ACE v2 / AVX10.3 Architecture Specification



x86 Ecosystem Advisory Group

Specification Release

Date:	10/22/2026
Version:	0.9.2.4

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 5
2 Introduction 6
2.1 Scope 6
2.2 Feature Overview and CPUID Enumeration 6
3 AVX10_CVT_BF16 — BF16 ↔ FP8 and BF16 → UF8 Converts 9
3.1 BF16 ↔ FP8 Converts 9
3.2 BF16 to UF8 Converts 14
3.3 Instruction Encodings 15
4 AVX10_INT4_FP4 — FP4 / INT4 Converts 18
4.1 Instructions 18
4.2 Instruction Encodings 25
5 AVX10_VPINTERLEAVE — Interleave Byte / Word 27
5.1 Instructions 27
5.2 Instruction Encodings 28
6 AVX10_BMM — Bit Matrix Multiply 30
6.1 Overview 30
6.2 Instruction Encodings 32
7 Tile Management Instructions 33
7.1 LDTILECFG — Load Tile Configuration 33
7.2 STTILECFG — Store Tile Configuration 33
7.3 Instruction Encodings 33
8 ACE v2 Extensions 34
8.1 Instruction Groups 34
8.2 Reduced-Precision AI Outer Products 34
8.3 FP16 Outer Products 37
8.4 Block Scale Format Enhancements 39
8.5 Instruction Encodings 39
9 ACE v2 Single and Double Precision Outer Products 40
9.1 CPUID Enumeration 40
9.2 TOP1PS — FP32 Rank-1 Outer Product, FP32 Accumulate 40
9.3 TOP1PD — FP64 Rank-1 Outer Product, Full Tile 40
9.4 MXCSR Interaction 41
9.5 Instruction Encodings 42
10 Exception Classes 43
10.1 Exception Classes by Instruction 43

10.2 Exceptions Type E2 44
10.3 Exceptions Type E4 and E4NF of EVEX-Encoded Instructions 45
10.4 Exceptions Type E6 47
10.5 Exceptions Type AMX 48
10.6 Exceptions Type ACE 50
10.7 #UD Equations for EVEX encoded instructions 53
10.8 Implementation Specific Alignment Checking for 16, 32 and 64 Byte Accesses 54
Appendix A — Data Formats 55
Floating-Point Element Formats 55
Integer and Fixed-Point Element Formats 55
OCP MX Block Scale Formats 55
Mnemonic Naming Conventions 57
Rounding Modes 57
Appendix B — Pseudocode Helper Functions 58
FP4 and INT4 Element Conversion 58
FP8 ↔ BF16 Element Helpers 59
FP16 Arithmetic Helpers 62
FP4 E2M1 Fixed-Point Helper 63
MX Scale Conversion Helpers 63
MX Integer Implicit Bias Helper 65
FP32 / FP64 Multiply-Accumulate Helpers 66
FP32 Accumulation Helper 66
MX Outer Product Subtile Helpers 67
Appendix C — APX Interaction 68
Appendix D — References 69

1 Revision History

Revision	Date	Notes
0.9.2.4	October 2026	Initial public release.

Table 1: Document revision history.

2 Introduction

2.1 Scope

This document specifies two related extensions to the x86-64 architecture.

The ACE v1 extensions introduced an outer product architecture with native support for modern AI datatypes. ACE and AVX10 provide complementary matrix and vector compute capabilities. With ACE v2, this capability is further extended with a focus on:

- Reduced-precision AI: New support for 4-bit formats and expanded 8-bit format capabilities
- Half-precision floating point: FP16
- Single and double precision floating point: FP32 and FP64

The AVX10.3 extensions complement this focus, and provide further enhancements to general purpose vectorization.

This document has the following structure:

- Overview of the extensions
- AVX10.3 features
- ACE v2 features
- Exception classes
- Appendices

2.2 Feature Overview and CPUID Enumeration

AVX10.3 and ACE v2 features are enumerated using a combination of major version and per-group sub-feature bits.

2.2.1 Major Version Fields

Feature	CPUID leaf	Bit / field	Description
AVX10	EAX:07H, ECX:1	EDX[19]	AVX10 converged vector ISA support.
ACE	EAX:07H, ECX:1	ECX[11]	ACE tile architecture support.
AVX10_VSN ≥ 3	EAX:24H, ECX:0	EBX[7:0] ≥ 3	Gate for all baseline AVX10.3 features. Requires AVX10.
ACE_VSN ≥ 2	EAX:1DH, ECX:2	EAX[7:0] ≥ 2	Gate for all baseline ACE v2 features. Requires ACE.

Table 2: Major version CPUID fields for AVX10.3 and ACE v2.

2.2.2 AVX10.3 Instruction Groups

The AVX10.3 specification defines baseline and optional extensions:

- **Baseline:** Data type conversion, interleave, and packing instructions enumerated by `AVX10_VSN` ≥ 3. This includes instructions previously enumerated under `AVX10_V2_AUX` and new AVX10.3 additions.
- **Optional:** Bit Matrix Multiply extensions, enumerated separately via `AVX10_BMM`.

Sub-feature bits at CPUID leaf 24H, sub-leaf 1 (ECX=1):

Bit	Name	Description
ECX[1]	AVX10_BMM	Bit matrix multiply: VBMACXOR16x16x16, VBMACOR16x16x16, VBITREV.
ECX[2]	AVX10_V1_AUX	AVX10.2 subset.[1]
ECX[3]	AVX10_V2_AUX	FP32↔FP8, FP4, FP6 converts; VUNPACKB; VPMOVSSDB. Also implied by AVX10_VSN ≥ 3.[2]

Table 3: AVX10 sub-feature CPUID bits (leaf 24H, sub-leaf 1).

[1] All AVX10.2+ implementations (AVX10_VSN ≥ 2) will enumerate AVX10_V1_AUX, including those that also enumerate AVX10.2 or later.

[2] All ACE implementations (ACE $= 1$) will enumerate AVX10_V2_AUX, including those that also enumerate AVX10.3.

2.2.2.1 Instructions defined under AVX10_V2_AUX

These AVX10 instructions were introduced alongside the ACE v1 extensions. All instructions enumerated under `AVX10_V2_AUX` are available on any AVX10.3 implementation.

Group	Mnemonics
FP32 → FP8 (RNE)	VCVTPS2BF8[S], VCVTPS2HF8[S]
FP32 → FP8 (RTO)	VCVTROPS2HF8[S]
FP32 → FP8 (Bias)	VCVTBIASPS2BF8[S], VCVTBIASPS2HF8[S]
FP8 → FP32	VCVTBF82PS, VCVTHF82PS
FP8 → FP4 E2M1	VCVTBF82BF4S, VCVTHF82BF4S
FP4 → FP8 E4M3	VCVTBF42HF8
FP8 → FP6 E3M2	VCVTBF82BF6S
FP8 → FP6 E2M3	VCVTHF82HF6S
FP6 → FP8 E4M3	VCVTBF62HF8, VCVTHF62HF8
Sub-byte unpack	VUNPACKB
Integer convert	VPMOVSSDB

Refer to the ACE v1 Specification [1] for full pseudocode, operand encodings, exception classes, and operation descriptions of these instructions.

2.2.2.2 New AVX10.3 instructions

New AVX10.3 baseline features are enumerated with AVX10_VSN ≥ 3.

Group	Mnemonics
FP8 → BF16 (up-convert)	VCVTBF82BF16, VCVTHF82BF16
BF16 → FP8 (single source)	VCVTBF162BF8[S], VCVTBF162HF8[S]
BF16 → FP8 (two sources)	VCVT2BF162BF8[S], VCVT2BF162HF8[S]
BF16 → FP8 with bias	VCVTBIASBF162BF8[S], VCVTBIASBF162HF8[S]
BF16 → UF8	VCVTBF162UF8[S], VCVT2BF162UF8[S]
FP16 / BF16 → FP4	VCVTPH2BF4S, VCVTBF162BF4S, VCVT2PH2BF4S, VCVT2BF162BF4S
FP16 / BF16 → FP4 with bias	VCVTBIASPH2BF4S, VCVTBIASBF162BF4S
FP16 / BF16 → INT4	VCVTPH2SNS, VCVTPH2UNS, VCVTBF162SNS, VCVTBF162UNS, VCVT2PH2SNS, VCVT2PH2UNS, VCVT2BF162SNS, VCVT2BF162UNS
Byte / word interleave	VPINTERLEAVELB, VPINTERLEAVEHB, VPINTERLEAVELW, VPINTERLEAVEHW

Table 4: New AVX10.3 baseline features.

Optional AVX10.3 features are separately enumerated.

Group	Mnemonics
Bit matrix multiply (AVX10_BMM)	VBMACXOR16x16x16, VBMACOR16x16x16, VBITREV

Table 5: New AVX10.3 optional features.

2.2.3 ACE v2 Instruction Groups

ACE v2 defines baseline and optional outer product extensions. All fields are in the Tile Palette 2 sub-leaf (EAX:1DH, ECX:2).

CPUID	Group	Mnemonics
ACE_VSN(EAX[7:0]) ≥ 2	MX FP4 outer product	TOP8FMXBF4PS
	MX INT4 outer products	TOP8FMXN[SS/SU/US/UU]PS
	MX INT8 extended variants	TOP4MXB[SU/US/UU]PS
	E5M3 block scale format	(format extension)

Table 6: ACE v2 baseline features (ACE_VSN ≥ 2).

CPUID	Group	Mnemonics
ACE_OP_FP16(EAX[16])	FP16 outer product	TOP2PHPS
ACE_OP_FP32(EAX[17])	FP32 outer product	TOP1PS
ACE_OP_FP64(EAX[18])	FP64 outer product	TOP1PD

Table 7: ACE v2 optional features.

Note: TOP2PHPS is a candidate for incorporation into the ACE v2 baseline.

2.2.3.1 Tile Management Instructions

Tile management instructions (LDTILECFG, STTILECFG) are inherited from ACE v1. This specification documents the APX-promoted EVEX forms that extend memory addressing to r16–r31; see Section 7.1 and Section 7.2.

2.2.4 Detection Algorithm

```
// ── Step 1: Check AVX10 and ACE primary enumerations ──────────
CPUID(EAX=07H, ECX=1) → ECX, EDX
AVX10                = EDX[19]
ACE                  = ECX[11]

IF NOT AVX10: AVX10 not supported → skip AVX10 checks
IF NOT ACE: ACE not supported → skip ACE checks

// ── Step 2: Check AVX10.3 version gate ────────────────────────
CPUID(EAX=24H, ECX=0) → EBX
AVX10_1              = EBX[7:0] >= 1
AVX10_2              = EBX[7:0] >= 2
AVX10_3              = EBX[7:0] >= 3   // All baseline AVX10.3 instructions available

// ── Step 3: Check AVX10 sub-feature bits (leaf 24H, sub-leaf 1) ─
CPUID(EAX=24H, ECX=1) → ECX
AVX10_BMM            = AVX     AND ECX[1]    // bit matrix multiply (VBMACXOR, VBMACOR,
VBITREV)
AVX10_V1_AUX         = AVX10_2 OR  ECX[2]    // AVX10.2 subset
AVX10_V2_AUX         = AVX10_3 OR  ECX[3]    // FP8↔FP32, FP4, FP6

// ── Step 4: Check ACE version and optional feature bits ──────────
// All fields are in the Tile Palette 2 sub-leaf (EAX:1DH, ECX:2).
CPUID(EAX=1DH, ECX=2) → EAX
ACE_v1               = EAX[7:0] >= 1
ACE_v2               = EAX[7:0] >= 2 // TOP8FMXBF4PS, TOP8FMXN*PS, TOP4MXB*PS

// ── Step 5: Check ACE optional outer products ─────────────────────
// ACE_OP_FP16, ACE_OP_FP32, ACE_OP_FP64 are independently enumerated.
// Each requires ACE (CPUID.07H.1:ECX[11], checked in Step 1).
// Software guidance: any CPU with ACE_OP_FP16/FP32/FP64 will have ACE_VSN >= 2.

// NOTE: TOP2PHPS is a candidate for inclusion in ACE v2 baseline
// If adopted into baseline, TOP2PHPS will be enumerated off of just ACE_v2.
ACE_OP_FP16          = ACE AND EAX[16]   // TOP2PHPS (FP16 rank-2 outer product)

ACE_OP_FP32          = ACE AND EAX[17]   // TOP1PS (FP32 outer product)
ACE_OP_FP64          = ACE AND EAX[18]   // TOP1PD (FP64 outer product)
```

3 AVX10_CVT_BF16 — BF16 ↔ FP8 and BF16 → UF8 Converts

These instructions convert between BF16 and FP8 (E5M2 or E4M3). They complement the converts defined in AVX10.2 and AVX10_V2_AUX [2] that introduced FP8 ↔ FP16 converts.

Group	Mnemonics
FP8 → BF16 (up-convert)	VCVTBF82BF16, VCVTHF82BF16
BF16 → FP8 (single source)	VCVTBF162BF8, VCVTBF162BF8S, VCVTBF162HF8, VCVTBF162HF8S
BF16 → FP8 (two sources)	VCVT2BF162BF8, VCVT2BF162BF8S, VCVT2BF162HF8, VCVT2BF162HF8S
BF16 → FP8 with bias	VCVTBIASBF162BF8, VCVTBIASBF162BF8S, VCVTBIASBF162HF8, VCVTBIASBF162HF8S
BF16 → UF8 (single source)	VCVTBF162UF8, VCVTBF162UF8S
BF16 → UF8 (two sources)	VCVT2BF162UF8, VCVT2BF162UF8S

3.1 BF16 ↔ FP8 Converts

3.1.1 Up-Convert: FP8 → BF16

Convert packed FP8 elements to BF16. Exact conversion; no rounding is required. Both FP8 source formats are supported.

Destination zero/mask at BF16 (output word) granularity is supported.

Variants

Mnemonic	Source format
VCVTBF82BF16	E5M2 (BF8)
VCVTHF82BF16	E4M3 (HF8)

Operand forms

The destination is twice the width of the source; each source element widens to the destination element size.

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

Instruction Operand Encoding

Instruction	Operand 1	Operand 2	Operand 3
VCVTBF82BF16 VCVTHF82BF16	MODRM.REG(w)	MODRM.R/M(r)	N/A

Operation

```
DEFINE vcvt_fp8_to_bf16(src, VL, k1, zeroing, src_is_mem, evex_b, fp8_type="bf8",
no_writemask=False):
    ASSERT VL in (128, 256, 512)
    ASSERT fp8_type in ("bf8", "hf8")
    KL = VL / 16
    orig_dest = copy(dest)
    FOR i = 0 TO KL-1:
        t = src.byte[0] if (src_is_mem AND evex_b) else src.byte[i]
        IF fp8_type == "bf8":
            val = fp8_e5m2_to_bf16(t)
        ELSE:
            val = fp8_e4m3_to_bf16(t)
        IF k1[i] OR no_writemask:
            dest.bf16[i] = val
        ELSE IF zeroing:
            dest.bf16[i] = 0
```

```
        ELSE:
            dest.bf16[i] = orig_dest.bf16[i]
    dest[MAXVL-1 : VL] = 0
```

Other Exceptions

Exception class E2. See Section 10.2.

3.1.2 Down-Convert: BF16 → FP8 (single source)

Convert packed BF16 elements to FP8. Both FP8 source formats are supported. Saturated variants clamp to format maximum; non-saturated variants overflow to Inf/NaN. Rounding mode: RNE.

Destination zero/mask at FP8 (output byte) granularity is supported. Destination upper bits zeroed.

Variants

Mnemonic	Destination format	Saturation
VCVTBF162BF8	E5M2 (BF8)	no
VCVTBF162BF8S	E5M2 (BF8)	yes
VCVTBF162HF8	E4M3 (HF8)	no
VCVTBF162HF8S	E4M3 (HF8)	yes

Operand forms

The operation is narrowing; the destination is half the width of the source.

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

Instruction Operand Encoding

Instruction	Operand 1	Operand 2	Operand 3
VCVTBF162BF8 VCVTBF162BF8S VCVTBF162HF8 VCVTBF162HF8S	MODRM.REG(w)	MODRM.R/M(r)	N/A

Operation

```
DEFINE vcvtbf162fp8(src, VL, k1, zeroing, src_is_mem, evex_b, fp8_type="bf8", saturate=False,
no_writemask=False):
    ASSERT VL in (128, 256, 512)
    ASSERT fp8_type in ("bf8", "hf8")

    KL = VL / 16
    orig_dest = copy(dest)
    FOR i = 0 TO KL-1:
        t = src.bf16[0] if (src_is_mem AND evex_b) else src.bf16[i]
        val = bf16_to_fp8_e5m2(t, rounding="RNE", saturate=saturate) if fp8_type == "bf8"
        else bf16_to_fp8_e4m3(t, rounding="RNE", saturate=saturate)
        IF k1[i] OR no_writemask:
            dest.byte[i] = val
        ELSE IF zeroing:
            dest.byte[i] = 0
        ELSE:
            dest.byte[i] = orig_dest.byte[i]
    dest[MAXVL-1 : VL / 2] = 0
```

Other Exceptions

Exception class E4NF. See Section 10.3.

3.1.3 Down-Convert: BF16 → FP8 (two sources)

Convert two packed BF16 source registers into a single FP8 destination. Both FP8 source formats are supported. Saturated variants clamp to format maximum; non-saturated variants overflow to Inf/NaN. Rounding mode: RNE.

Destination zero/mask at FP8 (output byte) granularity is supported.

Variants

Mnemonic	Destination format	Saturation
VCVT2BF162BF8	E5M2 (BF8)	no
VCVT2BF162BF8S	E5M2 (BF8)	yes
VCVT2BF162HF8	E4M3 (HF8)	no
VCVT2BF162HF8S	E4M3 (HF8)	yes

Operand forms

Source and destination share the same vector width; two source operands are consumed. Lower KL/2 elements come from src2; upper KL/2 come from src1.

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 Encoding

Instruction	Operand 1	Operand 2	Operand 3
VCVT2BF162BF8 VCVT2BF162BF8S VCVT2BF162HF8 VCVT2BF162HF8S	MODRM.REG(w)	VVVV(r)	MODRM.R/M(r)

Operation

```
DEFINE vcvt2bf162fp8(src1, src2, VL, k1, zeroing, src_is_mem, evex_b, fp8_type="bf8",
saturate=False, no_writemask=False):
    ASSERT VL in (128, 256, 512)
    ASSERT fp8_type in ("bf8", "hf8")

    KL = VL / 8
    orig_dest = copy(dest)
    FOR i = 0 TO KL-1:
        IF i < KL / 2:
            t = src2.bf16[0] if (src_is_mem AND evex_b) else src2.bf16[i]
        ELSE:
            t = src1.bf16[i - KL / 2]

        val = bf16_to_fp8_e5m2(t, saturate=saturate) if fp8_type == "bf8" else
        bf16_to_fp8_e4m3(t, saturate=saturate)

        IF k1[i] OR no_writemask:
            dest.byte[i] = val
        ELSE IF zeroing:
            dest.byte[i] = 0
        ELSE:
            dest.byte[i] = orig_dest.byte[i]

    dest[MAXVL-1 : VL] = 0
```

Other Exceptions

Exception class E4NF. See Section 10.3.

3.1.4 Down-Convert with Bias: BF16 → FP8

Convert BF16 to FP8 with biased rounding. Both FP8 source formats are supported. Saturated variants clamp to format maximum; non-saturated variants overflow to Inf/NaN.

The first source provides a bias value which is added to the mantissa of source BF16 before truncation. The instruction is agnostic to how the bias is generated; supplying pseudo-random bias achieves stochastic rounding.

Destination zero/mask at FP8 (output byte) granularity is supported. Destination upper bits zeroed.

Variants

Mnemonic	Destination format	Saturation
VCVTBIASBF162BF8	E5M2 (BF8)	no
VCVTBIASBF162BF8S	E5M2 (BF8)	yes
VCVTBIASBF162HF8	E4M3 (HF8)	no
VCVTBIASBF162HF8S	E4M3 (HF8)	yes

Operand forms

The operation is narrowing; the destination is half the width of the data source.

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 Encoding

Instruction	Operand 1	Operand 2	Operand 3
VCVTBIASBF162BF8 VCVTBIASBF162BF8S VCVTBIASBF162HF8 VCVTBIASBF162HF8S	MODRM.REG(w)	VVVV(r)	MODRM.R/M(r)

Operation

```
DEFINE vcvtbiasbf162fp8(src1, src2, VL, k1, zeroing, fp8_type="bf8", saturate=False,
no_writemask=False, src2_is_mem=False, evex_b=False):
    ASSERT VL in (128, 256, 512)
    ASSERT fp8_type in ("bf8", "hf8")
    KL = VL / 16
    orig_dest = copy(dest)
    FOR i = 0 TO KL-1:
        bias = src1.byte[2 * i]
        t    = src2.bf16[0] if (src2_is_mem AND evex_b) else src2.bf16[i]
        val  = (bf16_to_fp8_e5m2_biased(bias, t, saturate=saturate)
                if fp8_type == "bf8"
                else bf16_to_fp8_e4m3_biased(bias, t, saturate=saturate))
        IF k1[i] OR no_writemask:
            dest.byte[i] = val
        ELSE IF zeroing:
            dest.byte[i] = 0
        ELSE:
            dest.byte[i] = orig_dest.byte[i]
    dest[MAXVL-1 : VL / 2] = 0
```

Other Exceptions

Exception class E4NF. See Section 10.3.

3.2 BF16 to UF8 Converts

Convert packed BF16 elements to UF8 (unsigned E5M3) bytes. Saturated variants clamp to format maximum; non-saturated variants overflow to ±Inf/NaN. Rounding mode: RNE.

3.2.1 Down-Convert: BF16 → UF8 (single source)

The operation is narrowing; the destination is half the width of the source. Destination zero/mask at UF8 (output byte) granularity is supported. Destination upper bits zeroed.

Variants

Mnemonic	Saturation
VCVTBF162UF8	no
VCVTBF162UF8S	yes

Operand forms

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

Instruction Operand Encoding

Instruction	Operand 1	Operand 2	Operand 3
VCVTBF162UF8	MODRM.REG(w)	MODRM.R/M(r)	N/A

Operation

```
DEFINE vcvtbf162uf8(src, VL, k1, zeroing, src_is_mem, evex_b, no_writemask=False):
    // VCVTBF162UF8: BF16 → UF8, non-saturating, narrowing 2:1.
    ASSERT VL in (128, 256, 512)
    KL = VL / 16
    orig_dest = copy(dest)
    FOR i = 0 TO KL-1:
        t = src.bf16[0] if (src_is_mem AND evex_b) else src.bf16[i]
        val = bf16_to_e5m3(t, saturate=0)
        IF k1[i] OR no_writemask:
            dest.byte[i] = val
        ELSE IF zeroing:
            dest.byte[i] = 0
        ELSE:
            dest.byte[i] = orig_dest.byte[i]
    dest[MAXVL-1 : VL / 2] = 0
```

Other Exceptions

Exception class E4NF. See Section 10.3.

3.2.2 Down-Convert: BF16 → UF8 (two sources)

Source and destination share the same vector width; two source operands are consumed. Lower KL/2 elements come from src2; upper KL/2 come from src1. Destination zero/mask at UF8 (output byte) granularity is supported.

Variants

Mnemonic	Saturation
VCVT2BF162UF8	no
VCVT2BF162UF8S	yes

Operand forms

Operand 1	Operand 2	Operand 3
xmm1{k1}{z}	xmm2	xmm3/m128/m16bcst

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

Instruction Operand Encoding

Instruction	Operand 1	Operand 2	Operand 3
VCVT2BF162UF8	MODRM.REG(w)	VVVV(r)	MODRM.R/M(r)

Operation

```
DEFINE vcvt2bf162uf8(src1, src2, VL, k1, zeroing, src2_is_mem, evex_b, no_writemask=False):
    // VCVT2BF162UF8: two BF16 sources → UF8, non-saturating, full-width output.
    ASSERT VL in (128, 256, 512)
    KL = VL / 16
    HALF = KL / 2
    orig_dest = copy(dest)
    FOR i = 0 TO KL-1:
        IF i < HALF:
            t = src2.bf16[0] if (src2_is_mem AND evex_b) else src2.bf16[i]
        ELSE:
            t = src1.bf16[i - HALF]
        val = bf16_to_e5m3(t, saturate=0)
        IF k1[i] OR no_writemask:
            dest.byte[i] = val
        ELSE IF zeroing:
            dest.byte[i] = 0
        ELSE:
            dest.byte[i] = orig_dest.byte[i]
    dest[MAXVL-1 : VL / 2] = 0
```

Other Exceptions

Exception class E4NF. See Section 10.3.

3.3 Instruction Encodings

3.3.1 FP8 to BF16 Converts

Mnemonic	Form	Encoding
VCVTBF82BF16	xmm1{k1}{z}, xmm2/m64	EVEX.128.F2.MAP5.W1 1F
VCVTBF82BF16	ymm1{k1}{z}, xmm2/m128	EVEX.256.F2.MAP5.W1 1F
VCVTBF82BF16	zmm1{k1}{z}, ymm2/m256	EVEX.512.F2.MAP5.W1 1F
VCVTHF82BF16	xmm1{k1}{z}, xmm2/m64	EVEX.128.F2.MAP5.W0 1F
VCVTHF82BF16	ymm1{k1}{z}, xmm2/m128	EVEX.256.F2.MAP5.W0 1F
VCVTHF82BF16	zmm1{k1}{z}, ymm2/m256	EVEX.512.F2.MAP5.W0 1F

3.3.2 BF16 to FP8 Converts (Single-Source)

Mnemonic	Form	Encoding
VCVTBF162BF8	xmm1{k1}{z}, xmm2/m128/m16bcst	EVEX.128.F3.0F38.W1 74
VCVTBF162BF8	xmm1{k1}{z}, ymm2/m256/m16bcst	EVEX.256.F3.0F38.W1 74
VCVTBF162BF8	ymm1{k1}{z}, zmm2/m512/m16bcst	EVEX.512.F3.0F38.W1 74
VCVTBF162BF8S	xmm1{k1}{z}, xmm2/m128/m16bcst	EVEX.128.F3.MAP5.W1 74
VCVTBF162BF8S	xmm1{k1}{z}, ymm2/m256/m16bcst	EVEX.256.F3.MAP5.W1 74
VCVTBF162BF8S	ymm1{k1}{z}, zmm2/m512/m16bcst	EVEX.512.F3.MAP5.W1 74
VCVTBF162HF8	xmm1{k1}{z}, xmm2/m128/m16bcst	EVEX.128.F3.MAP5.W1 18
VCVTBF162HF8	xmm1{k1}{z}, ymm2/m256/m16bcst	EVEX.256.F3.MAP5.W1 18
VCVTBF162HF8	ymm1{k1}{z}, zmm2/m512/m16bcst	EVEX.512.F3.MAP5.W1 18

Mnemonic	Form	Encoding
VCVTBF162HF8S	xmm1{k1}{z}, xmm2/m128/m16bcst	EVEX.128.F3.MAP5.W1 1B
VCVTBF162HF8S	xmm1{k1}{z}, ymm2/m256/m16bcst	EVEX.256.F3.MAP5.W1 1B
VCVTBF162HF8S	ymm1{k1}{z}, zmm2/m512/m16bcst	EVEX.512.F3.MAP5.W1 1B

3.3.3 BF16 to FP8 Converts (Two-Source)

Mnemonic	Form	Encoding
VCVT2BF162BF8	xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	EVEX.128.F2.0F38.W1 74
VCVT2BF162BF8	ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst	EVEX.256.F2.0F38.W1 74
VCVT2BF162BF8	zmm1{k1}{z}, zmm2, zmm3/m512/m16bcst	EVEX.512.F2.0F38.W1 74
VCVT2BF162BF8S	xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	EVEX.128.F2.MAP5.W1 74
VCVT2BF162BF8S	ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst	EVEX.256.F2.MAP5.W1 74
VCVT2BF162BF8S	zmm1{k1}{z}, zmm2, zmm3/m512/m16bcst	EVEX.512.F2.MAP5.W1 74
VCVT2BF162HF8	xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	EVEX.128.F2.MAP5.W1 18
VCVT2BF162HF8	ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst	EVEX.256.F2.MAP5.W1 18
VCVT2BF162HF8	zmm1{k1}{z}, zmm2, zmm3/m512/m16bcst	EVEX.512.F2.MAP5.W1 18
VCVT2BF162HF8S	xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	EVEX.128.F2.MAP5.W1 1B
VCVT2BF162HF8S	ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst	EVEX.256.F2.MAP5.W1 1B
VCVT2BF162HF8S	zmm1{k1}{z}, zmm2, zmm3/m512/m16bcst	EVEX.512.F2.MAP5.W1 1B

3.3.4 BF16 to FP8 Converts (Biased)

Mnemonic	Form	Encoding
VCVTBIASBF162BF8	xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	EVEX.128.NP.0F38.W1 74
VCVTBIASBF162BF8	xmm1{k1}{z}, ymm2, ymm3/m256/m16bcst	EVEX.256.NP.0F38.W1 74
VCVTBIASBF162BF8	ymm1{k1}{z}, zmm2, zmm3/m512/m16bcst	EVEX.512.NP.0F38.W1 74
VCVTBIASBF162BF8S	xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	EVEX.128.NP.MAP5.W1 74
VCVTBIASBF162BF8S	xmm1{k1}{z}, ymm2, ymm3/m256/m16bcst	EVEX.256.NP.MAP5.W1 74
VCVTBIASBF162BF8S	ymm1{k1}{z}, zmm2, zmm3/m512/m16bcst	EVEX.512.NP.MAP5.W1 74
VCVTBIASBF162HF8	xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	EVEX.128.NP.MAP5.W1 18
VCVTBIASBF162HF8	xmm1{k1}{z}, ymm2, ymm3/m256/m16bcst	EVEX.256.NP.MAP5.W1 18
VCVTBIASBF162HF8	ymm1{k1}{z}, zmm2, zmm3/m512/m16bcst	EVEX.512.NP.MAP5.W1 18
VCVTBIASBF162HF8S	xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	EVEX.128.NP.MAP5.W1 1B
VCVTBIASBF162HF8S	xmm1{k1}{z}, ymm2, ymm3/m256/m16bcst	EVEX.256.NP.MAP5.W1 1B
VCVTBIASBF162HF8S	ymm1{k1}{z}, zmm2, zmm3/m512/m16bcst	EVEX.512.NP.MAP5.W1 1B

3.3.5 BF16 to UF8 Converts

Mnemonic	Form	Encoding
VCVTBF162UF8	xmm1{k1}{z}, xmm2/m128/m16bcst	EVEX.128.F3.MAP5.W1 16
VCVTBF162UF8	xmm1{k1}{z}, ymm2/m256/m16bcst	EVEX.256.F3.MAP5.W1 16
VCVTBF162UF8	ymm1{k1}{z}, zmm2/m512/m16bcst	EVEX.512.F3.MAP5.W1 16
VCVTBF162UF8S	xmm1{k1}{z}, xmm2/m128/m16bcst	EVEX.128.F3.MAP5.W1 17
VCVTBF162UF8S	xmm1{k1}{z}, ymm2/m256/m16bcst	EVEX.256.F3.MAP5.W1 17
VCVTBF162UF8S	ymm1{k1}{z}, zmm2/m512/m16bcst	EVEX.512.F3.MAP5.W1 17

3.3.6 BF16 to UF8 Converts (Two-Source)

Mnemonic	Form	Encoding
VCVT2BF162UF8	xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	EVEX.128.F2.MAP5.W1 16
VCVT2BF162UF8	ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst	EVEX.256.F2.MAP5.W1 16
VCVT2BF162UF8	zmm1{k1}{z}, zmm2, zmm3/m512/m16bcst	EVEX.512.F2.MAP5.W1 16
VCVT2BF162UF8S	xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	EVEX.128.F2.MAP5.W1 17

Mnemonic	Form	Encoding
VCVT2BF162UF8S	ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst	EVEX.256.F2.MAP5.W1 17
VCVT2BF162UF8S	zmm1{k1}{z}, zmm2, zmm3/m512/m16bcst	EVEX.512.F2.MAP5.W1 17

4 AVX10_INT4_FP4 — FP4 / INT4 Converts

These instructions convert between 16-bit floating-point formats (FP16 or BF16) and 4-bit formats (FP4 E2M1, signed INT4, or unsigned INT4) [2]. Output is nibble-packed, two per output byte.

Write-masking is not supported for these instructions.

Group	Mnemonics
FP16 → FP4 E2M1 (single source)	VCVTPH2BF4S
FP16 → INT4 (single source)	VCVTPH2SNS, VCVTPH2UNS
FP16 → FP4 E2M1 (two sources)	VCVT2PH2BF4S
FP16 → INT4 (two sources)	VCVT2PH2SNS, VCVT2PH2UNS
FP16 → FP4 with bias	VCVTBIASPH2BF4S
BF16 → FP4 E2M1 (single source)	VCVTBF162BF4S
BF16 → INT4 (single source)	VCVTBF162SNS, VCVTBF162UNS
BF16 → FP4 E2M1 (two sources)	VCVT2BF162BF4S
BF16 → INT4 (two sources)	VCVT2BF162SNS, VCVT2BF162UNS
BF16 → FP4 with bias	VCVTBIASBF162BF4S

4.1 Instructions

4.1.1 FP16 → FP4 E2M1 (single source)

Convert packed FP16 source register to FP4 destination. Results always saturate to format maximum. Rounding mode: RNE.

Output nibble-packed (2 per byte). Result masking is not supported. Destination upper bits zeroed.

Variants

Mnemonic	Output format
VCVTPH2BF4S	FP4 E2M1

Operand forms

The operation is narrowing; the destination is a quarter the width of the source.

Operand 1	Operand 2
xmm1	xmm2/m128/m16bcst
xmm1	ymm2/m256/m16bcst
xmm1	zmm2/m512/m16bcst

Instruction Operand Encoding

Instruction	Operand 1	Operand 2	Operand 3
VCVTPH2BF4S	MODRM.REG(w)	MODRM.R/M(r)	N/A

Operation

```
DEFINE vcvtph2bf4s(src, VL, src_is_mem, evex_b):
    ASSERT VL in (128, 256, 512)
    KL = VL / 16
    FOR i = 0 TO KL-1:
        t = src.bf16[0] if (src_is_mem AND evex_b) else src.bf16[i]
        dest[4*i+3 : 4*i] = fp16_to_fp4(t)
    dest[MAXVL-1 : VL / 4] = 0
```

Other Exceptions

Exception class E4NF. See Section 10.3.

4.1.2 FP16 → INT4 (single source)

Convert packed FP16 source register to INT4 (signed or unsigned) destination. Results always saturate to format maximum. Rounding mode: RNE.

Output nibble-packed (2 per byte). Result masking is not supported. Destination upper bits zeroed.

Variants

Mnemonic	Output format
VCVTPH2SNS	Signed INT4, with saturation
VCVTPH2UNS	Unsigned INT4, with saturation

Operand forms

The operation is narrowing; the destination is a quarter the width of the source.

Operand 1	Operand 2
xmm1	xmm2/m128/m16bcst
xmm1	ymm2/m256/m16bcst
xmm1	zmm2/m512/m16bcst

Instruction Operand Encoding

Instruction	Operand 1	Operand 2	Operand 3
VCVTPH2SNS VCVTPH2UNS	MODRM.REG(w)	MODRM.R/M(r)	N/A

Operation

```
DEFINE vcvtph2sns(src, VL, src_is_mem, evex_b):
    ASSERT VL in (128, 256, 512)
    KL = VL / 16
    FOR i = 0 TO KL-1:
        t = src.bf16[0] if (src_is_mem AND evex_b) else src.bf16[i]
        dest[4*i+3 : 4*i] = fp16_to_int4(t, signed=1)
    dest[MAXVL-1 : VL / 4] = 0
```

```
DEFINE vcvtph2uns(src, VL, src_is_mem, evex_b):
    ASSERT VL in (128, 256, 512)
    KL = VL / 16
    FOR i = 0 TO KL-1:
        t = src.bf16[0] if (src_is_mem AND evex_b) else src.bf16[i]
        dest[4*i+3 : 4*i] = fp16_to_int4(t, signed=0)
    dest[MAXVL-1 : VL / 4] = 0
```

Other Exceptions

Exception class E4NF. See Section 10.3.

4.1.3 FP16 → FP4 E2M1 (two sources)

Convert two packed FP16 source registers into a single FP4 destination. Results always saturate to format maximum. Rounding mode: RNE.

Lower KL/2 elements from src2; upper KL/2 from src1. Output nibble-packed (2 per byte). Result masking is not supported.

Variants

Mnemonic	Output format
VCVT2PH2BF4S	FP4 E2M1

Operand forms

The operation is narrowing; the destination is half the width of each source.

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

Instruction Operand Encoding

Instruction	Operand 1	Operand 2	Operand 3
VCVT2PH2BF4S	MODRM.REG(w)	VVVV(r)	MODRM.R/M(r)

Operation

```
DEFINE vcvt2ph2bf4s(src1, src2, VL, src_is_mem, evex_b):
    ASSERT VL in (128, 256, 512)
    KL = VL / 8
    FOR i = 0 TO KL / 2-1:
        t2 = src2.fp16[0] if (src_is_mem AND evex_b) else src2.fp16[i]
        dest[4*i+3 : 4*i] = fp16_to_fp4(t2)
    FOR i = 0 TO KL / 2-1:
        dest[4*(KL/2+i)+3 : 4*(KL/2+i)] = fp16_to_fp4(src1.fp16[i])
    dest[MAXVL-1 : VL / 2] = 0
```

Other Exceptions

Exception class E4NF. See Section 10.3.

4.1.4 FP16 → INT4 (two sources)

Convert two packed FP16 source registers into a INT4 (signed or unsigned) destination. Results always saturate to format maximum. Rounding mode: RNE.

Lower KL/2 elements from src2; upper KL/2 from src1. Output nibble-packed (2 per byte). Result masking is not supported.

Variants

Mnemonic	Output format
VCVT2PH2SNS	S4 (signed INT4)
VCVT2PH2UNS	U4 (unsigned INT4)

Operand forms

The operation is narrowing; the destination is half the width of each source.

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

Instruction Operand Encoding

Instruction	Operand 1	Operand 2	Operand 3
VCVT2PH2SNS VCVT2PH2UNS	MODRM.REG(w)	VVVV(r)	MODRM.R/M(r)

Operation

```
DEFINE vcvt2ph2sns(src1, src2, VL, src_is_mem, evex_b):
    ASSERT VL in (128, 256, 512)
    KL = VL / 8
    FOR i = 0 TO KL / 2-1:
        t2 = src2.fp16[0] if (src_is_mem AND evex_b) else src2.fp16[i]
        dest[4*i+3 : 4*i] = fp16_to_int4(t2, signed=1)
    FOR i = 0 TO KL / 2-1:
        dest[4*(KL/2+i)+3 : 4*(KL/2+i)] = fp16_to_int4(src1.fp16[i], signed=1)
```

```
    dest[MAXVL-1 : VL / 2] = 0
```

```
DEFINE vcvt2ph2uns(src1, src2, VL, src_is_mem, evex_b):
    ASSERT VL in (128, 256, 512)
    KL = VL / 8
    FOR i = 0 TO KL / 2-1:
        t2 = src2.fp16[0] if (src_is_mem AND evex_b) else src2.fp16[i]
        dest[4*i+3 : 4*i] = fp16_to_int4(t2, signed=0)
    FOR i = 0 TO KL / 2-1:
        dest[4*(KL/2+i)+3 : 4*(KL/2+i)] = fp16_to_int4(src1.fp16[i], signed=0)
    dest[MAXVL-1 : VL / 2] = 0
```

Other Exceptions
Exception class E4NF. See Section 10.3.

4.1.5 FP16 → FP4 with Bias

Convert packed FP16 source register to FP4 with biased rounding. Results always saturate to format maximum.

The first source provides a bias value which is added to the mantissa of source FP16 before truncation. The instruction is agnostic to how the bias is generated; supplying pseudo-random bias achieves stochastic rounding.

Output nibble-packed (2 per byte). Result masking is not supported. Destination upper bits zeroed.

Variants

Mnemonic	Output format
VCVTBIASPH2BF4S	FP4 E2M1, saturate

Operand forms
The operation is narrowing; the destination is a quarter the width of the source.

Operand 1	Operand 2	Operand 3
xmm1	xmm2	xmm3/m128/m16bcst
xmm1	ymm2	ymm3/m256/m16bcst
xmm1	zmm2	zmm3/m512/m16bcst

Instruction Operand Encoding

Instruction	Operand 1	Operand 2	Operand 3
VCVTBIASPH2BF4S	MODRM.REG(w)	VVVV(r)	MODRM.R/M(r)

Operation

```
DEFINE vcvtbiasph2bf4s(bias_src, data_src, VL, src_is_mem, evex_b):
    ASSERT VL in (128, 256, 512)
    KL = VL / 16
    FOR i = 0 TO KL-1:
        bias = bias_src.byte[2 * i]
        t    = data_src.fp16[0] if (src_is_mem AND evex_b) else data_src.fp16[i]
        dest[4*i+3 : 4*i] = bias_round_fp16_to_bf4(bias, t)
    dest[MAXVL-1 : VL / 4] = 0
```

Other Exceptions
Exception class E4NF. See Section 10.3.

4.1.6 BF16 → FP4 E2M1 (single source)

Convert packed BF16 source register to FP4 destination. Results always saturate to format maximum. Rounding mode: RNE.

Output nibble-packed (2 per byte). Result masking is not supported. Destination upper bits zeroed.

Variants

Mnemonic	Output format
VCVTBF162BF4S	FP4 E2M1

Operand forms

The operation is narrowing; the destination is a quarter the width of the source.

Operand 1	Operand 2
xmm1	xmm2/m128/m16bcst
xmm1	ymm2/m256/m16bcst
xmm1	zmm2/m512/m16bcst

Instruction Operand Encoding

Instruction	Operand 1	Operand 2	Operand 3
VCVTBF162BF4S	MODRM.REG(w)	MODRM.R/M(r)	N/A

Operation

```
DEFINE vcvtbf162bf4s(src, VL, src_is_mem, evex_b):
    ASSERT VL in (128, 256, 512)
    KL = VL / 16
    FOR i = 0 TO KL-1:
        t = src.bf16[0] if (src_is_mem AND evex_b) else src.bf16[i]
        dest[4*i+3 : 4*i] = bf16_to_fp4(t)
    dest[MAXVL-1 : VL / 4] = 0
```

Other Exceptions

Exception class E4NF. See Section 10.3.

4.1.7 BF16 → INT4 (single source)

Convert packed BF16 source register to INT4 (signed or unsigned) destination. Results always saturate to format maximum. Rounding mode: RNE.

Output nibble-packed (2 per byte). Result masking is not supported. Destination upper bits zeroed.

Variants

Mnemonic	Output format
VCVTBF162SNS	S4 (signed INT4)
VCVTBF162UNS	U4 (unsigned INT4)

Operand forms

The operation is narrowing; the destination is a quarter the width of the source.

Operand 1	Operand 2
xmm1	xmm2/m128/m16bcst
xmm1	ymm2/m256/m16bcst
xmm1	zmm2/m512/m16bcst

Instruction Operand Encoding

Instruction	Operand 1	Operand 2	Operand 3
VCVTBF162SNS VCVTBF162UNS	MODRM.REG(w)	MODRM.R/M(r)	N/A

Operation

```
DEFINE vcvtbf162sns(src, VL, src_is_mem, evex_b):
    ASSERT VL in (128, 256, 512)
    KL = VL / 16
    FOR i = 0 TO KL-1:
        t = src.bf16[0] if (src_is_mem AND evex_b) else src.bf16[i]
        dest[4*i+3 : 4*i] = bf16_to_int4(t, signed=1)
    dest[MAXVL-1 : VL / 4] = 0
```

```
DEFINE vcvtbf162uns(src, VL, src_is_mem, evex_b):
    ASSERT VL in (128, 256, 512)
    KL = VL / 16
    FOR i = 0 TO KL-1:
        t = src.bf16[0] if (src_is_mem AND evex_b) else src.bf16[i]
        dest[4*i+3 : 4*i] = bf16_to_int4(t, signed=0)
    dest[MAXVL-1 : VL / 4] = 0
```

Other Exceptions

Exception class E4NF. See Section 10.3.

4.1.8 BF16 → FP4 E2M1 (two sources)

Convert two packed BF16 source registers into a single FP4 destination. Results always saturate to format maximum. Rounding mode: RNE.

Lower KL/2 elements from src2; upper KL/2 from src1. Output nibble-packed (2 per byte). Result masking is not supported.

Variants

Mnemonic	Output format
VCVT2BF162BF4S	FP4 E2M1

Operand forms

The operation is narrowing; the destination is half the width of each source.

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

Instruction Operand Encoding

Instruction	Operand 1	Operand 2	Operand 3
VCVT2BF162BF4S	MODRM.REG(w)	VVVV(r)	MODRM.R/M(r)

Operation

```
DEFINE vcvt2bf162bf4s(src1, src2, VL, src_is_mem, evex_b):
    ASSERT VL in (128, 256, 512)
    KL = VL / 8
    FOR i = 0 TO KL / 2-1:
        t2 = src2.bf16[0] if (src_is_mem AND evex_b) else src2.bf16[i]
        dest[4*i+3 : 4*i] = bf16_to_fp4(t2)
    FOR i = 0 TO KL / 2-1:
        dest[4*(KL/2+i)+3 : 4*(KL/2+i)] = bf16_to_fp4(src1.bf16[i])
    dest[MAXVL-1 : VL / 2] = 0
```

Other Exceptions

Exception class E4NF. See Section 10.3.

4.1.9 BF16 → INT4 (two sources)

Convert two packed BF16 source registers into a INT4 (signed or unsigned) destination. Results always saturate to format maximum. Rounding mode: RNE.

Lower KL/2 elements from src2; upper KL/2 from src1. Output nibble-packed (2 per byte). Result masking is not supported.

Variants

Mnemonic	Output format
VCVT2BF162SNS	S4 (signed INT4)
VCVT2BF162UNS	U4 (unsigned INT4)

Operand forms

The operation is narrowing; the destination is half the width of each source.

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

Instruction Operand Encoding

Instruction	Operand 1	Operand 2	Operand 3
VCVT2BF162SNS VCVT2BF162UNS	MODRM.REG(w)	VVVV(r)	MODRM.R/M(r)

Operation

```
DEFINE vcvt2bf162sns(src1, src2, VL, src_is_mem, evex_b):
    ASSERT VL in (128, 256, 512)
    KL = VL / 8
    FOR i = 0 TO KL / 2-1:
        t2 = src2.bf16[0] if (src_is_mem AND evex_b) else src2.bf16[i]
        dest[4*i+3 : 4*i] = bf16_to_int4(t2, signed=1)
    FOR i = 0 TO KL / 2-1:
        dest[4*(KL/2+i)+3 : 4*(KL/2+i)] = bf16_to_int4(src1.bf16[i], signed=1)
    dest[MAXVL-1 : VL / 2] = 0
```

```
DEFINE vcvt2bf162uns(src1, src2, VL, src_is_mem, evex_b):
    ASSERT VL in (128, 256, 512)
    KL = VL / 8
    FOR i = 0 TO KL / 2-1:
        t2 = src2.bf16[0] if (src_is_mem AND evex_b) else src2.bf16[i]
        dest[4*i+3 : 4*i] = bf16_to_int4(t2, signed=0)
    FOR i = 0 TO KL / 2-1:
        dest[4*(KL/2+i)+3 : 4*(KL/2+i)] = bf16_to_int4(src1.bf16[i], signed=0)
    dest[MAXVL-1 : VL / 2] = 0
```

Other Exceptions

Exception class E4NF. See Section 10.3.

4.1.10 BF16 → FP4 with Bias

Convert packed BF16 source register to FP4 with biased rounding. Results always saturate to format maximum.

The first source provides a bias value which is added to the mantissa of source BF16 before truncation. The instruction is agnostic to how the bias is generated; supplying pseudo-random bias achieves stochastic rounding.

Output nibble-packed (2 per byte). Result masking is not supported. Destination upper bits zeroed.

Variants

Mnemonic	Output format
VCVTBIASBF162BF4S	FP4 E2M1, saturate

Operand forms

The operation is narrowing; the destination is a quarter the width of the source.

Operand 1	Operand 2	Operand 3
xmm1	xmm2	xmm3/m128/m16bcst
xmm1	ymm2	ymm3/m256/m16bcst
xmm1	zmm2	zmm3/m512/m16bcst

Instruction Operand Encoding

Instruction	Operand 1	Operand 2	Operand 3
VCVTBIASBF162BF4S	MODRM.REG(w)	VVVV(r)	MODRM.R/M(r)

Operation

```
DEFINE vcvtbiasbf162bf4s(bias_src, data_src, VL, src_is_mem, evex_b):
    ASSERT VL in (128, 256, 512)
    KL = VL / 16
    FOR i = 0 TO KL-1:
        bias = bias_src.byte[2 * i]
        t    = data_src.bf16[0] if (src_is_mem AND evex_b) else data_src.bf16[i]
        dest[4*i+3 : 4*i] = bias_round_fp16_to_bf4(bias, float(t))
    dest[MAXVL-1 : VL / 4] = 0
```

Other Exceptions

Exception class E4NF. See Section 10.3.

4.2 Instruction Encodings

4.2.1 FP16 to FP4/INT4 Converts (Single-Source)

Mnemonic	Form	Encoding
VCVTPH2BF4S	xmm1, xmm2/m128/m16bcst	EVEX.128.F3.MAP5.W0 76
VCVTPH2BF4S	xmm1, ymm2/m256/m16bcst	EVEX.256.F3.MAP5.W0 76
VCVTPH2BF4S	xmm1, zmm2/m512/m16bcst	EVEX.512.F3.MAP5.W0 76
VCVTPH2SNS	xmm1, xmm2/m128/m16bcst	EVEX.128.F3.MAP5.W0 77
VCVTPH2SNS	xmm1, ymm2/m256/m16bcst	EVEX.256.F3.MAP5.W0 77
VCVTPH2SNS	xmm1, zmm2/m512/m16bcst	EVEX.512.F3.MAP5.W0 77
VCVTPH2UNS	xmm1, xmm2/m128/m16bcst	EVEX.128.F3.MAP5.W0 75
VCVTPH2UNS	xmm1, ymm2/m256/m16bcst	EVEX.256.F3.MAP5.W0 75
VCVTPH2UNS	xmm1, zmm2/m512/m16bcst	EVEX.512.F3.MAP5.W0 75

4.2.2 FP16 to FP4/INT4 Converts (Two-Source)

Mnemonic	Form	Encoding
VCVT2PH2BF4S	xmm1, xmm2, xmm3/m128/m16bcst	EVEX.128.F2.MAP5.W0 76
VCVT2PH2BF4S	xmm1, ymm2, ymm3/m256/m16bcst	EVEX.256.F2.MAP5.W0 76
VCVT2PH2BF4S	ymm1, zmm2, zmm3/m512/m16bcst	EVEX.512.F2.MAP5.W0 76
VCVT2PH2SNS	xmm1, xmm2, xmm3/m128/m16bcst	EVEX.128.F2.MAP5.W0 77
VCVT2PH2SNS	xmm1, ymm2, ymm3/m256/m16bcst	EVEX.256.F2.MAP5.W0 77
VCVT2PH2SNS	ymm1, zmm2, zmm3/m512/m16bcst	EVEX.512.F2.MAP5.W0 77
VCVT2PH2UNS	xmm1, xmm2, xmm3/m128/m16bcst	EVEX.128.F2.MAP5.W0 75
VCVT2PH2UNS	xmm1, ymm2, ymm3/m256/m16bcst	EVEX.256.F2.MAP5.W0 75
VCVT2PH2UNS	ymm1, zmm2, zmm3/m512/m16bcst	EVEX.512.F2.MAP5.W0 75

4.2.3 FP16 to FP4/INT4 Converts (Biased)

Mnemonic	Form	Encoding
VCVTBIASPH2BF4S	xmm1, xmm2, xmm3/m128/m16bcst	EVEX.128.NP.MAP5.W0 76
VCVTBIASPH2BF4S	xmm1, ymm2, ymm3/m256/m16bcst	EVEX.256.NP.MAP5.W0 76
VCVTBIASPH2BF4S	xmm1, zmm2, zmm3/m512/m16bcst	EVEX.512.NP.MAP5.W0 76

4.2.4 BF16 to INT4 Converts

Mnemonic	Form	Encoding
VCVTBF162SNS	xmm1, xmm2/m128/m16bcst	EVEX.128.F3.MAP5.W1 77
VCVTBF162SNS	xmm1, ymm2/m256/m16bcst	EVEX.256.F3.MAP5.W1 77
VCVTBF162SNS	xmm1, zmm2/m512/m16bcst	EVEX.512.F3.MAP5.W1 77
VCVTBF162UNS	xmm1, xmm2/m128/m16bcst	EVEX.128.F3.MAP5.W1 75
VCVTBF162UNS	xmm1, ymm2/m256/m16bcst	EVEX.256.F3.MAP5.W1 75
VCVTBF162UNS	xmm1, zmm2/m512/m16bcst	EVEX.512.F3.MAP5.W1 75

4.2.5 BF16 to FP4/INT4 Converts (Two-Source)

Mnemonic	Form	Encoding
VCVT2BF162BF4S	xmm1, xmm2, xmm3/m128/m16bcst	EVEX.128.F2.MAP5.W1 76
VCVT2BF162BF4S	xmm1, ymm2, ymm3/m256/m16bcst	EVEX.256.F2.MAP5.W1 76
VCVT2BF162BF4S	ymm1, zmm2, zmm3/m512/m16bcst	EVEX.512.F2.MAP5.W1 76
VCVT2BF162SNS	xmm1, xmm2, xmm3/m128/m16bcst	EVEX.128.F2.MAP5.W1 77
VCVT2BF162SNS	xmm1, ymm2, ymm3/m256/m16bcst	EVEX.256.F2.MAP5.W1 77
VCVT2BF162SNS	ymm1, zmm2, zmm3/m512/m16bcst	EVEX.512.F2.MAP5.W1 77
VCVT2BF162UNS	xmm1, xmm2, xmm3/m128/m16bcst	EVEX.128.F2.MAP5.W1 75
VCVT2BF162UNS	xmm1, ymm2, ymm3/m256/m16bcst	EVEX.256.F2.MAP5.W1 75
VCVT2BF162UNS	ymm1, zmm2, zmm3/m512/m16bcst	EVEX.512.F2.MAP5.W1 75

4.2.6 BF16 to FP4/INT4 Converts (Biased)

Mnemonic	Form	Encoding
VCVTBIASBF162BF4S	xmm1, xmm2, xmm3/m128/m16bcst	EVEX.128.NP.MAP5.W1 76
VCVTBIASBF162BF4S	xmm1, ymm2, ymm3/m256/m16bcst	EVEX.256.NP.MAP5.W1 76
VCVTBIASBF162BF4S	xmm1, zmm2, zmm3/m512/m16bcst	EVEX.512.NP.MAP5.W1 76

4.2.7 BF16 to FP4 Converts

Mnemonic	Form	Encoding
VCVTBF162BF4S	xmm1, xmm2/m128/m16bcst	EVEX.128.F3.MAP5.W1 76
VCVTBF162BF4S	xmm1, ymm2/m256/m16bcst	EVEX.256.F3.MAP5.W1 76
VCVTBF162BF4S	xmm1, zmm2/m512/m16bcst	EVEX.512.F3.MAP5.W1 76

5 AVX10_VPINTERLEAVE — Interleave Byte / Word

These instructions interleave elements from two source registers. Unlike `VPUNPCK[H|L]P[D|S]`, which performs interleaving independently within each 128-bit lane, the VPINTERLEAVE instructions treat the full vector as a single linear array. The L and H suffixes select which half of the full vector is consumed from each source — the low half (`[VL/2-1:0]`) for L variants and the high half (`[VL-1:VL/2]`) for H variants.

Write masking applies at output element granularity: byte elements for byte interleave variants, word elements for word interleave variants.

Group	Mnemonics
Interleave bytes (low / high)	VPINTERLEAVELB / VPINTERLEAVEHB
Interleave words (low / high)	VPINTERLEAVELW / VPINTERLEAVEHW

5.1 Instructions

5.1.1 VPINTERLEAVELB / VPINTERLEAVEHB — Interleave Bytes

Two-source byte interleave. Each instruction reads VL/16 bytes of data from each source and produces VL/8 output bytes. Output bytes `2j` and `2j+1` contain byte `j` from src1 and byte `j` from src2 respectively.

The [H|L] variant selects which half of each source is read:

- VPINTERLEAVELB: reads bytes 0..(VL/16)−1 of each source (lower half).
- VPINTERLEAVEHB: reads bytes (VL/16)..(VL/8)−1 of each source (upper half).

Write-mask at output byte element granularity.

Operand forms

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

Instruction Operand Encoding

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

Operation

```
DEFINE vpinterleaveb(src1, src2, VL, k1, zeroing, hi=False, no_writemask=False):
    ASSERT VL in (128, 256, 512)
    KL   = VL / 8
    base = KL / 2 if hi else 0
    orig_dest = copy(dest)
    FOR i = 0 TO KL-1:
        IF k1[i] OR no_writemask:
            IF i % 2 == 0:
                dest.byte[i] = src1.byte[base + i / 2]
            ELSE:
                dest.byte[i] = src2.byte[base + i / 2]
        ELSE IF zeroing:
            dest.byte[i] = 0
        ELSE:
            dest.byte[i] = orig_dest.byte[i]
    dest[MAXVL-1 : VL] = 0
```

Other Exceptions

Exception class E4NF. See Section 10.3.

5.1.2 VPINTERLEAVELW / VPINTERLEAVEHW — Interleave Words

Two-source word interleave. Each instruction reads VL/32 words of data from each source and produces VL/16 output words. Output words `2j` and `2j+1` contain word `j` from src1 and word `j` from src2 respectively.

The [H|L] variant selects which half of each source is read:

- VPINTERLEAVELW: reads words 0..(VL/32)−1 of each source (lower half).
- VPINTERLEAVEHW: reads words (VL/32)..(VL/16)−1 of each source (upper half).

Write-mask at output word element granularity.

Operand forms

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

Instruction Operand Encoding

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

Operation

```
DEFINE vpinterleavew(src1, src2, VL, k1, zeroing, hi=False, no_writemask=False):
    ASSERT VL in (128, 256, 512)
    KL   = VL / 16
    base = KL / 2 if hi else 0
    orig_dest = copy(dest)
    FOR i = 0 TO KL-1:
        IF k1[i] OR no_writemask:
            IF i % 2 == 0:
                dest.word[i] = src1.word[base + i / 2]
            ELSE:
                dest.word[i] = src2.word[base + i / 2]
        ELSE IF zeroing:
            dest.word[i] = 0
        ELSE:
            dest.word[i] = orig_dest.word[i]
    dest[MAXVL-1 : VL] = 0
```

Other Exceptions

Exception class E4NF. See Section 10.3.

5.2 Instruction Encodings

5.2.1 VPINTERLEAVEB

Mnemonic	Form	Encoding
VPINTERLEAVELB	xmm1{k1}{z}, xmm2, xmm3/m128	EVEX.128.66.MAP6.W0 60
VPINTERLEAVELB	ymm1{k1}{z}, ymm2, ymm3/m256	EVEX.256.66.MAP6.W0 60
VPINTERLEAVELB	zmm1{k1}{z}, zmm2, zmm3/m512	EVEX.512.66.MAP6.W0 60
VPINTERLEAVEHB	xmm1{k1}{z}, xmm2, xmm3/m128	EVEX.128.66.MAP6.W0 68
VPINTERLEAVEHB	ymm1{k1}{z}, ymm2, ymm3/m256	EVEX.256.66.MAP6.W0 68
VPINTERLEAVEHB	zmm1{k1}{z}, zmm2, zmm3/m512	EVEX.512.66.MAP6.W0 68

5.2.2 VPINTERLEAVEW

Mnemonic	Form	Encoding
VPINTERLEAVELW	xmm1{k1}{z}, xmm2, xmm3/m128	EVEX.128.66.MAP6.W0 61
VPINTERLEAVELW	ymm1{k1}{z}, ymm2, ymm3/m256	EVEX.256.66.MAP6.W0 61
VPINTERLEAVELW	zmm1{k1}{z}, zmm2, zmm3/m512	EVEX.512.66.MAP6.W0 61
VPINTERLEAVEHW	xmm1{k1}{z}, xmm2, xmm3/m128	EVEX.128.66.MAP6.W0 69
VPINTERLEAVEHW	ymm1{k1}{z}, ymm2, ymm3/m256	EVEX.256.66.MAP6.W0 69
VPINTERLEAVEHW	zmm1{k1}{z}, zmm2, zmm3/m512	EVEX.512.66.MAP6.W0 69

6 AVX10_BMM — Bit Matrix Multiply

CPUID: Leaf 24H, sub-leaf 1, ECX[1] (AVX10_BMM)

The AVX10_BMM instructions are optional; they are independently enumerated and not implied by AVX10_VSN ≥ 3. See Table 5.

6.1 Overview

Group	Mnemonics
Bit matrix multiply-accumulate (XOR reduction)	VBMACXOR16x16x16
Bit matrix multiply-accumulate (OR reduction)	VBMACOR16x16x16
Bit reversal within byte	VBITREV

The BMM instructions compute non-transposed GF(2) matrix multiplication over 16×16 bit matrices [3]. Each 256-bit register lane encodes one 16×16 matrix in row-major order: `reg[255:0] = {M[15][15:0], M[14][15:0], ..., M[0][15:0]}`. Both A (src2) and B (src3) are stored in this row-major layout; B is accessed by column j across rows k — standard non-transposed matrix multiply.

These instructions extend GF(2) linear algebra capabilities beyond the 8×8 matrix-vector products available via `GF2P8AFFINEQB` (GFNI) to full 16×16 matrix-matrix products, with two independent computations per ZMM.

6.1.1 VBMACXOR16x16x16 — Bit Matrix Multiply Accumulate (XOR reduction)

Computes `D = D XOR (A × B)` where D, A, B are 16×16 GF(2) matrices (non-transposed). Two independent 16×16 multiplications per ZMM (one per 256-bit half). This is the standard GF(2) matrix multiply (LFSR, FEC, linear code operations): `D[i][j] ^= XOR_k( A[i][k] AND B[k][j] )` for all (i,j).

Operand forms

No write-masking or memory broadcast; instruction is available only at VL=256 and VL=512.

Operand 1	Operand 2	Operand 3
ymm1	ymm2	ymm3/m256
zmm1	zmm2	zmm3/m512

Instruction Operand Encoding

Instruction	Operand 1	Operand 2	Operand 3
VBMACXOR16x16x16	MODRM.REG(rw)	VVVV(r)	MODRM.R/M(r)

Other Exceptions

Exception class E4NF. See Section 10.3.

Operation

```
DEFINE vbmacxor16x16x16(dst_acc, src1_a, src2_b, VL):
    // VBMACXOR16x16x16: D ^= A × B over GF(2), XOR reduction. B non-transposed.
    // dest[i][j] = src1[i][j] XOR_k( A[i][k] AND B[k][j] ) — standard GF(2) matrix multiply.
    ASSERT VL in (256, 512)
    num_mats = VL / 256
    FOR m = 0 TO num_mats-1:
        base = m * 4
        FOR i = 0 TO 15:
            a_row = (src1_a.qword[base / 4 + i / 4] >> ((i % 4) * 16)) & 0xFFFF
            FOR j = 0 TO 15:
                bit_pos = m * 256 + i * 16 + j
                reduction_bit = dst_acc[bit_pos]
                FOR k = 0 TO 15:
                    b_row_k = (src2_b.qword[base / 4 + k / 4] >> ((k % 4) * 16)) & 0xFFFF
```

```
                    reduction_bit ^= ((a_row >> k) & 1) & ((b_row_k >> j) & 1)
                dst_acc[bit_pos] = reduction_bit
```

6.1.2 VBMACOR16x16x16 — Bit Matrix Multiply Accumulate (OR reduction)

Computes `D = D OR (A × B)` with Boolean OR reduction (non-transposed). Used for Boolean matrix multiplication (reachability, transitive closure): `D[i][j] |= OR_k( A[i][k] AND B[k][j] )` for all (i,j).

Operand forms

No write-masking or memory broadcast; instruction is available only at VL=256 and VL=512.

Operand 1	Operand 2	Operand 3
ymm1	ymm2	ymm3/m256
zmm1	zmm2	zmm3/m512

Instruction Operand Encoding

Instruction	Operand 1	Operand 2	Operand 3
VBMACOR16x16x16	MODRM.REG(rw)	VVVV(r)	MODRM.R/M(r)

Other Exceptions

Exception class E4NF. See Section 10.3.

Operation

```
DEFINE vbmacor16x16x16(dst_acc, src1_a, src2_b, VL):
    // VBMACOR16x16x16: D |= A × B, Boolean OR reduction. B non-transposed.
    // dest[i][j] = src1[i][j] OR_k( A[i][k] AND B[k][j] ) — Boolean matrix multiply.
    ASSERT VL in (256, 512)
    num_mats = VL / 256
    FOR m = 0 TO num_mats-1:
        base = m * 4
        FOR i = 0 TO 15:
            a_row = (src1_a.qword[base / 4 + i / 4] >> ((i % 4) * 16)) & 0xFFFF
            FOR j = 0 TO 15:
                bit_pos = m * 256 + i * 16 + j
                reduction_bit = dst_acc[bit_pos]
                FOR k = 0 TO 15:
                    b_row_k = (src2_b.qword[base / 4 + k / 4] >> ((k % 4) * 16)) & 0xFFFF
                    reduction_bit |= ((a_row >> k) & 1) & ((b_row_k >> j) & 1)
                dst_acc[bit_pos] = reduction_bit
```

6.1.3 VBITREV — Bit Reversal within Each Byte (K-masked)

Reverses the bit order within each byte of the source register. Masking is available at byte granularity; if the mask bit is set, the byte is bit-reversed; otherwise merge or zero masking applies.

Operand forms

Source and destination share the same vector width; masking applies at byte (8-bit element) granularity.

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

Instruction Operand Encoding

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

Other Exceptions

Exception class E4. See Section 10.3.

Operation

```
DEFINE vbitrev(src, VL, k1, zeroing, no_writemask=False):
    // VBITREV: bit reversal within each byte element.
    ASSERT VL in (128, 256, 512)
    KL = VL / 8
    orig_dest = copy(dest)
    FOR i = 0 TO KL-1:
        IF k1[i] OR no_writemask:
            b = src.byte[i]
            dest.byte[i] = bit_reverse_byte(b)
        ELSE IF zeroing:
            dest.byte[i] = 0
        ELSE:
            dest.byte[i] = orig_dest.byte[i]
    dest[MAXVL-1 : VL] = 0
```

6.2 Instruction Encodings

6.2.1 Bit Matrix Multiply (BMM)

Mnemonic	Form	Encoding
VBMACXOR16x16x16	ymm1, ymm2, ymm3/m256	EVEX.256.NP.MAP6.W1 80
VBMACXOR16x16x16	zmm1, zmm2, zmm3/m512	EVEX.512.NP.MAP6.W1 80
VBMACOR16x16x16	ymm1, ymm2, ymm3/m256	EVEX.256.NP.MAP6.W0 80
VBMACOR16x16x16	zmm1, zmm2, zmm3/m512	EVEX.512.NP.MAP6.W0 80
VBITREV	xmm1{k1}{z}, xmm2/m128	EVEX.128.NP.MAP6.W0 81
VBITREV	ymm1{k1}{z}, ymm2/m256	EVEX.256.NP.MAP6.W0 81
VBITREV	zmm1{k1}{z}, zmm2/m512	EVEX.512.NP.MAP6.W0 81

7 Tile Management Instructions

7.1 LDTILECFG — Load Tile Configuration

LDTILECFG loads a 64-byte tile configuration descriptor from the memory operand. If the descriptor specifies a valid palette, the tile configuration is set accordingly. An invalid descriptor results in a #GP fault.

A successful execution zeroes all TILEDATA bytes and initialises SCALEDATA (Block Scale Register) bytes to INIT state (`0x7F`).

Operand 1
m512

Instruction Operand Encoding

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

Operation

See ACE v1 Specification [1] for pseudocode and configuration descriptor layout.

Other Exceptions

VEX form: AMX-E1, see Section 10.5.1.
EVEX form: AMX-E1-EVEX, see Section 10.5.3.

7.2 STTILECFG — Store Tile Configuration

STTILECFG stores the current 64-byte tile configuration descriptor to the memory operand. If tiles are not configured, 64 bytes of zero are stored.

Operand 1
m512

Instruction Operand Encoding

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

Operation

See ACE v1 Specification [1] for pseudocode.

Other Exceptions

VEX form: AMX-E2, see Section 10.5.2.
EVEX form: AMX-E2-EVEX, see Section 10.5.4.

7.3 Instruction Encodings

7.3.1 LDTILECFG

Mnemonic	Form	Encoding
LDTILECFG	m512	VEX.128.NP.0F38.W0 49 !(11):000:bbb
LDTILECFG	m512	EVEX.128.NP.0F38.W0 49 !(11):000:bbb {NF=0}

7.3.2 STTILECFG

Mnemonic	Form	Encoding
STTILECFG	m512	VEX.128.66.0F38.W0 49 !(11):000:bbb
STTILECFG	m512	EVEX.128.66.0F38.W0 49 !(11):000:bbb {NF=0}

8 ACE v2 Extensions

8.1 Instruction Groups

ACE v2 extends the ACE v1 tile architecture with outer product operations for reduced-precision AI, wider AI, and HPC formats.

All operations use the ACE tile register file: 16 rows × 512 bits (64 bytes) per row, holding 16 × 16 FP32 or INT32 elements, or 16 × 8 FP64 elements.

The following table summarizes the extensions defined in ACE v2.

Mnemonic	Source format	Rank	Enumeration	Description
TOP8FMXBF4PS	MX FP4	8	ACE v2	OCP MX FP4 rank-8, FP32 acc
TOP8FMXN[SS/SU/US/UU]PS	MX INT4	8	ACE v2	OCP MX INT4 rank-8, FP32 acc
TOP4MXB[SU/US/UU]PS	MX INT8	4	ACE v2	MX INT8 extended variants, FP32 acc
TOP2PHPS	FP16	2	ACE_OP_FP16 (See Table 7)	FP16 rank-2, FP32 acc
TOP1PS	FP32	1	ACE_OP_FP32	FP32 rank-1, FP32 FMA acc
TOP1PD	FP64	1	ACE_OP_FP64	FP64 rank-1, FP64 FMA acc
(format extension)	E5M3	–	ACE v2	E5M3 non-power-of-two block scale

8.2 Reduced-Precision AI Outer Products

These instructions target neural network inference and training with sub-byte and byte-width data formats, using OCP MX block scaling [4] where applicable.

Memory Layout

The ACE v2 byte width outer products use the same rank-4 dword packing as ACE v1. The ACE v2 sub-byte width outer products use rank-8 dword packing which is a natural progression.

Rank-8 memory layout (for FP4 and INT4 — 8 nibbles per 32-bit dword lane):

```
// A matrix: transposed at 32-bit granularity; 8 nibbles per lane
uint8_t AT_mem[K/8][M][4];  // 4 bytes = 8 nibbles
AT_mem[k/8][m][j_byte] = packed_nibbles(A[m][k + 2*j_byte],
                                        A[m][k + 2*j_byte + 1]);

// B matrix: column-interleaved at nibble granularity
uint8_t B_pack[K/8][N][4];
B_pack[k/8][n][j_byte] = packed_nibbles(B[k + 2*j_byte    ][n],
                                        B[k + 2*j_byte + 1][n]);
```

Both `AT_mem` and `B_pack` share the same `[K/8][16][4]` structure, making the transposed GEMM pattern (operand swap to obtain column-slice output) equally applicable for rank-8. See ACE v1 Specification [1] section 10.4 for the transposed GEMM pattern.

8.2.1 TOP8FMXBF4PS — MX FP4 Rank-8 Outer Product, FP32 Accumulate

Rank-8 outer product of MX FP4 (FP4 E2M1) inputs accumulating into a 16×16 FP32 tile.

Block scaling via BSR. ACE v2 introduces E5M3 block scaling support for eligible outer product operations, see Section 8.4. Both A and B inputs use the same block scale format, selected by imm8 bit [6]: E8M0 (power-of-two, default) or E5M3 (non-power-of-two).

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

Instruction Operand Encoding

Instruction	Operand 1	Operand 2	Operand 3	Operand 4
TOP8FMXBF4PS	MODRM.REG(rw)	MODRM.R/M(r)	VVVV(r)	IMM8(r)

Immediate Encoding

Bits	Field	Description
[1:0]	ASCALE_GRP	A-scale group select (BSR upper half, groups 0–3)
[2]	—	Reserved, must be 0
[4:3]	BSCALE_GRP	B-scale group select (BSR lower half, groups 0–3)
[5]	—	Reserved, must be 0
[6]	SCALE_FMT	A- and B-scale format: 0 = E8M0, 1 = E5M3
[7]	—	Reserved, must be 0

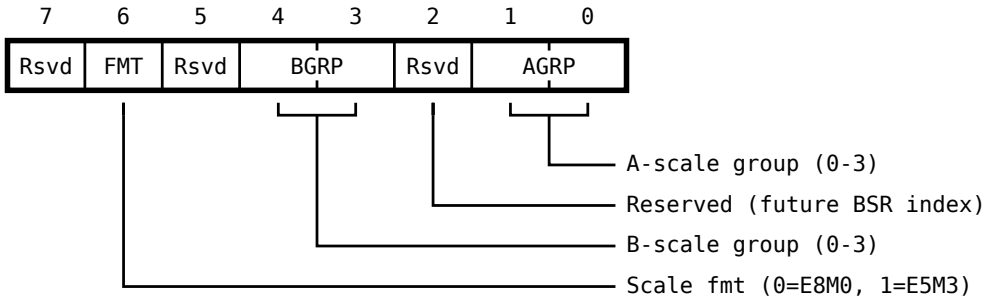


Figure 1: MX outer product imm8 encoding: scale group select and E8M0/E5M3 format select bits.

Operation

```
DEFINE top8fmxbf4ps(dst, src1, src2, imm8):
    // TOP8FMXBF4PS: MX FP4 E2M1 rank-8, FP32 accumulate, E8M0/E5M3 scaled.
    // imm8[1:0]=A-group, imm8[4:3]=B-group, imm8[6]=FMT(0=E8M0/1=E5M3), imm8[5]/[7]
    reserved.
    ASSERT VL in (512)
    a_group  = imm8[1:0]
    b_group  = imm8[4:3]
    fmt      = imm8[6]
    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:
        FOR j = 0 TO 15:
            dst.fp32[i][j] = op8mxbf4_subtile(
                dst.fp32[i][j],
                src1.dword[i], src1_scales[i], fmt,
                src2.dword[j], src2_scales[j], fmt)
```

Other Exceptions

Exception class ACE-E4. See Section 10.6.4.

8.2.2 TOP8FMXN[SS/SU/US/UU]PS — MX INT4 Rank-8 Outer Products, FP32 Accumulate

Rank-8 outer products of MX-scaled signed and/or unsigned INT4 inputs, accumulating into a 16×16 FP32 tile.

Block scaling via BSR. ACE v2 introduces E5M3 block scaling support for eligible outer product operations, see Section 8.4. Both A and B inputs use the same block scale format, selected by imm8 bit [6]: E8M0 (power-of-two, default) or E5M3 (non-power-of-two).

The integer sum of products is converted to FP32 using the specified OCP MX block scales. The scaling is done precisely and only one rounding event occurs on the convert to FP32. The FP32 scaled sum of products is accumulated with the FP32 tile destination.

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

Instruction Operand Encoding

Instruction	Operand 1	Operand 2	Operand 3	Operand 4
TOP8FMXNSSPS TOP8FMXNSUPS TOP8FMXNUSPS TOP8FMXNUUPS	MODRM.REG(rw)	MODRM.R/M(r)	VVVV(r)	IMM8(r)

Immediate encoding

Identical to TOP8FMXBF4PS (Section 8.2.1 table).

Implicit scale of the sum of products

This specification defines a uniform unity implicit bias ($2^0 = 1$) for all four MX INT4 sign variants:

Variant	A bias	B bias
SS / SU / US / UU	2^0	2^0

Table 8: MX INT4 implicit bias.

Operation

```
DEFINE top8fmxn_int4(dst, src1, src2, imm8, a_signed=True, b_signed=True):
    // TOP8FMXN[SS/SU/US/UU]PS: MX INT4 rank-8, FP32 accumulate, E8M0/E5M3 scaled.
    // imm8[1:0]=A-group, imm8[4:3]=B-group, imm8[6]=FMT(0=E8M0/1=E5M3), imm8[5]/[7]
    reserved.
    ASSERT VL in (512)
    a_group = imm8[1:0];  b_group = imm8[4:3]
    fmt     = imm8[6]
    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:
        FOR j = 0 TO 15:
            dst.fp32[i][j] = op8mxn_subtile(
                dst.fp32[i][j],
                src1.dword[i], src1_scales[i], fmt,
                src2.dword[j], src2_scales[j], fmt,
                a_signed, b_signed)
```

Other Exceptions

Exception class ACE-E4. See Section 10.6.4.

8.2.3 TOP4MXB[SU/US/UU]PS — MX INT8 Expanded Variants, FP32 Accumulate

ACE v1 defines only the signed×signed MX INT8 outer product (TOP4MXBSSPS). ACE v2 expands this with signed×unsigned, unsigned×signed, and unsigned×unsigned variants. All variants use the same implicit bias as the ACE v1 SS case (adjust = −12).

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

Instruction Operand Encoding

Instruction	Operand 1	Operand 2	Operand 3	Operand 4
TOP4MXBSUPS TOP4MXBUSPS TOP4MXBUUPS	MODRM.REG(rw)	MODRM.R/M(r)	VVVV(r)	IMM8(r)

Immediate encoding

`imm8[1:0]` = A-scale group; `imm8[4:3]` = B-scale group (same as ACE v1).

Implicit scale of the sum of products

Both signed INT8 (S8) and unsigned INT8 (U8) use the same OCP MX implicit bias of 2^{-6} per element. Signed INT8 is treated as SX.XXXXXX fixed-point. Unsigned INT8 is treated as XX.XXXXXX fixed-point.

Operation

```
DEFINE top4mxb_int8(dst, src1, src2, imm8, a_signed=True, b_signed=False):
    // TOP4MXB[SS/SU/US/UU]PS: MX INT8 rank-4, FP32 accumulate, E8M0 scaled.
    a_group = imm8[1:0]
    b_group = imm8[4:3]
    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:
        FOR j = 0 TO 15:
            a_elems = [int8_to_int32(src1.byte[4*i+k]) if a_signed else
            uint8_to_int32(src1.byte[4*i+k]) for k in range(4)]
            b_elems = [int8_to_int32(src2.byte[4*j+k]) if b_signed else
            uint8_to_int32(src2.byte[4*j+k]) for k in range(4)]
            acc = sum(a_elems[k] * b_elems[k] for k in range(4))
            adjust = _mxint_adjust(8, a_signed, b_signed)
            sop_fp32 = convert_fixpoint32_to_fp32_mx_scaled(
                acc, src1_scales[i], src2_scales[j], a_fmt=0, b_fmt=0, adjust=adjust)
            // DAZ=1 on accumulator; FTZ=1 on output
            dst.fp32[i][j] = float32_add(dst.fp32[i][j], sop_fp32, daz_a=1, daz_b=0, ftz=1)
```

Other Exceptions

Exception class ACE-E4. See Section 10.6.4.

8.3 FP16 Outer Products

These instructions introduce FP16 floating-point inputs, complementing the ACE v1 BF16 outer product.

No block scale is applied; accumulation is FP32.

8.3.1 TOP2PHPS — FP16 Rank-2 Outer Product, FP32 Accumulate

Rank-2 outer product of FP16 inputs accumulating into a 16×16 FP32 tile.

Two FP16 elements are packed per 32-bit dword lane.

Each outer product call contributes 2 multiply-add terms at each of the 256 tile positions.

MXCSR is neither consulted nor updated.

Operand 1	Operand 2	Operand 3
tmm1	zmm2	zmm3

Instruction Operand Encoding

Instruction	Operand 1	Operand 2	Operand 3
TOP2PHPS	MODRM.REG(rw)	MODRM.R/M(r)	VVVV(r)

Operation

FP16 → FP32 conversion: Each FP16 input is widened to FP32 exactly. FP16 subnormals are preserved as FP32 normal numbers — they are not flushed to zero.

Products: $p_0 = a_0 \times b_0$ and $p_1 = a_1 \times b_1$, each computed in FP32.

Sum and accumulate (both FP32, DAZ=FTZ=1, RNE):

- **Rounding event 1:** $\mathrm{tmp} = p_0 + p_1$, FP32 RNE.
- **Rounding event 2:** $\mathrm{dst}[i][j] += \mathrm{tmp}$, FP32 RNE.

```
DEFINE top2phps(dst, src1, src2):
    // TOP2PHPS: FP16 rank-2 outer product, FP32 accumulate.
    ASSERT VL in (512)
    FOR i = 0 TO dst.rows-1:
        FOR j = 0 TO 15:
            // Exact FP16 → FP32 widening; subnormals become FP32 normals
            a0 = fp16_to_fp32_exact(src1.fp16[2*i])
            a1 = fp16_to_fp32_exact(src1.fp16[2*i+1])
            b0 = fp16_to_fp32_exact(src2.fp16[2*j])
            b1 = fp16_to_fp32_exact(src2.fp16[2*j+1])
            // Products exact in FP32 (mantissa product ≤ 20 bits fits in 23-bit FP32)
            p0 = a0 * b0  // FP32 multiply, exact result
            p1 = a1 * b1  // FP32 multiply, exact result
            // Rounding event 1: sum of products; FTZ=1 on SoP output
            tmp = float32_add(p0, p1, daz_a=0, daz_b=0, ftz=1)
            // Rounding event 2: accumulate; DAZ=1 on tile acc; FTZ=1 on output
            dst.fp32[i][j] = float32_add(dst.fp32[i][j], tmp, daz_a=1, daz_b=0, ftz=1)
```

Other Exceptions

Exception class ACE-E4. See Section 10.6.4.

8.4 Block Scale Format Enhancements

ACE v2 retains the Block Scale Register (BSR) from ACE v1:

- A-scales (BSR[1023:512])
- B-scales (BSR[511:0])

Each half is divided into 4 groups of 16 scale bytes interleaved by element index.

ACE v2 introduces an E5M3 block scale option to eligible outer products. For MX FP4 (Section 8.2.1) and MX INT4 (Section 8.2.2) outer products, the block scale interpretation for both A and B inputs is selected via imm8 bit [6]:

imm8 bit	0	1
[6]	E8M0	E5M3

The BSR storage format and BSRMOVH/BSRMOVL load/store instructions are unchanged. The same byte value in the BSR is interpreted as either E8M0 or E5M3 depending on the imm8 bit. It is the programmer's responsibility to load scale bytes consistent with the format selected by each outer product call.

8.5 Instruction Encodings

8.5.1 MX FP4 Rank-8 Outer Products

Mnemonic	Form	Encoding
TOP8FMXBF4PS	tmm1, zmm2, zmm3, imm8	EVEX.512.F3.0F3A.W0 8C

8.5.2 MX INT4 Rank-8 Outer Products

Mnemonic	Form	Encoding
TOP8FMXNSSPS	tmm1, zmm2, zmm3, imm8	EVEX.512.F2.0F3A.W0 8E
TOP8FMXNSUPS	tmm1, zmm2, zmm3, imm8	EVEX.512.F3.0F3A.W0 8E
TOP8FMXNUSPS	tmm1, zmm2, zmm3, imm8	EVEX.512.66.0F3A.W0 8E
TOP8FMXNUUPS	tmm1, zmm2, zmm3, imm8	EVEX.512.NP.0F3A.W0 8E

8.5.3 MX INT8 Rank-4 Outer Products (Sign Extensions)

Mnemonic	Form	Encoding
TOP4MXBSUPS	tmm1, zmm2, zmm3, imm8	EVEX.512.F3.0F3A.W0 8F
TOP4MXBUSPS	tmm1, zmm2, zmm3, imm8	EVEX.512.66.0F3A.W0 8F
TOP4MXBUUPS	tmm1, zmm2, zmm3, imm8	EVEX.512.NP.0F3A.W0 8F

8.5.4 FP16 Rank-2 Outer Product

Mnemonic	Form	Encoding
TOP2PHPS	tmm1, zmm2, zmm3	EVEX.512.F2.0F38.W0 5C

9 ACE v2 Single and Double Precision Outer Products

9.1 CPUID Enumeration

- **ACE_OP_FP32** (EAX:1DH, ECX:2, EAX[17]) — TOP1PS
- **ACE_OP_FP64** (EAX:1DH, ECX:2, EAX[18]) — TOP1PD

These instructions are optional and independently enumerated. They are not implied by ACE_VSN ≥ 2. See Table 7.

These instructions target single and double precision floating-point workloads. They use the same tile register file as existing ACE outer products. FP32 accumulates into FP32; FP64 accumulates into FP64. The FP64 outer product reinterprets the tile register as 16 rows × 8 columns of FP64 accumulators.

9.2 TOP1PS — FP32 Rank-1 Outer Product, FP32 Accumulate

Rank-1 outer product of FP32 inputs accumulating into a 16×16 FP32 tile. Each element of src1 (ZMM) provides one FP32 value per tile row; each element of src2 provides one FP32 value per tile column. The outer product $\mathrm{dst}[i][j] += \mathrm{src1}[i] \times \mathrm{src2}[j]$ is computed for all (i, j).

Operand 1	Operand 2	Operand 3
tmm1	zmm2	zmm3

Instruction Operand Encoding

Instruction	Operand 1	Operand 2	Operand 3
TOP1PS	MODRM.REG(rw)	MODRM.R/M(r)	VVVV(r)

MXCSR Interaction

See Section 9.4.

Fused Multiply-Accumulate

Each element-wise accumulation uses IEEE 754-2008 `fusedMultiplyAdd` semantics: the intermediate product $\mathrm{src1}[i] \times \mathrm{src2}[j]$ is computed in extended precision and is not independently rounded before addition to the accumulator. A single rounding event produces the final FP32 result per element per instruction.

Operation

```
DEFINE top1ps(dst, src1, src2, evex_b=False, evex_rc=0):
    // TOP1PS: FP32 rank-1 outer product, FP32 accumulate.
    ASSERT VL in (512)
    FOR i = 0 TO dst.rows-1:
        FOR j = 0 TO 15:
            dst.fp32[i][j] = float32_mad(
                dst.fp32[i][j], src1.fp32[i], src2.fp32[j],
                mxcsr_rc=(evex_rc if evex_b else MXCSR.RC),
                mxcsr_daz=MXCSR.DAZ, mxcsr_ftz=MXCSR.FTZ)
```

Other Exceptions

Exception class ACE-E7. See Section 10.6.7.

9.3 TOP1PD — FP64 Rank-1 Outer Product, Full Tile

Rank-1 outer product of FP64 inputs accumulating into the full 16×8 FP64 tile register. The A-source is specified by an even-aligned ZMM register pair: `zmm_base` (rows 0–7) and `zmm_base+1` (rows 8–15). #UD if the base register number is odd. The B-source is a single ZMM column vector (8 FP64 elements).

The outer product $\mathrm{dst}[i][j] += \mathrm{src1}[i] \times \mathrm{src2}[j]$ is computed for all $(i, j) \in [0, 15] \times [0, 7]$, where `src1[i]` is drawn from `zmm_base` for $i \in [0, 7]$ and from `zmm_base+1` for $i \in [8, 15]$.

One FP64 element occupies one 64-bit qword lane; each ZMM provides 8 FP64 elements. The ZMM pair provides 16 FP64 elements, one per tile row.

The tile register holds 16 rows × 64 bytes. When interpreted as FP64, this accommodates 16 rows × 8 columns of FP64.

Operand 1	Operand 2	Operand 3
tmm1	zmm2{+1}	zmm3

Instruction Operand Encoding

Instruction	Operand 1	Operand 2	Operand 3
TOP1PD	MODRM.REG(rw)	MODRM.R/M(r)	VVVV(r)

MXCSR Interaction

See Section 9.4.

Fused Multiply-Accumulate

Each element-wise accumulation uses IEEE 754-2008 `fusedMultiplyAdd` semantics: the intermediate product $\mathrm{src1}[i] \times \mathrm{src2}[j]$ is computed in extended precision and is not independently rounded before addition to the accumulator. A single rounding event produces the final FP64 result per element per instruction.

Operation

```
DEFINE top1pd(dst, src1_lo, src1_hi, src2, evex_b=False, evex_rc=0):
    // TOP1PD: FP64 rank-1 outer product, full tile (16r × 8c). ZMM-pair A-source.
    ASSERT VL in (512)
    FOR i = 0 TO 15:
        a_val = src1_lo.fp64[i] if i < 8 else src1_hi.fp64[i - 8]
        FOR j = 0 TO 7:
            dst.fp64[i][j] = float64_mad(
                dst.fp64[i][j], a_val, src2.fp64[j],
                mxcsr_rc=(evex_rc if evex_b else MXCSR.RC),
                mxcsr_daz=MXCSR.DAZ, mxcsr_ftz=MXCSR.FTZ)
```

Other Exceptions

Exception class ACE-E8. See Section 10.6.8.

9.4 MXCSR Interaction

TOP1PS and TOP1PD are IEEE 754-conforming instructions. MXCSR is consulted for rounding mode (RC), subnormal input handling (DAZ), and subnormal output handling (FTZ). ZM is not consulted (no division is performed).

Each instruction comprises many element-wise fused multiply-accumulate (FMA) operations: 256 for TOP1PS (16 × 16 FP32) and 128 for TOP1PD (16 × 8 FP64). Each FMA independently evaluates IEEE 754 exception conditions. Different elements may encounter different conditions — for example, one element position may produce an inexact result while another encounters a denormal input.

The MXCSR exception status flags (IE, DE, OE, UE, PE) accumulate via sticky OR: each flag is ORed with the conditions encountered across all element-wise FMA operations. A condition occurring in any single element is sufficient to set the corresponding flag.

9.4.1 EVEX Embedded Rounding

When EVEX.b = 1, the LL field encodes the rounding control directly (00 = RN, 01 = RD, 10 = RU, 11 = RZ), overriding MXCSR.RC. Suppress-all- exceptions (SAE) is implied: exception status flags are not written to MXCSR, and #XM is not raised regardless of mask bit settings. DAZ and FTZ are still read from MXCSR.

MXCSR bit	Read	Written	Notes
RC (14:13)	Yes	No	Rounding mode for all MACs; overridden by EVEX.LL when EVEX.b = 1
PM (12)	Yes	No	Inexact mask — see Floating-Point Exceptions
UM (11)	Yes	No	Underflow mask
OM (10)	Yes	No	Overflow mask
ZM (9)	No	No	Zero-divide — not applicable
DM (8)	Yes	No	Denormal input mask
IM (7)	Yes	No	Invalid operation mask
DAZ (6)	Yes	No	Flush subnormal inputs to ±0
PE (5)	No	Yes	Inexact flag; not written when EVEX.b = 1 (SAE)
UE (4)	No	Yes	Underflow flag; not written when EVEX.b = 1 (SAE)
OE (3)	No	Yes	Overflow flag; not written when EVEX.b = 1 (SAE)
ZE (2)	No	No	Zero-divide — never set
DE (1)	No	Yes	Denormal input consumed (when DAZ = 0); not written when EVEX.b = 1 (SAE)
IE (0)	No	Yes	Invalid operation (NaN input, ∞×0, ∞−∞); not written when EVEX.b = 1 (SAE)
FTZ (15)	Yes	No	Flush subnormal results to ±0

Table 9: MXCSR bit usage for TOP1PS and TOP1PD. When EVEX.b = 1, RC is from EVEX.LL and exception flags (PE, UE, OE, DE, IE) are not written.

9.4.2 Exception Delivery

If any MXCSR exception status flag is set by the instruction and the corresponding mask bit is clear (unmasked), #XM is raised. The exception is a fault: the saved instruction pointer identifies the faulting instruction and the destination tile register is not updated. MXCSR status flags are updated prior to exception delivery. The exception handler may inspect MXCSR, correct the condition, and return to retry the instruction.

When EVEX.b = 1 (SAE), #XM is never raised regardless of mask bit settings.

9.4.3 Software Guidance

Software should keep PM masked (PM = 1) or use embedded rounding (EVEX.b = 1) which suppresses all floating-point exceptions.

9.5 Instruction Encodings

Mnemonic	Form	Encoding
TOP1PS	tmm1, zmm2, zmm3 {er}	EVEX.512.NP.0F38.W0 48
TOP1PD	tmm1, zmm2{+1}, zmm3 {er}	EVEX.512.F2.0F38.W1 48

10 Exception Classes

10.1 Exception Classes by Instruction

The following table lists exception classes used by instructions defined in this specification. ACE-E1 through ACE-E3, ACE-E5, and ACE-E6 are inherited from ACE v1 and reproduced here for the APX cross-reference in Appendix C. ACE-E4, ACE-E7, and ACE-E8 are new to this specification. See also the ACE v1 Specification [1].

Class	Description
AMX-E1	Tile configuration load — LDTILECFG (VEX)
AMX-E2	Tile configuration store — STTILECFG (VEX)
AMX-E1-EVEX	Tile configuration load — LDTILECFG (EVEX/APX)
AMX-E2-EVEX	Tile configuration store — STTILECFG (EVEX/APX)
E2	FP8 to BF16 widening converts (VCVTBF82BF16, VCVTHF82BF16)
E4	Bit reversal (VBITREV)
E4NF	BF16 / FP16 narrowing converts to FP8, FP4, and INT4 Biased BF16 / FP16 narrowing converts BF16 to UF8 converts Two-source narrowing converts (VCVT2*) Byte and word interleave (VPINTERLEAVELB, VPINTERLEAVEHB, VPINTERLEAVELW, VPINTERLEAVEHW) Bit matrix multiply (VBMACXOR, VBMACOR)
E6	Saturation narrowing store (VPMOVSSDB — memory form). See also ACE v1 [1].
ACE-E4	Outer products without MXCSR interaction: MX FP4, MX INT4, MX INT8, FP16 (TOP8FMXBF4PS, TOP8FMXN*, TOP4MXBSUPS/USPS/UUPS, TOP2PHPS)
ACE-E7	FP32 outer product with MXCSR/ER interaction (TOP1PS)
ACE-E8	FP64 outer product with MXCSR/ER interaction and multi-register alignment (TOP1PD)

10.2 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 10.8.
SIMD Floating-Point Exception, #XM	X	X	X	X	If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1.

10.3 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 10.8.

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 10.8.

10.4 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 10.8.

10.5 Exceptions Type AMX

10.5.1 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.

10.5.2 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.

10.5.3 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.

10.5.4 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.

10.5.5 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

10.5.6 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

10.6 Exceptions Type ACE

10.6.1 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

10.6.2 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 10.7)
- #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 10.8.

10.6.3 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 10.7)
- #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 10.8.

10.6.4 ACE-E4

ACE outer product instructions (exception-free; no MXCSR interaction).

- #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

10.6.5 ACE-E5

ACE BSR Init instruction (operandless).[3]

- #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

[3] 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

10.6.6 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 10.7)
- #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

10.6.7 ACE-E7

ACE outer product instructions with IEEE 754 floating-point semantics and MXCSR/embedded-rounding interaction (TOP1PS).

- #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 != 0b0
- #UD if EVEX.L'L != 0b10 (when EVEX.b = 0)
- #UD if EVEX.aaa != 0b000
- #XM if EVEX.b = 0, an unmasked SIMD floating-point exception occurs, and CR4.OSXMMEXCPT[bit 10] = 1.
- #NM if CR0.TS[3]=1
- #NM if IA32_XFD[18] == 1

10.6.8 ACE-E8

ACE outer product instructions with IEEE 754 floating-point semantics, MXCSR/embedded-rounding interaction, and multi-operand register alignment (TOP1PD).

ACE-E8 extends ACE-E7 with one additional constraint on multi-register operands:

- All conditions from ACE-E7 (Section 10.6.7) apply.
- #UD if the base register number of a multi-register operand is odd.

10.7 #UD Equations for EVEX encoded instructions

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

10.7.1 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 10: 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).

10.8 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

Appendix A — Data Formats

See also ACE v1 Specification [1] for FP8/FP6 format details.

Floating-Point Element Formats

Format	Abbrev	Bits	Bias	Max normal	Special values
FP4 E2M1	FP4	1s + 2e + 1m	1	±6.0	No Inf, no NaN; −0 encodable
FP8 E5M2	BF8	1s + 5e + 2m	15	57344	E=0x1F, M≠0 → NaN; E=0x1F, M=0 → ±Inf
FP8 E4M3	HF8	1s + 4e + 3m	7	448	E=0xF, M=0x7 → NaN; no Inf
FP16	PH	1s + 5e + 10m	15	65504	±Inf; NaN (IEEE 754)
BF16	BF16	1s + 8e + 7m	127	$\approx 3.4 \times 10^{38}$	±Inf; NaN (IEEE 754)
FP32	PS	1s + 8e + 23m	127	$\approx 3.4 \times 10^{38}$	±Inf; NaN (IEEE 754)
FP64	PD	1s + 11e + 52m	1023	$\approx 1.8 \times 10^{308}$	±Inf; NaN (IEEE 754)

Table 11: Floating-point element formats supported by AVX10.3 and ACE v2.

Integer and Fixed-Point Element Formats

ACE outer products use integer element types scaled by OCP MX block scales. The implicit bias (adjust) determines how the integer product maps to the FP32 accumulator.

10.8.1 MX INT8

MX INT8 elements are 8-bit signed (S8) or unsigned (U8) values used by the rank-4 MX outer product family (TOP4MXB*PS).

Both signed and unsigned forms use the same implicit bias of 2^{-6} per element (adjust = −12 for the rank-4 product of two elements).

Format	Abbrev	Bits	Range	Implicit bias (per element)
Signed INT8	S8	8-bit two’s complement (SX.XXXXXX)	[−2.0, +1.984375]	2^{-6}
Unsigned INT8	U8	8-bit unsigned (XX.XXXXXX)	[0, +3.984375]	2^{-6}

Table 12: MX INT8 element formats. Both sign variants share the same implicit bias.

10.8.2 MX INT4

MX INT4 elements are 4-bit signed (S4) or unsigned (U4) nibbles used by the rank-8 MX outer product family (TOP8FMXN*PS).

All INT4 sign variants use a unity implicit bias ($2^0 = 1$; adjust = 0).

Format	Abbrev	Bits	Range	Implicit bias (per element)
Signed INT4	S4 / SNS	4-bit two’s complement	[−8, +7]	2^0 (unity)
Unsigned INT4	U4 / UNS	4-bit unsigned	[0, +15]	2^0 (unity)

Table 13: MX INT4 element formats. Unity bias — no implicit scaling.

Nibble packing: two nibbles per byte, little-endian; nibble 0 in bits[3:0], nibble 1 in bits[7:4].

OCP MX Block Scale Formats

E8M0 is an unsigned 8-bit block scale format (no sign bit) defined by the OCP Microscaling Formats specification [4]. E5M3 is proposed for a future version of the OCP Microscaling Formats specification [4].

The byte value 0xFF is the NaN sentinel for both formats.

The byte 0x00 has different semantics: for E8M0 it encodes 2^{-127} (the minimum power of two), while for E5M3 it encodes exact 0.0 and suppresses the corresponding row or column’s contribution to the outer product entirely.

Property	E8M0	E5M3
Width	8 bits	8 bits
Sign	None (unsigned)	None (unsigned)
Structure	`{E8[7:0]}`	`{E5[7:3], M3[2:0]}`
Bias	127	15
NaN encoding	0xFF `{11111111}`	0xFF `{11111,111}`
Zero / minimum	0x00 = 2^{-127} (not zero)	0x00 `{00000,000}` = exact 0.0
Max normal	0xFE = 2^{127}	0xFE `{11111,110}` = $2^{16} \times 1.75$ = **114,688**
Min normal	0x01 = 2^{-126}	0x08 `{00001,000}` = $2^{1-15} = 2^{-14}$
Max subnormal	None	0x07 `{00000,111}` = $2^{-14} \times \frac{7}{8} = 7 \times 2^{-17}$
Min subnormal	None	0x01 `{00000,001}` = 2^{-17}
Range	$[2^{-127}, 2^{127}]$	[0.0, 114,688]
Representable values	Powers of two only	Non-power-of-two values possible

Table 14: OCP MX block scale format comparison: E8M0 (power-of-two only) vs E5M3 (non-power-of-two capable).

E5M3 byte layout:

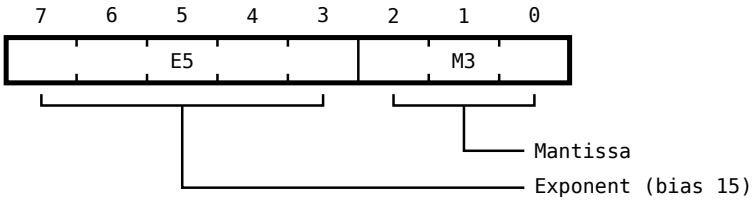


Figure 2: E5M3 byte format: 5-bit exponent (bias 15) and 3-bit mantissa. NaN = 0xFF, zero = 0x00.

E8M0 byte layout:

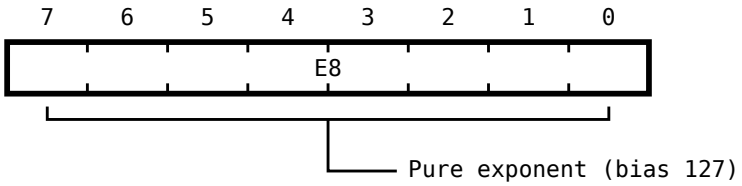


Figure 3: E8M0 byte format: 8-bit pure exponent (bias 127), no mantissa. NaN = 0xFF, min = 0x00 = 2^(-127).

Mnemonic Naming Conventions

Suffix/Prefix	Meaning
B / BF	Brain Float (BF8-E5M2 or BF16)
H / HF	Half Float (HF8-E4M3)
P / PH	Packed half-precision (FP16)
PS	Packed single (FP32)
PD	Packed double (FP64)
S / SNS	Signed (with saturation)
U / UNS	Unsigned (with saturation)
S suffix	Saturate on overflow (mandatory)
[S] suffix	Saturation optional (base form = no sat)
BIAS prefix	Biased rounding (INT8 bias operand added to mantissa before truncation)
2 prefix	Two-source double-density form
V prefix (shifts)	Variable per-element count

Table 15: Mnemonic naming conventions for AVX10.3 and ACE v2 instructions.

Rounding Modes

Mode	Abbrev	Description
Round to Nearest Even	RNE	Default IEEE 754 rounding
Round to Odd	RTO	Preserves sticky bit for subsequent rounding
Biased rounding	BIAS	Adds integer bias to mantissa before truncation; stochastic when bias is pseudo-random

Table 16: Rounding modes used by AVX10.3 and ACE v2 data conversion instructions. MXCSR is not consulted by any AVX10.3 or ACE v2 instruction unless explicitly noted.

Appendix B — Pseudocode Helper Functions

This appendix collects the pseudocode helper functions referenced throughout the instruction descriptions. The implementations are the authoritative source of truth for the functions described informally in prose.

Helper functions for this specification reference the ACE v1 pseudocode helpers (Appendix B of [1]) plus the following new helpers.

FP4 and INT4 Element Conversion

Widening (FP4/INT4 → FP32/INT32):

```
DEFINE fp4_e2m1_to_fp32(nibble):
    // FP4 E2M1 nibble → FP32. Format: 1s + 2e + 1m, bias=1.
    // 16-entry exact LUT.
    // (see spec §4.1.2 LUT table).
    lut = [0.0, 0.5, 1.0, 1.5, 2.0, 3.0, 4.0, 6.0,
           -0.0,-0.5,-1.0,-1.5,-2.0,-3.0,-4.0,-6.0]
    RETURN lut[nibble & 0xF]
```

```
DEFINE int4_s_to_int32(nibble):
    // Sign-extend 4-bit nibble to int32.
    v = nibble & 0xF
    RETURN v - 16 if v >= 8 else v
```

```
DEFINE uint4_to_int32(nibble):
    // Zero-extend 4-bit nibble to int32.
    RETURN nibble & 0xF
```

Narrowing (FP16/BF16 → FP4/INT4):

```
DEFINE fp16_to_fp4(fp16_val):
    // FP16 → FP4 E2M1. Saturating RNE rounding to nearest of 8 positive magnitudes.
    IF IsNaN(fp16_val):
        RETURN 0x8  // -0: NaN sentinel for FP4 E2M1
    mags = [0.0, 0.5, 1.0, 1.5, 2.0, 3.0, 4.0, 6.0]
    sign = 1 if fp16_val < 0.0 else 0
    abs_val = ABS(fp16_val)
    IF IsInf(abs_val) OR abs_val > 6.0:
        abs_val = 6.0  // saturate Inf and overflow to max magnitude
    best_idx = 0
    best_dist = ABS(mags[0] - abs_val)
    FOR idx = 1 TO 7:
        dist = ABS(mags[idx] - abs_val)
        IF dist < best_dist OR (dist == best_dist AND (idx & 1) == 0):
            best_dist = dist
            best_idx  = idx
    RETURN (sign << 3) | best_idx
```

```
DEFINE fp16_to_int4(fp16_val, signed=True):
    // FP16 → INT4. Saturating. signed=True: S4 range [−8, 7]; signed=False: U4 [0, 15].
    IF IsNaN(fp16_val):
        RETURN 0x0  // NaN → zero (block scale carries NaN)
    IF IsInf(fp16_val):
        IF signed:
            RETURN (0x8 if fp16_val < 0 else 0x7) & 0xF  // -8 or +7
        ELSE:
            RETURN 0x0 if fp16_val < 0 else 0xF  // 0 or 15
    v = round(fp16_val)
    IF signed:
        v = max(-8, min(7, v))
    ELSE:
```

```
        v = max(0, min(15, v))
    RETURN v & 0xF
```

```
DEFINE bias_round_fp16_to_bf4(bias_byte, fp16_val):
    // Biased rounding: add INT8 bias to mantissa bits of fp16_val, then truncate to FP4.
    IF IsNaN(fp16_val):
        RETURN 0x8  // -0: NaN sentinel for FP4 E2M1
    raw_bits = BitCast_FP16_to_UINT16(fp16_val)
    sign = raw_bits[15]
    exp  = (raw_bits >> 10) & 0x1F
    mantissa_raw = raw_bits & 0x3FF
    IF exp == 0x1F:  // Inf → saturate to max magnitude
        RETURN (sign << 3) | 0x7  // ±6.0
    IF exp == 0 AND mantissa_raw == 0:
        RETURN (sign << 3)  // ±0
    biased_mant = (mantissa_raw >> 7) + bias_byte
    mag_idx = min(7, max(0, ((exp - 14) << 1) | (biased_mant[7])))
    RETURN (sign << 3) | mag_idx
```

`bf16_to_fp4` and `bf16_to_int4` convert BF16→FP16 and delegate to the FP16 variants above.

FP8 ↔ BF16 Element Helpers

Per-element scalar conversions underlying the SIMD wrappers above.

```
DEFINE fp8_e5m2_to_bf16(fp8_byte):
    // BF8 E5M2 → BF16. Exact widening. Format: 1s+5e+2m, bias=15.
    // Special: E=0x1F,M≠0 → QNaN; E=0x1F,M=0 → ±Inf.
    sign  = fp8_byte[7]
    exp5  = (fp8_byte >> 2) & 0x1F
    mant2 =  fp8_byte       & 0x03
    IF exp5 == 0x1F:
        IF mant2 != 0:
            RETURN (sign << 15) | 0x7FC0  // NaN → BF16 QNaN
        RETURN (sign << 15) | 0x7F80  // ±Inf → BF16 ±Inf
    IF exp5 == 0:
        IF mant2 == 0:
            RETURN sign << 15  // ±0
        IF mant2 & 0x2:  // leading 1 at bit 1 (mant2 = 10 or 11)
            exp8  = 112  // 127 - 14 - 1
            mant7 = (mant2 & 0x1) << 6
        ELSE:  // mant2 = 01: leading 1 at bit 0
            exp8  = 111  // 127 - 14 - 2
            mant7 = 0
    ELSE:
        exp8  = exp5 + 112  // bias adjust: 127 - 15 = 112
        mant7 = mant2 << 5
    RETURN (sign << 15) | (exp8 << 7) | mant7
```

```
DEFINE fp8_e4m3_to_bf16(fp8_byte):
    // HF8 E4M3 → BF16. Exact widening. Format: 1s+4e+3m, bias=7.
    // Special: E=0xF,M=0x7 → QNaN only (no ±Inf; E=0xF,M<7 are valid normals).
    sign  = fp8_byte[7]
    exp4  = (fp8_byte >> 3) & 0xF
    mant3 =  fp8_byte       & 0x7
    IF exp4 == 0xF AND mant3 == 0x7:  // sole NaN encoding
        RETURN (sign << 15) | 0x7FC0  // BF16 QNaN
    IF exp4 == 0:
        IF mant3 == 0:
            RETURN sign << 15  // ±0
        IF   mant3 & 0x4:  // leading 1 position in mant3
            p = 2
```

```
        ELSE IF mant3 & 0x2:
            p = 1
        ELSE:
            p = 0
        exp8  = 118 + p  // 127 - 9 + p  (subnormal base = 2^(1-7) = 2^-6)
        mant7 = (mant3 & ~(1 << p)) << (7 - p)
    ELSE:
        exp8  = exp4 + 120  // bias adjust: 127 - 7 = 120
        mant7 = mant3 << 4  // 3-bit mantissa → top 3 of 7
    RETURN (sign << 15) | (exp8 << 7) | mant7
```

```
DEFINE bf16_to_fp8_e5m2(bf16_val, rounding="RNE", saturate=False):
    // BF16 → FP8 E5M2 (BF8). Bias adjust: 127-15=112. Rounds mant7 → mant2.
    // saturate=True: overflow → max_normal (E=30,M=3 = 0x7B) instead of ±Inf.
    MAX_NORMAL = 0x7B  // E=30, M=3 = 57344
    sign  = bf16_val[15]
    exp8  = (bf16_val >>  7) & 0xFF
    mant7 =  bf16_val        & 0x7F
    IF exp8 == 0xFF:
        IF mant7 != 0:
            RETURN (sign << 7) | 0x7E  // NaN → canonical E5M2 NaN (E=0x1F,M=0b10)
        RETURN (sign << 7) | (MAX_NORMAL if saturate else 0x7C)
    IF exp8 == 0:
        RETURN sign << 7  // BF16 subnormals underflow E5M2 range
    exp5 = exp8 - 112
    IF exp5 >= 31:
        RETURN (sign << 7) | (MAX_NORMAL if saturate else 0x7C)
    IF exp5 >= 1:  // normal E5M2 result
        mant2  = mant7 >> 5
        guard  = mant7[4]
        sticky = 1 if (mant7 & 0xF) else 0
    ELSE:  // subnormal E5M2 result
        shift   = 1 - exp5
        full    = (1 << 7) | mant7  // 1.mant7 as 8-bit integer
        s_total = 5 + shift
        IF s_total >= 9:
            RETURN sign << 7  // underflow to ±0
        mant2  = full >> s_total
        guard  = (full >> (s_total - 1)) & 0x1
        sticky = 1 if (full & ((1 << (s_total - 1)) - 1)) else 0
        exp5   = 0
    IF _fp8_round_up(mant2, guard, sticky, rounding):
        mant2 += 1
    IF mant2 >> 2:  // carry out of mantissa
        mant2  = 0
        exp5  += 1
        IF exp5 >= 31:
            RETURN (sign << 7) | (MAX_NORMAL if saturate else 0x7C)
    RETURN (sign << 7) | ((exp5 & 0x1F) << 2) | (mant2 & 0x3)
```

```
DEFINE bf16_to_fp8_e4m3(bf16_val, rounding="RNE", saturate=False):
    // BF16 → FP8 E4M3 (HF8). Bias adjust: 127-7=120. Rounds mant7 → mant3.
    // No ±Inf in E4M3. saturate=True: overflow → max_normal (0x7E = 448).
    MAX_NORMAL = 0x7E  // E=15, M=6 = 448
    NAN_E4M3   = 0x7F  // sole NaN encoding: E=15, M=7
    sign  = bf16_val[15]
    exp8  = (bf16_val >>  7) & 0xFF
    mant7 =  bf16_val        & 0x7F
    IF exp8 == 0xFF:
        IF mant7 != 0:
```

```
            RETURN (sign << 7) | NAN_E4M3
        RETURN (sign << 7) | (MAX_NORMAL if saturate else NAN_E4M3)
    IF exp8 == 0:
        RETURN sign << 7
    exp4 = exp8 - 120
    IF exp4 > 15:
        RETURN (sign << 7) | (MAX_NORMAL if saturate else NAN_E4M3)
    IF exp4 >= 1:  // normal E4M3 result
        mant3   = mant7 >> 4
        guard   = mant7[3]
        sticky  = 1 if (mant7 & 0x7) else 0
        exp_out = exp4
    ELSE:  // subnormal E4M3 result
        shift   = 1 - exp4
        full    = (1 << 7) | mant7
        s_total = 4 + shift
        IF s_total >= 9:
            RETURN sign << 7
        mant3   = full >> s_total
        guard   = (full >> (s_total - 1)) & 0x1
        sticky  = 1 if (full & ((1 << (s_total - 1)) - 1)) else 0
        exp_out = 0
    IF _fp8_round_up(mant3, guard, sticky, rounding):
        mant3 += 1
    IF mant3 >> 3:  // carry out of mantissa
        mant3   = 0
        exp_out += 1
    IF exp_out > 15 OR (exp_out == 15 AND mant3 == 7):  // overflow or NaN encoding
        RETURN (sign << 7) | (MAX_NORMAL if saturate else NAN_E4M3)
    RETURN (sign << 7) | ((exp_out & 0xF) << 3) | (mant3 & 0x7)
```

```
DEFINE bf16_to_fp8_e5m2_biased(bias_byte, bf16_val, saturate=False):
    // BF16 → FP8 E5M2 with biased rounding. bias_byte (signed INT8) is added
    // to the 5 sub-mantissa bits before truncation; carry propagates into mant2.
    MAX_NORMAL = 0x7B
    sign  = bf16_val[15]
    exp8  = (bf16_val >>  7) & 0xFF
    mant7 =  bf16_val        & 0x7F
    IF exp8 == 0xFF:
        IF mant7 != 0:
            RETURN (sign << 7) | 0x7E
        RETURN (sign << 7) | (MAX_NORMAL if saturate else 0x7C)
    IF exp8 == 0:
        RETURN sign << 7
    exp5 = exp8 - 112
    IF exp5 >= 31:
        RETURN (sign << 7) | (MAX_NORMAL if saturate else 0x7C)
    IF exp5 >= 1:
        biased = (mant7 & 0x1F) + bias_byte  // bias_byte is signed INT8
        mant2  = (mant7 >> 5) + (biased >> 5)  // carry from sub-mantissa field
        IF mant2 >> 2:
            mant2  = 0
            exp5  += 1
            IF exp5 >= 31:
                RETURN (sign << 7) | (MAX_NORMAL if saturate else 0x7C)
        RETURN (sign << 7) | ((exp5 & 0x1F) << 2) | (mant2 & 0x3)
    RETURN sign << 7  // subnormal: biased path → RTZ to ±0
```

```
DEFINE bf16_to_fp8_e4m3_biased(bias_byte, bf16_val, saturate=False):
    // BF16 → FP8 E4M3 with biased rounding. bias_byte (signed INT8) is added
```

```
    // to the 4 sub-mantissa bits before truncation; carry propagates into mant3.
    MAX_NORMAL = 0x7E
    NAN_E4M3   = 0x7F
    sign  = bf16_val[15]
    exp8  = (bf16_val >>  7) & 0xFF
    mant7 =  bf16_val        & 0x7F
    IF exp8 == 0xFF:
        IF mant7 != 0:
            RETURN (sign << 7) | NAN_E4M3
        RETURN (sign << 7) | (MAX_NORMAL if saturate else NAN_E4M3)
    IF exp8 == 0:
        RETURN sign << 7
    exp4 = exp8 - 120
    IF exp4 > 15:
        RETURN (sign << 7) | (MAX_NORMAL if saturate else NAN_E4M3)
    IF exp4 >= 1:
        biased = (mant7 & 0xF) + bias_byte  // bias_byte is signed INT8
        mant3  = (mant7 >> 4) + (biased >> 4)  // carry from sub-mantissa field
        IF mant3 >> 3:
            mant3  = 0
            exp4  += 1
        IF exp4 > 15 OR (exp4 == 15 AND mant3 == 7):
            RETURN (sign << 7) | (MAX_NORMAL if saturate else NAN_E4M3)
        RETURN (sign << 7) | ((exp4 & 0xF) << 3) | (mant3 & 0x7)
    RETURN sign << 7  // subnormal: biased path → RTZ to ±0
```

SIMD wrappers (dispatcher layer):

```
DEFINE convert_fp8_to_bf16(fp8_byte, fp8_type="bf8"):
    // FP8 E5M2 (BF8) or E4M3 (HF8) → BF16. Exact, no rounding.
    IF fp8_type == "bf8":
        RETURN fp8_e5m2_to_bf16(fp8_byte)
    ELSE:
        RETURN fp8_e4m3_to_bf16(fp8_byte)
```

```
DEFINE convert_bf16_to_fp8(bf16_val, fp8_type="bf8", rounding="RNE", saturate=False):
    // BF16 → FP8 E5M2 (BF8) or E4M3 (HF8). Returns FP8 byte.
    IF fp8_type == "bf8":
        RETURN bf16_to_fp8_e5m2(bf16_val, rounding=rounding, saturate=saturate)
    ELSE:
        RETURN bf16_to_fp8_e4m3(bf16_val, rounding=rounding, saturate=saturate)
```

```
DEFINE convert_bf16_to_fp8_biased(bias_byte, bf16_val, fp8_type="bf8", saturate=False):
    // BF16 → FP8 with biased rounding. bias_byte added to mantissa before truncation.
    IF fp8_type == "bf8":
        RETURN bf16_to_fp8_e5m2_biased(bias_byte, bf16_val, saturate=saturate)
    ELSE:
        RETURN bf16_to_fp8_e4m3_biased(bias_byte, bf16_val, saturate=saturate)
```

FP16 Arithmetic Helpers

```
DEFINE fp16_to_fp32_daz(fp16_bits):
    // Widen FP16 to FP32. DAZ=1: flush subnormals to signed zero.
    sign = fp16_bits[15]
    exp  = (fp16_bits >> 10) & 0x1F
    frac =  fp16_bits        & 0x3FF
    IF exp == 0x1F AND frac != 0:  // FP16 NaN → FP32 QNaN
        RETURN fp32.from_bits((sign<<31)|(0xFF<<23)|(frac<<13)|0x400000)
    IF exp == 0:  // subnormal or zero → DAZ flush to ±0
        RETURN fp32.from_bits(sign << 31)
    RETURN fp32.from_bits((sign<<31)|((exp+112)<<23)|(frac<<13))
```

```
DEFINE fp16_to_fp32_exact(fp16_bits):
    // FP16 → FP32 exact widening. FP16 subnormals become FP32 normals — not flushed.
    // All FP16 values (including subnormals) are exactly representable in FP32.
    sign = fp16_bits[15]
    exp  = (fp16_bits >> 10) & 0x1F
    frac =  fp16_bits        & 0x3FF
    IF exp == 0x1F:  // NaN or Inf
        IF frac != 0:  // NaN → FP32 QNaN
            RETURN fp32.from_bits((sign<<31)|(0xFF<<23)|(frac<<13)|0x400000)
        RETURN fp32.from_bits((sign<<31)|(0xFF<<23))  // ±Inf
    IF exp == 0:  // zero or FP16 subnormal
        IF frac == 0:
            RETURN fp32.from_bits(sign << 31)  // ±zero
        // FP16 subnormal: value = frac × 2^(-24). Normalize to FP32.
        // Find leading bit k (highest set bit position, 0=LSB)
        k = 9
        WHILE k >= 0 AND NOT (frac >> k & 1):
            k -= 1
        fp32_exp = (k - 24) + 127  // k + 103
        fp32_frac = (frac & ((1 << k) - 1)) << (23 - k)
        RETURN fp32.from_bits((sign << 31) | (fp32_exp << 23) | fp32_frac)
    // Normal FP16: exponent bias 15 → FP32 bias 127 (add 112)
    RETURN fp32.from_bits((sign<<31)|((exp+112)<<23)|(frac<<13))
```

FP4 E2M1 Fixed-Point Helper

```
DEFINE fp4_e2m1_to_fixpoint(nibble):
    // FP4 E2M1 nibble → 64-bit integer for exact int128 accumulation.
    // Fixed-point factor = 2: all values become integers {0,1,2,3,4,6,8,12}.
    sign = nibble[3]
    exp  = (nibble >> 1) & 0x3
    mant =  nibble       & 0x1
    mag  = mant if exp == 0 else (2 + mant) << (exp - 1)
    RETURN -mag if sign else mag
```

MX Scale Conversion Helpers

E5M3 and E8M0 block scale byte → FP32 conversions, plus the integer decomposition helpers used by all MX outer products.

```
DEFINE bf16_to_e5m3(bf16_val, saturate=False):
    // BF16 bit pattern (uint16) → E5M3 unsigned block scale byte.
    // Format: {exp5, mant3}, bias=15. Input sign bit is ignored (unsigned output).
    // RNE rounding on 3-bit mantissa.
    // NaN/Inf (exp8=0xFF) → 0xFF (NaN sentinel) or 0xFE (saturating Inf).
    // Zero/subnormal (exp8=0) → 0x00.
    // Overflow (exp5 ≥ 32) → 0xFE (saturating) or 0xFF (non-saturating).
    // 0xFF is the NaN sentinel; valid input rounding to 0xFF is clamped to 0xFE.
    // BF16: 1 sign + 8 exp (bias 127) + 7 mantissa
    exp8  = (bf16_val >> 7) & 0xFF
    mant7 =  bf16_val       & 0x7F
    // Sign bit ignored — E5M3 is unsigned

    IF exp8 == 0xFF:  // NaN or Inf
        IF mant7 != 0:
            RETURN 0xFF  // NaN → NaN sentinel
        RETURN 0xFE if saturate else 0xFF  // Inf → max normal or NaN sentinel

    IF exp8 == 0:  // zero or BF16 subnormal
        RETURN 0x00  // underflows E5M3 range

    // Rebias: BF16 bias 127 → E5M3 bias 15, delta = -112
```

```
    exp5 = exp8 - 112

    IF exp5 >= 32:  // overflow
        RETURN 0xFE if saturate else 0xFF

    IF exp5 >= 1:  // normal E5M3 result
        mant3  = mant7 >> 4  // top 3 of 7 mantissa bits
        guard  = mant7[3]
        sticky = 1 if (mant7 & 0x7) else 0
    ELSE IF exp5 >= -3:  // subnormal E5M3 (exp8 ∈ [109..112])
        shift   = 1 - exp5  // 1..4
        full    = (1 << 7) | mant7  // 1.mant7 as 8-bit integer
        s_total = 4 + shift  // 5..8
        mant3  = full >> s_total
        guard  = (full >> (s_total - 1)) & 0x1
        sticky = 1 if (full & ((1 << (s_total - 1)) - 1)) else 0
        exp5   = 0
    ELSE:  // deep underflow (exp8 ≤ 108)
        RETURN 0x00

    // RNE rounding
    IF guard AND (sticky OR (mant3 & 0x1)):
        mant3 += 1
    IF mant3 >> 3:  // carry out of 3-bit mantissa
        mant3  = 0
        exp5  += 1
        IF exp5 >= 32:  // overflow after rounding carry
            RETURN 0xFE if saturate else 0xFF

    result = ((exp5 & 0x1F) << 3) | (mant3 & 0x7)
    IF result == 0xFF:  // 0xFF is NaN sentinel
        RETURN 0xFE if saturate else 0xFF
    RETURN result
```

```
DEFINE e5m3_to_fp32(byte_val):
    // E5M3 block scale byte → FP32. Unsigned: 5e + 3m, bias=15.
    // NaN: 0xFF. Zero: 0x00 = exact 0.0 (unlike E8M0 where 0x00 = 2^-127).
    IF byte_val == 0xFF:
        RETURN QNaN_Indefinite
    exp5  = (byte_val >> 3) & 0x1F
    mant3 =  byte_val       & 0x07
    IF exp5 == 0:
        RETURN (2.0 ^ -14) * (mant3 / 8.0)  // subnormal; 0.0 when mant3=0
    RETURN (2.0 ^ (exp5 - 15)) * (1.0 + mant3 / 8.0)
```

```
DEFINE e8m0_to_fp32(byte_val):
    // E8M0 block scale byte → FP32. 8-bit pure exponent, bias=127. NaN: 0xFF.
    // Note: 0x00 = 2^-127 (not zero — unlike E5M3 where 0x00 = 0.0).
    IF byte_val == 0xFF:
        RETURN QNaN_Indefinite
    RETURN 2.0 ^ (byte_val - 127)
```

```
DEFINE int_to_fp32_rne(sign, mag, exp_adjust):
    // Convert sign × mag × 2^exp_adjust to nearest FP32 (RNE), ftz=False.
    // Uses the ACE v1 left-normalise algorithm: J-bit to bit 126, Ovf folded into
    // biased before FTZ check. Eliminates the keep-window carry class of bugs.
    // sign: 0 or 1. mag: non-negative integer. exp_adjust: combined exponent shift.
    IF mag == 0:
        RETURN fp32.from_bits(sign << 31)
    Jbit_position = bit_length(mag) - 1
    shift = 126 - Jbit_position
```

```
    normalised = (mag << shift) if shift >= 0 else (mag >> (-shift))
    sticky = 1 if (normalised & mask(102)) else 0
    Gbit   = normalised[102]
    Lbit   = normalised[103]
    RndAdd = Gbit & (Lbit | sticky)
    Mantissa    = normalised >> 103
    RndMantissa = Mantissa + RndAdd
    Ovf         = RndMantissa[24]
    biased = 127 + Jbit_position + exp_adjust + Ovf
    IF biased > 254:
        RETURN fp32.from_bits((sign << 31) | 0x7F800000)
    IF biased <= 0:
        RETURN fp32.from_bits(sign << 31)
    frac = RndMantissa & 0x7FFFFF
    RETURN fp32.from_bits((sign << 31) | (biased << 23) | frac)
```

```
DEFINE convert_fixpoint128_to_fp32_mx_scaled(acc, fmt_a, fmt_b, scale_a, scale_b, a_fmt=0,
b_fmt=0):
    // Convert int128 accumulator to FP32 with E8M0/E5M3 block scale incorporation. a_fmt/
    b_fmt: 0=E8M0, 1=E5M3.
    IF scale_a == 0xFF OR scale_b == 0xFF:
        RETURN QNaN_Indefinite
    IF acc == 0:
        RETURN fp32.from_bits(0)
    sign, magnitude = (1, -acc) if acc < 0 else (0, acc)
    factor = (32 if fmt_a == "bf8"  AND fmt_b == "bf8"  else
              18 if fmt_a == "hf8"  AND fmt_b == "hf8"  else
               2 if fmt_a in ("bf4") AND fmt_b in ("bf4") else 25)
    ma, ea = _scale_int_mantissa(scale_a, a_fmt)
    mb, eb = _scale_int_mantissa(scale_b, b_fmt)
    RETURN int_to_fp32_rne(sign, magnitude * ma * mb, -factor + ea + eb)
```

```
DEFINE convert_fixpoint32_to_fp32_mx_scaled(acc, scale_a, scale_b, a_fmt, b_fmt, adjust):
    // Convert int32 accumulator to FP32 with E8M0/E5M3 block scale incorporation. a_fmt/
    b_fmt: 0=E8M0, 1=E5M3. adjust: implicit bias exponent (MX_INT4_ADJUST=0 or
    MX_INT8_ADJUST=-12).
    IF scale_a == 0xFF OR scale_b == 0xFF:
        RETURN QNaN_Indefinite
    IF acc == 0:
        RETURN fp32.from_bits(0)
    sign, mag = (1, ABS(acc)) if acc < 0 else (0, acc)
    ma, ea = _scale_int_mantissa(scale_a, a_fmt)
    mb, eb = _scale_int_mantissa(scale_b, b_fmt)
    RETURN int_to_fp32_rne(sign, mag * ma * mb, adjust + ea + eb)
```

MX Integer Implicit Bias Helper

The `_mxint_adjust` helper returns the combined implicit bias exponent for MX INT4 and INT8 outer products, based on format-specific design decisions rather than a general signedness-derived formula:

- **INT8 (n=8):** OCP MX assigns a step of 2^{-6} per element. This step applies to both signed and unsigned INT8 operands (U8 treated as XX.XXXXXX fixed-point). The combined bias is $2^{-6} \times 2^{-6} = 2^{-12}$ for all sign variants — adjust = **−12**.
- **INT4 (n=4):** AMD design decision — unity implicit bias (2^0) for all sign variants. Combined bias = $2^0 \times 2^0 = 2^0$ — adjust = **0**.

```
DEFINE _mxint_adjust(n_bits, a_signed, b_signed):
    // Return the MX integer product implicit scale exponent.
    // INT8 (n_bits=8): OCP MX signed step 2^(-6) applied to both signed and unsigned
      // operands -> combined adjust = -12 for all sign variants (SS/SU/US/UU).
    // INT4 (n_bits=4): unity implicit bias (2^0) for all sign
```

```
        // variants -> combined adjust = 0 for all (SS/SU/US/UU).
    // Other widths: general OCP formula (signed: 2^-(n-2), unsigned: 2^-(n-1)).
    IF n_bits == 8:
        RETURN -12  // OCP MX INT8 signed step 2^(-6), applied to both S and U
    IF n_bits == 4:
        RETURN 0  // unity bias, all sign variants
    a_exp = -(n_bits - 2) if a_signed else -(n_bits - 1)
    b_exp = -(n_bits - 2) if b_signed else -(n_bits - 1)
    RETURN a_exp + b_exp
```

FP32 / FP64 Multiply-Accumulate Helpers

`float32_mad` and `float64_mad` implement a fused multiply-accumulate `acc + a × b` under MXCSR control, used by the single and double precision outer products (TOP1PS and TOP1PD). These helpers model the numerical result only; MXCSR exception flag state is specified in the instruction pages and is not modelled in pseudocode.

```
DEFINE float32_mad(acc, a, b, mxcsr_rc=0, mxcsr_daz=False, mxcsr_ftz=False):
    // FP32 fused multiply-accumulate under MXCSR control.
    fa = float(a)
    fb = float(b)
    fc = float(acc)
    IF mxcsr_daz:
        IF is_denormal(a):
            fa = 0.0
        IF is_denormal(b):
            fb = 0.0
        IF is_denormal(acc):
            fc = 0.0
    result = _fma32(fa, fb, fc)  // FMA: single rounding, intermediate not rounded
    IF mxcsr_ftz AND is_denormal(result):
        result = fp32(0.0)
    RETURN result
```

```
DEFINE float64_mad(acc, a, b, mxcsr_rc=0, mxcsr_daz=False, mxcsr_ftz=False):
    // FP64 fused multiply-accumulate under MXCSR control.
    _FP64_MIN_NORMAL = 2.2250738585072014e-308
    fa = float(a)
    fb = float(b)
    fc = float(acc)
    IF mxcsr_daz:
        IF fa != 0.0 AND ABS(fa) < _FP64_MIN_NORMAL:
            fa = 0.0
        IF fb != 0.0 AND ABS(fb) < _FP64_MIN_NORMAL:
            fb = 0.0
        IF fc != 0.0 AND ABS(fc) < _FP64_MIN_NORMAL:
            fc = 0.0
    result = _fma64(fa, fb, fc)  // FMA: single rounding, intermediate not rounded
    IF mxcsr_ftz AND result != 0.0 AND ABS(result) < _FP64_MIN_NORMAL:
        result = 0.0
    RETURN result
```

FP32 Accumulation Helper

`float32_add` implements a controlled FP32 addition with explicit DAZ and FTZ parameters, used by all AI outer products to accumulate the scaled sum-of-products into the tile register. MXCSR is not consulted.

```
DEFINE float32_add(a, b, daz_a=False, daz_b=False, ftz=True):
    // FP32 addition with explicit DAZ/FTZ controls. MXCSR is not consulted.
    fa = float(a)
    fb = float(b)
```

```
    IF daz_a AND is_denormal(a):
        fa = 0.0
    IF daz_b AND is_denormal(b):
        fb = 0.0
    result = fp32(fa + fb)
    IF ftz AND is_denormal(result):
        result = fp32(0.0)
    RETURN result
```

MX Outer Product Subtile Helpers

```
DEFINE op8mxbf4_subtile(srcdest, src1_dword, src1_scale, a_fmt, src2_dword, src2_scale,
b_fmt):
    // MX FP4 E2M1 rank-8 subtile: 8 nibble pairs → int128 accumulator → scaled FP32.
    IF src1_scale == 0xFF OR src2_scale == 0xFF:
        RETURN QNaN_Indefinite
    acc = 0
    FOR k = 0 TO 7:
        a_nib = (src1_dword >> (4*k)) & 0xF
        b_nib = (src2_dword >> (4*k)) & 0xF
        acc  += fp4_e2m1_to_fixpoint(a_nib) * fp4_e2m1_to_fixpoint(b_nib)
    sop_fp32 = convert_fixpoint128_to_fp32_mx_scaled(
        acc, "bf4", "bf4", src1_scale, src2_scale, a_fmt=a_fmt, b_fmt=b_fmt)
    RETURN float32_add(srcdest, sop_fp32, daz_a=1, daz_b=0, ftz=1)  // DAZ=1 on acc; FTZ=1
```

```
DEFINE op8mxn_subtile(srcdest, src1_dword, src1_scale, a_fmt, src2_dword, src2_scale, b_fmt,
a_signed, b_signed):
    // MX INT4 rank-8 subtile: 8 nibble pairs → int32 accumulator → scaled FP32.

    IF src1_scale == 0xFF OR src2_scale == 0xFF:
        RETURN QNaN_Indefinite
    acc = 0
    FOR k = 0 TO 7:
        a_nib = (src1_dword >> (4*k)) & 0xF
        b_nib = (src2_dword >> (4*k)) & 0xF
        a_val = int4_s_to_int32(a_nib) if a_signed else uint4_to_int32(a_nib)
        b_val = int4_s_to_int32(b_nib) if b_signed else uint4_to_int32(b_nib)
        acc  += a_val * b_val
    adjust = _mxint_adjust(4, a_signed, b_signed)
    sop_fp32 = convert_fixpoint32_to_fp32_mx_scaled(
        acc, src1_scale, src2_scale, a_fmt, b_fmt, adjust=adjust)
    RETURN float32_add(srcdest, sop_fp32, daz_a=1, daz_b=0, ftz=1)  // DAZ=1 on acc; FTZ=1
```

Appendix C — APX Interaction

This specification is designed to be implemented on processors that also implement Advanced Performance Extensions (APX) [5]. 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
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
ACE-E4, ACE-E7, ACE-E8	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 17: 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 10).

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

Appendix D — References

[1] Advanced Micro Devices, Inc. and Intel Corporation, “AI Compute Extensions (ACE) Specification,” 2026. [Online]. Available: https://x86ecosystem.org/resource/ai-compute-extensions-ace-specification/

[2] Intel Corporation, “Intel® Advanced Vector Extensions 10.2 Architecture Specification,” Jan. 2026. [Online]. Available: https://cdrdv2-public.intel.com/913918/361050-006-intel-avx10.2-arch-spec.pdf

[3] Advanced Micro Devices, Inc., “AMD64 Bit Matrix Multiply and Bit Reversal Instructions,” Jan. 2026. [Online]. Available: https://docs.amd.com/v/u/en-US/69192-PUB

[4] Open Compute Project, “OCP Microscaling Formats (MX) Specification 1.0,” Sep. 2023. [Online]. Available: https://www.opencompute.org/documents/ocp-microscaling-formats-mx-v1-0-spec-final-pdf/

[5] Intel Corporation, “Intel® Advanced Performance Extensions (Intel® APX) Architecture Specification,” Jul. 2025. [Online]. Available: https://cdrdv2-public.intel.com/861610/355828-007-intel-apx-spec.pdf