP8 to FP32 46
9.4 Convert from FP8 to FP4 47
9.5 Convert from FP4 to FP8 49
9.6 Convert from FP8 to FP6 49
9.7 Convert from FP6 to FP8 50
9.8 Down Convert DWord to Byte with Symmetric Signed Saturation 51
9.9 Unpack to Byte (VUNPACKB) 53
10 AI Compute Extensions (ACE) 55
10.1 Introduction to the AI Compute Extensions 55
10.2 Processor State 55
10.3 Matrix Multiplication 57
10.4 Outer Product Operation 57
10.5 Block Scale Support 63
10.6 Operations Introduced with the ACE Extensions 65
11 Tile Management Instructions 66
11.1 TILEZERO – Zero Tile 66
11.2 LDTILECFG – Load Tile Configuration 66
11.3 STTILECFG – Store Tile Configuration 68
11.4 TILERELEASE – Release Tile 69
12 Tile Data Movement 70
12.1 Tile Move Operations – Overview 70
12.2 Tile Move Row (TILEMOVROW) 70
12.3 Tile Move Column (TILEMOVCOL) 71
12.4 Tile Move Row and Convert INT32 to FP32 (TCVTROWD2PS) 72
12.5 Tile Move Row and Convert FP32 to BF16 (TCVTROWPS2BF16[H,L]) 73
12.6 Tile Move Row and Convert FP32 to FP16 (TCVTROWPS2PH[H,L]) 75
13 Block Scale Register Operations 77
13.1 BSRINIT – Block Scale Register Init 77
13.2 BSRMOVF – Block Scale Register Move Full 77
13.3 BSRMOVH / BSRMOVL – Block Scale Register Move Half 78
14 Tile Outer Product Instructions 80

14.1 Tile Outer Product MX FP8 Rank 4 (TOP4MX[B|H][B|H]F8PS) 80
14.2 Tile Outer Product MX INT8 Rank 4 (TOP4MXBSSPS) 82
14.3 Tile Outer Product BF16 Rank 2 (TOP2BF16PS) 84
14.4 Tile Outer Product Byte Rank 4 (TOP4B[U|S][U|S]D) 86
15 Application/System Programmer Model 89
15.1 Background 89
15.2 AMX State 89
15.3 Feature Flags and Supported Instructions 90
15.4 Changes to XSAVE Feature Set 91
15.5 CPUID State Enumeration (Detailed) 94
16 Pseudocode Helper Functions 98
16.1 Conversions from FP32 98
16.2 Conversions from FP16 101
16.3 Conversions from FP8 104
16.4 Conversions from FP6 and FP4 107
16.5 AMX and ACE Functions 108
Appendix A – CPUID Summary 113
Appendix B – References 114
Appendix C — APX Interaction 115

1 Revision History

Revision	Date	Notes
1.12	March 2026	Initial merge of the component specifications: • AI Compute Extensions, v1.11 • Instruction Set Extensions for Reduced Precision Format and ML Workloads, v1.11 • Instruction Set Extensions for Reduced Precision Format Conversion, v1.2 • Split ACE-E1 exception class into ACE-E1, ACE-E6.
1.13	April 2026	Restated outer FP8 and MX INT8 outer products to apply scaling in the precise, not FP, domain. Floating-point conversion clarifications including FP32 to FP8 E4M3 RTO handling and fixed two reversions in FP6 converts.
1.14	April 2026	Migrated to new document build flow, enhancements to pseudocode. • Clarified TOP2BF16PS pseudocode, rationalised outer product helper function usage. • VCVTHF82PH: exception class corrected to E2 in summary table (was incorrectly listed as E4). • TOP4MXBSSPS: combined 2^(−12) MX INT8 implicit product bias now stated explicitly in prose.
1.14.1	April 2026	• ACE-E3 / ACE-E5: added missing VVVV != 0b1111 condition.
1.15	May 2026	Specification corrections and intrinsics additions. • §2.4.1: fixed MX INT8 format table column layout. • §9.9.6 (VUNPACKB): added ACE_UNPACKB_SIZE / ACE_UNPACKB_START / ACE_UNPACKB_SEXT macro definitions to C/C++ intrinsic section. • Tile instruction forms tables (TILEZERO, LDTILECFG, STTILECFG, TILEMOVROW, TILEMOVCOL etc.): corrected spurious blank rows caused by VEX.128 width being misclassified as a vector-width variant.
1.16	June 2026	• Correct imm8 BSR scale field positions: A_SCALE=imm8[1:0], B_SCALE=imm8[4:3]. • Correct TILEMOVCOL to dword-granularity column operations. • Separate CR0.TS (#NM) from OSXSAVE/XCR0 (#UD) in enabling section. • Detail BSR group mapping. • VCVT{B|H}F82BF4S, VCVTBF82BF6S, VCVTHF82HF6S: change DAZ=1 to DAZ=0 for FP8→FP4 and FP8→FP6 converts (aligns with exponent-width convention; no numerical change).
1.16.1	July 2026	• VCVTBIASPS2BF8, VCVTBIASPS2HF8: added missing top-level pseudocode. • VCVT{B|H}F82BF4S, VPMOVSSDB: corrected operand encoding — destination is ModRM:r/m, source is ModRM:reg. Pseudocode fixes: zeros MAXVL-1.. only for register destination variants. Zero masking not available for memory destination variants. • Programmer model: added Palette 2 fixed-parameter table and XSAVE state component summary. • TOP4MXBSSPS: expanded imm8 bitfield diagram and group table inline.
1.16.2	July 2026	• Rounding mode identifier renamed from RTNE to RNE throughout spec, pseudocode, and intrinsics to align with the conventional x86/IEEE term. • XSAVE enumeration: TILECFG ECX{1} (COMP_64B_ALIGNED) corrected from 0 to 1. • XSAVE enumeration: ACE_VSN pinned as 1 for ACE v1 implementations.

Revision	Date	Notes
		• ACE detection algorithm: CR4.OSXSAVE condition listed before XCR0 checks. • Programmer model: AMX feature flag prescription in CPUID 07H restated with explicit values; AMX-TILE: 1, AMX-INT8/BF16/FP16/ COMPLEX: 0. • ACE-E2 (BSRMOVF), ACE-E3 (BSRMOVH/BSRMOVL): EVEX.R and EVEX.R' conditions added to explicitly restrict BSR register extension to 3-bit space, independent of how many BSRs are defined. • ACE-E5 (BSRINIT): ModRM notation corrected; REX prefix restriction preserves 3-bit BSR index space consistent with BSR move instructions. • VCVTBIASPH2BF8/HF8: instruction table descriptions updated to note "Bias rounding term is Operand 2", consistent with VCVTBIASPS2* equivalents and Word spec S8.4. • Bias rounding converts: note added clarifying that the bias term is Operand 2. • Operand encoding tables: form table redesigned to show Form and individual operand columns. • BSRMOVH/BSRMOVL: combined into a single operand encoding table. • Operand sections added to TCVTROWPS2BF16[H,L] and TCVTROWPS2PH[H,L].
1.16.3	October 2026	• LDTILECFG/STTILECFG/TILERELEASE pseudocode refactored: palette 2 populates tilecfg.t[N] fields (rows=16, colsb=64, valid=1); CPUID-level palette check added (`cpu_supports_palette`); palette 1/2 validation moved to dedicated helper functions; `zero_all_tile_data` uses XSAVE-size-based zeroing. • `bsr_init()` guarded by `if XCR0[SCALEDATA]` in LDTILECFG and TILERELEASE. • §15.4.4 (XFD): LDTILECFG/TILERELEASE do not generate #NM for SCALEDATA when IA32_XFD[18]=1. STTILECFG does not access SCALEDATA. • §15.4.5 (XRSTOR): SCALEDATA generates #NM when XSTATE_BV[20]=1 and IA32_XFD[18]=1; initializes to 0x7F when XSTATE_BV[20]=0. • §15.4.5 (XSAVE): SCALEDATA saved as initial state (0x7F, XSTATE_BV[20]=0) when IA32_XFD[18]=1. • TILEMOVCOL write forms: REG(w) corrected to REG(r/w). • Tile instructions: OPND3 tables split to per-form values (VVVV(r) on r32 forms, imm8 on imm8 forms). • §3 CPUID Detection: replaced duplicate feature flag table with summary and cross-reference to §15.5. Added AMX-TILE = 1 to detection algorithm. • VCVTBF82BF6S, VCVTHF82HF6S: operand encoding corrected — destination is ModRM:r/m, source is ModRM:reg (narrowing-convert family convention, consistent with FP8→FP4). • §2.4.1 MX FP8 format table: FP8 E5M2 Infinities corrected from N/A to S.11111.00 = ±∞. • Appendix B: OCP MX Specification [2] publication date added (September 2023).
1.16.3.1	October 2026	• Exception class alignment with ACE v2 and APX specifications. • BSRINIT: operandless BSR-state initialization. ModRM byte is fixed C0; no explicit BSR register operand. Initializes all architectural BSR state.

2 Introduction

2.1 AI Compute Extensions (ACE) Overview

This document defines x86 extensions for accelerating computation tasks, initially focusing on matrix multiplication kernels and reduced precision data formats important to ML workloads.

The ACE extensions define matrix multiplication primitives that augment AVX and scalar code with new capabilities, adding:

- ACE register state, including tile and block scale registers
- Data processing operations that consume AVX register input and operate on tile register state
- Data move operations to move data between ACE register state and AVX registers
- State and operations for system management

ACE provides tight integration between AVX vectors and ACE tile registers, combining high compute density tile processing operations with the comprehensive data processing features of AVX.

In addition to matrix acceleration, a number of dedicated format convert operations are provided under the AVX10 framework.

2.2 Architecture Components & Baseline

Implementations compliant with this specification will support an architectural baseline of at least AVX10.1.

2.3 Data Formats

The extensions described in this document include support for several data formats. This may include native format support for operations such as matrix multiplication, scaling support for OCP MX-style operations, accumulation format and format conversion support between different formats. Support for additional data formats may be introduced in the future.

Format	Description	Notes
INT8	8-bit integer	
INT32	32-bit integer	
FP32	SE8M23	As defined by IEEE-754
BF16	SE8M7	
FP16	SE5M10	
E8M0	8-bit unsigned exponent	Used for power-of-two block scale formats
FP8	8-bit floating point	Defined in OCP 8-bit Floating Point Specification (OFP8) [1]. Also refer to OCP Microscaling Formats (MX) Specification [2].
MX FP8	8-bit floating point formats (SE5M2, SE4M3)	
MX FP6	6-bit floating point formats (SE3M2, SE2M3)	
MX FP4	4-bit floating point format (SE2M1)	
MX INT8	8-bit fixed point fractional format	

2.4 OCP MX Formats – Numeric Definition

The ACE extensions provide data format support for formats defined in the OCP Microscaling Formats (MX) Specification [2].

2.4.1 MX FP8 & MX INT8 Formats

The MX FP8 and MX INT8 formats are natively supported in ACE matrix multiplication operations; additionally there are dedicated convert operations to and from wider formats.

	FP8 E4M3	FP8 E5M2	MXINT8
Exponent bias	7	15	N/A

	FP8 E4M3	FP8 E5M2	MXINT8
Implicit scale	N/A	N/A	2^{-6}
Infinities	N/A	$S.11111.00_2 = \pm\infty$	N/A
NaN	$S.1111.111_2$	$S.11111.\{01, 10, 11\}_2$	N/A
Zeros	$S.0000.000_2$	$S.00000.00_2$	$0\ 0.000000_2$
Max normal number	$S.1111.110_2 = \pm 448$	$S.11110.11_2 = \pm 57{,}344$	$0\ 1.111111_2 = +1\ 63/64$ $1\ 0.000001_2 = -1\ 63/64$
Min normal number	$S.0001.000_2 = \pm 2^{-6}$	$S.00001.00_2 = \pm 2^{-14}$	$0\ 0.000001_2 = +1/64$ $1\ 1.111111_2 = -1/64$
Max subnormal number	$S.0000.111_2 = \pm 0.875 \cdot 2^{-6}$	$S.00000.11_2 = \pm 0.75 \cdot 2^{-14}$	N/A
Min subnormal number	$S.0000.001_2 = \pm 2^{-9}$	$S.00000.01_2 = \pm 2^{-16}$	N/A

2.4.2 FP4 & FP6 Formats

The MX FP4 and MX FP6 formats have dedicated convert operations to and from FP8 formats.

	MX FP4 E2M1	MX FP6 E2M3	MX FP6 E3M2
Exponent bias	1	1	3
Infinities	N/A	N/A	N/A
NaN	N/A	N/A	N/A
Zeros	$S.00.0_2$	$S.00.000_2$	$S.000.00_2$
Max normal number	$S.11.1_2 = \pm 6.0$	$S.11.111_2 = \pm 7.5$	$S.111.11_2 = \pm 28.0$
Min normal number	$S.01.0_2 = \pm 1.0$	$S.01.000_2 = \pm 1.0$	$S.001.00_2 = \pm 0.25$
Max subnormal number	$S.00.1_2 = \pm 0.5$	$S.00.111_2 = \pm 0.875$	$S.000.11_2 = \pm 0.1875$
Min subnormal number	$S.00.1_2 = \pm 0.5$	$S.00.001_2 = \pm 0.125$	$S.000.01_2 = \pm 0.0625$

2.4.3 E8M0 Format

The power-of-two E8M0 block scale format is supported natively by ACE matrix multiplication operations.

	E8M0
Exponent bias	127
Supported exponent range	−127 to 127
Infinities	N/A
NaN	1111_1111_2
Zeros	N/A
Max normal number	$1111_1110_2 = 2^{127}$
Min normal number	$0000_0000_2 = 2^{-127}$
Max subnormal	N/A
Min subnormal	N/A

2.5 Assembler Mnemonic Naming

ACE instructions follow mnemonic naming conventions consistent with existing x86 AVX instructions and Intel AMX extensions. Outer product mnemonics use the `TOP` prefix (Tile Outer Product). Block scale register operations use the `BSR` prefix.

The following conventions are used in assembler naming for brevity:

Format	Abbreviated Name	Notes
FP8 E5M2	BF8	BF = Brain Float
FP8 E4M3	HF8	HF = Half Float
FP6 E3M2	BF6	
FP6 E2M3	HF6	
FP4 E2M1	BF4	
MXINT8	MXB	MXB = Microscaling Byte

2.6 Rounding

Several data format convert operations are documented in this specification. For floating-point format conversion, a variety of rounding modes may be supported dependent on the specific convert operation.

2.6.1 Round Nearest Ties Even

This implements IEEE roundTiesToEven (RNE).

2.6.2 Round to Odd

Round to odd is useful for avoiding double rounding errors in repeated downsize conversion.

2.6.3 Bias Rounding

Bias rounding in the context of format conversion allows software to introduce a bias to the rounding function prior to conversion to the target data format. Bias rounding facilitates stochastic rounding when the bias term used is generated from a random source. Other rounding behaviors are possible by selecting specific bias rounding values.

3 CPUID Detection

ACE builds upon AVX10 and the AMX accelerator framework architecture. ACE v1 requires support for AVX10, AMX, and ACE-specific CPUID feature flags. The complete CPUID enumeration tables — including feature flags, AMX tile information leaves, XSAVE state components, and palette parameters — are documented in Section 15.5.

The key feature flags are:

- `AVX10` (EAX:07H, ECX:1, EDX[19]) and `AVX10_VSN` >= 1 with `AVX10_V1_AUX` (or `AVX10_VSN` >= 2)
- `AVX10_V2_AUX` (EAX:24H, ECX:1, ECX[3])
- `AMX-TILE` (EAX:07H, ECX:0, EDX[24])
- `ACE` (EAX:07H, ECX:1, ECX[11])
- `ACE_VSN` >= 1 (EAX:1DH, ECX:2, EAX[7:0])

3.1 Detection Algorithm

To detect full ACE v1 support, software should verify:

1. (`AVX10.1` and `AVX10_V1_AUX`) or `AVX10.2`
2. `AVX10_V2_AUX`
3. `AMX-TILE` = 1
4. `ACE`
5. `ACE_VSN` >= 1
6. `CR4.OSXSAVE = 1`
7. `XCR0[20,18:17] = 0b111` (XSAVE state enabled for tile + BSR)
8. `XCR0[7:5] = 0b111` (AVX-512 state)

4 Instruction Summary

The ACE v1 specification defines instructions across several feature groups:

4.1 AVX-VNNI-INT8 and AVX-VNNI-INT16

Multiply-and-add instructions for integer VNNI workloads (VEX-encoded). Refer to Intel ISA Extensions reference for detailed operation.

- VPDPBSSD, VPDPBSSDS, VPDPBSUD, VPDPBSUDS, VPDPBUUD, VPDPBUUDS
- VPDPWSUD, VPDPWSUDS, VPDPWUSD, VPDPWUSDS, VPDPWUUD, VPDPWUUDS

4.2 AVX10.2 Subset (AVX10_V1_AUX)

FP16-to-FP8 conversions and EVEX VNNI forms:

- VCVTPH2BF8, VCVTPH2BF8S, VCVTPH2HF8, VCVTPH2HF8S
- VCVT2PH2BF8, VCVT2PH2BF8S, VCVT2PH2HF8, VCVT2PH2HF8S
- VCVT2PS2PHX
- VCVTBIASPH2BF8, VCVTBIASPH2BF8S, VCVTBIASPH2HF8, VCVTBIASPH2HF8S
- VCVTHF82PH
- VPDPBSSD, VPDPBSSDS, VPDPBSUD, VPDPBSUDS, VPDPBUUD, VPDPBUUDS (EVEX)
- VPDPWSUD, VPDPWSUDS, VPDPWUSD, VPDPWUSDS, VPDPWUUD, VPDPWUUDS (EVEX)

4.3 OCP Format Conversions (AVX10_V2_AUX)

FP32-FP8, FP8-FP4, FP8-FP6 conversions and utility instructions:

- VCVTPS2BF8, VCVTPS2BF8S, VCVTPS2HF8, VCVTPS2HF8S
- VCVTROPS2HF8, VCVTROPS2HF8S
- VCVTBIASPS2BF8, VCVTBIASPS2BF8S, VCVTBIASPS2HF8, VCVTBIASPS2HF8S
- VCVTBF82PS, VCVTHF82PS
- VCVTBF82BF4S, VCVTHF82BF4S, VCVTBF42HF8
- VCVTBF82BF6S, VCVTHF82HF6S, VCVTBF62HF8, VCVTHF62HF8
- VPMOVSSDB
- VUNPACKB

4.4 ACE Tile Instructions (ACE v1)

Tile management, data movement, and outer product operations:

- TILEZERO, LDTILECFG, STTILECFG, TILERELEASE
- TILEMOVROW, TILEMOVCOL
- TCVTROWD2PS, TCVTROWPS2BF16H, TCVTROWPS2BF16L, TCVTROWPS2PHH, TCVTROWPS2PHL
- BSRINIT, BSRMOVF, BSRMOVH, BSRMOVL
- TOP4MXBF8PS, TOP4MXBHF8PS, TOP4MXHBF8PS, TOP4MXHF8PS
- TOP4MXBSSPS
- TOP2BF16PS
- TOP4BSSD, TOP4BSUD, TOP4BUSD, TOP4BUUD

5 Exception Classes

5.1 Exception Classes by Instruction

Class	Description
E2	Convert FP32 pair to FP16 Convert FP8 E4M3 to FP16
E4	Convert from FP32 Single Precision to FP8 (E4M3/E5M2) Convert from FP8 (E4M3/E5M2) to FP32 Single Precision Multiply and Add Unsigned and Signed Bytes with and without Saturation (EVEX encoded versions) Multiply and Add Unsigned and Signed Words with and without Saturation (EVEX encoded versions)
E4NF	Convert from FP8 (E4M3/E5M2) to FP4 (E2M1) Convert from FP4 (E2M1) to FP8 (E4M3) Convert from FP16 to FP8 Convert FP16 to FP8 with Bias VUNPACKB: Unpack to byte
E6	Down Convert DWord to Byte with Symmetric Signed Saturation
E7NM	Convert from FP8 (E4M3/E5M2) to FP6 (E2M3/E3M2) Convert from FP6 (E2M3/E3M2) to FP8 (E4M3)
AMX-E1	LDTILECFG (VEX)
AMX-E2	STTILECFG (VEX)
AMX-E1-EVEX	LDTILECFG (EVEX/APX)
AMX-E2-EVEX	STTILECFG (EVEX/APX)
AMX-E5	TILEZERO
AMX-E6	TILERELEASE
ACE-E1	Tile Move Row / Tile Move Col (Immediate form)
ACE-E2	BSR MOVE FULL
ACE-E3	BSR MOVE HALF
ACE-E4	TOP with/without OCP MX SCALE
ACE-E5	BSR Init
ACE-E6	Tile Move Row / Tile Move Col (GPR form)

5.2 Exceptions Type E2

EVEX-encoded vector instructions with arithmetic semantics follow exception class E2.

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			If EVEX prefix present.
	X	X	X	X	If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0.
			X	X	If CR4.OSXSAVE[18]=0. If any one of the following conditions applies: – State requirement not met. – Opcode independent #UD conditions. – Opcode dependent #UD conditions.
	X	X	X	X	If preceded by a LOCK prefix (F0H).
			X	X	If any REX, F2, F3 or 66 prefixes precede a EVEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is ‘0’.
Device Not Available, #NM	X	X	X	X	If CR0.TS[3]=1.
Stack, #SS(0)			X		If fault suppression not set, and an illegal address in the SS segment.
				X	If fault suppression not set, and a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)			X		If fault suppression not set, and an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If fault suppression not set, and the memory address is in a non-canonical form.
	X	X			If fault suppression not set, and any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault, #PF(fault-code)		X	X	X	If fault suppression not set, and a page fault.
Alignment Check, #AC(0)		X	X	X	For 2, 4, or 8 byte memory access if alignment checking is enabled and an unaligned memory access is made while the current privilege level is 3. For 16, 32, or 64 byte memory accesses if alignment checking is enabled and an unaligned memory access is made while the current privilege level is 3. See Section 5.9.
SIMD Floating-Point Exception, #XM	X	X	X	X	If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1.

5.3 Exceptions Type E4 and E4NF of EVEX-Encoded Instructions

EVEX-encoded vector instructions that cause no SIMD FP exceptions and support memory fault suppression follow exception class E4.

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			If EVEX prefix present.
			X	X	If CR4.OSXSAVE[18]=0. If any one of the following conditions applies: – State requirement not met. – Opcode independent #UD conditions. – Opcode dependent #UD conditions.
	X	X	X	X	If preceded by a LOCK prefix (F0H).
			X	X	If any REX, F2, F3 or 66 prefixes precede a EVEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is ‘0’.
Device Not Available, #NM	X	X	X	X	If CR0.TS[3]=1.
Stack, #SS(0)			X		If fault suppression not set, and an illegal address in the SS segment.
				X	If fault suppression not set, and a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)			X		If fault suppression not set, and an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If fault suppression not set, and the memory address is in a non-canonical form.
	X	X			If fault suppression not set, and any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault, #PF(fault-code)		X	X	X	If fault suppression not set, and a page fault.
Alignment Check, #AC(0)		X	X	X	For 2, 4, or 8 byte memory access if alignment checking is enabled and an unaligned memory access is made while the current privilege level is 3. For 16, 32, or 64 byte memory accesses if alignment checking is enabled and an unaligned memory access is made while the current privilege level is 3. See Section 5.9.

EVEX-encoded vector instructions that do not cause SIMD FP exceptions nor support memory fault suppression follow exception class E4NF.

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			If EVEX prefix present.
			X	X	If CR4.OSXSAVE[18]=0. If any one of the following conditions applies: – State requirement not met. – Opcode independent #UD conditions. – Opcode dependent #UD conditions.
	X	X	X	X	If preceded by a LOCK prefix (F0H).
			X	X	If any REX, F2, F3 or 66 prefixes precede a EVEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is ‘0’.
Device Not Available, #NM	X	X	X	X	If CR0.TS[3]=1.
Stack, #SS(0)			X		For an illegal address in the SS segment.
				X	If a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)			X		If an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If the memory address is in a non-canonical form.
	X	X			If any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault, #PF(fault-code)		X	X	X	For a page fault.
Alignment Check, #AC(0)		X	X	X	For 16, 32, or 64 byte memory accesses if alignment checking is enabled and an unaligned memory access is made while the current privilege level is 3. See Section 5.9.

5.4 Exceptions Type E6

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			If EVEX prefix present.
			X	X	If CR4.OSXSAVE[18]=0. If any one of the following conditions applies: – State requirement not met. – Opcode independent #UD conditions. – Opcode dependent #UD conditions.
			X	X	If preceded by a LOCK prefix (F0H).
			X	X	If any REX, F2, F3 or 66 prefixes precede a EVEX prefix.
			X	X	If any corresponding CPUID feature flag is ‘0’.
Device Not Available, #NM			X	X	If CR0.TS[3]=1.
Stack, #SS(0)			X		If fault suppression not set, and an illegal address in the SS segment.
				X	If fault suppression not set, and a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)			X		If fault suppression not set, and an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If fault suppression not set, and the memory address is in a non-canonical form.
Page Fault, #PF(fault-code)			X	X	If fault suppression not set, and a page fault.
Alignment Check, #AC(0)			X	X	For 2, 4, or 8 byte memory access if alignment checking is enabled and an unaligned memory access is made while the current privilege level is 3. For 16, 32, or 64 byte memory accesses if alignment checking is enabled and an unaligned memory access is made while the current privilege level is 3. See Section 5.9.

5.5 Exceptions Type E7NM

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			If EVEX prefix present.
			X	X	If CR4.OSXSAVE[18]=0. If any one of the following conditions applies: – State requirement not met. – Opcode independent #UD conditions. – Opcode dependent #UD conditions.
	X	X	X	X	If preceded by a LOCK prefix (F0H).
			X	X	If any REX, F2, F3 or 66 prefixes precede a EVEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is ‘0’.
Device Not Available, #NM	X	X	X	X	If CR0.TS[3]=1.

5.6 Exceptions Type AMX

5.6.1 AMX-E1 (LDTILECFG)

Refer to S2.10 “INTEL AMX INSTRUCTION EXCEPTION CLASSES” in Intel SDM Vol. 2A 2-69.

- #UD if preceded by LOCK, 66H, F2H, F3H or REX prefixes.
- #UD if CR4.OSXSAVE != 1
- #UD if XCR0[18:17] != 0b11
- #UD if IA32_EFER.LMA != 1 OR CS.L != 1
- #UD if VVVV != 0b1111
- #GP based on palette and configuration checks (see pseudocode).
- #GP if the memory address is in a non-canonical form.
- #SS(0) if the memory address referencing the SS segment is in a non-canonical form.
- #PF if a page fault occurs.

5.6.2 AMX-E2 (STTILECFG)

- #UD if preceded by LOCK, 66H, F2H, F3H or REX prefixes.
- #UD if CR4.OSXSAVE != 1
- #UD if XCR0[18:17] != 0b11
- #UD if IA32_EFER.LMA != 1 OR CS.L != 1
- #UD if VVVV != 0b1111
- #GP if the memory address is in a non-canonical form.
- #SS(0) if the memory address referencing the SS segment is in a non-canonical form.
- #PF if a page fault occurs.

5.6.3 AMX-E5 (TILEZERO)

- #UD if preceded by LOCK, 66H, F2H, F3H or REX prefixes.
- #UD if CR4.OSXSAVE != 1
- #UD if XCR0[18:17] != 0b11
- #UD if IA32_EFER.LMA != 1 OR CS.L != 1
- #UD if VVVV != 0b1111
- #UD if TILES_CONFIGURED == 0
- #UD if tdest is not a valid tile.
- #UD if tdest is >= palette_table[tilecfg.palette_id].max_names.
- #NM if IA32_XFD[18] == 1

5.6.4 AMX-E6 (TILERELEASE)

- #UD if preceded by LOCK, 66H, F2H, F3H or REX prefixes.
- #UD if CR4.OSXSAVE != 1
- #UD if XCR0[18:17] != 0b11
- #UD if IA32_EFER.LMA != 1 OR CS.L != 1
- #UD if VVVV != 0b1111

5.6.5 AMX-E1-EVEX (LDTILECFG, EVEX-promoted)

EVEX-promoted form of LDTILECFG for APX.

- #UD if CR4.OSXSAVE != 1
- #UD if XCR0[18:17] != 0b11
- #UD if XCR0[19] != 0b1
- #UD if IA32_EFER.LMA != 1 OR CS.L != 1
- #UD if APX_F CPUID feature flag is 0
- #UD if any REX, F2, F3 or 66 prefixes precede the EVEX prefix.
- #UD if preceded by a LOCK prefix (F0H).
- #UD if EVEX.VVVV != 0b1111
- #UD if EVEX.V’ != 0b1
- #UD if EVEX.z != 0b0
- #UD if EVEX.LL’ != 0b00
- #UD if EVEX.b != 0b0

- #UD if EVEX.aaa != 0b000
- #GP based on palette and configuration checks (see pseudocode).
- #GP if the memory address is in a non-canonical form.
- #SS(0) if the memory address referencing the SS segment is in a non-canonical form.
- #PF if a page fault occurs.

5.6.6 AMX-E2-EVEX (STTILECFG, EVEX-promoted)

EVEX-promoted form of STTILECFG for APX.

- #UD if CR4.OSXSAVE != 1
- #UD if XCR0[18:17] != 0b11
- #UD if XCR0[19] != 0b1
- #UD if IA32_EFER.LMA != 1 OR CS.L != 1
- #UD if APX_F CPUID feature flag is 0
- #UD if any REX, F2, F3 or 66 prefixes precede the EVEX prefix.
- #UD if preceded by a LOCK prefix (F0H).
- #UD if EVEX.VVVV != 0b1111
- #UD if EVEX.V’ != 0b1
- #UD if EVEX.z != 0b0
- #UD if EVEX.LL’ != 0b00
- #UD if EVEX.b != 0b0
- #UD if EVEX.aaa != 0b000
- #GP if the memory address is in a non-canonical form.
- #SS(0) if the memory address referencing the SS segment is in a non-canonical form.
- #PF if a page fault occurs.

5.7 Exceptions Type ACE

5.7.1 ACE-E1

ACE tile instructions with immediate row/column specifier.

- #UD if preceded by LOCK, 66H, F2H, F3H or REX prefixes.
- #UD if CR4.OSXSAVE != 1
- #UD if (palette_id != 1) and (palette_id != 2)
- #UD if (palette_id == 1 and XCR0[18:17] != 0b11) or (palette_id == 2 and XCR0[20, 18:17] != 0b111)
- #UD if (TMM is dest and palette_id != 2)
- #UD if XCR0[7:5] != 0b111
- #UD if XCR0[2:1] != 0b11
- #UD if IA32_EFER.LMA != 1 OR CS.L != 1
- #UD if TILES_CONFIGURED == 0
- #UD if tsrc/tdest is not a valid tile name for configured palette (i.e. tsrc/tdest is >= palette_table[tilecfg.palette_id].max_names).
- #UD if tsrc.colsb % 4 != 0
- #UD if EVEX.z != 0b0
- #UD if EVEX.L'L != 0b10
- #UD if EVEX.b != 0b0
- #UD if EVEX.aaa != 0b000
- #UD if EVEX.u != 0b1
- #UD if EVEX.V' != 0b1
- #UD if EVEX.VVVV != 0b1111
- #NM if CR0.TS[3]=1
- #NM if IA32_XFD[18] == 1

5.7.2 ACE-E2

ACE BSR MOVE FULL instruction.

- #UD if preceded by LOCK, 66H, F2H, F3H or REX prefixes.
- #UD if State or Opcode #UD conditions not met (see Section 5.8)
- #UD if CR4.OSXSAVE != 1
- #UD if palette_id != 2
- #UD if XCR0[20, 18:17] != 0b111
- #UD if XCR0[7:5] != 0b111
- #UD if XCR0[2:1] != 0b11
- #UD if IA32_EFER.LMA != 1 OR CS.L != 1
- #UD if TILES_CONFIGURED == 0
- #UD if EVEX.z != 0.
- #UD if EVEX.L'L != 0b10
- #UD if EVEX.b != 0b0
- #UD if EVEX.aaa != 0b000
- #UD if EVEX.R != 0b1
- #UD if EVEX.R' != 0b1
- #UD if BSR dest is not bsr0 (ModRM.REG != 0b000)
- #NM if CR0.TS[3]=1
- #NM if IA32_XFD[18] == 1
- #GP if the memory address is in a non-canonical form.
- #SS(0) if the memory address referencing the SS segment is in a non-canonical form.
- #PF if a page fault occurs.
- #AC(0): For 16, 32, or 64 byte memory accesses if alignment checking is enabled and an unaligned memory access is made while the current privilege level is 3. See Section 5.9.

5.7.3 ACE-E3

ACE BSR MOVE HALF instruction.

- #UD if preceded by LOCK, 66H, F2H, F3H or REX prefixes.
- #UD if State or Opcode #UD conditions not met (see Section 5.8)
- #UD if CR4.OSXSAVE != 1
- #UD if palette_id != 2
- #UD if XCR0[20, 18:17] != 0b111
- #UD if XCR0[7:5] != 0b111
- #UD if XCR0[2:1] != 0b11
- #UD if IA32_EFER.LMA != 1 OR CS.L != 1
- #UD if TILES_CONFIGURED == 0
- #UD if EVEX.z != 0.
- #UD if EVEX.L'L != 0b10
- #UD if EVEX.b != 0b0
- #UD if EVEX.aaa != 0b000
- #UD if EVEX.R != 0b1
- #UD if EVEX.R' != 0b1
- #UD if BSR src/dest is not bsr0 (ModRM.REG != 0b000)
- #UD if EVEX.V' != 0b1
- #UD if EVEX.VVVV != 0b1111
- #NM if CR0.TS[3]=1
- #NM if IA32_XFD[18] == 1
- #GP if the memory address is in a non-canonical form.
- #SS(0) if the memory address referencing the SS segment is in a non-canonical form.
- #PF if a page fault occurs.
- #AC(0): For 16, 32, or 64 byte memory accesses if alignment checking is enabled and an unaligned memory access is made while the current privilege level is 3. See Section 5.9.

5.7.4 ACE-E4

ACE outer product instructions.

- #UD if preceded by LOCK, 66H, F2H, F3H or REX prefixes.
- #UD if CR4.OSXSAVE != 1
- #UD if palette_id != 2
- #UD if XCR0[20, 18:17] != 0b111
- #UD if XCR0[7:5] != 0b111
- #UD if XCR0[2:1] != 0b11
- #UD if IA32_EFER.LMA != 1 OR CS.L != 1
- #UD if TILES_CONFIGURED == 0
- #UD if tsrc/tdest is not a valid tile name for configured palette (i.e. tsrc/tdest is >= palette_table[tilecfg.palette_id].max_names).
- #UD if EVEX.z != 0.
- #UD if EVEX.L'L != 0b10
- #UD if EVEX.b != 0b0
- #UD if EVEX.aaa != 0b000
- #NM if CR0.TS[3]=1
- #NM if IA32_XFD[18] == 1

5.7.5 ACE-E5

ACE BSR Init instruction.[1]

- #UD if preceded by LOCK, 66H, F2H, F3H or REX prefixes.
- #UD if CR4.OSXSAVE != 1
- #UD if palette_id != 2
- #UD if XCR0[20, 18:17] != 0b111
- #UD if IA32_EFER.LMA != 1 OR CS.L != 1
- #UD if TILES_CONFIGURED == 0

[1] BSR Init does not access AVX registers; no check is made of XCR0[7:5] or XCR0[2:1].

- #UD if VVVV != 0b1111
- #NM if IA32_XFD[18] == 1

5.7.6 ACE-E6

ACE tile instructions with GPR row/column specifier.

- #UD if preceded by LOCK, 66H, F2H, F3H or REX prefixes.
- #UD if CR4.OSXSAVE != 1
- #UD if State or Opcode #UD conditions not met (see Section 5.8)
- #UD if (palette_id != 1) and (palette_id != 2)
- #UD if (palette_id == 1 and XCR0[18:17] != 0b11) or (palette_id == 2 and XCR0[20, 18:17] != 0b111)
- #UD if (TMM is dest and palette_id != 2)
- #UD if XCR0[7:5] != 0b111
- #UD if XCR0[2:1] != 0b11
- #UD if IA32_EFER.LMA != 1 OR CS.L != 1
- #UD if TILES_CONFIGURED == 0
- #UD if tsrc/tdest is not a valid tile name for configured palette (i.e. tsrc/tdest is >= palette_table[tilecfg.palette_id].max_names).
- #UD if tsrc.colsb % 4 != 0
- #UD if EVEX.z != 0b0
- #UD if EVEX.L’L != 0b10
- #UD if EVEX.b != 0b0
- #UD if EVEX.aaa != 0b000
- #UD if EVEX.u != 0b1
- #NM if CR0.TS[3]=1
- #NM if IA32_XFD[18] == 1

5.8 #UD Equations for EVEX Encoded Instructions

EVEX encoded instructions can face #UD conditions that are state, opcode or state and opcode dependent.

5.8.1 State and Opcode Dependent #UD

The introduction of APX introduces #UD conditions that are dependent on the status of XCR0[APX_F].

Where an EVEX encoded instruction specifies a GPR as an argument, the presence of APX extends the field used to specify the GPR:

- If a register field is not used to specify a GPR, the existing encoding constraint applies, irrespective of XCR0[APX_F].
- If XCR0[APX_F] = 0, the existing encoding constraint applies.
- If XCR0[APX_F] = 1, the field encoding a GPR is extended and the exception conditions modified according to the following table.

Field	XCR0[APX_F] = 0	XCR0[APX_F] = 1
ModRM:reg	#UD if EVEX.R4 != 1	EVEX.R4 permitted to extend GPR specifier
ModRM:RM(GPR)	#UD if EVEX.P0[3] != 0	EVEX.B4 (=EVEX.P0[3]) permitted to extend GPR specifier
EVEX.VVVV	#UD if EVEX.V' != 1	EVEX.V' permitted to extend GPR specifier
BASE	#UD if EVEX.P0[3] != 0	EVEX.B4 (=EVEX.P0[3]) permitted to extend GPR specifier
INDEX	#UD if EVEX.P1[2] != 1	EVEX.X4 (= EVEX.U = EVEX.P1[2]) permitted to extend SIB.INDEX specifier Note: EVEX.X4 is only defined when ModRM.Mod != 3.

Table 1: EVEX exception conditions dependent on XCR0[APX_F].

If any of the bits R4, B4, and X4 is not used by an EVEX instruction, it is ignored, and code-generators should set these bits to their logical 0 value (i.e., 1 for inverted bit fields, 0 for regular bit fields). Note, however, that EVEX.X4 is defined only when ModRM.Mod != 3.

5.9 Implementation Specific Alignment Checking for 16, 32 and 64 Byte Accesses

Some implementations extend alignment checking to include checking for 16-byte alignment on 16-, 32- and 64-byte accesses. On these implementations, a non-16-byte aligned memory reference will cause an #AC exception on noted instructions if the following conditions are met:

- Current execution level is 3
- EFLAGS.AC = 1
- CR0.AM = 1

6 Instruction Encodings

6.1 Instructions Enumerated under AVX10_V1_AUX

6.1.1 CPUID

`CPUID.(EAX=24H, ECX=1): ECX[2]  (AVX10_V1_AUX)`

6.1.2 FP16 to FP8 Converts (Single-Source)

Mnemonic	Form	Encoding
VCVTPH2BF8	xmm1{k1}{z}, xmm2/m128/m16bcst	EVEX.128.F3.0F38.W0 74 /r
VCVTPH2BF8	xmm1{k1}{z}, ymm2/m256/m16bcst	EVEX.256.F3.0F38.W0 74 /r
VCVTPH2BF8	ymm1{k1}{z}, zmm2/m512/m16bcst	EVEX.512.F3.0F38.W0 74 /r
VCVTPH2BF8S	xmm1{k1}{z}, xmm2/m128/m16bcst	EVEX.128.F3.MAP5.W0 74 /r
VCVTPH2BF8S	xmm1{k1}{z}, ymm2/m256/m16bcst	EVEX.256.F3.MAP5.W0 74 /r
VCVTPH2BF8S	ymm1{k1}{z}, zmm2/m512/m16bcst	EVEX.512.F3.MAP5.W0 74 /r
VCVTPH2HF8	xmm1{k1}{z}, xmm2/m128/m16bcst	EVEX.128.F3.MAP5.W0 18 /r
VCVTPH2HF8	xmm1{k1}{z}, ymm2/m256/m16bcst	EVEX.256.F3.MAP5.W0 18 /r
VCVTPH2HF8	ymm1{k1}{z}, zmm2/m512/m16bcst	EVEX.512.F3.MAP5.W0 18 /r
VCVTPH2HF8S	xmm1{k1}{z}, xmm2/m128/m16bcst	EVEX.128.F3.MAP5.W0 1B /r
VCVTPH2HF8S	xmm1{k1}{z}, ymm2/m256/m16bcst	EVEX.256.F3.MAP5.W0 1B /r
VCVTPH2HF8S	ymm1{k1}{z}, zmm2/m512/m16bcst	EVEX.512.F3.MAP5.W0 1B /r

6.1.3 FP16 to FP8 Converts (Two-Source)

Mnemonic	Form	Encoding
VCVT2PH2BF8	xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	EVEX.128.F2.0F38.W0 74 /r
VCVT2PH2BF8	ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst	EVEX.256.F2.0F38.W0 74 /r
VCVT2PH2BF8	zmm1{k1}{z}, zmm2, zmm3/m512/m16bcst	EVEX.512.F2.0F38.W0 74 /r
VCVT2PH2BF8S	xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	EVEX.128.F2.MAP5.W0 74 /r
VCVT2PH2BF8S	ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst	EVEX.256.F2.MAP5.W0 74 /r
VCVT2PH2BF8S	zmm1{k1}{z}, zmm2, zmm3/m512/m16bcst	EVEX.512.F2.MAP5.W0 74 /r
VCVT2PH2HF8	xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	EVEX.128.F2.MAP5.W0 18 /r
VCVT2PH2HF8	ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst	EVEX.256.F2.MAP5.W0 18 /r
VCVT2PH2HF8	zmm1{k1}{z}, zmm2, zmm3/m512/m16bcst	EVEX.512.F2.MAP5.W0 18 /r
VCVT2PH2HF8S	xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	EVEX.128.F2.MAP5.W0 1B /r
VCVT2PH2HF8S	ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst	EVEX.256.F2.MAP5.W0 1B /r
VCVT2PH2HF8S	zmm1{k1}{z}, zmm2, zmm3/m512/m16bcst	EVEX.512.F2.MAP5.W0 1B /r

6.1.4 FP16 to FP8 Converts (Biased)

Mnemonic	Form	Encoding
VCVTBIASPH2BF8	xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	EVEX.128.NP.0F38.W0 74 /r
VCVTBIASPH2BF8	xmm1{k1}{z}, ymm2, ymm3/m256/m16bcst	EVEX.256.NP.0F38.W0 74 /r
VCVTBIASPH2BF8	ymm1{k1}{z}, zmm2, zmm3/m512/m16bcst	EVEX.512.NP.0F38.W0 74 /r
VCVTBIASPH2BF8S	xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	EVEX.128.NP.MAP5.W0 74 /r
VCVTBIASPH2BF8S	xmm1{k1}{z}, ymm2, ymm3/m256/m16bcst	EVEX.256.NP.MAP5.W0 74 /r
VCVTBIASPH2BF8S	ymm1{k1}{z}, zmm2, zmm3/m512/m16bcst	EVEX.512.NP.MAP5.W0 74 /r
VCVTBIASPH2HF8	xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	EVEX.128.NP.MAP5.W0 18 /r
VCVTBIASPH2HF8	xmm1{k1}{z}, ymm2, ymm3/m256/m16bcst	EVEX.256.NP.MAP5.W0 18 /r
VCVTBIASPH2HF8	ymm1{k1}{z}, zmm2, zmm3/m512/m16bcst	EVEX.512.NP.MAP5.W0 18 /r
VCVTBIASPH2HF8S	xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	EVEX.128.NP.MAP5.W0 1B /r
VCVTBIASPH2HF8S	xmm1{k1}{z}, ymm2, ymm3/m256/m16bcst	EVEX.256.NP.MAP5.W0 1B /r

Mnemonic	Form	Encoding
VCVTBIASPH2HF8S	ymm1{k1}{z}, zmm2, zmm3/m512/m16bcst	EVEX.512.NP.MAP5.W0 1B /r

6.1.5 FP8 to FP16 Converts

Mnemonic	Form	Encoding
VCVTHF82PH	xmm1{k1}{z}, xmm2/m64	EVEX.128.F2.MAP5.W0 1E /r
VCVTHF82PH	ymm1{k1}{z}, xmm2/m128	EVEX.256.F2.MAP5.W0 1E /r
VCVTHF82PH	zmm1{k1}{z}, ymm2/m256	EVEX.512.F2.MAP5.W0 1E /r

6.1.6 FP32-Pair to FP16 Converts

Mnemonic	Form	Encoding
VCVT2PS2PHX	xmm1{k1}{z}, xmm2, xmm3/m128/m32bcst	EVEX.128.66.0F38.W0 67 /r
VCVT2PS2PHX	ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst	EVEX.256.66.0F38.W0 67 /r
VCVT2PS2PHX	zmm1{k1}{z}, zmm2, zmm3/m512/m32bcst {er}	EVEX.512.66.0F38.W0 67 /r

6.1.7 Byte VNNI (EVEX)

Mnemonic	Form	Encoding
VPDPBSSD	xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	EVEX.128.F2.0F38.W0 50 /r
VPDPBSSD	ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	EVEX.256.F2.0F38.W0 50 /r
VPDPBSSD	zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	EVEX.512.F2.0F38.W0 50 /r
VPDPBSSDS	xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	EVEX.128.F2.0F38.W0 51 /r
VPDPBSSDS	ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	EVEX.256.F2.0F38.W0 51 /r
VPDPBSSDS	zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	EVEX.512.F2.0F38.W0 51 /r
VPDPBSUD	xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	EVEX.128.F3.0F38.W0 50 /r
VPDPBSUD	ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	EVEX.256.F3.0F38.W0 50 /r
VPDPBSUD	zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	EVEX.512.F3.0F38.W0 50 /r
VPDPBSUDS	xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	EVEX.128.F3.0F38.W0 51 /r
VPDPBSUDS	ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	EVEX.256.F3.0F38.W0 51 /r
VPDPBSUDS	zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	EVEX.512.F3.0F38.W0 51 /r
VPDPBUUD	xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	EVEX.128.NP.0F38.W0 50 /r
VPDPBUUD	ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	EVEX.256.NP.0F38.W0 50 /r
VPDPBUUD	zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	EVEX.512.NP.0F38.W0 50 /r
VPDPBUUDS	xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	EVEX.128.NP.0F38.W0 51 /r
VPDPBUUDS	ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	EVEX.256.NP.0F38.W0 51 /r
VPDPBUUDS	zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	EVEX.512.NP.0F38.W0 51 /r

6.1.8 Word VNNI (EVEX)

Mnemonic	Form	Encoding
VPDPWSUD	xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	EVEX.128.F3.0F38.W0 D2 /r
VPDPWSUD	ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	EVEX.256.F3.0F38.W0 D2 /r
VPDPWSUD	zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	EVEX.512.F3.0F38.W0 D2 /r
VPDPWSUDS	xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	EVEX.128.F3.0F38.W0 D3 /r
VPDPWSUDS	ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	EVEX.256.F3.0F38.W0 D3 /r
VPDPWSUDS	zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	EVEX.512.F3.0F38.W0 D3 /r
VPDPWUSD	xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	EVEX.128.66.0F38.W0 D2 /r
VPDPWUSD	ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	EVEX.256.66.0F38.W0 D2 /r
VPDPWUSD	zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	EVEX.512.66.0F38.W0 D2 /r
VPDPWUSDS	xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	EVEX.128.66.0F38.W0 D3 /r
VPDPWUSDS	ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	EVEX.256.66.0F38.W0 D3 /r
VPDPWUSDS	zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	EVEX.512.66.0F38.W0 D3 /r

Mnemonic	Form	Encoding
VPDPWUUD	xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	EVEX.128.NP.0F38.W0 D2 /r
VPDPWUUD	ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	EVEX.256.NP.0F38.W0 D2 /r
VPDPWUUD	zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	EVEX.512.NP.0F38.W0 D2 /r
VPDPWUUDS	xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	EVEX.128.NP.0F38.W0 D3 /r
VPDPWUUDS	ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	EVEX.256.NP.0F38.W0 D3 /r
VPDPWUUDS	zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	EVEX.512.NP.0F38.W0 D3 /r

6.2 Instructions Enumerated under AVX10_V2_AUX

6.2.1 CPUID

`CPUID.(EAX=24H, ECX=1): ECX[3]  (AVX10_V2_AUX)`

6.2.2 FP32 to FP8 Converts (Single-Source, RNE)

Mnemonic	Form	Encoding
VCVTPS2BF8	xmm1 {k1}{z}, xmm2/m128/m32bcst	EVEX.128.F3.MAP5.W0 39 /r
VCVTPS2BF8	xmm1 {k1}{z}, ymm2/m256/m32bcst	EVEX.256.F3.MAP5.W0 39 /r
VCVTPS2BF8	xmm1 {k1}{z}, zmm2/m512/m32bcst	EVEX.512.F3.MAP5.W0 39 /r
VCVTPS2BF8S	xmm1 {k1}{z}, xmm2/m128/m32bcst	EVEX.128.F3.MAP5.W0 3B /r
VCVTPS2BF8S	xmm1 {k1}{z}, ymm2/m256/m32bcst	EVEX.256.F3.MAP5.W0 3B /r
VCVTPS2BF8S	xmm1 {k1}{z}, zmm2/m512/m32bcst	EVEX.512.F3.MAP5.W0 3B /r
VCVTPS2HF8	xmm1 {k1}{z}, xmm2/m128/m32bcst	EVEX.128.F3.MAP5.W0 38 /r
VCVTPS2HF8	xmm1 {k1}{z}, ymm2/m256/m32bcst	EVEX.256.F3.MAP5.W0 38 /r
VCVTPS2HF8	xmm1 {k1}{z}, zmm2/m512/m32bcst	EVEX.512.F3.MAP5.W0 38 /r
VCVTPS2HF8S	xmm1 {k1}{z}, xmm2/m128/m32bcst	EVEX.128.F3.MAP5.W0 3A /r
VCVTPS2HF8S	xmm1 {k1}{z}, ymm2/m256/m32bcst	EVEX.256.F3.MAP5.W0 3A /r
VCVTPS2HF8S	xmm1 {k1}{z}, zmm2/m512/m32bcst	EVEX.512.F3.MAP5.W0 3A /r

6.2.3 FP32 to FP8 Converts (Single-Source, RTO)

Mnemonic	Form	Encoding
VCVTROPS2HF8	xmm1 {k1}{z}, xmm2/m128/m32bcst	EVEX.128.66.MAP5.W0 38 /r
VCVTROPS2HF8	xmm1 {k1}{z}, ymm2/m256/m32bcst	EVEX.256.66.MAP5.W0 38 /r
VCVTROPS2HF8	xmm1 {k1}{z}, zmm2/m512/m32bcst	EVEX.512.66.MAP5.W0 38 /r
VCVTROPS2HF8S	xmm1 {k1}{z}, xmm2/m128/m32bcst	EVEX.128.66.MAP5.W0 3A /r
VCVTROPS2HF8S	xmm1 {k1}{z}, ymm2/m256/m32bcst	EVEX.256.66.MAP5.W0 3A /r
VCVTROPS2HF8S	xmm1 {k1}{z}, zmm2/m512/m32bcst	EVEX.512.66.MAP5.W0 3A /r

6.2.4 FP32 to FP8 Converts (Biased)

Mnemonic	Form	Encoding
VCVTBIASPS2BF8	xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	EVEX.128.NP.MAP5.W0 39 /r
VCVTBIASPS2BF8	xmm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	EVEX.256.NP.MAP5.W0 39 /r
VCVTBIASPS2BF8	xmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	EVEX.512.NP.MAP5.W0 39 /r
VCVTBIASPS2BF8S	xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	EVEX.128.NP.MAP5.W0 3B /r
VCVTBIASPS2BF8S	xmm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	EVEX.256.NP.MAP5.W0 3B /r
VCVTBIASPS2BF8S	xmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	EVEX.512.NP.MAP5.W0 3B /r
VCVTBIASPS2HF8	xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	EVEX.128.NP.MAP5.W0 38 /r
VCVTBIASPS2HF8	xmm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	EVEX.256.NP.MAP5.W0 38 /r
VCVTBIASPS2HF8	xmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	EVEX.512.NP.MAP5.W0 38 /r
VCVTBIASPS2HF8S	xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	EVEX.128.NP.MAP5.W0 3A /r
VCVTBIASPS2HF8S	xmm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	EVEX.256.NP.MAP5.W0 3A /r
VCVTBIASPS2HF8S	xmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	EVEX.512.NP.MAP5.W0 3A /r

6.2.5 FP8 to FP32 Converts

Mnemonic	Form	Encoding
VCVTBF82PS	xmm1 {k1}{z}, xmm2/m32	EVEX.128.NP.MAP5.W1 36 /r
VCVTBF82PS	ymm1 {k1}{z}, xmm2/m64	EVEX.256.NP.MAP5.W1 36 /r
VCVTBF82PS	zmm1 {k1}{z}, xmm2/m128	EVEX.512.NP.MAP5.W1 36 /r
VCVTHF82PS	xmm1 {k1}{z}, xmm2/m32	EVEX.128.NP.MAP5.W0 36 /r
VCVTHF82PS	ymm1 {k1}{z}, xmm2/m64	EVEX.256.NP.MAP5.W0 36 /r
VCVTHF82PS	zmm1 {k1}{z}, xmm2/m128	EVEX.512.NP.MAP5.W0 36 /r

6.2.6 FP8 to FP4 Converts

Mnemonic	Form	Encoding
VCVTBF82BF4S	xmm1/m64, xmm2	EVEX.128.F3.MAP5.W1 3D /r
VCVTBF82BF4S	xmm1/m128, ymm2	EVEX.256.F3.MAP5.W1 3D /r
VCVTBF82BF4S	ymm1/m256, zmm2	EVEX.512.F3.MAP5.W1 3D /r
VCVTHF82BF4S	xmm1/m64, xmm2	EVEX.128.F3.MAP5.W0 3D /r
VCVTHF82BF4S	xmm1/m128, ymm2	EVEX.256.F3.MAP5.W0 3D /r
VCVTHF82BF4S	ymm1/m256, zmm2	EVEX.512.F3.MAP5.W0 3D /r

6.2.7 FP4 to FP8 Converts

Mnemonic	Form	Encoding
VCVTBF42HF8	xmm1 {k1}{z}, xmm2/m64	EVEX.128.NP.MAP5.W0 37 /r
VCVTBF42HF8	ymm1 {k1}{z}, xmm2/m128	EVEX.256.NP.MAP5.W0 37 /r
VCVTBF42HF8	zmm1 {k1}{z}, ymm2/m256	EVEX.512.NP.MAP5.W0 37 /r

6.2.8 FP8 to FP6 Converts

Mnemonic	Form	Encoding
VCVTBF82BF6S	xmm1, xmm2	EVEX.128.F3.MAP5.W1 3E /r
VCVTBF82BF6S	ymm1, ymm2	EVEX.256.F3.MAP5.W1 3E /r
VCVTBF82BF6S	zmm1, zmm2	EVEX.512.F3.MAP5.W1 3E /r
VCVTHF82HF6S	xmm1, xmm2	EVEX.128.F3.MAP5.W0 3C /r
VCVTHF82HF6S	ymm1, ymm2	EVEX.256.F3.MAP5.W0 3C /r
VCVTHF82HF6S	zmm1, zmm2	EVEX.512.F3.MAP5.W0 3C /r

6.2.9 FP6 to FP8 Converts

Mnemonic	Form	Encoding
VCVTBF62HF8	xmm1 {k1}{z}, xmm2	EVEX.128.66.MAP5.W1 37 /r
VCVTBF62HF8	ymm1 {k1}{z}, ymm2	EVEX.256.66.MAP5.W1 37 /r
VCVTBF62HF8	zmm1 {k1}{z}, zmm2	EVEX.512.66.MAP5.W1 37 /r
VCVTHF62HF8	xmm1 {k1}{z}, xmm2	EVEX.128.66.MAP5.W0 37 /r
VCVTHF62HF8	ymm1 {k1}{z}, ymm2	EVEX.256.66.MAP5.W0 37 /r
VCVTHF62HF8	zmm1 {k1}{z}, zmm2	EVEX.512.66.MAP5.W0 37 /r

6.2.10 Sub-Byte Element Extraction

Mnemonic	Form	Encoding
VUNPACKB	xmm1 {k1}{z}, xmm2/m128, imm8	EVEX.128.NP.0F3A.W0 3D /r
VUNPACKB	ymm1 {k1}{z}, ymm2/m256, imm8	EVEX.256.NP.0F3A.W0 3D /r
VUNPACKB	zmm1 {k1}{z}, zmm2/m512, imm8	EVEX.512.NP.0F3A.W0 3D /r

6.2.11 Symmetric Signed Saturation Narrow

Mnemonic	Form	Encoding
VPMOVSSDB	xmm1/m32 {k1}{z}, xmm2	EVEX.128.F3.0F38.W0 41 /r
VPMOVSSDB	xmm1/m64 {k1}{z}, ymm2	EVEX.256.F3.0F38.W0 41 /r
VPMOVSSDB	xmm1/m128 {k1}{z}, zmm2	EVEX.512.F3.0F38.W0 41 /r

6.3 Instructions Enumerated under ACE v1 (Tile)

6.3.1 CPUID

`CPUID.(EAX=07H, ECX=1): ECX[11]  (ACE)`

6.3.2 Tile Management (Imported)

Mnemonic	Form	Encoding
TILEZERO	tmm1	VEX.128.F2.0F38.W0 49 11:rrr:000 /
LDTILECFG	m512	VEX.128.NP.0F38.W0 49 !(11):000:bbb /
STTILECFG	m512	VEX.128.66.0F38.W0 49 !(11):000:bbb /
TILERELEASE		VEX.128.NP.0F38.W0 49 C0 /

6.3.3 Tile Data Movement

Mnemonic	Form	Encoding
TILEMOVROW	zmm1, tmm2, r32	EVEX.512.66.0F38.W0 4A 11:rrr:bbb
TILEMOVROW	zmm1, tmm2, imm8	EVEX.512.66.0F3A.W0 07 11:rrr:bbb /ib
TILEMOVROW	tmm1, zmm2, r32	EVEX.512.66.0F38.W1 4A 11:rrr:bbb
TILEMOVROW	tmm1, zmm2, imm8	EVEX.512.66.0F3A.W1 07 11:rrr:bbb /ib
TILEMOVCOL	tmm1, zmm2, r32	EVEX.512.66.0F38.W1 4B 11:rrr:bbb
TILEMOVCOL	tmm1, zmm2, imm8	EVEX.512.66.0F3A.W1 2F 11:rrr:bbb /ib

6.3.4 Tile Row Convert and Move

Mnemonic	Form	Encoding
TCVTROWD2PS	zmm1, tmm2, r32	EVEX.512.F3.0F38.W0 4A 11:rrr:bbb
TCVTROWD2PS	zmm1, tmm2, imm8	EVEX.512.F3.0F3A.W0 07 11:rrr:bbb /ib
TCVTROWPS2BF16H	zmm1, tmm2, r32	EVEX.512.F2.0F38.W0 6D 11:rrr:bbb
TCVTROWPS2BF16H	zmm1, tmm2, imm8	EVEX.512.F2.0F3A.W0 07 11:rrr:bbb /ib
TCVTROWPS2BF16L	zmm1, tmm2, r32	EVEX.512.F3.0F38.W0 6D 11:rrr:bbb
TCVTROWPS2BF16L	zmm1, tmm2, imm8	EVEX.512.F3.0F3A.W0 77 11:rrr:bbb /ib
TCVTROWPS2PHH	zmm1, tmm2, r32	EVEX.512.NP.0F38.W0 6D 11:rrr:bbb
TCVTROWPS2PHH	zmm1, tmm2, imm8	EVEX.512.NP.0F3A.W0 07 11:rrr:bbb /ib
TCVTROWPS2PHL	zmm1, tmm2, r32	EVEX.512.66.0F38.W0 6D 11:rrr:bbb
TCVTROWPS2PHL	zmm1, tmm2, imm8	EVEX.512.F2.0F3A.W0 77 11:rrr:bbb /ib

6.3.5 Block Scale Register (BSR) Operations

Mnemonic	Form	Encoding
BSRINIT		VEX.128.F2.0F38.W1 49 C0
BSRMOVF	bsr0, zmm1, zmm2/m512	EVEX.512.NP.MAP6.W1 95 mm:000:bbb
BSRMOVH	bsr0, zmm1/m512	EVEX.512.F2.MAP6.W1 95 mm:000:bbb
BSRMOVH	zmm1/m512, bsr0	EVEX.512.F2.MAP6.W0 95 mm:000:bbb
BSRMOVL	bsr0, zmm1/m512	EVEX.512.F3.MAP6.W1 95 mm:000:bbb
BSRMOVL	zmm1/m512, bsr0	EVEX.512.F3.MAP6.W0 95 mm:000:bbb

6.3.6 MX FP8 Rank-4 Outer Products

Mnemonic	Form	Encoding
TOP4MXBF8PS	tmm1, zmm2, zmm3, imm8	EVEX.512.NP.0F3A.W0 8D 11:rrr:bbb /ib

Mnemonic	Form	Encoding
TOP4MXBHF8PS	tmm1, zmm2, zmm3, imm8	EVEX.512.F2.0F3A.W0 8D 11:rrr:bbb /ib
TOP4MXHBF8PS	tmm1, zmm2, zmm3, imm8	EVEX.512.F3.0F3A.W0 8D 11:rrr:bbb /ib
TOP4MXHF8PS	tmm1, zmm2, zmm3, imm8	EVEX.512.66.0F3A.W0 8D 11:rrr:bbb /ib

6.3.7 MX INT8 Rank-4 Outer Product

Mnemonic	Form	Encoding
TOP4MXBSSPS	tmm1, zmm2, zmm3, imm8	EVEX.512.F2.0F3A.W0 8F 11:rrr:bbb /ib

6.3.8 BF16 Rank-2 Outer Product

Mnemonic	Form	Encoding
TOP2BF16PS	tmm1, zmm2, zmm3	EVEX.512.F3.0F38.W0 5C 11:rrr:bbb

6.3.9 INT8 Byte Rank-4 Outer Products

Mnemonic	Form	Encoding
TOP4BSSD	tmm1, zmm2, zmm3	EVEX.512.F2.0F38.W0 5E 11:rrr:bbb
TOP4BSUD	tmm1, zmm2, zmm3	EVEX.512.F3.0F38.W0 5E 11:rrr:bbb
TOP4BUSD	tmm1, zmm2, zmm3	EVEX.512.66.0F38.W0 5E 11:rrr:bbb
TOP4BUUD	tmm1, zmm2, zmm3	EVEX.512.NP.0F38.W0 5E 11:rrr:bbb

7 AVX-VNNI-INT8 and AVX-VNNI-INT16

Implementations compliant with this specification must include the operations enumerated under feature flags `AVX-VNNI-INT8` and `AVX-VNNI-INT16`. These instructions are documented in the Intel 64 and IA-32 Architectures Software Developer's Manual [4].

- Multiply and Add Unsigned and Signed Bytes With and Without Saturation (VEX)
- Multiply and Add Unsigned and Signed Words With and Without Saturation (VEX)

8 AVX10.2 Subset

8.1 Introduction

Implementations compliant with the ACE v1 architecture must implement the subset of AVX10.2 enumerated by `AVX10_V1_AUX` or the entirety of `AVX10.2`.

From [5] §9 — AVX10.2 Convert Instructions:

- `VCVT[,2]PH2[B,H]F8[,S]` – FP16 to FP8 conversions
- `VCVT2PS2PHX` – FP32 pair to FP16
- `VCVTBIASPH2[B,H]F8[,S]` – FP16 to FP8 with bias
- `VCVTHF82PH` – FP8 E4M3 to FP16

From [5] §10 — AVX10.2 Integer and FP16 VNNI, Media New Instructions:

- `VPDPB[SU,UU,SS]D[,S]` – Byte VNNI (EVEX)
- `VPDPW[SU,US,UU]D[,S]` – Word VNNI (EVEX)

8.2 Convert from FP16 to FP8

8.2.1 Description

These operations convert from FP16 to FP8. They convert one or two SIMD registers of packed FP16 data into a single register of packed FP8 (E5M2 or E4M3) format data. Saturating and non-saturating modes are available. The upper bits of the destination register beyond converted elements are zeroed.

MXCSR is not consulted and execution assumes all MXCSR exceptions are masked. DAZ is not obeyed and is always assumed DAZ=0. FTZ is not obeyed and is always assumed FTZ=0. No floating-point exceptions are generated and no status updated.

From	To	Saturate mode	Rounding mode
FP16	FP8 E4M3	SAT/NSAT	RNE
FP16	FP8 E5M2	SAT/NSAT	RNE

8.2.2 Operands

VCVTPH reads one vector of input and writes one vector of output.

Operand 1	Operand 2	Operand 3
xmm1{k1}{z}	xmm2/m128/m16bcst	
xmm1{k1}{z}	ymm2/m256/m16bcst	
ymm1{k1}{z}	zmm2/m512/m16bcst	

VCVT2PH reads two vectors of input and writes one vector of output.

Operand 1	Operand 2	Operand 3
xmm1{k1}{z}	xmm2	xmm3/m128/m16bcst
ymm1{k1}{z}	ymm2	ymm3/m256/m16bcst
zmm1{k1}{z}	zmm2	zmm3/m512/m16bcst

Instruction	Operand 1	Operand 2	Operand 3
VCVTPH2BF8 VCVTPH2BF8S VCVTPH2HF8 VCVTPH2HF8S	MODRM.REG(w)	MODRM.R/M(r)	N/A
VCVT2PH2BF8 VCVT2PH2BF8S VCVT2PH2HF8 VCVT2PH2HF8S	MODRM.REG(w)	VVVV(r)	MODRM.R/M(r)

8.2.3 SIMD Floating-Point Exceptions

None.

8.2.4 Other Exceptions

Exception class E4NF, see Section 5.3.

8.2.5 Operation

```
DEFINE vcvtph2f8(src, dst_format, saturation_mode, VL, k1, zeroing, no_writemask, src_is_mem,
evex_b):
    // FP16 → FP8 (E5M2 or E4M3), single source
    ASSERT VL in (128, 256, 512)
    KL = VL / 8
    orig_dest = copy(dest)
    FOR i = 0 TO KL / 2-1:
        t = src.fp16[0] if (src_is_mem AND evex_b) else src.fp16[i]
        IF k1[i] OR no_writemask:
            IF dst_format == "E5M2":
                dest.byte[i] = fp16_to_fp8_e5m2(t, saturation_mode)
            ELSE:
                dest.byte[i] = fp16_to_fp8_e4m3(t, saturation_mode)
        ELSE IF zeroing:
            dest.byte[i] = 0
        ELSE:
            dest.byte[i] = orig_dest.byte[i]
    dest[MAXVL-1 : VL/2] = 0
```

```
DEFINE vcvt2ph2f8(src1, src2, dst_format, saturation_mode, VL, k1, zeroing, no_writemask,
src2_is_mem, evex_b):
    // FP16 → FP8, two sources concatenated into same-width output
    ASSERT VL in (128, 256, 512)
    KL = VL / 8
    orig_dest = copy(dest)
    FOR i = 0 TO KL-1:
        IF i < KL / 2:
            t = src2.fp16[0] if (src2_is_mem AND evex_b) else src2.fp16[i]
        ELSE:
            t = src1.fp16[i - KL / 2]
        IF k1[i] OR no_writemask:
            IF dst_format == "E5M2":
                dest.byte[i] = fp16_to_fp8_e5m2(t, saturation_mode)
            ELSE:
                dest.byte[i] = fp16_to_fp8_e4m3(t, saturation_mode)
        ELSE IF zeroing:
            dest.byte[i] = 0
        ELSE:
            dest.byte[i] = orig_dest.byte[i]
    dest[MAXVL-1 : VL] = 0
```

8.2.6 Instruction Table

Mnemonic	Operands	Description
VCVTPH2BF8	xmm1{k1}{z}, xmm2/m128/m16bcst	Convert one packed FP16 vector to FP8 E5M2, non-saturating.
	xmm1{k1}{z}, ymm2/m256/m16bcst	
	ymm1{k1}{z}, zmm2/m512/m16bcst	
VCVTPH2BF8S	xmm1{k1}{z}, xmm2/m128/m16bcst	Convert one packed FP16 vector to FP8 E5M2, saturating.
	xmm1{k1}{z}, ymm2/m256/m16bcst	
	ymm1{k1}{z}, zmm2/m512/m16bcst	
VCVTPH2HF8	xmm1{k1}{z}, xmm2/m128/m16bcst	Convert one packed FP16 vector to FP8 E4M3, non-saturating.
	xmm1{k1}{z}, ymm2/m256/m16bcst	
	ymm1{k1}{z}, zmm2/m512/m16bcst	
VCVTPH2HF8S	xmm1{k1}{z}, xmm2/m128/m16bcst	Convert one packed FP16 vector to FP8 E4M3, saturating.
	xmm1{k1}{z}, ymm2/m256/m16bcst	

Mnemonic	Operands	Description
	ymm1{k1}{z}, zmm2/m512/m16bcst	

Mnemonic	Operands	Description
VCVT2PH2BF8	xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	Convert two packed FP16 vectors to FP8 E5M2, non-saturating.
	ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst	
	zmm1{k1}{z}, zmm2, zmm3/m512/m16bcst	
VCVT2PH2BF8S	xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	Convert two packed FP16 vectors to FP8 E5M2, saturating.
	ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst	
	zmm1{k1}{z}, zmm2, zmm3/m512/m16bcst	
VCVT2PH2HF8	xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	Convert two packed FP16 vectors to FP8 E4M3, non-saturating.
	ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst	
	zmm1{k1}{z}, zmm2, zmm3/m512/m16bcst	
VCVT2PH2HF8S	xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	Convert two packed FP16 vectors to FP8 E4M3, saturating.
	ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst	
	zmm1{k1}{z}, zmm2, zmm3/m512/m16bcst	

8.2.7 C/C++ Compiler Intrinsic Equivalent

```
/* cvtph_bf8 — BF8 (E5M2), non-saturating */
__m256i _mm512_cvtph_bf8(__m512h a);
/* cvtph_hf8 — HF8 (E4M3), non-saturating */
__m256i _mm512_cvtph_hf8(__m512h a);
/* cvtphs_bf8 — BF8 (E5M2), saturating */
__m256i _mm512_cvtphs_bf8(__m512h a);
/* cvtphs_hf8 — HF8 (E4M3), saturating */
__m256i _mm512_cvtphs_hf8(__m512h a);
```

```
/* cvt2ph_bf8 — BF8 (E5M2), non-saturating */
__m512i _mm512_cvt2ph_bf8(__m512h a, __m512h b);
/* cvt2ph_hf8 — HF8 (E4M3), non-saturating */
__m512i _mm512_cvt2ph_hf8(__m512h a, __m512h b);
/* cvt2phs_bf8 — BF8 (E5M2), saturating */
__m512i _mm512_cvt2phs_bf8(__m512h a, __m512h b);
/* cvt2phs_hf8 — HF8 (E4M3), saturating */
__m512i _mm512_cvt2phs_hf8(__m512h a, __m512h b);
```

8.3 Convert FP32 Pair to FP16

8.3.1 Description

These operations convert from FP32 to FP16. They convert two vectors of packed FP32 format data into a single register of packed FP16 format data. These instructions do not support memory fault suppression.

MXCSR (and EVEX embedded rounding) determine the rounding mode. MXCSR.DAZ is respected on FP32 inputs. FTZ is not obeyed and is always assumed FTZ=0.

8.3.2 Operands

VCVT2PS2PHX reads two vectors of input and writes one vector of output.

Operand 1	Operand 2	Operand 3
xmm1{k1}{z}	xmm2	xmm3/m128/m32bcst
ymm1{k1}{z}	ymm2	ymm3/m256/m32bcst
zmm1{k1}{z}	zmm2	zmm3/m512/m32bcst {er}

Instruction	Operand 1	Operand 2	Operand 3
VCVT2PS2PHX	MODRM.REG(w)	VVVV(r)	MODRM.R/M(r)

8.3.3 SIMD Floating-Point Exceptions
No floating-point exceptions are generated; the instruction can set MXCSR[DE, IE, OE, PE, UE].

8.3.4 Other Exceptions
Exception class E2, see Section 5.2.

8.3.5 Operation

```
DEFINE vcvt2ps2phx(src1, src2, VL, k1, zeroing, no_writemask, src2_is_mem, evex_b):
    // Pair of FP32 vectors → single FP16 vector
    ASSERT VL in (128, 256, 512)
    KL = VL / 16
    orig_dest = copy(dest)
    FOR i = 0 TO KL-1:
        IF i < KL / 2:
            t = src2.fp32[0] if (src2_is_mem AND evex_b) else src2.fp32[i]
        ELSE:
            t = src1.fp32[i - KL / 2]
        IF k1[i] OR no_writemask:
            dest.word[i] = fp32_to_fp16(t)
        ELSE IF zeroing:
            dest.word[i] = 0
        ELSE:
            dest.word[i] = orig_dest.word[i]
    dest[MAXVL-1 : VL] = 0
```

8.3.6 Instruction Table

Mnemonic	Operands	Description
VCVT2PS2PHX	xmm1{k1}{z}, xmm2, xmm3/m128/m32bcst	Convert two packed FP32 vectors to one packed FP16 vector. Supports embedded rounding at 512-bit width.
	ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst	
	zmm1{k1}{z}, zmm2, zmm3/m512/m32bcst {er}	

8.3.7 C/C++ Compiler Intrinsic Equivalent

```
/* cvt2ps_phx — FP32-pair→FP16 */
__m512h _mm512_cvt2ps_phx(__m512 a, __m512 b);
```

8.4 Convert FP16 to FP8 with Bias

8.4.1 Description
These operations convert from FP16 to FP8 using a bias rounding term. They convert one vector of packed FP16 data into a single vector of packed FP8 format data.

MXCSR is not consulted and execution assumes all MXCSR exceptions are masked. DAZ is not obeyed and is always assumed DAZ=0. FTZ is not obeyed and is always assumed FTZ=0. No floating-point exceptions are generated and no status updated.

8.4.2 Operands
VCVTBIASPH2[B,H]F8[,S] reads two vectors of input and writes one vector of output.

Operand 1	Operand 2	Operand 3
xmm1{k1}{z}	xmm2	xmm3/m128/m16bcst
xmm1{k1}{z}	ymm2	ymm3/m256/m16bcst
ymm1{k1}{z}	zmm2	zmm3/m512/m16bcst

Instruction	Operand 1	Operand 2	Operand 3
VCVTBIASPH2BF8 VCVTBIASPH2BF8S VCVTBIASPH2HF8 VCVTBIASPH2HF8S	MODRM.REG(w)	VVVV(r)	MODRM.R/M(r)

8.4.3 SIMD Floating-Point Exceptions
None.

8.4.4 Other Exceptions
Exception class E4NF, see Section 5.3.

8.4.5 Operation

```
DEFINE vcvtbiasph2f8(src1, src2, dst_format, saturation_mode, VL, k1, zeroing, no_writemask,
src2_is_mem, evex_b):
    // FP16 → FP8 with stochastic rounding bias (src1 carries bias bytes)
    ASSERT VL in (128, 256, 512)
    KL = VL / 8
    orig_dest = copy(dest)
    FOR i = 0 TO KL / 2-1:
        t = src2.fp16[0] if (src2_is_mem AND evex_b) else src2.fp16[i]
        bias = src1.byte[2 * i]
        IF k1[i] OR no_writemask:
            IF dst_format == "E5M2":
                dest.byte[i] = fp16_to_fp8_e5m2(t, saturation_mode, "BIAS", bias)
            ELSE:
                dest.byte[i] = fp16_to_fp8_e4m3(t, saturation_mode, "BIAS", bias)
        ELSE IF zeroing:
            dest.byte[i] = 0
        ELSE:
            dest.byte[i] = orig_dest.byte[i]
    dest[MAXVL-1 : VL/2] = 0
```

8.4.6 Instruction Table

Mnemonic	Operands	Description
VCVTBIASPH2BF8	xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	Convert packed FP16 to FP8 E5M2 using bias rounding term, non-saturating. Bias rounding term is Operand 2.
	xmm1{k1}{z}, ymm2, ymm3/m256/m16bcst	
	ymm1{k1}{z}, zmm2, zmm3/m512/m16bcst	
VCVTBIASPH2BF8S	xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	Convert packed FP16 to FP8 E5M2 using bias rounding term, saturating. Bias rounding term is Operand 2.
	xmm1{k1}{z}, ymm2, ymm3/m256/m16bcst	
	ymm1{k1}{z}, zmm2, zmm3/m512/m16bcst	
VCVTBIASPH2HF8	xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	Convert packed FP16 to FP8 E4M3 using bias rounding term, non-saturating. Bias rounding term is Operand 2.
	xmm1{k1}{z}, ymm2, ymm3/m256/m16bcst	
	ymm1{k1}{z}, zmm2, zmm3/m512/m16bcst	
VCVTBIASPH2HF8S	xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	Convert packed FP16 to FP8 E4M3 using bias rounding term, saturating. Bias rounding term is Operand 2.
	xmm1{k1}{z}, ymm2, ymm3/m256/m16bcst	
	ymm1{k1}{z}, zmm2, zmm3/m512/m16bcst	

8.4.7 C/C++ Compiler Intrinsic Equivalent

```
/* cvtbiasph_bf8 — BF8 (E5M2), non-saturating */
__m256i _mm512_cvtbiasph_bf8(__m512i a, __m512h b);
/* cvtbiasph_hf8 — HF8 (E4M3), non-saturating */
__m256i _mm512_cvtbiasph_hf8(__m512i a, __m512h b);
/* cvtbiasphs_bf8 — BF8 (E5M2), saturating */
__m256i _mm512_cvtbiasphs_bf8(__m512i a, __m512h b);
/* cvtbiasphs_hf8 — HF8 (E4M3), saturating */
__m256i _mm512_cvtbiasphs_hf8(__m512i a, __m512h b);
```

8.5 Convert FP8 E4M3 to FP16

8.5.1 Description

This operation converts from FP8 E4M3 to FP16. MXCSR is ignored. DAZ is not obeyed and is always assumed DAZ=0. FTZ is not obeyed and is always assumed FTZ=0. The conversion is always precise, and no rounding is needed. No floating-point exceptions are generated and no status updated.

8.5.2 Operands

Operand 1	Operand 2	Operand 3
xmm1{k1}{z}	xmm2/m64	
ymm1{k1}{z}	xmm2/m128	
zmm1{k1}{z}	ymm2/m256	

Instruction	Operand 1	Operand 2	Operand 3
VCVTHF82PH	MODRM.REG(w)	MODRM.R/M(r)	N/A

8.5.3 SIMD Floating-Point Exceptions

None.

8.5.4 Other Exceptions

Exception class E2, see Section 5.2.

8.5.5 Operation

```
DEFINE vcvthf82ph(src2, VL, k1, zeroing, no_writemask, src2_is_mem, evex_b):
    // FP8 E4M3 → FP16
    ASSERT VL in (128, 256, 512)
    KL = VL / 16
    orig_dest = copy(dest)
    FOR i = 0 TO KL-1:
        t = src2.hf8[0] if (src2_is_mem AND evex_b) else src2.hf8[i]
        IF k1[i] OR no_writemask:
            dest.fp16[i] = fp8_e4m3_to_fp16(t)
        ELSE IF zeroing:
            dest.fp16[i] = 0
        ELSE:
            dest.fp16[i] = orig_dest.fp16[i]
    dest[MAXVL-1 : VL] = 0
```

8.5.6 Instruction Table

Mnemonic	Operands	Description
VCVTHF82PH	xmm1{k1}{z}, xmm2/m64 ymm1{k1}{z}, xmm2/m128 zmm1{k1}{z}, ymm2/m256	Convert packed FP8 E4M3 to packed FP16. Conversion is exact; no rounding required.

8.5.7 C/C++ Compiler Intrinsic Equivalent

```
/* cvthf8_ph — FP8 E4M3→FP16 */
__m512h _mm512_cvthf8_ph(__m256i a);
```

8.6 Multiply and Add Bytes (EVEX)

8.6.1 Description

Multiplies the individual bytes of the first source operand by the corresponding bytes of the second source operand, producing intermediate word results. The word results are then summed and accumulated in the destination dword element size operand.

8.6.2 Operands

Operand 1	Operand 2	Operand 3
xmm1 {k1}{z}	xmm2	xmm3/m128/m32bcst
ymm1 {k1}{z}	ymm2	ymm3/m256/m32bcst
zmm1 {k1}{z}	zmm2	zmm3/m512/m32bcst

Instruction	Operand 1	Operand 2	Operand 3
VPDPBSSD VPDPBSSDS VPDPBSUD VPDPBSUDS VPDPBUUD VPDPBUUDS	MODRM.REG(rw)	VVVV(r)	MODRM.R/M(r)

8.6.3 SIMD Floating-Point Exceptions

None.

8.6.4 Other Exceptions

Exception class E4, see Section 5.3.

8.6.5 Operation

```
DEFINE vpdpb_d(dest, src1, src2, VL, k1, zeroing, no_writemask, src1_signed, src2_signed,
saturating, src2_is_mem, evex_b):
    // Byte VNNI: 4 byte dot products per dword lane, INT32 accumulation.
    ASSERT VL in (128, 256, 512)
    KL = VL / 32  // number of dword accumulator lanes
    orig_dest = copy(dest)
    FOR i = 0 TO KL-1:
        IF k1[i] OR no_writemask:
            t = src2.dword[0] if (src2_is_mem AND evex_b) else src2.dword[i]
            // 8-bit → 16-bit sign/zero extension before multiplication
            extend1 = sign_extend8 if src1_signed else zero_extend8
            extend2 = sign_extend8 if src2_signed else zero_extend8
            p1 = extend1(src1.byte[4*i+0]) * extend2(t.byte[0])  // word product
            p2 = extend1(src1.byte[4*i+1]) * extend2(t.byte[1])
            p3 = extend1(src1.byte[4*i+2]) * extend2(t.byte[2])
            p4 = extend1(src1.byte[4*i+3]) * extend2(t.byte[3])
            total = orig_dest.dword[i] + p1 + p2 + p3 + p4
            IF saturating:
                dest.dword[i] = (unsigned_dword_saturate(total)
                                 if NOT src1_signed AND NOT src2_signed
                                 else signed_dword_saturate(total))
            ELSE:
                dest.dword[i] = total & UINT32_MAX  // INT32 truncation
        ELSE IF zeroing:
            dest.dword[i] = 0
        ELSE:
            dest.dword[i] = orig_dest.dword[i]
    dest[MAX_VL-1 : VL] = 0
```

8.6.6 Instruction Table

Mnemonic	Operands	Description
VPDPBSSD	xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	Multiply signed byte x signed byte pairs, sum adjacent 16-bit products, accumulate to 32-bit dword.
VPDPBSSDS	xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	Multiply signed byte x signed byte pairs, sum adjacent 16-bit products,

Mnemonic	Operands	Description
		accumulate to 32-bit dword with signed saturation.
VPDPBSUD	xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	Multiply signed byte x unsigned byte pairs, sum adjacent 16-bit products, accumulate to 32-bit dword.
	ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	
	zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	
VPDPBSUDS	xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	Multiply signed byte x unsigned byte pairs, sum adjacent 16-bit products, accumulate to 32-bit dword with signed saturation.
	ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	
	zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	
VPDPBUUD	xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	Multiply unsigned byte x unsigned byte pairs, sum adjacent 16-bit products, accumulate to 32-bit dword.
	ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	
	zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	
VPDPBUUDS	xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	Multiply unsigned byte x unsigned byte pairs, sum adjacent 16-bit products, accumulate to 32-bit dword with unsigned saturation.
	ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	
	zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	

8.6.7 C/C++ Compiler Intrinsic Equivalent

```
/* dpbssd — signed A × signed B */
__m512i _mm512_dpbssd(__m512i a, __m512i b, __m512i c);
/* dpbssds — signed A × signed B, saturating */
__m512i _mm512_dpbssds(__m512i a, __m512i b, __m512i c);
/* dpbsud — signed A × unsigned B */
__m512i _mm512_dpbsud(__m512i a, __m512i b, __m512i c);
/* dpbsuds — signed A × unsigned B, saturating */
__m512i _mm512_dpbsuds(__m512i a, __m512i b, __m512i c);
/* dpbuud — unsigned A × unsigned B */
__m512i _mm512_dpbuud(__m512i a, __m512i b, __m512i c);
/* dpbuuds — unsigned A × unsigned B, saturating */
__m512i _mm512_dpbuuds(__m512i a, __m512i b, __m512i c);
```

8.7 Multiply and Add Words (EVEX)

8.7.1 Description

Multiplies the individual words of the first source operand by the corresponding words of the second source operand, producing intermediate dword results. The dword results are then summed and accumulated in the destination dword element size operand.

8.7.2 Operands

Operand 1	Operand 2	Operand 3
xmm1 {k1}{z}	xmm2	xmm3/m128/m32bcst
ymm1 {k1}{z}	ymm2	ymm3/m256/m32bcst
zmm1 {k1}{z}	zmm2	zmm3/m512/m32bcst

Instruction	Operand 1	Operand 2	Operand 3
VPDPWSUD VPDPWSUDS VPDPWUSD VPDPWUSDS VPDPWUUD VPDPWUUDS	MODRM.REG(rw)	VVVV(r)	MODRM.R/M(r)

8.7.3 SIMD Floating-Point Exceptions

None.

8.7.4 Other Exceptions

Exception class E4, see Section 5.3.

8.7.5 Operation

```
DEFINE vpdpw_d(dest, src1, src2, VL, k1, zeroing, no_writemask, src1_signed, src2_signed,
saturating, src2_is_mem, evex_b):
    // Word VNNI: 2 word dot products per dword lane, INT32 accumulation.
    ASSERT VL in (128, 256, 512)
    KL = VL / 32
    orig_dest = copy(dest)
    // Sign-extension decision is per-instruction (made once, outside loop)
    extend1 = sign_extend16 if src1_signed else zero_extend16
    extend2 = sign_extend16 if src2_signed else zero_extend16
    FOR i = 0 TO KL-1:
        IF k1[i] OR no_writemask:
            t = src2.dword[0] if (src2_is_mem AND evex_b) else src2.dword[i]
            // 16-bit → 32-bit extension before multiplication
            p1 = extend1(src1.word[2*i+0]) * extend2(t.word[0])  // dword product
            p2 = extend1(src1.word[2*i+1]) * extend2(t.word[1])
            total = orig_dest.dword[i] + p1 + p2
            IF saturating:
                dest.dword[i] = (unsigned_dword_saturate(total)
                                 if NOT src1_signed AND NOT src2_signed
                                 else signed_dword_saturate(total))
            ELSE:
                dest.dword[i] = total & UINT32_MAX
        ELSE IF zeroing:
            dest.dword[i] = 0
        ELSE:
            dest.dword[i] = orig_dest.dword[i]
    dest[MAX_VL-1 : VL] = 0
```

8.7.6 Instruction Table

Mnemonic	Operands	Description
VPDPWSUD	xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	Multiply signed word x unsigned word pairs, sum 32-bit products, accumulate to 32-bit dword.
	ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	
	zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	
VPDPWSUDS	xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	Multiply signed word x unsigned word pairs, sum 32-bit products, accumulate to 32-bit dword with signed saturation.
	ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	
	zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	
VPDPWUSD	xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	Multiply unsigned word x signed word pairs, sum 32-bit products, accumulate to 32-bit dword.
	ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	
	zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	
VPDPWUSDS	xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	Multiply unsigned word x signed word pairs, sum 32-bit products, accumulate to 32-bit dword with signed saturation.
	ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	
	zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	
VPDPWUUD	xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	Multiply unsigned word x unsigned word pairs, sum 32-bit products, accumulate to 32-bit dword.
	ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	
	zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	
VPDPWUUDS	xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	Multiply unsigned word x unsigned word pairs, sum 32-bit products, accumulate to 32-bit dword with unsigned saturation.
	ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	
	zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	

8.7.7 C/C++ Compiler Intrinsic Equivalent

```
/* dpwsud — signed A × unsigned B (word) */
__m512i _mm512_dpwsud(__m512i a, __m512i b, __m512i c);
/* dpwsuds — signed A × unsigned B (word), saturating */
__m512i _mm512_dpwsuds(__m512i a, __m512i b, __m512i c);
/* dpwusd — unsigned A × signed B (word) */
__m512i _mm512_dpwusd(__m512i a, __m512i b, __m512i c);
/* dpwusds — unsigned A × signed B (word), saturating */
__m512i _mm512_dpwusds(__m512i a, __m512i b, __m512i c);
/* dpwuud — unsigned A × unsigned B (word) */
__m512i _mm512_dpwuud(__m512i a, __m512i b, __m512i c);
/* dpwuuds — unsigned A × unsigned B (word), saturating */
__m512i _mm512_dpwuuds(__m512i a, __m512i b, __m512i c);
```

9 OCP Format Conversion Functions

9.1 Introduction

Implementations compliant with the ACE v1 architecture must implement the format conversion instructions specified in this section. The instructions in this section are enumerated by AVX10_V2_AUX.

From	Encoding	To	Encoding	Rounding	Saturation	Section
FP32	SE8M23	FP8 E4M3	SE4M3	RNE, RTO, BIAS	SAT, NSAT	Convert from FP32 to FP8
		FP8 E5M2	SE5M2	RNE, BIAS	SAT, NSAT	Convert from FP32 to FP8
FP8 E4M3	SE4M3	FP32	SE8M23	–	–	Convert from FP8 to FP32
		FP4	SE2M1	RNE	SAT	Convert from FP8 to FP4
		FP6	SE2M3	RNE	SAT	Convert from FP8 to FP6
FP8 E5M2	SE5M2	FP32	SE8M23	–	–	Convert from FP8 to FP32
		FP4	SE2M1	RNE	SAT	Convert from FP8 to FP4
		FP6	SE3M2	RNE	SAT	Convert from FP8 to FP6
FP4	SE2M1	FP8 E4M3	SE4M3	–	–	Convert from FP4 to FP8
FP6 E2M3	SE2M3	FP8 E4M3	SE4M3	–	–	Convert from FP6 to FP8
FP6 E3M2	SE3M2	FP8 E4M3	SE4M3	–	–	Convert from FP6 to FP8
INT32	–	INT8	–	–	SSAT	Down Convert DWord to Byte with Symmetric Signed Saturation

9.2 Convert from FP32 to FP8

From	To	Saturate mode	Rounding mode
FP32	FP8 E5M2	SAT, NSAT	RNE, BIAS
FP32	FP8 E4M3	SAT, NSAT	RNE, RTO, BIAS

9.2.1 Description

These operations convert from FP32 to FP8; for a specific convert operation, the target format is the same for each data element processed. The convert operations support saturating and non-saturating modes that modify the handling of numbers (after rounding) above the maximum exponent magnitude of the target format. MXCSR is not consulted nor updated. DAZ is not obeyed and is always assumed DAZ=1. FTZ is not obeyed and is always assumed FTZ=0. No floating-point exceptions are generated, and status is not updated.

The destination format may be E5M2 (BF8) or E4M3 (HF8); saturating or non-saturating modes may be selected. The saturation modes follow the saturate (SAT) and non-saturate (NONSAT) behaviors described in the table below. Rounding modes include round to nearest even, bias rounding, and for E4M3, round to odd. The rounding modes supported are determined by the destination FP8 format.

Source value (after rounding)	E5M2 (BF8)		E4M3 (HF8)	
SAT	NONSAT	SAT	NONSAT	NaN
NaN	NaN	NaN	NaN	±Inf
±max_E5M2	±Inf	±max_E4M3	NaN	Greater than max OFP8 magnitude
±max_E5M2	±Inf	±max_E4M3	NaN	In OFP8 range
Rounded value	Rounded value	Rounded value	Rounded value	Smaller than min OFP8 subnormal magnitude
±0	±0	±0	±0	±0
±0	±0	±0	±0	

Rounding Mode	Target format		Notes
	FP8 E4M3	FP8 E5M2	
RNE	Y	Y	Round to Nearest Even. Default IEEE rounding mode, default mode under OCP MX standard.
RTO	Y	N	Round to Odd. Facilitates later narrowing and rounding to a smaller number format with only one rounding error.
BIAS	Y	Y	Bias Rounding. May be used to implement stochastic rounding by using a random number as the bias term. Often found useful for deep learning applications using low precision computation. The bias rounding term is provided as an additional input to the instruction.

9.2.2 Operands

Note: For Bias rounding, the bias term is contained in Operand 2.

Form	Operand 1	Operand 2	Operand 3
RNE / RTO	xmm1 {k1}{z}	xmm2/m128/m32bcst	
	xmm1 {k1}{z}	ymm2/m256/m32bcst	
	xmm1 {k1}{z}	zmm2/m512/m32bcst	
Bias	xmm1 {k1}{z}	xmm2	xmm3/m128/m32bcst
	xmm1 {k1}{z}	ymm2	ymm3/m256/m32bcst
	xmm1 {k1}{z}	zmm2	zmm3/m512/m32bcst

Instruction	Operand 1	Operand 2	Operand 3
VCVTPS2BF8 VCVTPS2BF8S VCVTPS2HF8 VCVTPS2HF8S VCVTROPS2HF8 VCVTROPS2HF8S	MODRM.REG(w)	MODRM.R/M(r)	N/A
VCVTBIASPS2BF8 VCVTBIASPS2BF8S VCVTBIASPS2HF8 VCVTBIASPS2HF8S	MODRM.REG(w)	VVVV(r)	MODRM.R/M(r)

9.2.3 SIMD Floating-Point Exceptions

None.

9.2.4 Other Exceptions

Exception class E4, see Section 5.3.

9.2.5 Operation

RNE / RTO rounding (2-operand form):

```
DEFINE vcvt_ps2f8(src, dst_format, saturation_mode, rounding_mode, VL, k1, zeroing,
no_writemask, src_is_mem, evex_b):
    ASSERT VL in (128, 256, 512)
    KL = VL / 8
    orig_dest = copy(dest)
    FOR i = 0 TO KL / 4-1:
        t = src.fp32[0] if (src_is_mem AND evex_b) else src.fp32[i]
        IF k1[i] OR no_writemask:
            IF dst_format == "E5M2":
                dest.byte[i] = fp32_to_fp8_e5m2(t, saturation_mode, rounding_mode)
            ELSE:
                dest.byte[i] = fp32_to_fp8_e4m3(t, saturation_mode, rounding_mode)
        ELSE IF zeroing:
            dest.byte[i] = 0
        ELSE:
            dest.byte[i] = orig_dest.byte[i]
    dest[MAXVL-1 : VL/4] = 0
```

Biased rounding (3-operand form — VCVTBIASPS2BF8/BF8S/HF8/HF8S):

```
DEFINE vcvtbiasps2f8(src1, src2, dst_format, saturate: bool, VL, k1, zeroing, no_writemask,
src2_is_mem, evex_b):
    ASSERT VL in (128, 256, 512)
    KL = VL / 8
    KO = KL / 4  // number of FP32→FP8 output elements (4:1 narrowing)
    orig_dest = copy(dest)
    FOR i = 0 TO KO-1:
        t    = src2.fp32[0] if (src2_is_mem AND evex_b) else src2.fp32[i]
        bias = src1.dword[i]  // OPND2: per-element bias dword, one per FP32 lane
        sat  = "SAT" if saturate else "NOSAT"
        IF k1[i] OR no_writemask:
            IF dst_format == "E5M2":
                dest.byte[i] = fp32_to_fp8_e5m2(t, sat, "BIAS", bias)
            ELSE:
                dest.byte[i] = fp32_to_fp8_e4m3(t, sat, "BIAS", bias)
        ELSE IF zeroing:
            dest.byte[i] = 0
        ELSE:
            dest.byte[i] = orig_dest.byte[i]
    dest[MAXVL-1 : VL/4] = 0
```

9.2.6 Instruction Table

Mnemonic	Operands	Description
VCVTPS2BF8	xmm1 {k1}{z}, xmm2/m128/m32bcst	Convert packed FP32 to FP8 E5M2 using RNE rounding, non-saturating.
	xmm1 {k1}{z}, ymm2/m256/m32bcst	
	xmm1 {k1}{z}, zmm2/m512/m32bcst	
VCVTPS2BF8S	xmm1 {k1}{z}, xmm2/m128/m32bcst	Convert packed FP32 to FP8 E5M2 using RNE rounding, saturating.
	xmm1 {k1}{z}, ymm2/m256/m32bcst	
	xmm1 {k1}{z}, zmm2/m512/m32bcst	
VCVTPS2HF8	xmm1 {k1}{z}, xmm2/m128/m32bcst	Convert packed FP32 to FP8 E4M3 using RNE rounding, non-saturating.
	xmm1 {k1}{z}, ymm2/m256/m32bcst	
	xmm1 {k1}{z}, zmm2/m512/m32bcst	
VCVTPS2HF8S	xmm1 {k1}{z}, xmm2/m128/m32bcst	Convert packed FP32 to FP8 E4M3 using RNE rounding, saturating.
	xmm1 {k1}{z}, ymm2/m256/m32bcst	
	xmm1 {k1}{z}, zmm2/m512/m32bcst	
VCVTROPS2HF8	xmm1 {k1}{z}, xmm2/m128/m32bcst	
	xmm1 {k1}{z}, ymm2/m256/m32bcst	

Mnemonic	Operands	Description
	xmm1 {k1}{z}, zmm2/m512/m32bcst	Convert packed FP32 to FP8 E4M3 using RTO rounding, non-saturating.
VCVTROPS2HF8S	xmm1 {k1}{z}, xmm2/m128/m32bcst	Convert packed FP32 to FP8 E4M3 using RTO rounding, saturating.
	xmm1 {k1}{z}, ymm2/m256/m32bcst	
	xmm1 {k1}{z}, zmm2/m512/m32bcst	
VCVTBIASPS2BF8	xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	Convert packed FP32 to FP8 E5M2 using bias rounding, non-saturating. Bias rounding term is Operand 2.
	xmm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	
	xmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	
VCVTBIASPS2BF8S	xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	Convert packed FP32 to FP8 E5M2 using bias rounding, saturating. Bias rounding term is Operand 2.
	xmm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	
	xmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	
VCVTBIASPS2HF8	xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	Convert packed FP32 to FP8 E4M3 using bias rounding, non-saturating. Bias rounding term is Operand 2.
	xmm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	
	xmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	
VCVTBIASPS2HF8S	xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	Convert packed FP32 to FP8 E4M3 using bias rounding, saturating. Bias rounding term is Operand 2.
	xmm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	
	xmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	

9.2.7 C/C++ Compiler Intrinsic Equivalent

```
/* cvtps_bf8 — BF8 (E5M2), RNE */
__m256i _mm512_cvtps_bf8(__m512 a);
/* cvtpss_bf8 — BF8 (E5M2), RNE, saturating */
__m256i _mm512_cvtpss_bf8(__m512 a);
/* cvtps_hf8 — HF8 (E4M3), RNE */
__m256i _mm512_cvtps_hf8(__m512 a);
/* cvtpss_hf8 — HF8 (E4M3), RNE, saturating */
__m256i _mm512_cvtpss_hf8(__m512 a);
/* cvtrops_hf8 — HF8 (E4M3), round-to-odd */
__m256i _mm512_cvtrops_hf8(__m512 a);
/* cvtropss_hf8 — HF8 (E4M3), round-to-odd, saturating */
__m256i _mm512_cvtropss_hf8(__m512 a);
/* cvtbiasps_bf8 — BF8 (E5M2), bias-scaled */
__m256i _mm512_cvtbiasps_bf8(__m512 a, __m512i b);
/* cvtbiaspss_bf8 — BF8 (E5M2), bias-scaled, saturating */
__m256i _mm512_cvtbiaspss_bf8(__m512 a, __m512i b);
/* cvtbiasps_hf8 — HF8 (E4M3), bias-scaled */
__m256i _mm512_cvtbiasps_hf8(__m512 a, __m512i b);
/* cvtbiaspss_hf8 — HF8 (E4M3), bias-scaled, saturating */
__m256i _mm512_cvtbiaspss_hf8(__m512 a, __m512i b);
```

9.3 Convert from FP8 to FP32

9.3.1 Description

These operations convert from FP8 to FP32. All FP8 encodings have a precise mapping to a single FP32 encoding; no rounding is required. MXCSR is not consulted nor updated. DAZ is not obeyed and is always assumed DAZ=0. FTZ is not obeyed and is always assumed FTZ=0. No floating-point exceptions are generated and no status updated. Converts are provided from both OCP FP8 encoding formats.

Note: Results written to destination registers smaller than ZMM are achieved through truncation.

9.3.2 Operands

Operand 1	Operand 2	Operand 3
xmm1 {k1}{z}	xmm2/m32	
ymm1 {k1}{z}	xmm2/m64	
zmm1 {k1}{z}	xmm2/m128	

Instruction	Operand 1	Operand 2	Operand 3
VCVTBF82PS VCVTHF82PS	MODRM.REG(w)	MODRM.R/M(r)	N/A

9.3.3 SIMD Floating-Point Exceptions

None.

9.3.4 Other Exceptions

Exception class E4, see Section 5.3.

9.3.5 Operation

```
DEFINE vcvtf82ps(src, dst_format, VL, k1, zeroing, no_writemask):
    // FP8 → FP32
    ASSERT VL in (128, 256, 512)
    KL = VL / 32
    orig_dest = copy(dest)
    FOR i = 0 TO KL-1:
        t = src.byte[i]
        IF k1[i] OR no_writemask:
            IF dst_format == "E5M2":
                dest.dword[i] = fp8_e5m2_to_fp32(t)
            ELSE:
                dest.dword[i] = fp8_e4m3_to_fp32(t)
        ELSE IF zeroing:
            dest.dword[i] = 0
        ELSE:
            dest.dword[i] = orig_dest.dword[i]
    dest[MAXVL-1 : VL] = 0
```

9.3.6 Instruction Table

Mnemonic	Operands	Description
VCVTBF82PS	xmm1 {k1}{z}, xmm2/m32 ymm1 {k1}{z}, xmm2/m64 zmm1 {k1}{z}, xmm2/m128	Convert packed FP8 E5M2 to packed FP32. Conversion is exact; output register is 4x wider than input.
VCVTHF82PS	xmm1 {k1}{z}, xmm2/m32 ymm1 {k1}{z}, xmm2/m64 zmm1 {k1}{z}, xmm2/m128	Convert packed FP8 E4M3 to packed FP32. Conversion is exact; output register is 4x wider than input.

9.3.7 C/C++ Compiler Intrinsic Equivalent

```
/* cvtbf8_ps — BF8 (E5M2), non-saturating */
__m512 _mm512_cvtbf8_ps(__m128i a);
/* cvthf8_ps — HF8 (E4M3), non-saturating */
__m512 _mm512_cvthf8_ps(__m128i a);
```

9.4 Convert from FP8 to FP4

9.4.1 Description

These operations convert from FP8 to FP4. The operation rounds the FP8 source data using RNE, with saturation for source data larger than the expressible range of the target format. MXCSR is not consulted nor updated. DAZ is not obeyed and is always assumed DAZ=0. FTZ is not obeyed and is always assumed FTZ=0. No floating-point exceptions are generated and no

status updated. Convert operations are provided for both of the OCP FP8 formats. No masking or zeroing support is provided.

Input FP8 data will be 8-bit packed in the source register. Output FP4 data will be nibble packed in the destination register or memory operand.

Note: Register results smaller than XMM are achieved through truncation.

Note: All FP8 subnormal inputs lie well below the RNE midpoint to the smallest representable target subnormal, and therefore round to ± 0.

9.4.2 Operands

Operand 1	Operand 2	Operand 3
xmm1/m64	xmm2	
xmm1/m128	ymm2	
ymm1/m256	zmm2	

Instruction	Operand 1	Operand 2	Operand 3
VCVTBF82BF4S VCVTHF82BF4S	MODRM.R/M(w)	MODRM.REG(r)	N/A

9.4.3 SIMD Floating-Point Exceptions

None.

9.4.4 Other Exceptions

Exception class E4NF, see Section 5.3.

9.4.5 Operation

```
DEFINE vcvtf82bf4s(src, dst_format, VL, dst_is_mem=False):
    // FP8 → FP4 (nibble-packed output). No writemask support.
    // When dst_is_mem: only VL//16 bytes written to memory; no upper-bits zeroing.
    ASSERT VL in (128, 256, 512)
    KL = VL / 8
    FOR i = 0 TO KL-1:
        t = src.byte[i]
        IF dst_format == "E5M2":
            dest[4*i+3 : 4*i] = fp8_e5m2_to_fp4(t)
        ELSE:
            dest[4*i+3 : 4*i] = fp8_e4m3_to_fp4(t)
    IF NOT dst_is_mem:
        dest[MAXVL-1 : VL/2] = 0
```

9.4.6 Instruction Table

Mnemonic	Operands	Description
VCVTBF82BF4S	xmm1/m64, xmm2 xmm1/m128, ymm2 ymm1/m256, zmm2	Convert packed FP8 E5M2 to packed FP4 E2M1, saturating. Output register is half the width of input.
VCVTHF82BF4S	xmm1/m64, xmm2 xmm1/m128, ymm2 ymm1/m256, zmm2	Convert packed FP8 E4M3 to packed FP4 E2M1, saturating. Output register is half the width of input.

9.4.7 C/C++ Compiler Intrinsic Equivalent

```
/* cvtf8_bf4s — FP8 E5M2→FP4 E2M1; no mask support */
__m256i _mm512_cvtf8_bf4s(__m512i a);
```

9.5 Convert from FP4 to FP8

9.5.1 Description

This operation converts from FP4 to FP8. All FP4 encodings have a precise mapping to a single FP8 encoding; no rounding or approximation is necessary. MXCSR is not consulted nor updated. DAZ is not obeyed and is always assumed DAZ=0. FTZ is not obeyed and is always assumed FTZ=0. No floating-point exceptions are generated and no status updated.

9.5.2 Operands

Operand 1	Operand 2	Operand 3
xmm1 {k1}{z}	xmm2/m64	
ymm1 {k1}{z}	xmm2/m128	
zmm1 {k1}{z}	ymm2/m256	

Instruction	Operand 1	Operand 2	Operand 3
VCVTBF42HF8	MODRM.REG(w)	MODRM.R/M(r)	N/A

9.5.3 SIMD Floating-Point Exceptions

None.

9.5.4 Other Exceptions

Exception class E4NF, see Section 5.3.

9.5.5 Operation

```
DEFINE vcvtbf42hf8(src, VL, k1, zeroing, no_writemask):
    // FP4 (nibble-packed) → FP8 E4M3
    ASSERT VL in (128, 256, 512)
    KL = VL / 8
    orig_dest = copy(dest)
    FOR i = 0 TO KL-1:
        t = src[4*i+3 : 4*i]
        IF k1[i] OR no_writemask:
            dest.byte[i] = fp4_to_fp8_e4m3(t)
        ELSE IF zeroing:
            dest.byte[i] = 0
        ELSE:
            dest.byte[i] = orig_dest.byte[i]
    dest[MAXVL-1 : VL] = 0
```

9.5.6 Instruction Table

Mnemonic	Operands	Description
VCVTBF42HF8	xmm1 {k1}{z}, xmm2/m64	Convert packed FP4 E2M1 to packed FP8 E4M3. Conversion is exact; output register is 2x wider than input.
	ymm1 {k1}{z}, xmm2/m128	
	zmm1 {k1}{z}, ymm2/m256	

9.5.7 C/C++ Compiler Intrinsic Equivalent

```
/* cvtbf4_hf8 — FP4 E2M1→FP8 E4M3 */
__m512i _mm512_cvtbf4_hf8(__m256i a);
```

9.6 Convert from FP8 to FP6

9.6.1 Description

These operations convert from FP8 to FP6 formats. The operation rounds the FP8 source data using RNE, with saturation for source data larger than the expressible range of the target format. MXCSR is not consulted nor updated. DAZ is not obeyed and is always assumed DAZ=0. FTZ is not obeyed and is always assumed FTZ=0. No floating-point exceptions are generated and no status updated. No masking or zeroing support is provided.

Note: The supported conversions have been selected to match source and target mantissa width to minimize rounding effects. All FP8 subnormal inputs lie well below the RNE midpoint to the smallest representable target subnormal, and therefore round to ± 0.

- FP8 E5M2 to FP6 E3M2
- FP8 E4M3 to FP6 E2M3

9.6.2 Operands

Operand 1	Operand 2	Operand 3
xmm1	xmm2	
ymm1	ymm2	
zmm1	zmm2	

Instruction	Operand 1	Operand 2	Operand 3
VCVTBF82BF6S VCVTHF82HF6S	MODRM.R/M(w)	MODRM.REG(r)	N/A

9.6.3 SIMD Floating-Point Exceptions

None.

9.6.4 Other Exceptions

Exception class E7NM, see Section 5.5.

9.6.5 Operation

```
DEFINE vcvtf82f6s(src, dst_format, VL):
    // FP8 → FP6 (6-bit packed output). No writemask support.
    ASSERT VL in (128, 256, 512)
    KL = VL / 8
    FOR i = 0 TO KL-1:
        t = src.byte[i]
        IF dst_format == "E3M2":
            dest[6*i+5 : 6*i] = fp8_e5m2_to_fp6_e3m2(t)
        ELSE:
            dest[6*i+5 : 6*i] = fp8_e4m3_to_fp6_e2m3(t)
    dest[MAXVL-1 : VL*6/8] = 0
```

9.6.6 Instruction Table

Mnemonic	Operands	Description
VCVTBF82BF6S	xmm1, xmm2 ymm1, ymm2 zmm1, zmm2	Convert packed FP8 E5M2 to packed FP6 E3M2, saturating.
VCVTHF82HF6S	xmm1, xmm2 ymm1, ymm2 zmm1, zmm2	Convert packed FP8 E4M3 to packed FP6 E2M3, saturating.

9.6.7 C/C++ Compiler Intrinsic Equivalent

```
/* cvtf8_bf6s — FP8 E5M2→FP6 E3M2 */
__m512i _mm512_cvtf8_bf6s(__m512i a);
```

9.7 Convert from FP6 to FP8

9.7.1 Description

These operations convert from FP6 to FP8. All FP6 encodings have a precise mapping to a single FP8 encoding; no rounding or approximation is necessary. MXCSR is not consulted nor updated. DAZ is not obeyed and is always assumed DAZ=0. FTZ is not obeyed and is always assumed FTZ=0. No floating-point exceptions are generated and no status updated. There are two convert variants (one per format).

9.7.2 Operands

Operand 1	Operand 2	Operand 3
xmm1 {k1}{z}	xmm2	
ymm1 {k1}{z}	ymm2	
zmm1 {k1}{z}	zmm2	

Instruction	Operand 1	Operand 2	Operand 3
VCVTBF62HF8 VCVTHF62HF8	MODRM.REG(w)	MODRM.R/M(r)	N/A

9.7.3 SIMD Floating-Point Exceptions

None.

9.7.4 Other Exceptions

Exception class E7NM, see Section 5.5.

9.7.5 Operation

```
DEFINE vcvtf62hf8(src, src_format, VL, k1, zeroing, no_writemask):
    // FP6 (6-bit packed, register-only source) → FP8 E4M3.
    ASSERT VL in (128, 256, 512)
    KL = VL / 8
    orig_dest = copy(dest)
    FOR i = 0 TO KL-1:
        t = src[6*i+5 : 6*i]
        IF k1[i] OR no_writemask:
            IF src_format == "E3M2":
                dest.byte[i] = fp6_e3m2_to_fp8_e4m3(t)
            ELSE:
                dest.byte[i] = fp6_e2m3_to_fp8_e4m3(t)
        ELSE IF zeroing:
            dest.byte[i] = 0
        ELSE:
            dest.byte[i] = orig_dest.byte[i]
    dest[MAXVL-1 : VL] = 0
```

9.7.6 Instruction Table

Mnemonic	Operands	Description
VCVTBF62HF8	xmm1 {k1}{z}, xmm2 ymm1 {k1}{z}, ymm2 zmm1 {k1}{z}, zmm2	Convert packed FP6 E3M2 to packed FP8 E4M3. Conversion is exact.
VCVTHF62HF8	xmm1 {k1}{z}, xmm2 ymm1 {k1}{z}, ymm2 zmm1 {k1}{z}, zmm2	Convert packed FP6 E2M3 to packed FP8 E4M3. Conversion is exact.

9.7.7 C/C++ Compiler Intrinsic Equivalent

```
/* cvtf6_hf8 — FP6 E3M2→FP8 E4M3 */
__m512i _mm512_cvtf6_hf8(__m512i a);
```

9.8 Down Convert DWord to Byte with Symmetric Signed Saturation

9.8.1 Description

VPMOVSSDB converts signed 32-bit integers into packed signed bytes using symmetric signed saturation.

Symmetric saturation means that the saturated result is balanced around 0:

- MAX_POSITIVE: 0x7F (+127)
- MAX_NEGATIVE: 0x81 (-127)

The source operand is a ZMM/YMM/XMM register; the destination operand is an XMM register or 128/64/32-bit memory destination. Byte masking is supported on the destination operand, with zeroing also available for register destination.

9.8.2 Operands

Operand 1	Operand 2	Operand 3
xmm1/m32 {k1}{z}	xmm2	
xmm1/m64 {k1}{z}	ymm2	
xmm1/m128 {k1}{z}	zmm2	

Instruction	Operand 1	Operand 2	Operand 3
VPMOVSSDB	MODRM.R/M(w)	MODRM.REG(r)	N/A

9.8.3 SIMD Floating-Point Exceptions

None.

9.8.4 Other Exceptions

Exception class E6, see Section 5.4. Additionally: `EVEX.vvvv` is reserved, #UD if `EVEX.vvvv != 1111B`.

9.8.5 Operation

```
DEFINE vpmovssdb(src, VL, k1, zeroing, no_writemask, dst_is_mem=False):
    // INT32 → INT8 with symmetric saturation to [-127, 127]
    // For memory destination: masked-off lanes are not written (merge semantics always).
    // Zeroing masking ({z}) with memory destination is #UD per EVEX background rules.
    ASSERT VL in (128, 256, 512)
    KL = VL / 32
    orig_dest = copy(dest)
    FOR i = 0 TO KL-1:
        IF k1[i] OR no_writemask:
            dest.byte[i] = saturate_int32_to_symmetric_int8(src.dword[i])
        ELSE IF zeroing AND NOT dst_is_mem:
            dest.byte[i] = 0
        ELSE:
            dest.byte[i] = orig_dest.byte[i]
    IF NOT dst_is_mem:
        dest[MAXVL-1 : VL/4] = 0
```

9.8.6 Instruction Table

Mnemonic	Operands	Description
VPMOVSSDB	xmm1/m32 {k1}{z}, xmm2	Truncate packed 32-bit signed integers to 8-bit with symmetric signed saturation, storing to destination register or memory.
	xmm1/m64 {k1}{z}, ymm2	
	xmm1/m128 {k1}{z}, zmm2	

9.8.7 C/C++ Compiler Intrinsic Equivalent

```
/* cvtssepi32_epi8 — Symmetric signed saturation INT32→INT8 */
__m128i _mm512_cvtssepi32_epi8(__m512i a);
```

Note: The `cvtss` prefix (convert symmetric-saturate) distinguishes these from existing `_mm{w}_cvtsepi32_epi8` family (VPMOVDB, ordinary signed saturation, clamp range [−128, +127]). VPMOVSSDB uses symmetric saturation, clamping to [−127, +127] to preserve two's complement sign symmetry.

9.9 Unpack to Byte (VUNPACKB)

9.9.1 Description

This operation unpacks a vector of packed data of the specified size into 8-bit lanes. The number of items unpacked is equal to the vector length divided by 8.

Packed data should be aligned with the first element starting at bit 0 of the input vector. For supported element sizes, a start offset may be specified to facilitate extracting multiple blocks of data from a single input vector, optimizing cache bandwidth and minimizing rereading of data.

Unpacked data may be zero or sign-extended into the destination lane.

The behavior of VUNPACKB is controlled by arguments provided as an immediate:

- **size** – Size of packed data [2..7]
- **start** – Starting offset
- **sign ext** – Optional sign extend unpacked data to 8 bits

Note: The size and start arguments are not fully orthogonal; only start options where a full extraction of VL/8 items of data is possible are permitted.

Vector width	Size	Start	Sign Extend
128/256/512	2	0, 1, 2, 3	Optional
	3	0, 1	
	4	0, 1	
	5	0	
	6	0	
	7	0	

The size, start, sign extend arguments are encoded in an 8-bit immediate (imm8):

imm8[7:0]								Notes
7	6	5	4	3	2	1	0	
0	0	sign ext	0	1	0	start		size=2; start=0..3
0	0	sign ext	0	1	1	0	start	size=3; start=0..1
0	0	sign ext	1	0	0	0	start	size=4; start=0..1
0	0	sign ext	1	0	1	0	0	size=5; start=0
0	0	sign ext	1	1	0	0	0	size=6; start=0
0	0	sign ext	1	1	1	0	0	size=7; start=0

Note: All unused encodings are RESERVED. `imm8[7:6]` is RESERVED/SBZ. Software should refrain from using RESERVED encodings, however no #UD or #GP is enforced.

9.9.2 Operands

Operand 1	Operand 2	Operand 3
xmm1 {k1}{z}	xmm2/m128	imm8
ymm1 {k1}{z}	ymm2/m256	imm8
zmm1 {k1}{z}	zmm2/m512	imm8

Instruction	Operand 1	Operand 2	Operand 3
VUNPACKB	MODRM.REG(w)	MODRM.R/M(r)	imm8

9.9.3 Other Exceptions

Exception class E4NF, see Section 5.3.

9.9.4 Operation

```
DEFINE vunpackb(data, imm8, VL, k1, zeroing, no_writemask):
    ASSERT VL in (128, 256, 512)
    KL      = VL / 8
```

```
    sign_ex = imm8[5]  // 1 = sign-extend, 0 = zero-extend

    // Size conditioning: imm8[4:2] clamped to minimum of 2
    // (encodings 0 and 1 are reserved but produce defined behaviour)
    size = max(imm8[4:2], 2)  // size in bits, range [2..7] after clamp

    // Start conditioning: depends on size — not all imm8[1:0] values valid
    // Only start values that allow a full KL-element extraction within VL bits.
    // Constraint after conditioning: (start + 1) * KL * size ≤ VL
    IF size == 2:
        start = imm8[1:0]  // full 2-bit range: 0..3
    ELSE IF size == 3 OR size == 4:
        start = min(imm8[1:0], 1)  // clamp to 0 or 1
    ELSE:  // size 5, 6, 7
        start = 0  // always 0; imm8[1:0] ignored

    orig_dest = copy(dest)
    FOR i = 0 TO KL-1:
        j    = (start * KL + i) * size  // bit offset of ith packed element
        elem = data[j + size - 1 : j]  // extract size-bit field
        IF sign_ex:
            elem[7 : size] = elem[size - 1]  // sign-extend from MSB of field
        ELSE:
            elem[7 : size] = 0  // zero-extend
        IF k1[i] OR no_writemask:
            dest.byte[i] = elem
        ELSE IF zeroing:
            dest.byte[i] = 0
        ELSE:
            dest.byte[i] = orig_dest.byte[i]
    dest[MAXVL-1 : VL] = 0
```

9.9.5 Instruction Table

Mnemonic	Operands	Description
VUNPACKB	xmm1 {k1}{z}, xmm2/m128, imm8	Unpack packed sub-byte elements (2–7 bits) to bytes; imm8 selects element size, start offset within the packed block, and optional sign extension.
	ymm1 {k1}{z}, ymm2/m256, imm8	
	zmm1 {k1}{z}, zmm2/m512, imm8	

9.9.6 C/C++ Compiler Intrinsic Equivalent

```
/* Compose the imm8 argument for _mm{w}_unpackb with | :
 *   ACE_UNPACKB_SIZE(n)    imm8[4:2]  element size in bits (2..7)
 *   ACE_UNPACKB_START(s)   imm8[1:0]  start offset (0..3, subject to size)
 *   ACE_UNPACKB_SEXT       imm8[5]    sign-extend output (all sizes)
 * imm8[7:6] are reserved and must be zero. */
#define ACE_UNPACKB_SIZE(n)   (((n) & 0x7) << 2)
#define ACE_UNPACKB_START(s)  (((s) & 0x3) << 0)
#define ACE_UNPACKB_SEXT      (1 << 5)

/* unpackb — Sub-byte element extraction */
__m512i _mm512_unpackb(__m512i a, unsigned int b);
```

10 AI Compute Extensions (ACE)

10.1 Introduction to the AI Compute Extensions

The AI Compute Extensions deliver higher compute density capabilities to the x86 architecture; the initial version of the architecture introduces operations to accelerate matrix math operations.

ACE is a set of architectural extensions for the x86 architecture; ACE augments AVX and scalar code with new capabilities, adding:

- ACE register state, including tile and block scale registers
- Data processing operations that consume AVX register input and operate on tile register state
- Data move operations to move data between ACE register state, AVX registers and memory
- State and operations for system management

ACE provides tight integration between AVX vectors and ACE tile registers, combining high compute density tile processing operations with the comprehensive data processing features of AVX.

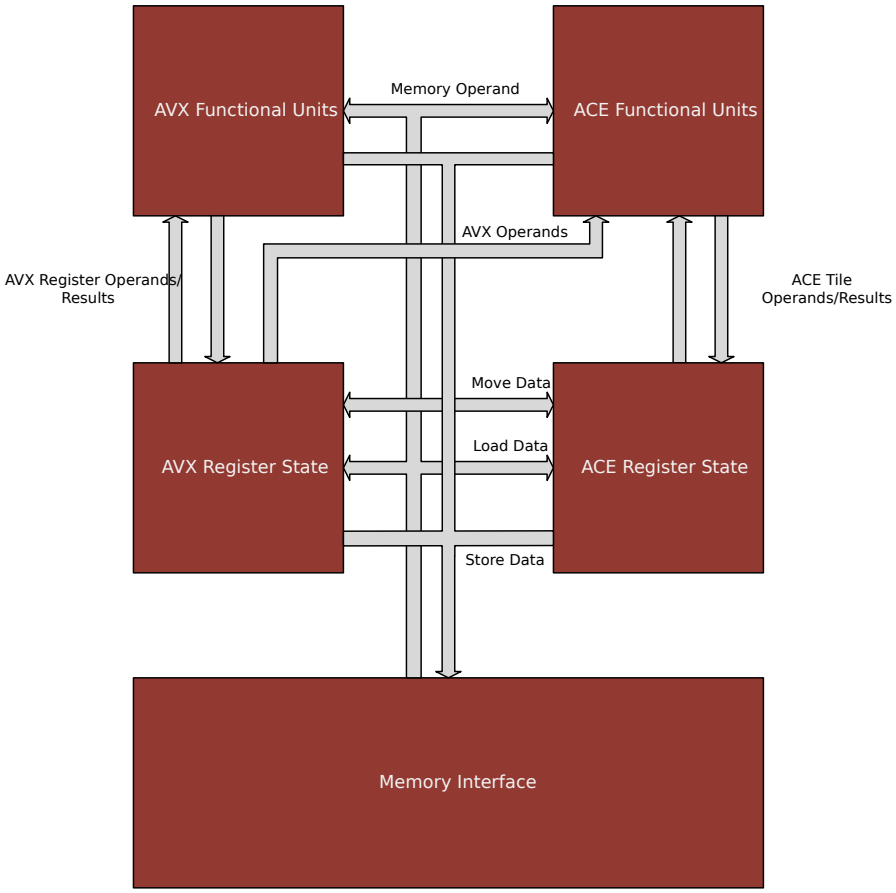

Figure 1: ACE Logical Organization.

10.2 Processor State

10.2.1 Tile Register File

The ACE extensions add a tile register file, containing a number of two-dimensional tile registers with dimensions outlined below.

Tile register dimension	Tile register file entries
512-bits x 16 rows	8

An ACE tile register has dimensions of 512-bits x 16 rows; each row is equivalent in size to a single AVX-512 vector. Each tile register row has width of 512-bits and may be viewed as consisting of a number of elements, dependent on the type of data being processed in the tile register. For the initial version of ACE, support for 32-bit (FP32 or INT32) accumulator types is provided; each ACE tile register row is therefore equivalent to 16 32-bit elements. The ACE extensions define eight tile registers.

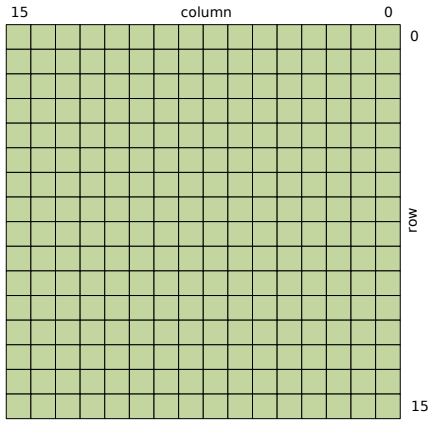

Figure 2: Tile register organization for 32-bit elements.

10.2.2 Block Scale Register

The ACE extensions add a block scale register that contains a number of scale elements; the scale elements may be applied to intermediate results prior to accumulation during the computation of select operations. One scale register is provided.

Block Scale register dimension	Block Scale register entries
1024-bits	1

The Block Scale register is organized as two segments of 512-bits (A scales and B scales), one segment is dedicated to each input to ACE operations that support inline scaling as defined by the OCP MX standard. Each segment is composed of four groups of 16 8-bit scale elements. The groups are organized by element index. For example, all element 0 scales for A are grouped in bits [31:0] of the A scales segment. Operations that use the scale register as input interpret the 8-bit scale elements as a power-of-two E8M0 scale, as defined in the OCP MX standard.

Block Scale Register bits	Content
1023:512	A scales[0..15][0..3]
511:0	B scales[0..15][0..3]

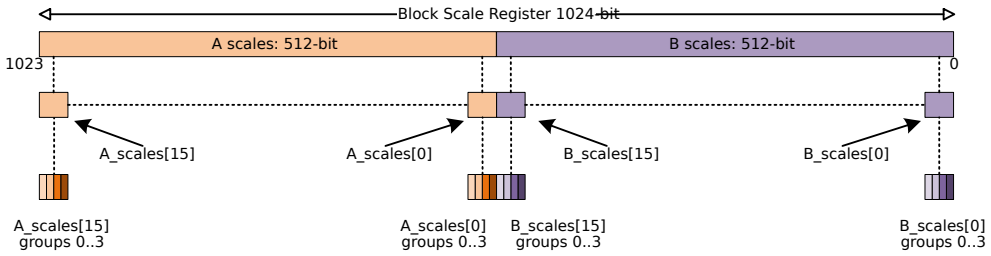

Figure 3: Block Scale register organization.

10.2.3 Reset State

At reset, each byte in the Block Scale Register is initialized to the value `0x7F`. This is the equivalent of 2^0 when the E8M0 bias is considered.

10.3 Matrix Multiplication

10.3.1 Terminology

In this specification the following terminology is used when describing matrix multiplication:

- Input matrices (A, B)
- Output matrix (C)
- Dimension (M, K, N)

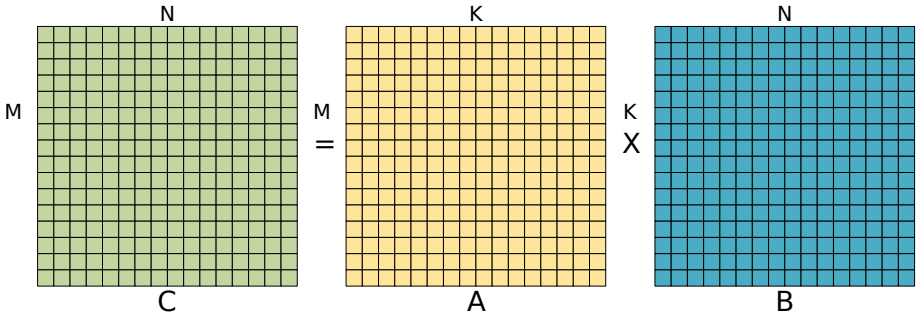

Figure 4: Illustration of matrix multiplication, matrices and dimensions.

In the matrix multiplication C = A × B, every row of A is combined with every column of B to form the M×N result matrix, C. This is depicted for a smaller matrix below. In this respect, A may be referred to as the row matrix and B as the column matrix.

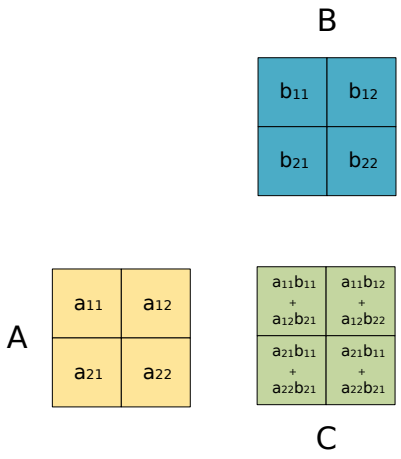

Figure 5: Illustration of matrix multiplication of A[M=2, K=2] by B[K=2, N=2].

10.4 Outer Product Operation

The outer product operation is a key feature of the ACE architecture. An outer product performs a matrix multiplication using two vectors as input matrices, accumulating the intermediate result (or sum of products) with the contents of the destination tile register.

The input vectors are divided into elements equal to the number of rows or columns in the destination tile register. With the ACE architecture, the input vectors are 512-bits in length and are divided into 16 elements of 32-bits.

10.4.1 Outer Product Rank

The input vectors provide matrix inputs of dimension (16 × K) or (K × 16). The hidden dimension K is determined by the element format size e of the specific outer product

operation where 32 = K × e. In the context of outer product operations, K is referred to as the rank of the outer product.

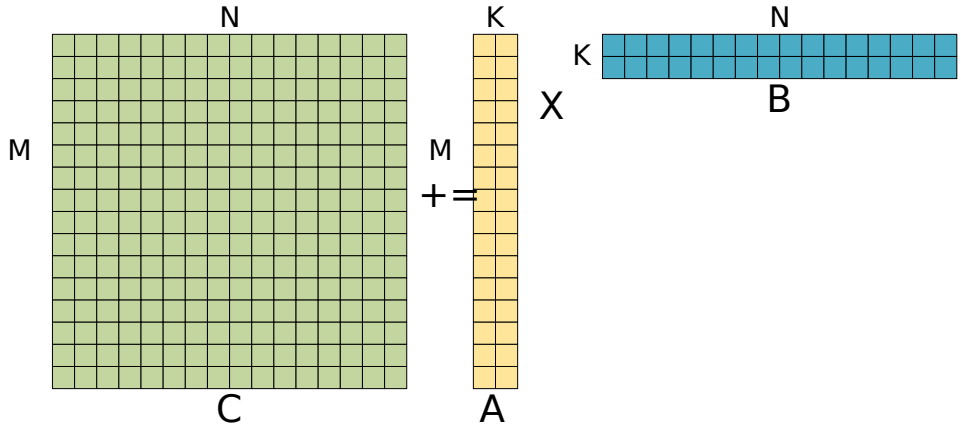

Figure 6: Matrix multiplication with accumulate performed by rank-2 outer product.

This is represented more compactly in subsequent descriptions in the following form: a grid notation is used where the A input matrix projects elements across the rows of the destination tile and the B matrix projects elements down the columns of the destination tile. This compact notation elides the per-sub-element detail and instead shows the accumulated dot product at each tile element intersection.

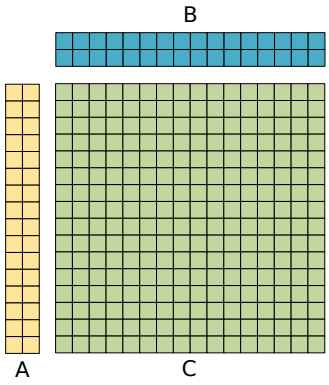

Figure 7: Compact representation of outer product operation.

10.4.2 Outer Product Rank-1

Note: Version 1.0 of the ACE architecture does not provide any Rank-1 outer product operations; this description is provided to give a baseline for other rank operations.

A rank-1 outer product has input vector elements of equal width to the tile element width. At each intersection of row and column elements, these elements are multiplied and accumulated with the corresponding destination tile element.

This is equivalent to a GEMM operation **C** = α **A B** + β **C**, with (α, β, M, N, K) = (1, 1, 16, 16, 1).

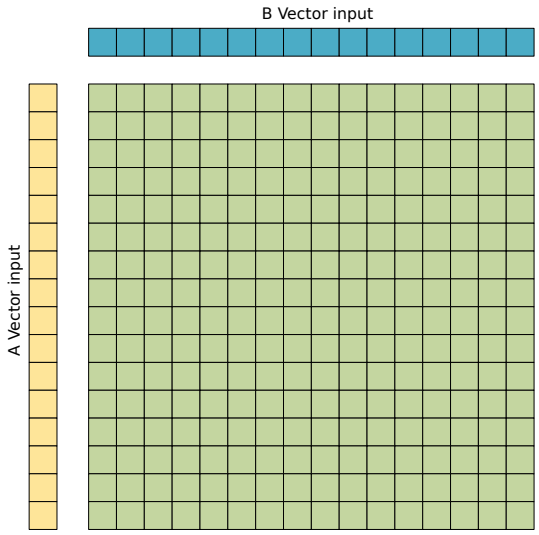

Figure 8: Outer Product Rank-1.

A larger 16 × K × K × 16 matrix multiply may be synthesized by applying consecutive Rank-1 outer product operations, stepping one row and column through the two input matrices at each iteration. Each 32-bit lane carries one element, so a single 512-bit ZMM register covers all 16 rows or columns simultaneously — enabling a single VMOVDQU to load all elements needed for one rank-1 step.

10.4.2.1 Memory Arrangement

The column input matrix (“B”) should be arranged in memory in row-major order; the row input matrix (“A”) should be arranged in column-major order (i.e. transposed at 32-bit granularity).

> **Note:** The outer product operation requires one matrix input to present row-major and the other column-major in the two input vectors presented to the output product. In this document we use the “B” and “A” names, in keeping with existing AMX documentation conventions where “B” is introduced as row-major.

```
uint32_t A_mem[M][K];       // Logical A matrix
uint32_t AT_mem[K][M];      // Transposed A
uint32_t B_mem[K][N];       // Logical B matrix

for (m=0; m<M; ++m) {
    for (k=0; k<K; ++k) {
        AT_mem[k][m] = A_mem[m][k];
    }
}
```

10.4.3 Outer Product Rank-2

A rank-2 outer product has input vector elements that are composed of sub-elements equal to half the width of a tile element. At each intersection of the row and column elements, the corresponding sub-elements are multiplied and accumulated with the corresponding destination tile element.

This is equivalent to performing two iterations of a Rank-1 outer product where the input element data is half the size of the destination tile element.

This is equivalent to a GEMM operation **C** = α **A B** + β **C**, with (α, β, M, N, K) = (1, 1, 16, 16, 2).

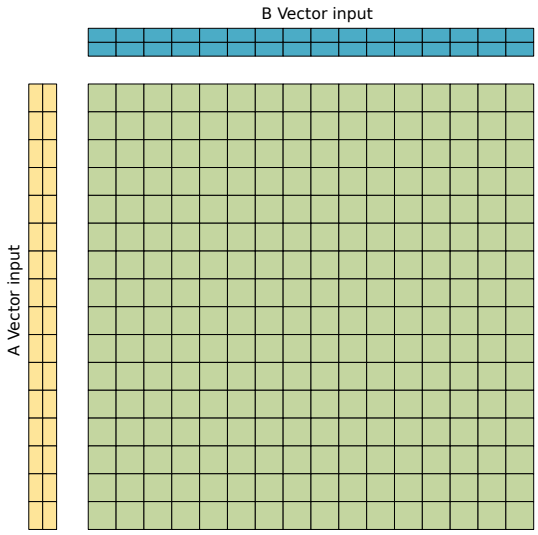

Figure 9: Outer Product Rank-2.

A larger 16 × K × K × 16 matrix multiply may be synthesized by applying consecutive Rank-2 outer product operations, stepping two rows and columns through the input matrices at each iteration.

With ACE the input sub-elements for a Rank-2 outer product will be 16-bits in size, arranged in 32-bit AVX-512 lanes with the first sub-element aligned at the bottom of the lane:

Bits 31:16	Bits 15:0
k1	k0

Table 2: Rank-2 sub-element lane layout within a 32-bit lane. Two 16-bit sub-elements (k0, k1) are packed with k0 at the least-significant position.

Packing both sub-elements into a single 32-bit lane means that one 512-bit ZMM register covers all 16 rows or columns simultaneously. A single VMOVDQU therefore loads all sub-elements needed for one rank-2 step across all 16 output rows at once.

10.4.3.1 Memory Arrangement

The row input matrix (“A”) should be arranged in column-major order (i.e. transposed at 32-bit granularity).

The column input matrix (“B”) should be arranged in memory in row-major order, with pairs of rows interleaved at 16-bit granularity.

```
uint16_t A_mem[M][K];              // Logical A matrix
uint16_t AT_mem[K/2][M][2];       // A matrix transposed at 32-bit granularity
uint16_t B_mem[K][N];              // Logical B matrix
uint16_t B_mem_pack[K/2][N][2];   // Row packed B matrix

for (m=0; m<M; ++m) {
    for (k=0; k<K; ++k) {
        AT_mem[k/2][m][k%2] = A_mem[m][k];
    }
}

for (k=0; k<K; ++k) {
    for (n=0; n<N; ++n) {
        B_mem_pack[k/2][n][k%2] = B_mem[k][n];
    }
}
```

10.4.4 Outer Product Rank-4

A rank-4 outer product has input vector elements that are composed of sub-elements equal to a quarter of the width of a tile element. At each intersection of the row and column elements, the corresponding sub-elements are multiplied and accumulated with the corresponding destination tile element.

This is equivalent to performing four iterations of a Rank-1 outer product where the input element data is a quarter the size of the destination tile element.

This is equivalent to a GEMM operation **C** = α **A B** + β **C**, with (α, β, M, N, K) = (1, 1, 16, 16, 4). For selected operations where the input data format is one of the OCP MX formats, a non-unity power-of-two E8M0 scale is supported for α. See Section 10.5.

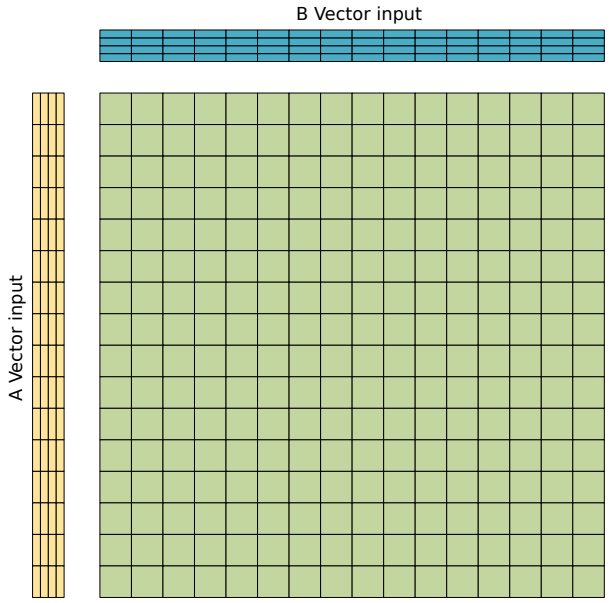

Figure 10: Outer Product Rank-4.

A larger 16 $\times$ K $\times$ K $\times$ 16 matrix multiply may be synthesized by applying consecutive Rank-4 outer product operations, stepping four rows and columns through the input matrices at each iteration.

With ACE the input sub-elements for a Rank-4 outer product will be 8-bits in size, arranged in 32-bit AVX-512 lanes with the first sub-element aligned at the bottom of the lane:

Bits 31:24	Bits 23:16	Bits 15:8	Bits 7:0
k3	k2	k1	k0

Table 3: Rank-4 sub-element lane layout within a 32-bit lane. Four 8-bit sub-elements (k0–k3) are packed with k0 at the least-significant position.

Details of per-element operation of the Outer Product Rank-4 are illustrated below.

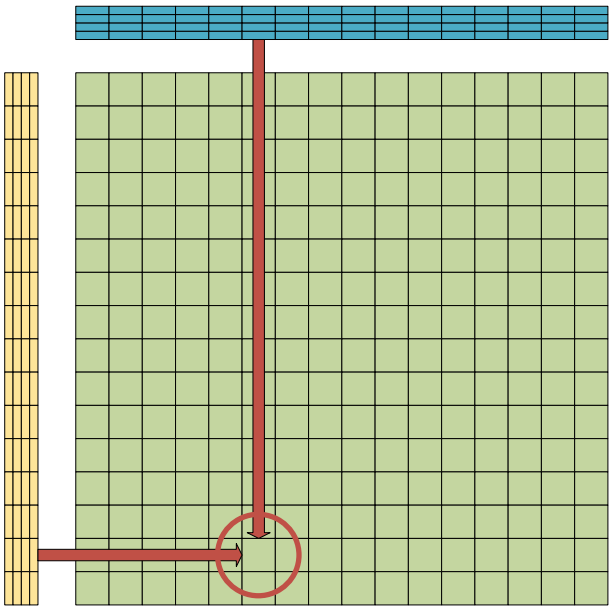

Figure 11: Outer Product Rank-4 details.

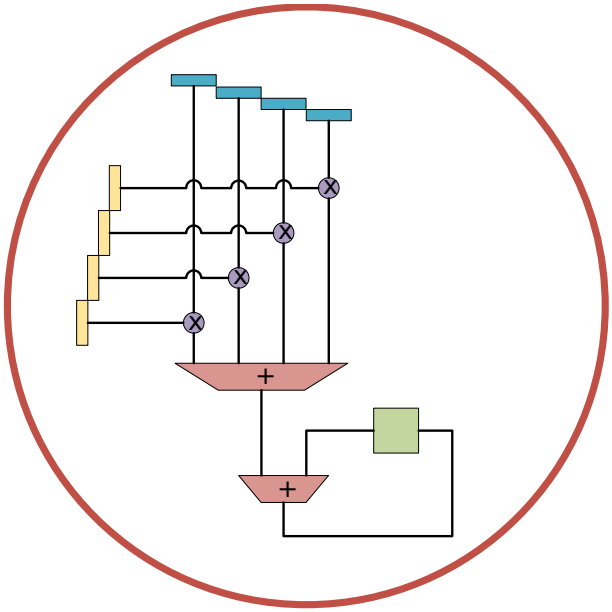

Figure 12: Operation performed at highlighted tile element.

Packing all four sub-elements into a single 32-bit lane means that one 512-bit ZMM register covers all 16 rows or columns simultaneously. A single VMOVDQU therefore loads all sub-elements needed for one rank-4 step across all 16 output rows at once.

10.4.4.1 Memory Arrangement

The row input matrix (“A”) should be arranged in column-major order (i.e. transposed at 32-bit granularity).

The column input matrix (“B”) should be arranged in memory in row-major order, with quads of rows interleaved at 8-bit granularity.

```
uint8_t A_mem[M][K];              // Logical A matrix
uint8_t AT_mem[K/4][M][4];       // A matrix transposed at 32-bit granularity
uint8_t B_mem[K][N];              // Logical B matrix
uint8_t B_mem_pack[K/4][N][4];   // Row packed B matrix

for (m=0; m<M; ++m) {
    for (k=0; k<K; ++k) {
```

```
            AT_mem[k/4][m][k%4] = A_mem[m][k];
        }
    }

    for (k=0; k<K; ++k) {
        for (n=0; n<N; ++n) {
            B_mem_pack[k/4][n][k%4] = B_mem[k][n];
        }
    }
```

10.5 Block Scale Support

The ACE extensions provide a number of outer product operations that accept OCP MX data formats as input. The OCP MX standard describes a number of block scaled formats where a power-of-two E8M0 scale is associated with a block of element data.

The ACE extensions provide support to apply this block scale as part of outer product operations. Outer product operations supporting block scale read two groups of scales from the Block Scale register. One group is relevant to the row (“A”) matrix and the other relevant to the column (“B”) input matrix data. Each group consists of 16 E8M0 scale values.

An example depicting block scales applied to a Rank-4 outer product is illustrated in the diagrams below.

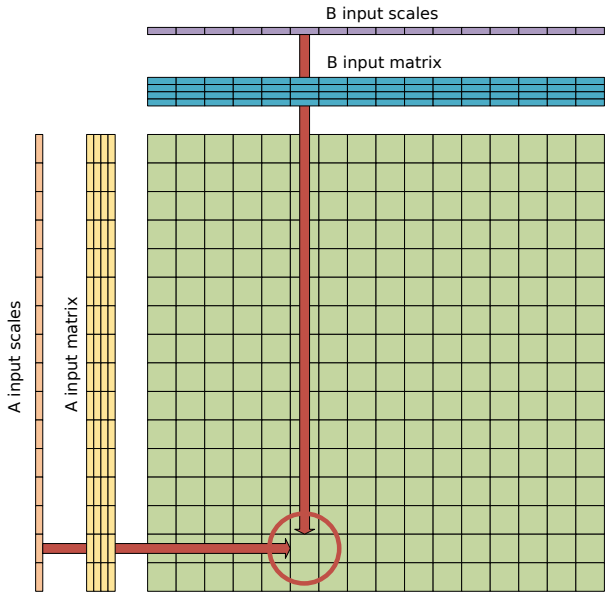

Figure 13: Outer Product Rank-4 with scale data.

At each intersection of the row and column elements, the corresponding sub-elements are multiplied and summed. The intersecting scale values are combined to form a power-of-two E8M0 scale to be applied to this sum prior to accumulation with the corresponding destination tile element.

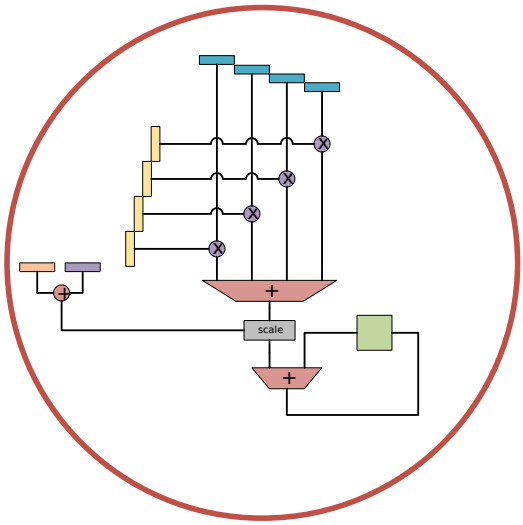

Figure 14: Operation performed at highlighted tile element.

The E8M0 format is reproduced from the OCP MX standard below.

	E8M0
Exponent bias	127
Supported exponent range	−127 to 127
Infinities	N/A
NaN	11111111_2
Zeros	N/A

Note: It is possible to achieve unscaled behavior (e.g. equivalent to OCP FP8 instead of OCP MX FP8) by ensuring that both scale terms encode an exponent of zero (`0x7F` after the inclusion of bias).

The input scales are obtained using an immediate argument for each input that selects a group of 16 8-bit scales from the appropriate half of the Block Scale Register.

Input Scale	Block Scale Register bits
A (row)	1023:512
B (col)	511:0

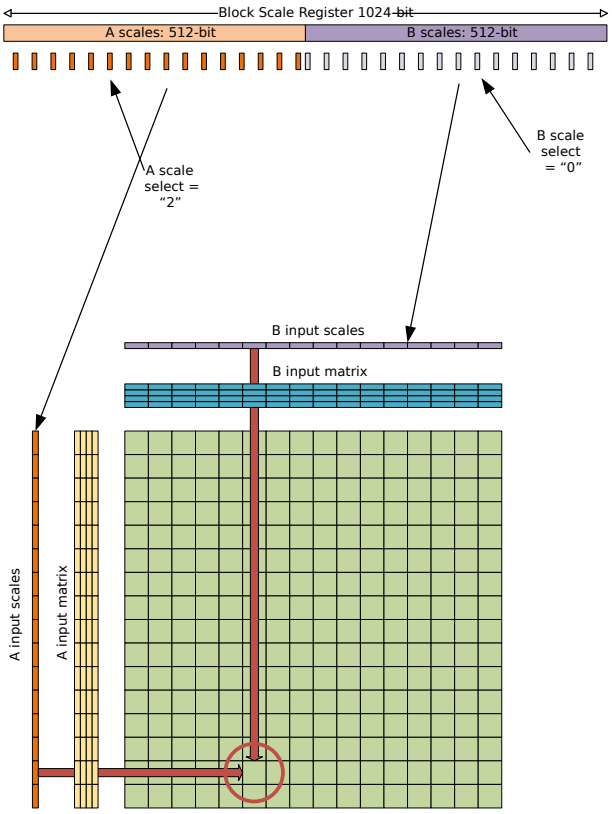

Figure 15: Block scale group selection in OCP MX outer product.

10.6 Operations Introduced with the ACE Extensions

The following new operations are introduced and are detailed in subsequent sections:

- Block Scale Register Init (BSRINIT). Refer to Section 13.
- Block Scale Register Move (BSRMOVF, BSRMOVH, BSRMOVL). Refer to Section 13.
- Tile Outer Product MX FP8 (Rank 4). Refer to Section 14.
- Tile Outer Product MX INT8 (Rank 4). Refer to Section 14.
- Tile Outer Product BF16 (Rank 2). Refer to Section 14.
- Tile Outer Product INT8 (Rank 4). Refer to Section 14.

The following instructions are enumerated under feature flag AMX-TILE and are imported from the AMX specification:

- TILEZERO, LDTILECFG, STTILECFG, TILERELEASE. Refer to Section 11.

The following instructions are enumerated under feature flag AMX-TILE (or ACE_VSN >= 1) and are imported from the AMX-AVX512 family:

- Tile Register Move Row (TILEMOVROW) – extended under ACE to allow both read and write. Refer to Section 12.
- Tile Register Move Column (TILEMOVCOL) – new operation class. Refer to Section 12.
- Tile Move Row and Convert INT32 to FP32 (TCVTROWD2PS). Refer to Section 12.
- Tile Move Row and Convert FP32 to BF16 (TCVTROWPS2BF16[H,L]). Refer to Section 12.
- Tile Move Row and Convert FP32 to FP16 (TCVTROWPS2PH[H,L]). Refer to Section 12.

11 Tile Management Instructions

11.1 TILEZERO – Zero Tile

11.1.1 Description

A zero tile operation is provided to initialize a tile with zero data. The specified tile register is written with zeros.

11.1.2 Operands

Operand 1
tmm1

Instruction	Operand 1	Operand 2	Operand 3
TILEZERO	MODRM.REG(w)	N/A	N/A

11.1.3 Other Exceptions

Exception class AMX-E5, see Section 5.6.3.

11.1.4 Operation

```
DEFINE tilezero(dst):
    FOR row = 0 TO dst.rows-1:
        FOR col = 0 TO dst.colsb-1:
            dst.byte[row][col] = 0
    zero_tileconfig_start()
```

11.1.5 Instruction Table

Mnemonic	Operands	Description
TILEZERO	tmm1	Zero all elements of the tile register tmm1.

11.1.6 Masking

None.

11.1.7 Rounding

N/A.

11.1.8 Floating-Point Exceptions

None.

11.1.9 Encoding Notes

Only 8 TMM registers are architected; note impact to register specifier field encodings.

11.1.10 C/C++ Compiler Intrinsic Equivalent

```
/* _tile_zero – Imported from AMX; zeros tile register */
void _tile_zero(__tile1024i *tdst);
```

11.2 LDTILECFG – Load Tile Configuration

11.2.1 Description

This instruction takes a pointer to a 64-byte memory location. The memory location contains a tile configuration description as described in Section 11.2.3. The descriptor is fetched from memory and, if valid, the tile configuration is set accordingly. An invalid tile configuration descriptor will result in a #GP fault.

Note: A successful execution of LDTILECFG updates TILECFG with the specified descriptor and initializes TILEDATA and SCALEDATA:

- TILEDATA bytes are set to 0.
- SCALEDATA (Block Scale Register) bytes are set to INIT (`0x7F`) state.

11.2.2 Operands

Operand 1
m512

Instruction	Operand 1	Operand 2	Operand 3
LDTILECFG	MODRM.R/M(r)	N/A	N/A

11.2.3 Configuration Memory Descriptor Layout

The following tile descriptor memory layouts may be selected, depending on which AMX palette IDs are supported by the implementation:

- **Palette 0 (AMX INIT)** – The Palette 0 descriptor is 64 bytes of zero.
- **Palette 1 (AMX TMUL)** – The Palette 1 descriptor is described in Intel SDM documentation of LDTILECFG.
- **Palette 2 (ACE)** – As described below.

The Palette 2 (ACE) descriptor:

Byte(s)	Field Name	Description
0	palette	Palette selects the supported configuration of the tiles that will be used.
1-63	Reserved	Must be zero.

11.2.4 Other Exceptions

Exception class AMX-E1, see Section 5.6.1.

11.2.5 Operation

```
DEFINE ldtilecfg(base):
    buf = read_memory(base, 64)
    temp_palette_id = buf.byte[0]
    IF temp_palette_id > max_palette:
        #GP(0)
    IF NOT cpu_supports_palette(temp_palette_id):
        #GP(0)
    IF NOT xcr0_supports_palette(temp_palette_id):
        #GP(0)
    // Validate palette-specific fields (raises GP on error)
    IF temp_palette_id == 1:
        temp = process_tilecfg_palette_1(buf)
    ELSE IF temp_palette_id == 2:
        temp = process_tilecfg_palette_2(buf)
    // Commit configuration
    IF temp_palette_id == 0:
        // INIT state: release tiles and BSR
        tilecfg = init_tilecfg()
        zero_all_tile_data()
        IF XCR0[SCALEDATA]:
            bsr_init()
        TILES_CONFIGURED = 0
    ELSE:
        // Palette 1 or 2: apply validated configuration
        tilecfg = temp
        zero_all_tile_data()
        TILES_CONFIGURED = 1
        IF XCR0[SCALEDATA]:
            bsr_init()
    zero_tileconfig_start()
```

11.2.6 Instruction Table

Mnemonic	Operands	Description
LDTILECFG	m512	Load tile configuration from the TILECFG structure at m512.

11.2.7 Masking
None.

11.2.8 Rounding
N/A.

11.2.9 Floating-Point Exceptions
None.

11.2.10 C/C++ Compiler Intrinsic Equivalent

```
/* _tile_loadconfig — Imported from AMX; loads tile config from memory */
void _tile_loadconfig(const void *mem);
```

11.3 STTILECFG – Store Tile Configuration

11.3.1 Description
This instruction takes a pointer to a 64-byte memory location and stores the current tile configuration at that address.

11.3.2 Operands

Operand 1
m512

Instruction	Operand 1	Operand 2	Operand 3
STTILECFG	MODRM.R/M(w)	N/A	N/A

11.3.3 Other Exceptions
Exception class AMX-E2, see Section 5.6.2.

11.3.4 Operation

```
DEFINE sttilecfg(base):
    buf = [0] * 64
    IF TILES_CONFIGURED != 0:
        buf[0] = tilecfg.palette_id
        IF tilecfg.palette_id == 1:  // AMX palette: write per-tile fields
            buf[1] = tilecfg.start_row
            FOR n = 0 TO palette_table[1].max_names-1:
                buf[16 + 2*n] = tilecfg.t[n].colsb & 0xFF
                buf[17 + 2*n] = tilecfg.t[n].colsb >> 8
                buf[48 + n]   = tilecfg.t[n].rows
        // Palette 2: only palette_id is stored; rows/colsb are fixed
        // and not written to the buffer (bytes 1-63 remain zero).
    write_memory(base, 64, buf)
```

11.3.5 Instruction Table

Mnemonic	Operands	Description
STTILECFG	m512	Store tile configuration to the TILECFG structure at m512.

11.3.6 Masking
None.

11.3.7 Rounding
N/A.

11.3.8 Floating-Point Exceptions
None.

11.3.9 C/C++ Compiler Intrinsic Equivalent

```
/* _tile_storeconfig — Imported from AMX; stores tile config to memory */
void _tile_storeconfig(void *mem);
```

11.4 TILERELEASE – Release Tile

11.4.1 Description

This instruction returns TILECFG and TILEDATA to the INIT state:

- TILECFG and TILEDATA bytes are set to 0.
- SCALEDATA (Block Scale Register, BSR) bytes are set to INIT (`0x7F`) state.
- Palette 0 is selected.

11.4.2 Other Exceptions

Exception class AMX-E6, see Section 5.6.4.

11.4.3 Operation

```
DEFINE tilerelease():
    zero_all_tile_data()
    IF XCR0[SCALEDATA]:
        bsr_init()  // SCALEDATA → 0x7F (E8M0 = 2^0 = 1.0)
    zero_tileconfig_start()
    tilecfg = init_tilecfg()
    TILES_CONFIGURED = 0
```

11.4.4 Instruction Table

Mnemonic	Operands	Description
TILERELEASE		Release tile state: mark TILEDATA as invalid and initialize TILECFG to zero.

11.4.5 Masking

None.

11.4.6 Rounding

N/A.

11.4.7 Floating-Point Exceptions

None.

11.4.8 C/C++ Compiler Intrinsic Equivalent

```
/* _tile_release — Imported from AMX; releases tile state */
void _tile_release(void);
```

12 Tile Data Movement

12.1 Tile Move Operations – Overview

The following operations are an expansion of the family of operations available under AMX-AVX512. The operations are primarily as documented in the Intel ISA Extensions reference, with some capability expansions and other differences outlined below:

- **Tile Move Row (TILEMOVROW)** – This class is extended under the ACE specification to allow both read and write of tile register row.
- **Tile Move Column (TILEMOVCOL)** – This is a new operation class introduced under the ACE specification to allow write to a column of a tile register.
- **Tile Move Row and Convert INT32 to Single Precision (TCVTROWD2PS)**
- **Tile Move Row and Convert FP32 Elements to BF16 Elements (TCVTROWPS2BF16[H,L])**
- **Tile Move Row and Convert FP32 Elements to FP16 Elements (TCVTROWPS2PH[H,L])**

12.1.1 Changes to Exception Behavior

The interpretation for immediate or register row/column specifier out of range is modified so that no fault is raised.

12.2 Tile Move Row (TILEMOVROW)

12.2.1 Description

Tile register move row operations transfer data between ACE tile registers and AVX registers. The operation moves one row of a tile register to or from a ZMM register. The row is specified using either an immediate or value contained within a 32-bit general purpose register.

Only bits [3:0] of the immediate or general-purpose register are relevant to specifying the row; other bits are RESERVED/SBZ.

12.2.2 Operands

Form	Operand 1	Operand 2	Operand 3
Read	zmm1	tmm2	r32
Read	zmm1	tmm2	imm8
Write	tmm1	zmm2	r32
Write	tmm1	zmm2	imm8

Instruction	Operand 1	Operand 2	Operand 3
TILEMOVROW	MODRM.REG(w)	MODRM.R/M(r)	VVVV(r)
TILEMOVROW	MODRM.REG(w)	MODRM.R/M(r)	imm8
TILEMOVROW	MODRM.REG(r/w)	MODRM.R/M(r)	VVVV(r)
TILEMOVROW	MODRM.REG(r/w)	MODRM.R/M(r)	imm8

12.2.3 Other Exceptions

Immediate form: exception class ACE-E1, see Section 5.7.1. Register form: exception class ACE-E6, see Section 5.7.6.

12.2.4 Operation

```
DEFINE tilemovrow_read_imm(src, imm8):
    ASSERT VL in (512)
    row = imm8[3:0]  // 4-bit: rows 0..15
    FOR col = 0 TO 63:  // 64 bytes = full 512-bit row, always valid
        dst.byte[col] = src.byte[row][col]
    dst[MAXVL-1 : VL] = 0
```

```
DEFINE tilemovrow_read_gpr(src, r32):
```

```
    ASSERT VL in (512)
    row = r32 & 0xF
    FOR col = 0 TO 63:
        dst.byte[col] = src.byte[row][col]
    dst[MAXVL-1 : VL] = 0
```

```
DEFINE tilemovrow_write_imm(dst, src, imm8):
    ASSERT VL in (512)
    row = imm8[3:0]
    FOR col = 0 TO 63:
        dst.byte[row][col] = src.byte[col]
```

```
DEFINE tilemovrow_write_gpr(dst, src, r32):
    ASSERT VL in (512)
    row = r32 & 0xF
    FOR col = 0 TO 63:
        dst.byte[row][col] = src.byte[col]
```

12.2.5 Instruction Table

Mnemonic	Operands	Description
TILEMOVROW	zmm1, tmm2, r32	Move a row between a tile register and a ZMM vector register. Read forms (zmm1←tmm2) and write forms (tmm1←zmm2). Row index selected by GPR or immediate.
	zmm1, tmm2, imm8	
	tmm1, zmm2, r32	
	tmm1, zmm2, imm8	

12.2.6 Masking
None.

12.2.7 Rounding
N/A.

12.2.8 Floating-Point Exceptions
None.

12.2.9 Encoding Notes
Only 8 TMM registers are architected; note impact to register specifier field encodings.

12.2.10 C/C++ Compiler Intrinsic Equivalent

```
/* _tile_movrow — read direction */
__m512i _tile_movrow(const __tile1024i *tsrc, unsigned int row);

/* _tile_setrow — write direction */
void _tile_setrow(__tile1024i *tdst, unsigned int row, __m512i src);
```

12.3 Tile Move Column (TILEMOVCOL)

12.3.1 Description
Tile register move column operations transfer data from AVX registers to ACE tile registers. The operation writes the contents of a ZMM register to a single column of an ACE tile register. The column is specified using either an immediate or value contained within a 32-bit general purpose register.

Only bits [3:0] of the immediate or general-purpose register are relevant to specifying the column; other bits are RESERVED/SBZ.

12.3.2 Operands

Operand 1	Operand 2	Operand 3
tmm1	zmm2	r32
tmm1	zmm2	imm8

Instruction	Operand 1	Operand 2	Operand 3
TILEMOVCOL	MODRM.REG(r/w)	MODRM.R/M(r)	VVVV(r)
TILEMOVCOL	MODRM.REG(r/w)	MODRM.R/M(r)	imm8

12.3.3 Other Exceptions

Immediate form: exception class ACE-E1, see Section 5.7.1. Register form: exception class ACE-E6, see Section 5.7.6.

12.3.4 Operation

```
DEFINE tilemovcol_imm(dst, src, imm8):
    ASSERT VL in (512)
    col = imm8[3:0]
    FOR row = 0 TO dst.rows-1:
        dst.dword[row][col] = src.dword[row]
```

```
DEFINE tilemovcol_gpr(dst, src, r32):
    ASSERT VL in (512)
    col = r32 & 0xF
    FOR row = 0 TO dst.rows-1:
        dst.dword[row][col] = src.dword[row]
```

12.3.5 Instruction Table

Mnemonic	Operands	Description
TILEMOVCOL	tmm1, zmm2, r32	Move a column from a ZMM vector register to a tile register. Column index selected by GPR or immediate.
	tmm1, zmm2, imm8	

12.3.6 Masking

None.

12.3.7 Rounding

N/A.

12.3.8 Floating-Point Exceptions

None.

12.3.9 Encoding Notes

Only 8 TMM registers are architected; note impact to register specifier field encodings.

12.3.10 C/C++ Compiler Intrinsic Equivalent

```
/* _tile_setcol – Write only: tmm←zmm col */
void _tile_setcol(__tile1024i *tdst, unsigned int row, __m512i src);
```

12.4 Tile Move Row and Convert INT32 to FP32 (TCVTROWD2PS)

12.4.1 Description

This operation moves a row from a tile register to a ZMM register, converting the INT32 source elements to FP32. Moves are available from an ACE tile register row; the row is specified using either an immediate or value contained within a 32-bit general purpose register.

No SIMD exceptions are generated; rounding is round ties to even, performed as if MXCSR.RC=RNE. Embedded rounding is not supported. MXCSR is neither consulted nor updated. No floating-point exceptions are generated.

12.4.2 Operands

Operand 1	Operand 2	Operand 3
zmm1	tmm2	r32
zmm1	tmm2	imm8

Instruction	Operand 1	Operand 2	Operand 3
TCVTROWD2PS	MODRM.REG(w)	MODRM.R/M(r)	VVVV(r)

Instruction	Operand 1	Operand 2	Operand 3
TCVTROWD2PS	MODRM.REG(w)	MODRM.R/M(r)	imm8

12.4.3 Other Exceptions

Immediate form: exception class ACE-E1, see Section 5.7.1. Register form: exception class ACE-E6, see Section 5.7.6.

12.4.4 Operation

```
DEFINE tcvtrowd2ps_imm(src, imm8):
    ASSERT VL in (512)
    row = imm8[3:0]
    FOR col = 0 TO 15:  // 16 dwords = 64 bytes = full 512-bit row
        dst.fp32[col] = convert_integer_to_fp32(src.dword[row][col])  // RNE
    dst[MAXVL-1 : VL] = 0
```

```
DEFINE tcvtrowd2ps_gpr(src, r32):
    ASSERT VL in (512)
    row = r32 & 0xF
    FOR col = 0 TO 15:
        dst.fp32[col] = convert_integer_to_fp32(src.dword[row][col])  // RNE
    dst[MAXVL-1 : VL] = 0
```

12.4.5 Instruction Table

Mnemonic	Operands	Description
TCVTROWD2PS	zmm1, tmm2, r32	Move a row from a tile register to zmm1, converting each INT32 element to FP32. Row index selected by GPR or immediate.
	zmm1, tmm2, imm8	

12.4.6 Masking

None.

12.4.7 Rounding

Round ties to even, performed as if MXCSR.RC=RNE. Embedded rounding is not supported.

12.4.8 Floating-Point Exceptions

None.

12.4.9 Encoding Notes

Only 8 TMM registers are architected; note impact to register specifier field encodings.

12.4.10 C/C++ Compiler Intrinsic Equivalent

```
/* _tile_cvtrowd2ps — Move row INT32→FP32 */
__m512 _tile_cvtrowd2ps(const __tile1024i *tsrc, unsigned int row);
```

12.5 Tile Move Row and Convert FP32 to BF16 (TCVTROWPS2BF16[H,L])

12.5.1 Description

This operation moves a row from a tile register to a ZMM register, converting the FP32 source elements to BF16. Moves are available from an ACE tile register row; the row is specified using either an immediate or value contained within a 32-bit general purpose register.

Two variants are provided, writing BF16 results in the high 16 bits of each dword or the low 16 bits of each dword. The alternate 16 bits in each dword are written with zeros.

No SIMD exceptions are generated; rounding is round ties to even, performed as if MXCSR.RC=RNE. Embedded rounding is not supported. MXCSR is neither consulted nor updated. No floating-point exceptions are generated. DAZ is not obeyed and is always assumed DAZ=1. FTZ is not obeyed and is always assumed FTZ=1.

12.5.2 Operands

Operand 1	Operand 2	Operand 3
zmm1	tmm2	r32
zmm1	tmm2	imm8

Instruction	Operand 1	Operand 2	Operand 3
TCVTROWPS2BF16H	MODRM.REG(w)	MODRM.R/M(r)	VVVV(r)
TCVTROWPS2BF16H	MODRM.REG(w)	MODRM.R/M(r)	imm8
TCVTROWPS2BF16L	MODRM.REG(w)	MODRM.R/M(r)	VVVV(r)
TCVTROWPS2BF16L	MODRM.REG(w)	MODRM.R/M(r)	imm8

12.5.3 Other Exceptions

Immediate form: exception class ACE-E1, see Section 5.7.1. Register form: exception class ACE-E6, see Section 5.7.6.

12.5.4 Operation

```
DEFINE tcvtrowps2bf16_imm(src, imm8, variant):
    // variant: "H" → bf16 in word[1] of each dword (bits [31:16])
    //          "L" → bf16 in word[0] of each dword (bits [15:0])
    ASSERT VL in (512)
    row     = imm8[3:0]
    pos     = 1 if variant == "H" else 0  // which word slot gets the BF16
    zeropos = 0 if variant == "H" else 1  // which word slot is zeroed
    FOR col = 0 TO 15:
        dst.word[2*col + zeropos] = 0
        dst.bf16[2*col + pos] = fp32_to_bfloat16(src.fp32[row][col])  // RNE
    dst[MAXVL-1 : VL] = 0
```

```
DEFINE tcvtrowps2bf16_gpr(src, r32, variant):
    ASSERT VL in (512)
    row     = r32 & 0xF
    pos     = 1 if variant == "H" else 0
    zeropos = 0 if variant == "H" else 1
    FOR col = 0 TO 15:
        dst.word[2*col + zeropos] = 0
        dst.bf16[2*col + pos] = fp32_to_bfloat16(src.fp32[row][col])  // RNE
    dst[MAXVL-1 : VL] = 0
```

12.5.5 Instruction Table

Mnemonic	Operands	Description
TCVTROWPS2BF16H	zmm1, tmm2, r32	Move a tile row to zmm1, converting FP32 to BF16 packed into the high word of each dword. Row index selected by GPR or immediate.
	zmm1, tmm2, imm8	
TCVTROWPS2BF16L	zmm1, tmm2, r32	Move a tile row to zmm1, converting FP32 to BF16 packed into the low word of each dword. Row index selected by GPR or immediate.
	zmm1, tmm2, imm8	

12.5.6 Masking

None.

12.5.7 Rounding

Round ties to even, performed as if MXCSR.RC=RNE. Embedded rounding is not supported.

12.5.8 Floating-Point Exceptions

None.

12.5.9 Encoding Notes

Only 8 TMM registers are architected; note impact to register specifier field encodings.

12.5.10 C/C++ Compiler Intrinsic Equivalent

```
/* _tile_cvtrowps2bf16h — high half-word of each dword */
__m512i _tile_cvtrowps2bf16h(const __tile1024i *tsrc, unsigned int row);

/* _tile_cvtrowps2bf16l — low half-word of each dword */
__m512i _tile_cvtrowps2bf16l(const __tile1024i *tsrc, unsigned int row);
```

12.6 Tile Move Row and Convert FP32 to FP16 (TCVTROWPS2PH[H,L])

12.6.1 Description

This operation moves a row from a tile register to a ZMM register, converting the FP32 source elements to FP16. Moves are available from an ACE tile register row; the row is specified using either an immediate or value contained within a 32-bit general purpose register.

Two variants are provided, writing FP16 results in the high 16 bits of each dword or the low 16 bits of each dword. The alternate 16 bits in each dword are written with zeros.

No SIMD exceptions are generated; rounding is round ties to even, performed as if MXCSR.RC=RNE. Embedded rounding is not supported. MXCSR is neither consulted nor updated. No floating-point exceptions are generated. Input FP32 denormals result in FP16 zero output; this instruction can generate FP16 denormal outputs.

12.6.2 Operands

Operand 1	Operand 2	Operand 3
zmm1	tmm2	r32
zmm1	tmm2	imm8

Instruction	Operand 1	Operand 2	Operand 3
TCVTROWPS2PHH	MODRM.REG(w)	MODRM.R/M(r)	VVVV(r)
TCVTROWPS2PHH	MODRM.REG(w)	MODRM.R/M(r)	imm8
TCVTROWPS2PHL	MODRM.REG(w)	MODRM.R/M(r)	VVVV(r)
TCVTROWPS2PHL	MODRM.REG(w)	MODRM.R/M(r)	imm8

12.6.3 Other Exceptions

Immediate form: exception class ACE-E1, see Section 5.7.1. Register form: exception class ACE-E6, see Section 5.7.6.

12.6.4 Operation

```
DEFINE tcvtrowps2ph_imm(src, imm8, variant):
    ASSERT VL in (512)
    row     = imm8[3:0]
    pos     = 1 if variant == "H" else 0
    zeropos = 0 if variant == "H" else 1
    FOR col = 0 TO 15:
        dst.word[2*col + zeropos] = 0
        dst.fp16[2*col + pos] = vcvt_s2h(src.fp32[row][col])  // FP32→FP16, RNE
    dst[MAXVL-1 : VL] = 0
```

```
DEFINE tcvtrowps2ph_gpr(src, r32, variant):
    ASSERT VL in (512)
    row     = r32 & 0xF
    pos     = 1 if variant == "H" else 0
    zeropos = 0 if variant == "H" else 1
    FOR col = 0 TO 15:
        dst.word[2*col + zeropos] = 0
        dst.fp16[2*col + pos] = vcvt_s2h(src.fp32[row][col])  // FP32→FP16, RNE
    dst[MAXVL-1 : VL] = 0
```

12.6.5 Instruction Table

Mnemonic	Operands	Description
TCVTROWPS2PHH	zmm1, tmm2, r32	Move a tile row to zmm1, converting FP32 to FP16 packed into the high word of each dword. Row index selected by GPR or immediate.
	zmm1, tmm2, imm8	
TCVTROWPS2PHL	zmm1, tmm2, r32	Move a tile row to zmm1, converting FP32 to FP16 packed into the low word of each dword. Row index selected by GPR or immediate.
	zmm1, tmm2, imm8	

12.6.6 Masking
None.

12.6.7 Rounding
Round ties to even, performed as if MXCSR.RC=RNE. Embedded rounding is not supported.

12.6.8 Floating-Point Exceptions
None.

12.6.9 Encoding Notes
Only 8 TMM registers are architected; note impact to register specifier field encodings.

12.6.10 C/C++ Compiler Intrinsic Equivalent

```
/* _tile_cvtrowps2phh — high half-word of each dword */
__m512i _tile_cvtrowps2phh(const __tile1024i *tsrc, unsigned int row);

/* _tile_cvtrowps2phl — low half-word of each dword */
__m512i _tile_cvtrowps2phl(const __tile1024i *tsrc, unsigned int row);
```

13 Block Scale Register Operations

13.1 BSRINIT – Block Scale Register Init

13.1.1 Description

BSRINIT initializes all architectural BSR state. All 128 bytes within SCALEDATA (Block Scale register) are set to `0x7F`, corresponding to an E8M0 scale factor of $2^0 = 1.0$. The instruction has no explicit operands; the ModRM byte is fixed at C0.

13.1.2 Other Exceptions

Exception class ACE-E5, see Section 5.7.5.

13.1.3 Operation

```
DEFINE bsrinit():
    // Initialize all architectural BSR state: 128 bytes to 0x7F = E8M0 scale 2^0 = 1.0
    FOR i = 0 TO 127:
        BSR.byte[i] = 0x7F
```

13.1.4 Instruction Table

Mnemonic	Operands	Description
BSRINIT		Initialize all architectural BSR state to 0x7F (E8M0 scale = 1).

13.1.5 Masking

None.

13.1.6 Rounding

N/A.

13.1.7 Floating-Point Exceptions

None.

13.1.8 C/C++ Compiler Intrinsic Equivalent

```
/* _bsrinit – Init all BSR state to 0x7F (E8M0 scale=1) */
void _bsr_init(void);
```

13.2 BSRMOVF – Block Scale Register Move Full

13.2.1 Description

Block Scale Register Full Move operations are provided to facilitate transfer of data to the ACE Block Scale register. Moves 1024-bits of data from the source operands to the Block Scale register. The upper 512-bits of the Block Scale register are provided by the first source operand. The lower 512-bits are provided by the second source operand.

13.2.2 Operands

Operand 1	Operand 2	Operand 3
bsr0	zmm1	zmm2/m512

Instruction	Operand 1	Operand 2	Operand 3
BSRMOVF	MODRM.REG(w)	VVVV(r)	MODRM.R/M(r)

13.2.3 Other Exceptions

Exception class ACE-E2, see Section 5.7.2.

13.2.4 Operation

```
DEFINE bsrmovf(src1, src2):
    ASSERT VL in (512)
    // Load full BSR: src2 → B scales (lower), src1 → A scales (upper)
    BSR[511:0]    = src2[511:0]  // B scales (col/B-input)
    BSR[1023:512] = src1[511:0]  // A scales (row/A-input)
```

13.2.5 Instruction Table

Mnemonic	Operands	Description
BSRMOVF	bsr0, zmm1, zmm2/m512	Write full BSR bsr0 from two ZMM sources: zmm1 provides A-scales, zmm2/m512 provides B-scales.

13.2.6 Masking
None.

13.2.7 Rounding
N/A.

13.2.8 Floating-Point Exceptions
None.

13.2.9 C/C++ Compiler Intrinsic Equivalent

```
/* _bsrmovf – Write full BSR from two ZMMs */
void _bsr_movf(__m512i a_scales, __m512i b_scales);
```

13.3 BSRMOVH / BSRMOVL – Block Scale Register Move Half

13.3.1 Description
Block Scale Register Move Half operations are provided to facilitate transfer of data to and from the ACE Block Scale register. Moves 512-bits of data between the selected half of the Block Scale register and a ZMM register or memory.

13.3.2 Operands
BSRMOVH and BSRMOVL have the same operand encoding form; the tables below apply to both.

Form	Operand 1	Operand 2
Write	bsr0	zmm1/m512
Read	zmm1/m512	bsr0

Instruction	Operand 1	Operand 2	Operand 3
BSRMOVH	MODRM.REG(w)	MODRM.R/M(r)	N/A
BSRMOVH	MODRM.R/M(w)	MODRM.REG(r)	N/A
BSRMOVL	MODRM.REG(w)	MODRM.R/M(r)	N/A
BSRMOVL	MODRM.R/M(w)	MODRM.REG(r)	N/A

13.3.3 Other Exceptions
Exception class ACE-E3, see Section 5.7.3.

13.3.4 Operation

```
DEFINE bsrmovh(src, w1):
    ASSERT VL in (512)
    IF w1:  // W1: load zmm → BSR upper half
        BSR[1023:512] = src[511:0]
    ELSE:  // W0: store BSR upper half → zmm
        dst[511:0] = BSR[1023:512]
        dst[MAXVL-1 : VL] = 0
```

```
DEFINE bsrmovl(src, w1):
    ASSERT VL in (512)
    IF w1:  // W1: load zmm → BSR lower half
        BSR[511:0] = src[511:0]
    ELSE:  // W0: store BSR lower half → zmm
        dst[511:0] = BSR[511:0]
        dst[MAXVL-1 : VL] = 0
```

13.3.5 Instruction Table

Mnemonic	Operands	Description
BSRMOVH	bsr0, zmm1/m512	Move the A-scale half of BSR bsr0 (high 512 bits). Write form: bsr0 ← zmm1/m512. Read form: zmm1/m512 ← bsr0.
	zmm1/m512, bsr0	
BSRMOVL	bsr0, zmm1/m512	Move the B-scale half of BSR bsr0 (low 512 bits). Write form: bsr0 ← zmm1/m512. Read form: zmm1/m512 ← bsr0.
	zmm1/m512, bsr0	

13.3.6 Masking

None.

13.3.7 Rounding

N/A.

13.3.8 Floating-Point Exceptions

None.

13.3.9 C/C++ Compiler Intrinsic Equivalent

```
/* _bsrmovh — write direction */
void _bsr_movh(__m512i a_scales);

/* _bsrmovh_r — read direction */
__m512i _bsr_movh_r(void);
```

```
/* _bsrmovl — write direction */
void _bsr_movl(__m512i b_scales);

/* _bsrmovl_r — read direction */
__m512i _bsr_movl_r(void);
```

14 Tile Outer Product Instructions

14.1 Tile Outer Product MX FP8 Rank 4 (TOP4MX[B|H][B|H]F8PS)

14.1.1 Description

The operations calculate the scaled outer product (rank-4) of two input vectors in MX FP8 format and accumulate with FP32 elements of the destination tile register. The format of the input vectors can independently be chosen as FP8 E4M3 (HF8) or FP8 E5M2 (BF8).

Each 32-bit element of the source ZMM registers contains 4 packed FP8 sub-elements. The output tile element is the sum of 4 FP8 products, scaled by the corresponding E8M0 block scale factor, accumulated with the existing tile element.

14.1.2 Operands

Operand 1	Operand 2	Operand 3	Operand 4
tmm1	zmm2	zmm3	imm8

Instruction	Operand 1	Operand 2	Operand 3	Operand 4
TOP4MXBF8PS TOP4MXBHF8PS TOP4MXHBF8PS TOP4MXHF8PS	MODRM.REG(rw)	MODRM.R/M(r)	VVVV(r)	imm8

14.1.3 Block Scale

OCP MX compliant block scaling is supported with these operations. The Block Scale register is read implicitly by the operation; a sub-element of 16 E8M0 scales is associated with each input operand. The two sub-elements are selected independently using an immediate operand.

The selected 128-bit sub-elements each contain 16 E8M0 scales that are associated with elements in the input data. The input elements are multiplied and summed, then scaled by the corresponding scale terms prior to accumulation with the destination tile. Referring to the table in the Operands section, ASCALE is associated with Operand 2; BSCALE is associated with Operand 3. Refer to Section 10.2.2 and Section 10.5.

Note: OCP FP8 behavior may be achieved by selecting a block scale of 2^0 for both inputs to the operation.

Note: Only bits imm8[1:0] and imm8[4:3] are relevant in specifying the block scale register group; other bits are ignored but are RESERVED and should be encoded as “0”. Software should refrain from using RESERVED encodings, however no #UD or #GP is enforced.

14.1.4 imm8 Encoding

The immediate byte selects the A-scale and B-scale groups from the Block Scale Register:

imm8[7:0]								Notes
7	6	5	4	3	2	1	0	
RSVD	RSVD	RSVD	B_SCALE		RSVD	A_SCALE		A_SCALE, B_SCALE select scale groups 0..3

Each group provides 16 E8M0 scale bytes, one per element (s=0..15), interleaved at stride 4 within each BSR half. The mapping from group value to BSR byte addresses is:

Group	A-scale — BSR.byte[64 + s*4 + group]	B-scale — BSR.byte[s*4 + group]
0	byte[64], [68], [72], …, [124]	byte[0], [4], [8], …, [60]
1	byte[65], [69], [73], …, [125]	byte[1], [5], [9], …, [61]
2	byte[66], [70], [74], …, [126]	byte[2], [6], [10], …, [62]
3	byte[67], [71], [75], …, [127]	byte[3], [7], [11], …, [63]

14.1.5 Other Exceptions

Exception class ACE-E4, see Section 5.7.4.

14.1.6 Operation

```
DEFINE top4mxf8ps(dst, src1, src2, imm8, fp8_format_a, fp8_format_b):
    // MX FP8 rank-4 outer product accumulate into FP32 tile.
    // fp8_format_a, fp8_format_b in {"bf8"=E5M2, "hf8"=E4M3}
    // MXCSR is neither consulted nor updated.
    ASSERT VL in (512)
    a_group = imm8[1:0]  // A-scale group selector
    b_group = imm8[4:3]  // B-scale group selector

    src1_scales = [BSR.byte[64 + s*4 + a_group] for s in range(16)]
    src2_scales = [BSR.byte[     s*4 + b_group] for s in range(16)]

    FOR i = 0 TO dst.rows-1:
        src1_quad  = src1.dword[i]  // 4 packed FP8 bytes for row i
        src1_scale = src1_scales[i]  // E8M0 byte for row i
        FOR j = 0 TO 15:
            src2_quad  = src2.dword[j]  // 4 packed FP8 bytes for col j
            src2_scale = src2_scales[j]  // E8M0 byte for col j
            dst.fp32[i][j] = op4mxf8_subtile(dst.fp32[i][j],
                                             src1_quad, fp8_format_a, src1_scale,
                                             src2_quad, fp8_format_b, src2_scale)
```

```
DEFINE op4mxf8_subtile(srcdest, src1_quad, fmt_a, src1_scale, src2_quad, fmt_b, src2_scale):
    // MXCSR is neither consulted nor updated. No FP exceptions raised.
    // Exceptional value handling:
    //   E8M0 NaN in either scale → QNaN_Indefinite
    //   mult(INF, num) = INF;  mult(INF, zero) = QNaN_Indefinite
    //   mult(NaN, any) = QNaN_Indefinite
    //   sum(+INF, -INF) = QNaN_Indefinite;  sum(NaN, any) = QNaN_Indefinite

    IF src1_scale == 0xFF OR src2_scale == 0xFF:  // E8M0 NaN → propagate QNaN
        RETURN QNaN_Indefinite

    // Accumulate FP8 products in 128-bit integer fixpoint (DAZ=0 for FP8 inputs)
    sop = 0
    FOR i = 0 TO 3:
        s1ei = convert_fp8_to_fixpoint64(src1_quad.byte[i], fmt_a)  // DAZ=0
        s2ei = convert_fp8_to_fixpoint64(src2_quad.byte[i], fmt_b)  // DAZ=0
        sop += s1ei * s2ei  // exact 128-bit integer accumulation

    // Combined exponent adjustment: fp8 fixpoint correction + E8M0 scale shifts.
    //   -factor                          undoes fp8 fixpoint scaling (BF8=2^16, HF8=2^9)
    //   + src1_scale + src2_scale - 254  both E8M0 scales as 2^(s-127) each
    IF   fmt_a == "bf8" AND fmt_b == "bf8":  // BF8 fixpoint scale = 2^16
        factor = 32
    ELSE IF fmt_a == "hf8" AND fmt_b == "hf8":  // HF8 fixpoint scale = 2^9
        factor = 18
    ELSE:  // mixed BF8/HF8
        factor = 25

    exp_adjust = -factor + src1_scale + src2_scale - 254

    // Convert scaled fixpoint sum to FP32 (FTZ=1, RNE).
    sop_fp32 = convert_fixpoint128_scaled_to_fp32_ftz_rne(sop, exp_adjust)
    // Accumulate: DAZ=1 on FP32 tile element (srcdest); daz_b=False because
    // sop_fp32 was produced by convert_fixpoint128_scaled_to_fp32_ftz_rne which
    // already applies FTZ — sop_fp32 is never subnormal on entry.  FTZ=1 on output.
```

```
    RETURN float32_add(srcdest, sop_fp32, daz_a=1, daz_b=0, ftz=1)
```

14.1.7 Instruction Table

Mnemonic	Operands	Description
TOP4MXBF8PS	tmm1, zmm2, zmm3, imm8	Rank-4 MX FP8 outer product. A = FP8 E5M2, B = FP8 E5M2. OCP MX block scaling via BSR. FP32 accumulate.
TOP4MXBHF8PS	tmm1, zmm2, zmm3, imm8	Rank-4 MX FP8 outer product. A = FP8 E5M2, B = FP8 E4M3. OCP MX block scaling via BSR. FP32 accumulate.
TOP4MXHBF8PS	tmm1, zmm2, zmm3, imm8	Rank-4 MX FP8 outer product. A = FP8 E4M3, B = FP8 E5M2. OCP MX block scaling via BSR. FP32 accumulate.
TOP4MXHF8PS	tmm1, zmm2, zmm3, imm8	Rank-4 MX FP8 outer product. A = FP8 E4M3, B = FP8 E4M3. OCP MX block scaling via BSR. FP32 accumulate.

14.1.8 Masking
None.

14.1.9 Rounding
The whole operation always uses RNE.

14.1.10 Floating-Point Exceptions
None.

14.1.11 Encoding Notes
Only 8 TMM registers are architected; note impact to register specifier field encodings.

14.1.12 C/C++ Compiler Intrinsic Equivalent

```
/* Compose the imm8 scale-group selector with | :
 *   ACE_SCALE_A(g)   imm8[1:0]  A-input scale group (0..3)
 *   ACE_SCALE_B(g)   imm8[4:3]  B-input scale group (0..3)
 * imm8[7:6,5,2] are reserved and must be zero. */
#define ACE_SCALE_A(g)   (((g) & 0x3) << 0)
#define ACE_SCALE_B(g)   (((g) & 0x3) << 3)

/* _tile_top4mxbf8ps — BF8 × BF8 */
void _tile_top4mxbf8ps(__tile1024i *tdst, __m512i src1, __m512i src2, const int imm8);

/* _tile_top4mxbhf8ps — BF8 × HF8 */
void _tile_top4mxbhf8ps(__tile1024i *tdst, __m512i src1, __m512i src2, const int imm8);

/* _tile_top4mxhbf8ps — HF8 × BF8 */
void _tile_top4mxhbf8ps(__tile1024i *tdst, __m512i src1, __m512i src2, const int imm8);

/* _tile_top4mxhf8ps — HF8 × HF8 */
void _tile_top4mxhf8ps(__tile1024i *tdst, __m512i src1, __m512i src2, const int imm8);
```

14.2 Tile Outer Product MX INT8 Rank 4 (TOP4MXBSSPS)

14.2.1 Description
The operations calculate the scaled outer product (rank-4) of two input vectors in MX INT8 format and accumulate with FP32 elements of the destination tile register.

Note: The OCP MX INT8 is a fixed-point integer format encoded as signed two's complement. It has an implicit scale of 2^{-6}, refer to the OCP MX specification for further details. The product of two MX INT8 terms will have an implicit scale of 2^{-12}.

Each 32-bit element of the source ZMM registers contains 4 packed signed INT8 sub-elements.

14.2.2 Operands

Operand 1	Operand 2	Operand 3	Operand 4
tmm1	zmm2	zmm3	imm8

Instruction	Operand 1	Operand 2	Operand 3	Operand 4
TOP4MXBSSPS	MODRM.REG(rw)	MODRM.R/M(r)	VVVV(r)	imm8

14.2.3 Block Scale

OCP MX compliant block scaling is supported with these operations. The Block Scale register is read implicitly by the operation; a sub-element of 16 E8M0 scales is associated with each input operand. The two sub-elements are selected independently using an immediate operand.

The selected 128-bit sub-elements each contain 16 E8M0 scales that are associated with elements in the input data. The input elements are multiplied and summed, then scaled by the corresponding scale terms prior to accumulation with the destination tile. Referring to the table in the Operands section, ASCALE is associated with Operand 2; BSCALE is associated with Operand 3. Refer to Section 10.2.2 and Section 10.5.

Note: Only bits imm8[1:0] and imm8[4:3] are relevant in specifying the block scale register group; other bits are ignored but are RESERVED and should be encoded as “0”. Software should refrain from using RESERVED encodings, however no #UD or #GP is enforced.

14.2.4 imm8 Encoding

The immediate byte selects the A-scale and B-scale groups from the Block Scale Register:

imm8[7:0]								Notes
7	6	5	4	3	2	1	0	
RSVD	RSVD	RSVD	B_SCALE		RSVD	A_SCALE		A_SCALE, B_SCALE select scale groups 0..3

Each group provides 16 E8M0 scale bytes, one per element (s=0..15), interleaved at stride 4 within each BSR half. The mapping from group value to BSR byte addresses is:

Group	A-scale — BSR.byte[64 + s*4 + group]	B-scale — BSR.byte[s*4 + group]
0	byte[64], [68], [72], ..., [124]	byte[0], [4], [8], ..., [60]
1	byte[65], [69], [73], ..., [125]	byte[1], [5], [9], ..., [61]
2	byte[66], [70], [74], ..., [126]	byte[2], [6], [10], ..., [62]
3	byte[67], [71], [75], ..., [127]	byte[3], [7], [11], ..., [63]

14.2.5 Other Exceptions

Exception class ACE-E4, see Section 5.7.4.

14.2.6 Operation

```
DEFINE top4mxbssps(dst, src1, src2, imm8):
    // MX INT8 rank-4 outer product accumulate into FP32 tile.
    // MXCSR is neither consulted nor updated. No FP exceptions raised.
    ASSERT VL in (512)
    a_group = imm8[1:0]  // A-scale group selector
    b_group = imm8[4:3]  // B-scale group selector

    src1_scales = [BSR.byte[64 + s*4 + a_group] for s in range(16)]
    src2_scales = [BSR.byte[     s*4 + b_group] for s in range(16)]

    FOR i = 0 TO dst.rows-1:
        src1_quad  = src1.dword[i]
        src1_scale = src1_scales[i]
        FOR j = 0 TO 15:
            src2_quad  = src2.dword[j]
            src2_scale = src2_scales[j]
```

```
            dst.fp32[i][j] = op4mxb_subtile(dst.fp32[i][j],
                                            src1_quad, src1_scale,
                                            src2_quad, src2_scale)
```

```
DEFINE op4mxb_subtile(srcdest, src1_quad, src1_scale, src2_quad, src2_scale):
    // MX INT8 outer product subtile (one output element). "b" = byte (INT8).
    // MXCSR is neither consulted nor updated. No FP exceptions raised.
    // Exceptional value handling:
    //   E8M0 NaN (0xFF) in either scale → NaN output
    //   sum(NaN, any) = QNaN_Indefinite

    IF src1_scale == 0xFF OR src2_scale == 0xFF:  // E8M0 NaN → propagate QNaN
        RETURN QNaN_Indefinite

    // Accumulate signed byte products in 32-bit integer (exact).
    // int8_to_int32 sign-extends each unsigned byte to a signed 32-bit value.
    sop = 0
    FOR i = 0 TO 3:
        sop += int8_to_int32(src1_quad.byte[i]) * int8_to_int32(src2_quad.byte[i])

    // MX INT8 encoding carries an implicit exponent bias of -6 per element term.
    // A product of two such terms accumulates a combined bias of -6*2 = -12 in
    // the sum of products, independent of rank.
    // exp_adjust combines this product bias with the two E8M0 block scale shifts:
    //   -12                              product implicit bias
    //   + src1_scale + src2_scale - 254  both E8M0 block scales as 2^(s-127) each
    exp_adjust = -12 + src1_scale + src2_scale - 254

    // Convert fixpoint sum to FP32 (FTZ=1, RNE).
    sop_fp32 = convert_fixpoint128_scaled_to_fp32_ftz_rne(sop, exp_adjust)
    // Accumulate: DAZ=1 on FP32 tile element (srcdest); daz_b=False because
    // sop_fp32 is never subnormal (FTZ applied by the fixpoint converter above).
    RETURN float32_add(srcdest, sop_fp32, daz_a=1, daz_b=0, ftz=1)
```

14.2.7 Instruction Table

Mnemonic	Operands	Description
TOP4MXBSSPS	tmm1, zmm2, zmm3, imm8	Rank-4 MX INT8 outer product. A = MX INT8 signed, B = MX INT8 signed. OCP MX block scaling via BSR. FP32 accumulate.

14.2.8 Masking
None.

14.2.9 Rounding
The whole operation always uses RNE.

14.2.10 Floating-Point Exceptions
None.

14.2.11 Encoding Notes
Only 8 TMM registers are architected; note impact to register specifier field encodings.

14.2.12 C/C++ Compiler Intrinsic Equivalent

```
/* _tile_top4mxbssps — MX INT8 rank-4 OP S×S; BSR scaled; FP32 accum */
void _tile_top4mxbssps(__tile1024i *tdst, __m512i src1, __m512i src2, const int imm8);
```

14.3 Tile Outer Product BF16 Rank 2 (TOP2BF16PS)

14.3.1 Description
The operation calculates the outer product (rank-2) of two input vectors in BF16 format and accumulates with FP32 elements of the destination tile register. No block scaling is used.

Each 32-bit element of the source ZMM registers contains 2 packed BF16 sub-elements. The output tile element is the sum of 2 BF16 products, accumulated with the existing tile element.

14.3.2 Operands

Operand 1	Operand 2	Operand 3
tmm1	zmm2	zmm3

Instruction	Operand 1	Operand 2	Operand 3
TOP2BF16PS	MODRM.REG(rw)	MODRM.R/M(r)	VVVV(r)

14.3.3 Block Scale

Block scaling is not supported for these operations.

14.3.4 Other Exceptions

Exception class ACE-E4, see Section 5.7.4.

14.3.5 Operation

```
DEFINE top2bf16ps(dst, src1, src2):
    // BF16 rank-2 outer product accumulate into FP32 tile (no block scale).
    // MXCSR is neither consulted nor updated. No FP exceptions raised.
    // DAZ=FTZ=1, RNE.
    ASSERT VL in (512)
    FOR i = 0 TO dst.rows-1:
        op1 = src1.dword[i]  // 2 packed BF16 elements for row i
        FOR j = 0 TO 15:
            op2 = src2.dword[j]  // 2 packed BF16 elements for col j
            dst.fp32[i][j] = op2bf16_subtile(dst.fp32[i][j], op1, op2)
```

```
DEFINE op2bf16_subtile(srcdest, op1, op2):
    // BF16 rank-2 subtile: two BF16 pairs → FP32 SoP → FP32 accumulate.
    // DAZ=FTZ=1, RNE. MXCSR neither consulted nor updated.
    //

    // Step 1: widen both BF16 sub-elements to FP32 (exact; BF16 ⊂ FP32).
    //         DAZ=1: flush BF16 denormals (exp==0, frac!=0) to signed zero.
    a0 = bf16_to_fp32_daz(op1[15:0])  // low  BF16 of src1 lane
    a1 = bf16_to_fp32_daz(op1[31:16])  // high BF16 of src1 lane
    b0 = bf16_to_fp32_daz(op2[15:0])  // low  BF16 of src2 lane
    b1 = bf16_to_fp32_daz(op2[31:16])  // high BF16 of src2 lane

    // Step 2: FP32 products (exact for BF16 inputs) + FP32 horizontal add.
    //         Special cases: NaN×any = QNaN_Indefinite; Inf×0 = QNaN_Indefinite.
    //         daz_a/daz_b omitted (False): BF16-derived products cannot be
    //         FP32 subnormals that require DAZ treatment on input.
    //         ftz=True (default): subnormal SoP result flushed to ±0.
    sop = float32_add(a0 * b0, a1 * b1, daz_a=0, daz_b=0, ftz=1)

    // Step 3: accumulate into FP32 tile element (FP32 addition, RNE, FTZ=1).
    //         daz_a=True: subnormal FP32 accumulator flushed to ±0 before addition,
    //         matching ace_sop_accumulate() which flushes a subnormal acc input.
    RETURN float32_add(srcdest, sop, daz_a=1, daz_b=0, ftz=1)
```

14.3.6 Instruction Table

Mnemonic	Operands	Description
TOP2BF16PS	tmm1, zmm2, zmm3	Rank-2 BF16 outer product. A = BF16, B = BF16. FP32 accumulate. No block scaling.

14.3.7 Masking

None.

14.3.8 Rounding
The whole operation always uses RNE.

14.3.9 Floating-Point Exceptions
None.

14.3.10 Encoding Notes
Only 8 TMM registers are architected; note impact to register specifier field encodings.

14.3.11 C/C++ Compiler Intrinsic Equivalent

```
/* _tile_top2bf16ps – BF16 rank-2 OP; FP32 accum; no scaling */
void _tile_top2bf16ps(__tile1024i *tdst, __m512i src1, __m512i src2);
```

14.4 Tile Outer Product Byte Rank 4 (TOP4B[U|S][U|S]D)

14.4.1 Description
The operations calculate the outer product (rank-4) of two input vectors of bytes and accumulate intermediate word results with INT32 (dword) elements of the destination tile register. There are a total of four variants with independent selection of signed and unsigned format for the input vectors. No block scaling is used.

Each 32-bit element of the source ZMM registers contains 4 packed INT8 sub-elements:

Mnemonic	A signedness	B signedness
TOP4BSSD	Signed	Signed
TOP4BSUD	Signed	Unsigned
TOP4BUSD	Unsigned	Signed
TOP4BUUD	Unsigned	Unsigned

14.4.2 Operands

Operand 1	Operand 2	Operand 3
tmm1	zmm2	zmm3

Instruction	Operand 1	Operand 2	Operand 3
TOP4BSSD TOP4BSUD TOP4BUSD TOP4BUUD	MODRM.REG(rw)	MODRM.R/M(r)	VVVV(r)

14.4.3 Block Scale
Block scaling is not supported for these operations.

14.4.4 Other Exceptions
Exception class ACE-E4, see Section 5.7.4.

14.4.5 Operation

```
DEFINE top4b_ss_d(dst, src1, src2):
    // No DAZ/FTZ: pure integer. No FP exceptions raised or denoted.
    ASSERT VL in (512)
    FOR row = 0 TO dst.rows-1:
        FOR col = 0 TO 15:
            acc = 0
            FOR k = 0 TO 3:
                a = int8_to_int32(src1.int8[row][k])  // signed
                b = int8_to_int32(src2.int8[col][k])  // signed
                acc += a * b
            dst.int32[row][col] += acc
```

```
DEFINE top4b_su_d(dst, src1, src2):
    // No DAZ/FTZ: pure integer. No FP exceptions raised or denoted.
    ASSERT VL in (512)
```

```
    FOR row = 0 TO dst.rows-1:
        FOR col = 0 TO 15:
            acc = 0
            FOR k = 0 TO 3:
                a = int8_to_int32(src1.int8[row][k])  // signed
                b = uint8_to_int32(src2.int8[col][k])  // unsigned
                acc += a * b
            dst.int32[row][col] += acc
```

```
DEFINE top4b_us_d(dst, src1, src2):
    // No DAZ/FTZ: pure integer. No FP exceptions raised or denoted.
    ASSERT VL in (512)
    FOR row = 0 TO dst.rows-1:
        FOR col = 0 TO 15:
            acc = 0
            FOR k = 0 TO 3:
                a = uint8_to_int32(src1.int8[row][k])  // unsigned
                b = int8_to_int32(src2.int8[col][k])  // signed
                acc += a * b
            dst.int32[row][col] += acc
```

```
DEFINE top4b_uu_d(dst, src1, src2):
    // No DAZ/FTZ: pure integer. No FP exceptions raised or denoted.
    ASSERT VL in (512)
    FOR row = 0 TO dst.rows-1:
        FOR col = 0 TO 15:
            acc = 0
            FOR k = 0 TO 3:
                a = uint8_to_int32(src1.int8[row][k])  // unsigned
                b = uint8_to_int32(src2.int8[col][k])  // unsigned
                acc += a * b
            dst.int32[row][col] += acc
```

14.4.6 Instruction Table

Mnemonic	Operands	Description
TOP4BSSD	tmm1, zmm2, zmm3	Rank-4 byte outer product. A = INT8 signed, B = INT8 signed. INT32 accumulate. No block scaling.
TOP4BSUD	tmm1, zmm2, zmm3	Rank-4 byte outer product. A = INT8 signed, B = INT8 unsigned. INT32 accumulate. No block scaling.
TOP4BUSD	tmm1, zmm2, zmm3	Rank-4 byte outer product. A = INT8 unsigned, B = INT8 signed. INT32 accumulate. No block scaling.
TOP4BUUD	tmm1, zmm2, zmm3	Rank-4 byte outer product. A = INT8 unsigned, B = INT8 unsigned. INT32 accumulate. No block scaling.

14.4.7 Masking

None.

14.4.8 Rounding

N/A.

14.4.9 Floating-Point Exceptions

None.

14.4.10 Encoding Notes

Only 8 TMM registers are architected; note impact to register specifier field encodings.

14.4.11 C/C++ Compiler Intrinsic Equivalent

```
/* _tile_top4bssd — signed A × signed B */
void _tile_top4bssd(__tile1024i *tdst, __m512i src1, __m512i src2);

/* _tile_top4bsud — signed A × unsigned B */
```

```
void _tile_top4bsud(__tile1024i *tdst, __m512i src1, __m512i src2);

/* _tile_top4busd — unsigned A × signed B */
void _tile_top4busd(__tile1024i *tdst, __m512i src1, __m512i src2);

/* _tile_top4buud — unsigned A × unsigned B */
void _tile_top4buud(__tile1024i *tdst, __m512i src1, __m512i src2);
```

15 Application/System Programmer Model

15.1 Background

The ACE extensions are presented as an extension within the general AMX framework. The framework is architected to provide implementation freedom to support the original AMX TMUL, ACE, or both.

15.2 AMX State

The AMX (TMUL) architecture introduced two state elements:

- Tile Configuration, TILECFG (64 bytes)
- Tile Register File data, TILEDATA (8192 bytes)

The ACE extensions inherit the existing AMX (TMUL) state elements and introduce a third state element for block scale register state:

- Tile Configuration, TILECFG (64 bytes)
- Tile Register File data, TILEDATA (8192 bytes)
- Block Scale data, SCALEDATA (128 bytes)

15.2.1 AMX Tile Configuration

The AMX architecture requires software to initialize the TILECFG prior to using AMX features. This is achieved by loading a 64-byte TILECFG descriptor from memory using a LDTILECFG instruction.

The TILECFG descriptor includes a palette entry, which selects from the operating modes supported by the AMX implementation. When uninitialized, palette is set to the INIT state (0) and TILEDATA is initialized. Attempting to execute AMX (or ACE) instructions other than LDTILECFG/STTILECFG/TILERELEASE when palette is in the INIT state results in #UD exception.

A process may only use one palette configuration at any point in time; there is no support to interleave instructions requiring different palette configurations within a process, although in principle application code could manage the switching of configuration state by itself.

The original AMX (TMUL) extensions supported two palette IDs:

- 0: INIT
- 1: AMX

The ACE extensions introduce a new palette:

- 2: ACE

Compatible implementations will support palette 0 and one or both of palettes 1 and 2.

15.2.2 Tile Configuration Memory Area Layout

The TILECFG state is only accessible through LDTILECFG/STTILECFG or XSAVE/XRSTOR operations.

The memory layout for the different palette configurations is shown in the tables below. Only palette configurations supported by the implementation may be configured, and palette configurations are validated as part of the LDTILECFG instruction; see Section 15.2.2.4.

15.2.2.1 Palette 0 (INIT)

Byte(s)	Field Name	Description
0	palette	Palette selects the supported configuration of the tiles that will be used
1-63	reserved, must be zero	

15.2.2.2 Palette 1 (TMUL)

Byte(s)	Field Name	Description
0	palette	Palette selects the supported configuration of the tiles that will be used.
1	start_row	start_row is used for storing the restart values for interrupted operations.
2-15	reserved, must be zero	
16-17	tile0.colsb	Tile 0 bytes per row.
18-19	tile1.colsb	Tile 1 bytes per row.
20-21	tile2.colsb	Tile 2 bytes per row.
...	(sequence continues)	
30-31	tile7.colsb	Tile 7 bytes per row.
32-47	reserved, must be zero	
48	tile0.rows	Tile 0 rows.
49	tile1.rows	Tile 1 rows.
50	tile2.rows	Tile 2 rows.
...	(sequence continues)	
55	tile7.rows	Tile 7 rows.
56-63	reserved, must be zero	

15.2.2.3 Palette 2 (ACE)

Byte(s)	Field Name	Description
0	palette	Palette selects the supported configuration of the tiles that will be used
1	reserved, must be zero	Reserved for consistency with start_row use under TMUL/palette 1. Not used by the ACE palette (no TILELOADD)
2-63	reserved, must be zero	

15.2.2.4 Invalid TILECFG Configuration

It is IMPLEMENTATION DEFINED which palettes, palette ID > 0, are supported in an implementation. An implementation may support TMUL, ACE or both palettes.

Loading a TILECFG descriptor specifying invalid configuration will result in a #GP fault:

- palette identifier that is not supported by the implementation
- tile parameters for more tiles than the implementation limit or the palette limit
- non-zero values in reserved fields

15.2.2.5 Palette 2 parameters

Palette 2 has a fixed configuration; dimensions are not configured through the TILECFG memory frame. The Palette 2 configuration parameters are listed in the table below.

Field	Description
BYTES_PER_TILE	Palette 2 BYTES_PER_TILE = 1024.
TOTAL_TILE_BYTES	Palette 2 TOTAL_TILE_BYTES = 8192
MAX_NAMES	Palette 2 MAX_NAMES = 8
BYTES_PER_ROW	Palette 2 BYTES_PER_ROW = 64
TOTAL_SCALE_BYTES	Palette 2 TOTAL_SCALE_BYTES = 128
MAX_ROWS	Palette 2 MAX_ROWS = 16

15.3 Feature Flags and Supported Instructions

AMX/ACE instructions may be executed dependent on the configured palette and feature flags as reported by the implementation. The following feature enumerations are defined:

- AMX-TILE
- ACE
- ACE_VSN

Refer to Section 15.5 for details of CPUID location.

The following table describes instruction support based on palette configuration; “Y” is predicated on the implementation reporting the corresponding feature flag:

Group	Feature Enumeration	Instruction	Palette		
			0 (INIT)	1 (TMUL)	2 (ACE)
AMX Tile	AMX-TILE	LDTILECFG	Y	Y	Y
		STTILECFG	Y	Y	Y
		TILEZERO	N	Y	Y
		TILERELEASE	Y	Y	Y
		TILELOADD	N	Y	N
		TILELOADDT1	N	Y	N
		TILESTORED	N	Y	N
AMX Tile	AMX-AVX512 || ACE_VSN >= 1	TILEMOVROW (R)	N	Y	Y
		TCVTROWD2PS	N	Y	Y
		TCVTROWPS2BF16[H,L]	N	Y	Y
		TCVTROWPS2PH[H,L]	N	Y	Y
ACE Tile	ACE = 1 || ACE_VSN >= 1	TILEMOVROW (W)	N	N	Y
		TILEMOVCOL (W)	N	N	Y
		BSRINIT	N	N	Y
		BSRMOVF	N	N	Y
		BSRMOV[H|L]	N	N	Y
ACE Data Processing	ACE = 1 || ACE_VSN >= 1	TOP4B[SS,SU,US,UU]D	N	N	Y
		TOP4MX[B,H,BH,HB]F8PS	N	N	Y
		TOP4MXB[SS]PS	N	N	Y
		TOP2BF16PS	N	N	Y

Attempts to execute an instruction not supported in the configured palette result in a #UD fault.

Under Palette 2, for ACE, no TILELOAD or TILESTORE instructions are provided; tile registers are used for accumulation with ACE outer products but not source operands to matrix multiplication. Tile move operations are provided to transfer to and from AVX vector registers.

15.4 Changes to XSAVE Feature Set

The ACE ISA is presented as an extension within the AMX framework and uses the XSAVE feature set to save and restore processor state.

15.4.1 XSAVE State Components

The ACE extensions use the following XSAVE state-components controlled by XCR0 for the additional register state used by AMX & ACE extensions:

- XCR0.TILECFG (bit 17) is used for the 64-byte TILECFG state component.
- XCR0.TILEDATA (bit 18) is used for the 8192-byte TILEDATA state component.
- XCR0.SCALEDATA (bit 20) is used for the 128-byte SCALEDATA state component.

Compared to the baseline AMX framework, ACE extends XCR0 to define:

- XCR0.SCALEDATA (bit 20): If 1, and XCR0.TILECFG and XCR0.TILEDATA are also 1, ACE instructions can be executed and the XSAVE feature set can be used to manage SCALEDATA.

The discovery and configuration of these bits is the same as specified in the AMX extensions documentation.

ACE instructions that access AVX state are also dependent on XCR0[7:5, 2:1] and CR0[3], see Section 15.4.6 and Section 5.

The existing AMX framework instructions (LDTILECFG, STTILECFG, TILEZERO, TILERELEASE) remain sensitive to only XCR0[18:17].

15.4.2 XSETBV

The XSETBV general-protection exception (#GP) conditions are extended to include:

- Attempting to set XCR0.SCALEDATA while not setting both XCR0.TILECFG and XCR0.TILEDATA

15.4.3 Extended Feature Disable, IA32_XFD

IA32_XFD is not extended to populate IA32_XFD[20]; system software can disable access to AMX and ACE instructions by setting IA32_XFD[18]. Note IA32_XFD[17] is also not populated in existing or future AMX compatible processors.

Attempts to write IA32_XFD[20] or IA32_XFD[17] to 1 result in #GP.

LDTILECFG and TILERELEASE initialize the TILEDATA and SCALEDATA state components. An execution of either of these instructions does not generate #NM when XCR0[18] = IA32_XFD[18] = 1; instead, it initializes TILEDATA and SCALEDATA normally. As noted above, these instructions are sensitive to XCR0[18:17] only and do not check XCR0[20] independently.

STTILECFG does not use the TILEDATA or SCALEDATA state components. An execution of this instruction does not generate #NM when IA32_XFD[18] = 1.

Register Address (Hex)	Architectural MSR Name
1C4H	IA32_XFD
Extended Feature Disable Control (R/W). Controls which XSAVE-enabled features are temporarily disabled. Available if CPUID.(EAX=0DH,ECX=1): EAX[4] = 1.	
1C5H	IA32_XFD_ERR
Extended Feature Disable Error Code (R/W). Reports which XSAVE-enabled features caused a fault due to being disabled. Available if CPUID.(EAX=0DH,ECX=1): EAX[4] = 1.	

15.4.4 Interaction with XSAVE Instructions

The following items describe special treatment of TILECFG, TILEDATA and SCALEDATA by the XSAVE feature set:

- Loading of TILECFG, TILEDATA and SCALEDATA by XRSTOR and XRSTORS:
  - While the LDTILECFG instruction generates a general-protection fault (#GP) if it would load the TILECFG register with an unsupported value, executions of XRSTOR and XRSTORS do not do so. Instead, they initialize the register (resulting in TILES_CONFIGURED = 0).
  - While executions of LDTILECFG initialize TILEDATA, executions of XRSTOR and XRSTORS do not modify TILEDATA unless loading it from memory.
  - While executions of LDTILECFG initialize SCALEDATA, executions of XRSTOR and XRSTORS do not modify SCALEDATA unless loading it from memory.
  - While the value of the TILECFG register can limit how AMX and ACE instructions access TILEDATA, such limitations do not apply to XRSTOR and XRSTORS. An execution of either of those instructions loads all 8 KBytes of TILEDATA regardless of the value in the TILECFG register.
  - While the value of the TILECFG register can limit how AMX and ACE instructions access SCALEDATA, such limitations do not apply to XRSTOR and XRSTORS. An execution of either of those instructions loads all 128 Bytes of SCALEDATA regardless of the value in the TILECFG register.
  - If XRSTOR or XRSTORS is loading TILEDATA (state component 18) and bit 18 of the XSTATE_BV field of the XSAVE header is 1, the instruction generates #NM when

XCR0[18] = IA32_XFD[18] = 1. If bit 18 of XSTATE_BV is 0, the instruction does not generate #NM; instead, it initializes TILEDATA normally.
    - If XRSTOR or XRSTORS is loading SCALEDATA (state component 20) and bit 20 of the XSTATE_BV field of the XSAVE header is 1, the instruction generates #NM when XCR0[20] = 1 and IA32_XFD[18] = 1. If bit 20 of XSTATE_BV is 0, the instruction does not generate #NM; instead, it initializes SCALEDATA to its initial state (128 bytes of 0x7F).
- Saving of TILEDATA and SCALEDATA by XSAVE, XSAVEC, XSAVEOPT, and XSAVES:
    - While the value of the TILECFG register can limit how AMX and ACE instructions access TILEDATA, such limitations do not apply to XSAVE, XSAVEC, XSAVEOPT, and XSAVES. An execution of any of those instructions saves all 8 KBytes of TILEDATA regardless of the value in the TILECFG register.
    - While the value of the TILECFG register can limit how AMX and ACE instructions access SCALEDATA, such limitations do not apply to XSAVE, XSAVEC, XSAVEOPT, and XSAVES. An execution of any of those instructions saves all 128 Bytes of SCALEDATA regardless of the value in the TILECFG register.
    - When XCR0[18] = IA32_XFD[18] = 1, XSAVE saves TILEDATA as if in its initial state: bit 18 of the XSTATE_BV field is saved as 0, and XSAVE writes the initial TILEDATA value (8192 bytes of zeros). XSAVEC, XSAVEOPT, and XSAVES do not save the state component.
    - When XCR0[20] = 1 and IA32_XFD[18] = 1, XSAVE saves SCALEDATA as if in its initial state: bit 20 of the XSTATE_BV field is saved as 0, and XSAVE writes the initial SCALEDATA value (128 bytes of 0x7F). XSAVEC, XSAVEOPT, and XSAVES do not save the state component.

15.4.5 Processor Tracking of XSAVE-Managed State

- **AMX state.** AMX state is in its initial configuration if the TILECFG register is zero and all tile data are zero.
- **SCALEDATA state.** ACE SCALEDATA state is in its initial configuration if the bytes of the scale register are initialized with 0x7F.

15.4.6 Enabling Access to ACE Extensions

The ACE extensions feature some instructions that only access AMX context and others that access both AMX and AVX state.

In order for ACE instructions to access AMX state, the following conditions must apply:

- CR4.OSXSAVE == 1
- XCR0[20, 18:17] == 111
- IA32_XFD[18] == 0

Additionally, for ACE instructions that access AVX state, the following conditions must apply:

- XCR0[7:5] == 111
- XCR0[2:1] == 11
- CR0.TS[3] == 0

Once ACE has been enabled, system software can disable it by clearing XCR0[20,18:17], by clearing CR4.OSXSAVE, or by clearing any of XCR0[7:5, 2:1].

Attempts to access any ACE instructions disabled through CR4.OSXSAVE or XCR0 result in a #UD fault.

ACE instructions that access AVX state will cause a device-not-available exception (#NM) if CR0.TS[3] is set.

Use of ACE instructions is also disabled if system software has used extended feature disable (XFD) and set IA32_XFD[18] to 1. Attempts to execute any ACE instructions when disabled by IA32_XFD[18] result in a #NM fault.

Access to existing AMX and TMUL instructions remains the same as defined in the Intel SDM (CR4.OSXSAVE, XCR0[18:17], IA32_XFD[18]).

Before disabling access, system software should first initialize ACE state (e.g. by using TILERELEASE). In addition, software should not rely on the state of the tile data after setting IA32_XFD[18]; software should always reload or reinitialize the tile data after clearing IA32_XFD[18].

System software should not use XFD to implement a “lazy restore” approach to management of the TILEDATA state component. This approach will not operate correctly for a variety of reasons. One is that the LDTILECFG and TILERELEASE instructions initialize TILEDATA and SCALEDATA and do not cause an #NM exception. Another is that an execution of XSAVE, XSAVEC, XSAVEOPT, or XSAVES by a user thread will save TILEDATA or SCALEDATA as initialized instead of the data expected by the user thread.

15.5 CPUID State Enumeration (Detailed)

Implementations that support the ACE extensions necessarily support components of AVX10 and AMX extensions.

15.5.1 Feature Flags

Extension	Feature Flag	CPUID Function	CPUID Location	Description
AVX10	**EAX:07H, ECX:1 (Structured Extended Feature Flags Enumeration Sub-Leaf 1)**			
	`AVX10`	EAX:07H, ECX:1	EDX[19]	Supports AVX10
	EAX:24H, ECX:0 (Converged Vector ISA Main Leaf)			
	`AVX10_VSN`	EAX:24H, ECX:0	EBX[7:0]	AVX10 Converged Vector ISA Version
	RESERVED	EAX:24H, ECX:0	EBX[15:8]	
	RESERVED	EAX:24H, ECX:0	EBX[18:16]	“111”
	RESERVED	EAX:24H, ECX:0	EBX[31:19]	
	EAX:24H, ECX:1 (Converged Vector ISA Sub-Leaf 1)			
	`AVX10_V1_AUX`	EAX:24H, ECX:1	ECX[2]	Supports VPDPB[SU,UU,SS]D[,S] and VPDPW[SU,US,UU]D[,S]. Supports AVX10.2 Convert Instructions. Requires “AVX10”.
	`AVX10_V2_AUX`	EAX:24H, ECX:1	ECX[3]	Supports FP32 to FP8 Convert Instructions, supports VUNPACKB, supports VPMOVSSDB. Requires “AVX10”.
AMX	**EAX:07H, ECX:0 (Structured Extended Feature Flags Enumeration Leaf)**			
	`AMX-TILE`	EAX:07H, ECX:0	EDX[24]	Supports AMX tile architecture
ACE v1	**EAX:07H, ECX:1 (Structured Extended Feature Flags Enumeration Sub-Leaf 1)**			
	`ACE`	EAX:07H, ECX:1	ECX[11]	Supports ACE
	EAX:1DH, ECX:0 (Tile Information Main Leaf)			
	`MAX_PALETTE`	EAX:1DH, ECX:0	EAX[31:0]	≥ 2 if ACE implemented
	EAX:1DH, ECX:2 (Tile Palette 2 Sub-Leaf)			
	`ACE_VSN`	EAX:1DH, ECX:2	EAX[7:0]	ACE Major Version >= 1 if ACE implemented. ACE v1 implementations will report ACE_VSN = 1.

CPUID Feature Flag	CPUID Function	Field	Description
`AVX-VNNI-INT8`	Fn0000_0007 ECX 01H	EDX[4]	Supports the AVX-VNNI-INT8 instructions [VEX]

CPUID Feature Flag	CPUID Function	Field	Description
`AVX-VNNI-INT16`	Fn0000_0007 ECX 01H	EDX[10]	Supports the AVX-VNNI-INT16 instructions [VEX]

15.5.2 AMX Enumeration

Field	CPUID Location	Notes
EAX:1DH, ECX:0 (Tile Information Main Leaf)		
MAX_PALETTE	EAX[31:0]	>= 2 if ACE implemented
RESERVED	EBX	0
RESERVED	ECX	0
RESERVED	EDX	0
EAX:1DH, ECX:1 (Tile Palette 1 Sub-Leaf)		
BYTES_PER_TILE	EAX[31:16]	If Palette 1 is not implemented, populate with 0s
TOTAL_TILE_BYTES	EAX[15:0]	If Palette 1 is not implemented, populate with 0s
MAX_NAMES	EBX[31:16]	If Palette 1 is not implemented, populate with 0s
BYTES_PER_ROW	EBX[15:0]	If Palette 1 is not implemented, populate with 0s
RESERVED	ECX[31:16]	If Palette 1 is not implemented, populate with 0s
MAX_ROWS	ECX[15:0]	If Palette 1 is not implemented, populate with 0s
RESERVED	EDX	If Palette 1 is not implemented, populate with 0s
EAX:1DH, ECX:2 (Tile Palette 2 Sub-Leaf)		
RESERVED	EAX[31:8]	
ACE_VSN	EAX[7:0]	ACE ISA major version. If ACE != 1 then ACE_VSN = 0. 0: no ACE present, rest of Tile Palette 2 Sub-Leaf will be populated with 0s. !=0: ACE ISA major version number. ACE v1 implementations will report ACE_VSN = 1. See Section 15.2.2.5 for fixed Palette 2 parameters.
RESERVED	EBX	
RESERVED	ECX	
RESERVED	EDX	
EAX:1EH, ECX:0 (TMUL Information Main Leaf)		
RESERVED	EAX	If Palette 1 is not implemented, populate with 0s
RESERVED	EBX	If Palette 1 is not implemented, populate with 0s
RESERVED	ECX	If Palette 1 is not implemented, populate with 0s
RESERVED	EDX	If Palette 1 is not implemented, populate with 0s
EAX:1EH, ECX:1 (TMUL Information Sub-Leaf 1)		
RESERVED	EAX	If Palette 1 is not implemented, populate with 0s
RESERVED	EBX	If Palette 1 is not implemented, populate with 0s
RESERVED	ECX	If Palette 1 is not implemented, populate with 0s
RESERVED	EDX	If Palette 1 is not implemented, populate with 0s
EAX:1EH, ECX:2–255 (RESERVED sub-leaves)		
RESERVED	EAX	0
RESERVED	EBX	0
RESERVED	ECX	0
RESERVED	EDX	0

15.5.3 XSAVE Enumeration

Field	CPUID Location	Notes
EAX:0DH, ECX:0 (Processor Extended State Enumeration Main Leaf)		

Field	CPUID Location	Notes
EAX	Bits 31–00: Reports the supported bits of the lower 32 bits of XCR0. XCR0[n] can be set to 1 only if EAX[n] is 1.	
	EAX{17}: TILECFG	TILECFG state present
	EAX{18}: TILEDATA	TILEDATA state present
	EAX{20}: SCALEDATA	SCALEDATA state present
EBX	EBX{31:0}	Maximum size (bytes, from the beginning of the XSAVE/XRSTOR save area) required by enabled features in XCR0. May be different than ECX if some features at the end of the XSAVE save area are not enabled.
ECX	ECX{31:0}	Maximum size (bytes, from the beginning of the XSAVE/XRSTOR save area) of the XSAVE/XRSTOR save area required by all supported features in the processor, i.e., all the valid bit fields in XCR0.
EDX	EDX{31:0}	Reports the supported bits of the upper 32 bits of XCR0. XCR0[n+32] can be set to 1 only if EDX[n] is 1. Bits 31–00: Reserved.
EAX:0DH, ECX:1 (Processor Extended State Enumeration Sub-leaf)		
EAX	EAX{0}	XSAVEOPT is available.
	EAX{1}	Supports XSAVEC and the compacted form of XRSTOR if set.
	EAX{2}	Supports XGETBV with ECX = 1 if set.
	EAX{3}	Supports XSAVES/XRSTORS and IA32_XSS if set.
	EAX{4}	Supports extended feature disable (XFD) if set.
	EAX{31:5}	RESERVED
EBX	EBX{31:0}	The size in bytes of the XSAVE area containing all states enabled by XCR0 | IA32_XSS.
ECX	ECX{31:0}	Reports the supported bits of the lower 32 bits of the IA32_XSS MSR. IA32_XSS[n] can be set to 1 only if ECX[n] is 1.
EDX	EDX{31:0}	Reports the supported bits of the upper 32 bits of the IA32_XSS MSR. IA32_XSS[n+32] can be set to 1 only if EDX[n] is 1.

Field	CPUID Location	Notes
EAX:0DH, ECX:{17, 18, 20} (Processor Extended State Enumeration Sub-leaves 17, 18, 20)		
ECX=17: TILECFG; ECX=18: TILEDATA; ECX=20: SCALEDATA		
Leaf 0DH output depends on the initial value in ECX. Each sub-leaf index (starting at position 2) is supported if it corresponds to a supported bit in either the XCR0 register or the IA32_XSS MSR. If ECX contains an invalid sub-leaf index, EAX/EBX/ECX/EDX return 0. Sub-leaf n (0 <= n <= 31) is invalid if sub-leaf 0 returns 0 in EAX[n] and sub-leaf 1 returns 0 in ECX[n]. Sub-leaf n (32 <= n <= 63) is invalid if sub-leaf 0 returns 0 in EDX[n-32] and sub-leaf 1 returns 0 in EDX[n-32].		
EAX	EAX{31:0}	The size in bytes (from the offset specified in EBX) of the save area for an extended state feature associated with a valid sub-leaf index, n.
EBX	EBX{31:0}	The offset in bytes of this extended state component’s save area from the beginning of the XSAVE/XRSTOR area.
ECX	ECX{0}	‘1’ if the bit n (corresponding to the sub-leaf index) is supported in the IA32_XSS MSR; it is clear if bit n is instead supported in XCR0. This field reports 0 if the sub-leaf index, n, is invalid.
	ECX{1}	‘1’ if when the compacted format of an XSAVE area is used, this extended state component located on the next 64-byte boundary following the preceding state component (otherwise, it is located immediately following the preceding state component).
	ECX{2}	‘1’ if the processor supports extended feature disable (XFD) for this state component; ‘0’ otherwise. See Section 15.4.3.

Field	CPUID Location	Notes
	ECX{31:3}	RESERVED
EDX	EDX{31:0}	This field reports 0 if the sub-leaf index, n, is invalid; otherwise it is reserved.

The following table summarises the expected CPUID output for each ACE/AMX XSAVE state component:

Sub-leaf	Component	EAX (size)	EBX (offset)	ECX{2} (XFD)	ECX{1} (align)	ECX{0} (XSS)
17	TILECFG	64	impl. dep.	0	1	0
18	TILEDATA	8192	impl. dep.	1	1	0
20	SCALEDATA	128	impl. dep.	0	1	0

Note: SCALEDATA does not have an independent XFD control (ECX{2} = 0). IA32_XFD does not define a separate bit 20 for SCALEDATA; ACE instructions that access SCALEDATA are disabled via IA32_XFD[18] (TILEDATA). See Section 15.4.3.

15.5.4 Software Discovery of Implemented AMX Palettes

The existing AMX discovery features assume a monotonically increasing accumulation of supported palettes. The MAX_PALETTE field only reveals the highest-numbered palette supported by the implementation; this tends to assume that implemented palettes start from Palette 1 and continue to MAX_PALETTE.

There is no optimized architectural approach to retire palettes that become less useful in the future, nor is there an optimized architectural model to allow implementations to choose non-contiguous subsets. Within the scope of the current AMX enumeration framework, it is proposed that software interrogate the enumeration sub-leaf information of palettes and non-zero entries indicate palettes that are populated in the implementation.

15.5.5 Introducing ACE Using the Current AMX Framework

To support retirement and non-contiguous palette choices in the existing framework, software will need to iterate through the palette space (from 1..MAX_PALETTE) and determine whether each palette is actually implemented by checking that the palette sub-leaf is non-zero.[2]

Within the current discovery framework, the approach to introduce only ACE requires the following changes:

- Enumerate ACE in CPUID EAX:07H, ECX:1: ECX[11] as 1
- Increase MAX_PALETTE to 2
- Populate Tile Palette 1 Sub-Leaf with 0s
- Populate Tile Palette 2 Sub-Leaf:
  - ACE_VSN = 1
- Populate the following fields in Structured Extended Feature Flags Enumeration Main Leaf (CPUID 07H):
  - AMX-TILE: 1
  - All other AMX fields (AMX-INT8, AMX-BF16, AMX-FP16, AMX-COMPLEX): 0
- Populate TMUL Information Main Leaf, TMUL Information Sub-Leaf 1 with 0s
- Require (portable) software to interrogate Tile Palette Sub-Leaves from 1–MAX_PALETTE

[2] It is recommended that software not short-circuit this discovery by cross referencing silicon stepping CPUID information.

16 Pseudocode Helper Functions

The following pseudocode helper functions are used throughout the pseudocode descriptions in this specification. They are presented here as a consolidated reference.

16.1 Conversions from FP32

```
DEFINE fp32_to_fp8_e5m2(i, saturating, rounding, bias=0):
    s_i = i[31]
    e_i = (i >> 23) & 0xFF
    m_i =  i        & 0x7FFFFF

    IF e_i == 0xFF:  // Inf or NaN
        IF saturating:
            IF m_i == 0:
                e_o, m_o = 0x1E, 0x3  // clamp to max_normal
            ELSE:
                e_o = 0x1F
                m_o = 0x2 | (m_i[21])
        ELSE:
            e_o = 0x1F
            m_o = 0x2 | (m_i[21]) if m_i != 0 else 0x0

    ELSE IF e_i == 0x00:  // Zero or denorm
        e_o, m_o = 0, 0

    ELSE IF rounding == "RNE":  // Normalised, round-to-nearest-even
        newexp = e_i - 127 + 15
        IF newexp >= 31:  // overflow
            IF saturating:
                e_o, m_o = 0x1E, 0x3
            ELSE:
                e_o, m_o = 0x1F, 0x0
        ELSE IF newexp <= 0:  // underflow → subnormal or zero
            e_o, m_o = 0, 0
            IF (22 - newexp) <= 24:
                mant  = m_i | 0x800000
                shift = 22 - newexp
                m_o   = mant >> shift
                low   = mant & mask(shift)
                half  = 1 << (shift - 1)
                IF low > half OR (low == half AND (m_o & 0x1)):
                    m_o += 1
                    IF (m_o & 0x3) == 0:
                        e_o += 1
        ELSE:  // normal
            e_o = newexp
            m_o = m_i >> 21
            IF m_i & 0x100000:
                IF ((m_i & 0x1FFFFF) > 0x100000) OR (m_o & 0x1):
                    IF NOT (saturating AND e_o == 0x1E AND m_o == 0x3):
                        m_o += 1
                        IF (m_o & 0x3) == 0:
                            e_o += 1

    ELSE:  // BIAS rounding
        e_b = e_i
        m_b = m_i + (bias & 0x1FFFFF)
        IF m_b & 0xFF800000:
            e_b += 1
```

```
        m_b &= 0x7FFFFF
        newexp = e_b - 127 + 15
        IF newexp >= 31:
            IF saturating:
                e_o, m_o = 0x1E, 0x3
            ELSE:
                e_o, m_o = 0x1F, 0x0
        ELSE IF newexp <= 0:
            e_o, m_o = 0, 0
            IF (22 - newexp) <= 24:
                mant  = m_b | 0x800000
                shift = 22 - newexp
                m_o   = mant >> shift
        ELSE:
            e_o = newexp
            m_o = m_b >> 21

    RETURN (s_i & 0x1) << 7 | (e_o & 0x1F) << 2 | (m_o & 0x3)
```

```
DEFINE fp32_to_fp8_e4m3(i, saturating, rounding, bias=0):
    s_i = i[31]
    e_i = (i >> 23) & 0xFF
    m_i =  i        & 0x7FFFFF

    IF e_i == 0xFF:  // Inf or NaN
        e_o, m_o = 0xF, 0x7
        IF saturating AND m_i == 0:
            m_o = 0x6  // clamp to max_normal

    ELSE IF e_i == 0x00:  // Zero or denorm
        e_o, m_o = 0, 0

    ELSE IF rounding == "RNE":
        newexp = e_i - 127 + 7
        IF newexp >= 16:  // overflow
            e_o = 0xF
            m_o = 0x6 if saturating else 0x7
        ELSE IF newexp <= 0:  // underflow
            e_o, m_o = 0, 0
            IF (21 - newexp) <= 24:
                mant  = m_i | 0x800000
                shift = 21 - newexp
                m_o   = mant >> shift
                low   = mant & mask(shift)
                half  = 1 << (shift - 1)
                IF low > half OR (low == half AND (m_o & 0x1)):
                    m_o += 1
                    IF (m_o & 0x7) == 0:
                        e_o += 1
        ELSE:  // normal
            e_o = newexp
            m_o = m_i >> 20
            IF saturating AND e_o == 0xF AND m_o == 0x7:
                m_o = 0x6
            IF m_i & 0x80000:
                IF ((m_i & 0xFFFFF) > 0x80000) OR (m_o & 0x1):
                    clamp_sat  = saturating AND e_o == 0xF AND m_o == 0x6
                    clamp_nan  = (NOT saturating) AND e_o == 0xF AND m_o == 0x7
                    IF NOT (clamp_sat OR clamp_nan):
                        m_o += 1
```

```
                        IF (m_o & 0x7) == 0:
                            e_o += 1

    ELSE IF rounding == "RTO":
        newexp = e_i - 127 + 7
        IF newexp >= 16:
            e_o = 0xF
            m_o = 0x6 if saturating else 0x7
        ELSE IF newexp <= 0:
            e_o, m_o = 0, 0
            IF (21 - newexp) <= 24:
                // FP32 normal, magnitude in E4M3 subnormal range (shift ≤ 24).
                // J-bit at or above bit 0 of the E4M3 mantissa field;
                // sticky bit propagates any nonzero residual under RTO.
                mant   = m_i | 0x800000
                shift  = 21 - newexp
                m_o    = mant >> shift
                sticky = 1 if (mant & mask(shift)) else 0
                m_o   |= sticky
            ELSE:
                // FP32 normal with magnitude < 2^-10 (shift > 24): J-bit falls
                // below bit 0 of the E4M3 subnormal mantissa field.
                m_o = 1
        ELSE:
            e_o = newexp
            m_o = m_i >> 20
            sticky = 1 if (m_i & 0xFFFFF) else 0
            m_o   |= sticky
            IF saturating AND e_o == 0xF AND m_o == 0x7:  // clamp NaN→max_normal
                m_o = 0x6

    ELSE:  // BIAS rounding
        e_b = e_i
        m_b = m_i + (bias & 0xFFFFF)  // 20-bit bias: covers 23→3 bit truncation
        IF m_b & 0xFF800000:
            e_b += 1
        m_b &= 0x7FFFFF
        newexp = e_b - 127 + 7
        IF newexp >= 16:
            e_o = 0xF
            m_o = 0x6 if saturating else 0x7
        ELSE IF newexp <= 0:
            e_o, m_o = 0, 0
        ELSE:
            e_o = newexp
            m_o = m_b >> 20

    RETURN (s_i & 0x1) << 7 | (e_o & 0xF) << 3 | (m_o & 0x7)
```

```
DEFINE fp32_to_bfloat16(x):
    // FP32 → BF16. RNE rounding via integer-add-then-truncate.
    // Zeros and denormals flush to signed zero (DAZ=1 implied).
    // NaN: truncate upper 16 bits and force QNaN bit (bit [6] of result).
    // Inf: truncate upper 16 bits directly.
    // Normal: add RNE bias (0x7FFF + LSB of truncated result), take upper 16 bits.
    s = x[31]
    e = (x >> 23) & 0xFF
    m = x & 0x7FFFFF
    IF e == 0:  // zero or denormal → signed zero
        RETURN (s << 15)
```

```
    ELSE IF e == 0xFF AND m == 0:  // Inf → propagate
        RETURN (x >> 16) & 0xFFFF
    ELSE IF e == 0xFF:  // NaN → truncate, force QNaN bit
        dest = (x >> 16) & 0xFFFF
        dest |= (1 << 6)  // set MSB of BF16 mantissa (QNaN bit)
        RETURN dest
    ELSE:  // normal: RNE via integer add
        lsb          = x[16]
        rounding_bias = 0x7FFF + lsb
        temp          = (x + rounding_bias) & UINT32_MAX
        RETURN (temp >> 16) & 0xFFFF
```

16.2 Conversions from FP16

```
DEFINE fp16_to_fp8_e5m2(i, saturating, rounding, bias=0):
    // FP16 S1E5M10 → FP8 E5M2; rounding in {"RNE", "SR"}
    // SR: bias = lower 8 bits of the per-element bias byte, no shift (E5M2 target)
    s_i = (i & 0x8000) >> 15
    e_i = (i & 0x7C00) >> 10
    m_i = (i & 0x03FF) >> 0

    IF e_i == 0x1F:  // Inf or NaN (same for all rounding modes)
        e_o = 0x1F
        m_o = 0x0
        IF m_i == 0x0:  // Inf → clamp or propagate
            IF saturating:
                e_o, m_o = 0x1E, 0x3
        ELSE:  // NaN → preserve payload bits
            m_o = 0x2 | (m_i[8])

    ELSE IF e_i == 0x00:
        IF m_i == 0x0:  // zero
            e_o = m_o = 0
        ELSE:  // FP16 subnormal → normalize then apply rounding
            mant         = m_i
            exp_unbiased = -14
            WHILE (mant & 0x400) == 0:
                mant <<= 1
                exp_unbiased -= 1
            m_n = mant & 0x3FF
            e_n = exp_unbiased + 15  // virtual FP16-normal exponent field

            IF rounding == "RNE":
                newexp = e_n
                IF newexp >= 31:  // overflow
                    e_o, m_o = (0x1E, 0x3) if saturating else (0x1F, 0x0)
                ELSE IF newexp <= 0:  // underflow into FP8 subnormal range
                    e_o = m_o = 0
                    IF (9 - newexp) <= 11:
                        mant_u  = m_n | 0x400
                        shift   = 9 - newexp
                        m_o     = mant_u >> shift
                        lowmant = mant_u & mask(shift)
                        halfway = 1 << (shift - 1)
                        IF lowmant > halfway OR (lowmant == halfway AND (m_o & 0x1)):
                            m_o += 1
                            IF (m_o & 0x3) == 0:
                                e_o += 1
                ELSE:  // normal FP8 result
                    e_o     = newexp
```

```
                    m_o     = m_n >> 8
                    lowmant = m_n & 0xFF
                    halfway = 0x80
                    IF lowmant > halfway OR (lowmant == halfway AND (m_o & 0x1)):
                        IF NOT (saturating AND e_o == 0x1E AND m_o == 0x3):
                            m_o += 1
                            IF (m_o & 0x3) == 0:
                                e_o += 1
                    IF e_o >= 0x1F:  // post-rounding overflow
                        e_o, m_o = (0x1E, 0x3) if saturating else (0x1F, 0x0)

            ELSE:  // SR: add bias to normalized mantissa, truncate
                e_b = e_n
                m_b = m_n + (bias & 0xFF)  // lower 8 bits of bias, no shift for E5M2
                IF m_b & 0xFC00:  // carry out of 10-bit mantissa
                    e_b += 1
                m_b &= 0x3FF
                IF e_b >= 0x1F:
                    e_o, m_o = (0x1E, 0x3) if saturating else (0x1F, 0x0)
                ELSE IF e_b <= 0:
                    e_o = m_o = 0
                    IF (9 - e_b) <= 11:
                        mant_u = m_b | 0x400
                        shift  = 9 - e_b
                        m_o    = mant_u >> shift  // truncate; no rounding in SR path
                ELSE:
                    e_o = e_b
                    m_o = m_b >> 8  // truncate to 2-bit mantissa

        ELSE:  // FP16 normal: e_i in [1..30]
            IF rounding == "RNE":  // same bias, no rebias needed
                e_o     = e_i
                m_o     = m_i >> 8
                lowmant = m_i & 0xFF
                halfway = 0x80
                IF lowmant > halfway OR (lowmant == halfway AND (m_o & 0x1)):
                    IF NOT (saturating AND e_o == 0x1E AND m_o == 0x3):
                        m_o += 1
                        IF (m_o & 0x3) == 0:
                            e_o += 1
                IF e_o >= 0x1F:  // post-rounding overflow
                    e_o, m_o = (0x1E, 0x3) if saturating else (0x1F, 0x0)

            ELSE:  // SR: add bias to 10-bit mantissa, truncate
                e_b = e_i
                m_b = m_i + (bias & 0xFF)  // lower 8 bits of bias, no shift for E5M2
                IF m_b & 0xFC00:  // carry out of 10-bit mantissa
                    e_b += 1
                m_b &= 0x3FF
                IF e_b >= 0x1F:
                    e_o, m_o = (0x1E, 0x3) if saturating else (0x1F, 0x0)
                ELSE:
                    e_o = e_b
                    m_o = m_b >> 8  // truncate to 2-bit mantissa

        RETURN ((s_i & 0x1) << 7) | ((e_o & 0x1F) << 2) | (m_o & 0x3)
```

```
DEFINE fp16_to_fp8_e4m3(i, saturating, rounding, bias=0):
    // FP16 S1E5M10 → FP8 E4M3; rounding in {"RNE", "SR"}
    // SR: bias = lower 8 bits >> 1 (E4M3 target, one fewer bit discarded than E5M2)
```

```
    s_i = (i & 0x8000) >> 15
    e_i = (i & 0x7C00) >> 10
    m_i = (i & 0x03FF) >> 0

    IF e_i == 0x1F:  // Inf or NaN → NaN (E4M3 has no Inf)
        e_o = 0xF
        m_o = 0x7
        IF saturating AND m_i == 0x0:  // Inf + saturate → max_normal
            m_o = 0x6

    ELSE IF e_i == 0x00:
        e_o = m_o = 0  // zero or FP16 subnormal → zero
                                                // (FP16 subnormals are all below E4M3
                                                minimum)
    ELSE:  // FP16 normal number
        IF rounding == "RNE":
            newexp = e_i - 15 + 7  // rebias FP16 (bias=15) → E4M3 (bias=7)
            IF newexp >= 16:  // overflow above E4M3 max
                e_o = 0xF
                m_o = 0x6 if saturating else 0x7
            ELSE IF newexp <= 0:  // underflow into E4M3 subnormal range
                e_o = m_o = 0
                IF (8 - newexp) <= 11:
                    mant    = m_i | 0x400
                    shift   = 8 - newexp
                    m_o     = mant >> shift
                    lowmant = mant & mask(shift)
                    halfway = 1 << (shift - 1)
                    IF lowmant > halfway OR (lowmant == halfway AND (m_o & 0x1)):
                        m_o += 1
                        IF (m_o & 0x7) == 0:
                            e_o += 1
            ELSE:  // normal E4M3 result
                e_o = newexp
                m_o = m_i >> 7
                IF saturating AND e_o == 0xF AND m_o == 0x7:
                    m_o = 0x6  // pre-clamp before rounding check
                // Gbit = m_i[6]; sticky = m_i[5:0] > 0
                IF m_i & 0x40:  // guard bit set
                    IF (m_i & 0x3F) > 0 OR (m_o & 0x1):
                        IF NOT (saturating AND e_o == 0xF AND m_o == 0x6):
                            IF NOT ((NOT saturating) AND e_o == 0xF AND m_o == 0x7):
                                m_o += 1
                                IF (m_o & 0x7) == 0:
                                    e_o += 1

        ELSE:  // SR: add shifted bias to mantissa, truncate
            e_b = e_i
            m_b = m_i + ((bias & 0xFF) >> 1)  // lower 8 bits of bias, shifted right 1 for
            E4M3
            IF m_b & 0xFC00:  // carry out of 10-bit mantissa
                e_b += 1
            m_b &= 0x3FF
            newexp = e_b - 15 + 7  // rebias after bias addition
            IF newexp >= 16:
                e_o = 0xF
                m_o = 0x6 if saturating else 0x7
            ELSE IF newexp <= 0:
                e_o = m_o = 0
                IF (8 - newexp) <= 11:
```

```
                    mant = m_b | 0x400
                    shift = 8 - newexp
                    m_o   = mant >> shift  // truncate; no rounding in SR path
            ELSE:
                e_o = newexp
                m_o = m_b >> 7  // truncate to 3-bit mantissa
                IF saturating AND e_o == 0xF AND m_o == 0x7:
                    m_o = 0x6

    RETURN ((s_i & 0x1) << 7) | ((e_o & 0xF) << 3) | (m_o & 0x7)
```

16.3 Conversions from FP8

```
DEFINE fp8_e5m2_to_fp32(i):
    s_i = (i & 0x80) >> 7
    e_i = (i & 0x7C) >> 2
    m_i = (i & 0x03) >> 0

    IF e_i == 0x1F:  // Inf or NaN
        e_o = 0xFF
        m_o = 0
        IF m_i != 0x0:
            m_o = (m_i | 0x2) << 21  // set qBit (bit 22 of FP32 mantissa)
    ELSE IF e_i == 0x0:
        IF m_i == 0x0:  // zero
            e_o, m_o = 0, 0
        ELSE IF m_i == 0x1:  // subnormal 0.01 × 2^-14 = 1.0 × 2^-16
            e_o = 127 - 16
            m_o = 0
        ELSE:  // subnormal 0.10 or 0.11 × 2^-14
            e_o = 127 - 15
            m_o = (m_i & 0x1) << 22
    ELSE:  // normal
        e_o = e_i + (127 - 15)
        m_o = m_i << 21

    RETURN ((s_i & 0x1) << 31) | ((e_o & 0xFF) << 23) | m_o
```

```
DEFINE fp8_e4m3_to_fp32(i):
    s_i = (i & 0x80) >> 7
    e_i = (i & 0x78) >> 3
    m_i = (i & 0x07) >> 0

    IF e_i == 0xF:
        IF m_i == 0x7:  // NaN → FP32 QNaN
            e_o = 0xFF
            m_o = (m_i | 0x4) << 20  // set qBit (bit 22 of FP32)
        ELSE:  // max_normal (E4M3 has no Inf)
            e_o = e_i + (127 - 7)
            m_o = m_i << 20
    ELSE IF e_i == 0x0:
        IF m_i == 0x0:  // zero
            e_o, m_o = 0, 0
        ELSE IF m_i == 0x1:  // 0.001 × 2^-6 = 2^-9
            e_o = 127 - 9
            m_o = 0
        ELSE IF (m_i & 0x4) == 0x0:  // 0.0xy × 2^-6, x or y set
            e_o = 127 - 8
            m_o = (m_i & 0x1) << 22
        ELSE:  // 0.1mm × 2^-6 = 1.mm × 2^-7
            e_o = 127 - 7
```

```
            m_o = (m_i & 0x3) << 21
    ELSE:  // normal
        e_o = e_i + (127 - 7)
        m_o = m_i << 20

    RETURN ((s_i & 0x1) << 31) | ((e_o & 0xFF) << 23) | m_o
```

```
DEFINE fp8_e4m3_to_fp16(i):
    // FP8 E4M3 → FP16 E5M10 (lossless widening; all E4M3 values representable in FP16)
    s_i = (i & 0x80) >> 7
    e_i = (i & 0x78) >> 3
    m_i = (i & 0x07) >> 0

    IF e_i == 0xF:
        IF m_i == 0x7:  // E4M3 NaN → FP16 NaN (preserve payload)
            e_o = 0x1F
            m_o = m_i << 7
        ELSE:  // E4M3 max_normal
            e_o = e_i + (15 - 7)  // rebias E4M3 (bias=7) → FP16 (bias=15)
            m_o = m_i << 7
    ELSE IF e_i == 0x0:
        IF m_i == 0x0:  // zero
            e_o = m_o = 0
        ELSE IF m_i == 0x1:  // 0.001 × 2^-6 = 2^-9
            e_o = 15 - 9
            m_o = 0
        ELSE IF (m_i & 0x4) == 0x0:  // 0.0xy × 2^-6, leading zero in mantissa
            e_o = 15 - 8
            m_o = (m_i & 0x1) << 9
        ELSE:  // 0.1mm × 2^-6 = 1.mm × 2^-7
            e_o = 15 - 7
            m_o = (m_i & 0x3) << 8
    ELSE:  // normal
        e_o = e_i + (15 - 7)
        m_o = m_i << 7

    RETURN ((s_i & 0x1) << 15) | ((e_o & 0x1F) << 10) | (m_o & 0x3FF)
```

```
DEFINE fp8_e5m2_to_fp6_e3m2(i):
    // FP8 E5M2 → FP6 E3M2; saturate overflow; zero subnormals
    s_i = (i & 0x80) >> 7
    e_i = (i & 0x7C) >> 2
    m_i = (i & 0x03) >> 0
    e_o = m_o = 0
    exp_rebias = 15 - 3  // FP6 E3M2 bias = 3; FP8 E5M2 bias = 15
    new_exp    = e_i - exp_rebias

    IF (e_i == 0x1F AND m_i != 0x0) OR (e_i == 0x1F AND m_i == 0x0):  // NaN/Inf
        e_o, m_o = 0x7, 0x3  // clamp to FP6 max
    ELSE IF (e_i > (exp_rebias + 7)) OR (e_i == (exp_rebias + 7) AND m_i > 0x3):
        e_o, m_o = 0x7, 0x3  // overflow → clamp
    ELSE IF (e_i == 0x00 AND m_i == 0x0) OR (e_i == 0x0 AND m_i != 0x0):
        e_o = m_o = 0  // zero / subnorm → zero
    ELSE IF new_exp <= 0:  // underflow → subnormal FP6 or zero
        IF (1 - new_exp) <= 3:
            mant    = m_i | 0x4
            shift   = 1 - new_exp
            m_o     = mant >> shift
            lowmant = mant & mask(shift)
            halfway = 1 << (shift - 1)
```

```
            IF lowmant > halfway OR (lowmant == halfway AND (m_o & 0x1)):
                m_o += 1
                IF (m_o & 0x3) == 0:
                    e_o += 1
    ELSE:  // normal: direct rebias (mantissa same width)
        e_o = e_i - exp_rebias
        m_o = m_i

    RETURN ((s_i & 0x1) << 5) | ((e_o & 0x7) << 2) | (m_o & 0x3)
```

```
DEFINE fp8_e4m3_to_fp6_e2m3(i):
    // FP8 E4M3 → FP6 E2M3; saturate overflow; zero subnormals
    s_i = (i & 0x80) >> 7
    e_i = (i & 0x78) >> 3
    m_i = (i & 0x07) >> 0
    e_o = m_o = 0
    exp_rebias = 7 - 1  // FP6 E2M3 bias = 1; FP8 E4M3 bias = 7
    new_exp    = e_i - exp_rebias

    IF (e_i == 0xF AND m_i != 0x0) OR (e_i == 0xF AND m_i == 0x0):  // NaN/max
        e_o, m_o = 0x3, 0x7  // clamp to FP6 max
    ELSE IF (e_i > (exp_rebias + 3)) OR (e_i == (exp_rebias + 3) AND m_i > 0x7):
        e_o, m_o = 0x3, 0x7  // overflow → clamp
    ELSE IF e_i == 0x00 AND m_i == 0x0:
        e_o = m_o = 0  // zero
    ELSE IF new_exp <= 0:  // underflow
        IF (1 - new_exp) <= 4:
            mant    = m_i | 0x8
            shift   = 1 - new_exp
            m_o     = mant >> shift
            lowmant = mant & mask(shift)
            halfway = 1 << (shift - 1)
            IF lowmant > halfway OR (lowmant == halfway AND (m_o & 0x1)):
                m_o += 1
                IF (m_o & 0x7) == 0:
                    e_o += 1
    ELSE:  // normal: direct rebias (mantissa same width)
        e_o = e_i - exp_rebias
        m_o = m_i

    RETURN ((s_i & 0x1) << 5) | ((e_o & 0x3) << 3) | (m_o & 0x7)
```

```
DEFINE fp8_e4m3_to_fp4_e2m1(i):
    // FP8 E4M3 → FP4 E2M1; NaN (0x7F) → clamp; RNE in subnormal
    s_i = (i & 0x80) >> 7
    e_i = (i & 0x78) >> 3
    m_i = (i & 0x07) >> 0
    e_o = m_o = 0
    exp_rebias = 7 - 1  // FP4 E2M1 bias = 1; FP8 E4M3 bias = 7
    new_exp    = e_i - exp_rebias

    IF e_i == 0xF AND m_i == 0x7:  // NaN
        e_o, m_o = 0x3, 0x1
    ELSE IF (e_i > (exp_rebias + 3)) OR (e_i == (exp_rebias + 3) AND m_i > 0x4):
        e_o, m_o = 0x3, 0x1  // overflow → clamp
    ELSE IF (e_i == 0x00 AND m_i == 0x0) OR (e_i == 0x00 AND m_i != 0x0):
        e_o = m_o = 0  // zero / denorm → zero
    ELSE IF new_exp <= 0:  // underflow → subnormal or zero
        IF (3 - new_exp) <= 4:
            mant    = m_i | 0x8  // restore hidden bit
```

```
            shift   = 3 - new_exp
            m_o     = mant >> shift
            lowmant = mant & mask(shift)
            halfway = 1 << (shift - 1)
            IF lowmant > halfway OR (lowmant == halfway AND (m_o & 0x1)):
                m_o += 1
                IF (m_o & 0x1) == 0:
                    e_o += 1
    ELSE:  // normal
        fixup = m_i[2]
        rnex  = i + 0x01 + fixup
        e_o   = ((rnex & 0x78) >> 3) - exp_rebias
        m_o   = (rnex & 0x07) >> 2

    RETURN ((s_i & 0x1) << 3) | ((e_o & 0x3) << 1) | (m_o & 0x1)
```

```
DEFINE fp8_e5m2_to_fp4_e2m1(i):
    // FP8 E5M2 → FP4 E2M1; overflow → max_normal (0x3, 0x1); RNE in subnormal
    s_i = (i & 0x80) >> 7
    e_i = (i & 0x7C) >> 2
    m_i = (i & 0x03) >> 0
    e_o = m_o = 0
    exp_rebias = 15 - 1  // FP4 E2M1 bias = 1; FP8 E5M2 bias = 15
    new_exp    = e_i - exp_rebias

    IF (e_i == 0x1F AND m_i != 0x0) OR (e_i == 0x1F AND m_i == 0x0):  // NaN/Inf
        e_o, m_o = 0x3, 0x1
    ELSE IF (e_i > (exp_rebias + 3)) OR (e_i == (exp_rebias + 3) AND m_i > 0x2):
        e_o, m_o = 0x3, 0x1  // overflow → clamp
    ELSE IF e_i == 0x00 AND m_i == 0x0:
        e_o = m_o = 0  // zero
    ELSE IF new_exp <= 0:  // underflow → subnormal or zero
        // J-bit insertion + RNE for subnormal FP4 output
        IF (2 - new_exp) <= 3:
            mant    = m_i | 0x4  // restore hidden bit
            shift   = 2 - new_exp
            m_o     = mant >> shift
            lowmant = mant & mask(shift)
            halfway = 1 << (shift - 1)
            IF lowmant > halfway OR (lowmant == halfway AND (m_o & 0x1)):
                m_o += 1
                IF (m_o & 0x1) == 0:  // FP4 mantissa is 1-bit
                    e_o += 1
    ELSE:  // normal: direct rebias + truncate
        fixup = m_i[1]
        rnex  = i + fixup
        e_o   = ((rnex & 0x7C) >> 2) - exp_rebias
        m_o   = (rnex & 0x03) >> 1

    RETURN ((s_i & 0x1) << 3) | ((e_o & 0x3) << 1) | (m_o & 0x1)
```

16.4 Conversions from FP6 and FP4

```
DEFINE fp4_e2m1_to_fp8_e4m3(i):
    // FP4 E2M1 → FP8 E4M3 (lossless; all FP4 values are representable in E4M3)
    s_i = (i & 0x8) >> 3
    // 8-entry LUT for unsigned FP4 magnitudes → FP8 E4M3 unsigned encoding
    lut = {
        0x0: 0x00,  //  0.0
        0x1: 0x30,  //  0.5
        0x2: 0x38,  //  1.0
```

```
        0x3: 0x3C,  //  1.5
        0x4: 0x40,  //  2.0
        0x5: 0x44,  //  3.0
        0x6: 0x48,  //  4.0
        0x7: 0x4C,  //  6.0
    }
    res = lut[i & 0x7] | (s_i << 7)
    RETURN res
```

```
DEFINE fp6_e3m2_to_fp8_e4m3(i):
    // FP6 E3M2 → FP8 E4M3 (lossless widening; all FP6 values representable in E4M3)
    // FP6 E3M2: sign[5], exp[4:2] (bias=3), frac[1:0]
    // FP8 E4M3: sign[7], exp[6:3] (bias=7), frac[2:0]
    s_i = (i & 0x20) >> 5
    e_i = (i & 0x1C) >> 2
    m_i = (i & 0x03) >> 0

    IF e_i == 0x0:  // FP6 subnormal: exp=0, frac in [0..3]
        IF m_i == 0:
            e_o = m_o = 0  // zero
        ELSE IF m_i == 1:  // 0.01 × 2^-2 = 2^-4 → FP8 normal e=3,m=0
            e_o = 3
            m_o = 0
        ELSE:  // m_i=2: 0.10×2^-2=2^-3; m_i=3: 0.11×2^-2
            e_o = 4
            m_o = (m_i & 0x1) << 2  // low bit of frac → FP8 mantissa[2] (I-75)
    ELSE:  // FP6 normal: rebias 3→7, shift mantissa left 1
        e_o = e_i + (7 - 3)  // rebias FP6 E3M2 (bias=3) → FP8 E4M3 (bias=7)
        m_o = m_i << 1  // 2-bit frac → bits [2:1] of 3-bit frac

    RETURN ((s_i & 0x1) << 7) | ((e_o & 0xF) << 3) | (m_o & 0x7)
```

```
DEFINE fp6_e2m3_to_fp8_e4m3(i):
    // FP6 E2M3 → FP8 E4M3 (lossless widening)
    // FP6 E2M3: sign[5], exp[4:3] (bias=1), frac[2:0]
    // FP8 E4M3: sign[7], exp[6:3] (bias=7), frac[2:0]
    s_i = (i & 0x20) >> 5
    e_i = (i & 0x18) >> 3
    m_i = (i & 0x07) >> 0

    IF e_i == 0x0:  // FP6 subnormal: 0.mmm × 2^0 = m_i × 2^-3
        IF m_i == 0:
            e_o = m_o = 0  // zero
        ELSE IF m_i == 1:  // 2^-3 → FP8 normal e=4 (=7-3), m=0
            e_o = 4
            m_o = 0
        ELSE IF m_i <= 3:  // m_i×2^-3, leading 1 at bit 1 → 2^-2 range
            e_o = 5  // unbiased -2 → biased 5
            m_o = (m_i & 0x1) << 2  // remaining frac bit → FP8 frac[2]
        ELSE:  // m_i×2^-3, leading 1 at bit 2 → 2^-1 range
            e_o = 6  // unbiased -1 → biased 6
            m_o = (m_i & 0x3) << 1  // remaining 2 frac bits → FP8 frac[2:1]
    ELSE:  // FP6 normal: rebias 1→7, mantissa unchanged
        e_o = e_i + (7 - 1)  // rebias FP6 E2M3 (bias=1) → FP8 E4M3 (bias=7)
        m_o = m_i  // 3-bit frac fits directly (I-76)

    RETURN ((s_i & 0x1) << 7) | ((e_o & 0xF) << 3) | (m_o & 0x7)
```

16.5 AMX and ACE Functions

```
DEFINE any_nonzero(buf, lo, hi):
```

```
    // Return True if any byte in buf[lo:hi+1] is nonzero.
    FOR i = lo TO hi + 1 - 1:
        IF buf.byte[i] != 0:
            RETURN 1
    RETURN 0
```

```
DEFINE init_tilecfg():
    // Return a zeroed TileConfig (palette_id=0, all tiles unconfigured).
    temp = TileConfig()
    temp.palette_id = 0
    temp.start_row = 0
    FOR n = 0 TO 7:
        temp.t[n].rows = 0
        temp.t[n].colsb = 0
        temp.t[n].valid = 0
    RETURN temp
```

```
DEFINE process_tilecfg_palette_1(buf):
    // Validate and load palette 1 tile configuration from 64-byte buffer.
    // Returns a TileConfig on success; raises GP(0) on any invalid field.
    error = 0
    temp = TileConfig()
    temp.palette_id = 1
    temp.start_row = buf.byte[1]
    // bytes 2-15 must be zero
    IF any_nonzero(buf, 2, 15):
        error = 1
    // configure columns (bytes 16-31: 8 x 16-bit LE colsb values)
    FOR n = 0 TO palette_table[1].max_names-1:
        temp.t[n].colsb = buf.byte[16 + 2*n] | (buf.byte[17 + 2*n] << 8)
        IF temp.t[n].colsb > palette_table[1].bytes_per_row:
            error = 1
    // bytes after last colsb through 47 must be zero
    IF any_nonzero(buf, 16 + 2 * palette_table[1].max_names, 47):
        error = 1
    // configure rows (bytes 48-55: 8 x 8-bit row counts)
    FOR n = 0 TO palette_table[1].max_names-1:
        temp.t[n].rows = buf.byte[48 + n]
        IF temp.t[n].rows > palette_table[1].max_rows:
            error = 1
    // bytes after last row through 63 must be zero
    IF any_nonzero(buf, 48 + palette_table[1].max_names, 63):
        error = 1
    // validate each tile: both zero or both nonzero
    FOR n = 0 TO palette_table[1].max_names-1:
        IF temp.t[n].rows != 0 AND temp.t[n].colsb != 0:
            temp.t[n].valid = 1
        ELSE IF temp.t[n].rows == 0 AND temp.t[n].colsb == 0:
            temp.t[n].valid = 0
        ELSE:
            error = 1  // one of rows or colsb was 0 but not both
    IF error:
        #GP(0)
    RETURN temp
```

```
DEFINE process_tilecfg_palette_2(buf):
    // Validate and load palette 2 (ACE) tile configuration.
    // Palette 2 has fixed dimensions: 8 tiles, 16 rows, 64 colsb.
    // Only buf.byte[0] (palette_id) is consumed; bytes 1-63 must be zero.
    // Returns a TileConfig on success; raises GP(0) if reserved bytes nonzero.
    IF any_nonzero(buf, 1, 63):
```

```
        #GP(0)
    temp = TileConfig()
    temp.palette_id = 2
    temp.start_row = 0
    // Fixed configuration: all 8 tiles are 16 rows x 64 bytes (512 bits)
    // These assignments help common processing pseudocode that iterates over tile rows/
    columns;
    // they are not actual programmer state.
    FOR n = 0 TO 7:
        temp.t[n].rows = 16
        temp.t[n].colsb = 64
        temp.t[n].valid = 1
    RETURN temp
```

```
DEFINE zero_all_tile_data():
    // Zero all tile data registers.
    // Uses the XSAVE state component size, not per-tile dimensions.
    IF XCR0[TILEDATA]:
        size = CPUID_TILEDATA_SIZE  // CPUID(0DH, XTILEDATA).EAX
        FOR j = 0 TO size-1:
            tile_xsave_region.byte[j] = 0
```

```
DEFINE cpu_supports_palette(palette_id):
    // Return 1 if the CPU enumerates the given palette via CPUID.
    // Palette 0 (INIT): always supported.
    // Palette 1 (AMX): supported if TOTAL_TILE_BYTES for palette 1 is nonzero.
    // Palette 2 (ACE): supported if ACE_VERSION > 0.
    IF palette_id == 0:
        RETURN 1
    ELSE IF palette_id == 1:
        IF CPUID_1DH_01H_TOTAL_TILE_BYTES > 0:
            RETURN 1
    ELSE IF palette_id == 2:
        IF CPUID_1DH_02H_ACE_VERSION > 0:
            RETURN 1
    RETURN 0
```

```
DEFINE xcr0_supports_palette(palette_id):
    // Return 1 if the current XCR0 state supports the given palette ID.
    // Palette 0 (INIT): always supported.
    // Palette 1 (AMX TMUL): requires XCR0[TILECFG] and XCR0[TILEDATA].
    // Palette 2 (ACE): requires XCR0[TILECFG], XCR0[TILEDATA], and XCR0[SCALEDATA].
    IF palette_id == 0:
        RETURN 1
    ELSE IF palette_id == 1:
        IF XCR0[TILECFG] AND XCR0[TILEDATA]:
            RETURN PALETTE_1_SUPPORTED  // implementation-defined: 1 if AMX supported
    ELSE IF palette_id == 2:
        IF XCR0[TILECFG] AND XCR0[TILEDATA] AND XCR0[SCALEDATA]:
            RETURN PALETTE_2_SUPPORTED  // implementation-defined: 1 if ACE supported
    RETURN 0
```

```
DEFINE e8m0_to_fp32(x):
    // Convert E8M0 biased exponent byte to FP32 bit pattern.
    // The E8M0 byte is placed directly into the FP32 exponent field (bits [30:23]).
    // E8M0 NaN (0xFF) would be FP32 Inf without non-zero mantissa; returned as NaN.
    // Spec §12.5: f = x << 23
    IF x == 0xFF:  // E8M0 NaN → FP32 NaN
        RETURN QNaN_Indefinite
    RETURN fp32.from_bits(x << 23)  // E8M0 byte → FP32 exponent field directly
```

```
DEFINE convert_fp8_to_fixpoint64(fp8_byte, fp8_type):
    // Convert FP8 (E5M2/BF8 or E4M3/HF8) to 64-bit fixed-point integer.
    // The result is an aligned integer: BF8 result = 2^16 * float_value;
    // HF8 result = 2^9 * float_value.
    // This alignment enables exact int128 accumulation.
    IF fp8_type == "bf8":  // E5M2
        RETURN convert_bf8_to_fixpoint64(fp8_byte)
    ELSE:  // E4M3 = "hf8"
        RETURN convert_hf8_to_fixpoint64(fp8_byte)
```

```
DEFINE convert_bf8_to_fixpoint64(fp8_byte):
    // BF8 (E5M2) → 64-bit fixed-point integer. Result = 2^16 × float_value.
    sign = (fp8_byte & 0x80) >> 7
    exp  = (fp8_byte & 0x7C) >> 2
    frac = (fp8_byte & 0x03)
    mant = frac if exp == 0 else (frac | 0x4)  // set J-bit for normal numbers
    e_count = 0 if exp == 0 else exp - 1  // integer alignment shift
    magnitude = mant << e_count
    RETURN -magnitude if sign else magnitude  // result is 2^16 × in
```

```
DEFINE convert_hf8_to_fixpoint64(fp8_byte):
    // HF8 (E4M3) → 64-bit fixed-point integer. Result = 2^9 × float_value.
    sign = (fp8_byte & 0x80) >> 7
    exp  = (fp8_byte & 0x78) >> 3
    frac = (fp8_byte & 0x07)
    mant = frac if exp == 0 else (frac | 0x8)  // set J-bit for normal numbers
    e_count = 0 if exp == 0 else exp - 1  // integer alignment shift
    magnitude = mant << e_count
    RETURN -magnitude if sign else magnitude  // result is 2^9 × in
```

```
DEFINE convert_fixpoint128_scaled_to_fp32_ftz_rne(x, adjust):
    // Convert a fixpoint integer sum of products to FP32 (FTZ=1, RNE).

    // x:      Signed arbitrary precision integer — in this case assumed to be 128b
            // as that is sufficient for the sum of products using the operation.
    // adjust: Combined exponent adjustment, precomputed by the caller:
                // MXFP8 path:   -factor + scale_a + scale_b - 254
                // MXINT8 path:  -12     + scale_a + scale_b - 254
            // where -factor / -12 corrects the fixpoint encoding implicit bias and
            // scale_a + scale_b - 254 applies both E8M0 block scales as 2^(s-127).

    // Left-normalizes the magnitude so the J-bit arrives at bit 126, giving fixed
    // extraction positions (Lbit=103, Gbit=102, sticky=OR(101:0)).  Rounding overflow
    // (Ovf) is added to the exponent before the FTZ/overflow checks, collapsing the
    // subnormal boundary into the single test biased <= 0.
    IF x == 0:
        RETURN 0.0
    sign      = x[127]
    magnitude = (-x) if sign else x

    // Left-normalize: shift magnitude left until J-bit reaches bit 126.
    // Jbit_position tracks the original leading-bit index (decremented each shift).
    Jbit_position = 126
    WHILE NOT magnitude[126]:
        Jbit_position -= 1
        magnitude    <<= 1
    // J-bit is now at magnitude[126]; guard and sticky are at fixed positions.

    sticky = 1 if (magnitude & mask(102)) else 0  // OR(magnitude[101:0])
    Gbit   = magnitude[102]  // guard bit
    Lbit   = magnitude[103]  // LSB of 23-bit fraction field
```

```
    RndAdd = Gbit & (Lbit | sticky)  // RNE: round up if G=1 and (L|S)=1

    Mantissa    = magnitude >> 103  // 24 bits: J at [23], fraction at [22:0]
    RndMantissa = Mantissa + RndAdd
    Ovf         = RndMantissa[24]  // rounding carry into exponent

    // Fold Ovf into biased before range checks — the subnormal boundary case
    // (biased_pre_Ovf==0, Ovf==1 → minimum normal) falls naturally into biased>=1.
    biased = 127 + Jbit_position + adjust + Ovf

    IF biased > 254:  // overflow → ±Inf
        RETURN fp32.from_bits((sign << 31) | 0x7F800000)
    IF biased <= 0:  // subnormal or below → FTZ=1 → ±0
        RETURN fp32.from_bits(sign << 31)

    frac = RndMantissa & 0x7FFFFF
    RETURN fp32.from_bits((sign << 31) | (biased << 23) | frac)
```

Appendix A – CPUID Summary

Feature Flag	CPUID Leaf	Description
`AVX10`	`EAX=07H,ECX=01H : EDX[19]`	AVX10 Converged Vector ISA presence
`AVX10_VSN`	`EAX=24H,ECX=00H : EBX[7:0]`	AVX10 version (≥ 2 for AVX10.2)
`ACE`	`EAX=07H,ECX=01H : ECX[11]`	ACE presence
`ACE_VSN`	`EAX=1DH,ECX=02H : EAX[7:0]`	ACE version (≥ 1 for v1)
`Palette 2`	`EAX=1DH,ECX=02H`	ACE tile palette (fixed 512b x 16)
`AMX-TILE`	`EAX=07H,ECX=00H : EDX[24]`	AMX tile management
`AVX-VNNI-INT8`	`EAX=07H,ECX=01H : EDX[4]`	VNNI INT8 (VEX)
`AVX-VNNI-INT16`	`EAX=07H,ECX=01H : EDX[10]`	VNNI INT16 (VEX)
`AVX10_V1_AUX`	`EAX=24H,ECX=01H : ECX[2]`	AVX10.2 subset
`AVX10_V2_AUX`	`EAX=24H,ECX=01H : ECX[3]`	OCP format conversions

Appendix B – References

[1] OCP 8-bit Floating Point Specification (OFP8) Revision 1.0
https://www.opencompute.org/documents/ocp-8-bit-floating-point-specification-ofp8-revision-1-0-2023-06-20-pdf

[2] OCP Microscaling Formats (MX) Specification 1.0, September 2023
https://www.opencompute.org/documents/ocp-microscaling-formats-mx-v1-0-spec-final-pdf

[3] Intel Architecture Instruction Set Extensions and Future Features
https://cdrdv2-public.intel.com/782879/architecture-instruction-set-extensions-programming-reference.pdf

[4] Intel 64 and IA-32 Architectures Software Developer’s Manual
https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html

[5] Intel Advanced Vector Extensions 10.2 Architecture Specification
https://cdrdv2-public.intel.com/828965/361050-intel-avx10.2-spec.pdf

[6] AMD Advanced Performance Extensions (APX) Architecture Specification

Appendix C — APX Interaction

This specification is designed to be implemented on processors that also implement Advanced Performance Extensions (APX) [6]. APX extends the EVEX prefix to address 32 general-purpose registers (r0–r31) via additional register index bits: B4, X4, and V4. The following table summarizes how APX extension bits interact with each exception class defined in this specification.

Exception class	B4 (base)	X4 (index)	V4 (VVVV)	Effect
E2, E4, E4NF, E6	Extends to r0–r31	Extends to r0–r31	Extends vector reg	Memory base/index can use EGPRs
E7NM	N/A	N/A	Extends vector reg	No change (register-only)
AMX-E1, AMX-E2	N/A (VEX)	N/A (VEX)	N/A (VEX)	No effect (VEX-encoded)
AMX-E1-EVEX, AMX-E2-EVEX	Extends to r0–r31	Extends to r0–r31	N/A	EVEX-promoted: memory base/index can use EGPRs
AMX-E5, AMX-E6	N/A (VEX)	N/A (VEX)	N/A (VEX)	No effect (VEX-encoded)
ACE-E4	Ignored	Ignored	Ignored	No effect (tile + ZMM only)
ACE-E1	Ignored	Ignored	Ignored	No effect (VVVV unused, register-only)
ACE-E2, ACE-E3	Extends to r0–r31	Extends to r0–r31	Ignored	Memory base/index can use EGPRs
ACE-E5	N/A	N/A	N/A (VEX)	No effect
ACE-E6	Ignored	Ignored	Extends GPR to r0–r31	Row/column specifier can use EGPRs

Table 4: APX extended register interaction by exception class. “Ignored” means the bit has no architectural effect.

When APX is not enabled (XCR0[APX_F] = 0): EVEX.B4 must be 0 and EVEX.X4 must be 1 (inverted encoding), or #UD results. EVEX.V4 (V’) constraints are per exception class (see individual definitions in Section 5).

When APX is enabled (XCR0[APX_F] = 1): unused extension bits are ignored. Code generators should set unused bits to their logical-0 values.