Intel® Advanced Vector Extensions 10.2

Architecture Specification

January, 2025
Revision 3.0

Document Number: 361050-003US

Notices & **Disclaimers**

This document contains information on products in the design phase of development. The information here is subject to change without notice. Do not finalize a design with this information.

Intel technologies may require enabled hardware, software or service activation.

No product or component can be absolutely secure.

Your costs and results may vary.

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

All product plans and roadmaps are subject to change without notice. The products described may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Intel disclaims 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.

Code names are used by Intel 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.

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 and b) code included in this document is licensed subject to the Zero-Clause BSD open source license (0BSD), https://opensource.org/licenses/0BSD. You may create software implementations based on this document and in compliance with the foregoing that are intended to execute on the Intel product(s) referenced in this document. No rights are granted to create modifications or derivatives of this document.

Contents

List of Figures

List of Tables

Chapter 1

CHANGES

Revision Number	Description	Date
1.0	1. Initial document release	July 11, 2024
2.0	1. Updated introduction with AVX10 CPUID enumeration clarification on VL128 bit reserved-at-1 behavior 2. Updated introduction with AVX10 state management and VMX extensions 3. Updated VFPCLASSPBF16 DAZ behavior (does not consult MXCSR) 4. Updated encodings of VCOM*/VUCOM* ops (swizzling of required prefixes, F2 and F3) 5. Updated encoding for VGETEXPPBF16 (updating map and required prefixes)	October 1, 2024

3.0	1. Update AVX10.2 ISA content to reflect latest ISE release; including new AVX10 variants of MOVRS and AMX ops. 2. Update Update AVX10.2 VMX architecture to highlight that the processor-based execution controls are an optional part of the architecture. 3. Update mnemonic of VCOMSBF16 to VCOMISBF16 for consistency for RFLAGS-updating V*COM ops. 4. Update mnemonics for a large class of BF16 operations to remove infix "NE" for all instructions and to remove "P" (for packed) as AI data-types (such as BF16) are implicitly packed. This affects the following instructions had that been present: • TCVTROWPS2PBF16L • VADDNEPBF16 • VCMPPBF16 • VDIVNEPBF16 • VF[N]M[ADD,SUB][132,321,213]NEPBF16 • VFPCLASSPBF16 • VGETEXPPBF16 • VGETMANTPBF16 • VMAXPBF16 • VMINMAXPBF16 • VMINPBF16 • VMULNEPBF16 • VRCPPBF16 • VREDUCENEPBF16 • VRNDSCALENEPBF16 • VRSQRTPBF16 • VSCALEFNEPBF16 • VSQRTNEPBF16 • VSUBNEPBF16 • VCVTNE* 5. Update pseudocode and usage of getexp_bf16 and getmant_bf16 in AVX10.2 BF16 ops. 6. Update pseudocode of VCVTBIASPH2* to fix an indentation issue.	November 5, 2024

Chapter 2

CPUID

This section summarizes the CPUID names and leaf mappings referenced in this document.

CPUID	Allocation
AMX-AVX512	CPUID.(0x1E.0x1).EAX[7]
AMX-BF16	CPUID.(0x7.0x0).EDX[22] , AND newly mirrored in CPUID.(0x1E.0x1).EAX[1]
AMX-COMPLEX	CPUID.(0x7.0x1).EDX[8] , AND newly mirrored in CPUID.(0x1E.0x1).EAX[2]
AMX-FP16	CPUID.(0x7.0x1).EAX[21] , AND newly mirrored in CPUID.(0x1E.0x1).EAX[3]
AMX-INT8	CPUID.(0x7.0x0).EDX[25] , AND newly mirrored in CPUID.(0x1E.0x1).EAX[0]
AVX10.2	CPUID.(0x7.0x1).EDX[19] and CPUID.(0x24.0x0).EBX[7:0] >= 2
MOVRS	CPUID.(0x7.0x1).EAX[31]
SM4	CPUID.(0x7.0x1).EAX[2]

Chapter 3

INTRODUCTION

3.1 INTEL® AVX10 INTRODUCTION

Intel® Advanced Vector Extensions 10 (Intel® AVX10) represents the first major new vector ISA since the introduction of Intel® Advanced Vector Extensions 512 (Intel® AVX-512) in 2013. This ISA will establish a common, converged vector instruction set across all Intel® architectures, incorporating the modern vectorization aspects of Intel® AVX-512. This ISA will be supported on all future processors, including Performance cores (P-cores) and Efficient cores (E-cores).

The Intel® AVX10 ISA represents the latest in ISA innovations, instructions, and features moving forward. Based on the Intel® AVX-512 ISA feature set and including all Intel® AVX-512 instructions introduced with future Intel® Xeon® processors based on the Granite Rapids microarchitecture, it will support all instruction vector lengths (128, 256, and 512), as well as scalar and opmask instructions. While Intel® AVX10 does not require all processors to support all vector lengths, it expected that all Intel® processors will support vector lengths of at least 128 bits and 256 bits.

3.1.1 FEATURES AND CAPABILITIES

The Intel® AVX10 architecture introduces several features and capabilities beyond the Intel® AVX2 ISA:

- Version-based instruction set enumeration.
- Intel® AVX10/256 – Converged implementation support on all Intel® processors to include all the existing Intel® AVX-512 capabilities such as EVEX encoding, 32 vector registers and eight mask registers at a maximum vector length of 256 bits.
- Embedded rounding and Suppress All Exceptions (SAE) control for YMM (256-bit) versions of the instructions.

3.1.2 FEATURE ENUMERATION

Intel® AVX10 introduces a versioned approach for enumeration that is monotonically increasing, inclusive, and supporting all vector lengths. This is introduced to simplify application development by ensuring that all Intel® processors support the same features and instructions at a given Intel® AVX10 version number, as well as reduce the number of CPUID feature flags required to be checked by an application to determine feature support. In this enumeration paradigm, the application developer will only need to check three fields:

1. A CPUID feature bit indicating that the Intel® AVX10 ISA is supported.
2. A version number to ensure that the supported version is greater than or equal to the desired version.
3. Vector length (VL) bits indicating the supported vector lengths. (All processors will support the 128-bit vector length; this is not enumerated explicitly.)

The “AVX10 Converged Vector ISA Enable” bit will indicate processor support for the ISA and the presence of an “AVX10 Converged Vector ISA” leaf containing fields for the version number and the supported vector bit lengths. See Table 3.1 for details.

CPUID Bit	Description	Type
CPUID.(EAX=07H, ECX=01H):EDX[bit 19]	If 1, the Intel® AVX10 Converged Vector ISA is supported.	Bit (0/1)
CPUID.(EAX=24H, ECX=00H):EAX[bits 31:0]	Reports the maximum supported sub-leaf.	Integer
CPUID.(EAX=24H, ECX=00H):EBX[bits 7:0]	Reports the Intel® AVX10 Converged Vector ISA version.	Integer (≥ 1)
CPUID.(EAX=24H, ECX=00H):EBX[bits 15:8]	Reserved	N/A
CPUID.(EAX=24H, ECX=00H):EBX[bit 16]	Reserved at 1	Bit (1)[1]
CPUID.(EAX=24H, ECX=00H):EBX[bit 17]	VL256 Support – If 1, indicates that 256-bit vector support is present.	Bit (0/1)
CPUID.(EAX=24H, ECX=00H):EBX[bit 18]	VL512 Support – If 1, indicates that 512-bit vector support is present.	Bit (0/1)
CPUID.(EAX=24H, ECX=00H):EBX[bits 31:19]	Reserved	N/A
CPUID.(EAX=24H, ECX=00H):ECX[bits 31:0]	Reserved	N/A
CPUID.(EAX=24H, ECX=00H):EDX[bits 31:0]	Reserved	N/A
CPUID.(EAX=24H, ECX=01H):EAX[bits 31:0]	Reserved for discrete feature bits.	N/A
CPUID.(EAX=24H, ECX=01H):EBX[bits 31:0]	Reserved for discrete feature bits.	N/A
CPUID.(EAX=24H, ECX=01H):ECX[bits 31:0]	Reserved for discrete feature bits.	N/A
CPUID.(EAX=24H, ECX=01H):EDX[bits 31:0]	Reserved for discrete feature bits.	N/A

Table 3.1: CPUID Enumeration of Intel® AVX10

The versioned approach to ISA enumeration is expected to adhere to the following rules when incrementing from version N to N+1:

- All contemporary processor families[2] support Intel® AVX10 Version N+1.
- Intel® AVX10 Version N+1 delivers significant value over version N to justify the associated software enabling efforts.

[1] Earlier versions of this specification documented this bit as enumerating VL128 support. All processors supporting Intel® AVX10 will include 128-bit vector support.

[2] Contemporary processor families supporting Intel® AVX10 begin with future Intel® Xeon processors based on Granite Rapids microarchitecture

In the rare case of a feature needing to be introduced in-between versions, a discrete CPUID feature bit of the form “AVX10-XXXX” may be allocated and enumerated in sub-leaf 1 of CPUID leaf 24H, i.e., CPUID.(EAX=24H, ECX=01H). However, this is expected to be the exception rather than the norm due to the necessity for entrenched legacy support.

Several other important tenets regarding Intel® AVX10 enumeration are as follows:

- Versions are expected to be inclusive such that version N+1 is a superset of version N. Once an instruction is introduced in Intel® AVX10.x, it is expected to be carried forward in all subsequent Intel® AVX10 versions, allowing a developer to check only for a version greater than or equal to the desired version.
- Any processor that enumerates support for Intel® AVX10 will also enumerate support for Intel® AVX and Intel® AVX2.
- Developers can assume that the highest supported vector length for a processor implies that all lesser vector lengths are also supported. Scalar Intel® AVX-512 instructions will be supported independent of the maximum vector length.

The initial, fully-featured version of Intel® AVX10 will be enumerated as Version 2 (denoted as Intel® AVX10.2). This will include the new YMM-form of embedded rounding and Suppress All Exceptions (SAE), the new enumeration scheme, and several new sets of instructions.

An early version of Intel® AVX10 (Version 1, or Intel® AVX10.1) that only enumerates the Intel® AVX-512 instruction set at 128, 256, and 512 bits will be enabled on the Granite Rapids microarchitecture for software pre-enabling. Applications written to Intel® AVX10.1 will run on any future Intel® processor (P-core or E-core) that enumerates Intel® AVX10.1 or higher at the matching desired vector lengths. Intel® AVX-512 instruction families included in Intel® AVX10.1 are shown in Table 3.2.

Feature Introduction	**Intel® AVX-512 CPUID Feature Flags Included in Intel® AVX10**
Intel® Xeon® Scalable Processor Family based on Skylake microarchitecture	AVX512F, AVX512CD, AVX512BW, AVX512DQ
Intel® Core™ processors based on Cannon Lake microarchitecture	AVX512-VBMI, AVX512-IFMA
2nd generation Intel® Xeon® Scalable Processor Family based on Cascade Lake product	AVX512-VNNI
3rd generation Intel® Xeon® Scalable Processor Family based on Cooper Lake product	AVX512-BF16
3rd generation Intel® Xeon® Scalable Processor Family based on Ice Lake microarchitecture	AVX512-VPOPCNTDQ, AVX512-VBMI2, VAES, GFNI, VPCLMULQDQ, AVX512-BITALG
4th generation Intel® Xeon® Scalable Processor Family based on Sapphire Rapids microarchitecture	AVX512-FP16

Table 3.2: Intel® AVX-512 CPUID Feature Flags Included in Intel® AVX10

Note

VAES, VPCLMULQDQ, and GFNI EVEX instructions will be supported on Intel® AVX10.1 machines but will continue to be enumerated by their existing discrete CPUID feature flags. This requires the developer to check for both the feature and Intel® AVX10, e.g., {AVX10.1 AND VAES}.

Intel® AVX-512 will continue to be supported on P-core-only processors for the foreseeable future to support legacy applications. However, new vector ISA features will only be added to the Intel® AVX10 ISA moving forward. While Intel® AVX10/512 includes all Intel® AVX-512 instructions, it important to note that applications compiled to Intel® AVX-512 with vector length limited to 256 bits are not guaranteed to be compatible on an Intel® AVX10/256 processor. Intel® will develop tools to enable developers to validate their code prior to deployment.

Feature	Intel® AVX-512	Intel® AVX10.1/256	Intel® AVX10.2/256	Intel® AVX10.1/512	Intel® AVX10.2/512
128-bit vector (XMM) register support	Yes	Yes	Yes	Yes	Yes
256-bit vector (YMM) register support	Yes	Yes	Yes	Yes	Yes
512-bit vector (ZMM) register support	Yes	No	No	Yes	Yes
YMM embedded rounding	No	No	Yes	No	Yes
ZMM embedded rounding	Yes	No	No	Yes	Yes
New AVX10.2 Instructions	No	No	Yes	No	Yes

Table 3.3: Feature Differences Between Intel® AVX-512 and Intel® AVX10

3.1.3 STATE MANAGEMENT

Intel® AVX10 state enumeration in CPUID leaf 0xD and enabling in XCR0 register are identical to Intel® AVX-512 regardless of the maximum vector length supported. The CPUID leaf D enumeration will enumerate the state components and its sizes exactly as Intel® AVX-512 state. Intel® AVX10 state will be enabled in XCR0 register exactly as Intel® AVX-512 state. Table 3.4 highlights the Intel® AVX10 state components and their corresponding XCR0 bits. Please refer to Section 13.1, Volume 1 of Intel Software Developer Manual for the complete definition of these state components.

State Component	XCR0 Index Bit
SSE State	1
AVX State	2
Opmask State	5
ZMM_Hi256 state	6
Hi16_ZMM state	7

Table 3.4: XCR0 Feature Bits for Intel® AVX10

XSAVE*/XRSTOR* instructions on Intel® AVX10/512 processors behave exactly as in Intel® AVX-512 processors.

XSAVE*/XRSTOR* instructions on Intel® AVX10/256 processors operate only on the lower 256 bits of each ZMM register.

- There are no changes to SSE, AVX and Opmask state save/restore and INIT tracking.
- There are no changes to Hi16_ZMM state INIT tracking. However, ZMM_Hi256 state will always be tracked as INIT.
- XRSTOR* when state component's XSTATE_BV header is 1:
  - Restore the lower 256 bit of each ZMM register in Hi16_ZMM state and ignore the contents of the memory corresponding to upper 256 bits.
  - Ignore the contents of the memory corresponding to ZMM_Hi256 state and the corresponding XSTATE_BV bit. ZMM_Hi256 state INIT tracker remains 0.
  - Note that XRSTOR* instructions are allowed to access memory corresponding to the upper 256 bits of ZMM registers.
- XSAVEOPT/C/S when state component is non-INIT or XSAVE
  - Save the lower 256 bits of each ZMM registers in Hi16_ZMM state and zero the memory corresponding to upper 256 bits.
  - Zero the memory corresponding ZMM_Hi256 (This applies only to XSAVE. Other XSAVE* instructions incorporate INIT optimization. Hence when the state component is INIT, they only update the XSTATE_BV to 0 and don't save the state.)

3.1.4 VMX FOR INTEL® AVX10 VIRTUAL MACHINES

Processors that support the Intel® AVX10 instruction set may provide a new processor-based VM execution control bit (tertiary control bit 13) in virtual machine control structure (VMCS) to create Intel® AVX10/256 virtual machines (VMs).

On Intel® AVX10/512 processors, when the execution control bit is set and the guest operating system (OS) enabled AVX10 by setting XCR0[7:5] = 111, the following behavior will be observed during VMX non-root operation.

- Intel® AVX10 vector instructions with vector length (VL) = 512 encoded as EVEX.512 will produce an undefined opcode (#UD) exception to provide #UD compatibility with Intel® AVX10/256 processors.
- XRSTOR* instructions will behave as follows.
  - Restore the lower 256 bit of each ZMM register in Hi16_ZMM state and zero the upper 256 bits of each ZMM register. Ignore the contents of the memory corresponding to upper 256 bits.
  - Zero the upper 256 bits of each ZMM register in Hi16_ZMM state irrespective of the corresponding XSTATE_BV value. Ignore the contents of the memory corresponding to Hi16_ZMM state.
  - Note that XRSTOR* instructions are allowed to access memory corresponding to the upper 256 bits ZMM registers.
- XSAVE* instructions will behave exactly as native Intel® AVX10/256 CPU as described in section 3.1.3

Intel® AVX10/256 processors may support the above-mentioned processor-based execution control bit for software compatibility, but it would not modify execution behavior. This would allow VMMs to set execution control bits whenever it creates an Intel® AVX10/256 VM in a way that enables migration of the VM between Intel® AVX10/256 processors and Intel® AVX10/512 processors without the need to modify execution control configuration.

3.1.5 ENCODING EXTENSIONS

As Intel® AVX10 is expected to closely follow the Intel® AVX-512 ISA specification, EVEX encoding changes will be minimal. In the current EVEX encoding specification, the EVEX.b bit encodes multiple functionality which can override the vector length L'L bits when combined with modrm.reg - 0x11 (register-register) to enable static/embedded rounding control. The L'L bits define the rounding mode definition, overriding the MXCSR.RC field while also implying Suppress All Exceptions (SAE). Because the vector length bits (L'L) have been repurposed, the vector length of instructions utilizing embedded rounding is constrained to 512-bits or scalar.

EVEX (0x62)	Payload byte 0							
	R3	X3	B3	R4	**B4**	M2	M1	M0
	Payload byte 1							
	W	V3	V2	V1	V0	**U**	p	p
	Payload byte 2							
	z	L	L	b	V4	a	a	a

Figure 3.1: Encoding of EVEX instructions

In order to support embedded rounding capabilities for YMM/256 bit Intel® AVX10.2 instructions, the EVEX.U bit is repurposed. It's current assignment is set to always be "1". For Intel® AVX10.2, an EVEX.U bit

value of ”0” combined with an EVEX.b value of ”1” will enable embedded rounding for supported YMM/256 bit instructions in any Intel® AVX10 ISA version 2 or greater. Embedded rounding for scalars will utilize the same flow as the current design (using the pp bits) to minimize duplication and validation.

EVEX.b	EVEX.U	MODRM.MOD	Semantics
0	1	11	Legacy. Reg-reg VL based on EVELX.LL. No embedded rounding
1	1	11	Legacy. 512 bit reg-reg supporting embedded rounding
1	0	11	New YMM/256 bit reg-reg supporting embedded rounding
0	0	11	Reserved
0/1	0	00/10/01	Reserved

Table 3.5: EVEX.b and EVEX.u Encoding For Embedded Rounding

The following pseudocode will be added for each Intel® AVX10.2 instruction to support embedded rounding at 256 bit and 512 bit vector lengths:

```
If ModR/M=1 and EVEX.b=1 and {VL=512 or {VL=256 and EVEX.u=0}}
        SET_ROUNDING_MODE{EVEX.RC}  \\ Embedded rounding control enabled
Else
        SET_ROUNDING_MODE{MXCSR.RC} \\ Rounding control determined from MXCSR
End
```

3.1.6 INTEL® AVX10.2 NEW INSTRUCTIONS

Intel® AVX10 Version 2 (Intel® AVX10.2) includes a suite of new instructions delivering new AI features and performance, accelerated media processing, expanded Web Assembly, and Cryptography support, along with enhancements to existing legacy instructions for completeness and efficiency. All new instructions will be enumerated via the Intel® AVX10.2 feature flags and can be considered to conform to the taxonomy below.

- AI Datatypes, Conversions, and post-Convolution Instructions: introduces a suite of AI instructions including FP8 datatypes, convert instructions supporting single, half, and quarter precision (FP32, FP/BF16, E5M2/E4M3 FP8), and post-convolution targeted instructions including arithmetic, scale, min/max, and transcendentals. Note: the FP8 data types are defined as in the Open Compute Project 'OCP 8-bit Floating Point Specification (OFP8)'. The two FP8 (OFP8) formats are E5M2 and E4M3. In instruction mnemonics and pseudo-code, these are denoted throughout this document by BF8 (or bf8) and HF8 (or hf8), respectively.
- Media Acceleration: hardware support for codecs through two new media-targeted instructions. VMPSADBW extends the existing MPSADBW instructions to 512 bits, accelerating motion estimation. Also, 16-bit VNNI now supports all sign combinations further accelerating 16-bit video processing.
- IEEE-754-2019 Minimum and Maximum Support: introduces min/max instructions supporting NAN behavior as specified in IEEE-754-2019, making it compatible for WebAssembly application development. Also applies to other numeric codes to indicate invalid results.

- Saturating Conversions: saturating conversion instructions to support languages such as RUST and WASM. Also applicable for machine learning.
- Zero-extending Partial Vector Copies: aligns the zero-extending partial vector copies with existing memory move instructions. The instructions will clear the destination registers irrespective of the load from memory or register.
- FP Scalar Comparison: extensions to test all common FP relationships directly. Previously not all flags were capable of being set directly with a single instruction. These new instructions remove previous limitations by more comprehensively setting the appropriate flags.

3.2 FP8 INTRODUCTION

FP8 consists of new datatypes of Floating Point numbers consisting of 8 bits. They are aimed to speedup both training and inference AI workloads. The two new FP8 formats, E5M2 and E4M3, are important optimizations for memory footprint and core AI compute density as well as power and performance efficiency. These formats are introduced in the Open Compute Project 'OCP 8-bit Floating Point Specification (OFP8)' and conversion to and from the two formats will be supported as part of Intel® AVX10.2 ISA.

3.2.1 NUMERIC DEFINITION

Two different FP8 formats are supported: E5M2 FP8 which has 1 sign bit, 5 exponent bits and 2 mantissa bits and E4M3 FP8 which has 1 sign bit, 4 exponent bits and 3 mantissa bits. Due to the very small range and precision of these datatypes, both formats are needed to converge and reach the required accuracy across a wide range of AI topologies. Table 3.6 describes the numerics of each format. While E5M2 follows standard floating point representations, the E4M3 format has a non-standard definition, including the same representation for Infinity and NaN in order to increase its range.

Number	E5M2	E4M3
Exponent Bias	15	7
Max Normal	S.11110.11 = 57344.0 $(1.75 * 2^{15})$	S.1111.110 = 448.0 $(1.75 * 2^{8})$
Min Normal	S.00001.00 = 6.10e-05 (2^{-14})	S.0001.000 = 1.56e-02 (2^{-6})
Max Denormal	S.00000.11 = 4.57e-05 $(0.75 * 2^{-14})$	S.0000.111 = 1.36e-02 $(0.875 * 2^{-6})$
Min Denormal	S.00000.01 = 1.52e-05 $(0.25 * 2^{-14})$	S.0000.001 = 1.95e-03 $(0.125 * 2^{-6})$
NaNs	S.11111.[01, 10, 11]	S.1111.111
Infinity	S.11111.00	NA

Table 3.6: FP8 Formats Numeric Definitions

3.2.2 FP8 ROUNDING, DENORMALS, SPECIAL NUMBERS, AND EXCEPTIONS

Intel® AVX10.2 dot product instructions, upconverts and downconverts are supported. The down converts have two flavors: *RNE* and *BIAS*, which indicate the rounding modes.

- FP rounding: Excluding the *BIAS* downconverts, all instructions use RNE (Round to nearest tie to even) rounding mode. The *BIAS* downconverts use RNE in case the input is denormal and truncate (round towards zero) for normal input.
- Denormal Handling: All instructions function as if FP exceptions are masked. For any type of input, instructions behave as if MXCSR.DAZ is not set. For FP8 output type, instructions behave as if MXCSR.FTZ is not set. For any other type of output, instructions behave as if MXCSR.FTZ is set.

- Special numbers - Nan/inf/overflow handling: Excluding downconvert instructions, all instructions behave as expected. Infinity is bypassed, NaN is bypassed as QNaN, and Overflow returns Infinity. The downconverts have saturation and non-saturation flavors. Table 3.7 describes the downconverts behavior in these cases:

Flavor	Scenario	E5M2	E4M3
Regular	NaN at input	S.11111.[10, 11]	S.1111.111
	+/- infinity at input	S.11111.00	S.1111.111
	Oveflow due to conversion/rounding	S.11111.00	S.1111.111
Saturated	NaN at input	S.11111.[10, 11]	S.1111.111
	+/- infinity at input	S.11111.00	S.1111.111
	Oveflow due to conversion/rounding	S.11110.11	S.1111.110

Table 3.7: FP8 Downconverts Special Numbers Handling

- FP exceptions:
  - No instructions consult MXCSR.
  - No instructions raise exceptions.
  - Special rules:
    - Upconverts and downconverts between different datatypes always update MXCSR.
    - All other instructions never update MXCSR.

3.3 SM4

“Commercial Cryptography” is a set of algorithms and standards used in the commercial area issued and regulated by the Office of State Commercial Cryptography Administration (OSCCA). SM3 and SM4 algorithms are part of the published OSCCA algorithms.

SM4 is a symmetric block cipher algorithm published in 2012.

More details can be found at: https://tools.ietf.org/html/draft-ribose-cfrg-sm4-10

3.4 MOVRS Instructions

MOVRS is a set of load instructions that carry a “read-shared” hint. Functionally, they are equivalent to existing load instructions, and the expectation is that they will be used by software as drop-in replacements for existing load instructions, where the read-shared hint is appropriate. From a performance point of view, they carry a hint that the data being accessed is expected to be concurrently read by multiple cores.

Our processors are typically optimized assuming that, on a first read, data is private and is reasonably likely to be written. This influences which caches data is placed into on an access, and the coherence state of the data after the access.

This is especially relevant for products with non-inclusive last-level caches. Accessing read-shared data, i.e., data concurrently read by multiple cores, with conventional loads can cause performance issues. In particular, for both a multi-readers scenario where data starts in DRAM and a single-producer multi-consumer scenario where data starts dirty in one core’s private cache, snoops are required to distribute the data to the readers.

Loads with a read-shared hint are expected to trigger different uncore behavior than conventional loads. The read-shared hint tells caches that the data is expected to be read by multiple (unspecified) cores within the socket. Thus, hardware, to the extent possible, installs the cache line into the appropriate cache(s) and in the appropriate coherence state to minimize snoops triggered by future reads.

On a cache miss, the read-shared hint may influence which cache(s) hardware installs data into, and the uncore transactions used to fetch data to the core. On a cache hit, the hint is not expected to trigger different behavior. Also, at least for our currently planned implementations, the hint is not stored in the caches; hardware will act (or not) on the hint immediately. The exception to this is hardware prefetchers, which will train on the hint, and attempt to issue requests matching the presence/absence of the hint.

Mixing conventional loads and read-shared loads to the same cache line and/or access stream will functionally behave as expected. However, the cache behavior will depend on the order of the requests and the behavior of the hardware prefetchers. It is easier to predict and understand the hardware behavior if software consistently uses (or not) read-shared loads for a given access pattern.

The exception to this is phased program behavior. Software may have phases, where a given data structure is read-shared in one phase, and not in the next phase. It is expected that the different phases will use the appropriate flavor of load instruction, i.e., read-shared loads for the phase with read sharing and conventional loads for other phases.

Since the read-shared hint simply influences which caches hold a copy of a line and the coherence state, it does not limit how the data is accessed by software in the future. In particular, a future write to a cache line that has been read with a read-shared load instruction, by the same core that executed the load instruction or another core, will architecturally behave as expected. The performance of such a write may be worse than if the data was read with a conventional load instruction, if the read-shared load leaves caches in a state such that additional cache and/or uncore transactions are needed for the write.

The MOVRS family includes scalar, vector, and tile variants. It also includes a software prefetch instruction, PREFETCHRST2, that carries the same read-shared hint. This is for applications that want to apply

software prefetching to read-shared data structures.

Chapter 4

EXCEPTION CLASSES

4.1 EXCEPTION CLASS INSTRUCTION SUMMARY

Type AMX-E7-EVEX			
TCVTROWD2PS	zmm1, tmm2, imm8	AMX-AVX512,AVX10.2	EVEX
TCVTROWPS2BF16H	zmm1, tmm2, imm8	AMX-AVX512,AVX10.2	EVEX
TCVTROWPS2BF16L	zmm1, tmm2, imm8	AMX-AVX512,AVX10.2	EVEX
TCVTROWPS2PHH	zmm1, tmm2, imm8	AMX-AVX512,AVX10.2	EVEX
TCVTROWPS2PHL	zmm1, tmm2, imm8	AMX-AVX512,AVX10.2	EVEX
TILEMOVROW	zmm1, tmm2, imm8	AMX-AVX512,AVX10.2	EVEX
Type AMX-E8-EVEX			
TCVTROWD2PS	zmm1, tmm2, r32	AMX-AVX512,AVX10.2	EVEX
TCVTROWPS2BF16H	zmm1, tmm2, r32	AMX-AVX512,AVX10.2	EVEX
TCVTROWPS2BF16L	zmm1, tmm2, r32	AMX-AVX512,AVX10.2	EVEX
TCVTROWPS2PHH	zmm1, tmm2, r32	AMX-AVX512,AVX10.2	EVEX
TCVTROWPS2PHL	zmm1, tmm2, r32	AMX-AVX512,AVX10.2	EVEX
TILEMOVROW	zmm1, tmm2, r32	AMX-AVX512,AVX10.2	EVEX
Type E10NF			
VCOMISBF16	xmm1, xmm2/m16	AVX10.2	EVEX
Type E11			
VCVTPH2PS	ymm1, xmm2/m128	AVX10.2	EVEX
VCVTPS2PH	xmm1, ymm2, imm8	AVX10.2	EVEX
Type E2			
VADDPD	ymm1, ymm2, ymm3/m256	AVX10.2	EVEX
VADDPH	ymm1, ymm2, ymm3/m256	AVX10.2	EVEX
VADDPS	ymm1, ymm2, ymm3/m256	AVX10.2	EVEX
VCMPPD	k1, ymm2, ymm3/m256, imm8	AVX10.2	EVEX
VCMPPH	k1, ymm2, ymm3/m256, imm8	AVX10.2	EVEX
VCMPPS	k1, ymm2, ymm3/m256, imm8	AVX10.2	EVEX
VCVTDQ2PH	xmm1, ymm2/m256	AVX10.2	EVEX
VCVTDQ2PS	ymm1, ymm2/m256	AVX10.2	EVEX
VCVTHF82PH	xmm1, xmm2/m64	AVX10.2	EVEX
VCVTHF82PH	ymm1, xmm2/m128	AVX10.2	EVEX
VCVTHF82PH	zmm1, ymm2/m256	AVX10.2	EVEX
VCVTPD2DQ	xmm1, ymm2/m256	AVX10.2	EVEX
VCVTPD2PH	xmm1, ymm2/m256	AVX10.2	EVEX
VCVTPD2PS	xmm1, ymm2/m256	AVX10.2	EVEX
VCVTPD2QQ	ymm1, ymm2/m256	AVX10.2	EVEX
VCVTPD2UDQ	xmm1, ymm2/m256	AVX10.2	EVEX
VCVTPD2UQQ	ymm1, ymm2/m256	AVX10.2	EVEX
VCVTPH2DQ	ymm1, xmm2/m128	AVX10.2	EVEX
VCVTPH2IBS	xmm1, xmm2/m128	AVX10.2	EVEX
VCVTPH2IBS	ymm1, ymm2/m256	AVX10.2	EVEX
VCVTPH2IBS	zmm1, zmm2/m512	AVX10.2	EVEX
VCVTPH2IUBS	xmm1, xmm2/m128	AVX10.2	EVEX
VCVTPH2IUBS	ymm1, ymm2/m256	AVX10.2	EVEX
VCVTPH2IUBS	zmm1, zmm2/m512	AVX10.2	EVEX
VCVTPH2PD	ymm1, xmm2/m64	AVX10.2	EVEX

VCVTPH2PSX	ymm1, xmm2/m128	AVX10.2	EVEX
VCVTPH2QQ	ymm1, xmm2/m64	AVX10.2	EVEX
VCVTPH2UDQ	ymm1, xmm2/m128	AVX10.2	EVEX
VCVTPH2UQQ	ymm1, xmm2/m64	AVX10.2	EVEX
VCVTPH2UW	ymm1, ymm2/m256	AVX10.2	EVEX
VCVTPH2W	ymm1, ymm2/m256	AVX10.2	EVEX
VCVTPS2DQ	ymm1, ymm2/m256	AVX10.2	EVEX
VCVTPS2IBS	xmm1, xmm2/m128	AVX10.2	EVEX
VCVTPS2IBS	ymm1, ymm2/m256	AVX10.2	EVEX
VCVTPS2IBS	zmm1, zmm2/m512	AVX10.2	EVEX
VCVTPS2IUBS	xmm1, xmm2/m128	AVX10.2	EVEX
VCVTPS2IUBS	ymm1, ymm2/m256	AVX10.2	EVEX
VCVTPS2IUBS	zmm1, zmm2/m512	AVX10.2	EVEX
VCVTPS2PD	ymm1, xmm2/m128	AVX10.2	EVEX
VCVTPS2PHX	xmm1, ymm2/m256	AVX10.2	EVEX
VCVTPS2QQ	ymm1, xmm2/m128	AVX10.2	EVEX
VCVTPS2UDQ	ymm1, ymm2/m256	AVX10.2	EVEX
VCVTPS2UQQ	ymm1, xmm2/m128	AVX10.2	EVEX
VCVTQQ2PD	ymm1, ymm2/m256	AVX10.2	EVEX
VCVTQQ2PH	xmm1, ymm2/m256	AVX10.2	EVEX
VCVTQQ2PS	xmm1, ymm2/m256	AVX10.2	EVEX
VCVTTPD2DQ	xmm1, ymm2/m256	AVX10.2	EVEX
VCVTTPD2DQS	xmm1, xmm2/m128	AVX10.2	EVEX
VCVTTPD2DQS	xmm1, ymm2/m256	AVX10.2	EVEX
VCVTTPD2DQS	ymm1, zmm2/m512	AVX10.2	EVEX
VCVTTPD2QQ	ymm1, ymm2/m256	AVX10.2	EVEX
VCVTTPD2QQS	xmm1, xmm2/m128	AVX10.2	EVEX
VCVTTPD2QQS	ymm1, ymm2/m256	AVX10.2	EVEX
VCVTTPD2QQS	zmm1, zmm2/m512	AVX10.2	EVEX
VCVTTPD2UDQ	xmm1, ymm2/m256	AVX10.2	EVEX
VCVTTPD2UDQS	xmm1, xmm2/m128	AVX10.2	EVEX
VCVTTPD2UDQS	xmm1, ymm2/m256	AVX10.2	EVEX
VCVTTPD2UDQS	ymm1, zmm2/m512	AVX10.2	EVEX
VCVTTPD2UQQ	ymm1, ymm2/m256	AVX10.2	EVEX
VCVTTPD2UQQS	xmm1, xmm2/m128	AVX10.2	EVEX
VCVTTPD2UQQS	ymm1, ymm2/m256	AVX10.2	EVEX
VCVTTPD2UQQS	zmm1, zmm2/m512	AVX10.2	EVEX
VCVTTPH2DQ	ymm1, xmm2/m128	AVX10.2	EVEX
VCVTTPH2IBS	xmm1, xmm2/m128	AVX10.2	EVEX
VCVTTPH2IBS	ymm1, ymm2/m256	AVX10.2	EVEX
VCVTTPH2IBS	zmm1, zmm2/m512	AVX10.2	EVEX
VCVTTPH2IUBS	xmm1, xmm2/m128	AVX10.2	EVEX
VCVTTPH2IUBS	ymm1, ymm2/m256	AVX10.2	EVEX
VCVTTPH2IUBS	zmm1, zmm2/m512	AVX10.2	EVEX
VCVTTPH2QQ	ymm1, xmm2/m64	AVX10.2	EVEX
VCVTTPH2UDQ	ymm1, xmm2/m128	AVX10.2	EVEX
VCVTTPH2UQQ	ymm1, xmm2/m64	AVX10.2	EVEX

VCVTTPH2UW	ymm1, ymm2/m256	AVX10.2	EVEX
VCVTTPH2W	ymm1, ymm2/m256	AVX10.2	EVEX
VCVTTPS2DQ	ymm1, ymm2/m256	AVX10.2	EVEX
VCVTTPS2DQS	xmm1, xmm2/m128	AVX10.2	EVEX
VCVTTPS2DQS	ymm1, ymm2/m256	AVX10.2	EVEX
VCVTTPS2DQS	zmm1, zmm2/m512	AVX10.2	EVEX
VCVTTPS2IBS	xmm1, xmm2/m128	AVX10.2	EVEX
VCVTTPS2IBS	ymm1, ymm2/m256	AVX10.2	EVEX
VCVTTPS2IBS	zmm1, zmm2/m512	AVX10.2	EVEX
VCVTTPS2IUBS	xmm1, xmm2/m128	AVX10.2	EVEX
VCVTTPS2IUBS	ymm1, ymm2/m256	AVX10.2	EVEX
VCVTTPS2IUBS	zmm1, zmm2/m512	AVX10.2	EVEX
VCVTTPS2QQ	ymm1, xmm2/m128	AVX10.2	EVEX
VCVTTPS2QQS	xmm1, xmm2/m64	AVX10.2	EVEX
VCVTTPS2QQS	ymm1, xmm2/m128	AVX10.2	EVEX
VCVTTPS2QQS	zmm1, ymm2/m256	AVX10.2	EVEX
VCVTTPS2UDQ	ymm1, ymm2/m256	AVX10.2	EVEX
VCVTTPS2UDQS	xmm1, xmm2/m128	AVX10.2	EVEX
VCVTTPS2UDQS	ymm1, ymm2/m256	AVX10.2	EVEX
VCVTTPS2UDQS	zmm1, zmm2/m512	AVX10.2	EVEX
VCVTTPS2UQQ	ymm1, xmm2/m128	AVX10.2	EVEX
VCVTTPS2UQQS	xmm1, xmm2/m64	AVX10.2	EVEX
VCVTTPS2UQQS	ymm1, xmm2/m128	AVX10.2	EVEX
VCVTTPS2UQQS	zmm1, ymm2/m256	AVX10.2	EVEX
VCVTUDQ2PH	xmm1, ymm2/m256	AVX10.2	EVEX
VCVTUDQ2PS	ymm1, ymm2/m256	AVX10.2	EVEX
VCVTUQQ2PD	ymm1, ymm2/m256	AVX10.2	EVEX
VCVTUQQ2PH	xmm1, ymm2/m256	AVX10.2	EVEX
VCVTUQQ2PS	xmm1, ymm2/m256	AVX10.2	EVEX
VCVTUW2PH	ymm1, ymm2/m256	AVX10.2	EVEX
VCVTW2PH	ymm1, ymm2/m256	AVX10.2	EVEX
VDIVPD	ymm1, ymm2, ymm3/m256	AVX10.2	EVEX
VDIVPH	ymm1, ymm2, ymm3/m256	AVX10.2	EVEX
VDIVPS	ymm1, ymm2, ymm3/m256	AVX10.2	EVEX
VFIXUPIMMPD	ymm1, ymm2, ymm3/m256, imm8	AVX10.2	EVEX
VFIXUPIMMPS	ymm1, ymm2, ymm3/m256, imm8	AVX10.2	EVEX
VFMADD132PD	ymm1, ymm2, ymm3/m256	AVX10.2	EVEX
VFMADD132PH	ymm1, ymm2, ymm3/m256	AVX10.2	EVEX
VFMADD132PS	ymm1, ymm2, ymm3/m256	AVX10.2	EVEX
VFMADD213PD	ymm1, ymm2, ymm3/m256	AVX10.2	EVEX
VFMADD213PH	ymm1, ymm2, ymm3/m256	AVX10.2	EVEX
VFMADD213PS	ymm1, ymm2, ymm3/m256	AVX10.2	EVEX
VFMADD231PD	ymm1, ymm2, ymm3/m256	AVX10.2	EVEX
VFMADD231PH	ymm1, ymm2, ymm3/m256	AVX10.2	EVEX
VFMADD231PS	ymm1, ymm2, ymm3/m256	AVX10.2	EVEX
VFMADDSUB132PD	ymm1, ymm2, ymm3/m256	AVX10.2	EVEX
VFMADDSUB132PH	ymm1, ymm2, ymm3/m256	AVX10.2	EVEX

VFMADDSUB132PS	ymm1, ymm2, ymm3/m256	AVX10.2	EVEX
VFMADDSUB213PD	ymm1, ymm2, ymm3/m256	AVX10.2	EVEX
VFMADDSUB213PH	ymm1, ymm2, ymm3/m256	AVX10.2	EVEX
VFMADDSUB213PS	ymm1, ymm2, ymm3/m256	AVX10.2	EVEX
VFMADDSUB231PD	ymm1, ymm2, ymm3/m256	AVX10.2	EVEX
VFMADDSUB231PH	ymm1, ymm2, ymm3/m256	AVX10.2	EVEX
VFMADDSUB231PS	ymm1, ymm2, ymm3/m256	AVX10.2	EVEX
VFMSUB132PD	ymm1, ymm2, ymm3/m256	AVX10.2	EVEX
VFMSUB132PH	ymm1, ymm2, ymm3/m256	AVX10.2	EVEX
VFMSUB132PS	ymm1, ymm2, ymm3/m256	AVX10.2	EVEX
VFMSUB213PD	ymm1, ymm2, ymm3/m256	AVX10.2	EVEX
VFMSUB213PH	ymm1, ymm2, ymm3/m256	AVX10.2	EVEX
VFMSUB213PS	ymm1, ymm2, ymm3/m256	AVX10.2	EVEX
VFMSUB231PD	ymm1, ymm2, ymm3/m256	AVX10.2	EVEX
VFMSUB231PH	ymm1, ymm2, ymm3/m256	AVX10.2	EVEX
VFMSUB231PS	ymm1, ymm2, ymm3/m256	AVX10.2	EVEX
VFMSUBADD132PD	ymm1, ymm2, ymm3/m256	AVX10.2	EVEX
VFMSUBADD132PH	ymm1, ymm2, ymm3/m256	AVX10.2	EVEX
VFMSUBADD132PS	ymm1, ymm2, ymm3/m256	AVX10.2	EVEX
VFMSUBADD213PD	ymm1, ymm2, ymm3/m256	AVX10.2	EVEX
VFMSUBADD213PH	ymm1, ymm2, ymm3/m256	AVX10.2	EVEX
VFMSUBADD213PS	ymm1, ymm2, ymm3/m256	AVX10.2	EVEX
VFMSUBADD231PD	ymm1, ymm2, ymm3/m256	AVX10.2	EVEX
VFMSUBADD231PH	ymm1, ymm2, ymm3/m256	AVX10.2	EVEX
VFMSUBADD231PS	ymm1, ymm2, ymm3/m256	AVX10.2	EVEX
VFNMADD132PD	ymm1, ymm2, ymm3/m256	AVX10.2	EVEX
VFNMADD132PH	ymm1, ymm2, ymm3/m256	AVX10.2	EVEX
VFNMADD132PS	ymm1, ymm2, ymm3/m256	AVX10.2	EVEX
VFNMADD213PD	ymm1, ymm2, ymm3/m256	AVX10.2	EVEX
VFNMADD213PH	ymm1, ymm2, ymm3/m256	AVX10.2	EVEX
VFNMADD213PS	ymm1, ymm2, ymm3/m256	AVX10.2	EVEX
VFNMADD231PD	ymm1, ymm2, ymm3/m256	AVX10.2	EVEX
VFNMADD231PH	ymm1, ymm2, ymm3/m256	AVX10.2	EVEX
VFNMADD231PS	ymm1, ymm2, ymm3/m256	AVX10.2	EVEX
VFNMSUB132PD	ymm1, ymm2, ymm3/m256	AVX10.2	EVEX
VFNMSUB132PH	ymm1, ymm2, ymm3/m256	AVX10.2	EVEX
VFNMSUB132PS	ymm1, ymm2, ymm3/m256	AVX10.2	EVEX
VFNMSUB213PD	ymm1, ymm2, ymm3/m256	AVX10.2	EVEX
VFNMSUB213PH	ymm1, ymm2, ymm3/m256	AVX10.2	EVEX
VFNMSUB213PS	ymm1, ymm2, ymm3/m256	AVX10.2	EVEX
VFNMSUB231PD	ymm1, ymm2, ymm3/m256	AVX10.2	EVEX
VFNMSUB231PH	ymm1, ymm2, ymm3/m256	AVX10.2	EVEX
VFNMSUB231PS	ymm1, ymm2, ymm3/m256	AVX10.2	EVEX
VGETEXPPD	ymm1, ymm2/m256	AVX10.2	EVEX
VGETEXPPH	ymm1, ymm2/m256	AVX10.2	EVEX
VGETEXPPS	ymm1, ymm2/m256	AVX10.2	EVEX
VGETMANTPD	ymm1, ymm2/m256, imm8	AVX10.2	EVEX

VGETMANTPH	ymm1, ymm2/m256, imm8	AVX10.2	EVEX
VGETMANTPS	ymm1, ymm2/m256, imm8	AVX10.2	EVEX
VMAXPD	ymm1, ymm2, ymm3/m256	AVX10.2	EVEX
VMAXPH	ymm1, ymm2, ymm3/m256	AVX10.2	EVEX
VMAXPS	ymm1, ymm2, ymm3/m256	AVX10.2	EVEX
VMINMAXPD	xmm1, xmm2, xmm3/m128, imm8	AVX10.2	EVEX
VMINMAXPD	ymm1, ymm2, ymm3/m256, imm8	AVX10.2	EVEX
VMINMAXPD	zmm1, zmm2, zmm3/m512, imm8	AVX10.2	EVEX
VMINMAXPH	xmm1, xmm2, xmm3/m128, imm8	AVX10.2	EVEX
VMINMAXPH	ymm1, ymm2, ymm3/m256, imm8	AVX10.2	EVEX
VMINMAXPH	zmm1, zmm2, zmm3/m512, imm8	AVX10.2	EVEX
VMINMAXPS	xmm1, xmm2, xmm3/m128, imm8	AVX10.2	EVEX
VMINMAXPS	ymm1, ymm2, ymm3/m256, imm8	AVX10.2	EVEX
VMINMAXPS	zmm1, zmm2, zmm3/m512, imm8	AVX10.2	EVEX
VMINPD	ymm1, ymm2, ymm3/m256	AVX10.2	EVEX
VMINPH	ymm1, ymm2, ymm3/m256	AVX10.2	EVEX
VMINPS	ymm1, ymm2, ymm3/m256	AVX10.2	EVEX
VMULPD	ymm1, ymm2, ymm3/m256	AVX10.2	EVEX
VMULPH	ymm1, ymm2, ymm3/m256	AVX10.2	EVEX
VMULPS	ymm1, ymm2, ymm3/m256	AVX10.2	EVEX
VRANGEPD	ymm1, ymm2, ymm3/m256, imm8	AVX10.2	EVEX
VRANGEPS	ymm1, ymm2, ymm3/m256, imm8	AVX10.2	EVEX
VREDUCEPD	ymm1, ymm2/m256, imm8	AVX10.2	EVEX
VREDUCEPH	ymm1, ymm2/m256, imm8	AVX10.2	EVEX
VREDUCEPS	ymm1, ymm2/m256, imm8	AVX10.2	EVEX
VRNDSCALEPD	ymm1, ymm2/m256, imm8	AVX10.2	EVEX
VRNDSCALEPH	ymm1, ymm2/m256, imm8	AVX10.2	EVEX
VRNDSCALEPS	ymm1, ymm2/m256, imm8	AVX10.2	EVEX
VSCALEFPD	ymm1, ymm2, ymm3/m256	AVX10.2	EVEX
VSCALEFPH	ymm1, ymm2, ymm3/m256	AVX10.2	EVEX
VSCALEFPS	ymm1, ymm2, ymm3/m256	AVX10.2	EVEX
VSQRTPD	ymm1, ymm2/m256	AVX10.2	EVEX
VSQRTPH	ymm1, ymm2/m256	AVX10.2	EVEX
VSQRTPS	ymm1, ymm2/m256	AVX10.2	EVEX
VSUBPD	ymm1, ymm2, ymm3/m256	AVX10.2	EVEX
VSUBPH	ymm1, ymm2, ymm3/m256	AVX10.2	EVEX
VSUBPS	ymm1, ymm2, ymm3/m256	AVX10.2	EVEX
Type E3			
VMINMAXSD	xmm1, xmm2, xmm3/m64, imm8	AVX10.2	EVEX
VMINMAXSH	xmm1, xmm2, xmm3/m16, imm8	AVX10.2	EVEX
VMINMAXSS	xmm1, xmm2, xmm3/m32, imm8	AVX10.2	EVEX
Type E3NF			
VCOMXSD	xmm1, xmm2/m64	AVX10.2	EVEX
VCOMXSH	xmm1, xmm2/m16	AVX10.2	EVEX
VCOMXSS	xmm1, xmm2/m32	AVX10.2	EVEX
VCVTTSD2SIS	r32, xmm1/m64	AVX10.2	EVEX
VCVTTSD2SIS	r64, xmm1/m64	AVX10.2	EVEX

VCVTTSD2USIS	r32, xmm1/m64	AVX10.2	EVEX
VCVTTSD2USIS	r64, xmm1/m64	AVX10.2	EVEX
VCVTTSS2SIS	r32, xmm1/m32	AVX10.2	EVEX
VCVTTSS2SIS	r64, xmm1/m32	AVX10.2	EVEX
VCVTTSS2USIS	r32, xmm1/m32	AVX10.2	EVEX
VCVTTSS2USIS	r64, xmm1/m32	AVX10.2	EVEX
VUCOMXSD	xmm1, xmm2/m64	AVX10.2	EVEX
VUCOMXSH	xmm1, xmm2/m16	AVX10.2	EVEX
VUCOMXSS	xmm1, xmm2/m32	AVX10.2	EVEX
Type E4			
VADDBF16	xmm1, xmm2, xmm3/m128	AVX10.2	EVEX
VADDBF16	ymm1, ymm2, ymm3/m256	AVX10.2	EVEX
VADDBF16	zmm1, zmm2, zmm3/m512	AVX10.2	EVEX
VCMPBF16	k1, xmm2, xmm3/m128, imm8	AVX10.2	EVEX
VCMPBF16	k1, ymm2, ymm3/m256, imm8	AVX10.2	EVEX
VCMPBF16	k1, zmm2, zmm3/m512, imm8	AVX10.2	EVEX
VCVTBF162IBS	xmm1, xmm2/m128	AVX10.2	EVEX
VCVTBF162IBS	ymm1, ymm2/m256	AVX10.2	EVEX
VCVTBF162IBS	zmm1, zmm2/m512	AVX10.2	EVEX
VCVTBF162IUBS	xmm1, xmm2/m128	AVX10.2	EVEX
VCVTBF162IUBS	ymm1, ymm2/m256	AVX10.2	EVEX
VCVTBF162IUBS	zmm1, zmm2/m512	AVX10.2	EVEX
VCVTTBF162IBS	xmm1, xmm2/m128	AVX10.2	EVEX
VCVTTBF162IBS	ymm1, ymm2/m256	AVX10.2	EVEX
VCVTTBF162IBS	zmm1, zmm2/m512	AVX10.2	EVEX
VCVTTBF162IUBS	xmm1, xmm2/m128	AVX10.2	EVEX
VCVTTBF162IUBS	ymm1, ymm2/m256	AVX10.2	EVEX
VCVTTBF162IUBS	zmm1, zmm2/m512	AVX10.2	EVEX
VDIVBF16	xmm1, xmm2, xmm3/m128	AVX10.2	EVEX
VDIVBF16	ymm1, ymm2, ymm3/m256	AVX10.2	EVEX
VDIVBF16	zmm1, zmm2, zmm3/m512	AVX10.2	EVEX
VDPPHPS	xmm1, xmm2, xmm3/m128	AVX10.2	EVEX
VDPPHPS	ymm1, ymm2, ymm3/m256	AVX10.2	EVEX
VDPPHPS	zmm1, zmm2, zmm3/m512	AVX10.2	EVEX
VFCMADDCPH	ymm1, ymm2, ymm3/m256	AVX10.2	EVEX
VFCMULCPH	ymm1, ymm2, ymm3/m256	AVX10.2	EVEX
VFMADD132BF16	xmm1, xmm2, xmm3/m128	AVX10.2	EVEX
VFMADD132BF16	ymm1, ymm2, ymm3/m256	AVX10.2	EVEX
VFMADD132BF16	zmm1, zmm2, zmm3/m512	AVX10.2	EVEX
VFMADD213BF16	xmm1, xmm2, xmm3/m128	AVX10.2	EVEX
VFMADD213BF16	ymm1, ymm2, ymm3/m256	AVX10.2	EVEX
VFMADD213BF16	zmm1, zmm2, zmm3/m512	AVX10.2	EVEX
VFMADD231BF16	xmm1, xmm2, xmm3/m128	AVX10.2	EVEX
VFMADD231BF16	ymm1, ymm2, ymm3/m256	AVX10.2	EVEX
VFMADD231BF16	zmm1, zmm2, zmm3/m512	AVX10.2	EVEX
VFMADDCPH	ymm1, ymm2, ymm3/m256	AVX10.2	EVEX
VFMSUB132BF16	xmm1, xmm2, xmm3/m128	AVX10.2	EVEX

VFMSUB132BF16	ymm1, ymm2, ymm3/m256	AVX10.2	EVEX
VFMSUB132BF16	zmm1, zmm2, zmm3/m512	AVX10.2	EVEX
VFMSUB213BF16	xmm1, xmm2, xmm3/m128	AVX10.2	EVEX
VFMSUB213BF16	ymm1, ymm2, ymm3/m256	AVX10.2	EVEX
VFMSUB213BF16	zmm1, zmm2, zmm3/m512	AVX10.2	EVEX
VFMSUB231BF16	xmm1, xmm2, xmm3/m128	AVX10.2	EVEX
VFMSUB231BF16	ymm1, ymm2, ymm3/m256	AVX10.2	EVEX
VFMSUB231BF16	zmm1, zmm2, zmm3/m512	AVX10.2	EVEX
VFMULCPH	ymm1, ymm2, ymm3/m256	AVX10.2	EVEX
VFNMADD132BF16	xmm1, xmm2, xmm3/m128	AVX10.2	EVEX
VFNMADD132BF16	ymm1, ymm2, ymm3/m256	AVX10.2	EVEX
VFNMADD132BF16	zmm1, zmm2, zmm3/m512	AVX10.2	EVEX
VFNMADD213BF16	xmm1, xmm2, xmm3/m128	AVX10.2	EVEX
VFNMADD213BF16	ymm1, ymm2, ymm3/m256	AVX10.2	EVEX
VFNMADD213BF16	zmm1, zmm2, zmm3/m512	AVX10.2	EVEX
VFNMADD231BF16	xmm1, xmm2, xmm3/m128	AVX10.2	EVEX
VFNMADD231BF16	ymm1, ymm2, ymm3/m256	AVX10.2	EVEX
VFNMADD231BF16	zmm1, zmm2, zmm3/m512	AVX10.2	EVEX
VFNMSUB132BF16	xmm1, xmm2, xmm3/m128	AVX10.2	EVEX
VFNMSUB132BF16	ymm1, ymm2, ymm3/m256	AVX10.2	EVEX
VFNMSUB132BF16	zmm1, zmm2, zmm3/m512	AVX10.2	EVEX
VFNMSUB213BF16	xmm1, xmm2, xmm3/m128	AVX10.2	EVEX
VFNMSUB213BF16	ymm1, ymm2, ymm3/m256	AVX10.2	EVEX
VFNMSUB213BF16	zmm1, zmm2, zmm3/m512	AVX10.2	EVEX
VFNMSUB231BF16	xmm1, xmm2, xmm3/m128	AVX10.2	EVEX
VFNMSUB231BF16	ymm1, ymm2, ymm3/m256	AVX10.2	EVEX
VFNMSUB231BF16	zmm1, zmm2, zmm3/m512	AVX10.2	EVEX
VFPCLASSBF16	k1, xmm2/m128, imm8	AVX10.2	EVEX
VFPCLASSBF16	k1, ymm2/m256, imm8	AVX10.2	EVEX
VFPCLASSBF16	k1, zmm2/m512, imm8	AVX10.2	EVEX
VGETEXPBF16	xmm1, xmm2/m128	AVX10.2	EVEX
VGETEXPBF16	ymm1, ymm2/m256	AVX10.2	EVEX
VGETEXPBF16	zmm1, zmm2/m512	AVX10.2	EVEX
VGETMANTBF16	xmm1, xmm2/m128, imm8	AVX10.2	EVEX
VGETMANTBF16	ymm1, ymm2/m256, imm8	AVX10.2	EVEX
VGETMANTBF16	zmm1, zmm2/m512, imm8	AVX10.2	EVEX
VMAXBF16	xmm1, xmm2, xmm3/m128	AVX10.2	EVEX
VMAXBF16	ymm1, ymm2, ymm3/m256	AVX10.2	EVEX
VMAXBF16	zmm1, zmm2, zmm3/m512	AVX10.2	EVEX
VMINBF16	xmm1, xmm2, xmm3/m128	AVX10.2	EVEX
VMINBF16	ymm1, ymm2, ymm3/m256	AVX10.2	EVEX
VMINBF16	zmm1, zmm2, zmm3/m512	AVX10.2	EVEX
VMINMAXBF16	xmm1, xmm2, xmm3/m128, imm8	AVX10.2	EVEX
VMINMAXBF16	ymm1, ymm2, ymm3/m256, imm8	AVX10.2	EVEX
VMINMAXBF16	zmm1, zmm2, zmm3/m512, imm8	AVX10.2	EVEX
VMOVRSB	xmm1, m128	AVX10-MOVRS	EVEX
VMOVRSB	ymm1, m256	AVX10-MOVRS	EVEX

VMOVRSB	zmm1, m512	AVX10-MOVRS	EVEX
VMOVRSD	xmm1, m128	AVX10-MOVRS	EVEX
VMOVRSD	ymm1, m256	AVX10-MOVRS	EVEX
VMOVRSD	zmm1, m512	AVX10-MOVRS	EVEX
VMOVRSQ	xmm1, m128	AVX10-MOVRS	EVEX
VMOVRSQ	ymm1, m256	AVX10-MOVRS	EVEX
VMOVRSQ	zmm1, m512	AVX10-MOVRS	EVEX
VMOVRSW	xmm1, m128	AVX10-MOVRS	EVEX
VMOVRSW	ymm1, m256	AVX10-MOVRS	EVEX
VMOVRSW	zmm1, m512	AVX10-MOVRS	EVEX
VMULBF16	xmm1, xmm2, xmm3/m128	AVX10.2	EVEX
VMULBF16	ymm1, ymm2, ymm3/m256	AVX10.2	EVEX
VMULBF16	zmm1, zmm2, zmm3/m512	AVX10.2	EVEX
VPDPBSSD	xmm1, xmm2, xmm3/m128	AVX10.2	EVEX
VPDPBSSD	ymm1, ymm2, ymm3/m256	AVX10.2	EVEX
VPDPBSSD	zmm1, zmm2, zmm3/m512	AVX10.2	EVEX
VPDPBSSDS	xmm1, xmm2, xmm3/m128	AVX10.2	EVEX
VPDPBSSDS	ymm1, ymm2, ymm3/m256	AVX10.2	EVEX
VPDPBSSDS	zmm1, zmm2, zmm3/m512	AVX10.2	EVEX
VPDPBSUD	xmm1, xmm2, xmm3/m128	AVX10.2	EVEX
VPDPBSUD	ymm1, ymm2, ymm3/m256	AVX10.2	EVEX
VPDPBSUD	zmm1, zmm2, zmm3/m512	AVX10.2	EVEX
VPDPBSUDS	xmm1, xmm2, xmm3/m128	AVX10.2	EVEX
VPDPBSUDS	ymm1, ymm2, ymm3/m256	AVX10.2	EVEX
VPDPBSUDS	zmm1, zmm2, zmm3/m512	AVX10.2	EVEX
VPDPBUUD	xmm1, xmm2, xmm3/m128	AVX10.2	EVEX
VPDPBUUD	ymm1, ymm2, ymm3/m256	AVX10.2	EVEX
VPDPBUUD	zmm1, zmm2, zmm3/m512	AVX10.2	EVEX
VPDPBUUDS	xmm1, xmm2, xmm3/m128	AVX10.2	EVEX
VPDPBUUDS	ymm1, ymm2, ymm3/m256	AVX10.2	EVEX
VPDPBUUDS	zmm1, zmm2, zmm3/m512	AVX10.2	EVEX
VPDPWSUD	xmm1, xmm2, xmm3/m128	AVX10.2	EVEX
VPDPWSUD	ymm1, ymm2, ymm3/m256	AVX10.2	EVEX
VPDPWSUD	zmm1, zmm2, zmm3/m512	AVX10.2	EVEX
VPDPWSUDS	xmm1, xmm2, xmm3/m128	AVX10.2	EVEX
VPDPWSUDS	ymm1, ymm2, ymm3/m256	AVX10.2	EVEX
VPDPWSUDS	zmm1, zmm2, zmm3/m512	AVX10.2	EVEX
VPDPWUSD	xmm1, xmm2, xmm3/m128	AVX10.2	EVEX
VPDPWUSD	ymm1, ymm2, ymm3/m256	AVX10.2	EVEX
VPDPWUSD	zmm1, zmm2, zmm3/m512	AVX10.2	EVEX
VPDPWUSDS	xmm1, xmm2, xmm3/m128	AVX10.2	EVEX
VPDPWUSDS	ymm1, ymm2, ymm3/m256	AVX10.2	EVEX
VPDPWUSDS	zmm1, zmm2, zmm3/m512	AVX10.2	EVEX
VPDPWUUD	xmm1, xmm2, xmm3/m128	AVX10.2	EVEX
VPDPWUUD	ymm1, ymm2, ymm3/m256	AVX10.2	EVEX
VPDPWUUD	zmm1, zmm2, zmm3/m512	AVX10.2	EVEX
VPDPWUUDS	xmm1, xmm2, xmm3/m128	AVX10.2	EVEX

VPDPWUUDS	ymm1, ymm2, ymm3/m256	AVX10.2	EVEX
VPDPWUUDS	zmm1, zmm2, zmm3/m512	AVX10.2	EVEX
VRCPBF16	xmm1, xmm2/m128	AVX10.2	EVEX
VRCPBF16	ymm1, ymm2/m256	AVX10.2	EVEX
VRCPBF16	zmm1, zmm2/m512	AVX10.2	EVEX
VREDUCEBF16	xmm1, xmm2/m128, imm8	AVX10.2	EVEX
VREDUCEBF16	ymm1, ymm2/m256, imm8	AVX10.2	EVEX
VREDUCEBF16	zmm1, zmm2/m512, imm8	AVX10.2	EVEX
VRNDSCALEBF16	xmm1, xmm2/m128, imm8	AVX10.2	EVEX
VRNDSCALEBF16	ymm1, ymm2/m256, imm8	AVX10.2	EVEX
VRNDSCALEBF16	zmm1, zmm2/m512, imm8	AVX10.2	EVEX
VRSQRTBF16	xmm1, xmm2/m128	AVX10.2	EVEX
VRSQRTBF16	ymm1, ymm2/m256	AVX10.2	EVEX
VRSQRTBF16	zmm1, zmm2/m512	AVX10.2	EVEX
VSCALEFBF16	xmm1, xmm2, xmm3/m128	AVX10.2	EVEX
VSCALEFBF16	ymm1, ymm2, ymm3/m256	AVX10.2	EVEX
VSCALEFBF16	zmm1, zmm2, zmm3/m512	AVX10.2	EVEX
VSQRTBF16	xmm1, xmm2/m128	AVX10.2	EVEX
VSQRTBF16	ymm1, ymm2/m256	AVX10.2	EVEX
VSQRTBF16	zmm1, zmm2/m512	AVX10.2	EVEX
VSUBBF16	xmm1, xmm2, xmm3/m128	AVX10.2	EVEX
VSUBBF16	ymm1, ymm2, ymm3/m256	AVX10.2	EVEX
VSUBBF16	zmm1, zmm2, zmm3/m512	AVX10.2	EVEX
Type E4NF			
VCVT2PH2BF8	xmm1, xmm2, xmm3/m128	AVX10.2	EVEX
VCVT2PH2BF8	ymm1, ymm2, ymm3/m256	AVX10.2	EVEX
VCVT2PH2BF8	zmm1, zmm2, zmm3/m512	AVX10.2	EVEX
VCVT2PH2BF8S	xmm1, xmm2, xmm3/m128	AVX10.2	EVEX
VCVT2PH2BF8S	ymm1, ymm2, ymm3/m256	AVX10.2	EVEX
VCVT2PH2BF8S	zmm1, zmm2, zmm3/m512	AVX10.2	EVEX
VCVT2PH2HF8	xmm1, xmm2, xmm3/m128	AVX10.2	EVEX
VCVT2PH2HF8	ymm1, ymm2, ymm3/m256	AVX10.2	EVEX
VCVT2PH2HF8	zmm1, zmm2, zmm3/m512	AVX10.2	EVEX
VCVT2PH2HF8S	xmm1, xmm2, xmm3/m128	AVX10.2	EVEX
VCVT2PH2HF8S	ymm1, ymm2, ymm3/m256	AVX10.2	EVEX
VCVT2PH2HF8S	zmm1, zmm2, zmm3/m512	AVX10.2	EVEX
VCVT2PS2PHX	xmm1, xmm2, xmm3/m128	AVX10.2	EVEX
VCVT2PS2PHX	ymm1, ymm2, ymm3/m256	AVX10.2	EVEX
VCVT2PS2PHX	zmm1, zmm2, zmm3/m512	AVX10.2	EVEX
VCVTBIASPH2BF8	xmm1, xmm2, xmm3/m128	AVX10.2	EVEX
VCVTBIASPH2BF8	xmm1, ymm2, ymm3/m256	AVX10.2	EVEX
VCVTBIASPH2BF8	ymm1, zmm2, zmm3/m512	AVX10.2	EVEX
VCVTBIASPH2BF8S	xmm1, xmm2, xmm3/m128	AVX10.2	EVEX
VCVTBIASPH2BF8S	xmm1, ymm2, ymm3/m256	AVX10.2	EVEX
VCVTBIASPH2BF8S	ymm1, zmm2, zmm3/m512	AVX10.2	EVEX
VCVTBIASPH2HF8	xmm1, xmm2, xmm3/m128	AVX10.2	EVEX

VCVTBIASPH2HF8	xmm1, ymm2, ymm3/m256	AVX10.2	EVEX
VCVTBIASPH2HF8	ymm1, zmm2, zmm3/m512	AVX10.2	EVEX
VCVTBIASPH2HF8S	xmm1, xmm2, xmm3/m128	AVX10.2	EVEX
VCVTBIASPH2HF8S	xmm1, ymm2, ymm3/m256	AVX10.2	EVEX
VCVTBIASPH2HF8S	ymm1, zmm2, zmm3/m512	AVX10.2	EVEX
VCVTPH2BF8	xmm1, xmm2/m128	AVX10.2	EVEX
VCVTPH2BF8	xmm1, ymm2/m256	AVX10.2	EVEX
VCVTPH2BF8	ymm1, zmm2/m512	AVX10.2	EVEX
VCVTPH2BF8S	xmm1, xmm2/m128	AVX10.2	EVEX
VCVTPH2BF8S	xmm1, ymm2/m256	AVX10.2	EVEX
VCVTPH2BF8S	ymm1, zmm2/m512	AVX10.2	EVEX
VCVTPH2HF8	xmm1, xmm2/m128	AVX10.2	EVEX
VCVTPH2HF8	xmm1, ymm2/m256	AVX10.2	EVEX
VCVTPH2HF8	ymm1, zmm2/m512	AVX10.2	EVEX
VCVTPH2HF8S	xmm1, xmm2/m128	AVX10.2	EVEX
VCVTPH2HF8S	xmm1, ymm2/m256	AVX10.2	EVEX
VCVTPH2HF8S	ymm1, zmm2/m512	AVX10.2	EVEX
VMPSADBW	xmm1, xmm2, xmm3/m128, imm8	AVX10.2	EVEX
VMPSADBW	ymm1, ymm2, ymm3/m256, imm8	AVX10.2	EVEX
VMPSADBW	zmm1, zmm2, zmm3/m512, imm8	AVX10.2	EVEX
Type E6			
VSM4KEY4	xmm1, xmm2, xmm3/m128	AVX10.2,SM4	EVEX
VSM4KEY4	ymm1, ymm2, ymm3/m256	AVX10.2,SM4	EVEX
VSM4KEY4	zmm1, zmm2, zmm3/m512	AVX10.2,SM4	EVEX
VSM4RNDS4	xmm1, xmm2, xmm3/m128	AVX10.2,SM4	EVEX
VSM4RNDS4	ymm1, ymm2, ymm3/m256	AVX10.2,SM4	EVEX
VSM4RNDS4	zmm1, zmm2, zmm3/m512	AVX10.2,SM4	EVEX
Type E9NF			
VMOVD	xmm1, xmm2/m32	AVX10.2	EVEX
VMOVD	xmm1/m32, xmm2	AVX10.2	EVEX
VMOVW	xmm1, xmm2/m16	AVX10.2	EVEX
VMOVW	xmm1/m16, xmm2	AVX10.2	EVEX

Table 4.1: Exception Class Summary for all Instructions

4.2 EXCEPTION CLASS SUMMARY

The following descriptions contain tabular summaries of the behavior of each exception class across the operating modes of Intel® processors.

Notations and abbreviations include:

- CM: Compatibility Mode
- PM: Protected Mode

4.2.1 EXCEPTION CLASS AMX-E7-EVEX

AMX-E7-EVEX	• #UD if preceded by LOCK, 66h, F2h, F3h or REX prefixes • #UD if CR4.OSXSAVE != 1 • #UD if XCR0[18:17] != 0b11 • #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 is not a valid tile (i.e. not configured) • #UD if tsrc is not a valid tile name for configured palette • #UD if tsrc.colsb % 4 != 0 • #UD if EVEX.z != 0b0 // P2[7] • #UD if EVEX.LL' != 0b10 // P2[6:5] • #UD if EVEX.b != 0b0 // P2[4] • #UD if EVEX.aaa != 0b000 // P2[2:0] • #UD if EVEX.u != 0b1 // P1[2] • #UD if EVEX.V' != 0b1 // P2[3] • #UD if EVEX.VVVV != 0b1111 // P1[6:3] • #NM if CR0[3] == 1 // TS • #NM if XFD[18] == 1 • #GP if imm8&0x3f >= tsrc.rows • #GP if (imm8»6) * VL.bytes >= tsrc.colsb

Table 4.2: Type AMX-E7-EVEX Class Exception Conditions

4.2.2 EXCEPTION CLASS AMX-E8-EVEX

AMX-E8-EVEX	• #UD if preceded by LOCK, 66h, F2h, F3h or REX prefixes • #UD if CR4.OSXSAVE != 1 • #UD if XCR0[18:17] != 0b11 • #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 is not a valid tile (i.e. not configured) • #UD if tsrc is not a valid tile name for configured palette • #UD if tsrc.colsb % 4 != 0 • #UD if EVEX.z != 0b0 // P2[7] • #UD if EVEX.LL' != 0b10 // P2[6:5] • #UD if EVEX.b != 0b0 // P2[4] • #UD if EVEX.aaa != 0b000 // P2[2:0] • #UD if EVEX.u != 0b1 // P1[2] • #UD if EVEX.V' != 0b1 // P2[3] • #NM if CR0[3] == 1. // TS • #NM if XFD[18] == 1 • #GP if r32&0xffff >= tsrc.rows • #GP if (((r32»16) & 0xffff) * VL.bytes) >= tsrc.colsb

Table 4.3: Type AMX-E8-EVEX Class Exception Conditions

4.2.3 EXCEPTION CLASS E10NF

EVEX-encoded scalar instructions that ignore EVEX.L'L vector length encoding, that cause no SIMD FP exceptions, and do not support memory fault suppression follow exception class E10NF.

Exception	Real	Virtual 8086	PM & CM	64-bit	Cause of exception
Invalid Opcode, #UD	X	X			EVEX prefix is present.
			X	X	If CR4.OSXSAVE[bit 18] = 0. If any of the following conditions applies • State requirements not met. • Opcode independent #UD conditions not met. • Operand encoding #UD conditions not met. • Opmask encoding #UD conditions not met. • EVEX.b encoding #UD conditions not met and in E4.nb sub-class.
	X	X	X	X	If preceded by a LOCK prefix (F0H)
			X	X	If any REX, F2, F3 or 66 prefixes precede a VEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 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		For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments. If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
				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(faultcode)		X	X	X	For 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 CPL=3

Table 4.4: Type E10NF Class Exception Conditions

4.2.4 EXCEPTION CLASS E11

EVEX-encoded instructions that can cause SIMD FP exception, memory operand fault suppression, but do not cause #AC follow exception class E11.

Exception	Real	Virtual 8086	PM & CM	64-bit	Cause of exception
Invalid Opcode, #UD	X	X			EVEX prefix is present.
			X	X	If CR4.OSXSAVE[bit 18] = 0. If any of the following conditions applies • State requirements not met. • Opcode independent #UD conditions not met. • Operand encoding #UD conditions not met. • Opmask encoding #UD conditions not met. • EVEX.b encoding #UD conditions not met. • Instruction specific EVEX.L'L restriction not met.
	X	X	X	X	If preceded by a LOCK prefix (F0H)
			X	X	If any REX, F2, F3 or 66 prefixes precede a VEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 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. If fault suppression not set, and the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
				X	If fault supression 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(faultcode)		X	X	X	If fault suppression not set, and a page fault.
SIMD Floating-point Exception, #XM	X	X	X	X	If an unmasked SIMD floating-point exception, sae not set, and CR4.OSXMMEXCPT[bit 10] = 1

Table 4.5: Type E11 Class Exception Conditions

4.2.5 EXCEPTION CLASS E2

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

Exception	Real	Virtual 8086	PM & CM	64-bit	Cause of exception
Invalid Opcode, #UD	X	X			EVEX prefix is present.
	X	X	X	X	If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0
			X	X	If CR4.OSXSAVE[bit 18] = 0. If any of the following conditions applies • State requirements not met. • Opcode independent #UD conditions not met. • Operand encoding #UD conditions not met. • Opmask encoding #UD conditions not met. • EVEX.b encoding #UD conditions not met. • Instruction specific EVEX.L'L restriction not met.
	X	X	X	X	If preceded by a LOCK prefix (F0H)
			X	X	If any REX, F2, F3 or 66 prefixes precede a VEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 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. If fault suppression not set, and the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
				X	If fault supression 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(faultcode)		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 CPL=3
SIMD Floating-point Exception, #XM	X	X	X	X	If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1

Table 4.6: Type E2 Class Exception Conditions

4.2.6 EXCEPTION CLASS E3

EVEX-encoded scalar instructions with arithmetic semantics that support memory fault suppression follow exception class E3.

Exception	Real	Virtual 8086	PM & CM	64-bit	Cause of exception
Invalid Opcode, #UD	X	X			EVEX prefix is present.
	X	X	X	X	If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0
			X	X	If CR4.OSXSAVE[bit 18] = 0. If any of the following conditions applies • State requirements not met. • Opcode independent #UD conditions not met. • Operand encoding #UD conditions not met. • Opmask encoding #UD conditions not met. • EVEX.b encoding #UD conditions not met.
	X	X	X	X	If preceded by a LOCK prefix (F0H)
			X	X	If any REX, F2, F3 or 66 prefixes precede a VEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 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. If fault suppression not set, and the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
				X	If fault supression 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(faultcode)		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 CPL=3
SIMD Floating-point Exception, #XM	X	X	X	X	If an unmasked SIMD floating-point exception, sae or er not set, and CR4.OSXMMEXCPT[bit 10] = 1

Table 4.7: Type E3 Class Exception Conditions

4.2.7 EXCEPTION CLASS E3NF

EVEX-encoded scalar instructions with arithmetic semantics that do not support memory fault suppression follow exception class E3NF.

Exception	Real	Virtual 8086	PM & CM	64-bit	Cause of exception
Invalid Opcode, #UD	X	X			EVEX prefix is present.
	X	X	X	X	If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0
			X	X	If CR4.OSXSAVE[bit 18] = 0. If any of the following conditions applies • State requirements not met. • Opcode independent #UD conditions not met. • Operand encoding #UD conditions not met. • Opmask encoding #UD conditions not met. • EVEX.b encoding #UD conditions not met.
	X	X	X	X	If preceded by a LOCK prefix (F0H)
			X	X	If any REX, F2, F3 or 66 prefixes precede a VEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 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		For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments. If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
				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(faultcode)		X	X	X	For 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 CPL=3
SIMD Floating-point Exception, #XM	X	X	X	X	If an unmasked SIMD floating-point exception, sae or er not set, and CR4.OSXMMEXCPT[bit 10] = 1

Table 4.8: Type E3NF Class Exception Conditions

4.2.8 EXCEPTION CLASS E4

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

Exception	Real	Virtual 8086	PM & CM	64-bit	Cause of exception
Invalid Opcode, #UD	X	X			EVEX prefix is present.
			X	X	If CR4.OSXSAVE[bit 18] = 0. If any of the following conditions applies • State requirements not met. • Opcode independent #UD conditions not met. • Operand encoding #UD conditions not met. • Opmask encoding #UD conditions not met. • EVEX.b encoding #UD conditions not met and in E4.nb subclass. • Instruction specific EVEX.L'L restriction not met
	X	X	X	X	If preceded by a LOCK prefix (F0H)
			X	X	If any REX, F2, F3 or 66 prefixes precede a VEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 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. If fault suppression not set, and the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
				X	If fault supression 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(faultcode)		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 CPL=3

Table 4.9: Type E4 Class Exception Conditions

4.2.9 EXCEPTION CLASS E4NF

EVEX-encoded scalar instructions that cause no SIMD FP exceptions, and that do not support memory fault suppression follow exception class E4NF.

Exception	Real	Virtual 8086	PM & CM	64-bit	Cause of exception
Invalid Opcode, #UD	X	X			EVEX prefix is present.
			X	X	If CR4.OSXSAVE[bit 18] = 0. If any of the following conditions applies • State requirements not met. • Opcode independent #UD conditions not met. • Operand encoding #UD conditions not met. • Opmask encoding #UD conditions not met. • EVEX.b encoding #UD conditions not met and in E4.nb sub-class. • Instruction specific EVEX.L'L restriction not met
	X	X	X	X	If preceded by a LOCK prefix (F0H)
			X	X	If any REX, F2, F3 or 66 prefixes precede a VEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 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		For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments. If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
				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(faultcode)		X	X	X	For 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 CPL=3

Table 4.10: Type E4NF Class Exception Conditions

4.2.10 EXCEPTION CLASS E6

EVEX-encoded instructions that cause no SIMD FP exceptions, and that support memory fault suppression follow exception class E6.

Exception	Real	Virtual 8086	PM & CM	64-bit	Cause of exception
Invalid Opcode, #UD	X	X			EVEX prefix is present.
			X	X	If CR4.OSXSAVE[bit 18] = 0. If any of the following conditions applies • State requirements not met. • Opcode independent #UD conditions not met. • Operand encoding #UD conditions not met. • Opmask encoding #UD conditions not met. • EVEX.b encoding #UD conditions not met. • Instruction specific EVEX.L'L restriction not met
	X	X	X	X	If preceded by a LOCK prefix (F0H)
			X	X	If any REX, F2, F3 or 66 prefixes precede a VEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 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. If fault suppression not set, and the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
				X	If fault supression 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(faultcode)		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 CPL=3

Table 4.11: Type E6 Class Exception Conditions

4.2.11 EXCEPTION CLASS E9NF

EVEX-encoded vector or partial-vector instructions that cause no SIMD FP exceptions, and do not support memory fault suppression follow exception class E9NF.

Exception	Real	Virtual 8086	PM & CM	64-bit	Cause of exception
Invalid Opcode, #UD	X	X			EVEX prefix is present.
			X	X	If CR4.OSXSAVE[bit 18] = 0. If any of the following conditions applies • State requirements not met. • Opcode independent #UD conditions not met. • Operand encoding #UD conditions not met. • Opmask encoding #UD conditions not met. • EVEX.b encoding #UD conditions not met and in E4.nb sub-class. • Instruction specific EVEX.L'L restriction not met
	X	X	X	X	If preceded by a LOCK prefix (F0H)
			X	X	If any REX, F2, F3 or 66 prefixes precede a VEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 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		For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments. If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
				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(faultcode)		X	X	X	For 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 CPL=3

Table 4.12: Type E9NF Class Exception Conditions

Chapter 5

HELPER FUNCTIONS

```
define convert_hf8_to_fp16( in ):
  fp16_bias = 15
  hf8_bias = 7
  s = ( in & 0x80 ) << 8
  e = ( in & 0x78 ) >> 3
  m = ( in & 0x07 )
  e_norm = e + (fp16_bias - hf8_bias)

  /* convert denormal hf8 number into a normal fp16 number */
  if ( (e == 0) && (m != 0) ):
    lz_cnt = 2
    lz_cnt = ( m > 0x1 ) ? 1 : lz_cnt
    lz_cnt = ( m > 0x3 ) ? 0 : lz_cnt
    e_norm -= lz_cnt
    m = (m << (lz_cnt+1)) & 0x07
  else if ( (e == 0) && (m == 0) ):
    e_norm = 0
  else if ( (e == 0xf) && (m == 0x7) ):
    e_norm = 0x1f

  /* set result */
  res = 0x0
  res |= (e_norm << 10)
  res |= (m << 7)
  res |= s

  return res
```

```
define convert_fp16_to_bf8(x,s):
// The x parameter is the data
// The s parameter indicates whether we saturate in case of
// overflow due to conversion or rounding

if *x is infinity*
        if (s==0x1):                    // Max Value
                dest[7] := x[15]
                dest[6:0] := 0x7B
        else:                                   // INF
                dest[7:0] := x[15:8]

else if *x is nan*:
        dest[7:0] := x[15:8]                    // truncate and set QNaN
        dest[1] := 1

else // normal, zero or denormal number apply RNE
        lsb := x[8]
        rounding_bias[15:0] := 0x007F + lsb
        temp[15:0] := x[15:0] + rounding_bias[15:0]        // temp is rounded data (RNE)

        if temp [14:8] == 0x7C && s==1   // saturating to E5M2_MAX due to infinity result
                dest[7] := temp[15]
                dest[6:0] := 0x7B
        else
                dest[7:0] := temp[15:8]
return dest


```

```
define convert_fp16_to_hf8(x, s):
// The x parameter is the data
// The s parameter indicates whether we saturate in case of
// overflow due to conversion or rounding
// Round mode RNE
fp16_bias := 15
hf8_bias := 7
fp16_to_hf8_exp_rebias = fp16_bias - hf8_bias


sign := ( x & 0x8000 ) >> 8
e_fp16 := ( x & 0x7c00 ) >> 10
m_fp16 := ( x & 0x03ff )

if *x is infinity*
        e := 0xF
```

```
        if (s==0x1):                    // Max Value
                m := 0x6
        else:                                   // NaN
                m := 0x7

else if *x is nan*:
        e := 0xF
        m := 0x7

/* overflow --> make it NaN or Saturate to E4M3_MAX*/
else if ((e_fp16 > (fp16_to_hf8_exp_rebias + 15)) ||
           ((e_fp16 == (fp16_to_hf8_exp_rebias + 15)) && (m_fp16 > 0x0340))):
        e := 0xf
        if ( s == 0x1 ):
                m := 0x6
        else:
                m := 0x7

/* Zero */
else if ((e_fp16 ==0) & (m_fp16==0)):
        e := 0
        m := 0

/* underflow */
else if ( e_fp16 <= fp16_to_hf8_exp_rebias ):
/* denormalized mantissa */
        /* set Jbit */
        m := m_fp16 | 0x0400

        /* addtionally subnormal shift */
        m = m >> ((fp16_to_hf8_exp_rebias) + 1 - e_fp16)

        /* preserve sticky bit (some sticky bits are lost when denormalizing) */
        ShiftOutSticky = (((m_fp16 & 0x007f) + 0x007f) >> 7)
        m = m | ShiftOutSticky                // OR m.lsb with sticky

        /* RNE Round */
        fixup := (m >> 7) & 0x1;           // get Lbit
        m := m + 0x003f + fixup;          // RNE

        // in case of round overflow, m>>10 declares the carry into exponent
        e := m>>10

        m := (m >> 7) & 0x7 //  Truncate Round and ignore carry


/* normal */
else:
```

```
        /* RNE round */
        fixup := (m_fp16 >> 7) & 0x1;
        RneX := x + 0x003f + fixup;
        e := ((RneX & 0x7c00) >> 10);
        m := ( RneX & 0x03ff );
        e = e - (fp16_to_hf8_exp_rebias);
        m := m >> 7;

res = 0x0
res |= e << 3 // set exp
res |= m        //set mant
res |= sign    // set sign

return res

```

```
define convert_fp16_to_fp8(x,y,s):
// The x parameter is the data
// The y parameter is the destination format indicating bfloat8 or hfloat8.
// The s parameter indicates whether we saturate in case of overflow due to
// conversion or rounding
  if *y is bf8*:
    return convert_fp16_to_bf8(x,s)
  else:
    return convert_fp16_to_hf8(x,s)
```

```
define convert_fp16_to_hf8_bias(x, b, s):
    // The x parameter is the data
    // The b parameter is the bias 8b integer to be added to the data for RS
    // The s parameter indicates whether we saturate in case of overflow due to
    // conversion or rounding

        fp16_bias := 15
        hf8_bias := 7
        fp16_to_hf8_exp_rebias = fp16_bias- hf8_bias

        // extract original sign, mantissa and exponent
        sign := ( x & 0x8000 ) >> 8
        e_fp16 := ( x & 0x7c00 ) >> 10
        m_fp16 := ( x & 0x03ff )

        // extract biased mantissa and biased exponent
        x_bias := x + ( b >> 1 )
        e_fp16_bias := ( x_bias & 0x7c00 ) >> 10
        m_fp16_bias := ( x_bias & 0x03ff )

```

```
21      if *x is infinity*
22              e := 0xF
23              if (s==0x1):                    // Max Value
24                      m := 0x6
25              else:                                   // NaN
26                      m := 0x7
27
28      else if *x is nan*:
29              e := 0xF
30              m := 0x7
31
32      /* overflow --> make it HF8 NaN/Inf == s.1111.111 */
33      /* or Saturate to E4M3_MAX == s.1111.110 */
34      else if ((e_fp16_bias > (fp16_to_hf8_exp_rebias + 15)) ||
35                       ((e_fp16_bias == (fp16_to_hf8_exp_rebias + 15)) &&
36                       ( m_fp16_bias >= 0x0380))) then:
37              e = 0xf
38              if ( s == 0x1 ) then:
39                      m = 0x6
40              else:
41                      m = 0x7
42
43      // input denormal (or zero)
44      // in this case the bias rounding will end up with min-denormal or 0
45
46      else if (e_fp16 == 0x0):
47              m = m_fp16+(b<<7)           // align bias
48              m = m >> (fp16_to_hf8_exp_rebias) + 7 // align mantissa 8+7
49              e = 0x0;                  // set exp to zero
50
51      /* underflow*/
52      // underflow happens if e_fp16_bias is too small for representing in
53      // the destination format in this case the Jbit should be set and bias
54      // rounding should be done after aligning to the destination format
55      else if ( e_fp16_bias <= fp16_to_hf8_exp_rebias ):
56              /* set Jbit */
57              m = m_fp16 | 0x0400
58
59              // m += aligned b to mantissa
60              m = m +( b<< (fp16_to_hf8_exp_rebias - e_fp16))
61
62              // now mantissa is aligned to destination exponent + subnormal shift
63              m = m >> ((fp16_to_hf8_exp_rebias) + 1 - e_fp16)
64
65              // in case of round overflow, m>>10 declares the carry into exponent
66              e = m >> 10;          // e=1 in case of overflow
67
68              /* Truncate Round and ignore carry*/
```

```
                m = (m >> 7) & 0x7;

        /* normal */
        else then:
                /* Stochastic Round by truncating */
                e = ( x_bias & 0x7c00 ) >> 10
                e -= (fp16_to_hf8_exp_rebias)
                m = ( x_bias & 0x03ff )
                m = m >> 7

        res = 0x0
        res |= e << 3 // set exp
        res |= m        //set mant
        res |= sign    // set sign

return dest
```

```
define convert_fp16_to_bf8_bias(x, b, s):
// The x parameter is the data
// The b parameter is the bias 8b integer to be added to
// the data before the downconvert
// The s parameter indicates whether we saturate in case of
// overflow due to conversion or rounding

if *x is infinity*
        if (s==0x1):                            // Max Value
                dest[7] := x[15]
                dest[6:0] := 0x7B
        else:                                  // INF
                dest[7:0] := x[15:8]

else if *x is nan*:
        dest[7:0] := x[15:8]                // truncate and set QNaN
        dest[1] := 1

else // normal, denormal or zero input operand apply RS
        rounding_bias[15:8] := 0
        rounding_bias[7:0] := b[7:0]
        temp[15:0] := x[15:0] + rounding_bias   // temp is rounded data (RS)
        // saturating to E5M2_MAX in case of infinity result
        if temp [14:8] == 0x7C && s==1
                dest[7] := temp[15]
                dest[6:0] := 0x7B
        else
                dest[7:0] := temp[15:8]
return dest

```

```

```

```
define convert_fp16_to_fp8_bias(x, b, y, s):
// The x parameter is the data
// The b parameter is the bias
// The y parameter is the destination format indicating bfloat8 or hfloat8
// The s parameter indicating saturation
  if *y is bf8*:
      return convert_fp16_to_bf8_bias(x, b, s)
  else:
      return convert_fp16_to_hf8_bias(x, b, s)
```

```
define convert_fp32_to_fp16(x):

// The x parameter is the data
// rm=MXCSR.RC or embedded.RC
Fp32_bias := 127
Fp16_bias := 15

sign := ( x & 0x80000000 ) >> 16
e_fp32 := ( x & 0x7F800000 ) >> 23
m_fp32 := ( x & 0x007FFFFF)

m_fp16 = m_fp32>>13
e_fp16 = e_fp32 - fp32_bias + fp16_bias

if (e_fp32==0xFF) && (m_fp32==0x000000) // *x is infinity*
        e_fp16 := 0x1F
        m_fp16 := 0x000

else if (e_fp32==0xFF) && (m_fp32 != 0x000000) // *x is nan*:
        e_fp16 := 0x1F
        m_fp16 := (m_fp32 | 0x400000) >> 13

/* Zero */
else if (e_fp32==0x00) && (DAZ || (m_fp32 == 0x000000)) // *x is zero*:
        e_fp16 := 0x00
        m_fp16 := 0x000

else if (e_fp32==0x00) && (m_fp32 != 0x000000) // *x is denormal*:
        // check if result is zero or min-denormal
        RoundAdd1 := (sign) ? (rm==RDN) : (rm==RUP)
        e_fp16 := 0x00
        m_fp16 := 0x000 + RoundAdd1

/* underflow */
```

```
else if ( e_fp32 <= fp32_bias - fp16_bias ):
/* denormalized mantissa */
        /* set Jbit */
        m := (m_fp32 | 0x800000);

       /* Calculate shift out sticky */
        ShiftOutGbitMask = (0x1 << ((fp32_bias - fp16_bias) - e_fp32));
        ShiftOutStickyMask = ShiftOutGbitMask - 1;

        ShiftOutSticky = OR (m & ShiftOutStickyMask);
        ShiftOutGbit = OR (m & ShiftOutGbitMask);

        /* denormalization */
        m := m >> (fp32_bias - fp16_bias) - e_fp32 + 1;
        Lbit = m & 1;

        /* Rounding */
        RoundAdd1 = (~sign && (rm==RUP) && (ShiftOutGbit | ShiftOutSticky))  ||
                    (sign && (rm==RDN) && (ShiftOutGbit | ShiftOutSticky)) ||
                    (rm==RNE && ShiftOutGbit &(Lbit | ShiftOutSticky));
        m = m + RoundAdd1;

        m_fp16 = m & 0x3FF;
        e_fp16 = (m>>10) & 1;

else
        /* normal round parameters */
        Lbit1 = (m_fp32 & 0x2000) >> 13
        Gbit1 = (m_fp32 & 0x1000) >> 12
        Stky1 = OR(m_fp32 & 0x0FFF)
        RoundAdd1 := (~sign && (rm==RUP) && (Gbit1 | Stky1))  ||
                     (sign && (rm==RDN) && (Gbit1 | Stky1)) ||
                     (rm==RNE && Gbit1 &(Lbit1 | Stky1));

        /* overflow --> make it INF (depending on round mode rm) */
        if (e_fp32   (fp32_bias - fp16_bias + 31) ||
        (e_fp32 == (fp32_bias - fp16_bias + 30)) && (m_fp32 > 0x7FE000) && RoundAdd1:
                e_fp16 := 0x1F
                m_fp16 := 0x000

        else /* normal */
                m = (m_fp32 & 0x7FE000) >> 13;
                m = m + RoundAdd1;
                m_fp16 = m & 0x3FF;
                e0 = (m>>10) & 1;
                e_fp16 = e_fp32 - fp32_bias + fp16_bias + e0;
```

```
83 /* set result */
84 res := 0x0
85 res |= e_fp16 << 10
86 res |= m_fp16
87 res |= sign
88 return res
```

Figure 5.1: minimum

```
def minimum(a,b):
    if a is SNAN or (a is QNAN and b is not SNAN):
        return QNAN(a)
    else if b is NAN:
        return QNAN(b)
    else if (a == +0.0 and b == -0.0) or (a == -0.0 and b == +0.0):
        return -0.0
    else if a <= b:
        return a
    else:
        return b
```

Figure 5.2: minimumNumber

```
def minimum_number(a,b):
    if a is NAN and B is NAN:
        if a is SNAN or (a is QNAN and b is QNAN):
            return QNAN(a)
        else: // a is QNAN and b is SNAN
            return QNAN(b)
    else if a is NAN:
        return b
    else if b is NAN:
        return a
    else if (a == +0.0 and b == -0.0) or (a == -0.0 and b == +0.0):
        return -0.0
    else if a <= b:
        return a
    else:
        return b
```

Figure 5.3: minimumMagnitude

```
def minimum_magnitude(a,b):
    if a is SNAN or (a is QNAN and b is not SNAN):
        return QNAN(a)
    else if b is NAN:
        return QNAN(b)
    else if abs(a) < abs(b):
        return a
    else if abs(b) < abs(a):
        return b
    else:
        return minimum(a,b)
```

Figure 5.4: minimumMagnitudeNumber

```
def minimum_magnitude_number(a,b):
    if a is NAN and B is NAN:
        if a is SNAN or (a is QNAN and b is QNAN):
            return QNAN(a)
        else: // a is QNAN and b is SNAN
            return QNAN(b)
    else if a is NAN:
        return b
    else if b is NAN:
        return a
    else if abs(a) < abs(b):
        return a
    else if abs(b) < abs(a):
        return b
    else:
        return minimum_number(a,b)
```

Figure 5.5: maximum

```
def maximum(a,b):
    if a is SNAN or (a is QNAN and b is not SNAN):
        return QNAN(a)
    else if b is NAN:
        return QNAN(b)
    else if (a == +0.0 and b == -0.0) or (a == -0.0 and b == +0.0):
        return +0.0
    else if a >= b:
        return a
    else:
        return b
```

Figure 5.6: maximumNumber

```
def maximum_number(a,b):
    if a is NAN and B is NAN:
        if a is SNAN or (a is QNAN and b is QNAN):
            return QNAN(a)
        else: // a is QNAN and b is SNAN
            return QNAN(b)
    else if a is NAN:
        return b
    else if b is NAN:
        return a
    else if (a == +0.0 and b == -0.0) or (a == -0.0 and b == +0.0):
        return +0.0
    else if a >= b:
        return a
    else:
        return b
```

Figure 5.7: maximumMagnitude

```
def maximum_magnitude(a,b):
    if a is SNAN or (a is QNAN and b is not SNAN):
        return QNAN(a)
    else if b is NAN:
        return QNAN(b)
    else if abs(a) > abs(b):
        return a
    else if abs(b) > abs(a):
        return b
    else:
        return maximum(a,b)
```

Figure 5.8: maximumMagnitudeNumber

```
def maximum_magnitude_number(a,b):
    if a is NAN and B is NAN:
        if a is SNAN or (a is QNAN and b is QNAN):
            return QNAN(a)
        else: // a is QNAN and b is SNAN
            return QNAN(b)
    else if a is NAN:
        return b
    else if b is NAN:
        return a
    else if abs(a) > abs(b):
        return a
    else if abs(b) > abs(a):
        return b
    else:
        return maximum_number(a,b)
```

Figure 5.9: minmax

```
def minmax(a,b,imm,daz,except):
    op_select := imm[1:0]; sign_control := imm[3:2]; nan_prop_select := imm[4]

    if daz == true:
        if a is denormal:
            a.fraction := 0
        if b is denormal:
            b.fraction := 0
    if except == true:
        if a is SNAN or b is SNAN:
            set_MXCSR(IE)
        else if a is QNAN or b is QNAN:
            // QNAN prevents lower-priority exceptions (SDM Vol.3A Table 6-8)
        else if a is denormal or b is denormal:
            set_MXCSR(DE)

    if nan_prop_select == 0: //propagate NaNs
        if op_select == 0:
            tmp := minimum(a,b)
        else if op_select == 1:
            tmp := maximum(a,b)
        else if op_select == 2:
            tmp := minimum_magnitude(a,b)
        else: //op_select == 3
            tmp := maximum_magnitude(a,b)
    else: //do not propagate NaNs
        if op_select == 0:
            tmp := minimum_number(a,b)
        else if op_select == 1:
            tmp := maximum_number(a,b)
        else if op_select == 2:
            tmp := minimum_magnitude_number(a,b)
        else: //op_select == 3
            tmp := maximum_magnitude_number(a,b)

    if tmp is not NAN:
        if (sign_control == 3):
            tmp.sign := 1
        else if (sign_control == 2):
            tmp.sign := 0
        else if (sign_control == 1) or (a is NAN):
            // Keep sign of comparison result, i.e. tmp.sign is un-changed
        else: // sign_control == 0
            tmp.sign := a.sign
    return tmp
```

```
define convert_bf16_to_signed_byte_rne_saturate(src.bf16):
        /* VCVTBF162IBS converts brain-float16 floating point elements into
        signed byte integer elements.  When a conversion is inexact, the rounding mode
        is RNE. If a converted result cannot be represented in the destination format
        then: In case value is too big, the INT_MAX value (2^(w-1)-1, where w
        represents the number of bits in the destination format) is returned. In case
        value is too small, the INT_MIN value -(2^(w-1)) is returned. In case of NaN,
        (0) is returned.  * /

        Dest;
        TMP = 0;

        IF (src.bf16==NaN):
                TMP[31:0]=0x00000000;  // return zero in case of NaN
        ELSE IF (src.bf16 > 127) ||  (src.bf16 == +INF):
                TMP[31:0]=0x0000007F;     // saturate to max signed value
        ELSE IF (src.bf16 < -128) ||  (src.bf16 == -INF):
                TMP[31:0]=0x00000080;    // saturate to min signed value
        ELSE:
                // make it fp32 and then convert to INT using RNE
                TMP[31:0] = src.bf16 << 16
                TMP[31:0] = Convert_Single_Precision_Floating_Point_To_Integer_RNE(TMP);

        Dest.b = TMP[7:0];

        return Dest;
```

```
define convert_bf16_to_signed_byte_truncate_saturate(src.bf16):

        /* VCVTTBF162IBS converts brain-float16 floating point elements into
        signed byte integer elements.  When a conversion is inexact, a truncated (round
        toward zero) result is returned. If a converted result cannot be represented in
        the destination format then: In case the value is too big, the INT_MAX value
        (2^(w-1)-1, where w represents the number of bits in the destination format) is
        returned. In case the value is too small, the INT_MIN value -(2^(w-1)) is returned.
        In case of NaN, (0) is returned.  * /

        Dest;
        TMP = 0;

        IF (src.bf16==NaN):
                TMP[31:0]=0x00000000;    // return zero in case of NaN
        ELSE IF (src.bf16 > 127) ||  (src.bf16 == +INF):
                TMP[31:0]=0x0000007F;   // saturate to max signed value
        ELSE IF (src.bf16 < -128) ||  (src.bf16 == -INF):
                TMP[31:0]=0x00000080;    // saturate to min signed value
        ELSE:
                // make it fp32 and then convert to INT using RTZ (Truncate)
                TMP[31:0] = src.bf16 << 16
                TMP[31:0] = Convert_Single_Precision_Floating_Point_To_Integer_RTZ(TMP);

        Dest.b = TMP[7:0];
        return Dest;

```

```
define convert_bf16_to_unsigned_byte_rne_saturate(src.bf16):
        /* VCVTBF162IUBS converts brain-float16 floating point elements into
        un-signed byte integer elements.  When a conversion is inexact, the rounding
        mode is RNE. If a converted result cannot be represented in the destination
        format then: In case value is too big, the UINT_MAX value (2^w-1, where w
        represents the number of bits in the destination format) is returned. In case
        value is too small, the UINT_MIN value (0) is returned. In case of NaN, (0) is
        returned.  * /

        Dest;
        TMP = 0;

        IF (src.bf16==NaN):
                TMP[31:0]=0x00000000;  // return zero in case of NaN
        ELSE IF (src.bf16 > 255) ||  (src.bf16 == +INF):
                TMP[31:0]=0x000000FF;  // saturate to max unsigned value
        ELSE IF (src.bf16 < 0) ||  (src.bf16 == -INF):
                TMP[31:0]=0x00000000;  // saturate to min unsigned value
        ELSE:
                // make it fp32 and then convert to INT using RNE
                TMP[31:0] = src.bf16 << 16
                TMP[31:0] = Convert_Single_Precision_Floating_Point_To_Integer_RNE(TMP);

        Dest.b = TMP[7:0];
        return Dest;

```

```
define convert_bf16_to_unsigned_byte_truncate_saturate(src.bf16):

        /* VCVTTBF162IUBS converts brain-float16 floating point elements into
        un-signed byte integer elements.  When a conversion is inexact, a truncated
        (round toward zero) result is returned. If a converted result cannot be
        represented in the destination format then: In case value is too big, the
        UINT_MAX value (2^w-1, where w represents the number of bits in the destination
        format) is returned. In case value is too small, the UINT_MIN value (0) is
        returned. In case of NaN, (0) is returned.  * /

    Dest;
        TMP = 0;

        IF (src.bf16==NaN):
                TMP[31:0]=0x00000000;  // return zero in case of NaN
        ELSE IF (src.bf16 > 255) ||  (src.bf16 == +INF):
                TMP[31:0]=0x000000FF;  // saturate to max unsigned value
        ELSE IF (src.bf16 < 0) ||  (src.bf16 == -INF):
                TMP[31:0]=0x00000000;  // saturate to min unsigned value
        ELSE:
                // make it fp32 and then convert to INT using RTZ (Truncate)
                TMP[31:0] = src.bf16 << 16
                TMP[31:0] = Convert_Single_Precision_Floating_Point_To_Integer_RTZ(TMP);

        Dest.b = TMP[7:0];
        return Dest;
```

```
define convert_fp16_to_signed_byte_saturate(src.fp16):

        /* VCVTPH2IBS converts half-precision floating point elements into
        signed byte integer elements. When a conversion is inexact, floating-point
        precision exception is raised and the value returned is rounded according to
        the rounding control bits in the MXCSR register or the embedded rounding
        control bits. If a converted result cannot be represented in the destination
        format, the floating-point invalid exception is raised, and if this exception
        is masked then: In case the value is too big, the INT_MAX value (2^(w-1)-1, where w
        represents the number of bits in the destination format) is returned. In case the
        value is too small, the INT_MIN value -(2^(w-1)) is returned. In case of NaN, (0)
        is returned.  * /
        W=8;        // Byte destination size
        RC = MXCSR.RC
        EXP = 2^(W-1)
        OutOfDestRepresentation = Case (RC) :
                RUP:        ((src.fp16 <= -(EXP + 1))  ||  (src.fp16 > (EXP - 1))),
                RDN:        ((src.fp16 < -(EXP))        ||  (src.fp16 >= (EXP))),
                RTZ:        ((src.fp16 <= -(EXP + 1))   ||  (src.fp16 >= (EXP))),
                RNE:        ((src.fp16 < -(EXP + 1/2))  ||  (src.fp16 >= (EXP - 1/2)))

        Check IF (src.fp16 ==NaN) ||  (src.fp16 == +/-INF) ||OutOfDestRepresentation;
                                                                                // IE=1

    Dest;
        TMP = 0;

        IF (src.fp16==NaN):
                TMP[15:0]=0x0000;   // return zero in case of NaN

        ELSE IF (src.fp16 >= 127) ||  (src.fp16 == +INF):
                TMP[15:0]=0x007F;  // saturate to max signed value

        ELSE IF (src.fp16 <=-128) ||  (src.fp16 == -INF):
                TMP[15:0]=0x0080; // saturate to min signed value

        ELSE:
                // convert to INT using MXCSR.RC
                TMP[15:0] = Convert_fp16_to_integer16(src.fp16);
                // MXCSR.PE is updated according to result
        IF (src.fp16 !=half(TMP)) && (NOT OutOfDestRepresentation ) : PE=1

        Dest.b = TMP[7:0];
        return Dest;
```

```
define convert_fp16_to_signed_byte_truncate_saturate(src.fp16):

        /* VCVTTPH2IBS converts half-precision floating point elements into
        signed byte integer elements. When a conversion is inexact, floating-point
        precision exception is raised and a truncated (round toward zero) result is
        returned. If a converted result cannot be represented in the destination
        format, the floating-point invalid exception is raised, and if this exception
        is masked then: In case the value is too big, the INT_MAX value (2^(w-1)-1, where w
        represents the number of bits in the destination format) is returned. In case the
        value is too small, the INT_MIN value -(2^(w-1)) is returned. In case of NaN, (0)
        is returned.  * /

        W=8;        // Byte destination size
        RC = RTZ
        EXP = 2^(W-1)
        OutOfDestRepresentation = ((src.fp16 <= -(EXP + 1))   ||  (src.fp16 >= (EXP)));
        Check IF (src.fp16 ==NaN) ||  (src.fp16 == +/-INF) ||OutOfDestRepresentation;
        // IE=1

    Dest;
        TMP = 0;
        IF (src.fp16==NaN):
                TMP[15:0]=0x0000;    // return zero in case of NaN

        ELSE IF (src.fp16 >= 127) ||  (src.fp16 == +INF):
                TMP[15:0]=0x007F;  // saturate to max signed value
                // PE=1
        ELSE IF (src.fp16 <=-128) ||  (src.fp16 == -INF):
                TMP[15:0]=0x0080; // saturate to min signed value
                // PE=1
        ELSE:
                // convert to INT using RTZ (Truncate)
                TMP[15:0] = Convert_fp16_to_integer16_truncate(src.fp16<<16);
                // MXCSR.PE is updated according to result

        IF (src.fp16 !=half(TMP)) && (NOT OutOfDestRepresentation ) : PE=1

        Dest.b = TMP[7:0];
        return Dest;
```

```
define convert_fp16_to_unsigned_byte_saturate(src.fp16):

        /* VCVTPH2IUBS converts half-precision floating point elements into
        un-signed byte integer elements. When a conversion is inexact, floating-point
        precision exception is raised and the value returned is rounded according to
        the rounding control bits in the MXCSR register or the embedded rounding
        control bits. If a converted result cannot be represented in the destination
        format, the floating-point invalid exception is raised, and if this exception
        is masked then: In case the value is too big, the UINT_MAX value (2^(w-1)), where w
        represents the number of bits in the destination format) is returned. In case the
        value is too small, the UINT_MIN value (0) is returned. In case of NaN, (0)
        is returned.  * /

        W=8;        // Byte Destination Size
        RC = MXCSR.RC
        EXP = 2^(W)
        OutOfDestRepresentation = Case (RC) :
                RUP:        ((src.fp16 <= -1)  ||  (src.fp16 > (EXP - 1))),
                RDN:        ((src.fp16 < 0)   ||  (src.fp16 >= (EXP))),
                RTZ:        ((src.fp16 <= -1)  ||  (src.fp16 >= (EXP))),
                RNE:        ((src.fp16 < -1/2) ||  (src.fp16 >= (EXP - 1/2)))

        Check IF (src.fp16 ==NaN) ||  (src.fp16 == +/-INF) ||OutOfDestRepresentation
                                                                              // IE=1

        Dest;
        TMP = 0;
        IF (src.fp16==NaN):
                TMP[15:0]=0x0000;   // return zero in case of NaN

        ELSE IF (src.fp16 >= 255) ||  (src.fp16 == +INF):
                TMP[15:0]=0x00FF;  // saturate to max unsigned value

        ELSE IF (src.fp16 <=0 ) ||  (src.fp16 == -INF):
                TMP[15:0]=0x0000; // saturate to min unsigned value

        ELSE:
                // convert to INT using MXCSR.RC
                TMP[15:0] = Convert_fp16_to_unsigned_integer16(src.fp16);
                // MXCSR.PE is updated according to result
        IF (src.fp16 !=half(TMP)) && (NOT OutOfDestRepresentation ) : PE=1

        Dest.b = TMP[7:0];
        return Dest;
```

```
define convert_fp16_to_unsigned_byte_truncate_saturate(src.fp16):

        /* VCVTTPH2IUBS converts half-precision floating point elements into
        un-signed byte integer elements.  When a conversion is inexact, floating-point
        precision exception is raised and a truncated (round toward zero) result is
        returned. If a converted result cannot be represented in the destination
        format, the floating-point invalid exception is raised, and if this exception
        is masked then: In case the value is too big, the UINT_MAX value (2^w-1, where w
        represents the number of bits in the destination format) is returned.  In case the
        value is too small, the UINT_MIN value (0) is returned. In case of NaN, (0) is
        returned.  * /

        W=8;        // Byte Destination Size
        RC = RTZ
        EXP = 2^(W)
        OutOfDestRepresentation = ((src.fp16 <= -1)  ||  (src.fp16 >= (EXP)));

        Check IF (src.fp16 ==NaN) ||  (src.fp16 == +/-INF) ||OutOfDestRepresentation
                                                                                // IE=1

        Dest;
        TMP = 0;
        IF (src.fp16==NaN):
                TMP[15:0]=0x0000;   // return zero in case of NaN

        ELSE IF (src.fp16 >= 255) ||  (src.fp16 == +INF):
                TMP[15:0]=0x00FF; // saturate to max unsigned value

        ELSE IF (src.bf16 <=0) ||  (src.bf16 == -INF):
                TMP[15:0]=0x0000; // saturate to min unsigned value

    ELSE:
                // make it fp32 and then convert to INT using RTZ (Truncate)
                TMP[15:0] = Convert_fp16_to_unsigned_integer16_truncate(src.fp16);
                // MXCSR.PE is updated according to result
        IF (src.fp16 !=half(TMP)) && (NOT OutOfDestRepresentation ) : PE=1
        Dest.b = TMP[7:0];
        return Dest;
```

```
define convert_fp32_to_signed_byte_saturate(src.fp32):


        /* VCVTPS2IBS converts single-precision floating point elements into
        signed byte integer elements.  When a conversion is inexact, floating-point
        precision exception is raised and the value returned is rounded according to
        the rounding control bits in the MXCSR register or the embedded rounding
        control bits.  If a converted result cannot be represented in the destination
        format, the floating-point invalid exception is raised, and if this exception
        is masked then: If value is too big, the INT_MAX value (2^(w-1)-1, where w
        represents the number of bits in the destination format) is returned. If value
        is too small, the INT_MIN value -(2^(w-1)) is returned. For NaN, (0) is
        returned.  * /

        W=8;        // Byte destination size
        RC = MXCSR.RC
        EXP = 2^(W-1)
        OutOfDestRepresentation = Case (RC) :
                RUP:        ((src.fp32 <= -(EXP + 1))  ||  (src.fp32 > (EXP - 1))),
                RDN:        ((src.fp32 < -(EXP))        ||  (src.fp32 >= (EXP))),
                RTZ:        ((src.fp32 <= -(EXP + 1))   ||  (src.fp32 >= (EXP))),
                RNE:        ((src.fp32 < -(EXP + 1/2))  ||  (src.fp32 >= (EXP - 1/2)))

        Check IF (src.fp32 ==NaN) ||  (src.fp32 == +/-INF) ||OutOfDestRepresentation;
                                                                                // IE=1

        Dest;
        TMP = 0;

        IF (src.fp32==NaN):
                TMP[31:0]=0x00000000;  // return zero in case of NaN
        ELSE IF (src.fp32 >= 127) ||  (src.fp32 == +INF):
                TMP[31:0]=0x0000007F;     // saturate to max signed value
        ELSE IF (src.fp32 <=-128) ||  (src.fp32 == -INF):
                TMP[31:0]=0x00000080;    // saturate to min signed value
        ELSE:
                // (-128<x<127) make it fp32 and then convert to INT using MXCSR.RC
                TMP[31:0] = Convert_Single_Precision_Floating_Point_To_Integer(src.fp32);
                                        // MXCSR.PE is updated according to result

        IF (src.fp32 != float(TMP)) && (NOT OutOfDestRepresentation) : PE=1
        Dest.b = TMP[7:0];
        return Dest;
```

```
define convert_fp32_to_signed_byte_truncate_saturate(src.fp32):

        /* VCVTTPS2IBS converts single-precision floating point elements into
        signed byte integer elements.  When a conversion is inexact, floating-point
        precision exception is raised and a truncated (round toward zero) result is
        returned.  If a converted result cannot be represented in the destination
        format, the floating-point invalid exception is raised, and if this exception
        is masked then: If value is too big, the INT_MAX value (2^(w-1)-1, where w
        represents the number of bits in the destination format) is returned. If value
        is too small, the INT_MIN value -(2^(w-1)) is returned. For NaN, (0) is
        returned.  * /

        Dest;
        TMP = 0;

        W=8; // Byte destination size
        RC = RTZ
        EXP = 2^(W-1)
        OutOfDestRepresentation = ((src.fp32 <=-(EXP + 1)) ||  (src.fp32 >= EXP));

        IF ((src.fp32 ==NaN) ||
           (src.fp32 == +INF ||src.fp32 == -INF) ||
           OutOfDestRepresentation):
           // signal IE=1

        IF (src.fp32 ==NaN):
           TMP[31:0]=0x00000000;  // return zero in case of NaN
        ELIF (src.fp32 >= 127) ||  (src.fp32 == +INF):
            TMP[31:0]=0x0000007F; // saturate to max signed value
        ELIF (src.fp32 <= -128) ||  (src.fp32 == -INF):
            TMP[31:0]=0x00000080; // saturate to min signed value
        ELSE:
            // make it fp32 and then convert to INT using RTZ (Truncate)
            TMP[31:0] = Convert_Single_Precision_Floating_Point_To_Integer_RTZ(src.fp32);
            // set PE if inexact conversion.
        IF (src.fp32 != float(TMP)) && (NOT OutOfDestRepresentation) : PE=1
        Dest.b = TMP[7:0];

        return Dest;
```

```
define convert_fp32_to_unsigned_byte_saturate(src.fp32):

        /* VCVTBF162IUBS converts brain-float16 floating point elements into
        un-signed byte integer elements.  When a conversion is inexact, the rounding
        mode is RNE. If a converted result cannot be represented in the destination
        format then: In case value is too big, the UINT_MAX value (2^w-1, where w
        represents the number of bits in the destination format) is returned. In case
        value is too small, the UINT_MIN value (0) is returned. In case of NaN, (0) is
        returned.  * /

        TMP = 0;
        W=8;        // Byte Destination Size
        RC = MXCSR.RC
        OutOfDestRepresentation = ((src.fp32 <= -1)  ||  (src.fp32 >= (2^w)))

        IF ((src.fp32 ==NaN) ||  (src.fp32 == +/-INF) ||OutOfDestRepresentation)):
            // signal IE=1

        Dest;
        TMP = 0;
        IF (src.fp32==NaN):
                TMP[31:0]=0x00000000;   // return zero in case of NaN

        ELSE IF (src.fp32 > 255) ||  (src.fp32 == +INF):
                TMP[31:0]=0x000000FF;    // saturate to max unsigned value

        ELSE IF (src.fp32 < 0) ||  (src.fp32 == -INF):
                TMP[31:0]=0x00000000;    // saturate to min unsigned value
        ELSE:
                // convert to INT using MXCSR or embedded rounding mode
                TMP[31:0] = Convert_Single_Precision_Floating_Point_To_UInteger(src.fp32);
        IF (src.fp32 != float(TMP)) && (NOT OutOfDestRepresentation) : PE=1
        Dest.b = TMP[7:0];
        return Dest;
```

```
define convert_fp32_to_unsigned_byte_truncate_saturate(src.fp32):

        /* VCVTTPS2IUBS converts single-precision floating point elements into
        un-signed byte integer elements.  When a conversion is inexact, floating-point
        precision exception is raised and a truncated (round toward zero) result is
        returned.  If a converted result cannot be represented in the destination
        format, the floating-point invalid exception is raised, and if this exception
        is masked then: If value is too big, the UINT_MAX value (2^w-1, where w
        represents the number of bits in the destination format) is returned. If value
        is too small, the UINT_MIN value (0) is returned. For NaN, (0) is returned.  */

        Dest;
        TMP = 0;
        W=8;        // Byte Destination Size
        RC = RTZ
        OutOfDestRepresentation = ((src.fp32 <= -1)  ||  (src.fp32 >= (2^w)))

        IF (src.fp32 ==NaN) ||  (src.fp32 == +/-INF) || OutOfDestRepresentation:
            // signal IE=1

        IF (src.fp32==NaN):
                TMP[31:0]=0x00000000; // return zero in case of NaN

           ELSE IF (src.fp32 > 255) ||  (src.fp32 == +INF):
                TMP[31:0]=0x000000FF;    // saturate to max unsigned value

        ELSE IF (src.fp32 < 0) ||  (src.fp32 == -INF):
                TMP[31:0]=0x00000000;    // saturate to min unsigned value

        ELSE:
                // convert to INT using RTZ (Truncate)
                TMP = src.fp32
                TMP[31:0] = Convert_Single_Precision_Floating_Point_To_UInteger_RTZ(TMP);
                        // MXCSR.PE is updated according to result
        IF (src.fp32 != float(TMP)) && (NOT OutOfDestRepresentation) : PE=1
        Dest.b = TMP[7:0];
        return Dest;
```

```
define convert_SP_to_DW_SignedInteger_TruncateSaturate(src.fp32):

        /* VCVTTPS2DQS converts single-precision floating point elements into signed
        double word Integer elements.  When a conversion is inexact, floating-point
        precision exception is raised and a truncated (round toward zero) result is
        returned.  If a converted result cannot be represented in the destination
        format, the floating-point invalid exception is raised, and if this exception
        is masked then: If value is too big, the UINT_MAX value (2^w-1, where w
        represents the number of bits in the destination format) is returned. If value
        is too small, the INT_MIN value -(2^(w-1)) is returned. For NaN, (0) is returned. */

        Dest;
        TMP = 0;

        W=32;        // DW destination size
        RC = RTZ
        OutOfDestRepresentation = ((src.fp32 <= -(2^(w-1) + 1)) ||
                                   (src.fp32 >= (2^(w-1)))
                                  );

        IF (src.fp32 ==NaN) ||  (src.fp32 == +/-INF) ||OutOfDestRepresentation:
            // signal IE=1

        IF (src.fp32==NaN):
                TMP[31:0]=0x00000000; // return zero in case of NaN
        ELSE IF (src.fp32 >= +2^31 - 1) ||  (src.fp32 == +INF):
                TMP[31:0]=0x7FFFFFFF;    // saturate to max signed value
        ELSE IF (src.fp32 <=-2^31) ||  (src.fp32 == -INF):
                TMP[31:0]=0x80000000;    // saturate to min signed value
        ELSE:
                // make it fp32 and then convert to INT using RTZ (Truncate)
                //set PE if inexact conversion
                TMP[31:0] = Convert_SP_TO_DW_SignedInteger_RTZ(src.fp32);
        IF (src.fp32 != float(TMP)) && (NOT OutOfDestRepresentation) : PE=1
        Dest.dw = TMP[31:0];
        return Dest;
```

```
define convert_SP_to_DW_UnSignedInteger_TruncateSaturate(src.fp32):

        /* VCVTTPS2UDQS converts single-precision floating point elements into
        unsigned double word Integer elements.  When a conversion is inexact,
        floating-point precision exception is raised and a truncated (round
        toward zero) result is returned.  If a converted result cannot be
        represented in the destination format, the floating-point invalid
        exception is raised, and if this exception is masked then: If value is
        too big, the UINT_MAX value (2^w-1, where w represents the number of
        bits in the destination format) is returned. If value is too small, the
        UINT_MIN value (0) is returned. For NaN, (0) is returned.  */

        Dest;
        TMP = 0;

        W=32;        // DW destination size
        RC = RTZ
        EXP = 2^(W)
        OutOfDestRepresentation = ((src.fp32 <=-1)  ||  (src.fp32 >= EXP));

        IF ((src.fp32 ==NaN) ||
           (src.fp32 == +INF ||src.fp32 == -INF) ||
           OutOfDestRepresentation):
            // signal IE=1

        IF (src.fp32 == NaN):
            TMP[31:0]=0x00000000;        // return zero in case of NaN
        ELIF (src.fp32 >= 2^32 - 1) ||  (src.fp32 == +INF):
            TMP[31:0]=0xFFFFFFFF;    // saturate to max unsigned value
        ELIF (src.fp32 <= 0) ||  (src.fp32 == -INF):
            TMP[31:0]=0x00000000;  // saturate to min unsigned value
        ELSE:
            // Convert to fp32 and then convert to INT using RTZ (Truncate)
            TMP[31:0] = Convert_Single_Precision_Floating_Point_To_Integer_RTZ(src.fp32);
            // set PE if inexact conversion.
        IF (src.fp32 != float(TMP)) && (NOT OutOfDestRepresentation) : PE=1
        Dest.Dword = TMP[31:0];

        return Dest;
```

```
define convert_SP_to_QW_SignedInteger_TruncateSaturate(src.fp32):

        /* VCVTTPS2QQS converts single-precision floating point elements into
        packed signed quadword Integer elements.  When a conversion is inexact,
        floating-point precision exception is raised and a truncated (round
        toward zero) result is returned.  If a converted result cannot be
        represented in the destination format, the floating-point invalid
        exception is raised, and if this exception is masked then: If value is
        too big, the UINT_MAX value (2^w-1, where w represents the number of
        bits in the destination format) is returned. If value is too small, the
        INT_MIN value -(2^(w-1)) is returned. For NaN, (0) is returned. */

        Dest;
        TMP = 0;
        W=64;        // DW destination size
        RC = RTZ
        EXP = 2^(W-1)
        OutOfDestRepresentation = ((src.fp32 <= -(EXP + 1)) ||  (src.fp32 >= (EXP)));

        IF (src.fp32 ==NaN) ||  (src.fp32 == +/-INF) ||OutOfDestRepresentation):
            // signal IE=1

        IF (src.fp32==NaN):
                TMP[63:0]=0x00000000.00000000; // return zero in case of NaN
        ELSE IF (src.fp32 >= +2^31 - 1) ||  (src.fp32 == +INF):
                TMP[63:0]=0x7FFFFFFF.FFFFFFFF;    // saturate to max signed value
        ELSE IF (src.fp32 <=-2^31) ||  (src.fp32 == -INF):
                TMP[63:0]=0x80000000.00000000;    // saturate to min signed value
        ELSE:
                // convert to INT using RTZ (Truncate)//set PE if inexact conversion
                TMP[63:0] = Convert_SP_TO_QW_SignedInteger_RTZ(src.fp32);
        IF (src.fp32 != float(TMP)) && (NOT OutOfDestRepresentation) : PE=1
        Dest.qw = TMP[63:0];
```

```
define convert_SP_to_QW_UnSignedInteger_TruncateSaturate(src.fp32):

        /* VCVTTPS2UQQS converts single-precision floating point elements into
        packed unsigned quadword Integer elements.  When a conversion is
        inexact, floating-point precision exception is raised and a truncated
        (round toward zero) result is returned.  If a converted result cannot
        be represented in the destination format, the floating-point invalid
        exception is raised, and if this exception is masked then: If value is
        too big, the UINT_MAX value (2^w-1, where w represents the number of
        bits in the destination format) is returned. If value is too small, the
        UINT_MIN value (0) is returned. For NaN, (0) is returned.  */

        Dest;
        TMP = 0;

        W=64;        // QW destination size
        EXP = 2^(W)
        OutOfDestRepresentation = ((src.fp32 <=-1)  ||  (src.fp32 >= EXP));

        IF ((src.fp32 == NaN) ||
           (src.fp32 == +INF ||src.fp32 == -INF) ||
           OutOfDestRepresentation):
            // Signal IE=1

        IF (src.fp32 == NaN):
            TMP[63:0]=0x00000000.00000000;        // return zero in case of NaN
        ELIF (src.fp32 >= EXP - 1) ||  (src.fp32 == +INF):
            TMP[63:0]=0xFFFFFFFF.FFFFFFFF;    // saturate to max unsigned value
        ELIF (src.fp32 <= 0) ||  (src.fp32 == -INF):
            TMP[63:0]=0x00000000.00000000;  // saturate to min unsigned value
        ELSE:
            // make it fp32 and then convert to INT using RTZ (Truncate)
            TMP[63:0] = cvt_SP_FP_To_QW_Integer_RTZ(src.fp32);
            // set PE if inexact conversion.
        IF (src.fp32!= float(TMP)) && (NOT OutOfDestRepresentation) : PE=1
        Dest.Qword = TMP[63:0];

        return Dest;
```

```
define convert_DP_to_DW_SignedInteger_TruncateSaturate(src.fp64):

        /* VCVTTPD2DQS converts double-precision floating point elements into
        signed double word Integer elements.  When a conversion is inexact,
        floating-point precision exception is raised and a truncated (round
        toward zero) result is returned.  If a converted result cannot be
        represented in the destination format, the floating-point invalid
        exception is raised, and if this exception is masked then: If value is
        too big, the UINT_MAX value (2^(w-1)-1), where w represents the number of
        bits in the destination format) is returned. If value is too small, the
        INT_MIN value -(2^(w-1)) is returned. For NaN, (0) is returned. */

        Dest;
        TMP = 0;
        W=32;
        EXP = 2^(W-1)
        RC = RTZ
        OutOfDestRepresentation = ((src.fp64 <= -(EXP + 1)) ||  (src.fp64 >= (EXP)));

        IF (src.fp64 ==NaN) ||  (src.fp64 == +/-INF) || OutOfDestRepresentation:
            // signal IE=1

        IF (src.fp64==NaN):
                TMP[31:0]=0x00000000; // return zero in case of NaN
                ELSE IF (src.fp64 >= +2^31 − 1) ||  (src.fp64 == +INF):
                TMP[31:0]=0x7FFFFFFF;    // saturate to max signed value
        ELSE IF (src.fp64 <=−2^31) ||  (src.fp64 == -INF):
                TMP[31:0]=0x80000000;    // saturate to min signed value
        ELSE:
                //make it fp64 and convert to INT using RTZ (Truncate)
                //set PE if inexact conversion
                TMP[31:0] = Convert_DP_T0_DW_SignedInteger_RTZ(src.fp64);
        IF (src.fp64 != double(TMP)) & (NOT OutOfDestRepresentation) : PE=1
        Dest.dw = TMP[31:0];
        return Dest;
```

```
define convert_DP_to_DW_UnSignedInteger_TruncateSaturate(src.fp64):

        /* VCVTTPD2UDQS converts double-precision floating point elements into
        unsigned double word Integer elements.  When a conversion is inexact,
        floating-point precision exception is raised and a truncated (round
        toward zero) result is returned.  If a converted result cannot be
        represented in the destination format, the floating-point invalid
        exception is raised, and if this exception is masked then: If value is
        too big, the UINT_MAX value (2^w-1, where w represents the number of
        bits in the destination format) is returned. If value is too small, the
        UINT_MIN value (0) is returned. For NaN, (0) is returned.  */

        Dest;
        TMP = 0;
        W=32;        // QW destination size
        EXP = 2^(W)
        OutOfDestRepresentation = ((src.fp64 <=-1)  ||  (src.fp64 >= EXP));

        IF (src.fp64 == NaN) ||
           (src.fp64 == +INF || src.fp64 == -INF) ||
           OutOfDestRepresentation:
            // Signal IE=1

        IF (src.fp64 < 0) ||  (src.fp64 > EXP - 1):
            // Signal PE=1

        IF (src.fp64==NaN):
                TMP[31:0]=0x00000000; // return zero in case of NaN
        ELSE IF (src.fp64 >= +2^32 - 1) ||  (src.fp64 == +INF):
                TMP[31:0]=0xFFFFFFFF;    // saturate to max signed value
        ELSE IF (src.fp64 <=0) ||  (src.fp64 == -INF):
                TMP[31:0]=0x00000000;    // saturate to min signed value
        ELSE:
                // make it fp64 and then convert to INT using RTZ (Truncate)
                //set PE if inexact conversion
                TMP[31:0] = Convert_DP_TO_DW_UnSignedInteger_RTZ(src.fp64);
        IF (src.fp64 != double(TMP)) & (NOT OutOfDestRepresentation) : PE=1
        Dest.dw = TMP[31:0];
        return Dest;
```

```
define convert_DP_to_QW_SignedInteger_TruncateSaturate(src.fp64):

        /* VCVTTPD2QQS converts double-precision floating point elements into
        packed signed quadword Integer elements.  When a conversion is inexact,
        floating-point precision exception is raised and a truncated (round
        toward zero) result is returned.  If a converted result cannot be
        represented in the destination format, the floating-point invalid
        exception is raised, and if this exception is masked then: If value is
        too big, the UINT_MAX value (2^w-1, where w represents the number of
        bits in the destination format) is returned. If value is too small, the
        INT_MIN value -(2^(w-1)) is returned. For NaN, (0) is returned. */

        Dest = 0;
        TMP = 0;
        W=64;        // QW destination size
        EXP=2^(W-1)
        RC = RTZ
        OutOfDestRepresentation = ((src.fp64 <= -(EXP + 1)) ||  (src.fp64 >= (EXP)));

        IF ((src.fp64 == NAN) || (x == +-INF) || OutOfDestRepresentation):
            // signal IE=1

        IF (src.fp64==NaN):
                TMP[63:0]=0x00000000.00000000; // return zero in case of NaN
        ELSE IF (src.fp64 >= EXP - 1) ||  (src.fp64 == +INF):
                TMP[63:0]=7FFFFFFF.FFFFFFFF;    // saturate to max signed value
        ELSE IF (src.fp64 <=EXP) ||  (src.fp64 == -INF):
                TMP[63:0]=0x80000000.00000000;    // saturate to min signed value
        ELSE:
                // convert to INT using RTZ (Truncate)
                //set PE if inexact conversion
                TMP[63:0] = Convert_DP_To_QW_SignedInteger_RTZ(src.fp64);
        IF (src.fp64 != double(TMP)) & (NOT OutOfDestRepresentation) : PE=1
        Dest.qw = TMP[63:0];
        return Dest;
```

```
define convert_DP_to_QW_UnSignedInteger_TruncateSaturate(src.fp64):

        /* VCVTTPD2UQQS converts double-precision floating point elements into
        packed unsigned quadword Integer elements.  When a conversion is
        inexact, floating-point precision exception is raised and a truncated
        (round toward zero) result is returned.  If a converted result cannot
        be represented in the destination format, the floating-point invalid
        exception is raised, and if this exception is masked then: If value is
        too big, the UINT_MAX value (2^w-1, where w represents the number of
        bits in the destination format) is returned. If value is too small, the
        UINT_MIN value (0) is returned. For NaN, (0) is returned.  */

        Dest;
        TMP = 0;
        EXP = 2^(W)
        OutOfDestRepresentation = ((src.fp64 <=-1)  ||  (src.fp64 >= EXP))

        IF (src.fp64 ==NaN) ||  (src.fp64 == +/-INF) || OutOfDestRepresentation:
            // signal IE=1

        IF (src.fp64==NaN):
                TMP[63:0]=0x00000000.00000000; // return zero in case of NaN
        ELSE IF (src.fp64 >= EXP) ||  (src.fp64 == +INF):
                TMP[63:0]=FFFFFFFF.FFFFFFFF;    // saturate to max signed value
        ELSE IF (src.fp64 <=0 ) ||  (src.fp64 == -INF):
                TMP[63:0]=0x00000000.00000000;    // saturate to min signed value
        ELSE:
                // make it fp32 and then convert to INT using RTZ (Truncate)
                //set PE if inexact conversion
                TMP[63:0] = Convert_DP_To_QW_UnSignedInteger_RTZ(src.fp64);
        IF (src.fp64 != double(TMP)) & (NOT OutOfDestRepresentation) : PE=1
        Dest.qw = TMP[63:0];
        return Dest;
```

Chapter 6

INSTRUCTION TABLE

CPUID: AMX-AVX512,AVX10.2	OPERANDS	ENCSPACE
TCVTROWD2PS	zmm1, tmm2, imm8	EVEX
TCVTROWD2PS	zmm1, tmm2, r32	EVEX
TCVTROWPS2BF16H	zmm1, tmm2, imm8	EVEX
TCVTROWPS2BF16H	zmm1, tmm2, r32	EVEX
TCVTROWPS2BF16L	zmm1, tmm2, imm8	EVEX
TCVTROWPS2BF16L	zmm1, tmm2, r32	EVEX
TCVTROWPS2PHH	zmm1, tmm2, imm8	EVEX
TCVTROWPS2PHH	zmm1, tmm2, r32	EVEX
TCVTROWPS2PHL	zmm1, tmm2, imm8	EVEX
TCVTROWPS2PHL	zmm1, tmm2, r32	EVEX
TILEMOVROW	zmm1, tmm2, imm8	EVEX
TILEMOVROW	zmm1, tmm2, r32	EVEX
CPUID: AVX10-MOVRS	**OPERANDS**	**ENCSPACE**
VMOVRSB	xmm1, m128	EVEX
VMOVRSB	ymm1, m256	EVEX
VMOVRSB	zmm1, m512	EVEX
VMOVRSD	xmm1, m128	EVEX
VMOVRSD	ymm1, m256	EVEX
VMOVRSD	zmm1, m512	EVEX
VMOVRSQ	xmm1, m128	EVEX
VMOVRSQ	ymm1, m256	EVEX
VMOVRSQ	zmm1, m512	EVEX
VMOVRSW	xmm1, m128	EVEX
VMOVRSW	ymm1, m256	EVEX
VMOVRSW	zmm1, m512	EVEX
CPUID: AVX10.2	**OPERANDS**	**ENCSPACE**
VADDBF16	xmm1, xmm2, xmm3/m128	EVEX
VADDBF16	ymm1, ymm2, ymm3/m256	EVEX
VADDBF16	zmm1, zmm2, zmm3/m512	EVEX
VADDPD	ymm1, ymm2, ymm3/m256	EVEX
VADDPH	ymm1, ymm2, ymm3/m256	EVEX
VADDPS	ymm1, ymm2, ymm3/m256	EVEX
VCMPBF16	k1, xmm2, xmm3/m128, imm8	EVEX
VCMPBF16	k1, ymm2, ymm3/m256, imm8	EVEX
VCMPBF16	k1, zmm2, zmm3/m512, imm8	EVEX
VCMPPD	k1, ymm2, ymm3/m256, imm8	EVEX
VCMPPH	k1, ymm2, ymm3/m256, imm8	EVEX
VCMPPS	k1, ymm2, ymm3/m256, imm8	EVEX
VCOMISBF16	xmm1, xmm2/m16	EVEX
VCOMXSD	xmm1, xmm2/m64	EVEX

Table continued on next page...

CPUID: AVX10.2	OPERANDS	ENCSPACE *(contd.)*
VCOMXSH	xmm1, xmm2/m16	EVEX
VCOMXSS	xmm1, xmm2/m32	EVEX
VCVT2PH2BF8	xmm1, xmm2, xmm3/m128	EVEX
VCVT2PH2BF8	ymm1, ymm2, ymm3/m256	EVEX
VCVT2PH2BF8	zmm1, zmm2, zmm3/m512	EVEX
VCVT2PH2BF8S	xmm1, xmm2, xmm3/m128	EVEX
VCVT2PH2BF8S	ymm1, ymm2, ymm3/m256	EVEX
VCVT2PH2BF8S	zmm1, zmm2, zmm3/m512	EVEX
VCVT2PH2HF8	xmm1, xmm2, xmm3/m128	EVEX
VCVT2PH2HF8	ymm1, ymm2, ymm3/m256	EVEX
VCVT2PH2HF8	zmm1, zmm2, zmm3/m512	EVEX
VCVT2PH2HF8S	xmm1, xmm2, xmm3/m128	EVEX
VCVT2PH2HF8S	ymm1, ymm2, ymm3/m256	EVEX
VCVT2PH2HF8S	zmm1, zmm2, zmm3/m512	EVEX
VCVT2PS2PHX	xmm1, xmm2, xmm3/m128	EVEX
VCVT2PS2PHX	ymm1, ymm2, ymm3/m256	EVEX
VCVT2PS2PHX	zmm1, zmm2, zmm3/m512	EVEX
VCVTBF162IBS	xmm1, xmm2/m128	EVEX
VCVTBF162IBS	ymm1, ymm2/m256	EVEX
VCVTBF162IBS	zmm1, zmm2/m512	EVEX
VCVTBF162IUBS	xmm1, xmm2/m128	EVEX
VCVTBF162IUBS	ymm1, ymm2/m256	EVEX
VCVTBF162IUBS	zmm1, zmm2/m512	EVEX
VCVTBIASPH2BF8	xmm1, xmm2, xmm3/m128	EVEX
VCVTBIASPH2BF8	xmm1, ymm2, ymm3/m256	EVEX
VCVTBIASPH2BF8	ymm1, zmm2, zmm3/m512	EVEX
VCVTBIASPH2BF8S	xmm1, xmm2, xmm3/m128	EVEX
VCVTBIASPH2BF8S	xmm1, ymm2, ymm3/m256	EVEX
VCVTBIASPH2BF8S	ymm1, zmm2, zmm3/m512	EVEX
VCVTBIASPH2HF8	xmm1, xmm2, xmm3/m128	EVEX
VCVTBIASPH2HF8	xmm1, ymm2, ymm3/m256	EVEX
VCVTBIASPH2HF8	ymm1, zmm2, zmm3/m512	EVEX
VCVTBIASPH2HF8S	xmm1, xmm2, xmm3/m128	EVEX
VCVTBIASPH2HF8S	xmm1, ymm2, ymm3/m256	EVEX
VCVTBIASPH2HF8S	ymm1, zmm2, zmm3/m512	EVEX
VCVTDQ2PH	xmm1, ymm2/m256	EVEX
VCVTDQ2PS	ymm1, ymm2/m256	EVEX
VCVTHF82PH	xmm1, xmm2/m64	EVEX
VCVTHF82PH	ymm1, xmm2/m128	EVEX
VCVTHF82PH	zmm1, ymm2/m256	EVEX
VCVTPD2DQ	xmm1, ymm2/m256	EVEX

Table continued on next page...

CPUID: AVX10.2	OPERANDS	ENCSPACE *(contd.)*
VCVTPD2PH	xmm1, ymm2/m256	EVEX
VCVTPD2PS	xmm1, ymm2/m256	EVEX
VCVTPD2QQ	ymm1, ymm2/m256	EVEX
VCVTPD2UDQ	xmm1, ymm2/m256	EVEX
VCVTPD2UQQ	ymm1, ymm2/m256	EVEX
VCVTPH2BF8	xmm1, xmm2/m128	EVEX
VCVTPH2BF8	xmm1, ymm2/m256	EVEX
VCVTPH2BF8	ymm1, zmm2/m512	EVEX
VCVTPH2BF8S	xmm1, xmm2/m128	EVEX
VCVTPH2BF8S	xmm1, ymm2/m256	EVEX
VCVTPH2BF8S	ymm1, zmm2/m512	EVEX
VCVTPH2DQ	ymm1, xmm2/m128	EVEX
VCVTPH2HF8	xmm1, xmm2/m128	EVEX
VCVTPH2HF8	xmm1, ymm2/m256	EVEX
VCVTPH2HF8	ymm1, zmm2/m512	EVEX
VCVTPH2HF8S	xmm1, xmm2/m128	EVEX
VCVTPH2HF8S	xmm1, ymm2/m256	EVEX
VCVTPH2HF8S	ymm1, zmm2/m512	EVEX
VCVTPH2IBS	xmm1, xmm2/m128	EVEX
VCVTPH2IBS	ymm1, ymm2/m256	EVEX
VCVTPH2IBS	zmm1, zmm2/m512	EVEX
VCVTPH2IUBS	xmm1, xmm2/m128	EVEX
VCVTPH2IUBS	ymm1, ymm2/m256	EVEX
VCVTPH2IUBS	zmm1, zmm2/m512	EVEX
VCVTPH2PD	ymm1, xmm2/m64	EVEX
VCVTPH2PS	ymm1, xmm2/m128	EVEX
VCVTPH2PSX	ymm1, xmm2/m128	EVEX
VCVTPH2QQ	ymm1, xmm2/m64	EVEX
VCVTPH2UDQ	ymm1, xmm2/m128	EVEX
VCVTPH2UQQ	ymm1, xmm2/m64	EVEX
VCVTPH2UW	ymm1, ymm2/m256	EVEX
VCVTPH2W	ymm1, ymm2/m256	EVEX
VCVTPS2DQ	ymm1, ymm2/m256	EVEX
VCVTPS2IBS	xmm1, xmm2/m128	EVEX
VCVTPS2IBS	ymm1, ymm2/m256	EVEX
VCVTPS2IBS	zmm1, zmm2/m512	EVEX
VCVTPS2IUBS	xmm1, xmm2/m128	EVEX
VCVTPS2IUBS	ymm1, ymm2/m256	EVEX
VCVTPS2IUBS	zmm1, zmm2/m512	EVEX
VCVTPS2PD	ymm1, xmm2/m128	EVEX
VCVTPS2PH	xmm1, ymm2, imm8	EVEX

Table continued on next page...

CPUID: AVX10.2	OPERANDS	ENCSPACE *(contd.)*
VCVTPS2PHX	xmm1, ymm2/m256	EVEX
VCVTPS2QQ	ymm1, xmm2/m128	EVEX
VCVTPS2UDQ	ymm1, ymm2/m256	EVEX
VCVTPS2UQQ	ymm1, xmm2/m128	EVEX
VCVTQQ2PD	ymm1, ymm2/m256	EVEX
VCVTQQ2PH	xmm1, ymm2/m256	EVEX
VCVTQQ2PS	xmm1, ymm2/m256	EVEX
VCVTTBF162IBS	xmm1, xmm2/m128	EVEX
VCVTTBF162IBS	ymm1, ymm2/m256	EVEX
VCVTTBF162IBS	zmm1, zmm2/m512	EVEX
VCVTTBF162IUBS	xmm1, xmm2/m128	EVEX
VCVTTBF162IUBS	ymm1, ymm2/m256	EVEX
VCVTTBF162IUBS	zmm1, zmm2/m512	EVEX
VCVTTPD2DQ	xmm1, ymm2/m256	EVEX
VCVTTPD2DQS	xmm1, xmm2/m128	EVEX
VCVTTPD2DQS	xmm1, ymm2/m256	EVEX
VCVTTPD2DQS	ymm1, zmm2/m512	EVEX
VCVTTPD2QQ	ymm1, ymm2/m256	EVEX
VCVTTPD2QQS	xmm1, xmm2/m128	EVEX
VCVTTPD2QQS	ymm1, ymm2/m256	EVEX
VCVTTPD2QQS	zmm1, zmm2/m512	EVEX
VCVTTPD2UDQ	xmm1, ymm2/m256	EVEX
VCVTTPD2UDQS	xmm1, xmm2/m128	EVEX
VCVTTPD2UDQS	xmm1, ymm2/m256	EVEX
VCVTTPD2UDQS	ymm1, zmm2/m512	EVEX
VCVTTPD2UQQ	ymm1, ymm2/m256	EVEX
VCVTTPD2UQQS	xmm1, xmm2/m128	EVEX
VCVTTPD2UQQS	ymm1, ymm2/m256	EVEX
VCVTTPD2UQQS	zmm1, zmm2/m512	EVEX
VCVTTPH2DQ	ymm1, xmm2/m128	EVEX
VCVTTPH2IBS	xmm1, xmm2/m128	EVEX
VCVTTPH2IBS	ymm1, ymm2/m256	EVEX
VCVTTPH2IBS	zmm1, zmm2/m512	EVEX
VCVTTPH2IUBS	xmm1, xmm2/m128	EVEX
VCVTTPH2IUBS	ymm1, ymm2/m256	EVEX
VCVTTPH2IUBS	zmm1, zmm2/m512	EVEX
VCVTTPH2QQ	ymm1, xmm2/m64	EVEX
VCVTTPH2UDQ	ymm1, xmm2/m128	EVEX
VCVTTPH2UQQ	ymm1, xmm2/m64	EVEX
VCVTTPH2UW	ymm1, ymm2/m256	EVEX
VCVTTPH2W	ymm1, ymm2/m256	EVEX

Table continued on next page...

CPUID: AVX10.2	OPERANDS	ENCSPACE *(contd.)*
VCVTTPS2DQ	ymm1, ymm2/m256	EVEX
VCVTTPS2DQS	xmm1, xmm2/m128	EVEX
VCVTTPS2DQS	ymm1, ymm2/m256	EVEX
VCVTTPS2DQS	zmm1, zmm2/m512	EVEX
VCVTTPS2IBS	xmm1, xmm2/m128	EVEX
VCVTTPS2IBS	ymm1, ymm2/m256	EVEX
VCVTTPS2IBS	zmm1, zmm2/m512	EVEX
VCVTTPS2IUBS	xmm1, xmm2/m128	EVEX
VCVTTPS2IUBS	ymm1, ymm2/m256	EVEX
VCVTTPS2IUBS	zmm1, zmm2/m512	EVEX
VCVTTPS2QQ	ymm1, xmm2/m128	EVEX
VCVTTPS2QQS	xmm1, xmm2/m64	EVEX
VCVTTPS2QQS	ymm1, xmm2/m128	EVEX
VCVTTPS2QQS	zmm1, ymm2/m256	EVEX
VCVTTPS2UDQ	ymm1, ymm2/m256	EVEX
VCVTTPS2UDQS	xmm1, xmm2/m128	EVEX
VCVTTPS2UDQS	ymm1, ymm2/m256	EVEX
VCVTTPS2UDQS	zmm1, zmm2/m512	EVEX
VCVTTPS2UQQ	ymm1, xmm2/m128	EVEX
VCVTTPS2UQQS	xmm1, xmm2/m64	EVEX
VCVTTPS2UQQS	ymm1, xmm2/m128	EVEX
VCVTTPS2UQQS	zmm1, ymm2/m256	EVEX
VCVTTSD2SIS	r32, xmm1/m64	EVEX
VCVTTSD2SIS	r64, xmm1/m64	EVEX
VCVTTSD2USIS	r32, xmm1/m64	EVEX
VCVTTSD2USIS	r64, xmm1/m64	EVEX
VCVTTSS2SIS	r32, xmm1/m32	EVEX
VCVTTSS2SIS	r64, xmm1/m32	EVEX
VCVTTSS2USIS	r32, xmm1/m32	EVEX
VCVTTSS2USIS	r64, xmm1/m32	EVEX
VCVTUDQ2PH	xmm1, ymm2/m256	EVEX
VCVTUDQ2PS	ymm1, ymm2/m256	EVEX
VCVTUQQ2PD	ymm1, ymm2/m256	EVEX
VCVTUQQ2PH	xmm1, ymm2/m256	EVEX
VCVTUQQ2PS	xmm1, ymm2/m256	EVEX
VCVTUW2PH	ymm1, ymm2/m256	EVEX
VCVTW2PH	ymm1, ymm2/m256	EVEX
VDIVBF16	xmm1, xmm2, xmm3/m128	EVEX
VDIVBF16	ymm1, ymm2, ymm3/m256	EVEX
VDIVBF16	zmm1, zmm2, zmm3/m512	EVEX
VDIVPD	ymm1, ymm2, ymm3/m256	EVEX

Table continued on next page...

CPUID: AVX10.2	OPERANDS	ENCSPACE *(contd.)*
VDIVPH	ymm1, ymm2, ymm3/m256	EVEX
VDIVPS	ymm1, ymm2, ymm3/m256	EVEX
VDPPHPS	xmm1, xmm2, xmm3/m128	EVEX
VDPPHPS	ymm1, ymm2, ymm3/m256	EVEX
VDPPHPS	zmm1, zmm2, zmm3/m512	EVEX
VFCMADDCPH	ymm1, ymm2, ymm3/m256	EVEX
VFCMULCPH	ymm1, ymm2, ymm3/m256	EVEX
VFIXUPIMMPD	ymm1, ymm2, ymm3/m256, imm8	EVEX
VFIXUPIMMPS	ymm1, ymm2, ymm3/m256, imm8	EVEX
VFMADD132BF16	xmm1, xmm2, xmm3/m128	EVEX
VFMADD132BF16	ymm1, ymm2, ymm3/m256	EVEX
VFMADD132BF16	zmm1, zmm2, zmm3/m512	EVEX
VFMADD132PD	ymm1, ymm2, ymm3/m256	EVEX
VFMADD132PH	ymm1, ymm2, ymm3/m256	EVEX
VFMADD132PS	ymm1, ymm2, ymm3/m256	EVEX
VFMADD213BF16	xmm1, xmm2, xmm3/m128	EVEX
VFMADD213BF16	ymm1, ymm2, ymm3/m256	EVEX
VFMADD213BF16	zmm1, zmm2, zmm3/m512	EVEX
VFMADD213PD	ymm1, ymm2, ymm3/m256	EVEX
VFMADD213PH	ymm1, ymm2, ymm3/m256	EVEX
VFMADD213PS	ymm1, ymm2, ymm3/m256	EVEX
VFMADD231BF16	xmm1, xmm2, xmm3/m128	EVEX
VFMADD231BF16	ymm1, ymm2, ymm3/m256	EVEX
VFMADD231BF16	zmm1, zmm2, zmm3/m512	EVEX
VFMADD231PD	ymm1, ymm2, ymm3/m256	EVEX
VFMADD231PH	ymm1, ymm2, ymm3/m256	EVEX
VFMADD231PS	ymm1, ymm2, ymm3/m256	EVEX
VFMADDCPH	ymm1, ymm2, ymm3/m256	EVEX
VFMADDSUB132PD	ymm1, ymm2, ymm3/m256	EVEX
VFMADDSUB132PH	ymm1, ymm2, ymm3/m256	EVEX
VFMADDSUB132PS	ymm1, ymm2, ymm3/m256	EVEX
VFMADDSUB213PD	ymm1, ymm2, ymm3/m256	EVEX
VFMADDSUB213PH	ymm1, ymm2, ymm3/m256	EVEX
VFMADDSUB213PS	ymm1, ymm2, ymm3/m256	EVEX
VFMADDSUB231PD	ymm1, ymm2, ymm3/m256	EVEX
VFMADDSUB231PH	ymm1, ymm2, ymm3/m256	EVEX
VFMADDSUB231PS	ymm1, ymm2, ymm3/m256	EVEX
VFMSUB132BF16	xmm1, xmm2, xmm3/m128	EVEX
VFMSUB132BF16	ymm1, ymm2, ymm3/m256	EVEX
VFMSUB132BF16	zmm1, zmm2, zmm3/m512	EVEX
VFMSUB132PD	ymm1, ymm2, ymm3/m256	EVEX

Table continued on next page...

CPUID: AVX10.2	OPERANDS	ENCSPACE *(contd.)*
VFMSUB132PH	ymm1, ymm2, ymm3/m256	EVEX
VFMSUB132PS	ymm1, ymm2, ymm3/m256	EVEX
VFMSUB213BF16	xmm1, xmm2, xmm3/m128	EVEX
VFMSUB213BF16	ymm1, ymm2, ymm3/m256	EVEX
VFMSUB213BF16	zmm1, zmm2, zmm3/m512	EVEX
VFMSUB213PD	ymm1, ymm2, ymm3/m256	EVEX
VFMSUB213PH	ymm1, ymm2, ymm3/m256	EVEX
VFMSUB213PS	ymm1, ymm2, ymm3/m256	EVEX
VFMSUB231BF16	xmm1, xmm2, xmm3/m128	EVEX
VFMSUB231BF16	ymm1, ymm2, ymm3/m256	EVEX
VFMSUB231BF16	zmm1, zmm2, zmm3/m512	EVEX
VFMSUB231PD	ymm1, ymm2, ymm3/m256	EVEX
VFMSUB231PH	ymm1, ymm2, ymm3/m256	EVEX
VFMSUB231PS	ymm1, ymm2, ymm3/m256	EVEX
VFMSUBADD132PD	ymm1, ymm2, ymm3/m256	EVEX
VFMSUBADD132PH	ymm1, ymm2, ymm3/m256	EVEX
VFMSUBADD132PS	ymm1, ymm2, ymm3/m256	EVEX
VFMSUBADD213PD	ymm1, ymm2, ymm3/m256	EVEX
VFMSUBADD213PH	ymm1, ymm2, ymm3/m256	EVEX
VFMSUBADD213PS	ymm1, ymm2, ymm3/m256	EVEX
VFMSUBADD231PD	ymm1, ymm2, ymm3/m256	EVEX
VFMSUBADD231PH	ymm1, ymm2, ymm3/m256	EVEX
VFMSUBADD231PS	ymm1, ymm2, ymm3/m256	EVEX
VFMULCPH	ymm1, ymm2, ymm3/m256	EVEX
VFNMADD132BF16	xmm1, xmm2, xmm3/m128	EVEX
VFNMADD132BF16	ymm1, ymm2, ymm3/m256	EVEX
VFNMADD132BF16	zmm1, zmm2, zmm3/m512	EVEX
VFNMADD132PD	ymm1, ymm2, ymm3/m256	EVEX
VFNMADD132PH	ymm1, ymm2, ymm3/m256	EVEX
VFNMADD132PS	ymm1, ymm2, ymm3/m256	EVEX
VFNMADD213BF16	xmm1, xmm2, xmm3/m128	EVEX
VFNMADD213BF16	ymm1, ymm2, ymm3/m256	EVEX
VFNMADD213BF16	zmm1, zmm2, zmm3/m512	EVEX
VFNMADD213PD	ymm1, ymm2, ymm3/m256	EVEX
VFNMADD213PH	ymm1, ymm2, ymm3/m256	EVEX
VFNMADD213PS	ymm1, ymm2, ymm3/m256	EVEX
VFNMADD231BF16	xmm1, xmm2, xmm3/m128	EVEX
VFNMADD231BF16	ymm1, ymm2, ymm3/m256	EVEX
VFNMADD231BF16	zmm1, zmm2, zmm3/m512	EVEX
VFNMADD231PD	ymm1, ymm2, ymm3/m256	EVEX
VFNMADD231PH	ymm1, ymm2, ymm3/m256	EVEX

Table continued on next page...

CPUID: AVX10.2	OPERANDS	ENCSPACE *(contd.)*
VFNMADD231PS	ymm1, ymm2, ymm3/m256	EVEX
VFNMSUB132BF16	xmm1, xmm2, xmm3/m128	EVEX
VFNMSUB132BF16	ymm1, ymm2, ymm3/m256	EVEX
VFNMSUB132BF16	zmm1, zmm2, zmm3/m512	EVEX
VFNMSUB132PD	ymm1, ymm2, ymm3/m256	EVEX
VFNMSUB132PH	ymm1, ymm2, ymm3/m256	EVEX
VFNMSUB132PS	ymm1, ymm2, ymm3/m256	EVEX
VFNMSUB213BF16	xmm1, xmm2, xmm3/m128	EVEX
VFNMSUB213BF16	ymm1, ymm2, ymm3/m256	EVEX
VFNMSUB213BF16	zmm1, zmm2, zmm3/m512	EVEX
VFNMSUB213PD	ymm1, ymm2, ymm3/m256	EVEX
VFNMSUB213PH	ymm1, ymm2, ymm3/m256	EVEX
VFNMSUB213PS	ymm1, ymm2, ymm3/m256	EVEX
VFNMSUB231BF16	xmm1, xmm2, xmm3/m128	EVEX
VFNMSUB231BF16	ymm1, ymm2, ymm3/m256	EVEX
VFNMSUB231BF16	zmm1, zmm2, zmm3/m512	EVEX
VFNMSUB231PD	ymm1, ymm2, ymm3/m256	EVEX
VFNMSUB231PH	ymm1, ymm2, ymm3/m256	EVEX
VFNMSUB231PS	ymm1, ymm2, ymm3/m256	EVEX
VFPCLASSBF16	k1, xmm2/m128, imm8	EVEX
VFPCLASSBF16	k1, ymm2/m256, imm8	EVEX
VFPCLASSBF16	k1, zmm2/m512, imm8	EVEX
VGETEXPBF16	xmm1, xmm2/m128	EVEX
VGETEXPBF16	ymm1, ymm2/m256	EVEX
VGETEXPBF16	zmm1, zmm2/m512	EVEX
VGETEXPPD	ymm1, ymm2/m256	EVEX
VGETEXPPH	ymm1, ymm2/m256	EVEX
VGETEXPPS	ymm1, ymm2/m256	EVEX
VGETMANTBF16	xmm1, xmm2/m128, imm8	EVEX
VGETMANTBF16	ymm1, ymm2/m256, imm8	EVEX
VGETMANTBF16	zmm1, zmm2/m512, imm8	EVEX
VGETMANTPD	ymm1, ymm2/m256, imm8	EVEX
VGETMANTPH	ymm1, ymm2/m256, imm8	EVEX
VGETMANTPS	ymm1, ymm2/m256, imm8	EVEX
VMAXBF16	xmm1, xmm2, xmm3/m128	EVEX
VMAXBF16	ymm1, ymm2, ymm3/m256	EVEX
VMAXBF16	zmm1, zmm2, zmm3/m512	EVEX
VMAXPD	ymm1, ymm2, ymm3/m256	EVEX
VMAXPH	ymm1, ymm2, ymm3/m256	EVEX
VMAXPS	ymm1, ymm2, ymm3/m256	EVEX
VMINBF16	xmm1, xmm2, xmm3/m128	EVEX

Table continued on next page...

CPUID: AVX10.2	OPERANDS	ENCSPACE *(contd.)*
VMINBF16	ymm1, ymm2, ymm3/m256	EVEX
VMINBF16	zmm1, zmm2, zmm3/m512	EVEX
VMINMAXBF16	xmm1, xmm2, xmm3/m128, imm8	EVEX
VMINMAXBF16	ymm1, ymm2, ymm3/m256, imm8	EVEX
VMINMAXBF16	zmm1, zmm2, zmm3/m512, imm8	EVEX
VMINMAXPD	xmm1, xmm2, xmm3/m128, imm8	EVEX
VMINMAXPD	ymm1, ymm2, ymm3/m256, imm8	EVEX
VMINMAXPD	zmm1, zmm2, zmm3/m512, imm8	EVEX
VMINMAXPH	xmm1, xmm2, xmm3/m128, imm8	EVEX
VMINMAXPH	ymm1, ymm2, ymm3/m256, imm8	EVEX
VMINMAXPH	zmm1, zmm2, zmm3/m512, imm8	EVEX
VMINMAXPS	xmm1, xmm2, xmm3/m128, imm8	EVEX
VMINMAXPS	ymm1, ymm2, ymm3/m256, imm8	EVEX
VMINMAXPS	zmm1, zmm2, zmm3/m512, imm8	EVEX
VMINMAXSD	xmm1, xmm2, xmm3/m64, imm8	EVEX
VMINMAXSH	xmm1, xmm2, xmm3/m16, imm8	EVEX
VMINMAXSS	xmm1, xmm2, xmm3/m32, imm8	EVEX
VMINPD	ymm1, ymm2, ymm3/m256	EVEX
VMINPH	ymm1, ymm2, ymm3/m256	EVEX
VMINPS	ymm1, ymm2, ymm3/m256	EVEX
VMOVD	xmm1, xmm2/m32	EVEX
VMOVD	xmm1/m32, xmm2	EVEX
VMOVW	xmm1, xmm2/m16	EVEX
VMOVW	xmm1/m16, xmm2	EVEX
VMPSADBW	xmm1, xmm2, xmm3/m128, imm8	EVEX
VMPSADBW	ymm1, ymm2, ymm3/m256, imm8	EVEX
VMPSADBW	zmm1, zmm2, zmm3/m512, imm8	EVEX
VMULBF16	xmm1, xmm2, xmm3/m128	EVEX
VMULBF16	ymm1, ymm2, ymm3/m256	EVEX
VMULBF16	zmm1, zmm2, zmm3/m512	EVEX
VMULPD	ymm1, ymm2, ymm3/m256	EVEX
VMULPH	ymm1, ymm2, ymm3/m256	EVEX
VMULPS	ymm1, ymm2, ymm3/m256	EVEX
VPDPBSSD	xmm1, xmm2, xmm3/m128	EVEX
VPDPBSSD	ymm1, ymm2, ymm3/m256	EVEX
VPDPBSSD	zmm1, zmm2, zmm3/m512	EVEX
VPDPBSSDS	xmm1, xmm2, xmm3/m128	EVEX
VPDPBSSDS	ymm1, ymm2, ymm3/m256	EVEX
VPDPBSSDS	zmm1, zmm2, zmm3/m512	EVEX
VPDPBSUD	xmm1, xmm2, xmm3/m128	EVEX
VPDPBSUD	ymm1, ymm2, ymm3/m256	EVEX

Table continued on next page...

CPUID: AVX10.2	OPERANDS	ENCSPACE *(contd.)*
VPDPBSUD	zmm1, zmm2, zmm3/m512	EVEX
VPDPBSUDS	xmm1, xmm2, xmm3/m128	EVEX
VPDPBSUDS	ymm1, ymm2, ymm3/m256	EVEX
VPDPBSUDS	zmm1, zmm2, zmm3/m512	EVEX
VPDPBUUD	xmm1, xmm2, xmm3/m128	EVEX
VPDPBUUD	ymm1, ymm2, ymm3/m256	EVEX
VPDPBUUD	zmm1, zmm2, zmm3/m512	EVEX
VPDPBUUDS	xmm1, xmm2, xmm3/m128	EVEX
VPDPBUUDS	ymm1, ymm2, ymm3/m256	EVEX
VPDPBUUDS	zmm1, zmm2, zmm3/m512	EVEX
VPDPWSUD	xmm1, xmm2, xmm3/m128	EVEX
VPDPWSUD	ymm1, ymm2, ymm3/m256	EVEX
VPDPWSUD	zmm1, zmm2, zmm3/m512	EVEX
VPDPWSUDS	xmm1, xmm2, xmm3/m128	EVEX
VPDPWSUDS	ymm1, ymm2, ymm3/m256	EVEX
VPDPWSUDS	zmm1, zmm2, zmm3/m512	EVEX
VPDPWUSD	xmm1, xmm2, xmm3/m128	EVEX
VPDPWUSD	ymm1, ymm2, ymm3/m256	EVEX
VPDPWUSD	zmm1, zmm2, zmm3/m512	EVEX
VPDPWUSDS	xmm1, xmm2, xmm3/m128	EVEX
VPDPWUSDS	ymm1, ymm2, ymm3/m256	EVEX
VPDPWUSDS	zmm1, zmm2, zmm3/m512	EVEX
VPDPWUUD	xmm1, xmm2, xmm3/m128	EVEX
VPDPWUUD	ymm1, ymm2, ymm3/m256	EVEX
VPDPWUUD	zmm1, zmm2, zmm3/m512	EVEX
VPDPWUUDS	xmm1, xmm2, xmm3/m128	EVEX
VPDPWUUDS	ymm1, ymm2, ymm3/m256	EVEX
VPDPWUUDS	zmm1, zmm2, zmm3/m512	EVEX
VRANGEPD	ymm1, ymm2, ymm3/m256, imm8	EVEX
VRANGEPS	ymm1, ymm2, ymm3/m256, imm8	EVEX
VRCPBF16	xmm1, xmm2/m128	EVEX
VRCPBF16	ymm1, ymm2/m256	EVEX
VRCPBF16	zmm1, zmm2/m512	EVEX
VREDUCEBF16	xmm1, xmm2/m128, imm8	EVEX
VREDUCEBF16	ymm1, ymm2/m256, imm8	EVEX
VREDUCEBF16	zmm1, zmm2/m512, imm8	EVEX
VREDUCEPD	ymm1, ymm2/m256, imm8	EVEX
VREDUCEPH	ymm1, ymm2/m256, imm8	EVEX
VREDUCEPS	ymm1, ymm2/m256, imm8	EVEX
VRNDSCALEBF16	xmm1, xmm2/m128, imm8	EVEX
VRNDSCALEBF16	ymm1, ymm2/m256, imm8	EVEX

Table continued on next page...

CPUID: AVX10.2	OPERANDS	ENCSPACE *(contd.)*
VRNDSCALEBF16	zmm1, zmm2/m512, imm8	EVEX
VRNDSCALEPD	ymm1, ymm2/m256, imm8	EVEX
VRNDSCALEPH	ymm1, ymm2/m256, imm8	EVEX
VRNDSCALEPS	ymm1, ymm2/m256, imm8	EVEX
VRSQRTBF16	xmm1, xmm2/m128	EVEX
VRSQRTBF16	ymm1, ymm2/m256	EVEX
VRSQRTBF16	zmm1, zmm2/m512	EVEX
VSCALEFBF16	xmm1, xmm2, xmm3/m128	EVEX
VSCALEFBF16	ymm1, ymm2, ymm3/m256	EVEX
VSCALEFBF16	zmm1, zmm2, zmm3/m512	EVEX
VSCALEFPD	ymm1, ymm2, ymm3/m256	EVEX
VSCALEFPH	ymm1, ymm2, ymm3/m256	EVEX
VSCALEFPS	ymm1, ymm2, ymm3/m256	EVEX
VSQRTBF16	xmm1, xmm2/m128	EVEX
VSQRTBF16	ymm1, ymm2/m256	EVEX
VSQRTBF16	zmm1, zmm2/m512	EVEX
VSQRTPD	ymm1, ymm2/m256	EVEX
VSQRTPH	ymm1, ymm2/m256	EVEX
VSQRTPS	ymm1, ymm2/m256	EVEX
VSUBBF16	xmm1, xmm2, xmm3/m128	EVEX
VSUBBF16	ymm1, ymm2, ymm3/m256	EVEX
VSUBBF16	zmm1, zmm2, zmm3/m512	EVEX
VSUBPD	ymm1, ymm2, ymm3/m256	EVEX
VSUBPH	ymm1, ymm2, ymm3/m256	EVEX
VSUBPS	ymm1, ymm2, ymm3/m256	EVEX
VUCOMXSD	xmm1, xmm2/m64	EVEX
VUCOMXSH	xmm1, xmm2/m16	EVEX
VUCOMXSS	xmm1, xmm2/m32	EVEX
CPUID: AVX10.2,SM4	**OPERANDS**	**ENCSPACE**
VSM4KEY4	xmm1, xmm2, xmm3/m128	EVEX
VSM4KEY4	ymm1, ymm2, ymm3/m256	EVEX
VSM4KEY4	zmm1, zmm2, zmm3/m512	EVEX
VSM4RNDS4	xmm1, xmm2, xmm3/m128	EVEX
VSM4RNDS4	ymm1, ymm2, ymm3/m256	EVEX
VSM4RNDS4	zmm1, zmm2, zmm3/m512	EVEX

Chapter 7

INTEL® AVX10 BF16 INSTRUCTIONS

7.1 VADDBF16

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.128.66.MAP5.W0 58 /r VADDBF16 xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	A	V/V	AVX10.2
EVEX.256.66.MAP5.W0 58 /r VADDBF16 ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst	A	V/V	AVX10.2
EVEX.512.66.MAP5.W0 58 /r VADDBF16 zmm1{k1}{z}, zmm2, zmm3/m512/m16bcst	A	V/V	AVX10.2

7.1.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(w)	VVVV(r)	MODRM.R/M(r)	N/A

7.1.2 DESCRIPTION

Note:

For instructions with a CPUID feature flag specifying Intel® AVX10, the programmer must check the available vector options on the processor at run-time via the VL256 and VL512 bits in Converged Vector ISA Leaf 0x24. These bits enumerate whether the corresponding vector lengths are supported (128-bit vectors are always supported) and as such will determine the set of instructions available to the programmer listed in the above opcode table.

This instruction adds packed BF16 values from source operands and stores the packed BF16 result in the destination operand. The destination elements are updated according to the writemask. This instruction does not generate floating point exceptions and does not consult or update MXCSR. Denormal BF16 input operands are treated as zeros (DAZ) and denormal BF16 outputs are flushed to zero (FTZ). Rounding Mode is always RNE.

7.1.3 OPERATION

```
VADDBF16 (EVEX encoded versions) when src2 operand is a register
(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        DEST.bf16[j] := SRC1.bf16[j] + SRC2.bf16[j] //DAZ, FTZ, RNE, SAE
    ELSE IF *zeroing*:
        DEST.bf16[j] := 0
    // else dest.bf16[j] remains unchanged

DEST[MAX_VL-1:VL] := 0
```

```
VADDBF16 (EVEX encoded versions) when src2 operand is a memory source
(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF EVEX.b == 1:
            DEST.bf16[j] := SRC1.bf16[j] + SRC2.bf16[0] //DAZ, FTZ, RNE, SAE
        ELSE:
            DEST.bf16[j] := SRC1.bf16[j] + SRC2.bf16[j] //DAZ, FTZ, RNE, SAE

    ELSE IF *zeroing*:
        DEST.bf16[j] := 0
    // else dest.bf16[j] remains unchanged

DEST[MAX_VL-1:VL] := 0
```

7.1.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VADDBF16 xmm1, xmm2, xmm3/m128	E4	N/A	AVX10.2
VADDBF16 ymm1, ymm2, ymm3/m256	E4	N/A	AVX10.2
VADDBF16 zmm1, zmm2, zmm3/m512	E4	N/A	AVX10.2

7.2 VCMPBF16

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.128.F2.0F3A.W0 C2 /r /ib VCMPBF16 k1{k2}, xmm2, xmm3/m128/m16bcst, imm8	A	V/V	AVX10.2
EVEX.256.F2.0F3A.W0 C2 /r /ib VCMPBF16 k1{k2}, ymm2, ymm3/m256/m16bcst, imm8	A	V/V	AVX10.2
EVEX.512.F2.0F3A.W0 C2 /r /ib VCMPBF16 k1{k2}, zmm2, zmm3/m512/m16bcst, imm8	A	V/V	AVX10.2

7.2.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(w)	VVVV(r)	MODRM.R/M(r)	IMM8(r)

7.2.2 DESCRIPTION

Note:

For instructions with a CPUID feature flag specifying Intel® AVX10, the programmer must check the available vector options on the processor at run-time via the VL256 and VL512 bits in Converged Vector ISA Leaf 0x24. These bits enumerate whether the corresponding vector lengths are supported (128-bit vectors are always supported) and as such will determine the set of instructions available to the programmer listed in the above opcode table.

This instruction compares packed BF16 values from source operands and stores the result in the destination mask operand. The comparison predicate operand (immediate byte bits 4:0) specifies the type of comparison performed on each of the pairs of packed values. The destination elements are updated according to the writemask. This instruction does not generate floating point exceptions and does not consult or update MXCSR. Denormal BF16 input operands are treated as zeros (DAZ)

7.2.3 OPERATION

```
CASE (imm8 & 0x1F) OF
0: CMP_OPERATOR := EQ_OQ;
1: CMP_OPERATOR := LT_OS;
2: CMP_OPERATOR := LE_OS;
3: CMP_OPERATOR := UNORD_Q;
4: CMP_OPERATOR := NEQ_UQ;
5: CMP_OPERATOR := NLT_US;
6: CMP_OPERATOR := NLE_US;
7: CMP_OPERATOR := ORD_Q;
8: CMP_OPERATOR := EQ_UQ;
9: CMP_OPERATOR := NGE_US;
10: CMP_OPERATOR := NGT_US;
11: CMP_OPERATOR := FALSE_OQ;
12: CMP_OPERATOR := NEQ_OQ;
13: CMP_OPERATOR := GE_OS;
14: CMP_OPERATOR := GT_OS;
15: CMP_OPERATOR := TRUE_UQ;
16: CMP_OPERATOR := EQ_OS;
17: CMP_OPERATOR := LT_OQ;
18: CMP_OPERATOR := LE_OQ;
19: CMP_OPERATOR := UNORD_S;
20: CMP_OPERATOR := NEQ_US;
21: CMP_OPERATOR := NLT_UQ;
22: CMP_OPERATOR := NLE_UQ;
23: CMP_OPERATOR := ORD_S;
24: CMP_OPERATOR := EQ_US;
25: CMP_OPERATOR := NGE_UQ;
26: CMP_OPERATOR := NGT_UQ;
27: CMP_OPERATOR := FALSE_OS;
28: CMP_OPERATOR := NEQ_OS;
29: CMP_OPERATOR := GE_OQ;
30: CMP_OPERATOR := GT_OQ;
31: CMP_OPERATOR := TRUE_US;
ESAC
```

```
VCMPBF16 (EVEX encoded versions)
(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j := 0 TO KL-1:
    IF k2[j] OR *no writemask*:
        IF EVEX.b == 1:
            tsrc2 := SRC2.bf16[0]
        ELSE:
            tsrc2 := SRC2.bf16[j]
        DEST.bit[j] := SRC1.bf16[j] CMP_OPERATOR tsrc2 //DAZ, SAE
    ELSE *zero masking only*:
        DEST.bit[j] := 0

DEST[MAX_KL-1:KL] := 0
```

7.2.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VCMPBF16 k1, xmm2, xmm3/m128, imm8	E4	N/A	AVX10.2
VCMPBF16 k1, ymm2, ymm3/m256, imm8	E4	N/A	AVX10.2
VCMPBF16 k1, zmm2, zmm3/m512, imm8	E4	N/A	AVX10.2

7.3 VCOMISBF16

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.LLIG.66.MAP5.W0 2F /r VCOMISBF16 xmm1, xmm2/m16	A	V/V	AVX10.2

7.3.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	SCALAR	MODRM.REG(r)	MODRM.R/M(r)	N/A	N/A

7.3.2 DESCRIPTION

Note:

For SCALAR instructions with a CPUID feature flag specifying Intel® AVX10, the programmer can skip the check for the available vector options on the processor at run-time via the VL256 and VL512 bits in Converged Vector ISA Leaf 0x24. These bits enumerate whether the corresponding vector lengths are supported and usually determine the set of instructions available to the programmer. However, scalar instructions are architected without the notion of vector length, so this step does not strictly apply.

Compares the half-precision floating-point values in the low word of operand 1 (first operand) and operand 2 (second operand), and sets the ZF, PF, and CF flags in the EFLAGS register according to the result (unordered, greater than, less than, or equal). The OF, SF and AF flags in the EFLAGS register are set to 0. The unordered result is returned if either source operand is a NaN (QNaN or SNaN). This instruction does not generate floating point exceptions and does not consult or update MXCSR. As such, only a single VCOM variant is defined, as the differentation between VCOM vs. VUCOM variants are that they only differ in exception behaviors. Denormal BF16 input operands are treated as zeros (DAZ).

Operand 1 is an XMM register; operand 2 can be an XMM register or a 16-bit memory location.

EVEX.vvvv are reserved and must be 1111b, otherwise instructions will #UD

7.3.3 OPERATION

```
VCOMISBF16

RESULT := Compare(SRC1.bf16[0],SRC2.bf16[0])
IF RESULT is UNORDERED:
    ZF, PF, CF := 1, 1, 1
ELIF RESULT is GREATER_THAN:
    ZF, PF, CF := 0, 0, 0
ELIF RESULT is LESS_THAN:
    ZF, PF, CF := 0, 0, 1
ELSE: // RESULT is EQUALS
    ZF, PF, CF := 1, 0, 0

OF, AF, SF := 0, 0, 0
```

7.3.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VCOMISBF16 xmm1, xmm2/m16	E10NF	N/A	AVX10.2

7.4 VDIVBF16

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.128.66.MAP5.W0 5E /r VDIVBF16 xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	A	V/V	AVX10.2
EVEX.256.66.MAP5.W0 5E /r VDIVBF16 ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst	A	V/V	AVX10.2
EVEX.512.66.MAP5.W0 5E /r VDIVBF16 zmm1{k1}{z}, zmm2, zmm3/m512/m16bcst	A	V/V	AVX10.2

7.4.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(w)	VVVV(r)	MODRM.R/M(r)	N/A

7.4.2 DESCRIPTION

Note:

For instructions with a CPUID feature flag specifying Intel® AVX10, the programmer must check the available vector options on the processor at run-time via the VL256 and VL512 bits in Converged Vector ISA Leaf 0x24. These bits enumerate whether the corresponding vector lengths are supported (128-bit vectors are always supported) and as such will determine the set of instructions available to the programmer listed in the above opcode table.

This instruction divides packed BF16 values from the first source operand by the corresponding elements in the second source operand, storing the packed BF16 result in the destination operand. The destination elements are updated according to the writemask. This instruction does not generate floating point exceptions and does not consult or update MXCSR. Denormal BF16 input operands are treated as zeros (DAZ) and denormal BF16 outputs are flushed to zero (FTZ). Rounding Mode is always RNE.

7.4.3 OPERATION

```
VDIVBF16 (EVEX encoded versions) when src2 operand is a register
(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        DEST.bf16[j] := SRC1.bf16[j] / SRC2.bf16[j] //DAZ, FTZ, RNE, SAE
    ELSE IF *zeroing*:
        DEST.bf16[j] := 0
    // else dest.bf16[j] remains unchanged

DEST[MAX_VL-1:VL] := 0
```

```
VDIVBF16 (EVEX encoded versions) when src2 operand is a memory source
(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF EVEX.b == 1:
            DEST.bf16[j] := SRC1.bf16[j] / SRC2.bf16[0] //DAZ, FTZ, RNE, SAE
        ELSE:
            DEST.bf16[j] := SRC1.bf16[j] / SRC2.bf16[j] //DAZ, FTZ, RNE, SAE

    ELSE IF *zeroing*:
        DEST.bf16[j] := 0
    // else dest.bf16[j] remains unchanged

DEST[MAX_VL-1:VL] := 0
```

7.4.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VDIVBF16 xmm1, xmm2, xmm3/m128	E4	N/A	AVX10.2
VDIVBF16 ymm1, ymm2, ymm3/m256	E4	N/A	AVX10.2
VDIVBF16 zmm1, zmm2, zmm3/m512	E4	N/A	AVX10.2

7.5 VF[,N]M[ADD,SUB][132,213,231]BF16

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.128.NP.MAP6.W0 98 /r VFMADD132BF16 xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	A	V/V	AVX10.2
EVEX.256.NP.MAP6.W0 98 /r VFMADD132BF16 ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst	A	V/V	AVX10.2
EVEX.512.NP.MAP6.W0 98 /r VFMADD132BF16 zmm1{k1}{z}, zmm2, zmm3/m512/m16bcst	A	V/V	AVX10.2
EVEX.128.NP.MAP6.W0 A8 /r VFMADD213BF16 xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	A	V/V	AVX10.2
EVEX.256.NP.MAP6.W0 A8 /r VFMADD213BF16 ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst	A	V/V	AVX10.2
EVEX.512.NP.MAP6.W0 A8 /r VFMADD213BF16 zmm1{k1}{z}, zmm2, zmm3/m512/m16bcst	A	V/V	AVX10.2
EVEX.128.NP.MAP6.W0 B8 /r VFMADD231BF16 xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	A	V/V	AVX10.2
EVEX.256.NP.MAP6.W0 B8 /r VFMADD231BF16 ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst	A	V/V	AVX10.2
EVEX.512.NP.MAP6.W0 B8 /r VFMADD231BF16 zmm1{k1}{z}, zmm2, zmm3/m512/m16bcst	A	V/V	AVX10.2
EVEX.128.NP.MAP6.W0 9A /r VFMSUB132BF16 xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	A	V/V	AVX10.2
EVEX.256.NP.MAP6.W0 9A /r VFMSUB132BF16 ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst	A	V/V	AVX10.2
EVEX.512.NP.MAP6.W0 9A /r VFMSUB132BF16 zmm1{k1}{z}, zmm2, zmm3/m512/m16bcst	A	V/V	AVX10.2
EVEX.128.NP.MAP6.W0 AA /r VFMSUB213BF16 xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	A	V/V	AVX10.2
EVEX.256.NP.MAP6.W0 AA /r VFMSUB213BF16 ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst	A	V/V	AVX10.2
EVEX.512.NP.MAP6.W0 AA /r VFMSUB213BF16 zmm1{k1}{z}, zmm2, zmm3/m512/m16bcst	A	V/V	AVX10.2
EVEX.128.NP.MAP6.W0 BA /r VFMSUB231BF16 xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	A	V/V	AVX10.2

Continued on next page...

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.NP.MAP6.W0 BA /r VFMSUB231BF16 ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst	A	V/V	AVX10.2
EVEX.512.NP.MAP6.W0 BA /r VFMSUB231BF16 zmm1{k1}{z}, zmm2, zmm3/m512/m16bcst	A	V/V	AVX10.2
EVEX.128.NP.MAP6.W0 9C /r VFNMADD132BF16 xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	A	V/V	AVX10.2
EVEX.256.NP.MAP6.W0 9C /r VFNMADD132BF16 ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst	A	V/V	AVX10.2
EVEX.512.NP.MAP6.W0 9C /r VFNMADD132BF16 zmm1{k1}{z}, zmm2, zmm3/m512/m16bcst	A	V/V	AVX10.2
EVEX.128.NP.MAP6.W0 AC /r VFNMADD213BF16 xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	A	V/V	AVX10.2
EVEX.256.NP.MAP6.W0 AC /r VFNMADD213BF16 ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst	A	V/V	AVX10.2
EVEX.512.NP.MAP6.W0 AC /r VFNMADD213BF16 zmm1{k1}{z}, zmm2, zmm3/m512/m16bcst	A	V/V	AVX10.2
EVEX.128.NP.MAP6.W0 BC /r VFNMADD231BF16 xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	A	V/V	AVX10.2
EVEX.256.NP.MAP6.W0 BC /r VFNMADD231BF16 ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst	A	V/V	AVX10.2
EVEX.512.NP.MAP6.W0 BC /r VFNMADD231BF16 zmm1{k1}{z}, zmm2, zmm3/m512/m16bcst	A	V/V	AVX10.2
EVEX.128.NP.MAP6.W0 9E /r VFNMSUB132BF16 xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	A	V/V	AVX10.2
EVEX.256.NP.MAP6.W0 9E /r VFNMSUB132BF16 ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst	A	V/V	AVX10.2
EVEX.512.NP.MAP6.W0 9E /r VFNMSUB132BF16 zmm1{k1}{z}, zmm2, zmm3/m512/m16bcst	A	V/V	AVX10.2
EVEX.128.NP.MAP6.W0 AE /r VFNMSUB213BF16 xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	A	V/V	AVX10.2
EVEX.256.NP.MAP6.W0 AE /r VFNMSUB213BF16 ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst	A	V/V	AVX10.2
EVEX.512.NP.MAP6.W0 AE /r VFNMSUB213BF16 zmm1{k1}{z}, zmm2, zmm3/m512/m16bcst	A	V/V	AVX10.2

Continued on next page...

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.128.NP.MAP6.W0 BE /r VFNMSUB231BF16 xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	A	V/V	AVX10.2
EVEX.256.NP.MAP6.W0 BE /r VFNMSUB231BF16 ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst	A	V/V	AVX10.2
EVEX.512.NP.MAP6.W0 BE /r VFNMSUB231BF16 zmm1{k1}{z}, zmm2, zmm3/m512/m16bcst	A	V/V	AVX10.2

7.5.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(rw)	VVVV(r)	MODRM.R/M(r)	N/A

7.5.2 DESCRIPTION

Note:

For instructions with a CPUID feature flag specifying Intel® AVX10, the programmer must check the available vector options on the processor at run-time via the VL256 and VL512 bits in Converged Vector ISA Leaf 0x24. These bits enumerate whether the corresponding vector lengths are supported (128-bit vectors are always supported) and as such will determine the set of instructions available to the programmer listed in the above opcode table.

This instruction performs a packed multiply-add, multiply-subtract, negated multiply-add or negated multiply-subtract computation on BF16 values using three source operands and writes the results in the destination operand. The destination operand is also the first source operand. The “N” (negated) forms of this instruction add/subtract the remaining operand to/from the negated infinite precision intermediate product. The notation’ “132”, “213” and “231” indicate the use of the operands in ±A * B - C, where each digit corresponds to the operand number, with the destination being operand 1. The destination elements are updated according to the writemask. This instruction does not generate floating point exceptions and does not consult or update MXCSR. Denormal BF16 input operands are treated as zeros (DAZ) and denormal BF16 outputs are flushed to zero (FTZ). Rounding Mode is always RNE.

Notation	Operands
132	dest = ± dest*src3-src2
231	dest = ± src2*src3-dest
213	dest = ± src2*dest-src3

Table 7.1: VF[,N]M[ADD,SUB][132,213,231]BF16 Notation for Operands

7.5.3 OPERATION

```
VF[,N]M[ADD,SUB][132,213,231]BF16 (EVEX encoded versions) when src3 operand is
a register
(KL, VL) = (8, 128), (16, 256), (32, 512)

IF *132 form*:
    a := DEST
    b := SRC3
    c := SRC2
ELIF *213 form*:
    a := SRC2
    b := DEST
    c := SRC3
ELIF *231 form*:
    a := SRC2
    b := SRC3
    c := DEST

IF *negative form*:
    a := -a

IF *add form*:
    OP := +
ELIF *sub form*:
    OP := -

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        //DAZ, FTZ, SAE
        DEST.bf16[j] := RoundFPControl_RNE(a.bf16[j]*b.bf16[j] OP c.bf16[j])
    ELSE IF *zeroing*:
        DEST.bf16[j] := 0
    // else dest.bf16[j] remains unchanged

DEST[MAX_VL-1:VL] := 0
```

```
VF[,N]M[ADD,SUB][132,213,231]BF16 (EVEX encoded versions) when src3 operand
is a memory source
(KL, VL) = (8, 128), (16, 256), (32, 512)

IF *132 form*:
    a := DEST
    b := SRC3
    c := SRC2
ELIF *213 form*:
    a := SRC2
    b := DEST
    c := SRC3
ELIF *231 form*:
    a := SRC2
    b := SRC3
    c := DEST

IF *negative form*:
    a := -a

IF *add form*:
    OP := +
ELIF *sub form*:
    OP := -

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF EVEX.b == 1:
            //DAZ, FTZ, SAE
            DEST.bf16[j] := RoundFPControl_RNE(a.bf16[j]*b.bf16[j] OP c.bf16[0])
        ELSE:
            //DAZ, FTZ, SAE
            DEST.bf16[j] := RoundFPControl_RNE(a.bf16[j]*b.bf16[j] OP c.bf16[j])
    ELSE IF *zeroing*:
        DEST.bf16[j] := 0
    // else dest.bf16[j] remains unchanged

DEST[MAX_VL-1:VL] := 0
```

7.5.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VFMADD132BF16 xmm1, xmm2, xmm3/m128	E4	N/A	AVX10.2
VFMADD132BF16 ymm1, ymm2, ymm3/m256	E4	N/A	AVX10.2
VFMADD132BF16 zmm1, zmm2, zmm3/m512	E4	N/A	AVX10.2
VFMADD213BF16 xmm1, xmm2, xmm3/m128	E4	N/A	AVX10.2
VFMADD213BF16 ymm1, ymm2, ymm3/m256	E4	N/A	AVX10.2
VFMADD213BF16 zmm1, zmm2, zmm3/m512	E4	N/A	AVX10.2
VFMADD231BF16 xmm1, xmm2, xmm3/m128	E4	N/A	AVX10.2
VFMADD231BF16 ymm1, ymm2, ymm3/m256	E4	N/A	AVX10.2
VFMADD231BF16 zmm1, zmm2, zmm3/m512	E4	N/A	AVX10.2
VFMSUB132BF16 xmm1, xmm2, xmm3/m128	E4	N/A	AVX10.2
VFMSUB132BF16 ymm1, ymm2, ymm3/m256	E4	N/A	AVX10.2
VFMSUB132BF16 zmm1, zmm2, zmm3/m512	E4	N/A	AVX10.2
VFMSUB213BF16 xmm1, xmm2, xmm3/m128	E4	N/A	AVX10.2
VFMSUB213BF16 ymm1, ymm2, ymm3/m256	E4	N/A	AVX10.2
VFMSUB213BF16 zmm1, zmm2, zmm3/m512	E4	N/A	AVX10.2

Continued on next page...

Instruction	Exception Type	Arithmetic Flags	CPUID
VFMSUB231BF16 xmm1, xmm2, xmm3/m128	E4	N/A	AVX10.2
VFMSUB231BF16 ymm1, ymm2, ymm3/m256	E4	N/A	AVX10.2
VFMSUB231BF16 zmm1, zmm2, zmm3/m512	E4	N/A	AVX10.2
VFNMADD132BF16 xmm1, xmm2, xmm3/m128	E4	N/A	AVX10.2
VFNMADD132BF16 ymm1, ymm2, ymm3/m256	E4	N/A	AVX10.2
VFNMADD132BF16 zmm1, zmm2, zmm3/m512	E4	N/A	AVX10.2
VFNMADD213BF16 xmm1, xmm2, xmm3/m128	E4	N/A	AVX10.2
VFNMADD213BF16 ymm1, ymm2, ymm3/m256	E4	N/A	AVX10.2
VFNMADD213BF16 zmm1, zmm2, zmm3/m512	E4	N/A	AVX10.2
VFNMADD231BF16 xmm1, xmm2, xmm3/m128	E4	N/A	AVX10.2
VFNMADD231BF16 ymm1, ymm2, ymm3/m256	E4	N/A	AVX10.2
VFNMADD231BF16 zmm1, zmm2, zmm3/m512	E4	N/A	AVX10.2
VFNMSUB132BF16 xmm1, xmm2, xmm3/m128	E4	N/A	AVX10.2
VFNMSUB132BF16 ymm1, ymm2, ymm3/m256	E4	N/A	AVX10.2
VFNMSUB132BF16 zmm1, zmm2, zmm3/m512	E4	N/A	AVX10.2

Continued on next page...

Instruction	Exception Type	Arithmetic Flags	CPUID
VFNMSUB213BF16 xmm1, xmm2, xmm3/m128	E4	N/A	AVX10.2
VFNMSUB213BF16 ymm1, ymm2, ymm3/m256	E4	N/A	AVX10.2
VFNMSUB213BF16 zmm1, zmm2, zmm3/m512	E4	N/A	AVX10.2
VFNMSUB231BF16 xmm1, xmm2, xmm3/m128	E4	N/A	AVX10.2
VFNMSUB231BF16 ymm1, ymm2, ymm3/m256	E4	N/A	AVX10.2
VFNMSUB231BF16 zmm1, zmm2, zmm3/m512	E4	N/A	AVX10.2

7.6 VFPCLASSBF16

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.128.F2.0F3A.W0 66 /r /ib VFPCLASSBF16 k1{k2}, xmm2/m128/m16bcst, imm8	A	V/V	AVX10.2
EVEX.256.F2.0F3A.W0 66 /r /ib VFPCLASSBF16 k1{k2}, ymm2/m256/m16bcst, imm8	A	V/V	AVX10.2
EVEX.512.F2.0F3A.W0 66 /r /ib VFPCLASSBF16 k1{k2}, zmm2/m512/m16bcst, imm8	A	V/V	AVX10.2

7.6.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(w)	MODRM.R/M(r)	IMM8(r)	N/A

7.6.2 DESCRIPTION

Note:

For instructions with a CPUID feature flag specifying Intel® AVX10, the programmer must check the available vector options on the processor at run-time via the VL256 and VL512 bits in Converged Vector ISA Leaf 0x24. These bits enumerate whether the corresponding vector lengths are supported (128-bit vectors are always supported) and as such will determine the set of instructions available to the programmer listed in the above opcode table.

The VFPCLASSBF16 instruction checks the packed bfloat16 floating point values for special categories, specified by the set bits in the imm8 byte. Each set bit in imm8 specifies a category of floating-point values that the input data element is classified against. The classified results of all specified categories of an input value are ORed together to form the final boolean result for the input element. The result of each element is written to the corresponding bit in a mask register k2 according to the writemask k1. Bits [MAX_KL-1:32/16/8] of the destination are cleared. This instruction does not generate floating point exceptions and does not consult or update MXCSR. Denormal BF16 input operands are treated as zeros (DAZ).

The source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 16-bit memory location.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

7.6.3 OPERATION

```
def check_fp_class_bf16(src, imm8):
    negative := src[15]
    exponent_all_ones := (src[14:7] == 0xFF)
    exponent_all_zeros := (src[14:7] == 0)
    IF(exponent_all_zeros):
        mantissa_all_zeros:=1
    ELSEIF (src[6:0] == 0 OR exponent_all_zeros):
        mantissa_all_zeros := 1
    zero := exponent_all_zeros and mantissa_all_zeros
    signaling_bit := src[6]

    snan := exponent_all_ones and not(mantissa_all_zeros) and not(signaling_bit)
    qnan := exponent_all_ones and not(mantissa_all_zeros) and signaling_bit
    positive_zero := not(negative) and exponent_all_zeros and mantissa_all_zeros
    negative_zero := negative and exponent_all_zeros and mantissa_all_zeros
    positive_infinity := not(negative) and exponent_all_ones and mantissa_all_zeros
    negative_infinity := negative and exponent_all_ones and mantissa_all_zeros
    denormal := exponent_all_zeros and not(mantissa_all_zeros)
    finite_negative := negative and not(exponent_all_ones) and not(zero)

    return (imm8[0] and qnan) OR
           (imm8[1] and positive_zero) OR
           (imm8[2] and negative_zero) OR
           (imm8[3] and positive_infinity) OR
           (imm8[4] and negative_infinity) OR
           (denormal) OR
           (imm8[6] and finite_negative) OR
           (imm8[7] and snan)
```

```
VFPCLASSBF16 destk2k1, src, imm8
(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF SRC is memory and (EVEX.b == 1):
            tsrc := SRC.bf16[0]
        ELSE:
            tsrc := SRC.bf16[j]
        DEST.bit[j] := check_fp_class_bf16_(tsrc, imm8)
    ELSE // zero masking only
        DEST.bit[j] := 0

DEST[MAX_KL-1:KL] := 0
```

7.6.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VFPCLASSBF16 k1, xmm2/m128, imm8	E4	N/A	AVX10.2
VFPCLASSBF16 k1, ymm2/m256, imm8	E4	N/A	AVX10.2
VFPCLASSBF16 k1, zmm2/m512, imm8	E4	N/A	AVX10.2

7.7 VGETEXPBF16

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.128.NP.MAP6.W0 42 /r VGETEXPBF16 xmm1{k1}{z}, xmm2/m128/m16bcst	A	V/V	AVX10.2
EVEX.256.NP.MAP6.W0 42 /r VGETEXPBF16 ymm1{k1}{z}, ymm2/m256/m16bcst	A	V/V	AVX10.2
EVEX.512.NP.MAP6.W0 42 /r VGETEXPBF16 zmm1{k1}{z}, zmm2/m512/m16bcst	A	V/V	AVX10.2

7.7.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(w)	MODRM.R/M(r)	N/A	N/A

7.7.2 DESCRIPTION

Note:

For instructions with a CPUID feature flag specifying Intel® AVX10, the programmer must check the available vector options on the processor at run-time via the VL256 and VL512 bits in Converged Vector ISA Leaf 0x24. These bits enumerate whether the corresponding vector lengths are supported (128-bit vectors are always supported) and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Extracts the biased exponents from the each bfloat16 data element of the source operand (the second operand) as unbiased signed integer value. Each integer value of the unbiased exponent is converted to bfloat16 FP value and written to the corresponding bfloat16 elements of the destination operand (the first operand) as bfloat16 FP numbers.

The destination operand is a ZMM/YMM/XMM register and updated under the writemask. The source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 16-bit memory location.

EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Each GETEXP operation converts the exponent value into a bfloat16 number.

The formula is: GETEXP(x) = floor($\log_2$(|x|))

Notation floor(x) stands for maximal integer not exceeding real number x.

This instruction does not generate floating point exceptions and does not consult or update MXCSR. Denormal BF16 input operands are treated as zeros (DAZ)

7.7.3 OPERATION

```
def getexp_bf16(src):
    IF (*src is nan*):
        return QNAN(src)
    ELIF (*src is positive infinity*):
        return INF
    ELIF (*src is denormal or zero*):
        return -INF
    ELSE:
        tmp := ((src & 0x7F80) >> 7) //shift arithmetic right
        tmp := tmp - 127 //subtract bias
        return convert_integer_to_bf16(tmp)
```

```
VGETEXPBF16 destk1, src
(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF SRC is memory and (EVEX.b == 1):
            tsrc := SRC.bf16[0]
        ELSE:
            tsrc := SRC.bf16[j]
        DEST.bf16[j] := getexp_bf16(tsrc)
    ELSE IF *zeroing*:
        DEST.bf16[j] := 0
    // else dest.bf16[j] remains unchanged

DEST[MAX_VL-1:VL] := 0
```

7.7.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VGETEXPBF16 xmm1, xmm2/m128	E4	N/A	AVX10.2
VGETEXPBF16 ymm1, ymm2/m256	E4	N/A	AVX10.2

Continued on next page...

Instruction	Exception Type	Arithmetic Flags	CPUID
VGETEXPBF16 zmm1, zmm2/m512	E4	N/A	AVX10.2

7.8 VGETMANTBF16

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.128.F2.0F3A.W0 26 /r /ib VGETMANTBF16 xmm1{k1}{z}, xmm2/m128/m16bcst, imm8	A	V/V	AVX10.2
EVEX.256.F2.0F3A.W0 26 /r /ib VGETMANTBF16 ymm1{k1}{z}, ymm2/m256/m16bcst, imm8	A	V/V	AVX10.2
EVEX.512.F2.0F3A.W0 26 /r /ib VGETMANTBF16 zmm1{k1}{z}, zmm2/m512/m16bcst, imm8	A	V/V	AVX10.2

7.8.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(w)	MODRM.R/M(r)	IMM8(r)	N/A

7.8.2 DESCRIPTION

Note:

For instructions with a CPUID feature flag specifying Intel® AVX10, the programmer must check the available vector options on the processor at run-time via the VL256 and VL512 bits in Converged Vector ISA Leaf 0x24. These bits enumerate whether the corresponding vector lengths are supported (128-bit vectors are always supported) and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Convert bfloat16 floating values in the source operand (the second operand) to bfloat16 FP values with the mantissa normalization specified by the imm8. The converted results are written to the destination operand (the first operand) using writemask k1. The normalized mantissa is specified by interv (imm8[1:0]) and the sign control (sc) is specified by bits 3:2 of the immediate byte.

The destination operand is a ZMM/YMM/XMM register updated under the writemask. The source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 16-bit memory location.

For each input SP FP value x, The conversion operation is: GetMant(x) = $\pm 2^{k}|x.significand|$ where:

1 <= |x.significand| < 2

Unbiased exponent k depends on the interval range defined by interv and whether the exponent of the source is even or odd. The sign of the final result is determined by sc and the source sign.

If interv != 0 then k = -1, otherwise k = 0.

Each converted bfloat16 FP result is encoded according to the sign control, the unbiased exponent k (adding bias) and a mantissa normalized to the range specified by interv.

This instruction is writemasked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1.

This instruction does not generate floating point exceptions and does not consult or update MXCSR. Denormal BF16 input operands are treated as zeros (DAZ)

Note: EVEX.vvvv is reserved and must be 1111b; otherwise instructions will #UD.

7.8.3 OPERATION

```
def getmant_bf16(src, sign_control, normalization_interval):
    dst.sign := sign_control[0] ? 0 : src.sign
    signed_one := sign_control[0] ? +1.0 : -1.0
    dst.exp := src.exp
    dst.fraction := src.fraction
    bias := 127

    IF (*src is nan*):
        return QNAN(src)
    ELIF (*src is positive zero or positive infinity*):
        return 1.0
    ELIF (*src is negative*):
        IF (*src is zero*):
            return signed_one
        ELIF (*src is infinity*):
            IF (sign_control[1]):
                return QNaN_Indefinite
            ELSE:
                return signed_one
        ELIF (sign_control[1]):
            return QNaN_Indefinite
    IF (*src is denormal*):
        dst.fraction := 0

    unbiased_exp := dst.exp - bias
    odd_exp := unbiased_exp[0]
    signaling_bit := dst.fraction[6]

    IF (normalization_interval := 0b00):
        dst.exp := bias
    ELIF (normalization_interval := 0b01):
        dst.exp := odd_exp ? bias-1 : bias
    ELIF (normalization_interval := 0b10):
        dst.exp := bias-1
    ELIF (normalization_interval := 0b11):
        dst.exp := signaling_bit ? bias-1 : bias

    return dst
```

```
VGETMANTBF16 destk1, src, imm8
(KL, VL) = (8, 128), (16, 256), (32, 512)

sign_control := imm8[3:2]
normalization_interval := imm8[1:0]

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF SRC is memory and (EVEX.b == 1):
            tsrc := SRC.bf16[0]
        ELSE:
            tsrc := SRC.bf16[j]
        DEST.bf16[j] := getmant_bf16(tsrc, sign_control, normalization_interval)
    ELSE IF *zeroing*:
        DEST.bf16[j] := 0
    // else dest.bf16[j] remains unchanged

DEST[MAX_VL-1:VL] := 0
```

7.8.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VGETMANTBF16 xmm1, xmm2/m128, imm8	E4	N/A	AVX10.2
VGETMANTBF16 ymm1, ymm2/m256, imm8	E4	N/A	AVX10.2
VGETMANTBF16 zmm1, zmm2/m512, imm8	E4	N/A	AVX10.2

7.9 VMAXBF16

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.128.66.MAP5.W0 5F /r VMAXBF16 xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	A	V/V	AVX10.2
EVEX.256.66.MAP5.W0 5F /r VMAXBF16 ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst	A	V/V	AVX10.2
EVEX.512.66.MAP5.W0 5F /r VMAXBF16 zmm1{k1}{z}, zmm2, zmm3/m512/m16bcst	A	V/V	AVX10.2

7.9.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(w)	VVVV(r)	MODRM.R/M(r)	N/A

7.9.2 DESCRIPTION

Note:

For instructions with a CPUID feature flag specifying Intel® AVX10, the programmer must check the available vector options on the processor at run-time via the VL256 and VL512 bits in Converged Vector ISA Leaf 0x24. These bits enumerate whether the corresponding vector lengths are supported (128-bit vectors are always supported) and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Performs a SIMD compare of the packed half-precision floating-point values in the first source operand and the second source operand and returns the maximum value for each pair of values to the destination operand.

If the values being compared are both 0.0s (of either sign), the value in the second operand (source operand) is returned. If a value in the second operand is an SNaN, then SNaN is forwarded unchanged to the destination (that is, a QNaN version of the SNaN is not returned).

If only one value is a NaN (SNaN or QNaN) for this instruction, the second operand (source operand), either a NaN or a valid floating-point value, is written to the result. If instead of this behavior, it is required that the NaN source operand (from either the first or second operand) be returned, the action of VMAXBF16 can be emulated using a sequence of instructions, such as, a comparison followed by AND, ANDN and OR. This instruction does not generate floating point exceptions and does not consult or update MXCSR. Denormal BF16 input operands are treated as zeros (DAZ)

EVEX encoded versions: The first source operand (the second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 16-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

7.9.3 OPERATION

```
DEFINE MAX(SRC1, SRC2):
    IF (SRC1 = 0.0) and (SRC2 = 0.0):
        DEST := SRC2
    ELSE IF (SRC1 = NaN):
        DEST := SRC2
    ELSE IF (SRC2 = NaN):
        DEST := SRC2
    ELSE IF (SRC1 > SRC2):
        DEST := SRC1
    ELSE:
        DEST := SRC2


VMAXPFB16 (EVEX encoded versions)
(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF EVEX.b == 1:
            tsrc2 := SRC2.bf16[0]
        ELSE:
            tsrc2 := SRC2.bf16[j]
        DEST.bf16[j] := MAX(SRC1.bf16[j], tsrc2) //DAZ, SAE
    ELSE IF *zeroing*:
        DEST.bf16[j] := 0
    // else dest.bf16[j] remains unchanged

DEST[MAX_VL-1:VL] := 0
```

7.9.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VMAXBF16 xmm1, xmm2, xmm3/m128	E4	N/A	AVX10.2
VMAXBF16 ymm1, ymm2, ymm3/m256	E4	N/A	AVX10.2

Continued on next page...

Instruction	Exception Type	Arithmetic Flags	CPUID
VMAXBF16 zmm1, zmm2, zmm3/m512	E4	N/A	AVX10.2

7.10 VMINBF16

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.128.66.MAP5.W0 5D /r VMINBF16 xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	A	V/V	AVX10.2
EVEX.256.66.MAP5.W0 5D /r VMINBF16 ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst	A	V/V	AVX10.2
EVEX.512.66.MAP5.W0 5D /r VMINBF16 zmm1{k1}{z}, zmm2, zmm3/m512/m16bcst	A	V/V	AVX10.2

7.10.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(w)	VVVV(r)	MODRM.R/M(r)	N/A

7.10.2 DESCRIPTION

Note:

For instructions with a CPUID feature flag specifying Intel® AVX10, the programmer must check the available vector options on the processor at run-time via the VL256 and VL512 bits in Converged Vector ISA Leaf 0x24. These bits enumerate whether the corresponding vector lengths are supported (128-bit vectors are always supported) and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Performs a SIMD compare of the packed half-precision floating-point values in the first source operand and the second source operand and returns the minimum value for each pair of values to the destination operand.

If the values being compared are both 0.0s (of either sign), the value in the second operand (source operand) is returned. If a value in the second operand is an SNaN, then SNaN is forwarded unchanged to the destination (that is, a QNaN version of the SNaN is not returned).

If only one value is a NaN (SNaN or QNaN) for this instruction, the second operand (source operand), either a NaN or a valid floating-point value, is written to the result. If instead of this behavior, it is required that the NaN source operand (from either the first or second operand) be returned, the action of VMINBF16 can be emulated using a sequence of instructions, such as, a comparison followed by AND, ANDN and OR. This instruction does not generate floating point exceptions and does not consult or update MXCSR. Denormal BF16 input operands are treated as zeros (DAZ)

EVEX encoded versions: The first source operand (the second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 16-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

7.10.3 OPERATION

```
DEFINE MIN(SRC1, SRC2):
    IF (SRC1 = 0.0) and (SRC2 = 0.0):
        DEST := SRC2
    ELSE IF (SRC1 = NaN):
        DEST := SRC2
    ELSE IF (SRC2 = NaN):
        DEST := SRC2
    ELSE IF (SRC1 < SRC2):
        DEST := SRC1
    ELSE:
        DEST := SRC2


VMINPFB16 (EVEX encoded versions)
(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF EVEX.b == 1:
            tsrc2 := SRC2.bf16[0]
        ELSE:
            tsrc2 := SRC2.bf16[j]
        DEST.bf16[j] := MIN(SRC1.bf16[j], tsrc2) //DAZ, SAE
    ELSE IF *zeroing*:
        DEST.bf16[j] := 0
    // else dest.bf16[j] remains unchanged

DEST[MAX_VL-1:VL] := 0
```

7.10.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VMINBF16 xmm1, xmm2, xmm3/m128	E4	N/A	AVX10.2
VMINBF16 ymm1, ymm2, ymm3/m256	E4	N/A	AVX10.2

Continued on next page...

Instruction	Exception Type	Arithmetic Flags	CPUID
VMINBF16 zmm1, zmm2, zmm3/m512	E4	N/A	AVX10.2

7.11 VMULBF16

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.128.66.MAP5.W0 59 /r VMULBF16 xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	A	V/V	AVX10.2
EVEX.256.66.MAP5.W0 59 /r VMULBF16 ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst	A	V/V	AVX10.2
EVEX.512.66.MAP5.W0 59 /r VMULBF16 zmm1{k1}{z}, zmm2, zmm3/m512/m16bcst	A	V/V	AVX10.2

7.11.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(w)	VVVV(r)	MODRM.R/M(r)	N/A

7.11.2 DESCRIPTION

Note:

For instructions with a CPUID feature flag specifying Intel® AVX10, the programmer must check the available vector options on the processor at run-time via the VL256 and VL512 bits in Converged Vector ISA Leaf 0x24. These bits enumerate whether the corresponding vector lengths are supported (128-bit vectors are always supported) and as such will determine the set of instructions available to the programmer listed in the above opcode table.

This instruction multiplies packed BF16 values from source operands and stores the packed BF16 result in the destination operand. The destination elements are updated according to the writemask. This instruction does not generate floating point exceptions and does not consult or update MXCSR. Denormal BF16 input operands are treated as zeros (DAZ) and denormal BF16 outputs are flushed to zero (FTZ). Rounding Mode is always RNE.

7.11.3 OPERATION

```
VMULBF16 (EVEX encoded versions) when src2 operand is a register
(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        DEST.bf16[j] := SRC1.bf16[j] * SRC2.bf16[j] //DAZ, FTZ, RNE, SAE
    ELSE IF *zeroing*:
        DEST.bf16[j] := 0
    // else dest.bf16[j] remains unchanged

DEST[MAX_VL-1:VL] := 0
```

```
VMULBF16 (EVEX encoded versions) when src2 operand is a memory source
(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF EVEX.b == 1:
            DEST.bf16[j] := SRC1.bf16[j] * SRC2.bf16[0] //DAZ, FTZ, RNE, SAE
        ELSE:
            DEST.bf16[j] := SRC1.bf16[j] * SRC2.bf16[j] //DAZ, FTZ, RNE, SAE

    ELSE IF *zeroing*:
        DEST.bf16[j] := 0
    // else dest.bf16[j] remains unchanged

DEST[MAX_VL-1:VL] := 0
```

7.11.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VMULBF16 xmm1, xmm2, xmm3/m128	E4	N/A	AVX10.2
VMULBF16 ymm1, ymm2, ymm3/m256	E4	N/A	AVX10.2
VMULBF16 zmm1, zmm2, zmm3/m512	E4	N/A	AVX10.2

7.12 VRCPBF16

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.128.NP.MAP6.W0 4C /r VRCPBF16 xmm1{k1}{z}, xmm2/m128/m16bcst	A	V/V	AVX10.2
EVEX.256.NP.MAP6.W0 4C /r VRCPBF16 ymm1{k1}{z}, ymm2/m256/m16bcst	A	V/V	AVX10.2
EVEX.512.NP.MAP6.W0 4C /r VRCPBF16 zmm1{k1}{z}, zmm2/m512/m16bcst	A	V/V	AVX10.2

7.12.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(w)	MODRM.R/M(r)	N/A	N/A

7.12.2 DESCRIPTION

Note:

For instructions with a CPUID feature flag specifying Intel® AVX10, the programmer must check the available vector options on the processor at run-time via the VL256 and VL512 bits in Converged Vector ISA Leaf 0x24. These bits enumerate whether the corresponding vector lengths are supported (128-bit vectors are always supported) and as such will determine the set of instructions available to the programmer listed in the above opcode table.

This instruction performs a SIMD computation of the approximate reciprocals of 8/16/32 packed BF16 values in the source operand (the second operand) and stores the packed BF16 results in the destination operand. The maximum relative error for this approximation is less than $2^{-8} + 2^{-14}$. For special cases, see Table 7.2.

Input Value	Result Value	Comments
$0 <= X < 2^{-126}$	+INF	DAZ
$-2^{-126} < X <= 0$	-INF	DAZ
X = +INF	+0	
X = -INF	-0	
$X = 2^{-n}$	2^{n}	
$X = -2^{-n}$	-2^{n}	

Table 7.2: VRCPBF16 Special Cases

7.12.3 OPERATION

```
VRCPBF16 destk1, src
(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF SRC is memory and (EVEX.b == 1):
            tsrc := SRC.bf16[0]
        ELSE:
            tsrc := SRC.bf16[j]
        DEST.bf16[j] := APPROXIMATE(1.0 / tsrc) //DAZ, FTZ, SAE
    ELSE IF *zeroing*:
        DEST.bf16[j] := 0
    // else dest.bf16[j] remains unchanged

DEST[MAX_VL-1:VL] := 0
```

7.12.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VRCPBF16 xmm1, xmm2/m128	E4	N/A	AVX10.2
VRCPBF16 ymm1, ymm2/m256	E4	N/A	AVX10.2
VRCPBF16 zmm1, zmm2/m512	E4	N/A	AVX10.2

7.13 VREDUCEBF16

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.128.F2.0F3A.W0 56 /r /ib VREDUCEBF16 xmm1{k1}{z}, xmm2/m128/m16bcst, imm8	A	V/V	AVX10.2
EVEX.256.F2.0F3A.W0 56 /r /ib VREDUCEBF16 ymm1{k1}{z}, ymm2/m256/m16bcst, imm8	A	V/V	AVX10.2
EVEX.512.F2.0F3A.W0 56 /r /ib VREDUCEBF16 zmm1{k1}{z}, zmm2/m512/m16bcst, imm8	A	V/V	AVX10.2

7.13.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(w)	MODRM.R/M(r)	IMM8(r)	N/A

7.13.2 DESCRIPTION

Note:

For instructions with a CPUID feature flag specifying Intel® AVX10, the programmer must check the available vector options on the processor at run-time via the VL256 and VL512 bits in Converged Vector ISA Leaf 0x24. These bits enumerate whether the corresponding vector lengths are supported (128-bit vectors are always supported) and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Extracts the reduced argument of bfloat16 Floating-Point values in the first source operand by the number of bits specified in the immediate operand (imm8) and places the result in the destination operand.

The reduced argument extraction is formulated below:

zmm1 := zmm2 – (ROUND(2^{M}*zmm2))*2^{-M};

Given zmm2 = 2^{exp2}*man2
Then 0<=|zmm1|<$2^{exp2-M-1}$

The scaling value M is determined by the imm8[7:4].

The operation is write masked.

7.13.3 OPERATION

```
def reduce_bf16_ne(src,imm8):
    IF (*src is nan*):
        return QNAN(src)
    m := imm8[7:4]
    tmp := 2^(-m) * ROUND(2^m * src, RNE) //DAZ, SAE
    tmp := src - tmp //FTZ, RNE, SAE
    return tmp
```

```
VREDUCEBF16 destk1, src, imm8
(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF SRC is memory and (EVEX.b == 1):
            tsrc := SRC.bf16[0]
        ELSE:
            tsrc := SRC.bf16[j]
        DEST.bf16[j] := reduce_bf16_ne(tsrc, imm8)
    ELSE IF *zeroing*:
        DEST.bf16[j] := 0
    // else dest.bf16[j] remains unchanged

DEST[MAX_VL-1:VL] := 0
```

7.13.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VREDUCEBF16 xmm1, xmm2/m128, imm8	E4	N/A	AVX10.2
VREDUCEBF16 ymm1, ymm2/m256, imm8	E4	N/A	AVX10.2
VREDUCEBF16 zmm1, zmm2/m512, imm8	E4	N/A	AVX10.2

7.14 VRNDSCALEBF16

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.128.F2.0F3A.W0 08 /r /ib VRNDSCALEBF16 xmm1{k1}{z}, xmm2/m128/m16bcst, imm8	A	V/V	AVX10.2
EVEX.256.F2.0F3A.W0 08 /r /ib VRNDSCALEBF16 ymm1{k1}{z}, ymm2/m256/m16bcst, imm8	A	V/V	AVX10.2
EVEX.512.F2.0F3A.W0 08 /r /ib VRNDSCALEBF16 zmm1{k1}{z}, zmm2/m512/m16bcst, imm8	A	V/V	AVX10.2

7.14.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(w)	MODRM.R/M(r)	IMM8(r)	N/A

7.14.2 DESCRIPTION

Note:

For instructions with a CPUID feature flag specifying Intel® AVX10, the programmer must check the available vector options on the processor at run-time via the VL256 and VL512 bits in Converged Vector ISA Leaf 0x24. These bits enumerate whether the corresponding vector lengths are supported (128-bit vectors are always supported) and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Round the bfloat16 floating-point values in the source operand by the rounding mode specified in the immediate operand and places the result in the destination operand.

The destination operand (the first operand) is a ZMM/YMM/XMM register conditionally updated according to the writemask. The source operand (the second operand) can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 16-bit memory location.

The rounding process rounds the input to an integral value, plus number bits of fraction that are specified by imm8[7:4] (to be included in the result) and returns the result as a bfloat16 floating-point value. RNE rounding mode is used.

If any source operand is an SNaN then it will be converted to a QNaN. Denormals will be converted to zero

before rounding. The sign of the result of this instruction is preserved, including the sign of zero. The formula of the operation on each data element for VRNDSCALEBF16 is

ROUND(x) = 2^{-M}*Round_to_INT(x*2^{M}, RNE), M=imm[7:4];

The operation of x*2^{M} is computed as if the exponent range is unlimited (i.e. no overflow ever occurs).

Note: EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

7.14.3 OPERATION

```
def round_bf16_to_integer_ne(src,imm8):
    IF (*src is nan*):
        return QNAN(src)
    m := imm8[7:4]
    tmp := ROUND_TO_NEAREST_EVEN_INTEGER(2^m * src) //DAZ, SAE
    tmp := 2^(-m) * tmp //FTZ, RNE, SAE
    return tmp
```

```
VRNDSCALEBF16 destk1, src, imm8
(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF SRC is memory and (EVEX.b == 1):
            tsrc := SRC.bf16[0]
        ELSE:
            tsrc := SRC.bf16[j]
        DEST.bf16[j] := round_bf16_to_integer_ne(tsrc, imm8)
    ELSE IF *zeroing*:
        DEST.bf16[j] := 0
    // else dest.bf16[j] remains unchanged

DEST[MAX_VL-1:VL] := 0
```

7.14.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VRNDSCALEBF16 xmm1, xmm2/m128, imm8	E4	N/A	AVX10.2

Continued on next page...

Instruction	Exception Type	Arithmetic Flags	CPUID
VRNDSCALEBF16 ymm1, ymm2/m256, imm8	E4	N/A	AVX10.2
VRNDSCALEBF16 zmm1, zmm2/m512, imm8	E4	N/A	AVX10.2

7.15 VRSQRTBF16

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.128.NP.MAP6.W0 4E /r VRSQRTBF16 xmm1{k1}{z}, xmm2/m128/m16bcst	A	V/V	AVX10.2
EVEX.256.NP.MAP6.W0 4E /r VRSQRTBF16 ymm1{k1}{z}, ymm2/m256/m16bcst	A	V/V	AVX10.2
EVEX.512.NP.MAP6.W0 4E /r VRSQRTBF16 zmm1{k1}{z}, zmm2/m512/m16bcst	A	V/V	AVX10.2

7.15.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(w)	MODRM.R/M(r)	N/A	N/A

7.15.2 DESCRIPTION

Note:

For instructions with a CPUID feature flag specifying Intel® AVX10, the programmer must check the available vector options on the processor at run-time via the VL256 and VL512 bits in Converged Vector ISA Leaf 0x24. These bits enumerate whether the corresponding vector lengths are supported (128-bit vectors are always supported) and as such will determine the set of instructions available to the programmer listed in the above opcode table.

This instruction performs a SIMD computation of the approximate reciprocals square-root of 8/16/32 packed BF16 floating-point values in the source operand (the second operand) and stores the packed BF16 floating-point results in the destination operand. The maximum relative error for this approximation is less than $2^{-8} + 2^{-14}$. For special cases, see Table 7.3. The destination elements are updated according to the writemask.

Input Value	Result Value	Comments
$0 <= X < 2^{-126}$	+INF	DAZ
$-2^{-126} < X <= 0$	-INF	DAZ
$X = 2^{-2n}$	2^n	
X < 0	QNaN Indefinite	Including -INF

X = +INF	+0	

Table 7.3: VRSQRTBF16 Special Cases

7.15.3 OPERATION

```
VRSQRTBF16 destk1, src
(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF SRC is memory and (EVEX.b == 1):
            tsrc := SRC.bf16[0]
        ELSE:
            tsrc := SRC.bf16[j]
        DEST.bf16[j] := APPROXIMATE(1.0 / SQRT(tsrc)) //DAZ, FTZ, SAE
    ELSE IF *zeroing*:
        DEST.bf16[j] := 0
    // else dest.bf16[j] remains unchanged

DEST[MAX_VL-1:VL] := 0
```

7.15.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VRSQRTBF16 xmm1, xmm2/m128	E4	N/A	AVX10.2
VRSQRTBF16 ymm1, ymm2/m256	E4	N/A	AVX10.2
VRSQRTBF16 zmm1, zmm2/m512	E4	N/A	AVX10.2

7.16 VSCALEFBF16

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.128.NP.MAP6.W0 2C /r VSCALEFBF16 xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	A	V/V	AVX10.2
EVEX.256.NP.MAP6.W0 2C /r VSCALEFBF16 ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst	A	V/V	AVX10.2
EVEX.512.NP.MAP6.W0 2C /r VSCALEFBF16 zmm1{k1}{z}, zmm2, zmm3/m512/m16bcst	A	V/V	AVX10.2

7.16.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(w)	VVVV(r)	MODRM.R/M(r)	N/A

7.16.2 DESCRIPTION

Note:

For instructions with a CPUID feature flag specifying Intel® AVX10, the programmer must check the available vector options on the processor at run-time via the VL256 and VL512 bits in Converged Vector ISA Leaf 0x24. These bits enumerate whether the corresponding vector lengths are supported (128-bit vectors are always supported) and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Performs a floating-point scale of the packed bfloat16 floating-point values in the first source operand by multiplying it by 2 power of the bfloat16 values in second source operand.

The equation of this operation is given by:

$zmm1 := zmm2 * 2^{floor(zmm3)}$.

Floor(zmm3) means maximum integer value $\leq$ zmm3.

Denormal BF16 input operands are treated as zeros (DAZ) and denormal BF16 outputs are flushed to zero (FTZ).

7.16.3 OPERATION

```
def scale_bf16(src1,src2):
    tmp1 := src1
    tmp2 := src2
    IF (src1 is denormal):
        tmp1 := 0
    IF (src2 is denormal):
        tmp2 := 0
    return tmp1 * POW(2, FLOOR(tmp2)) //FTZ, SAE
```

```
VSCALEFBF16 destk1, src1, src2
(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF SRC2 is memory and (EVEX.b == 1):
            tsrc2 := SRC2.bf16[0]
        ELSE:
            tsrc2 := SRC2.bf16[j]
        DEST.bf16[j] := scale_bf16(src1, tsrc2)
    ELSE IF *zeroing*:
        DEST.bf16[j] := 0
    // else dest.bf16[j] remains unchanged

DEST[MAX_VL-1:VL] := 0
```

7.16.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VSCALEFBF16 xmm1, xmm2, xmm3/m128	E4	N/A	AVX10.2
VSCALEFBF16 ymm1, ymm2, ymm3/m256	E4	N/A	AVX10.2
VSCALEFBF16 zmm1, zmm2, zmm3/m512	E4	N/A	AVX10.2

7.17 VSQRTBF16

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.128.66.MAP5.W0 51 /r VSQRTBF16 xmm1{k1}{z}, xmm2/m128/m16bcst	A	V/V	AVX10.2
EVEX.256.66.MAP5.W0 51 /r VSQRTBF16 ymm1{k1}{z}, ymm2/m256/m16bcst	A	V/V	AVX10.2
EVEX.512.66.MAP5.W0 51 /r VSQRTBF16 zmm1{k1}{z}, zmm2/m512/m16bcst	A	V/V	AVX10.2

7.17.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(w)	MODRM.R/M(r)	N/A	N/A

7.17.2 DESCRIPTION

Note:

For instructions with a CPUID feature flag specifying Intel® AVX10, the programmer must check the available vector options on the processor at run-time via the VL256 and VL512 bits in Converged Vector ISA Leaf 0x24. These bits enumerate whether the corresponding vector lengths are supported (128-bit vectors are always supported) and as such will determine the set of instructions available to the programmer listed in the above opcode table.

This instruction computes the SIMD square root of 8/16/32 packed BF16 floating-point values in the source operand (second operand), rounded to nearest. Outputs for special cases follow the IEEE specification for this operation.

This instruction does not generate floating point exceptions and does not consult or update MXCSR. Denormal bfloat16 input operands are treated as zeros (DAZ) and denormal bfloat16 outputs are flushed to zero (FTZ).

7.17.3 OPERATION

```
VSQRTBF16 dest{k1}, src
(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF SRC is memory and (EVEX.b == 1):
            tsrc := SRC.bf16[0]
        ELSE:
            tsrc := SRC.bf16[j]
        DEST.bf16[j] := SQRT(tsrc) //DAZ, FTZ, RNE, SAE
    ELSE IF *zeroing*:
        DEST.bf16[j] := 0
    // else dest.bf16[j] remains unchanged

DEST[MAX_VL-1:VL] := 0
```

7.17.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VSQRTBF16 xmm1, xmm2/m128	E4	N/A	AVX10.2
VSQRTBF16 ymm1, ymm2/m256	E4	N/A	AVX10.2
VSQRTBF16 zmm1, zmm2/m512	E4	N/A	AVX10.2

7.18 VSUBBF16

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.128.66.MAP5.W0 5C /r VSUBBF16 xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	A	V/V	AVX10.2
EVEX.256.66.MAP5.W0 5C /r VSUBBF16 ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst	A	V/V	AVX10.2
EVEX.512.66.MAP5.W0 5C /r VSUBBF16 zmm1{k1}{z}, zmm2, zmm3/m512/m16bcst	A	V/V	AVX10.2

7.18.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(w)	VVVV(r)	MODRM.R/M(r)	N/A

7.18.2 DESCRIPTION

Note:

For instructions with a CPUID feature flag specifying Intel® AVX10, the programmer must check the available vector options on the processor at run-time via the VL256 and VL512 bits in Converged Vector ISA Leaf 0x24. These bits enumerate whether the corresponding vector lengths are supported (128-bit vectors are always supported) and as such will determine the set of instructions available to the programmer listed in the above opcode table.

This instruction subtracts packed BF16 values from second source operand from the corresponding elements in the first source operand, storing the packed BF16 result in the destination operand. The destination elements are updated according to the writemask This instruction does not generate floating point exceptions and does not consult or update MXCSR. Denormal BF16 input operands are treated as zeros (DAZ) and denormal BF16 outputs are flushed to zero (FTZ). Rounding Mode is always RNE.

7.18.3 OPERATION

```
VSUBBF16 (EVEX encoded versions) when src2 operand is a register
(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        DEST.bf16[j] := SRC1.bf16[j] - SRC2.bf16[j] //DAZ, FTZ, RNE, SAE
    ELSE IF *zeroing*:
        DEST.bf16[j] := 0
    // else dest.bf16[j] remains unchanged

DEST[MAX_VL-1:VL] := 0
```

```
VSUBBF16 (EVEX encoded versions) when src2 operand is a memory source
(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF EVEX.b == 1:
            DEST.bf16[j] := SRC1.bf16[j] - SRC2.bf16[0] //DAZ, FTZ, RNE, SAE
        ELSE:
            DEST.bf16[j] := SRC1.bf16[j] - SRC2.bf16[j] //DAZ, FTZ, RNE, SAE

    ELSE IF *zeroing*:
        DEST.bf16[j] := 0
    // else dest.bf16[j] remains unchanged

DEST[MAX_VL-1:VL] := 0
```

7.18.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VSUBBF16 xmm1, xmm2, xmm3/m128	E4	N/A	AVX10.2
VSUBBF16 ymm1, ymm2, ymm3/m256	E4	N/A	AVX10.2
VSUBBF16 zmm1, zmm2, zmm3/m512	E4	N/A	AVX10.2

Chapter 8

INTEL® AVX10 COMPARE SCALAR FP WITH ENHANCED EFLAGS INSTRUCTIONS

8.1 VCOMXSD

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.LLIG.F2.0F.W1 2F /r VCOMXSD xmm1, xmm2/m64 {sae}	A	V/V	AVX10.2

8.1.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	SCALAR	MODRM.REG(r)	MODRM.R/M(r)	N/A	N/A

8.1.2 DESCRIPTION

Note:

For SCALAR instructions with a CPUID feature flag specifying Intel® AVX10, the programmer can skip the check for the available vector options on the processor at run-time via the VL256 and VL512 bits in Converged Vector ISA Leaf 0x24. These bits enumerate whether the corresponding vector lengths are supported and usually determine the set of instructions available to the programmer. However, scalar instructions are architected without the notion of vector length, so this step does not strictly apply.

VCOMXSD: Compares the double-precision floating-point values in the low quadwords of operand 1 (first operand) and operand 2 (second operand), and sets the OF, SF, ZF, PF, and CF flags in the EFLAGS register according to the result (unordered, greater than, less than, or equal). The AF flag in the EFLAGS register is set to 0. The unordered result is returned if either source operand is a NaN (QNaN or SNaN). Operand 1 is an XMM register; operand 2 can be an XMM register or a 64-bit memory location. The VCOMXSD instruction differs from the VCOMISD instruction in that the EFLAGS register is set differently to allow more relationships to be tested directly, with unsigned integer tests used for testing greater than conditions and signed integer tests used for testing less than. The VCOMXSD instruction differs from the VUCOMXSD instruction in that it signals a SIMD floating-point invalid operation exception (#XM, specifically #I) when a source operand is either a QNaN or SNaN. The VUCOMXSD instruction signals an invalid operation exception only if a source operand is an SNaN. The EFLAGS register is not updated if an unmasked SIMD floating-point exception is generated. EVEX.vvvv are reserved and must be 1111b, otherwise instructions will #UD. Software should ensure VCOMXSD is encoded with EVEX.LL=00b, otherwise instructions will #UD.

VCOMXSD allows any combination of unordered, greater than, less than, and equal to be tested for using a single instruction.

Additionally: #UD If EVEX.vvvv ≠ 1111B, EVEX.vvvv ≠ 1111B, EVEX.L ≠ 0, or EVEX.LL ≠ 00B

Meaning	True	Inverse
Greater than	JA, CMOVA, SETA	JNA, CMOVNA, SETNA
Greater than or equal	JAE, CMOVAE, SETAE	JNAE, CMOVNAE, SETNAE
Equal	JE, CMOVE, SETE	JNE, CMOVNE, SETNE
Less than or equal	JLE, CMOVLE, SETLE	JNLE, CMOVNLE, SETNLE
Less than	JL, CMOVL, SETL	JNL, CMOVNL, SETNL
Ordered	JNP, CMOVNP, SETNP	JP, CMOVP, SETP

Table 8.1: Valid Comparison Tests

8.1.3 OPERATION

```
VCOMXSD (all versions)
RESULT := OrderedCompare(DEST[63:0] != SRC[63:0]) {
(* Set EFLAGS *) CASE (RESULT) OF
        UNORDERED: OF,SF,ZF,PF,CF := 11011;
        GREATER_THAN: OF,SF,ZF,PF,CF := 00000;
        LESS_THAN: OF,SF,ZF,PF,CF := 10001;
        EQUAL: OF,SF,ZF,PF,CF := 11100;
ESAC;
AF := 0; }
```

8.1.4 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Denormal, Invalid In abbreviated form, this includes: DE, IE

8.1.5 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VCOMXSD xmm1, xmm2/m64	E3NF	ID	AVX10.2

8.2 VCOMXSH

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.LLIG.F3.MAP5.W0 2F /r VCOMXSH xmm1, xmm2/m16 {sae}	A	V/V	AVX10.2

8.2.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	SCALAR	MODRM.REG(r)	MODRM.R/M(r)	N/A	N/A

8.2.2 DESCRIPTION

Note:

For SCALAR instructions with a CPUID feature flag specifying Intel® AVX10, the programmer can skip the check for the available vector options on the processor at run-time via the VL256 and VL512 bits in Converged Vector ISA Leaf 0x24. These bits enumerate whether the corresponding vector lengths are supported and usually determine the set of instructions available to the programmer. However, scalar instructions are architected without the notion of vector length, so this step does not strictly apply.

VCOMXSH: Compares the 16 bit floating point values in the low words of operand 1 (first operand) and operand 2 (second operand), and sets the OF, SF, ZF, PF, and CF flags in the EFLAGS register according to the result (unordered, greater than, less than, or equal). The AF flag in the EFLAGS register is set to 0. The unordered result is returned if either source operand is a NaN (QNaN or SNaN). Operand 1 is an XMM register; operand 2 can be an XMM register or a 16-bit memory location. The VCOMXSH instruction differs from the VUCOMISH instruction in that the EFLAGS register is set differently to allow more relationships to be tested directly, with unsigned integer tests used for testing greater than conditions and signed integer tests used for testing less than. The EFLAGS register is not updated if an unmasked SIMD floating-point exception is generated. Note: EVEX.vvvv are reserved and must be 1111b, otherwise instructions will #UD. Software should ensure VCOMXSH is encoded with EVEX.LL=00b, otherwise instructions will #UD.

Additionally: #UD If EVEX.vvvv ≠ 1111B, EVEX.vvvv ≠ 1111B, EVEX.L ≠ 0, or EVEX.LL ≠ 00B

Meaning	True	Inverse
Greater than	JA, CMOVA, SETA	JNA, CMOVNA, SETNA
Greater than or equal	JAE, CMOVAE, SETAE	JNAE, CMOVNAE, SETNAE
Equal	JE, CMOVE, SETE	JNE, CMOVNE, SETNE
Less than or equal	JLE, CMOVLE, SETLE	JNLE, CMOVNLE, SETNLE
Less than	JL, CMOVL, SETL	JNL, CMOVNL, SETNL
Ordered	JNP, CMOVNP, SETNP	JP, CMOVP, SETP

Table 8.2: Valid Comparison Tests

8.2.3 OPERATION

```
1
2  VCOMXSH (all versions)
3  RESULT := OrderedCompare(DEST[16:0] != SRC[16:0]) {
4  (* Set EFLAGS *) CASE (RESULT) OF
5          UNORDERED: OF,SF,ZF,PF,CF := 11011;
6          GREATER_THAN: OF,SF,ZF,PF,CF := 00000;
7          LESS_THAN: OF,SF,ZF,PF,CF := 10001;
8          EQUAL: OF,SF,ZF,PF,CF := 11100;
9  ESAC;
10 AF := 0; }
11
```

8.2.4 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Denormal, Invalid In abbreviated form, this includes: DE, IE

8.2.5 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VCOMXSH xmm1, xmm2/m16	E3NF	ID	AVX10.2

8.3 VCOMXSS

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.LLIG.F3.0F.W0 2F /r VCOMXSS xmm1, xmm2/m32 {sae}	A	V/V	AVX10.2

8.3.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	SCALAR	MODRM.REG(r)	MODRM.R/M(r)	N/A	N/A

8.3.2 DESCRIPTION

Note:

For SCALAR instructions with a CPUID feature flag specifying Intel® AVX10, the programmer can skip the check for the available vector options on the processor at run-time via the VL256 and VL512 bits in Converged Vector ISA Leaf 0x24. These bits enumerate whether the corresponding vector lengths are supported and usually determine the set of instructions available to the programmer. However, scalar instructions are architected without the notion of vector length, so this step does not strictly apply.

VCOMXSS: Compares the single-precision floating-point values in the low doublewords of operand 1 (first operand) and operand 2 (second operand), and sets the OF, SF, ZF, PF, and CF flags in the EFLAGS register according to the result (unordered, greater than, less than, or equal). The AF flag in the EFLAGS register is set to 0. The unordered result is returned if either source operand is a NaN (QNaN or SNaN). Operand 1 is an XMM register; operand 2 can be an XMM register or a 32-bit memory location. The VCOMXSS instruction differs from the VCOMISS instruction in that the EFLAGS register is set differently to allow more relationships to be tested directly, with unsigned integer tests used for testing greater than conditions and signed integer tests used for testing less than. The VCOMXSS instruction differs from the VUCOMXSS instruction in that it signals a SIMD floating-point invalid operation exception (#XM, specificaly #I) when a source operand is either a QNaN or SNaN. The VUCOMXSS instruction signals an invalid operation exception only if a source operand is an SNaN. The EFLAGS register is not updated if an unmasked SIMD floating-point exception is generated. EVEX.vvvv are reserved and must be 1111b, otherwise instructions will #UD. Software should ensure VCOMXSS is encoded with EVEX.LL=00b, otherwise instructions will #UD.

Additionally: #UD If EVEX.vvvv ≠ 1111B, EVEX.vvvv ≠ 1111B, EVEX.L ≠ 0, or EVEX.LL ≠ 00B

Meaning	True	Inverse
Greater than	JA, CMOVA, SETA	JNA, CMOVNA, SETNA
Greater than or equal	JAE, CMOVAE, SETAE	JNAE, CMOVNAE, SETNAE
Equal	JE, CMOVE, SETE	JNE, CMOVNE, SETNE
Less than or equal	JLE, CMOVLE, SETLE	JNLE, CMOVNLE, SETNLE
Less than	JL, CMOVL, SETL	JNL, CMOVNL, SETNL
Ordered	JNP, CMOVNP, SETNP	JP, CMOVP, SETP

Table 8.3: Valid Comparison Tests

8.3.3 OPERATION

```
1
2  VCOMXSS (all versions)
3  RESULT := OrderedCompare(DEST[31:0] != SRC[31:0]) {
4  (* Set EFLAGS *) CASE (RESULT) OF
5          UNORDERED: OF,SF,ZF,PF,CF := 11011;
6          GREATER_THAN: OF,SF,ZF,PF,CF := 00000;
7          LESS_THAN: OF,SF,ZF,PF,CF := 10001;
8          EQUAL: OF,SF,ZF,PF,CF := 11100;
9  ESAC;
10 AF := 0; }
11
```

8.3.4 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Denormal, Invalid In abbreviated form, this includes: DE, IE

8.3.5 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VCOMXSS xmm1, xmm2/m32	E3NF	ID	AVX10.2

8.4 VUCOMXSD

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.LLIG.F2.0F.W1 2E /r VUCOMXSD xmm1, xmm2/m64 {sae}	A	V/V	AVX10.2

8.4.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	SCALAR	MODRM.REG(r)	MODRM.R/M(r)	N/A	N/A

8.4.2 DESCRIPTION

Note:

For SCALAR instructions with a CPUID feature flag specifying Intel® AVX10, the programmer can skip the check for the available vector options on the processor at run-time via the VL256 and VL512 bits in Converged Vector ISA Leaf 0x24. These bits enumerate whether the corresponding vector lengths are supported and usually determine the set of instructions available to the programmer. However, scalar instructions are architected without the notion of vector length, so this step does not strictly apply.

VUCOMXSD: Performs an unordered compare of the double-precision floating-point values in the low quadwords of operand 1 (first operand) and operand 2 (second operand), and sets the OF, SF, ZF, PF, and CF flags in the EFLAGS register according to the result (unordered, greater than, less than, or equal). The AF flag in the EFLAGS register is set to 0. The unordered result is returned if either source operand is a NaN (QNaN or SNaN). Operand 1 is an XMM register; operand 2 can be an XMM register or a 64-bit memory location. The VUCOMXSD instruction differs from the VUCOMISD instruction in that the EFLAGS register is set differently to allow more relationships to be tested directly, with unsigned integer tests used for testing greater than conditions and signed integer tests used for testing less than. The VUCOMXSD instruction differs from the VCOMXSD instruction in that it signals a SIMD floating-point invalid operation exception (#XM, specificaly #I) only when a source operand is an SNaN. The EFLAGS register is not updated if an unmasked SIMD floating-point exception is generated. Note: EVEX.vvvv are reserved and must be 1111b, otherwise instructions will #UD. Software should ensure VUCOMXSD is encoded with EVEX.LL=00b, otherwise instructions will #UD.

Additionally: #UD If EVEX.vvvv ≠ 1111B, EVEX.vvvv ≠ 1111B, EVEX.L ≠ 0, or EVEX.LL ≠ 00B

Meaning	True	Inverse
Greater than	JA, CMOVA, SETA	JNA, CMOVNA, SETNA
Greater than or equal	JAE, CMOVAE, SETAE	JNAE, CMOVNAE, SETNAE
Equal	JE, CMOVE, SETE	JNE, CMOVNE, SETNE
Less than or equal	JLE, CMOVLE, SETLE	JNLE, CMOVNLE, SETNLE
Less than	JL, CMOVL, SETL	JNL, CMOVNL, SETNL
Ordered	JNP, CMOVNP, SETNP	JP, CMOVP, SETP

Table 8.4: Valid Comparison Tests

8.4.3 OPERATION

```
1
2  VUCOMXSD (all versions)
3  RESULT := OrderedCompare(DEST[63:0] != SRC[63:0]) {
4  (* Set EFLAGS *) CASE (RESULT) OF
5          UNORDERED: OF,SF,ZF,PF,CF := 11011;
6          GREATER_THAN: OF,SF,ZF,PF,CF := 00000;
7          LESS_THAN: OF,SF,ZF,PF,CF := 10001;
8          EQUAL: OF,SF,ZF,PF,CF := 11100;
9  ESAC;
10 AF := 0; }
11
```

8.4.4 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Denormal, Invalid In abbreviated form, this includes: DE, IE

8.4.5 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VUCOMXSD xmm1, xmm2/m64	E3NF	ID	AVX10.2

8.5 VUCOMXSH

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.LLIG.F3.MAP5.W0 2E /r VUCOMXSH xmm1, xmm2/m16 {sae}	A	V/V	AVX10.2

8.5.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	SCALAR	MODRM.REG(r)	MODRM.R/M(r)	N/A	N/A

8.5.2 DESCRIPTION

Note:

For SCALAR instructions with a CPUID feature flag specifying Intel® AVX10, the programmer can skip the check for the available vector options on the processor at run-time via the VL256 and VL512 bits in Converged Vector ISA Leaf 0x24. These bits enumerate whether the corresponding vector lengths are supported and usually determine the set of instructions available to the programmer. However, scalar instructions are architected without the notion of vector length, so this step does not strictly apply.

VUCOMXSH: Performs an unordered compare of the 16 bit floating point values in the low words of operand 1 (first operand) and operand 2 (second operand), and sets the OF, SF, ZF, PF, and CF flags in the EFLAGS register according to the result (unordered, greater than, less than, or equal). The AF flag in the EFLAGS register is set to 0. The unordered result is returned if either source operand is a NaN (QNaN or SNaN). Operand 1 is an XMM register; operand 2 can be an XMM register or a 16-bit memory location. The VUCOMXSH instruction differs from the VCOMISH instruction in that the EFLAGS register is set differently to allow more relationships to be tested directly, with unsigned integer tests used for testing greater than conditions and signed integer tests used for testing less than. The VUCOMXSH instruction differs from the VCOMXSH instruction in that it signals a SIMD floating-point invalid operation exception (#XM, specificaly #I) only if a source operand is an SNaN. The VCOMXSH instruction signals an invalid operation exception when a source operand is either a QNaN or SNaN. The EFLAGS register is not updated if an unmasked SIMD floating-point exception is generated. Note: EVEX.vvvv are reserved and must be 1111b, otherwise instructions will #UD. Software should ensure VUCOMXSH is encoded with EVEX.LL=00b, otherwise instructions will #UD.

Additionally: #UD If EVEX.vvvv ≠ 1111B, EVEX.vvvv ≠ 1111B, EVEX.L ≠ 0, or EVEX.LL ≠ 00B

Meaning	True	Inverse
Greater than	JA, CMOVA, SETA	JNA, CMOVNA, SETNA
Greater than or equal	JAE, CMOVAE, SETAE	JNAE, CMOVNAE, SETNAE
Equal	JE, CMOVE, SETE	JNE, CMOVNE, SETNE
Less than or equal	JLE, CMOVLE, SETLE	JNLE, CMOVNLE, SETNLE
Less than	JL, CMOVL, SETL	JNL, CMOVNL, SETNL
Ordered	JNP, CMOVNP, SETNP	JP, CMOVP, SETP

Table 8.5: Valid Comparison Tests

8.5.3 OPERATION

```

VUCOMXSH (all versions)
RESULT := OrderedCompare(DEST[16:0] != SRC[16:0]) {
(* Set EFLAGS *) CASE (RESULT) OF
        UNORDERED: OF,SF,ZF,PF,CF := 11011;
        GREATER_THAN: OF,SF,ZF,PF,CF := 00000;
        LESS_THAN: OF,SF,ZF,PF,CF := 10001;
        EQUAL: OF,SF,ZF,PF,CF := 11100;
ESAC;
AF := 0; }

```

8.5.4 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Denormal, Invalid In abbreviated form, this includes: DE, IE

8.5.5 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VUCOMXSH xmm1, xmm2/m16	E3NF	ID	AVX10.2

8.6 VUCOMXSS

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.LLIG.F3.0F.W0 2E /r VUCOMXSS xmm1, xmm2/m32 {sae}	A	V/V	AVX10.2

8.6.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	SCALAR	MODRM.REG(r)	MODRM.R/M(r)	N/A	N/A

8.6.2 DESCRIPTION

Note:

For SCALAR instructions with a CPUID feature flag specifying Intel® AVX10, the programmer can skip the check for the available vector options on the processor at run-time via the VL256 and VL512 bits in Converged Vector ISA Leaf 0x24. These bits enumerate whether the corresponding vector lengths are supported and usually determine the set of instructions available to the programmer. However, scalar instructions are architected without the notion of vector length, so this step does not strictly apply.

VUCOMXSS: Compares the single-precision floating-point values in the low doublewords of operand 1 (first operand) and operand 2 (second operand), and sets the OF, SF, ZF, PF, and CF flags in the EFLAGS register according to the result (unordered, greater than, less than, or equal). The AF flag in the EFLAGS register is set to 0. The unordered result is returned if either source operand is a NaN (QNaN or SNaN). Operand 1 is an XMM register; operand 2 can be an XMM register or a 32-bit memory location. The VUCOMXSS instruction differs from the VUCOMISS instruction in that the EFLAGS register is set differently to allow more relationships to be tested directly, with unsigned integer tests used for testing greater than conditions and signed integer tests used for testing less than. The VUCOMXSS instruction differs from the VCOMXSS instruction in that it signals a SIMD floating-point invalid operation exception (#XM, specificaly #I) only if a source operand is an SNaN. The VCOMXSS instruction signals an invalid operation exception when a source operand is either a QNaN or SNaN. The EFLAGS register is not updated if an unmasked SIMD floating-point exception is generated. Note: EVEX.vvvv are reserved and must be 1111b, otherwise instructions will #UD. Software should ensure VUCOMXSS is encoded with EVEX.LL=00b, otherwise instructions will #UD.

Additionally: #UD If EVEX.vvvv ≠ 1111B, EVEX.vvvv ≠ 1111B, EVEX.L ≠ 0, or EVEX.LL ≠ 00B

Meaning	True	Inverse
Greater than	JA, CMOVA, SETA	JNA, CMOVNA, SETNA
Greater than or equal	JAE, CMOVAE, SETAE	JNAE, CMOVNAE, SETNAE
Equal	JE, CMOVE, SETE	JNE, CMOVNE, SETNE
Less than or equal	JLE, CMOVLE, SETLE	JNLE, CMOVNLE, SETNLE
Less than	JL, CMOVL, SETL	JNL, CMOVNL, SETNL
Ordered	JNP, CMOVNP, SETNP	JP, CMOVP, SETP

Table 8.6: Valid Comparison Tests

8.6.3 OPERATION

```
VUCOMXSS (all versions)
RESULT := OrderedCompare(DEST[31:0] != SRC[31:0]) {
(* Set EFLAGS *) CASE (RESULT) OF
        UNORDERED: OF,SF,ZF,PF,CF := 11011;
        GREATER_THAN: OF,SF,ZF,PF,CF := 00000;
        LESS_THAN: OF,SF,ZF,PF,CF := 10001;
        EQUAL: OF,SF,ZF,PF,CF := 11100;
ESAC;
AF := 0; }
```

8.6.4 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Denormal, Invalid In abbreviated form, this includes: DE, IE

8.6.5 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VUCOMXSS xmm1, xmm2/m32	E3NF	ID	AVX10.2

Chapter 9

INTEL® AVX10 CONVERT INSTRUCTIONS

9.1 VCVT[,2]PH2[B,H]F8[,S]

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.128.F2.0F38.W0 74 /r VCVT2PH2BF8 xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	A	V/V	AVX10.2
EVEX.256.F2.0F38.W0 74 /r VCVT2PH2BF8 ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst	A	V/V	AVX10.2
EVEX.512.F2.0F38.W0 74 /r VCVT2PH2BF8 zmm1{k1}{z}, zmm2, zmm3/m512/m16bcst	A	V/V	AVX10.2
EVEX.128.F2.MAP5.W0 74 /r VCVT2PH2BF8S xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	A	V/V	AVX10.2
EVEX.256.F2.MAP5.W0 74 /r VCVT2PH2BF8S ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst	A	V/V	AVX10.2
EVEX.512.F2.MAP5.W0 74 /r VCVT2PH2BF8S zmm1{k1}{z}, zmm2, zmm3/m512/m16bcst	A	V/V	AVX10.2
EVEX.128.F2.MAP5.W0 18 /r VCVT2PH2HF8 xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	A	V/V	AVX10.2
EVEX.256.F2.MAP5.W0 18 /r VCVT2PH2HF8 ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst	A	V/V	AVX10.2
EVEX.512.F2.MAP5.W0 18 /r VCVT2PH2HF8 zmm1{k1}{z}, zmm2, zmm3/m512/m16bcst	A	V/V	AVX10.2
EVEX.128.F2.MAP5.W0 1B /r VCVT2PH2HF8S xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	A	V/V	AVX10.2
EVEX.256.F2.MAP5.W0 1B /r VCVT2PH2HF8S ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst	A	V/V	AVX10.2
EVEX.512.F2.MAP5.W0 1B /r VCVT2PH2HF8S zmm1{k1}{z}, zmm2, zmm3/m512/m16bcst	A	V/V	AVX10.2
EVEX.128.F3.0F38.W0 74 /r VCVTPH2BF8 xmm1{k1}{z}, xmm2/m128/m16bcst	B	V/V	AVX10.2
EVEX.256.F3.0F38.W0 74 /r VCVTPH2BF8 xmm1{k1}{z}, ymm2/m256/m16bcst	B	V/V	AVX10.2
EVEX.512.F3.0F38.W0 74 /r VCVTPH2BF8 ymm1{k1}{z}, zmm2/m512/m16bcst	B	V/V	AVX10.2
EVEX.128.F3.MAP5.W0 74 /r VCVTPH2BF8S xmm1{k1}{z}, xmm2/m128/m16bcst	B	V/V	AVX10.2

Continued on next page...

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.F3.MAP5.W0 74 /r VCVTPH2BF8S xmm1{k1}{z}, ymm2/m256/m16bcst	B	V/V	AVX10.2
EVEX.512.F3.MAP5.W0 74 /r VCVTPH2BF8S ymm1{k1}{z}, zmm2/m512/m16bcst	B	V/V	AVX10.2
EVEX.128.F3.MAP5.W0 18 /r VCVTPH2HF8 xmm1{k1}{z}, xmm2/m128/m16bcst	B	V/V	AVX10.2
EVEX.256.F3.MAP5.W0 18 /r VCVTPH2HF8 xmm1{k1}{z}, ymm2/m256/m16bcst	B	V/V	AVX10.2
EVEX.512.F3.MAP5.W0 18 /r VCVTPH2HF8 ymm1{k1}{z}, zmm2/m512/m16bcst	B	V/V	AVX10.2
EVEX.128.F3.MAP5.W0 1B /r VCVTPH2HF8S xmm1{k1}{z}, xmm2/m128/m16bcst	B	V/V	AVX10.2
EVEX.256.F3.MAP5.W0 1B /r VCVTPH2HF8S xmm1{k1}{z}, ymm2/m256/m16bcst	B	V/V	AVX10.2
EVEX.512.F3.MAP5.W0 1B /r VCVTPH2HF8S ymm1{k1}{z}, zmm2/m512/m16bcst	B	V/V	AVX10.2

9.1.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(w)	VVVV(r)	MODRM.R/M(r)	N/A
B	FULL	MODRM.REG(w)	MODRM.R/M(r)	N/A	N/A

9.1.2 DESCRIPTION

Note:

For instructions with a CPUID feature flag specifying Intel® AVX10, the programmer must check the available vector options on the processor at run-time via the VL256 and VL512 bits in Converged Vector ISA Leaf 0x24. These bits enumerate whether the corresponding vector lengths are supported (128-bit vectors are always supported) and as such will determine the set of instructions available to the programmer listed in the above opcode table.

These instructions convert one or two SIMD registers of packed FP16 data into a single register of packed E5M2 FP8 values or E4M3 FP8 values. The upper bits of the destination register beyond the

downconverted E5M2/E4M3 elements are zeroed. Rounding Mode is always RNE (Round To Nearest Ties to Even). DAZ is not obeyed and always assumed DAZ==0. FTZ is not obeyed and always assumed FTZ==0. These instructions do not raise exceptions but do set status. For saturated instructions, infinity at input is saturated to maximal normal values. For non-saturated versions, infinity at input is preserved (PH2BF8) or converted to NaN (PH2HF8). In case of overflow due to conversion or rounding, the Saturated instructions (with 'S' suffix) returns the corresponding E5M2_MAX or E4M3_MAX, the non-saturated versions return infinity (PH2BF8) or NaN (PH2HF8). All MXCSR mask bits DM, IM, OM, PM, UM are implicitly set for these instructions.

VCVTPH2BF8 converts eight, sixteen or thirty-two packed FP16 values in the source operand to eight, sixteen or thirty-two packed E5M2 FP8 values in the destination operand. The UE (underflow) flag is set when the result is both denormal and inexact.

VCVTPH2HF8 converts eight, sixteen or thirty-two packed FP16 values in the source operand to eight, sixteen or thirty-two packed E4M3 FP8 values in the destination operand.

VCVTPH2BF8S converts eight, sixteen or thirty-two packed FP16 values in the source operand to eight, sixteen or thirty-two packed E5M2 FP8 values in the destination operand. Returns Saturated values in case of overflow due to conversion or rounding. The UE (underflow) flag is set when the result is both denormal and inexact.

VCVTPH2HF8S converts eight, sixteen or thirty-two packed FP16 values in the source operand to eight, sixteen or thirty-two packed E4M3 FP8 values in the destination operand. Returns Saturated values in case of overflow due to conversion or rounding.

VCVT2PH2BF8 converts sixteen, thirty-two or sixty-four packed FP16 values in the 2 source operands to sixteen, thirty-two or sixty-four packed E5M2 FP8 values in the destination operand. The UE (underflow) flag is set when the result is both denormal and inexact.

VCVT2PH2HF8 converts sixteen, thirty-two or sixty-four packed FP16 values in the 2 source operands to sixteen, thirty-two or sixty-four packed E4M3 FP8 values in the destination operand.

VCVT2PH2BF8S converts sixteen, thirty-two or sixty-four packed FP16 values in the 2 source operands to sixteen, thirty-two or sixty-four packed E5M2 FP8 values in the destination operand. Returns Saturated values in case of overflow due to conversion or rounding. The UE (underflow) flag is set when the result is both denormal and inexact.

VCVT2PH2HF8S converts sixteen, thirty-two or sixty-four packed FP16 values in the 2 source operands to sixteen, thirty-two or sixty-four packed E4M3 FP8 values in the destination operand. Returns Saturated values in case of overflow due to conversion or rounding.

9.1.3 OPERATION

```
VCVTPH2[B,H]F8[,S] dest {k1}, src
VL = (128,256,512)
KL = VL/8
origdest := dest
if *dest is bfloat8*:
  y := bf8
else:
  y := hf8
if *saturation*:
  s := 1
else:
  s := 0

for i := 0 to KL/2-1:
  if k1[i] or *no writemask*:
    if src is memory and evex.b == 1:
      t := src.fp16[0]
    else:
      t := src.fp16[i]

    dest.fp8[i] := convert_fp16_to_fp8(t, y, s)

  else if *zeroing*:
    dest.fp8[i] := 0
  else: // merge masking, dest element unchanged
    dest.fp8[i] := origdest.fp8[i]
dest[max_vl-1:vl/2] := 0
```

```
VCVT2PH2[B,H]F8[,S] dest, src1, src2, k1
VL = (128,256,512)
KL = VL/8
if *dest is bfloat8*:
  y := bf8
else:
  y := hf8
if *saturation*:
  s := 1
else:
  s := 0

for i := 0 to KL-1:
  if (k1[i] or *no writemask*):
      if i < KL/2:
        if src2 is memory and evex.b == 1:
          t := src2.fp16[0]
        else:
          t := src2.fp16[i]
      else:
        t := src1.fp16[i-KL/2]

      dest.fp8[i] := convert_fp16_to_fp8(t, y, s)
   else:
      if *merging-masking*:
         dest.fp8[i] remains unchanged
      else:
         dest.fp8[i] := 0

dest[max_vl-1:vl] := 0
```

9.1.4 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Denormal, Invalid, Overflow, Precision, Underflow In abbreviated form, this includes: DE, IE, OE, PE, UE

9.1.5 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VCVT2PH2BF8 xmm1, xmm2, xmm3/m128	E4NF	IOUPD	AVX10.2

Continued on next page...

Instruction	Exception Type	Arithmetic Flags	CPUID
VCVT2PH2BF8 ymm1, ymm2, ymm3/m256	E4NF	IOUPD	AVX10.2
VCVT2PH2BF8 zmm1, zmm2, zmm3/m512	E4NF	IOUPD	AVX10.2
VCVT2PH2BF8S xmm1, xmm2, xmm3/m128	E4NF	IOUPD	AVX10.2
VCVT2PH2BF8S ymm1, ymm2, ymm3/m256	E4NF	IOUPD	AVX10.2
VCVT2PH2BF8S zmm1, zmm2, zmm3/m512	E4NF	IOUPD	AVX10.2
VCVT2PH2HF8 xmm1, xmm2, xmm3/m128	E4NF	IOUPD	AVX10.2
VCVT2PH2HF8 ymm1, ymm2, ymm3/m256	E4NF	IOUPD	AVX10.2
VCVT2PH2HF8 zmm1, zmm2, zmm3/m512	E4NF	IOUPD	AVX10.2
VCVT2PH2HF8S xmm1, xmm2, xmm3/m128	E4NF	IOUPD	AVX10.2
VCVT2PH2HF8S ymm1, ymm2, ymm3/m256	E4NF	IOUPD	AVX10.2
VCVT2PH2HF8S zmm1, zmm2, zmm3/m512	E4NF	IOUPD	AVX10.2
VCVTPH2BF8 xmm1, xmm2/m128	E4NF	IOUPD	AVX10.2
VCVTPH2BF8 xmm1, ymm2/m256	E4NF	IOUPD	AVX10.2
VCVTPH2BF8 ymm1, zmm2/m512	E4NF	IOUPD	AVX10.2
VCVTPH2BF8S xmm1, xmm2/m128	E4NF	IOUPD	AVX10.2
VCVTPH2BF8S xmm1, ymm2/m256	E4NF	IOUPD	AVX10.2
VCVTPH2BF8S ymm1, zmm2/m512	E4NF	IOUPD	AVX10.2
VCVTPH2HF8 xmm1, xmm2/m128	E4NF	IOUPD	AVX10.2
VCVTPH2HF8 xmm1, ymm2/m256	E4NF	IOUPD	AVX10.2
VCVTPH2HF8 ymm1, zmm2/m512	E4NF	IOUPD	AVX10.2
VCVTPH2HF8S xmm1, xmm2/m128	E4NF	IOUPD	AVX10.2

Continued on next page...

Instruction	Exception Type	Arithmetic Flags	CPUID
VCVTPH2HF8S xmm1, ymm2/m256	E4NF	IOUPD	AVX10.2
VCVTPH2HF8S ymm1, zmm2/m512	E4NF	IOUPD	AVX10.2

9.2 VCVT2PS2PHX

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.128.66.0F38.W0 67 /r VCVT2PS2PHX xmm1{k1}{z}, xmm2, xmm3/m128/m32bcst	A	V/V	AVX10.2
EVEX.256.66.0F38.W0 67 /r VCVT2PS2PHX ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst {er}	A	V/V	AVX10.2
EVEX.512.66.0F38.W0 67 /r VCVT2PS2PHX zmm1{k1}{z}, zmm2, zmm3/m512/m32bcst {er}	A	V/V	AVX10.2

9.2.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(w)	VVVV(r)	MODRM.R/M(r)	N/A

9.2.2 DESCRIPTION

Note:

For instructions with a CPUID feature flag specifying Intel® AVX10, the programmer must check the available vector options on the processor at run-time via the VL256 and VL512 bits in Converged Vector ISA Leaf 0x24. These bits enumerate whether the corresponding vector lengths are supported (128-bit vectors are always supported) and as such will determine the set of instructions available to the programmer listed in the above opcode table.

This instruction converts two SIMD registers of Packed Single data into a single register of packed FP16 data. This instruction does not support memory fault suppression.

This instruction updates MXCSR as if all MXCSR numerical exceptions flags are masked and does not generate floating point exceptions. MXCSR (and EVEX embedded rounding) determine the rounding mode. Input FP32 denormals are affected by MXCSR.DAZ but output FP16 denormals are not affected by MXCSR.FTZ and not flushed to zero.

9.2.3 OPERATION

```
VCVT2PS2PH dest, src1, src2, k1     // EVEX encoded version
VL = (128,256,512)
KL := VL/16

for i := 0 to KL-1:
    if (k1[i] or *no writemask*):
        if i < KL/2:
              if src2 is memory and evex.b == 1:
                  t := src2.fp32[0]
              else:
                  t := src2.fp32[i]
        else:
              t := src1.fp32[i-KL/2]
        dest.word[i] := convert_fp32_to_fp16(t)

    else:
        if *merging-masking*:
            dest.word[i] remains unchanged
        else: // zero masking
            dest.word[i] := 0

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

9.2.4 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Denormal, Invalid, Overflow, Precision, Underflow In abbreviated form, this includes: DE, IE, OE, PE, UE

9.2.5 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VCVT2PS2PHX xmm1, xmm2, xmm3/m128	E4NF	IOUPD	AVX10.2
VCVT2PS2PHX ymm1, ymm2, ymm3/m256	E4NF	IOUPD	AVX10.2
VCVT2PS2PHX zmm1, zmm2, zmm3/m512	E4NF	IOUPD	AVX10.2

9.3 VCVTBIASPH2[B,H]F8[,S]

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.128.NP.0F38.W0 74 /r VCVTBIASPH2BF8 xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	A	V/V	AVX10.2
EVEX.256.NP.0F38.W0 74 /r VCVTBIASPH2BF8 xmm1{k1}{z}, ymm2, ymm3/m256/m16bcst	A	V/V	AVX10.2
EVEX.512.NP.0F38.W0 74 /r VCVTBIASPH2BF8 ymm1{k1}{z}, zmm2, zmm3/m512/m16bcst	A	V/V	AVX10.2
EVEX.128.NP.MAP5.W0 74 /r VCVTBIASPH2BF8S xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	A	V/V	AVX10.2
EVEX.256.NP.MAP5.W0 74 /r VCVTBIASPH2BF8S xmm1{k1}{z}, ymm2, ymm3/m256/m16bcst	A	V/V	AVX10.2
EVEX.512.NP.MAP5.W0 74 /r VCVTBIASPH2BF8S ymm1{k1}{z}, zmm2, zmm3/m512/m16bcst	A	V/V	AVX10.2
EVEX.128.NP.MAP5.W0 18 /r VCVTBIASPH2HF8 xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	A	V/V	AVX10.2
EVEX.256.NP.MAP5.W0 18 /r VCVTBIASPH2HF8 xmm1{k1}{z}, ymm2, ymm3/m256/m16bcst	A	V/V	AVX10.2
EVEX.512.NP.MAP5.W0 18 /r VCVTBIASPH2HF8 ymm1{k1}{z}, zmm2, zmm3/m512/m16bcst	A	V/V	AVX10.2
EVEX.128.NP.MAP5.W0 1B /r VCVTBIASPH2HF8S xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	A	V/V	AVX10.2
EVEX.256.NP.MAP5.W0 1B /r VCVTBIASPH2HF8S xmm1{k1}{z}, ymm2, ymm3/m256/m16bcst	A	V/V	AVX10.2
EVEX.512.NP.MAP5.W0 1B /r VCVTBIASPH2HF8S ymm1{k1}{z}, zmm2, zmm3/m512/m16bcst	A	V/V	AVX10.2

9.3.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(w)	VVVV(r)	MODRM.R/M(r)	N/A

9.3.2 DESCRIPTION

Note:

For instructions with a CPUID feature flag specifying Intel® AVX10, the programmer must check the available vector options on the processor at run-time via the VL256 and VL512 bits in Converged Vector ISA Leaf 0x24. These bits enumerate whether the corresponding vector lengths are supported (128-bit vectors are always supported) and as such will determine the set of instructions available to the programmer listed in the above opcode table.

This group of instructions add packed 8-bit unsigned INT numbers to the 8 bits below the LSB of the rounded FP16 numbers and then converts the result into a packed set of E5M2 FP8 values or E4M3 FP8 values by normalizing and truncation to align with E5M2/E4M3 dynamic range. Infinity is preserved for the non-saturated versions. NaN at input is propagated as qNaN. If the result is infinity or too big to be represented then for the saturated version, E5M2_MAX or E4M3_MAX is returned, for the non-saturated versions, infinity is returned (PH2BF8) or NaN (PH2HF8). In the 2 source versions the upper bits of the destination register beyond the downconverted E5M2/E4M3 elements are zeroed. DAZ is not obeyed and always assumed DAZ==0. FTZ is not obeyed and always assumed FTZ==0. Execution occurs as if all MXCSR exceptions are masked. All MXCSR mask bits DM, IM, OM, PM, UM are implicitly set for these instructions.

VCVTBIASPH2BF8 integer adds eight, sixteen or thirty-two packed INT8 in the lower half of the first source operand with eight, sixteen or thirty-two packed FP16 values in the second source operand. Addition is aligned to the FP16 LSB. The result is converted to eight, sixteen or thirty-two packed E5M2 FP8 values in the dest operand. The result is truncated. The UE (underflow) flag is set when the result is both denormal and inexact.

VCVTBIASPH2HF8 integer adds eight, sixteen or thirty-two packed INT8 in the lower half of the first source operand with eight, sixteen or thirty-two packed FP16 values in the second source operand. First the first source operand is shited right by '1 and then the addition is aligned to the FP16 LSB. The result is converted to eight, sixteen or thirty-two packed E4M3 FP8 values in the dest operand. The result is truncated.

VCVTBIASPH2BF8S integer adds eight, sixteen or thirty-two packed INT8 in the lower half of the first source operand with eight, sixteen or thirty-two packed FP16 values in the second source operand. Addition is aligned to the FP16 LSB. The result is converted to eight, sixteen or thirty-two packed E5M2 FP8 values in the dest operand. The result is truncated. If the result is too big to be represented E5M2_MAX is returned. The UE (underflow) flag is set when the result is both denormal and inexact.

VCVTBIASPH2HF8S integer adds eight, sixteen or thirty-two packed INT8 in the lower half of the first source operand with eight, sixteen or thirty-two packed FP16 values in the second source operand. First the first source operand is shited right by '1 and then the addition is aligned to the FP16 LSB. The result is converted to eight, sixteen or thirty-two packed E4M3 FP8 values in the dest operand. The result is truncated. If the result is too big to be represented E4M3_MAX is returned.

9.3.3 OPERATION

```
VCVTBIASPH2[B,H]F8[,S] dest {k1}, src1, src2
VL = (128,256,512)
KL = VL/8
origdest := dest

if *dest is bfloat8*:
  type := bf8
else:
  type := hf8

if *saturation*:
  s := 1
else:
  s := 0

for i := 0 to KL/2-1:
  if k1[i] or *no writemask*:
    if src is memory and evex.b == 1:
      t := src2.fp16[0]
    else:
      t := src2.fp16[i]

    dest.fp8[i] := convert_fp16_to_fp8_bias(t, src1.byte[2i], type, s)
  else if *zeroing*:
      dest.fp8[i] := 0
  else: // merge masking, dest element unchanged
     dest.fp8[i] := origdest.fp8[i]
dest[max_vl-1:vl/2] := 0
```

9.3.4 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Denormal, Invalid, Overflow, Precision, Underflow In abbreviated form, this includes: DE, IE, OE, PE, UE

9.3.5 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VCVTBIASPH2BF8 xmm1, xmm2, xmm3/m128	E4NF	IOUPD	AVX10.2

Continued on next page...

Instruction	Exception Type	Arithmetic Flags	CPUID
VCVTBIASPH2BF8 xmm1, ymm2, ymm3/m256	E4NF	IOUPD	AVX10.2
VCVTBIASPH2BF8 ymm1, zmm2, zmm3/m512	E4NF	IOUPD	AVX10.2
VCVTBIASPH2BF8S xmm1, xmm2, xmm3/m128	E4NF	IOUPD	AVX10.2
VCVTBIASPH2BF8S xmm1, ymm2, ymm3/m256	E4NF	IOUPD	AVX10.2
VCVTBIASPH2BF8S ymm1, zmm2, zmm3/m512	E4NF	IOUPD	AVX10.2
VCVTBIASPH2HF8 xmm1, xmm2, xmm3/m128	E4NF	IOUPD	AVX10.2
VCVTBIASPH2HF8 xmm1, ymm2, ymm3/m256	E4NF	IOUPD	AVX10.2
VCVTBIASPH2HF8 ymm1, zmm2, zmm3/m512	E4NF	IOUPD	AVX10.2
VCVTBIASPH2HF8S xmm1, xmm2, xmm3/m128	E4NF	IOUPD	AVX10.2
VCVTBIASPH2HF8S xmm1, ymm2, ymm3/m256	E4NF	IOUPD	AVX10.2
VCVTBIASPH2HF8S ymm1, zmm2, zmm3/m512	E4NF	IOUPD	AVX10.2

9.4 VCVTHF82PH

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.128.F2.MAP5.W0 1E /r VCVTHF82PH xmm1{k1}{z}, xmm2/m64	A	V/V	AVX10.2
EVEX.256.F2.MAP5.W0 1E /r VCVTHF82PH ymm1{k1}{z}, xmm2/m128	A	V/V	AVX10.2
EVEX.512.F2.MAP5.W0 1E /r VCVTHF82PH zmm1{k1}{z}, ymm2/m256	A	V/V	AVX10.2

9.4.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	HALF	MODRM.REG(w)	MODRM.R/M(r)	N/A	N/A

9.4.2 DESCRIPTION

Note:

For instructions with a CPUID feature flag specifying Intel® AVX10, the programmer must check the available vector options on the processor at run-time via the VL256 and VL512 bits in Converged Vector ISA Leaf 0x24. These bits enumerate whether the corresponding vector lengths are supported (128-bit vectors are always supported) and as such will determine the set of instructions available to the programmer listed in the above opcode table.

VCVTHF82PH converts E4M3 FP8 value datatype elements into FP16 elements. DAZ is not obeyed and always assumed DAZ==0. Since any E4M3 FP8 number can be represented in FP16, the conversion result is exact and no rounding is needed. This instruction only updates DE in MXCSR.

9.4.3 OPERATION

```
VCVTHF82PH dest k1, src
VL = (128,256,512)
KL := VL/16
origdest := dest
for i := 0 to kl-1:
   if k1[i] OR *no writemask*:
      tsrc := src.hf8[i]
      dest.fp16[i] := convert_hf8_to_fp16(tsrc)
   else if *zeroing*:
      dest.fp16[i] := 0
   else: // merge masking, dest element unchanged
      dest.fp16[i] := origdest.fp16[i]
dest[max_vl-1:vl] := 0
```

9.4.4 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Denormal In abbreviated form, this includes: DE

9.4.5 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VCVTHF82PH xmm1, xmm2/m64	E2	D	AVX10.2
VCVTHF82PH ymm1, xmm2/m128	E2	D	AVX10.2
VCVTHF82PH zmm1, ymm2/m256	E2	D	AVX10.2

Chapter 10

INTEL® AVX10 INTEGER AND FP16 VNNI, MEDIA NEW INSTRUCTIONS

10.1 VDPPHPS

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.128.NP.0F38.W0 52 /r VDPPHPS xmm1{k1}{z}, xmm2, xmm3/m128/m32bcst	A	V/V	AVX10.2
EVEX.256.NP.0F38.W0 52 /r VDPPHPS ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst	A	V/V	AVX10.2
EVEX.512.NP.0F38.W0 52 /r VDPPHPS zmm1{k1}{z}, zmm2, zmm3/m512/m32bcst	A	V/V	AVX10.2

10.1.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(rw)	VVVV(r)	MODRM.R/M(r)	N/A

10.1.2 DESCRIPTION

Note:

For instructions with a CPUID feature flag specifying Intel® AVX10, the programmer must check the available vector options on the processor at run-time via the VL256 and VL512 bits in Converged Vector ISA Leaf 0x24. These bits enumerate whether the corresponding vector lengths are supported (128-bit vectors are always supported) and as such will determine the set of instructions available to the programmer listed in the above opcode table.

This instruction performs a SIMD dot-product of two FP16 pairs and accumulates into a packed single precision register.

“Round to nearest even” rounding mode is used when doing each accumulation of the FMA. Output denormals are always flushed to zero. DAZ==0 is assumed for FP16 input. DAZ==1 is assumed for FP32 input. MXCSR is not consulted nor updated.

When more than one input is NaN, the NaN that is propagated to the result (after being quietized) is the first NaN that occurs in the expression Src2low*Src3low + Src2high*Src3high + SrcDest.

10.1.3 OPERATION

```
VDPPHPS srcdest, src1, src2   // EVEX ENCODED VERSION
VL = (128,256,512)

kl := vl/32
origdest := srcdest
for i := 0 to kl-1:
    if k1[i] or *no writemask*:
        if src2 is memory and evex.b == 1:
             t := src2.dword[0]
        else:
             t := src2.dword[i]

        s1o = convert_fp16_to_fp32(src1.fp16[2*i+1])
        s2o = convert_fp16_to_fp32(t.fp16[1])
        s1e = convert_fp16_to_fp32(src1.fp16[2*i+0])
        s2e = convert_fp16_to_fp32(t.fp16[0])

        // MXCSR is neither consulted nor updated.
        // Masked response for all floating point exceptional conditions.
        // MXCSR.DAZ=0 is assumed for FP16 input.
        // MXCSR.DAZ=1 is assumed for FP32 input.
        // MXCSR.FTZ=1 is assumed for FP32 outputs.
        // Uses RNE rounding.

        srcdest.fp32[i] = fma32(srcdest.fp32[i], s1o, s2o, daz=1, ftz=1, sae=1, rc=RNE)
        srcdest.fp32[i] = fma32(srcdest.fp32[i], s1e, s2e, daz=1, ftz=1, sae=1, rc=RNE)

    else if *zeroing*:1
        srcdest.dword[i] := 0

    else: // merge masking, dest element unchanged
        srcdest.dword[i] := origdest.dword[i]

srcdest[max_vl-1:vl] := 0
```

10.1.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VDPPHPS xmm1, xmm2, xmm3/m128	E4	N/A	AVX10.2
VDPPHPS ymm1, ymm2, ymm3/m256	E4	N/A	AVX10.2

Continued on next page...

Instruction	Exception Type	Arithmetic Flags	CPUID
VDPPHPS zmm1, zmm2, zmm3/m512	E4	N/A	AVX10.2

10.2 VMPSADBW

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.128.F3.0F3A.W0 42 /r /ib VMPSADBW xmm1{k1}{z}, xmm2, xmm3/m128, imm8	A	V/V	AVX10.2
EVEX.256.F3.0F3A.W0 42 /r /ib VMPSADBW ymm1{k1}{z}, ymm2, ymm3/m256, imm8	A	V/V	AVX10.2
EVEX.512.F3.0F3A.W0 42 /r /ib VMPSADBW zmm1{k1}{z}, zmm2, zmm3/m512, imm8	A	V/V	AVX10.2

10.2.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULLMEM	MODRM.REG(w)	VVVV(r)	MODRM.R/M(r)	IMM8(r)

10.2.2 DESCRIPTION

Note:

For instructions with a CPUID feature flag specifying Intel® AVX10, the programmer must check the available vector options on the processor at run-time via the VL256 and VL512 bits in Converged Vector ISA Leaf 0x24. These bits enumerate whether the corresponding vector lengths are supported (128-bit vectors are always supported) and as such will determine the set of instructions available to the programmer listed in the above opcode table.

This is the EVEX encoded version of the VMPSADBW instruction. There are AVX and AVX2 versions of this instruction which were introduced on Sandybridge and Haswell respectively.

The EVEX VMPSADBW calculates packed word results of sum-absolute-difference (SAD) of unsigned bytes from two blocks of 32-bit dword elements, using two select fields in the immediate byte to select the offsets of the two blocks within the first source operand and the second operand. Packed SAD word results are calculated within each 128-bit lane. Each SAD word result is calculated between a stationary block_2 (whose offset within the second source operand is selected by a two bit select control, multiplied by 32 bits) and a sliding block_1 at consecutive byte-granular position within the first source operand. The offset of the first 32-bit block of block_1 is selectable using a one bit select control, multiplied by 32 bits. For the 512-bit version of this instruction the control bits for the lower two lanes are replicated to the upper two lanes.

10.2.3 OPERATION

```
def zero_to_max_vl(dst, vl):
    for i in range(vl, MAXVL):
        dst.bit[i] = 0
```

```
def emulate_vmpsadbw(inst, dst, msk, src1, src2, control):
    blk2_pos = (control&3) * 4
    blk1_pos = ((control>>2) & 1) * 4

    # range(x,y) is x through y-1 inclusive
    for i in range(0, 11):
        blk1.byte[i] = src1.byte[i+blk1_pos]

    for i in range(0, 4):
        blk2.byte[i] = src2.byte[i+blk2_pos]

    for i in range(0, 8):
        for j in range(0, 4):
            x = blk1.byte[j+i] - blk2.byte[j]  # 16b signed arithmetic
            tmp.word[j] = abs(x)

        if inst.write_masking == 0 or msk[i]:
            s = 0
            for j in range(0, 4):
                s += tmp.word[j]
            dst.word[i] = s
        elif inst.zeroing:
            dst.word[i] = 0
        #else dst.word[i] remains unchanged


def vmpsadbw_128(inst, dst, msk, src1, src2, imm8):
    emulate_vmpsadbw(inst, dst.xmm[0], msk>>0, src1.xmm[0], src2.xmm[0], imm8)
    zero_to_max_vl(dst, inst.VL)

def vmpsadbw_256(inst, dst, msk, src1, src2, imm8):
    emulate_vmpsadbw(inst, dst.xmm[0], msk>>0, src1.xmm[0], src2.xmm[0], imm8)
    emulate_vmpsadbw(inst, dst.xmm[1], msk>>8, src1.xmm[1], src2.xmm[1], imm8>>3)
    zero_to_max_vl(dst, inst.VL)

def vmpsadbw_512(inst, dst, msk, src1, src2, imm8):
    emulate_vmpsadbw(inst, dst.xmm[0], msk>>0, src1.xmm[0], src2.xmm[0], imm8)
    emulate_vmpsadbw(inst, dst.xmm[1], msk>>8, src1.xmm[1], src2.xmm[1], imm8>>3)
    emulate_vmpsadbw(inst, dst.xmm[2], msk>>16, src1.xmm[2], src2.xmm[2], imm8)
    emulate_vmpsadbw(inst, dst.xmm[3], msk>>24, src1.xmm[3], src2.xmm[3], imm8>>3)
    zero_to_max_vl(dst, inst.VL)
```

10.2.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VMPSADBW xmm1, xmm2, xmm3/m128, imm8	E4NF	N/A	AVX10.2
VMPSADBW ymm1, ymm2, ymm3/m256, imm8	E4NF	N/A	AVX10.2
VMPSADBW zmm1, zmm2, zmm3/m512, imm8	E4NF	N/A	AVX10.2

10.3 VPDPB[SU,UU,SS]D[,S]

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.128.F2.0F38.W0 50 /r VPDPBSSD xmm1{k1}{z}, xmm2, xmm3/m128/m32bcst	A	V/V	AVX10.2
EVEX.256.F2.0F38.W0 50 /r VPDPBSSD ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst	A	V/V	AVX10.2
EVEX.512.F2.0F38.W0 50 /r VPDPBSSD zmm1{k1}{z}, zmm2, zmm3/m512/m32bcst	A	V/V	AVX10.2
EVEX.128.F2.0F38.W0 51 /r VPDPBSSDS xmm1{k1}{z}, xmm2, xmm3/m128/m32bcst	A	V/V	AVX10.2
EVEX.256.F2.0F38.W0 51 /r VPDPBSSDS ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst	A	V/V	AVX10.2
EVEX.512.F2.0F38.W0 51 /r VPDPBSSDS zmm1{k1}{z}, zmm2, zmm3/m512/m32bcst	A	V/V	AVX10.2
EVEX.128.F3.0F38.W0 50 /r VPDPBSUD xmm1{k1}{z}, xmm2, xmm3/m128/m32bcst	A	V/V	AVX10.2
EVEX.256.F3.0F38.W0 50 /r VPDPBSUD ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst	A	V/V	AVX10.2
EVEX.512.F3.0F38.W0 50 /r VPDPBSUD zmm1{k1}{z}, zmm2, zmm3/m512/m32bcst	A	V/V	AVX10.2
EVEX.128.F3.0F38.W0 51 /r VPDPBSUDS xmm1{k1}{z}, xmm2, xmm3/m128/m32bcst	A	V/V	AVX10.2
EVEX.256.F3.0F38.W0 51 /r VPDPBSUDS ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst	A	V/V	AVX10.2
EVEX.512.F3.0F38.W0 51 /r VPDPBSUDS zmm1{k1}{z}, zmm2, zmm3/m512/m32bcst	A	V/V	AVX10.2
EVEX.128.NP.0F38.W0 50 /r VPDPBUUD xmm1{k1}{z}, xmm2, xmm3/m128/m32bcst	A	V/V	AVX10.2
EVEX.256.NP.0F38.W0 50 /r VPDPBUUD ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst	A	V/V	AVX10.2
EVEX.512.NP.0F38.W0 50 /r VPDPBUUD zmm1{k1}{z}, zmm2, zmm3/m512/m32bcst	A	V/V	AVX10.2
EVEX.128.NP.0F38.W0 51 /r VPDPBUUDS xmm1{k1}{z}, xmm2, xmm3/m128/m32bcst	A	V/V	AVX10.2

Continued on next page...

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.NP.0F38.W0 51 /r VPDPBUUDS ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst	A	V/V	AVX10.2
EVEX.512.NP.0F38.W0 51 /r VPDPBUUDS zmm1{k1}{z}, zmm2, zmm3/m512/m32bcst	A	V/V	AVX10.2

10.3.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(rw)	VVVV(r)	MODRM.R/M(r)	N/A

10.3.2 DESCRIPTION

Note:

For instructions with a CPUID feature flag specifying Intel® AVX10, the programmer must check the available vector options on the processor at run-time via the VL256 and VL512 bits in Converged Vector ISA Leaf 0x24. These bits enumerate whether the corresponding vector lengths are supported (128-bit vectors are always supported) and as such will determine the set of instructions available to the programmer listed in the above opcode table.

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

For unsigned saturation, when an individual result value is beyond the range of an unsigned doubleword (that is, greater than FFFFF_FFFFH), the saturated unsigned doubleword integer value of FFFF_FFFFH stored in the doubleword destination.

For signed saturation, when an individual result is beyond the range of a signed doubleword integer (that is, greater than 7FFF_FFFFH or less than 8000_0000H), the saturated value of 7FFF_FFFFH or 8000_0000H, respectively, is written to the destination operand.

The EVEX encoded form of this instruction supports memory fault suppression.

10.3.3 OPERATION

```
VPDPB[UU,SS,SU]D[,S] dest, src1, src2    // EVEX encoded version
VL = (128,256,512)

KL := VL/32
ORIGDEST := DEST
FOR i := 0 TO KL-1:
   IF k1[i] or *no writemask*:
        // Elements of SRC1 are zero-extended to 16b and
        // byte elements of SRC2 are sign extended to 16b before multiplication.
        if SRC2 is memory and EVEX.b == 1:
             t := SRC2.dword[0]
        ELSE:
             t := SRC2.dword[i]

        if *src1 is signed*:
            src1extend := SIGN_EXTEND    // SU, SS
        else:
            src1extend := ZERO_EXTEND    // UU
        if *src2 is signed*:
            src2extend := SIGN_EXTEND    // SS
        else:
            src2extend := ZERO_EXTEND    // UU, SU

        p1word := src1extend(SRC1.byte[4*i+0]) * src2extend(t.byte[0])
        p2word := src1extend(SRC1.byte[4*i+1]) * src2extend(t.byte[1])
        p3word := src1extend(SRC1.byte[4*i+2]) * src2extend(t.byte[2])
        p4word := src1extend(SRC1.byte[4*i+3]) * src2extend(t.byte[3])

        IF *saturating*:
            IF *UU instruction version*:
                DEST.dword[i] := UNSIGNED_DWORD_SATURATE(ORIGDEST.dword[i] +
                                                         p1word + p2word + p3word + p4word)
            ELSE:
                DEST.dword[i] := SIGNED_DWORD_SATURATE(ORIGDEST.dword[i] +
                                                         p1word + p2word + p3word + p4word)
        ELSE:
            DEST.dword[i] := ORIGDEST.dword[i] + p1word + p2word + p3word + p4word

   ELSE IF *zeroing*:
        DEST.dword[i] := 0
   ELSE: // Merge masking, dest element unchanged
        DEST.dword[i] := ORIGDEST.dword[i]
DEST[MAX_VL-1:VL] := 0
```

```
VPDPBUU,SS,SUD,S dest, src1, src2    // VEX encoded version
VL = (128,256)

KL := VL/32
ORIGDEST := DEST
FOR i := 0 TO KL-1:
   // Elements of SRC1 are zero-extended to 16b and
   // byte elements of SRC2 are sign extended to 16b before multiplication.

   if *src1 is signed*:
       src1extend := SIGN_EXTEND   // SU, SS
   else:
       src1extend := ZERO_EXTEND   // UU
   if *src2 is signed*:
       src2extend := SIGN_EXTEND   // SS
   else:
       src2extend := ZERO_EXTEND   // UU, SU

   p1word := src1extend(SRC1.byte[4*i+0]) * src2extend(SRC2.dword[4*i+0])
   p2word := src1extend(SRC1.byte[4*i+1]) * src2extend(SRC2.dword[4*i+1])
   p3word := src1extend(SRC1.byte[4*i+2]) * src2extend(SRC2.dword[4*i+2])
   p4word := src1extend(SRC1.byte[4*i+3]) * src2extend(SRC2.dword[4*i+3])

   IF *saturating*:
       IF *UU instruction version*:
           DEST.dword[i] := UNSIGNED_DWORD_SATURATE(ORIGDEST.dword[i] +
                                                    p1word + p2word + p3word + p4word)
       ELSE:
           DEST.dword[i] := SIGNED_DWORD_SATURATE(ORIGDEST.dword[i] +
                                                  p1word + p2word + p3word + p4word)
   ELSE:
       DEST.dword[i] := ORIGDEST.dword[i] + p1word + p2word + p3word + p4word

DEST[MAX_VL-1:VL] := 0
```

10.3.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VPDPBSSD xmm1, xmm2, xmm3/m128	E4	N/A	AVX10.2
VPDPBSSD ymm1, ymm2, ymm3/m256	E4	N/A	AVX10.2

Continued on next page...

Instruction	Exception Type	Arithmetic Flags	CPUID
VPDPBSSD zmm1, zmm2, zmm3/m512	E4	N/A	AVX10.2
VPDPBSSDS xmm1, xmm2, xmm3/m128	E4	N/A	AVX10.2
VPDPBSSDS ymm1, ymm2, ymm3/m256	E4	N/A	AVX10.2
VPDPBSSDS zmm1, zmm2, zmm3/m512	E4	N/A	AVX10.2
VPDPBSUD xmm1, xmm2, xmm3/m128	E4	N/A	AVX10.2
VPDPBSUD ymm1, ymm2, ymm3/m256	E4	N/A	AVX10.2
VPDPBSUD zmm1, zmm2, zmm3/m512	E4	N/A	AVX10.2
VPDPBSUDS xmm1, xmm2, xmm3/m128	E4	N/A	AVX10.2
VPDPBSUDS ymm1, ymm2, ymm3/m256	E4	N/A	AVX10.2
VPDPBSUDS zmm1, zmm2, zmm3/m512	E4	N/A	AVX10.2
VPDPBUUD xmm1, xmm2, xmm3/m128	E4	N/A	AVX10.2
VPDPBUUD ymm1, ymm2, ymm3/m256	E4	N/A	AVX10.2
VPDPBUUD zmm1, zmm2, zmm3/m512	E4	N/A	AVX10.2
VPDPBUUDS xmm1, xmm2, xmm3/m128	E4	N/A	AVX10.2
VPDPBUUDS ymm1, ymm2, ymm3/m256	E4	N/A	AVX10.2
VPDPBUUDS zmm1, zmm2, zmm3/m512	E4	N/A	AVX10.2

10.4 VPDPW[SU,US,UU]D[,S]

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.128.F3.0F38.W0 D2 /r VPDPWSUD xmm1{k1}{z}, xmm2, xmm3/m128/m32bcst	A	V/V	AVX10.2
EVEX.256.F3.0F38.W0 D2 /r VPDPWSUD ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst	A	V/V	AVX10.2
EVEX.512.F3.0F38.W0 D2 /r VPDPWSUD zmm1{k1}{z}, zmm2, zmm3/m512/m32bcst	A	V/V	AVX10.2
EVEX.128.F3.0F38.W0 D3 /r VPDPWSUDS xmm1{k1}{z}, xmm2, xmm3/m128/m32bcst	A	V/V	AVX10.2
EVEX.256.F3.0F38.W0 D3 /r VPDPWSUDS ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst	A	V/V	AVX10.2
EVEX.512.F3.0F38.W0 D3 /r VPDPWSUDS zmm1{k1}{z}, zmm2, zmm3/m512/m32bcst	A	V/V	AVX10.2
EVEX.128.66.0F38.W0 D2 /r VPDPWUSD xmm1{k1}{z}, xmm2, xmm3/m128/m32bcst	A	V/V	AVX10.2
EVEX.256.66.0F38.W0 D2 /r VPDPWUSD ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst	A	V/V	AVX10.2
EVEX.512.66.0F38.W0 D2 /r VPDPWUSD zmm1{k1}{z}, zmm2, zmm3/m512/m32bcst	A	V/V	AVX10.2
EVEX.128.66.0F38.W0 D3 /r VPDPWUSDS xmm1{k1}{z}, xmm2, xmm3/m128/m32bcst	A	V/V	AVX10.2
EVEX.256.66.0F38.W0 D3 /r VPDPWUSDS ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst	A	V/V	AVX10.2
EVEX.512.66.0F38.W0 D3 /r VPDPWUSDS zmm1{k1}{z}, zmm2, zmm3/m512/m32bcst	A	V/V	AVX10.2
EVEX.128.NP.0F38.W0 D2 /r VPDPWUUD xmm1{k1}{z}, xmm2, xmm3/m128/m32bcst	A	V/V	AVX10.2
EVEX.256.NP.0F38.W0 D2 /r VPDPWUUD ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst	A	V/V	AVX10.2
EVEX.512.NP.0F38.W0 D2 /r VPDPWUUD zmm1{k1}{z}, zmm2, zmm3/m512/m32bcst	A	V/V	AVX10.2
EVEX.128.NP.0F38.W0 D3 /r VPDPWUUDS xmm1{k1}{z}, xmm2, xmm3/m128/m32bcst	A	V/V	AVX10.2

Continued on next page...

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.NP.0F38.W0 D3 /r VPDPWUUDS ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst	A	V/V	AVX10.2
EVEX.512.NP.0F38.W0 D3 /r VPDPWUUDS zmm1{k1}{z}, zmm2, zmm3/m512/m32bcst	A	V/V	AVX10.2

10.4.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(rw)	VVVV(r)	MODRM.R/M(r)	N/A

10.4.2 DESCRIPTION

Note:

For instructions with a CPUID feature flag specifying Intel® AVX10, the programmer must check the available vector options on the processor at run-time via the VL256 and VL512 bits in Converged Vector ISA Leaf 0x24. These bits enumerate whether the corresponding vector lengths are supported (128-bit vectors are always supported) and as such will determine the set of instructions available to the programmer listed in the above opcode table.

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

For unsigned saturation, when an individual result value is beyond the range of an unsigned doubleword (that is, greater than FFFFF_FFFFH), the saturated unsigned doubleword integer value of FFFF_FFFFH stored in the doubleword destination.

For signed saturation, when an individual result is beyond the range of a signed doubleword integer (that is, greater than 7FFF_FFFFH or less than 8000_0000H), the saturated value of 7FFF_FFFFH or 8000_0000H, respectively, is written to the destination operand.

The EVEX encoded form of this instruction supports memory fault suppression.

The EVEX vesion of VPDPWSSD{,S} was previously introduced with AVX512-VNNI. The VEX version of of VPDPWSSD{,S} was previously introduced with AVX-VNNI.

10.4.3 OPERATION

```
VPDPW[UU,SU,US]D[,S] dest, src1, src2   // VEX version
VL = (128, 256)
KL = VL/32
ORIGDEST := DEST

IF *src1 is signed*:   // SU
    src1extend := SIGN_EXTEND
ELSE:                  // UU,US
    src1extend := ZERO_EXTEND
IF *src2 is signed*:   // US
    src2extend := SIGN_EXTEND
ELSE:                  // UU, SU
    src2extend := ZERO_EXTEND

FOR i := 0 TO KL-1:
   p1dword := src1extend(SRC1.word[2*i+0]) * src2extend(SRC2.word[2*i+0])
   p2dword := src1extend(SRC1.word[2*i+1]) * src2extend(SRC2.word[2*i+1])
   IF *saturating version*:
        IF *UU instruction version*:
           DEST.dword[i] := UNSIGNED_DWORD_SATURATE(ORIGDEST.dword[i] + p1dword + p2dword)
        ELSE:
           DEST.dword[i] := SIGNED_DWORD_SATURATE(ORIGDEST.dword[i] + p1dword + p2dword)
   ELSE:
        DEST.dword[i] := ORIGDEST.dword[i] + p1dword + p2dword
DEST[MAX_VL-1:VL] := 0
```

```
VPDPW[UU,US,SU]D[,S] dest, src1, src2      // EVEX  version
VL = (128,256,512)
KL = VL/32
ORIGDEST := DEST

IF *src1 is signed*:   // SU
    src1extend := SIGN_EXTEND
ELSE:                  // UU,US
    src1extend := ZERO_EXTEND
IF *src2 is signed*:   // US
    src2extend := SIGN_EXTEND
ELSE:                  // UU, SU
    src2extend := ZERO_EXTEND

FOR i := 0 TO KL-1:
   IF k1[i] or *no writemask*:
        IF SRC2 is memory and EVEX.b == 1:
             t := SRC2.dword[0]
        ELSE:
             t := SRC2.dword[i]

        p1dword := src1extend(SRC1.word[2*i+0]) * src2extend(t.word[0])
        p2dword := src1extend(SRC1.word[2*i+1]) * src2extend(t.word[1])

        IF *saturating*:
            IF *UU instruction version*:
                DEST.dword[i] := UNSIGNED_DWORD_SATURATE(ORIGDEST.dword[i] +
                                                         p1dword + p2dword)
            ELSE:
                DEST.dword[i] := SIGNED_DWORD_SATURATE(ORIGDEST.dword[i] +
                                                         p1dword + p2dword)
        ELSE:
            DEST.dword[i] := ORIGDEST.dword[i] + p1dword + p2dword

   ELSE IF *zeroing*:
        DEST.dword[i] := 0
   ELSE: // Merge masking, dest element unchanged
        DEST.dword[i] := ORIGDEST.dword[i]
DEST[MAX_VL-1:VL] := 0
```

10.4.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VPDPWSUD xmm1, xmm2, xmm3/m128	E4	N/A	AVX10.2
VPDPWSUD ymm1, ymm2, ymm3/m256	E4	N/A	AVX10.2
VPDPWSUD zmm1, zmm2, zmm3/m512	E4	N/A	AVX10.2
VPDPWSUDS xmm1, xmm2, xmm3/m128	E4	N/A	AVX10.2
VPDPWSUDS ymm1, ymm2, ymm3/m256	E4	N/A	AVX10.2
VPDPWSUDS zmm1, zmm2, zmm3/m512	E4	N/A	AVX10.2
VPDPWUSD xmm1, xmm2, xmm3/m128	E4	N/A	AVX10.2
VPDPWUSD ymm1, ymm2, ymm3/m256	E4	N/A	AVX10.2
VPDPWUSD zmm1, zmm2, zmm3/m512	E4	N/A	AVX10.2
VPDPWUSDS xmm1, xmm2, xmm3/m128	E4	N/A	AVX10.2
VPDPWUSDS ymm1, ymm2, ymm3/m256	E4	N/A	AVX10.2
VPDPWUSDS zmm1, zmm2, zmm3/m512	E4	N/A	AVX10.2
VPDPWUUD xmm1, xmm2, xmm3/m128	E4	N/A	AVX10.2
VPDPWUUD ymm1, ymm2, ymm3/m256	E4	N/A	AVX10.2
VPDPWUUD zmm1, zmm2, zmm3/m512	E4	N/A	AVX10.2
VPDPWUUDS xmm1, xmm2, xmm3/m128	E4	N/A	AVX10.2
VPDPWUUDS ymm1, ymm2, ymm3/m256	E4	N/A	AVX10.2
VPDPWUUDS zmm1, zmm2, zmm3/m512	E4	N/A	AVX10.2

Chapter 11

INTEL® AVX10 MINMAX INSTRUCTIONS

11.1 VMINMAXBF16

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.128.F2.0F3A.W0 52 /r /ib VMINMAXBF16 xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst, imm8	A	V/V	AVX10.2
EVEX.256.F2.0F3A.W0 52 /r /ib VMINMAXBF16 ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst, imm8	A	V/V	AVX10.2
EVEX.512.F2.0F3A.W0 52 /r /ib VMINMAXBF16 zmm1{k1}{z}, zmm2, zmm3/m512/m16bcst, imm8	A	V/V	AVX10.2

11.1.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(w)	VVVV(r)	MODRM.R/M(r)	IMM8(r)

11.1.2 DESCRIPTION

Note:

For instructions with a CPUID feature flag specifying Intel® AVX10, the programmer must check the available vector options on the processor at run-time via the VL256 and VL512 bits in Converged Vector ISA Leaf 0x24. These bits enumerate whether the corresponding vector lengths are supported (128-bit vectors are always supported) and as such will determine the set of instructions available to the programmer listed in the above opcode table.

This instruction perform min/max comparison between two SIMD registers. An immediate control selects which comparison operation is performed, and also allows to override the sign of the comparison result.

The numeric behavior of the comparison operations are slightly different than VRANGE* and FP MIN/MAX instructions, and follows the "Minimum and Maximum operations" definitions in IEEE-754-2019 standard section 9.6.

The immediate bytes controls the comparison operation according to Table 11.1 and the sign control according to Table 11.2.

The sign control indication ignores NAN signs: it does not manipulate the sign if the result is a NAN, and does not copy the sign of SRC1 (for sign control = 0b00) if SRC1 is a NAN. NaN propagation and sign control behavior in case of NaN operands are described in Table 11.3 and Table 11.4.

This instruction does not generate floating point exceptions and does not consult or update MXCSR. Denormal bfloat16 input operands are treated as zeros (DAZ) and denormal bfloat16 outputs are flushed to zero (FTZ).

11.1.3 OPERATION

```
VMINMAXBF16 dest {k1}, src1, src2, imm8
VL = 128, 256 or 512
KL := VL / 16

for i := 0 to KL-1:
    if k1[i] or *no writemask*:
        if src2 is memory and (EVEX.b == 1):
            dest.bf16[i] := minmax(src1.bf16[i], src2.bf16[0],
                                   imm8, daz=true, except=false)
        else:
            dest.bf16[i] := minmax(src1.bf16[i], src2.bf16[i],
                                  imm8, daz=true, except=false)
    else if *zeroing*:
            dest.bf16[i] := 0
    //else dest.bf16[i] remains unchanged

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

11.1.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VMINMAXBF16 xmm1, xmm2, xmm3/m128, imm8	E4	N/A	AVX10.2
VMINMAXBF16 ymm1, ymm2, ymm3/m256, imm8	E4	N/A	AVX10.2
VMINMAXBF16 zmm1, zmm2, zmm3/m512, imm8	E4	N/A	AVX10.2

11.2 VMINMAX[PH,PS,PD]

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.128.66.0F3A.W1 52 /r /ib VMINMAXPD xmm1{k1}{z}, xmm2, xmm3/m128/m64bcst, imm8	A	V/V	AVX10.2
EVEX.256.66.0F3A.W1 52 /r /ib VMINMAXPD ymm1{k1}{z}, ymm2, ymm3/m256/m64bcst {sae}, imm8	A	V/V	AVX10.2
EVEX.512.66.0F3A.W1 52 /r /ib VMINMAXPD zmm1{k1}{z}, zmm2, zmm3/m512/m64bcst {sae}, imm8	A	V/V	AVX10.2
EVEX.128.NP.0F3A.W0 52 /r /ib VMINMAXPH xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst, imm8	A	V/V	AVX10.2
EVEX.256.NP.0F3A.W0 52 /r /ib VMINMAXPH ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst {sae}, imm8	A	V/V	AVX10.2
EVEX.512.NP.0F3A.W0 52 /r /ib VMINMAXPH zmm1{k1}{z}, zmm2, zmm3/m512/m16bcst {sae}, imm8	A	V/V	AVX10.2
EVEX.128.66.0F3A.W0 52 /r /ib VMINMAXPS xmm1{k1}{z}, xmm2, xmm3/m128/m32bcst, imm8	A	V/V	AVX10.2
EVEX.256.66.0F3A.W0 52 /r /ib VMINMAXPS ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst {sae}, imm8	A	V/V	AVX10.2
EVEX.512.66.0F3A.W0 52 /r /ib VMINMAXPS zmm1{k1}{z}, zmm2, zmm3/m512/m32bcst {sae}, imm8	A	V/V	AVX10.2

11.2.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(w)	VVVV(r)	MODRM.R/M(r)	IMM8(r)

11.2.2 DESCRIPTION

Note:

For instructions with a CPUID feature flag specifying Intel® AVX10, the programmer must check the available vector options on the processor at run-time via the VL256 and VL512 bits in Converged Vector ISA Leaf 0x24. These bits enumerate whether the corresponding vector lengths are supported (128-bit vectors are always supported) and as such will determine the set of instructions available to the programmer listed in the above opcode table.

This instruction perform min/max comparison between two SIMD registers. An immediate control selects which comparison operation is performed, and also allows to override the sign of the comparison result.

The numeric behavior of the comparison operations are slightly different than VRANGE* and FP MIN/MAX instructions, and follows the "Minimum and Maximum operations" definitions in IEEE-754-2019 standard section 9.6.

Table 11.1 describes the encoding of the imm8 operand. Bits [7:5] are reserved. #IE signalling behavior differs betwen operations which do not have a "Number" suffix[1] and those that are suffixed with "Number"[2].

This instruction raises Invalid exception (#IE) if either of the operands is an SNAN, and a denormal exception (#DE) if either operand is a denormal and none of the operands are NAN.

The sign control indication ignores NAN signs: it does not manipulate the sign if the result is a NAN, and does not copy the sign of SRC1 (for sign control = 0b00) if SRC1 is a NAN. NaN propagation and sign control behavior in case of NaN operands are described in Table 11.3 and Table 11.4.

[1] -0 < +0, #IE is reported in case of any sNaN source operand
[2] -0 < +0, #IE if both are NaN, and either one is sNaN

imm8[4] min/max	imm8[1:0] Op-select	Operation	Description
0b0	0b00	minimum[1]	x if x <=y, y if y< x, and a quiet NaN if either operand is a NaN.
0b0	0b01	maximum[1]	x if x >=y, y if y >x, and a quiet NaN if either operand is a NaN.
0b0	0b10	minimumMagnitude[1]	x if |x|< |y|, y if |y|< |x|, otherwise minimum(x, y).
0b0	0b11	maximumMagnitude[1]	x if |x|> |y|, y if |y|> |x|, otherwise maximum(x, y).
0b1	0b00	minimumNumber[2]	x if x <=y, y if y< x, and the number if one operand is a number and the other is a NaN (including signaling NaN which is ignored and not converted to a quiet NaN). If both operands are NaNs, a quiet NaN is returned. If either operand is a signaling NaN, an invalid operation exception is signaled.
0b1	0b01	maximumNumber[2]	x if x >=y, y if y >x, and the number if one operand is a number and the other is a NaN (including signaling NaN which is ignored and not converted to a quiet NaN). If both operands are NaNs, a quiet NaN is returned. If either operand is a signaling NaN, an invalid operation exception is signaled.
0b1	0b10	minimumMagnitudeNumber[2]	x if |x|< |y|, y if |y| < |x|, otherwise minimumNumber(x, y).
0b1	0b11	maximumMagnitudeNumber[2]	x if |x| >| y|, y if |y| >| x|, otherwise maximumNumber(x, y).

Table 11.1: MINMAX operation selection according to imm8[4] and imm8[1:0].

imm8[3:2] Sign control	Sign
0b00	Select sign (src1)
0b01	Select sign (compare result)
0b10	Set sign to 0
0b11	Set sign to 1

Table 11.2: MINMAX sign control according to imm8[3:2].

src1	src2	Result	IE Signaling Due To Comparison	Imm8[3:2] effect to range output
sNaN1	sNAN2	Quiet(sNaN1)	Yes	Ignored
sNaN1	qNaN2	Quiet(sNaN1)	Yes	Ignored
sNaN1	Norm2	Quiet(sNaN1)	Yes	Ignored
qNaN1	sNaN2	Quiet(sNaN2)	Yes	Ignored
qNaN1	qNaN2	qNaN1	No	Ignored
qNaN1	Norm2	qNaN1	No	Ignored
Norm1	sNaN2	Quiet(sNaN2)	Yes	Ignored
Norm1	qNaN2	qNaN2	No	Ignored

Table 11.3: NaN propagation of one or more NaN input values and effect of imm8[3:2] for minimum, minimumMagnitude, maximum, maximumMagnitude MINMAX operations.

src1	src2	Result	IE Signaling Due To Comparison	Imm8[3:2] effect to range output
sNaN1	sNAN2	Quiet(sNaN1)	Yes	Ignored
sNaN1	qNaN2	Quiet(sNaN1)	Yes	Ignored
sNaN1	Norm2	Norm2	Yes	Imm8[3:2] == 0b00 ignored other values applicable
qNaN1	sNaN2	Quiet(sNaN2)	Yes	Ignored
qNaN1	qNaN2	qNaN1	No	Ignored
qNaN1	Norm2	Norm2	No	Imm8[3:2] == 0b00 ignored other values applicable
Norm1	sNaN2	Norm1	Yes	Applicable
Norm1	qNaN2	Norm1	No	Applicable

Table 11.4: NaN propagation of one or more NaN input values and effect of imm8[3:2] for minimumNumber, minimumMagnitudeNumber, maximumNumber, maximumMagnitudeNumber MINMAX operations.

imm8[4] min/max	imm8[1:0] Op-select	Operation	Comparison Result for Opposite-Signed Zero
0	00	Minimum	-0
1	00	MinimumNumber	-0
0	10	MinimumMagnitude	-0
1	10	MinimumMagnitudeNumber	-0
0	01	Maximum	+0
1	01	MaximumNumber	+0
0	11	MaximumMagnitude	+0
1	11	MaximumMagnitudeNumber	+0

Table 11.5: MINMAX Operation Behavior With Signed Comparison of Opposite-Signed Zeros (src1=-0 and src2=+0, or src1=+0 and src2=-0)

imm8[4] min/max	imm8[1:0] Op-select	Operation	Comparison Result for Equal Magnitude Inputs
0	00	Minimum	b
1	00	MinimumNumber	b
0	10	MinimumMagnitude	b
1	10	MinimumMagnitudeNumber	b
0	01	Maximum	a
1	01	MaximumNumber	a
0	11	MaximumMagnitude	a
1	11	MaximumMagnitudeNumber	a

Table 11.6: MINMAX Operation Behavior With Equal Magnitude Comparisons (src1=a and src2=b, or src1=b and src2=a, where |a|=|b| and a>0 and b<0)

11.2.3 OPERATION

```
VMINMAXPD dest {k1}, src1, src2, imm8
VL = 128, 256 or 512
KL := VL / 64

for i := 0 to KL-1:
    if k1[i] or *no writemask*:
        if src2 is memory and (EVEX.b == 1):
            dest.f64[i] := minmax(src1.f64[i], src2.f64[0],
                                  imm8, daz=MXCSR.DAZ, except=true)
        else:
            dest.f64[i] := minmax(src1.f64[i], src2.f64[i],
                                  imm8, daz=MXCSR.DAZ, except=true)
    else if *zeroing*:
            dest.f64[i] := 0
    //else dest.f64[i] remains unchanged

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

```
VMINMAXPS dest {k1}, src1, src2, imm8
VL = 128, 256 or 512
KL := VL / 32

for i := 0 to KL-1:
    if k1[i] or *no writemask*:
        if src2 is memory and (EVEX.b == 1):
            dest.f32[i] := minmax(src1.f32[i], src2.f32[0],
                                  imm8, daz=MXCSR.DAZ, except=true)
        else:
            dest.f32[i] := minmax(src1.f32[i], src2.f32[i],
                                  imm8, daz=MXCSR.DAZ, except=true)
    else if *zeroing*:
            dest.f32[i] := 0
    //else dest.f32[i] remains unchanged

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

```
VMINMAXPH dest {k1}, src1, src2, imm8
VL = 128, 256 or 512
KL := VL / 16

for i := 0 to KL-1:
    if k1[i] or *no writemask*:
        if src2 is memory and (EVEX.b == 1):
            dest.f16[i] := minmax(src1.f16[i], src2.f16[0],
                                  imm8, daz=false, except=true)
        else:
            dest.f16[i] := minmax(src1.f16[i], src2.f16[i],
                                  imm8, daz=false, except=true)
    else if *zeroing*:
            dest.f16[i] := 0
    //else dest.f16[i] remains unchanged

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

11.2.4 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Denormal, Invalid In abbreviated form, this includes: DE, IE

11.2.5 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VMINMAXPD xmm1, xmm2, xmm3/m128, imm8	E2	ID	AVX10.2
VMINMAXPD ymm1, ymm2, ymm3/m256, imm8	E2	ID	AVX10.2
VMINMAXPD zmm1, zmm2, zmm3/m512, imm8	E2	ID	AVX10.2
VMINMAXPH xmm1, xmm2, xmm3/m128, imm8	E2	ID	AVX10.2
VMINMAXPH ymm1, ymm2, ymm3/m256, imm8	E2	ID	AVX10.2

Continued on next page...

Instruction	Exception Type	Arithmetic Flags	CPUID
VMINMAXPH zmm1, zmm2, zmm3/m512, imm8	E2	ID	AVX10.2
VMINMAXPS xmm1, xmm2, xmm3/m128, imm8	E2	ID	AVX10.2
VMINMAXPS ymm1, ymm2, ymm3/m256, imm8	E2	ID	AVX10.2
VMINMAXPS zmm1, zmm2, zmm3/m512, imm8	E2	ID	AVX10.2

11.3 VMINMAX[SH,SS,SD]

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.LLIG.66.0F3A.W1 53 /r /ib VMINMAXSD xmm1{k1}{z}, xmm2, xmm3/m64 {sae}, imm8	A	V/V	AVX10.2
EVEX.LLIG.NP.0F3A.W0 53 /r /ib VMINMAXSH xmm1{k1}{z}, xmm2, xmm3/m16 {sae}, imm8	A	V/V	AVX10.2
EVEX.LLIG.66.0F3A.W0 53 /r /ib VMINMAXSS xmm1{k1}{z}, xmm2, xmm3/m32 {sae}, imm8	A	V/V	AVX10.2

11.3.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	SCALAR	MODRM.REG(w)	VVVV(r)	MODRM.R/M(r)	IMM8(r)

11.3.2 DESCRIPTION

Note:

For SCALAR instructions with a CPUID feature flag specifying Intel® AVX10, the programmer can skip the check for the available vector options on the processor at run-time via the VL256 and VL512 bits in Converged Vector ISA Leaf 0x24. These bits enumerate whether the corresponding vector lengths are supported and usually determine the set of instructions available to the programmer. However, scalar instructions are architected without the notion of vector length, so this step does not strictly apply.

This instruction perform min/max comparison between two elements in SIMD registers. An immediate control selects which comparison operation is performed, and also allows to override the sign of the comparison result.

The numeric behavior of the comparison operations are slightly different than VRANGE* and FP MIN/MAX instructions, and follows the "Minimum and Maximum operations" definitions in IEEE-754-2019 standard section 9.6.

The immediate bytes controls the comparison operation according to Table 11.1 and the sign control according to Table 11.2.

This instruction raises Invalid exception (#IE) if either of the operands is an SNAN, and a denormal exception (#DE) if either operand is a denormal and none of the operands are NAN.

The sign control indication ignores NAN signs: it does not manipulate the sign if the result is a NAN, and does not copy the sign of SRC1 (for sign control = 0b00) if SRC1 is a NAN. NaN propagation and sign control behavior in case of NaN operands are described in Table 11.3 and Table 11.4.

Bits 127:16/32/64 are copied to the destination from the respective elements of the first operand

11.3.3 OPERATION

```
VMINMAXSD dest {k1}, src1, src2, imm8
VL = 128, 256 or 512

if k1[0] or *no writemask*:
    dest.f64[0] := minmax(src1.f64[0], src2.f64[0],
                          imm8, daz=MXCSR.DAZ, except=true)
else if *zeroing*:
        dest.f64[0] := 0
//else dest.f64[0] remains unchanged

dest[127:64] := src1[127:64]
dest[MAX_VL-1:VL] := 0
```

```
VMINMAXSS dest {k1}, src1, src2, imm8
VL = 128, 256 or 512

if k1[0] or *no writemask*:
    dest.f32[0] := minmax(src1.f32[0], src2.f32[0],
                          imm8, daz=MXCSR.DAZ, except=true)
else if *zeroing*:
    dest.f32[0] := 0
//else dest.f32[0] remains unchanged

dest[127:32] := src1[127:32]
dest[MAX_VL-1:VL] := 0
```

```
VMINMAXSH dest {k1}, src1, src2, imm8
VL = 128, 256 or 512

if k1[0] or *no writemask*:
    dest.f16[0] := minmax(src1.f16[0], src2.f16[0],
                          imm8, daz=false, except=true)
else if *zeroing*:
    dest.f16[0] := 0
//else dest.f16[0] remains unchanged

dest[127:16] := src1[127:16]
dest[MAX_VL-1:VL] := 0
```

11.3.4 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Denormal, Invalid In abbreviated form, this includes: DE, IE

11.3.5 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VMINMAXSD xmm1, xmm2, xmm3/m64, imm8	E3	ID	AVX10.2
VMINMAXSH xmm1, xmm2, xmm3/m16, imm8	E3	ID	AVX10.2
VMINMAXSS xmm1, xmm2, xmm3/m32, imm8	E3	ID	AVX10.2

Chapter 12

INTEL® AVX10 NEW INSTRUCTION FORMS

12.1 VADDPD

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.66.0F.W1 58 /r VADDPD ymm1{k1}{z}, ymm2, ymm3/m256/m64bcst {er}	A	V/V	AVX10.2

12.1.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(w)	VVVV(r)	MODRM.R/M(r)	N/A

12.1.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.1.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Denormal, Invalid, Overflow, Precision, Underflow In abbreviated form, this includes: DE, IE, OE, PE, UE

12.1.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VADDPD ymm1, ymm2, ymm3/m256	E2	IOUPD	AVX10.2

12.2 VADDPH

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.NP.MAP5.W0 58 /r VADDPH ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst {er}	A	V/V	AVX10.2

12.2.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(w)	VVVV(r)	MODRM.R/M(r)	N/A

12.2.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.2.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Denormal, Invalid, Overflow, Precision, Underflow In abbreviated form, this includes: DE, IE, OE, PE, UE

12.2.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VADDPH ymm1, ymm2, ymm3/m256	E2	IOUPD	AVX10.2

12.3 VADDPS

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.NP.0F.W0 58 /r VADDPS ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst {er}	A	V/V	AVX10.2

12.3.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(w)	VVVV(r)	MODRM.R/M(r)	N/A

12.3.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.3.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Denormal, Invalid, Overflow, Precision, Underflow In abbreviated form, this includes: DE, IE, OE, PE, UE

12.3.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VADDPS ymm1, ymm2, ymm3/m256	E2	IOUPD	AVX10.2

12.4 VCMPPD

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.66.0F.W1 C2 /r /ib VCMPPD k1{k2}, ymm2, ymm3/m256/m64bcst {sae}, imm8	A	V/V	AVX10.2

12.4.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(w)	VVVV(r)	MODRM.R/M(r)	IMM8(r)

12.4.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.4.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Denormal, Invalid In abbreviated form, this includes: DE, IE

12.4.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VCMPPD k1, ymm2, ymm3/m256, imm8	E2	ID	AVX10.2

12.5 VCMPPH

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.NP.0F3A.W0 C2 /r /ib VCMPPH k1{k2}, ymm2, ymm3/m256/m16bcst {sae}, imm8	A	V/V	AVX10.2

12.5.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(w)	VVVV(r)	MODRM.R/M(r)	IMM8(r)

12.5.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.5.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Denormal, Invalid In abbreviated form, this includes: DE, IE

12.5.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VCMPPH k1, ymm2, ymm3/m256, imm8	E2	ID	AVX10.2

12.6 VCMPPS

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.NP.0F.W0 C2 /r /ib VCMPPS k1{k2}, ymm2, ymm3/m256/m32bcst {sae}, imm8	A	V/V	AVX10.2

12.6.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(w)	VVVV(r)	MODRM.R/M(r)	IMM8(r)

12.6.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.6.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Denormal, Invalid In abbreviated form, this includes: DE, IE

12.6.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VCMPPS k1, ymm2, ymm3/m256, imm8	E2	ID	AVX10.2

12.7 VCVTDQ2PH

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.NP.MAP5.W0 5B /r VCVTDQ2PH xmm1{k1}{z}, ymm2/m256/m32bcst {er}	A	V/V	AVX10.2

12.7.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(w)	MODRM.R/M(r)	N/A	N/A

12.7.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.7.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Overflow, Precision In abbreviated form, this includes: OE, PE

12.7.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VCVTDQ2PH xmm1, ymm2/m256	E2	PO	AVX10.2

12.8 VCVTDQ2PS

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.NP.0F.W0 5B /r VCVTDQ2PS ymm1{k1}{z}, ymm2/m256/m32bcst {er}	A	V/V	AVX10.2

12.8.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(w)	MODRM.R/M(r)	N/A	N/A

12.8.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.8.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Precision In abbreviated form, this includes: PE

12.8.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VCVTDQ2PS ymm1, ymm2/m256	E2	P	AVX10.2

12.9 VCVTPD2DQ

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.F2.0F.W1 E6 /r VCVTPD2DQ xmm1{k1}{z}, ymm2/m256/m64bcst {er}	A	V/V	AVX10.2

12.9.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(w)	MODRM.R/M(r)	N/A	N/A

12.9.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.9.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Invalid, Precision In abbreviated form, this includes: IE, PE

12.9.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VCVTPD2DQ xmm1, ymm2/m256	E2	IP	AVX10.2

12.10 VCVTPD2PH

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.66.MAP5.W1 5A /r VCVTPD2PH xmm1{k1}{z}, ymm2/m256/m64bcst {er}	A	V/V	AVX10.2

12.10.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(w)	MODRM.R/M(r)	N/A	N/A

12.10.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.10.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Denormal, Invalid, Overflow, Precision, Underflow In abbreviated form, this includes: DE, IE, OE, PE, UE

12.10.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VCVTPD2PH xmm1, ymm2/m256	E2	IOUPD	AVX10.2

12.11 VCVTPD2PS

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.66.0F.W1 5A /r VCVTPD2PS xmm1{k1}{z}, ymm2/m256/m64bcst {er}	A	V/V	AVX10.2

12.11.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(w)	MODRM.R/M(r)	N/A	N/A

12.11.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.11.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Denormal, Invalid, Overflow, Precision, Underflow In abbreviated form, this includes: DE, IE, OE, PE, UE

12.11.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VCVTPD2PS xmm1, ymm2/m256	E2	IOUPD	AVX10.2

12.12 VCVTPD2QQ

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.66.0F.W1 7B /r VCVTPD2QQ ymm1{k1}{z}, ymm2/m256/m64bcst {er}	A	V/V	AVX10.2

12.12.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(w)	MODRM.R/M(r)	N/A	N/A

12.12.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.12.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Invalid, Precision In abbreviated form, this includes: IE, PE

12.12.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VCVTPD2QQ ymm1, ymm2/m256	E2	IP	AVX10.2

12.13 VCVTPD2UDQ

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.NP.0F.W1 79 /r VCVTPD2UDQ xmm1{k1}{z}, ymm2/m256/m64bcst {er}	A	V/V	AVX10.2

12.13.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(w)	MODRM.R/M(r)	N/A	N/A

12.13.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.13.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Invalid, Precision In abbreviated form, this includes: IE, PE

12.13.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VCVTPD2UDQ xmm1, ymm2/m256	E2	IP	AVX10.2

12.14 VCVTPD2UQQ

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.66.0F.W1 79 /r VCVTPD2UQQ ymm1{k1}{z}, ymm2/m256/m64bcst {er}	A	V/V	AVX10.2

12.14.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(w)	MODRM.R/M(r)	N/A	N/A

12.14.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.14.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Invalid, Precision In abbreviated form, this includes: IE, PE

12.14.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VCVTPD2UQQ ymm1, ymm2/m256	E2	IP	AVX10.2

12.15 VCVTPH2DQ

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.66.MAP5.W0 5B /r VCVTPH2DQ ymm1{k1}{z}, xmm2/m128/m16bcst {er}	A	V/V	AVX10.2

12.15.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	HALF	MODRM.REG(w)	MODRM.R/M(r)	N/A	N/A

12.15.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.15.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Invalid, Precision In abbreviated form, this includes: IE, PE

12.15.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VCVTPH2DQ ymm1, xmm2/m128	E2	IP	AVX10.2

12.16 VCVTPH2PD

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.NP.MAP5.W0 5A /r VCVTPH2PD ymm1{k1}{z}, xmm2/m64/m16bcst {sae}	A	V/V	AVX10.2

12.16.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	QUARTER	MODRM.REG(w)	MODRM.R/M(r)	N/A	N/A

12.16.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.16.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Denormal, Invalid In abbreviated form, this includes: DE, IE

12.16.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VCVTPH2PD ymm1, xmm2/m64	E2	ID	AVX10.2

12.17 VCVTPH2PS

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.66.0F38.W0 13 /r VCVTPH2PS ymm1{k1}{z}, xmm2/m128 {sae}	A	V/V	AVX10.2

12.17.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	HALFMEM	MODRM.REG(w)	MODRM.R/M(r)	N/A	N/A

12.17.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.17.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Invalid In abbreviated form, this includes: IE

12.17.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VCVTPH2PS ymm1, xmm2/m128	E11	I	AVX10.2

12.18 VCVTPH2PSX

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.66.MAP6.W0 13 /r VCVTPH2PSX ymm1{k1}{z}, xmm2/m128/m16bcst {sae}	A	V/V	AVX10.2

12.18.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	HALF	MODRM.REG(w)	MODRM.R/M(r)	N/A	N/A

12.18.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.18.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Denormal, Invalid In abbreviated form, this includes: DE, IE

12.18.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VCVTPH2PSX ymm1, xmm2/m128	E2	ID	AVX10.2

12.19 VCVTPH2QQ

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.66.MAP5.W0 7B /r VCVTPH2QQ ymm1{k1}{z}, xmm2/m64/m16bcst {er}	A	V/V	AVX10.2

12.19.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	QUARTER	MODRM.REG(w)	MODRM.R/M(r)	N/A	N/A

12.19.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.19.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Invalid, Precision In abbreviated form, this includes: IE, PE

12.19.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VCVTPH2QQ ymm1, xmm2/m64	E2	IP	AVX10.2

12.20 VCVTPH2UDQ

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.NP.MAP5.W0 79 /r VCVTPH2UDQ ymm1{k1}{z}, xmm2/m128/m16bcst {er}	A	V/V	AVX10.2

12.20.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	HALF	MODRM.REG(w)	MODRM.R/M(r)	N/A	N/A

12.20.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.20.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Invalid, Precision In abbreviated form, this includes: IE, PE

12.20.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VCVTPH2UDQ ymm1, xmm2/m128	E2	IP	AVX10.2

12.21 VCVTPH2UQQ

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.66.MAP5.W0 79 /r VCVTPH2UQQ ymm1{k1}{z}, xmm2/m64/m16bcst {er}	A	V/V	AVX10.2

12.21.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	QUARTER	MODRM.REG(w)	MODRM.R/M(r)	N/A	N/A

12.21.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.21.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Invalid, Precision In abbreviated form, this includes: IE, PE

12.21.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VCVTPH2UQQ ymm1, xmm2/m64	E2	IP	AVX10.2

12.22 VCVTPH2UW

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.NP.MAP5.W0 7D /r VCVTPH2UW ymm1{k1}{z}, ymm2/m256/m16bcst {er}	A	V/V	AVX10.2

12.22.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(w)	MODRM.R/M(r)	N/A	N/A

12.22.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.22.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Invalid, Precision In abbreviated form, this includes: IE, PE

12.22.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VCVTPH2UW ymm1, ymm2/m256	E2	IP	AVX10.2

12.23 VCVTPH2W

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.66.MAP5.W0 7D /r VCVTPH2W ymm1{k1}{z}, ymm2/m256/m16bcst {er}	A	V/V	AVX10.2

12.23.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(w)	MODRM.R/M(r)	N/A	N/A

12.23.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.23.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Invalid, Precision In abbreviated form, this includes: IE, PE

12.23.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VCVTPH2W ymm1, ymm2/m256	E2	IP	AVX10.2

12.24 VCVTPS2DQ

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.66.0F.W0 5B /r VCVTPS2DQ ymm1{k1}{z}, ymm2/m256/m32bcst {er}	A	V/V	AVX10.2

12.24.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(w)	MODRM.R/M(r)	N/A	N/A

12.24.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.24.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Invalid, Precision In abbreviated form, this includes: IE, PE

12.24.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VCVTPS2DQ ymm1, ymm2/m256	E2	IP	AVX10.2

12.25 VCVTPS2PD

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.NP.0F.W0 5A /r VCVTPS2PD ymm1{k1}{z}, xmm2/m128/m32bcst {sae}	A	V/V	AVX10.2

12.25.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	HALF	MODRM.REG(w)	MODRM.R/M(r)	N/A	N/A

12.25.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.25.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Denormal, Invalid In abbreviated form, this includes: DE, IE

12.25.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VCVTPS2PD ymm1, xmm2/m128	E2	ID	AVX10.2

12.26 VCVTPS2PH

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.66.0F3A.W0 1D 11:rrr:bbb /ib VCVTPS2PH xmm1{k1}{z}, ymm2 {sae}, imm8	A	V/V	AVX10.2

12.26.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	HALFMEM	MODRM.R/M(w)	MODRM.REG(r)	IMM8(r)	N/A

12.26.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.26.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Denormal, Invalid, Overflow, Precision, Underflow In abbreviated form, this includes: DE, IE, OE, PE, UE

12.26.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VCVTPS2PH xmm1, ymm2, imm8	E11	IOUPD	AVX10.2

12.27 VCVTPS2PHX

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.66.MAP5.W0 1D /r VCVTPS2PHX xmm1{k1}{z}, ymm2/m256/m32bcst {er}	A	V/V	AVX10.2

12.27.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(w)	MODRM.R/M(r)	N/A	N/A

12.27.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.27.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Denormal, Invalid, Overflow, Precision, Underflow In abbreviated form, this includes: DE, IE, OE, PE, UE

12.27.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VCVTPS2PHX xmm1, ymm2/m256	E2	IOUPD	AVX10.2

12.28 VCVTPS2QQ

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.66.0F.W0 7B /r VCVTPS2QQ ymm1{k1}{z}, xmm2/m128/m32bcst {er}	A	V/V	AVX10.2

12.28.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	HALF	MODRM.REG(w)	MODRM.R/M(r)	N/A	N/A

12.28.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.28.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Invalid, Precision In abbreviated form, this includes: IE, PE

12.28.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VCVTPS2QQ ymm1, xmm2/m128	E2	IP	AVX10.2

12.29 VCVTPS2UDQ

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.NP.0F.W0 79 /r VCVTPS2UDQ ymm1{k1}{z}, ymm2/m256/m32bcst {er}	A	V/V	AVX10.2

12.29.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(w)	MODRM.R/M(r)	N/A	N/A

12.29.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.29.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Invalid, Precision In abbreviated form, this includes: IE, PE

12.29.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VCVTPS2UDQ ymm1, ymm2/m256	E2	IP	AVX10.2

12.30 VCVTPS2UQQ

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.66.0F.W0 79 /r VCVTPS2UQQ ymm1{k1}{z}, xmm2/m128/m32bcst {er}	A	V/V	AVX10.2

12.30.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	HALF	MODRM.REG(w)	MODRM.R/M(r)	N/A	N/A

12.30.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.30.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Invalid, Precision In abbreviated form, this includes: IE, PE

12.30.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VCVTPS2UQQ ymm1, xmm2/m128	E2	IP	AVX10.2

12.31 VCVTQQ2PD

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.F3.0F.W1 E6 /r VCVTQQ2PD ymm1{k1}{z}, ymm2/m256/m64bcst {er}	A	V/V	AVX10.2

12.31.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(w)	MODRM.R/M(r)	N/A	N/A

12.31.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.31.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Precision In abbreviated form, this includes: PE

12.31.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VCVTQQ2PD ymm1, ymm2/m256	E2	P	AVX10.2

12.32 VCVTQQ2PH

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.NP.MAP5.W1 5B /r VCVTQQ2PH xmm1{k1}{z}, ymm2/m256/m64bcst {er}	A	V/V	AVX10.2

12.32.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(w)	MODRM.R/M(r)	N/A	N/A

12.32.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.32.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Overflow, Precision In abbreviated form, this includes: OE, PE

12.32.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VCVTQQ2PH xmm1, ymm2/m256	E2	PO	AVX10.2

12.33 VCVTQQ2PS

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.NP.0F.W1 5B /r VCVTQQ2PS xmm1{k1}{z}, ymm2/m256/m64bcst {er}	A	V/V	AVX10.2

12.33.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(w)	MODRM.R/M(r)	N/A	N/A

12.33.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.33.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Precision In abbreviated form, this includes: PE

12.33.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VCVTQQ2PS xmm1, ymm2/m256	E2	P	AVX10.2

12.34 VCVTTPD2DQ

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.66.0F.W1 E6 /r VCVTTPD2DQ xmm1{k1}{z}, ymm2/m256/m64bcst {sae}	A	V/V	AVX10.2

12.34.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(w)	MODRM.R/M(r)	N/A	N/A

12.34.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.34.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Invalid, Precision In abbreviated form, this includes: IE, PE

12.34.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VCVTTPD2DQ xmm1, ymm2/m256	E2	IP	AVX10.2

12.35 VCVTTPD2QQ

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.66.0F.W1 7A /r VCVTTPD2QQ ymm1{k1}{z}, ymm2/m256/m64bcst {sae}	A	V/V	AVX10.2

12.35.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(w)	MODRM.R/M(r)	N/A	N/A

12.35.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.35.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Invalid, Precision In abbreviated form, this includes: IE, PE

12.35.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VCVTTPD2QQ ymm1, ymm2/m256	E2	IP	AVX10.2

12.36 VCVTTPD2UDQ

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.NP.0F.W1 78 /r VCVTTPD2UDQ xmm1{k1}{z}, ymm2/m256/m64bcst {sae}	A	V/V	AVX10.2

12.36.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(w)	MODRM.R/M(r)	N/A	N/A

12.36.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.36.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Invalid, Precision In abbreviated form, this includes: IE, PE

12.36.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VCVTTPD2UDQ xmm1, ymm2/m256	E2	IP	AVX10.2

12.37 VCVTTPD2UQQ

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.66.0F.W1 78 /r VCVTTPD2UQQ ymm1{k1}{z}, ymm2/m256/m64bcst {sae}	A	V/V	AVX10.2

12.37.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(w)	MODRM.R/M(r)	N/A	N/A

12.37.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.37.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Invalid, Precision In abbreviated form, this includes: IE, PE

12.37.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VCVTTPD2UQQ ymm1, ymm2/m256	E2	IP	AVX10.2

12.38 VCVTTPH2DQ

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.F3.MAP5.W0 5B /r VCVTTPH2DQ ymm1{k1}{z}, xmm2/m128/m16bcst {sae}	A	V/V	AVX10.2

12.38.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	HALF	MODRM.REG(w)	MODRM.R/M(r)	N/A	N/A

12.38.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.38.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Invalid, Precision In abbreviated form, this includes: IE, PE

12.38.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VCVTTPH2DQ ymm1, xmm2/m128	E2	IP	AVX10.2

12.39 VCVTTPH2QQ

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.66.MAP5.W0 7A /r VCVTTPH2QQ ymm1{k1}{z}, xmm2/m64/m16bcst {sae}	A	V/V	AVX10.2

12.39.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	QUARTER	MODRM.REG(w)	MODRM.R/M(r)	N/A	N/A

12.39.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.39.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Invalid, Precision In abbreviated form, this includes: IE, PE

12.39.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VCVTTPH2QQ ymm1, xmm2/m64	E2	IP	AVX10.2

12.40 VCVTTPH2UDQ

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.NP.MAP5.W0 78 /r VCVTTPH2UDQ ymm1{k1}{z}, xmm2/m128/m16bcst {sae}	A	V/V	AVX10.2

12.40.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	HALF	MODRM.REG(w)	MODRM.R/M(r)	N/A	N/A

12.40.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.40.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Invalid, Precision In abbreviated form, this includes: IE, PE

12.40.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VCVTTPH2UDQ ymm1, xmm2/m128	E2	IP	AVX10.2

12.41 VCVTTPH2UQQ

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.66.MAP5.W0 78 /r VCVTTPH2UQQ ymm1{k1}{z}, xmm2/m64/m16bcst {sae}	A	V/V	AVX10.2

12.41.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	QUARTER	MODRM.REG(w)	MODRM.R/M(r)	N/A	N/A

12.41.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.41.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Invalid, Precision In abbreviated form, this includes: IE, PE

12.41.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VCVTTPH2UQQ ymm1, xmm2/m64	E2	IP	AVX10.2

12.42 VCVTTPH2UW

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.NP.MAP5.W0 7C /r VCVTTPH2UW ymm1{k1}{z}, ymm2/m256/m16bcst {sae}	A	V/V	AVX10.2

12.42.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(w)	MODRM.R/M(r)	N/A	N/A

12.42.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.42.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Invalid, Precision In abbreviated form, this includes: IE, PE

12.42.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VCVTTPH2UW ymm1, ymm2/m256	E2	IP	AVX10.2

12.43 VCVTTPH2W

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.66.MAP5.W0 7C /r VCVTTPH2W ymm1{k1}{z}, ymm2/m256/m16bcst {sae}	A	V/V	AVX10.2

12.43.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(w)	MODRM.R/M(r)	N/A	N/A

12.43.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.43.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Invalid, Precision In abbreviated form, this includes: IE, PE

12.43.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VCVTTPH2W ymm1, ymm2/m256	E2	IP	AVX10.2

12.44 VCVTTPS2DQ

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.F3.0F.W0 5B /r VCVTTPS2DQ ymm1{k1}{z}, ymm2/m256/m32bcst {sae}	A	V/V	AVX10.2

12.44.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(w)	MODRM.R/M(r)	N/A	N/A

12.44.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.44.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Invalid, Precision In abbreviated form, this includes: IE, PE

12.44.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VCVTTPS2DQ ymm1, ymm2/m256	E2	IP	AVX10.2

12.45 VCVTTPS2QQ

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.66.0F.W0 7A /r VCVTTPS2QQ ymm1{k1}{z}, xmm2/m128/m32bcst {sae}	A	V/V	AVX10.2

12.45.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	HALF	MODRM.REG(w)	MODRM.R/M(r)	N/A	N/A

12.45.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.45.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Invalid, Precision In abbreviated form, this includes: IE, PE

12.45.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VCVTTPS2QQ ymm1, xmm2/m128	E2	IP	AVX10.2

12.46 VCVTTPS2UDQ

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.NP.0F.W0 78 /r VCVTTPS2UDQ ymm1{k1}{z}, ymm2/m256/m32bcst {sae}	A	V/V	AVX10.2

12.46.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(w)	MODRM.R/M(r)	N/A	N/A

12.46.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.46.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Invalid, Precision In abbreviated form, this includes: IE, PE

12.46.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VCVTTPS2UDQ ymm1, ymm2/m256	E2	IP	AVX10.2

12.47 VCVTTPS2UQQ

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.66.0F.W0 78 /r VCVTTPS2UQQ ymm1{k1}{z}, xmm2/m128/m32bcst {sae}	A	V/V	AVX10.2

12.47.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	HALF	MODRM.REG(w)	MODRM.R/M(r)	N/A	N/A

12.47.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.47.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Invalid, Precision In abbreviated form, this includes: IE, PE

12.47.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VCVTTPS2UQQ ymm1, xmm2/m128	E2	IP	AVX10.2

12.48 VCVTUDQ2PH

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.F2.MAP5.W0 7A /r VCVTUDQ2PH xmm1{k1}{z}, ymm2/m256/m32bcst {er}	A	V/V	AVX10.2

12.48.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(w)	MODRM.R/M(r)	N/A	N/A

12.48.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.48.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Overflow, Precision In abbreviated form, this includes: OE, PE

12.48.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VCVTUDQ2PH xmm1, ymm2/m256	E2	PO	AVX10.2

12.49 VCVTUDQ2PS

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.F2.0F.W0 7A /r VCVTUDQ2PS ymm1{k1}{z}, ymm2/m256/m32bcst {er}	A	V/V	AVX10.2

12.49.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(w)	MODRM.R/M(r)	N/A	N/A

12.49.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.49.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Precision In abbreviated form, this includes: PE

12.49.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VCVTUDQ2PS ymm1, ymm2/m256	E2	P	AVX10.2

12.50 VCVTUQQ2PD

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.F3.0F.W1 7A /r VCVTUQQ2PD ymm1{k1}{z}, ymm2/m256/m64bcst {er}	A	V/V	AVX10.2

12.50.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(w)	MODRM.R/M(r)	N/A	N/A

12.50.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.50.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Precision In abbreviated form, this includes: PE

12.50.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VCVTUQQ2PD ymm1, ymm2/m256	E2	P	AVX10.2

12.51 VCVTUQQ2PH

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.F2.MAP5.W1 7A /r VCVTUQQ2PH xmm1{k1}{z}, ymm2/m256/m64bcst {er}	A	V/V	AVX10.2

12.51.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(w)	MODRM.R/M(r)	N/A	N/A

12.51.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.51.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Overflow, Precision In abbreviated form, this includes: OE, PE

12.51.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VCVTUQQ2PH xmm1, ymm2/m256	E2	PO	AVX10.2

12.52 VCVTUQQ2PS

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.F2.0F.W1 7A /r VCVTUQQ2PS xmm1{k1}{z}, ymm2/m256/m64bcst {er}	A	V/V	AVX10.2

12.52.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(w)	MODRM.R/M(r)	N/A	N/A

12.52.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.52.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Precision In abbreviated form, this includes: PE

12.52.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VCVTUQQ2PS xmm1, ymm2/m256	E2	P	AVX10.2

12.53 VCVTUW2PH

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.F2.MAP5.W0 7D /r VCVTUW2PH ymm1{k1}{z}, ymm2/m256/m16bcst {er}	A	V/V	AVX10.2

12.53.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(w)	MODRM.R/M(r)	N/A	N/A

12.53.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.53.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Overflow, Precision In abbreviated form, this includes: OE, PE

12.53.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VCVTUW2PH ymm1, ymm2/m256	E2	PO	AVX10.2

12.54 VCVTW2PH

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.F3.MAP5.W0 7D /r VCVTW2PH ymm1{k1}{z}, ymm2/m256/m16bcst {er}	A	V/V	AVX10.2

12.54.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(w)	MODRM.R/M(r)	N/A	N/A

12.54.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.54.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Precision In abbreviated form, this includes: PE

12.54.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VCVTW2PH ymm1, ymm2/m256	E2	P	AVX10.2

12.55 VDIVPD

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.66.0F.W1 5E /r VDIVPD ymm1{k1}{z}, ymm2, ymm3/m256/m64bcst {er}	A	V/V	AVX10.2

12.55.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(w)	VVVV(r)	MODRM.R/M(r)	N/A

12.55.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.55.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Denormal, Divide-by-Zero, Invalid, Overflow, Precision, Underflow In abbreviated form, this includes: DE, IE, OE, PE, UE, ZE

12.55.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VDIVPD ymm1, ymm2, ymm3/m256	E2	IOUPDZ	AVX10.2

12.56 VDIVPH

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.NP.MAP5.W0 5E /r VDIVPH ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst {er}	A	V/V	AVX10.2

12.56.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(w)	VVVV(r)	MODRM.R/M(r)	N/A

12.56.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.56.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Denormal, Divide-by-Zero, Invalid, Overflow, Precision, Underflow In abbreviated form, this includes: DE, IE, OE, PE, UE, ZE

12.56.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VDIVPH ymm1, ymm2, ymm3/m256	E2	IOUPDZ	AVX10.2

12.57 VDIVPS

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.NP.0F.W0 5E /r VDIVPS ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst {er}	A	V/V	AVX10.2

12.57.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(w)	VVVV(r)	MODRM.R/M(r)	N/A

12.57.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.57.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Denormal, Divide-by-Zero, Invalid, Overflow, Precision, Underflow In abbreviated form, this includes: DE, IE, OE, PE, UE, ZE

12.57.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VDIVPS ymm1, ymm2, ymm3/m256	E2	IOUPDZ	AVX10.2

12.58 VFCMADDCPH

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.F2.MAP6.W0 56 /r VFCMADDCPH ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst {er}	A	V/V	AVX10.2

12.58.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(rw)	VVVV(r)	MODRM.R/M(r)	N/A

12.58.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.58.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Denormal, Invalid, Overflow, Precision, Underflow In abbreviated form, this includes: DE, IE, OE, PE, UE

12.58.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VFCMADDCPH ymm1, ymm2, ymm3/m256	E4	IOUPD	AVX10.2

12.59 VFCMULCPH

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.F2.MAP6.W0 D6 /r VFCMULCPH ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst {er}	A	V/V	AVX10.2

12.59.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(w)	VVVV(r)	MODRM.R/M(r)	N/A

12.59.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.59.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Denormal, Invalid, Overflow, Precision, Underflow In abbreviated form, this includes: DE, IE, OE, PE, UE

12.59.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VFCMULCPH ymm1, ymm2, ymm3/m256	E4	IOUPD	AVX10.2

12.60 VFIXUPIMMPD

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.66.0F3A.W1 54 /r /ib VFIXUPIMMPD ymm1{k1}{z}, ymm2, ymm3/m256/m64bcst {sae}, imm8	A	V/V	AVX10.2

12.60.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(rw)	VVVV(r)	MODRM.R/M(r)	IMM8(r)

12.60.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.60.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Divide-by-Zero, Invalid In abbreviated form, this includes: IE, ZE

12.60.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VFIXUPIMMPD ymm1, ymm2, ymm3/m256, imm8	E2	IZ	AVX10.2

12.61 VFIXUPIMMPS

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.66.0F3A.W0 54 /r /ib VFIXUPIMMPS ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst {sae}, imm8	A	V/V	AVX10.2

12.61.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(rw)	VVVV(r)	MODRM.R/M(r)	IMM8(r)

12.61.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.61.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Divide-by-Zero, Invalid In abbreviated form, this includes: IE, ZE

12.61.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VFIXUPIMMPS ymm1, ymm2, ymm3/m256, imm8	E2	IZ	AVX10.2

12.62 VFMADD132PD

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.66.0F38.W1 98 /r VFMADD132PD ymm1{k1}{z}, ymm2, ymm3/m256/m64bcst {er}	A	V/V	AVX10.2

12.62.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(rw)	VVVV(r)	MODRM.R/M(r)	N/A

12.62.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.62.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Denormal, Invalid, Overflow, Precision, Underflow In abbreviated form, this includes: DE, IE, OE, PE, UE

12.62.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VFMADD132PD ymm1, ymm2, ymm3/m256	E2	IOUPD	AVX10.2

12.63 VFMADD132PH

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.66.MAP6.W0 98 /r VFMADD132PH ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst {er}	A	V/V	AVX10.2

12.63.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(rw)	VVVV(r)	MODRM.R/M(r)	N/A

12.63.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.63.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Denormal, Invalid, Overflow, Precision, Underflow In abbreviated form, this includes: DE, IE, OE, PE, UE

12.63.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VFMADD132PH ymm1, ymm2, ymm3/m256	E2	IOUPD	AVX10.2

12.64 VFMADD132PS

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.66.0F38.W0 98 /r VFMADD132PS ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst {er}	A	V/V	AVX10.2

12.64.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(rw)	VVVV(r)	MODRM.R/M(r)	N/A

12.64.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.64.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Denormal, Invalid, Overflow, Precision, Underflow In abbreviated form, this includes: DE, IE, OE, PE, UE

12.64.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VFMADD132PS ymm1, ymm2, ymm3/m256	E2	IOUPD	AVX10.2

12.65 VFMADD213PD

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.66.0F38.W1 A8 /r VFMADD213PD ymm1{k1}{z}, ymm2, ymm3/m256/m64bcst {er}	A	V/V	AVX10.2

12.65.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(rw)	VVVV(r)	MODRM.R/M(r)	N/A

12.65.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.65.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Denormal, Invalid, Overflow, Precision, Underflow In abbreviated form, this includes: DE, IE, OE, PE, UE

12.65.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VFMADD213PD ymm1, ymm2, ymm3/m256	E2	IOUPD	AVX10.2

12.66 VFMADD213PH

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.66.MAP6.W0 A8 /r VFMADD213PH ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst {er}	A	V/V	AVX10.2

12.66.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(rw)	VVVV(r)	MODRM.R/M(r)	N/A

12.66.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.66.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Denormal, Invalid, Overflow, Precision, Underflow In abbreviated form, this includes: DE, IE, OE, PE, UE

12.66.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VFMADD213PH ymm1, ymm2, ymm3/m256	E2	IOUPD	AVX10.2

12.67 VFMADD213PS

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.66.0F38.W0 A8 /r VFMADD213PS ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst {er}	A	V/V	AVX10.2

12.67.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(rw)	VVVV(r)	MODRM.R/M(r)	N/A

12.67.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.67.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Denormal, Invalid, Overflow, Precision, Underflow In abbreviated form, this includes: DE, IE, OE, PE, UE

12.67.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VFMADD213PS ymm1, ymm2, ymm3/m256	E2	IOUPD	AVX10.2

12.68 VFMADD231PD

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.66.0F38.W1 B8 /r VFMADD231PD ymm1{k1}{z}, ymm2, ymm3/m256/m64bcst {er}	A	V/V	AVX10.2

12.68.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(rw)	VVVV(r)	MODRM.R/M(r)	N/A

12.68.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.68.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Denormal, Invalid, Overflow, Precision, Underflow In abbreviated form, this includes: DE, IE, OE, PE, UE

12.68.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VFMADD231PD ymm1, ymm2, ymm3/m256	E2	IOUPD	AVX10.2

12.69 VFMADD231PH

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.66.MAP6.W0 B8 /r VFMADD231PH ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst {er}	A	V/V	AVX10.2

12.69.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(rw)	VVVV(r)	MODRM.R/M(r)	N/A

12.69.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.69.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Denormal, Invalid, Overflow, Precision, Underflow In abbreviated form, this includes: DE, IE, OE, PE, UE

12.69.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VFMADD231PH ymm1, ymm2, ymm3/m256	E2	IOUPD	AVX10.2

12.70 VFMADD231PS

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.66.0F38.W0 B8 /r VFMADD231PS ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst {er}	A	V/V	AVX10.2

12.70.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(rw)	VVVV(r)	MODRM.R/M(r)	N/A

12.70.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.70.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Denormal, Invalid, Overflow, Precision, Underflow In abbreviated form, this includes: DE, IE, OE, PE, UE

12.70.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VFMADD231PS ymm1, ymm2, ymm3/m256	E2	IOUPD	AVX10.2

12.71 VFMADDCPH

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.F3.MAP6.W0 56 /r VFMADDCPH ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst {er}	A	V/V	AVX10.2

12.71.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(rw)	VVVV(r)	MODRM.R/M(r)	N/A

12.71.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.71.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Denormal, Invalid, Overflow, Precision, Underflow In abbreviated form, this includes: DE, IE, OE, PE, UE

12.71.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VFMADDCPH ymm1, ymm2, ymm3/m256	E4	IOUPD	AVX10.2

12.72 VFMADDSUB132PD

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.66.0F38.W1 96 /r VFMADDSUB132PD ymm1{k1}{z}, ymm2, ymm3/m256/m64bcst {er}	A	V/V	AVX10.2

12.72.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(rw)	VVVV(r)	MODRM.R/M(r)	N/A

12.72.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.72.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Denormal, Invalid, Overflow, Precision, Underflow In abbreviated form, this includes: DE, IE, OE, PE, UE

12.72.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VFMADDSUB132PD ymm1, ymm2, ymm3/m256	E2	IOUPD	AVX10.2

12.73 VFMADDSUB132PH

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.66.MAP6.W0 96 /r VFMADDSUB132PH ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst {er}	A	V/V	AVX10.2

12.73.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(rw)	VVVV(r)	MODRM.R/M(r)	N/A

12.73.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.73.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Denormal, Invalid, Overflow, Precision, Underflow In abbreviated form, this includes: DE, IE, OE, PE, UE

12.73.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VFMADDSUB132PH ymm1, ymm2, ymm3/m256	E2	IOUPD	AVX10.2

12.74 VFMADDSUB132PS

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.66.0F38.W0 96 /r VFMADDSUB132PS ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst {er}	A	V/V	AVX10.2

12.74.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(rw)	VVVV(r)	MODRM.R/M(r)	N/A

12.74.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.74.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Denormal, Invalid, Overflow, Precision, Underflow In abbreviated form, this includes: DE, IE, OE, PE, UE

12.74.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VFMADDSUB132PS ymm1, ymm2, ymm3/m256	E2	IOUPD	AVX10.2

12.75 VFMADDSUB213PD

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.66.0F38.W1 A6 /r VFMADDSUB213PD ymm1{k1}{z}, ymm2, ymm3/m256/m64bcst {er}	A	V/V	AVX10.2

12.75.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(rw)	VVVV(r)	MODRM.R/M(r)	N/A

12.75.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.75.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Denormal, Invalid, Overflow, Precision, Underflow In abbreviated form, this includes: DE, IE, OE, PE, UE

12.75.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VFMADDSUB213PD ymm1, ymm2, ymm3/m256	E2	IOUPD	AVX10.2

12.76 VFMADDSUB213PH

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.66.MAP6.W0 A6 /r VFMADDSUB213PH ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst {er}	A	V/V	AVX10.2

12.76.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(rw)	VVVV(r)	MODRM.R/M(r)	N/A

12.76.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.76.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Denormal, Invalid, Overflow, Precision, Underflow In abbreviated form, this includes: DE, IE, OE, PE, UE

12.76.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VFMADDSUB213PH ymm1, ymm2, ymm3/m256	E2	IOUPD	AVX10.2

12.77 VFMADDSUB213PS

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.66.0F38.W0 A6 /r VFMADDSUB213PS ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst {er}	A	V/V	AVX10.2

12.77.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(rw)	VVVV(r)	MODRM.R/M(r)	N/A

12.77.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.77.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Denormal, Invalid, Overflow, Precision, Underflow In abbreviated form, this includes: DE, IE, OE, PE, UE

12.77.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VFMADDSUB213PS ymm1, ymm2, ymm3/m256	E2	IOUPD	AVX10.2

12.78 VFMADDSUB231PD

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.66.0F38.W1 B6 /r VFMADDSUB231PD ymm1{k1}{z}, ymm2, ymm3/m256/m64bcst {er}	A	V/V	AVX10.2

12.78.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(rw)	VVVV(r)	MODRM.R/M(r)	N/A

12.78.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.78.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Denormal, Invalid, Overflow, Precision, Underflow In abbreviated form, this includes: DE, IE, OE, PE, UE

12.78.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VFMADDSUB231PD ymm1, ymm2, ymm3/m256	E2	IOUPD	AVX10.2

12.79 VFMADDSUB231PH

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.66.MAP6.W0 B6 /r VFMADDSUB231PH ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst {er}	A	V/V	AVX10.2

12.79.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(rw)	VVVV(r)	MODRM.R/M(r)	N/A

12.79.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.79.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Denormal, Invalid, Overflow, Precision, Underflow In abbreviated form, this includes: DE, IE, OE, PE, UE

12.79.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VFMADDSUB231PH ymm1, ymm2, ymm3/m256	E2	IOUPD	AVX10.2

12.80 VFMADDSUB231PS

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.66.0F38.W0 B6 /r VFMADDSUB231PS ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst {er}	A	V/V	AVX10.2

12.80.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(rw)	VVVV(r)	MODRM.R/M(r)	N/A

12.80.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.80.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Denormal, Invalid, Overflow, Precision, Underflow In abbreviated form, this includes: DE, IE, OE, PE, UE

12.80.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VFMADDSUB231PS ymm1, ymm2, ymm3/m256	E2	IOUPD	AVX10.2

12.81 VFMSUB132PD

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.66.0F38.W1 9A /r VFMSUB132PD ymm1{k1}{z}, ymm2, ymm3/m256/m64bcst {er}	A	V/V	AVX10.2

12.81.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(rw)	VVVV(r)	MODRM.R/M(r)	N/A

12.81.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.81.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Denormal, Invalid, Overflow, Precision, Underflow In abbreviated form, this includes: DE, IE, OE, PE, UE

12.81.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VFMSUB132PD ymm1, ymm2, ymm3/m256	E2	IOUPD	AVX10.2

12.82 VFMSUB132PH

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.66.MAP6.W0 9A /r VFMSUB132PH ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst {er}	A	V/V	AVX10.2

12.82.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(rw)	VVVV(r)	MODRM.R/M(r)	N/A

12.82.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.82.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Denormal, Invalid, Overflow, Precision, Underflow In abbreviated form, this includes: DE, IE, OE, PE, UE

12.82.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VFMSUB132PH ymm1, ymm2, ymm3/m256	E2	IOUPD	AVX10.2

12.83 VFMSUB132PS

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.66.0F38.W0 9A /r VFMSUB132PS ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst {er}	A	V/V	AVX10.2

12.83.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(rw)	VVVV(r)	MODRM.R/M(r)	N/A

12.83.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.83.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Denormal, Invalid, Overflow, Precision, Underflow In abbreviated form, this includes: DE, IE, OE, PE, UE

12.83.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VFMSUB132PS ymm1, ymm2, ymm3/m256	E2	IOUPD	AVX10.2

12.84 VFMSUB213PD

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.66.0F38.W1 AA /r VFMSUB213PD ymm1{k1}{z}, ymm2, ymm3/m256/m64bcst {er}	A	V/V	AVX10.2

12.84.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(rw)	VVVV(r)	MODRM.R/M(r)	N/A

12.84.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.84.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Denormal, Invalid, Overflow, Precision, Underflow In abbreviated form, this includes: DE, IE, OE, PE, UE

12.84.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VFMSUB213PD ymm1, ymm2, ymm3/m256	E2	IOUPD	AVX10.2

12.85 VFMSUB213PH

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.66.MAP6.W0 AA /r VFMSUB213PH ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst {er}	A	V/V	AVX10.2

12.85.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(rw)	VVVV(r)	MODRM.R/M(r)	N/A

12.85.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.85.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Denormal, Invalid, Overflow, Precision, Underflow In abbreviated form, this includes: DE, IE, OE, PE, UE

12.85.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VFMSUB213PH ymm1, ymm2, ymm3/m256	E2	IOUPD	AVX10.2

12.86 VFMSUB213PS

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.66.0F38.W0 AA /r VFMSUB213PS ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst {er}	A	V/V	AVX10.2

12.86.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(rw)	VVVV(r)	MODRM.R/M(r)	N/A

12.86.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.86.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Denormal, Invalid, Overflow, Precision, Underflow In abbreviated form, this includes: DE, IE, OE, PE, UE

12.86.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VFMSUB213PS ymm1, ymm2, ymm3/m256	E2	IOUPD	AVX10.2

12.87 VFMSUB231PD

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.66.0F38.W1 BA /r VFMSUB231PD ymm1{k1}{z}, ymm2, ymm3/m256/m64bcst {er}	A	V/V	AVX10.2

12.87.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(rw)	VVVV(r)	MODRM.R/M(r)	N/A

12.87.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.87.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Denormal, Invalid, Overflow, Precision, Underflow In abbreviated form, this includes: DE, IE, OE, PE, UE

12.87.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VFMSUB231PD ymm1, ymm2, ymm3/m256	E2	IOUPD	AVX10.2

12.88 VFMSUB231PH

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.66.MAP6.W0 BA /r VFMSUB231PH ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst {er}	A	V/V	AVX10.2

12.88.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(rw)	VVVV(r)	MODRM.R/M(r)	N/A

12.88.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.88.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Denormal, Invalid, Overflow, Precision, Underflow In abbreviated form, this includes: DE, IE, OE, PE, UE

12.88.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VFMSUB231PH ymm1, ymm2, ymm3/m256	E2	IOUPD	AVX10.2

12.89 VFMSUB231PS

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.66.0F38.W0 BA /r VFMSUB231PS ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst {er}	A	V/V	AVX10.2

12.89.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(rw)	VVVV(r)	MODRM.R/M(r)	N/A

12.89.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.89.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Denormal, Invalid, Overflow, Precision, Underflow In abbreviated form, this includes: DE, IE, OE, PE, UE

12.89.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VFMSUB231PS ymm1, ymm2, ymm3/m256	E2	IOUPD	AVX10.2

12.90 VFMSUBADD132PD

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.66.0F38.W1 97 /r VFMSUBADD132PD ymm1{k1}{z}, ymm2, ymm3/m256/m64bcst {er}	A	V/V	AVX10.2

12.90.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(rw)	VVVV(r)	MODRM.R/M(r)	N/A

12.90.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.90.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Denormal, Invalid, Overflow, Precision, Underflow In abbreviated form, this includes: DE, IE, OE, PE, UE

12.90.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VFMSUBADD132PD ymm1, ymm2, ymm3/m256	E2	IOUPD	AVX10.2

12.91 VFMSUBADD132PH

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.66.MAP6.W0 97 /r VFMSUBADD132PH ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst {er}	A	V/V	AVX10.2

12.91.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(rw)	VVVV(r)	MODRM.R/M(r)	N/A

12.91.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.91.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Denormal, Invalid, Overflow, Precision, Underflow In abbreviated form, this includes: DE, IE, OE, PE, UE

12.91.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VFMSUBADD132PH ymm1, ymm2, ymm3/m256	E2	IOUPD	AVX10.2

12.92 VFMSUBADD132PS

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.66.0F38.W0 97 /r VFMSUBADD132PS ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst {er}	A	V/V	AVX10.2

12.92.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(rw)	VVVV(r)	MODRM.R/M(r)	N/A

12.92.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.92.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Denormal, Invalid, Overflow, Precision, Underflow In abbreviated form, this includes: DE, IE, OE, PE, UE

12.92.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VFMSUBADD132PS ymm1, ymm2, ymm3/m256	E2	IOUPD	AVX10.2

12.93 VFMSUBADD213PD

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.66.0F38.W1 A7 /r VFMSUBADD213PD ymm1{k1}{z}, ymm2, ymm3/m256/m64bcst {er}	A	V/V	AVX10.2

12.93.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(rw)	VVVV(r)	MODRM.R/M(r)	N/A

12.93.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.93.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Denormal, Invalid, Overflow, Precision, Underflow In abbreviated form, this includes: DE, IE, OE, PE, UE

12.93.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VFMSUBADD213PD ymm1, ymm2, ymm3/m256	E2	IOUPD	AVX10.2

12.94 VFMSUBADD213PH

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.66.MAP6.W0 A7 /r VFMSUBADD213PH ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst {er}	A	V/V	AVX10.2

12.94.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(rw)	VVVV(r)	MODRM.R/M(r)	N/A

12.94.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.94.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Denormal, Invalid, Overflow, Precision, Underflow In abbreviated form, this includes: DE, IE, OE, PE, UE

12.94.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VFMSUBADD213PH ymm1, ymm2, ymm3/m256	E2	IOUPD	AVX10.2

12.95 VFMSUBADD213PS

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.66.0F38.W0 A7 /r VFMSUBADD213PS ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst {er}	A	V/V	AVX10.2

12.95.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(rw)	VVVV(r)	MODRM.R/M(r)	N/A

12.95.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.95.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Denormal, Invalid, Overflow, Precision, Underflow In abbreviated form, this includes: DE, IE, OE, PE, UE

12.95.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VFMSUBADD213PS ymm1, ymm2, ymm3/m256	E2	IOUPD	AVX10.2

12.96 VFMSUBADD231PD

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.66.0F38.W1 B7 /r VFMSUBADD231PD ymm1{k1}{z}, ymm2, ymm3/m256/m64bcst {er}	A	V/V	AVX10.2

12.96.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(rw)	VVVV(r)	MODRM.R/M(r)	N/A

12.96.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.96.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Denormal, Invalid, Overflow, Precision, Underflow In abbreviated form, this includes: DE, IE, OE, PE, UE

12.96.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VFMSUBADD231PD ymm1, ymm2, ymm3/m256	E2	IOUPD	AVX10.2

12.97 VFMSUBADD231PH

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.66.MAP6.W0 B7 /r VFMSUBADD231PH ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst {er}	A	V/V	AVX10.2

12.97.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(rw)	VVVV(r)	MODRM.R/M(r)	N/A

12.97.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.97.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Denormal, Invalid, Overflow, Precision, Underflow In abbreviated form, this includes: DE, IE, OE, PE, UE

12.97.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VFMSUBADD231PH ymm1, ymm2, ymm3/m256	E2	IOUPD	AVX10.2

12.98 VFMSUBADD231PS

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.66.0F38.W0 B7 /r VFMSUBADD231PS ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst {er}	A	V/V	AVX10.2

12.98.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(rw)	VVVV(r)	MODRM.R/M(r)	N/A

12.98.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.98.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Denormal, Invalid, Overflow, Precision, Underflow In abbreviated form, this includes: DE, IE, OE, PE, UE

12.98.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VFMSUBADD231PS ymm1, ymm2, ymm3/m256	E2	IOUPD	AVX10.2

12.99 VFMULCPH

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.F3.MAP6.W0 D6 /r VFMULCPH ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst {er}	A	V/V	AVX10.2

12.99.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(w)	VVVV(r)	MODRM.R/M(r)	N/A

12.99.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.99.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Denormal, Invalid, Overflow, Precision, Underflow In abbreviated form, this includes: DE, IE, OE, PE, UE

12.99.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VFMULCPH ymm1, ymm2, ymm3/m256	E4	IOUPD	AVX10.2

12.100 VFNMADD132PD

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.66.0F38.W1 9C /r VFNMADD132PD ymm1{k1}{z}, ymm2, ymm3/m256/m64bcst {er}	A	V/V	AVX10.2

12.100.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(rw)	VVVV(r)	MODRM.R/M(r)	N/A

12.100.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.100.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Denormal, Invalid, Overflow, Precision, Underflow In abbreviated form, this includes: DE, IE, OE, PE, UE

12.100.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VFNMADD132PD ymm1, ymm2, ymm3/m256	E2	IOUPD	AVX10.2

12.101 VFNMADD132PH

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.66.MAP6.W0 9C /r VFNMADD132PH ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst {er}	A	V/V	AVX10.2

12.101.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(rw)	VVVV(r)	MODRM.R/M(r)	N/A

12.101.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.101.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Denormal, Invalid, Overflow, Precision, Underflow In abbreviated form, this includes: DE, IE, OE, PE, UE

12.101.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VFNMADD132PH ymm1, ymm2, ymm3/m256	E2	IOUPD	AVX10.2

12.102 VFNMADD132PS

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.66.0F38.W0 9C /r VFNMADD132PS ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst {er}	A	V/V	AVX10.2

12.102.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(rw)	VVVV(r)	MODRM.R/M(r)	N/A

12.102.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.102.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Denormal, Invalid, Overflow, Precision, Underflow In abbreviated form, this includes: DE, IE, OE, PE, UE

12.102.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VFNMADD132PS ymm1, ymm2, ymm3/m256	E2	IOUPD	AVX10.2

12.103 VFNMADD213PD

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.66.0F38.W1 AC /r VFNMADD213PD ymm1{k1}{z}, ymm2, ymm3/m256/m64bcst {er}	A	V/V	AVX10.2

12.103.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(rw)	VVVV(r)	MODRM.R/M(r)	N/A

12.103.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.103.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Denormal, Invalid, Overflow, Precision, Underflow In abbreviated form, this includes: DE, IE, OE, PE, UE

12.103.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VFNMADD213PD ymm1, ymm2, ymm3/m256	E2	IOUPD	AVX10.2

12.104 VFNMADD213PH

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.66.MAP6.W0 AC /r VFNMADD213PH ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst {er}	A	V/V	AVX10.2

12.104.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(rw)	VVVV(r)	MODRM.R/M(r)	N/A

12.104.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.104.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Denormal, Invalid, Overflow, Precision, Underflow In abbreviated form, this includes: DE, IE, OE, PE, UE

12.104.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VFNMADD213PH ymm1, ymm2, ymm3/m256	E2	IOUPD	AVX10.2

12.105 VFNMADD213PS

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.66.0F38.W0 AC /r VFNMADD213PS ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst {er}	A	V/V	AVX10.2

12.105.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(rw)	VVVV(r)	MODRM.R/M(r)	N/A

12.105.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.105.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Denormal, Invalid, Overflow, Precision, Underflow In abbreviated form, this includes: DE, IE, OE, PE, UE

12.105.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VFNMADD213PS ymm1, ymm2, ymm3/m256	E2	IOUPD	AVX10.2

12.106 VFNMADD231PD

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.66.0F38.W1 BC /r VFNMADD231PD ymm1{k1}{z}, ymm2, ymm3/m256/m64bcst {er}	A	V/V	AVX10.2

12.106.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(rw)	VVVV(r)	MODRM.R/M(r)	N/A

12.106.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.106.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Denormal, Invalid, Overflow, Precision, Underflow In abbreviated form, this includes: DE, IE, OE, PE, UE

12.106.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VFNMADD231PD ymm1, ymm2, ymm3/m256	E2	IOUPD	AVX10.2

12.107 VFNMADD231PH

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.66.MAP6.W0 BC /r VFNMADD231PH ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst {er}	A	V/V	AVX10.2

12.107.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(rw)	VVVV(r)	MODRM.R/M(r)	N/A

12.107.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.107.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Denormal, Invalid, Overflow, Precision, Underflow In abbreviated form, this includes: DE, IE, OE, PE, UE

12.107.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VFNMADD231PH ymm1, ymm2, ymm3/m256	E2	IOUPD	AVX10.2

12.108 VFNMADD231PS

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.66.0F38.W0 BC /r VFNMADD231PS ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst {er}	A	V/V	AVX10.2

12.108.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(rw)	VVVV(r)	MODRM.R/M(r)	N/A

12.108.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.108.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Denormal, Invalid, Overflow, Precision, Underflow In abbreviated form, this includes: DE, IE, OE, PE, UE

12.108.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VFNMADD231PS ymm1, ymm2, ymm3/m256	E2	IOUPD	AVX10.2

12.109 VFNMSUB132PD

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.66.0F38.W1 9E /r VFNMSUB132PD ymm1{k1}{z}, ymm2, ymm3/m256/m64bcst {er}	A	V/V	AVX10.2

12.109.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(rw)	VVVV(r)	MODRM.R/M(r)	N/A

12.109.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.109.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Denormal, Invalid, Overflow, Precision, Underflow In abbreviated form, this includes: DE, IE, OE, PE, UE

12.109.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VFNMSUB132PD ymm1, ymm2, ymm3/m256	E2	IOUPD	AVX10.2

12.110 VFNMSUB132PH

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.66.MAP6.W0 9E /r VFNMSUB132PH ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst {er}	A	V/V	AVX10.2

12.110.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(rw)	VVVV(r)	MODRM.R/M(r)	N/A

12.110.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.110.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Denormal, Invalid, Overflow, Precision, Underflow In abbreviated form, this includes: DE, IE, OE, PE, UE

12.110.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VFNMSUB132PH ymm1, ymm2, ymm3/m256	E2	IOUPD	AVX10.2

12.111 VFNMSUB132PS

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.66.0F38.W0 9E /r VFNMSUB132PS ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst {er}	A	V/V	AVX10.2

12.111.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(rw)	VVVV(r)	MODRM.R/M(r)	N/A

12.111.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.111.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Denormal, Invalid, Overflow, Precision, Underflow In abbreviated form, this includes: DE, IE, OE, PE, UE

12.111.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VFNMSUB132PS ymm1, ymm2, ymm3/m256	E2	IOUPD	AVX10.2

12.112 VFNMSUB213PD

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.66.0F38.W1 AE /r VFNMSUB213PD ymm1{k1}{z}, ymm2, ymm3/m256/m64bcst {er}	A	V/V	AVX10.2

12.112.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(rw)	VVVV(r)	MODRM.R/M(r)	N/A

12.112.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.112.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Denormal, Invalid, Overflow, Precision, Underflow In abbreviated form, this includes: DE, IE, OE, PE, UE

12.112.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VFNMSUB213PD ymm1, ymm2, ymm3/m256	E2	IOUPD	AVX10.2

12.113 VFNMSUB213PH

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.66.MAP6.W0 AE /r VFNMSUB213PH ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst {er}	A	V/V	AVX10.2

12.113.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(rw)	VVVV(r)	MODRM.R/M(r)	N/A

12.113.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.113.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Denormal, Invalid, Overflow, Precision, Underflow In abbreviated form, this includes: DE, IE, OE, PE, UE

12.113.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VFNMSUB213PH ymm1, ymm2, ymm3/m256	E2	IOUPD	AVX10.2

12.114 VFNMSUB213PS

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.66.0F38.W0 AE /r VFNMSUB213PS ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst {er}	A	V/V	AVX10.2

12.114.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(rw)	VVVV(r)	MODRM.R/M(r)	N/A

12.114.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.114.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Denormal, Invalid, Overflow, Precision, Underflow In abbreviated form, this includes: DE, IE, OE, PE, UE

12.114.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VFNMSUB213PS ymm1, ymm2, ymm3/m256	E2	IOUPD	AVX10.2

12.115 VFNMSUB231PD

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.66.0F38.W1 BE /r VFNMSUB231PD ymm1{k1}{z}, ymm2, ymm3/m256/m64bcst {er}	A	V/V	AVX10.2

12.115.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(rw)	VVVV(r)	MODRM.R/M(r)	N/A

12.115.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.115.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Denormal, Invalid, Overflow, Precision, Underflow In abbreviated form, this includes: DE, IE, OE, PE, UE

12.115.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VFNMSUB231PD ymm1, ymm2, ymm3/m256	E2	IOUPD	AVX10.2

12.116 VFNMSUB231PH

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.66.MAP6.W0 BE /r VFNMSUB231PH ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst {er}	A	V/V	AVX10.2

12.116.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(rw)	VVVV(r)	MODRM.R/M(r)	N/A

12.116.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.116.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Denormal, Invalid, Overflow, Precision, Underflow In abbreviated form, this includes: DE, IE, OE, PE, UE

12.116.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VFNMSUB231PH ymm1, ymm2, ymm3/m256	E2	IOUPD	AVX10.2

12.117 VFNMSUB231PS

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.66.0F38.W0 BE /r VFNMSUB231PS ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst {er}	A	V/V	AVX10.2

12.117.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(rw)	VVVV(r)	MODRM.R/M(r)	N/A

12.117.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.117.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Denormal, Invalid, Overflow, Precision, Underflow In abbreviated form, this includes: DE, IE, OE, PE, UE

12.117.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VFNMSUB231PS ymm1, ymm2, ymm3/m256	E2	IOUPD	AVX10.2

12.118 VGETEXPPD

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.66.0F38.W1 42 /r VGETEXPPD ymm1{k1}{z}, ymm2/m256/m64bcst {sae}	A	V/V	AVX10.2

12.118.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(w)	MODRM.R/M(r)	N/A	N/A

12.118.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.118.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Denormal, Invalid In abbreviated form, this includes: DE, IE

12.118.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VGETEXPPD ymm1, ymm2/m256	E2	ID	AVX10.2

12.119 VGETEXPPH

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.66.MAP6.W0 42 /r VGETEXPPH ymm1{k1}{z}, ymm2/m256/m16bcst {sae}	A	V/V	AVX10.2

12.119.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(w)	MODRM.R/M(r)	N/A	N/A

12.119.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.119.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Denormal, Invalid In abbreviated form, this includes: DE, IE

12.119.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VGETEXPPH ymm1, ymm2/m256	E2	ID	AVX10.2

12.120 VGETEXPPS

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.66.0F38.W0 42 /r VGETEXPPS ymm1{k1}{z}, ymm2/m256/m32bcst {sae}	A	V/V	AVX10.2

12.120.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(w)	MODRM.R/M(r)	N/A	N/A

12.120.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.120.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Denormal, Invalid In abbreviated form, this includes: DE, IE

12.120.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VGETEXPPS ymm1, ymm2/m256	E2	ID	AVX10.2

12.121 VGETMANTPD

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.66.0F3A.W1 26 /r /ib VGETMANTPD ymm1{k1}{z}, ymm2/m256/m64bcst {sae}, imm8	A	V/V	AVX10.2

12.121.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(w)	MODRM.R/M(r)	IMM8(r)	N/A

12.121.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.121.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Denormal, Invalid In abbreviated form, this includes: DE, IE

12.121.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VGETMANTPD ymm1, ymm2/m256, imm8	E2	ID	AVX10.2

12.122 VGETMANTPH

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.NP.0F3A.W0 26 /r /ib VGETMANTPH ymm1{k1}{z}, ymm2/m256/m16bcst {sae}, imm8	A	V/V	AVX10.2

12.122.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(w)	MODRM.R/M(r)	IMM8(r)	N/A

12.122.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.122.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Denormal, Invalid In abbreviated form, this includes: DE, IE

12.122.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VGETMANTPH ymm1, ymm2/m256, imm8	E2	ID	AVX10.2

12.123 VGETMANTPS

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.66.0F3A.W0 26 /r /ib VGETMANTPS ymm1{k1}{z}, ymm2/m256/m32bcst {sae}, imm8	A	V/V	AVX10.2

12.123.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(w)	MODRM.R/M(r)	IMM8(r)	N/A

12.123.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.123.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Denormal, Invalid In abbreviated form, this includes: DE, IE

12.123.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VGETMANTPS ymm1, ymm2/m256, imm8	E2	ID	AVX10.2

12.124 VMAXPD

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.66.0F.W1 5F /r VMAXPD ymm1{k1}{z}, ymm2, ymm3/m256/m64bcst {sae}	A	V/V	AVX10.2

12.124.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(w)	VVVV(r)	MODRM.R/M(r)	N/A

12.124.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.124.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Denormal, Invalid In abbreviated form, this includes: DE, IE

12.124.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VMAXPD ymm1, ymm2, ymm3/m256	E2	ID	AVX10.2

12.125 VMAXPH

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.NP.MAP5.W0 5F /r VMAXPH ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst {sae}	A	V/V	AVX10.2

12.125.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(w)	VVVV(r)	MODRM.R/M(r)	N/A

12.125.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.125.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Denormal, Invalid In abbreviated form, this includes: DE, IE

12.125.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VMAXPH ymm1, ymm2, ymm3/m256	E2	ID	AVX10.2

12.126 VMAXPS

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.NP.0F.W0 5F /r VMAXPS ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst {sae}	A	V/V	AVX10.2

12.126.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(w)	VVVV(r)	MODRM.R/M(r)	N/A

12.126.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.126.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Denormal, Invalid In abbreviated form, this includes: DE, IE

12.126.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VMAXPS ymm1, ymm2, ymm3/m256	E2	ID	AVX10.2

12.127 VMINPD

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.66.0F.W1 5D /r VMINPD ymm1{k1}{z}, ymm2, ymm3/m256/m64bcst {sae}	A	V/V	AVX10.2

12.127.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(w)	VVVV(r)	MODRM.R/M(r)	N/A

12.127.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.127.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Denormal, Invalid In abbreviated form, this includes: DE, IE

12.127.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VMINPD ymm1, ymm2, ymm3/m256	E2	ID	AVX10.2

12.128 VMINPH

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.NP.MAP5.W0 5D /r VMINPH ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst {sae}	A	V/V	AVX10.2

12.128.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(w)	VVVV(r)	MODRM.R/M(r)	N/A

12.128.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.128.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Denormal, Invalid In abbreviated form, this includes: DE, IE

12.128.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VMINPH ymm1, ymm2, ymm3/m256	E2	ID	AVX10.2

12.129 VMINPS

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.NP.0F.W0 5D /r VMINPS ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst {sae}	A	V/V	AVX10.2

12.129.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(w)	VVVV(r)	MODRM.R/M(r)	N/A

12.129.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.129.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Denormal, Invalid In abbreviated form, this includes: DE, IE

12.129.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VMINPS ymm1, ymm2, ymm3/m256	E2	ID	AVX10.2

12.130 VMULPD

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.66.0F.W1 59 /r VMULPD ymm1{k1}{z}, ymm2, ymm3/m256/m64bcst {er}	A	V/V	AVX10.2

12.130.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(w)	VVVV(r)	MODRM.R/M(r)	N/A

12.130.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.130.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Denormal, Invalid, Overflow, Precision, Underflow In abbreviated form, this includes: DE, IE, OE, PE, UE

12.130.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VMULPD ymm1, ymm2, ymm3/m256	E2	IOUPD	AVX10.2

12.131 VMULPH

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.NP.MAP5.W0 59 /r VMULPH ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst {er}	A	V/V	AVX10.2

12.131.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(w)	VVVV(r)	MODRM.R/M(r)	N/A

12.131.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.131.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Denormal, Invalid, Overflow, Precision, Underflow In abbreviated form, this includes: DE, IE, OE, PE, UE

12.131.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VMULPH ymm1, ymm2, ymm3/m256	E2	IOUPD	AVX10.2

12.132 VMULPS

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.NP.0F.W0 59 /r VMULPS ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst {er}	A	V/V	AVX10.2

12.132.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(w)	VVVV(r)	MODRM.R/M(r)	N/A

12.132.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.132.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Denormal, Invalid, Overflow, Precision, Underflow In abbreviated form, this includes: DE, IE, OE, PE, UE

12.132.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VMULPS ymm1, ymm2, ymm3/m256	E2	IOUPD	AVX10.2

12.133 VRANGEPD

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.66.0F3A.W1 50 /r /ib VRANGEPD ymm1{k1}{z}, ymm2, ymm3/m256/m64bcst {sae}, imm8	A	V/V	AVX10.2

12.133.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(w)	VVVV(r)	MODRM.R/M(r)	IMM8(r)

12.133.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.133.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Denormal, Invalid In abbreviated form, this includes: DE, IE

12.133.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VRANGEPD ymm1, ymm2, ymm3/m256, imm8	E2	ID	AVX10.2

12.134 VRANGEPS

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.66.0F3A.W0 50 /r /ib VRANGEPS ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst {sae}, imm8	A	V/V	AVX10.2

12.134.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(w)	VVVV(r)	MODRM.R/M(r)	IMM8(r)

12.134.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.134.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Denormal, Invalid In abbreviated form, this includes: DE, IE

12.134.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VRANGEPS ymm1, ymm2, ymm3/m256, imm8	E2	ID	AVX10.2

12.135 VREDUCEPD

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.66.0F3A.W1 56 /r /ib VREDUCEPD ymm1{k1}{z}, ymm2/m256/m64bcst {sae}, imm8	A	V/V	AVX10.2

12.135.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(w)	MODRM.R/M(r)	IMM8(r)	N/A

12.135.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.135.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Invalid, Precision In abbreviated form, this includes: IE, PE

12.135.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VREDUCEPD ymm1, ymm2/m256, imm8	E2	IP	AVX10.2

12.136 VREDUCEPH

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.NP.0F3A.W0 56 /r /ib VREDUCEPH ymm1{k1}{z}, ymm2/m256/m16bcst {sae}, imm8	A	V/V	AVX10.2

12.136.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(w)	MODRM.R/M(r)	IMM8(r)	N/A

12.136.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.136.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Invalid, Precision In abbreviated form, this includes: IE, PE

12.136.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VREDUCEPH ymm1, ymm2/m256, imm8	E2	IP	AVX10.2

12.137 VREDUCEPS

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.66.0F3A.W0 56 /r /ib VREDUCEPS ymm1{k1}{z}, ymm2/m256/m32bcst {sae}, imm8	A	V/V	AVX10.2

12.137.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(w)	MODRM.R/M(r)	IMM8(r)	N/A

12.137.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.137.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Invalid, Precision In abbreviated form, this includes: IE, PE

12.137.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VREDUCEPS ymm1, ymm2/m256, imm8	E2	IP	AVX10.2

12.138 VRNDSCALEPD

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.66.0F3A.W1 09 /r /ib VRNDSCALEPD ymm1{k1}{z}, ymm2/m256/m64bcst {sae}, imm8	A	V/V	AVX10.2

12.138.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(w)	MODRM.R/M(r)	IMM8(r)	N/A

12.138.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.138.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Invalid, Precision In abbreviated form, this includes: IE, PE

12.138.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VRNDSCALEPD ymm1, ymm2/m256, imm8	E2	IP	AVX10.2

12.139 VRNDSCALEPH

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.NP.0F3A.W0 08 /r /ib VRNDSCALEPH ymm1{k1}{z}, ymm2/m256/m16bcst {sae}, imm8	A	V/V	AVX10.2

12.139.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(w)	MODRM.R/M(r)	IMM8(r)	N/A

12.139.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.139.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Invalid, Precision, Underflow In abbreviated form, this includes: IE, PE, UE

12.139.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VRNDSCALEPH ymm1, ymm2/m256, imm8	E2	IPU	AVX10.2

12.140 VRNDSCALEPS

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.66.0F3A.W0 08 /r /ib VRNDSCALEPS ymm1{k1}{z}, ymm2/m256/m32bcst {sae}, imm8	A	V/V	AVX10.2

12.140.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(w)	MODRM.R/M(r)	IMM8(r)	N/A

12.140.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.140.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Invalid, Precision In abbreviated form, this includes: IE, PE

12.140.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VRNDSCALEPS ymm1, ymm2/m256, imm8	E2	IP	AVX10.2

12.141 VSCALEFPD

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.66.0F38.W1 2C /r VSCALEFPD ymm1{k1}{z}, ymm2, ymm3/m256/m64bcst {er}	A	V/V	AVX10.2

12.141.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(w)	VVVV(r)	MODRM.R/M(r)	N/A

12.141.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.141.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Denormal, Invalid, Overflow, Precision, Underflow In abbreviated form, this includes: DE, IE, OE, PE, UE

12.141.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VSCALEFPD ymm1, ymm2, ymm3/m256	E2	IOUPD	AVX10.2

12.142 VSCALEFPH

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.66.MAP6.W0 2C /r VSCALEFPH ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst {er}	A	V/V	AVX10.2

12.142.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(w)	VVVV(r)	MODRM.R/M(r)	N/A

12.142.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.142.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Denormal, Invalid, Overflow, Precision, Underflow In abbreviated form, this includes: DE, IE, OE, PE, UE

12.142.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VSCALEFPH ymm1, ymm2, ymm3/m256	E2	IOUPD	AVX10.2

12.143 VSCALEFPS

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.66.0F38.W0 2C /r VSCALEFPS ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst {er}	A	V/V	AVX10.2

12.143.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(w)	VVVV(r)	MODRM.R/M(r)	N/A

12.143.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.143.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Denormal, Invalid, Overflow, Precision, Underflow In abbreviated form, this includes: DE, IE, OE, PE, UE

12.143.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VSCALEFPS ymm1, ymm2, ymm3/m256	E2	IOUPD	AVX10.2

12.144 VSQRTPD

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.66.0F.W1 51 /r VSQRTPD ymm1{k1}{z}, ymm2/m256/m64bcst {er}	A	V/V	AVX10.2

12.144.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(w)	MODRM.R/M(r)	N/A	N/A

12.144.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.144.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Denormal, Invalid, Precision In abbreviated form, this includes: DE, IE, PE

12.144.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VSQRTPD ymm1, ymm2/m256	E2	IPD	AVX10.2

12.145 VSQRTPH

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.NP.MAP5.W0 51 /r VSQRTPH ymm1{k1}{z}, ymm2/m256/m16bcst {er}	A	V/V	AVX10.2

12.145.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(w)	MODRM.R/M(r)	N/A	N/A

12.145.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.145.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Denormal, Invalid, Precision In abbreviated form, this includes: DE, IE, PE

12.145.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VSQRTPH ymm1, ymm2/m256	E2	IPD	AVX10.2

12.146 VSQRTPS

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.NP.0F.W0 51 /r VSQRTPS ymm1{k1}{z}, ymm2/m256/m32bcst {er}	A	V/V	AVX10.2

12.146.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(w)	MODRM.R/M(r)	N/A	N/A

12.146.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.146.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Denormal, Invalid, Precision In abbreviated form, this includes: DE, IE, PE

12.146.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VSQRTPS ymm1, ymm2/m256	E2	IPD	AVX10.2

12.147 VSUBPD

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.66.0F.W1 5C /r VSUBPD ymm1{k1}{z}, ymm2, ymm3/m256/m64bcst {er}	A	V/V	AVX10.2

12.147.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(w)	VVVV(r)	MODRM.R/M(r)	N/A

12.147.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.147.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Denormal, Invalid, Overflow, Precision, Underflow In abbreviated form, this includes: DE, IE, OE, PE, UE

12.147.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VSUBPD ymm1, ymm2, ymm3/m256	E2	IOUPD	AVX10.2

12.148 VSUBPH

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.NP.MAP5.W0 5C /r VSUBPH ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst {er}	A	V/V	AVX10.2

12.148.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(w)	VVVV(r)	MODRM.R/M(r)	N/A

12.148.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.148.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Denormal, Invalid, Overflow, Precision, Underflow In abbreviated form, this includes: DE, IE, OE, PE, UE

12.148.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VSUBPH ymm1, ymm2, ymm3/m256	E2	IOUPD	AVX10.2

12.149 VSUBPS

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.256.NP.0F.W0 5C /r VSUBPS ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst {er}	A	V/V	AVX10.2

12.149.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(w)	VVVV(r)	MODRM.R/M(r)	N/A

12.149.2 DESCRIPTION

This instruction's description remains substantially the same as that found in Volume 2A of the Intel® 64 and IA-32 Architectures Software Developer's Manual, except being suitably modified by new Intel® AVX10 functionalities as explained in Section 3.1 of this document.

12.149.3 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Denormal, Invalid, Overflow, Precision, Underflow In abbreviated form, this includes: DE, IE, OE, PE, UE

12.149.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VSUBPS ymm1, ymm2, ymm3/m256	E2	IOUPD	AVX10.2

Chapter 13

INTEL® AVX10 SATURATING CONVERT INSTRUCTIONS

13.1 VCVT[,T]BF162I[,U]BS

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.128.F2.MAP5.W0 69 /r VCVTBF162IBS xmm1{k1}{z}, xmm2/m128/m16bcst	A	V/V	AVX10.2
EVEX.256.F2.MAP5.W0 69 /r VCVTBF162IBS ymm1{k1}{z}, ymm2/m256/m16bcst	A	V/V	AVX10.2
EVEX.512.F2.MAP5.W0 69 /r VCVTBF162IBS zmm1{k1}{z}, zmm2/m512/m16bcst	A	V/V	AVX10.2
EVEX.128.F2.MAP5.W0 6B /r VCVTBF162IUBS xmm1{k1}{z}, xmm2/m128/m16bcst	A	V/V	AVX10.2
EVEX.256.F2.MAP5.W0 6B /r VCVTBF162IUBS ymm1{k1}{z}, ymm2/m256/m16bcst	A	V/V	AVX10.2
EVEX.512.F2.MAP5.W0 6B /r VCVTBF162IUBS zmm1{k1}{z}, zmm2/m512/m16bcst	A	V/V	AVX10.2
EVEX.128.F2.MAP5.W0 68 /r VCVTTBF162IBS xmm1{k1}{z}, xmm2/m128/m16bcst	A	V/V	AVX10.2
EVEX.256.F2.MAP5.W0 68 /r VCVTTBF162IBS ymm1{k1}{z}, ymm2/m256/m16bcst	A	V/V	AVX10.2
EVEX.512.F2.MAP5.W0 68 /r VCVTTBF162IBS zmm1{k1}{z}, zmm2/m512/m16bcst	A	V/V	AVX10.2
EVEX.128.F2.MAP5.W0 6A /r VCVTTBF162IUBS xmm1{k1}{z}, xmm2/m128/m16bcst	A	V/V	AVX10.2
EVEX.256.F2.MAP5.W0 6A /r VCVTTBF162IUBS ymm1{k1}{z}, ymm2/m256/m16bcst	A	V/V	AVX10.2
EVEX.512.F2.MAP5.W0 6A /r VCVTTBF162IUBS zmm1{k1}{z}, zmm2/m512/m16bcst	A	V/V	AVX10.2

13.1.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(w)	MODRM.R/M(r)	N/A	N/A

13.1.2 DESCRIPTION

Note:

For instructions with a CPUID feature flag specifying Intel® AVX10, the programmer must check the available vector options on the processor at run-time via the VL256 and VL512 bits in Converged Vector ISA Leaf 0x24. These bits enumerate whether the corresponding vector lengths are supported (128-bit vectors are always supported) and as such will determine the set of instructions available to the programmer listed in the above opcode table.

These instructions convert eight, sixteen or thirty-two packed brain-float16 values (aka bf16) in the source operand to eight, sixteen or thirty-two signed or un-signed byte integers in the destination operand

The downconverted 8-bit result is written inplace at the lower 8-bit of the corresponding 16-bit element. The upper byte is zeroed.

These instructions does not generate floating point exceptions and does not consult or update MXCSR. Denormal bfloat16 input operands are treated as zeros (DAZ)

VCVTBF162IBS converts brain-float16 floating point elements into signed byte integer elements. When a conversion is inexact, the rounding mode is RNE. If a converted result cannot be represented in the destination format then: If value is too big, then INT_MAX value (2^(w-1)-1, where w represents the number of bits in the destination format) is returned. If value is too small, then INT_MIN value (2^(w-1)) is returned. For NaN, (0) is returned.

VCVTBF162IUBS converts brain-float16 floating point elements into un-signed byte integer elements. When a conversion is inexact, the rounding mode is RNE. If a converted result cannot be represented in the destination format then: If value is too big, the UINT_MAX value (2^w-1, where w represents the number of bits in the destination format) is returned. If value is too small, the UINT_MIN value (0) is returned. For NaN, (0) is returned.

VCVTTBF162IBS converts brain-float16 floating point elements into signed byte integer elements. When a conversion is inexact, a truncated (round toward zero) result is returned. If a converted result cannot be represented in the destination format then: If value is too big, the INT_MAX value (2^(w-1)-1, where w represents the number of bits in the destination format) is returned. If value is too small, the INT_MIN value (2^(w-1)) is returned. For NaN, (0) is returned.

VCVTTBF162IUBS converts brain-float16 floating point elements into un-signed byte integer elements. When a conversion is inexact, a truncated (round toward zero) result is returned. If a converted result cannot be represented in the destination format then: If value is too big, the UINT_MAX value (2^w-1, where w represents the number of bits in the destination format) is returned. If value is too small, the UINT_MIN value (0) is returned. For NaN, (0) is returned.

13.1.3 OPERATION

```
VCVT[,T]BF162[,U]IBS dest {k1}, src
(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF src is memory and (EVEX.b == 1):
            tsrc := src.bf16[0]
        ELSE:
            tsrc := src.bf16[j]
        tmp := 0
        IF *TRUNCATED*:
            IF *dest is signed*:
                tmp.byte[0] := convert_bf16_to_signed_byte_truncate_saturate(tsrc)
            ELSE:
                tmp.byte[0] := convert_bf16_to_unsigned_byte_truncate_saturate(tsrc)
        ELSE:
            IF *dest is signed*:
                tmp.byte[0] := convert_bf16_to_signed_byte_rne_saturate(tsrc)
            ELSE:
                tmp.byte[0] := convert_bf16_to_unsigned_byte_rne_saturate(tsrc)
        dest.word[j] := tmp
    ELSE IF *zeroing*:
        dest.word[j] := 0
    // else dest.word[j] remains unchanged

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

13.1.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VCVTBF162IBS xmm1, xmm2/m128	E4	N/A	AVX10.2
VCVTBF162IBS ymm1, ymm2/m256	E4	N/A	AVX10.2
VCVTBF162IBS zmm1, zmm2/m512	E4	N/A	AVX10.2
VCVTBF162IUBS xmm1, xmm2/m128	E4	N/A	AVX10.2
VCVTBF162IUBS ymm1, ymm2/m256	E4	N/A	AVX10.2
VCVTBF162IUBS zmm1, zmm2/m512	E4	N/A	AVX10.2

Continued on next page...

Instruction	Exception Type	Arithmetic Flags	CPUID
VCVTTBF162IBS xmm1, xmm2/m128	E4	N/A	AVX10.2
VCVTTBF162IBS ymm1, ymm2/m256	E4	N/A	AVX10.2
VCVTTBF162IBS zmm1, zmm2/m512	E4	N/A	AVX10.2
VCVTTBF162IUBS xmm1, xmm2/m128	E4	N/A	AVX10.2
VCVTTBF162IUBS ymm1, ymm2/m256	E4	N/A	AVX10.2
VCVTTBF162IUBS zmm1, zmm2/m512	E4	N/A	AVX10.2

13.2 VCVTTPD2DQS

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.128.NP.MAP5.W1 6D /r VCVTTPD2DQS xmm1{k1}{z}, xmm2/m128/m64bcst	A	V/V	AVX10.2
EVEX.256.NP.MAP5.W1 6D /r VCVTTPD2DQS xmm1{k1}{z}, ymm2/m256/m64bcst {sae}	A	V/V	AVX10.2
EVEX.512.NP.MAP5.W1 6D /r VCVTTPD2DQS ymm1{k1}{z}, zmm2/m512/m64bcst {sae}	A	V/V	AVX10.2

13.2.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(w)	MODRM.R/M(r)	N/A	N/A

13.2.2 DESCRIPTION

Note:

For instructions with a CPUID feature flag specifying Intel® AVX10, the programmer must check the available vector options on the processor at run-time via the VL256 and VL512 bits in Converged Vector ISA Leaf 0x24. These bits enumerate whether the corresponding vector lengths are supported (128-bit vectors are always supported) and as such will determine the set of instructions available to the programmer listed in the above opcode table.

VCVTTPD2DQS: Converts packed double-precision floating-point values in the source operand (the second operand) with truncation and saturation to packed doubleword integers in the destination operand (the first operand). The source operand is a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operand is a YMM/XMM register conditionally updated with writemask k1. When a conversion is inexact, the source value is first truncated (round toward zero) and then checked against the range of the destination type. If the truncated value is representable in the result type that value is returned, otherwise the floating-point invalid exception is raised, and if this exception is masked, the representable value closest to the truncated value is returned. If the source value is NaN, zero is returned. If some rounding mode other than truncation is desired, it can be applied using the VROUNDPD or VRNDSCALEPD instruction before the conversion. Note: EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

13.2.3 OPERATION

```
VCVTTPD2DQS (EVEX encoded versions) when src2 operand is a register
(KL, VL) = (2, 128), (4, 256),(8, 512)
FOR j := 0 TO KL-1
   i := j * 32
   k := j * 64
   IF k1[j] OR *no writemask*
   THEN
      DEST[i+31:i] := convert_DP_to_DW_SignedInteger_TruncateSaturate(SRC[k+63:k])
   ELSE
      IF *merging-masking*    ; merging-masking
      THEN *DEST[i+31:i] remains unchanged*
      ELSE
      DEST[i+31:i] := 0   ; zeroing-masking
      FI
   FI;
ENDFOR
DEST[MAXVL-1:VL/2] := 0
VCVTTPD2DQS (EVEX encoded versions) when src operand is a memory source
(KL, VL) = (2, 128), (4, 256),(8, 512)
FOR j := 0 TO KL-1
   i := j * 32
   k := j * 64
   IF k1[j] OR *no writemask*
   THEN
      IF (EVEX.b = 1)
      THEN
         DEST[i+31:i] := convert_DP_to_DW_SignedInteger_TruncateSaturate(SRC[63:0])
      ELSE
         DEST[i+31:i] := convert_DP_to_DW_SignedInteger_TruncateSaturate(SRC[k+63:k])
      FI;
   ELSE
      IF *merging-masking*    ; merging-masking
         THEN *DEST[i+31:i] remains unchanged*
      ELSE
         DEST[i+31:i] := 0   ; zeroing-masking
      FI
   FI;
ENDFOR
DEST[MAXVL-1:VL/2] := 0
```

13.2.4 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Invalid, Precision In abbreviated form, this includes: IE, PE

13.2.5 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VCVTTPD2DQS xmm1, xmm2/m128	E2	IP	AVX10.2
VCVTTPD2DQS xmm1, ymm2/m256	E2	IP	AVX10.2
VCVTTPD2DQS ymm1, zmm2/m512	E2	IP	AVX10.2

13.3 VCVTTPD2QQS

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.128.66.MAP5.W1 6D /r VCVTTPD2QQS xmm1{k1}{z}, xmm2/m128/m64bcst	A	V/V	AVX10.2
EVEX.256.66.MAP5.W1 6D /r VCVTTPD2QQS ymm1{k1}{z}, ymm2/m256/m64bcst {sae}	A	V/V	AVX10.2
EVEX.512.66.MAP5.W1 6D /r VCVTTPD2QQS zmm1{k1}{z}, zmm2/m512/m64bcst {sae}	A	V/V	AVX10.2

13.3.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(w)	MODRM.R/M(r)	N/A	N/A

13.3.2 DESCRIPTION

Note:

For instructions with a CPUID feature flag specifying Intel® AVX10, the programmer must check the available vector options on the processor at run-time via the VL256 and VL512 bits in Converged Vector ISA Leaf 0x24. These bits enumerate whether the corresponding vector lengths are supported (128-bit vectors are always supported) and as such will determine the set of instructions available to the programmer listed in the above opcode table.

VCVTTPD2QQS: Converts packed double-precision floating-point values in the source operand (second operand) to packed quadword integers in the destination operand (first operand) with truncation and saturation. The source operand is a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1. When a conversion is inexact, the source value is first truncated (round toward zero) and then checked against the range of the destination type. If the truncated value is representable in the result type that value is returned, otherwise the floating-point invalid exception is raised, and if this exception is masked, the representable value closest to the truncated value is returned. If the source value is NaN, zero is returned. If some rounding mode other than truncation is desired, it can be applied using the VROUNDPD or VRNDSCALEPD instruction before the conversion. Note: EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

13.3.3 OPERATION

```
VCVTTPD2QQS (EVEX encoded version) when src operand is a register
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
i := j * 64
IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] := convert_DP_to_QW_SignedInteger_TruncateSaturate(SRC[i+63:i])
ELSE
    IF *merging-masking*     ; merging-masking
        THEN *DEST[i+63:i] remains unchanged*
    ELSE
        DEST[i+63:i] := 0    ; zeroing-masking
    FI
FI;
ENDFOR
DEST[MAXVL-1:VL] := 0

VCVTTPD2QQS (EVEX encoded version) when src operand is a memory source
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
i := j * 64
IF k1[j] OR *no writemask*
    THEN
        IF (EVEX.b == 1)
        THEN
            DEST[i+63:i] := convert_DP_to_QW_SignedInteger_TruncateSaturate(SRC[63:0])
        ELSE
            DEST[i+63:i] := convert_DP_to_QW_SignedInteger_TruncateSaturate(SRC[i+63:i])
        FI;
ELSE
    IF *merging-masking*     ; merging-masking
        THEN *DEST[i+63:i] remains unchanged*
    ELSE
        DEST[i+63:i] := 0    ; zeroing-masking
    FI
FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

13.3.4 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Invalid, Precision In abbreviated form, this includes: IE, PE

13.3.5 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VCVTTPD2QQS xmm1, xmm2/m128	E2	IP	AVX10.2
VCVTTPD2QQS ymm1, ymm2/m256	E2	IP	AVX10.2
VCVTTPD2QQS zmm1, zmm2/m512	E2	IP	AVX10.2

13.4 VCVTTPD2UDQS

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.128.NP.MAP5.W1 6C /r VCVTTPD2UDQS xmm1{k1}{z}, xmm2/m128/m64bcst	A	V/V	AVX10.2
EVEX.256.NP.MAP5.W1 6C /r VCVTTPD2UDQS xmm1{k1}{z}, ymm2/m256/m64bcst {sae}	A	V/V	AVX10.2
EVEX.512.NP.MAP5.W1 6C /r VCVTTPD2UDQS ymm1{k1}{z}, zmm2/m512/m64bcst {sae}	A	V/V	AVX10.2

13.4.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(w)	MODRM.R/M(r)	N/A	N/A

13.4.2 DESCRIPTION

Note:

For instructions with a CPUID feature flag specifying Intel® AVX10, the programmer must check the available vector options on the processor at run-time via the VL256 and VL512 bits in Converged Vector ISA Leaf 0x24. These bits enumerate whether the corresponding vector lengths are supported (128-bit vectors are always supported) and as such will determine the set of instructions available to the programmer listed in the above opcode table.

VCVTTPD2UDQS: Converts packed double-precision floating-point values in the source operand (the second operand) with truncation and saturation to packed unsigned doubleword integers in the destination operand (the first operand). The source operand is a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operand is a YMM/XMM register conditionally updated with writemask k1. When a conversion is inexact, the source value is first truncated (round toward zero) and then checked against the range of the destination type. If the truncated value is representable in the result type that value is returned, otherwise the floating-point invalid exception is raised, and if this exception is masked, the representable value closest to the truncated value is returned. If the source value is NaN, zero is returned. If some rounding mode other than truncation is desired, it can be applied using the VROUNDPD or VRNDSCALEPD instruction before the conversion. Note: EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

13.4.3 OPERATION

```
VCVTTPD2UDQS (EVEX encoded versions) when src2 operand is a register
(KL, VL) = (2, 128), (4, 256),(8, 512)
FOR j := 0 TO KL-1
    i := j * 32
    k := j * 64
    IF k1[j] OR *no writemask*
    THEN
        DEST[i+31:i] := convert_DP_to_DW_UnSignedInteger_TruncateSaturate(SRC[k+63:k])
    ELSE
        IF *merging-masking*     ; merging-masking
        THEN *DEST[i+31:i] remains unchanged*
        ELSE
        DEST[i+31:i] := 0    ; zeroing-masking
        FI
    FI;
ENDFOR
DEST[MAXVL-1:VL/2] := 0

VCVTTPD2UDQS (EVEX encoded versions) when src operand is a memory source
(KL, VL) = (2, 128), (4, 256),(8, 512)
FOR j := 0 TO KL-1
    i := j * 32
    k := j * 64
    IF k1[j] OR *no writemask*
    THEN
        IF (EVEX.b = 1)
        THEN
            DEST[i+31:i] := convert_DP_to_DW_UnSignedInteger_TruncateSaturate(SRC[63:0])
        ELSE
            DEST[i+31:i] := convert_DP_to_DW_UnSignedInteger_TruncateSaturate(SRC[k+63:k])
        FI;
    ELSE
        IF *merging-masking*     ; merging-masking
           THEN *DEST[i+31:i] remains unchanged*
        ELSE
           DEST[i+31:i] := 0    ; zeroing-masking
        FI
    FI;
ENDFOR
DEST[MAXVL-1:VL/2] := 0
```

13.4.4 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Invalid, Precision In abbreviated form, this includes: IE, PE

13.4.5 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VCVTTPD2UDQS xmm1, xmm2/m128	E2	IP	AVX10.2
VCVTTPD2UDQS xmm1, ymm2/m256	E2	IP	AVX10.2
VCVTTPD2UDQS ymm1, zmm2/m512	E2	IP	AVX10.2

13.5 VCVTTPD2UQQS

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.128.66.MAP5.W1 6C /r VCVTTPD2UQQS xmm1{k1}{z}, xmm2/m128/m64bcst	A	V/V	AVX10.2
EVEX.256.66.MAP5.W1 6C /r VCVTTPD2UQQS ymm1{k1}{z}, ymm2/m256/m64bcst {sae}	A	V/V	AVX10.2
EVEX.512.66.MAP5.W1 6C /r VCVTTPD2UQQS zmm1{k1}{z}, zmm2/m512/m64bcst {sae}	A	V/V	AVX10.2

13.5.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(w)	MODRM.R/M(r)	N/A	N/A

13.5.2 DESCRIPTION

Note:

For instructions with a CPUID feature flag specifying Intel® AVX10, the programmer must check the available vector options on the processor at run-time via the VL256 and VL512 bits in Converged Vector ISA Leaf 0x24. These bits enumerate whether the corresponding vector lengths are supported (128-bit vectors are always supported) and as such will determine the set of instructions available to the programmer listed in the above opcode table.

VCVTTPD2UQQS: Converts packed double-precision floating-point values in the source operand (second operand) to packed unsigned quadword integers in the destination operand (first operand) with truncation and saturation. The source operand is a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1. When a conversion is inexact, the source value is first truncated (round toward zero) and then checked against the range of the destination type. If the truncated value is representable in the result type that value is returned, otherwise the floating-point invalid exception is raised, and if this exception is masked, the representable value closest to the truncated value is returned. If the source value is NaN, zero is returned. If some rounding mode other than truncation is desired, it can be applied using the VROUNDPD or VRNDSCALEPD instruction before the conversion. Note: EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

13.5.3 OPERATION

```
VCVTTPD2UQQS (EVEX encoded version) when src operand is a register
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
i := j * 64
IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] := convert_DP_to_QW_UnSignedInteger_TruncateSaturate(SRC[i+63:i])
ELSE
    IF *merging-masking*     ; merging-masking
        THEN *DEST[i+63:i] remains unchanged*
    ELSE
        DEST[i+63:i] := 0    ; zeroing-masking
    FI
FI;
ENDFOR
DEST[MAXVL-1:VL] := 0

VCVTTPD2UQQS (EVEX encoded version) when src operand is a memory source
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
i := j * 64
IF k1[j] OR *no writemask*
    THEN
        IF (EVEX.b == 1)
        THEN
            DEST[i+63:i] :=
        ELSE    convert_DP_to_QW_UnSignedInteger_TruncateSaturate(SRC[63:0])
            DEST[i+63:i] := convert_DP_to_QW_UnSignedInteger_TruncateSaturate(SRC[i+63:i])
        FI;
ELSE
    IF *merging-masking*     ; merging-masking
        THEN *DEST[i+63:i] remains unchanged*
    ELSE
        DEST[i+63:i] := 0    ; zeroing-masking
    FI
FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

13.5.4 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Invalid, Precision In abbreviated form, this includes: IE, PE

13.5.5 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VCVTTPD2UQQS xmm1, xmm2/m128	E2	IP	AVX10.2
VCVTTPD2UQQS ymm1, ymm2/m256	E2	IP	AVX10.2
VCVTTPD2UQQS zmm1, zmm2/m512	E2	IP	AVX10.2

13.6 VCVT[,T]PH2I[,U]BS

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.128.NP.MAP5.W0 69 /r VCVTPH2IBS xmm1{k1}{z}, xmm2/m128/m16bcst	A	V/V	AVX10.2
EVEX.256.NP.MAP5.W0 69 /r VCVTPH2IBS ymm1{k1}{z}, ymm2/m256/m16bcst {er}	A	V/V	AVX10.2
EVEX.512.NP.MAP5.W0 69 /r VCVTPH2IBS zmm1{k1}{z}, zmm2/m512/m16bcst {er}	A	V/V	AVX10.2
EVEX.128.NP.MAP5.W0 6B /r VCVTPH2IUBS xmm1{k1}{z}, xmm2/m128/m16bcst	A	V/V	AVX10.2
EVEX.256.NP.MAP5.W0 6B /r VCVTPH2IUBS ymm1{k1}{z}, ymm2/m256/m16bcst {er}	A	V/V	AVX10.2
EVEX.512.NP.MAP5.W0 6B /r VCVTPH2IUBS zmm1{k1}{z}, zmm2/m512/m16bcst {er}	A	V/V	AVX10.2
EVEX.128.NP.MAP5.W0 68 /r VCVTTPH2IBS xmm1{k1}{z}, xmm2/m128/m16bcst	A	V/V	AVX10.2
EVEX.256.NP.MAP5.W0 68 /r VCVTTPH2IBS ymm1{k1}{z}, ymm2/m256/m16bcst {sae}	A	V/V	AVX10.2
EVEX.512.NP.MAP5.W0 68 /r VCVTTPH2IBS zmm1{k1}{z}, zmm2/m512/m16bcst {sae}	A	V/V	AVX10.2
EVEX.128.NP.MAP5.W0 6A /r VCVTTPH2IUBS xmm1{k1}{z}, xmm2/m128/m16bcst	A	V/V	AVX10.2
EVEX.256.NP.MAP5.W0 6A /r VCVTTPH2IUBS ymm1{k1}{z}, ymm2/m256/m16bcst {sae}	A	V/V	AVX10.2
EVEX.512.NP.MAP5.W0 6A /r VCVTTPH2IUBS zmm1{k1}{z}, zmm2/m512/m16bcst {sae}	A	V/V	AVX10.2

13.6.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(w)	MODRM.R/M(r)	N/A	N/A

13.6.2 DESCRIPTION

Note:

For instructions with a CPUID feature flag specifying Intel® AVX10, the programmer must check the available vector options on the processor at run-time via the VL256 and VL512 bits in Converged Vector ISA Leaf 0x24. These bits enumerate whether the corresponding vector lengths are supported (128-bit vectors are always supported) and as such will determine the set of instructions available to the programmer listed in the above opcode table.

These instructions convert eight, sixteen or thirty-two packed half-precision floating-point values (aka fp16) in the source operand to eight, sixteen or thirty-two signed or un-signed byte integers in the destination operand

The downconverted 8-bit result is written inplace at the lower 8-bit of the corresponding 16-bit element. The upper byte is zeroed.

VCVTPH2IBS converts half-precision floating point elements into signed byte integer elements. When a conversion is inexact, floating-point precision exception is raised and the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked then: If value is too big, the INT_MAX value (2^(w-1)-1, where w represents the number of bits in the destination format) is returned. If value is too small, the INT_MIN value (2^(w-1)) is returned. For NaN, (0) is returned.

VCVTPH2IUBS converts half-precision floating point elements into un-signed byte integer elements. When a conversion is inexact, floating-point precision exception is raised and the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked then: If value is too big, the UINT_MAX value (2^w-1, where w represents the number of bits in the destination format) is returned. If value is too small, the UINT_MIN value (0) is returned. For NaN, (0) is returned.

VCVTTPH2IBS converts half-precision floating point elements into signed byte integer elements. When a conversion is inexact, floating-point precision exception is raised and a truncated (round toward zero) result is returned. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked then: If value is too big, the INT_MAX value (2^(w-1)-1, where w represents the number of bits in the destination format) is returned. If value is too small, the INT_MIN value (2^(w-1)) is returned. For NaN, (0) is returned.

VCVTTPH2IUBS converts half-precision floating point elements into un-signed byte integer elements. When a conversion is inexact, floating-point precision exception is raised and a truncated (round toward zero) result is returned. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked then: If value is too big, the UINT_MAX value (2^w-1, where w represents the number of bits in the destination format) is returned. If value is too small, the UINT_MIN value (0) is returned. For NaN, (0) is returned.

13.6.3 OPERATION

```
VCVT[,T]PH2I[,U]BS dest {k1}, src
(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF src is memory and (EVEX.b == 1):
            tsrc := src.fp16[0]
        ELSE:
            tsrc := src.fp16[j]
        tmp := 0
        IF *TRUNCATED*:
            IF *dest is signed*:
                tmp.byte[0] := convert_fp16_to_signed_byte_truncate_saturate(tsrc)
            ELSE:
                tmp.byte[0] := convert_fp16_to_unsigned_byte_truncate_saturate(tsrc)
        ELSE:
            IF *dest is signed*:
                tmp.byte[0] := convert_fp16_to_signed_byte_saturate(tsrc)
            ELSE:
                tmp.byte[0] := convert_fp16_to_unsigned_byte_saturate(tsrc)
        dest.word[j] := tmp
    ELSE IF *zeroing*:
        dest.word[j] := 0
    // else dest.word[j] remains unchanged

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

13.6.4 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Invalid, Precision In abbreviated form, this includes: IE, PE

13.6.5 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VCVTPH2IBS xmm1, xmm2/m128	E2	IP	AVX10.2
VCVTPH2IBS ymm1, ymm2/m256	E2	IP	AVX10.2

Continued on next page...

Instruction	Exception Type	Arithmetic Flags	CPUID
VCVTPH2IBS zmm1, zmm2/m512	E2	IP	AVX10.2
VCVTPH2IUBS xmm1, xmm2/m128	E2	IP	AVX10.2
VCVTPH2IUBS ymm1, ymm2/m256	E2	IP	AVX10.2
VCVTPH2IUBS zmm1, zmm2/m512	E2	IP	AVX10.2
VCVTTPH2IBS xmm1, xmm2/m128	E2	IP	AVX10.2
VCVTTPH2IBS ymm1, ymm2/m256	E2	IP	AVX10.2
VCVTTPH2IBS zmm1, zmm2/m512	E2	IP	AVX10.2
VCVTTPH2IUBS xmm1, xmm2/m128	E2	IP	AVX10.2
VCVTTPH2IUBS ymm1, ymm2/m256	E2	IP	AVX10.2
VCVTTPH2IUBS zmm1, zmm2/m512	E2	IP	AVX10.2

13.7 VCVTTPS2DQS

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.128.NP.MAP5.W0 6D /r VCVTTPS2DQS xmm1{k1}{z}, xmm2/m128/m32bcst	A	V/V	AVX10.2
EVEX.256.NP.MAP5.W0 6D /r VCVTTPS2DQS ymm1{k1}{z}, ymm2/m256/m32bcst {sae}	A	V/V	AVX10.2
EVEX.512.NP.MAP5.W0 6D /r VCVTTPS2DQS zmm1{k1}{z}, zmm2/m512/m32bcst {sae}	A	V/V	AVX10.2

13.7.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(w)	MODRM.R/M(r)	N/A	N/A

13.7.2 DESCRIPTION

Note:

For instructions with a CPUID feature flag specifying Intel® AVX10, the programmer must check the available vector options on the processor at run-time via the VL256 and VL512 bits in Converged Vector ISA Leaf 0x24. These bits enumerate whether the corresponding vector lengths are supported (128-bit vectors are always supported) and as such will determine the set of instructions available to the programmer listed in the above opcode table.

VCVTTPS2DQS: Converts packed single-precision floating-point values in the source operand to doubleword integers in the destination operand with truncation and saturation. The source operand is a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1. When a conversion is inexact, the source value is first truncated (round toward zero) and then checked against the range of the destination type. If the truncated value is representable in the result type that value is returned, otherwise the floating-point invalid exception is raised, and if this exception is masked, the representable value closest to the truncated value is returned. If the source value is NaN, zero is returned. If some rounding mode other than truncation is desired, it can be applied using the VROUNDPS or VRNDSCALEPS instruction before the conversion. Note: EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD

13.7.3 OPERATION

```
VCVTTPS2DQS (EVEX encoded versions) when src2 operand is a register
(KL, VL) = (2, 128), (4, 256),(8, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
    THEN
        DEST[i+31:i] := convert_SP_to_DW_SignedInteger_TruncateSaturate(SRC[k+31:k])
    ELSE
        IF *merging-masking*     ; merging-masking
        THEN *DEST[i+31:i] remains unchanged*
        ELSE
        DEST[i+31:i] := 0    ; zeroing-masking
        FI
    FI;
ENDFOR
DEST[MAXVL-1:VL/2] := 0

VCVTTPS2DQS (EVEX encoded versions) when src operand is a memory source
(KL, VL) = (2, 128), (4, 256),(8, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
    THEN
        IF (EVEX.b = 1)
        THEN
            DEST[i+31:i] := convert_SP_to_DW_SignedInteger_TruncateSaturate(SRC[31:0])
        ELSE
            DEST[i+31:i] := convert_SP_to_DW_SignedInteger_TruncateSaturate(SRC[k+31:k])
        FI;
    ELSE
        IF *merging-masking*     ; merging-masking
           THEN *DEST[i+31:i] remains unchanged*
        ELSE
           DEST[i+31:i] := 0    ; zeroing-masking
        FI
    FI;
ENDFOR
DEST[MAXVL-1:VL/2] := 0
```

13.7.4 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Invalid, Precision In abbreviated form, this includes: IE, PE

13.7.5 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VCVTTPS2DQS xmm1, xmm2/m128	E2	IP	AVX10.2
VCVTTPS2DQS ymm1, ymm2/m256	E2	IP	AVX10.2
VCVTTPS2DQS zmm1, zmm2/m512	E2	IP	AVX10.2

13.8 VCVT[,T]PS2I[,U]BS

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.128.66.MAP5.W0 69 /r VCVTPS2IBS xmm1{k1}{z}, xmm2/m128/m32bcst	A	V/V	AVX10.2
EVEX.256.66.MAP5.W0 69 /r VCVTPS2IBS ymm1{k1}{z}, ymm2/m256/m32bcst {er}	A	V/V	AVX10.2
EVEX.512.66.MAP5.W0 69 /r VCVTPS2IBS zmm1{k1}{z}, zmm2/m512/m32bcst {er}	A	V/V	AVX10.2
EVEX.128.66.MAP5.W0 6B /r VCVTPS2IUBS xmm1{k1}{z}, xmm2/m128/m32bcst	A	V/V	AVX10.2
EVEX.256.66.MAP5.W0 6B /r VCVTPS2IUBS ymm1{k1}{z}, ymm2/m256/m32bcst {er}	A	V/V	AVX10.2
EVEX.512.66.MAP5.W0 6B /r VCVTPS2IUBS zmm1{k1}{z}, zmm2/m512/m32bcst {er}	A	V/V	AVX10.2
EVEX.128.66.MAP5.W0 68 /r VCVTTPS2IBS xmm1{k1}{z}, xmm2/m128/m32bcst	A	V/V	AVX10.2
EVEX.256.66.MAP5.W0 68 /r VCVTTPS2IBS ymm1{k1}{z}, ymm2/m256/m32bcst {sae}	A	V/V	AVX10.2
EVEX.512.66.MAP5.W0 68 /r VCVTTPS2IBS zmm1{k1}{z}, zmm2/m512/m32bcst {sae}	A	V/V	AVX10.2
EVEX.128.66.MAP5.W0 6A /r VCVTTPS2IUBS xmm1{k1}{z}, xmm2/m128/m32bcst	A	V/V	AVX10.2
EVEX.256.66.MAP5.W0 6A /r VCVTTPS2IUBS ymm1{k1}{z}, ymm2/m256/m32bcst {sae}	A	V/V	AVX10.2
EVEX.512.66.MAP5.W0 6A /r VCVTTPS2IUBS zmm1{k1}{z}, zmm2/m512/m32bcst {sae}	A	V/V	AVX10.2

13.8.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(w)	MODRM.R/M(r)	N/A	N/A

13.8.2 DESCRIPTION

Note:

For instructions with a CPUID feature flag specifying Intel® AVX10, the programmer must check the available vector options on the processor at run-time via the VL256 and VL512 bits in Converged Vector ISA Leaf 0x24. These bits enumerate whether the corresponding vector lengths are supported (128-bit vectors are always supported) and as such will determine the set of instructions available to the programmer listed in the above opcode table.

These instructions convert four, eight or sixteen packed single-precision floating-point values in the source operand to four, eight or sixteen signed or unsigned byte integers in the destination operand.

The downconverted 8-bit result is written inplace at the lower 8-bit of the corresponding 32-bit element. The upper 3 bytes are zeroed.

VCVTPS2IBS converts single-precision floating point elements into signed byte integer elements. When a conversion is inexact, floating-point precision exception is raised and the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked then: If value is too big, the INT_MAX value (2^(w-1)-1, where w represents the number of bits in the destination format) is returned. If value is too small, the INT_MIN value (2^(w-1)) is returned. For NaN, (0) is returned.

VCVTPS2IUBS converts single-precision floating point elements into un-signed byte integer elements. When a conversion is inexact, floating-point precision exception is raised and the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked then: If value is too big, the UINT_MAX value (2^w-1, where w represents the number of bits in the destination format) is returned. If value is too small, the UINT_MIN value (0) is returned. For NaN, (0) is returned.

VCVTTPS2IBS converts single-precision floating point elements into signed byte integer elements. When a conversion is inexact, floating-point precision exception is raised and a truncated (round toward zero) result is returned. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked then: If value is too big, the INT_MAX value (2^(w-1)-1, where w represents the number of bits in the destination format) is returned. If value is too small, the INT_MIN value (2^(w-1)) is returned. For NaN, (0) is returned.

VCVTTPS2IUBS converts single-precision floating point elements into un-signed byte integer elements. When a conversion is inexact, floating-point precision exception is raised and a truncated (round toward zero) result is returned. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked then: If value is too big, the UINT_MAX value (2^w-1, where w represents the number of bits in the destination format) is returned. If value is too small, the UINT_MIN value (0) is returned. For NaN, (0) is returned.

13.8.3 OPERATION

```
VCVT[,T]PS2I[,U]BS dest {k1}, src
(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF src is memory and (EVEX.b == 1):
            tsrc := src.fp32[0]
        ELSE:
            tsrc := src.fp32[j]
        tmp := 0
        IF *TRUNCATED*:
            IF *dest is signed*:
                tmp.byte[0] := convert_fp32_to_signed_byte_truncate_saturate(tsrc)
            ELSE:
                tmp.byte[0] := convert_fp32_to_unsigned_byte_truncate_saturate(tsrc)
        ELSE:
            IF *dest is signed*:
                tmp.byte[0] := convert_fp32_to_signed_byte_saturate(tsrc)
            ELSE:
                tmp.byte[0] := convert_fp32_to_unsigned_byte_saturate(tsrc)
        dest.dword[j] := tmp
    ELSE IF *zeroing*:
        dest.dword[j] := 0
    // else dest.dword[j] remains unchanged

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

13.8.4 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Invalid, Precision In abbreviated form, this includes: IE, PE

13.8.5 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VCVTPS2IBS xmm1, xmm2/m128	E2	IP	AVX10.2
VCVTPS2IBS ymm1, ymm2/m256	E2	IP	AVX10.2

Continued on next page...

Instruction	Exception Type	Arithmetic Flags	CPUID
VCVTPS2IBS zmm1, zmm2/m512	E2	IP	AVX10.2
VCVTPS2IUBS xmm1, xmm2/m128	E2	IP	AVX10.2
VCVTPS2IUBS ymm1, ymm2/m256	E2	IP	AVX10.2
VCVTPS2IUBS zmm1, zmm2/m512	E2	IP	AVX10.2
VCVTTPS2IBS xmm1, xmm2/m128	E2	IP	AVX10.2
VCVTTPS2IBS ymm1, ymm2/m256	E2	IP	AVX10.2
VCVTTPS2IBS zmm1, zmm2/m512	E2	IP	AVX10.2
VCVTTPS2IUBS xmm1, xmm2/m128	E2	IP	AVX10.2
VCVTTPS2IUBS ymm1, ymm2/m256	E2	IP	AVX10.2
VCVTTPS2IUBS zmm1, zmm2/m512	E2	IP	AVX10.2

13.9 VCVTTPS2QQS

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.128.66.MAP5.W0 6D /r VCVTTPS2QQS xmm1{k1}{z}, xmm2/m64/m32bcst	A	V/V	AVX10.2
EVEX.256.66.MAP5.W0 6D /r VCVTTPS2QQS ymm1{k1}{z}, xmm2/m128/m32bcst {sae}	A	V/V	AVX10.2
EVEX.512.66.MAP5.W0 6D /r VCVTTPS2QQS zmm1{k1}{z}, ymm2/m256/m32bcst {sae}	A	V/V	AVX10.2

13.9.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	HALF	MODRM.REG(w)	MODRM.R/M(r)	N/A	N/A

13.9.2 DESCRIPTION

Note:

For instructions with a CPUID feature flag specifying Intel® AVX10, the programmer must check the available vector options on the processor at run-time via the VL256 and VL512 bits in Converged Vector ISA Leaf 0x24. These bits enumerate whether the corresponding vector lengths are supported (128-bit vectors are always supported) and as such will determine the set of instructions available to the programmer listed in the above opcode table.

VCVTTPS2QQS: Converts packed single-precision floating-point values in the source operand to quadword integers in the destination operand with truncation and saturation. The source operand is a YMM/XMM register or a 256/128/64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1. When a conversion is inexact, the source value is first truncated (round toward zero) and then checked against the range of the destination type. If the truncated value is representable in the result type that value is returned, otherwise the floating-point invalid exception is raised, and if this exception is masked, the representable value closest to the truncated value is returned. If the source value is NaN, zero is returned. If some rounding mode other than truncation is desired, it can be applied using the VROUNDPS or VRNDSCALEPS instruction before the conversion. Note: EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

13.9.3 OPERATION

```
VCVTTPS2QQS (EVEX encoded versions) when src operand is a register
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    k := j * 32
    IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] := convert_SP_to_QW_SignedInteger_TruncateSaturate(SRC[k+31:k])
    ELSE
        IF *merging-masking*     ; merging-masking
        THEN *DEST[i+63:i] remains unchanged*
        ELSE
            DEST[i+63:i] := 0    ; zeroing-masking
        FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0

VCVTTPS2QQS (EVEX encoded versions) when src operand is a memory source
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    k := j * 32
    IF k1[j] OR *no writemask*
    THEN
        IF (EVEX.b == 1)
        THEN DEST[i+63:i] := convert_SP_to_QW_SignedInteger_TruncateSaturate(SRC[31:0])
        ELSE
            DEST[i+63:i] := convert_SP_to_QW_SignedInteger_TruncateSaturate(SRC[k+31:k])
        FI;
    ELSE
        IF *merging-masking*     ; merging-masking
           THEN *DEST[i+63:i] remains unchanged*
        ELSE
           DEST[i+63:i] := 0    ; zeroing-masking
        FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

13.9.4 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Invalid, Precision In abbreviated form, this includes: IE, PE

13.9.5 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VCVTTPS2QQS xmm1, xmm2/m64	E2	IP	AVX10.2
VCVTTPS2QQS ymm1, xmm2/m128	E2	IP	AVX10.2
VCVTTPS2QQS zmm1, ymm2/m256	E2	IP	AVX10.2

13.10 VCVTTPS2UDQS

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.128.NP.MAP5.W0 6C /r VCVTTPS2UDQS xmm1{k1}{z}, xmm2/m128/m32bcst	A	V/V	AVX10.2
EVEX.256.NP.MAP5.W0 6C /r VCVTTPS2UDQS ymm1{k1}{z}, ymm2/m256/m32bcst {sae}	A	V/V	AVX10.2
EVEX.512.NP.MAP5.W0 6C /r VCVTTPS2UDQS zmm1{k1}{z}, zmm2/m512/m32bcst {sae}	A	V/V	AVX10.2

13.10.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULL	MODRM.REG(w)	MODRM.R/M(r)	N/A	N/A

13.10.2 DESCRIPTION

Note:

For instructions with a CPUID feature flag specifying Intel® AVX10, the programmer must check the available vector options on the processor at run-time via the VL256 and VL512 bits in Converged Vector ISA Leaf 0x24. These bits enumerate whether the corresponding vector lengths are supported (128-bit vectors are always supported) and as such will determine the set of instructions available to the programmer listed in the above opcode table.

VCVTTPS2UDQS: Converts packed single-precision floating-point values in the source operand to unsigned doubleword integers in the destination operand with truncation and saturation. The source operand is a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1. When a conversion is inexact, the source value is first truncated (round toward zero) and then checked against the range of the destination type. If the truncated value is representable in the result type that value is returned, otherwise the floating-point invalid exception is raised, and if this exception is masked, the representable value closest to the truncated value is returned. If the source value is NaN, zero is returned. If some rounding mode other than truncation is desired, it can be applied using the VROUNDPS or VRNDSCALEPS instruction before the conversion. Note: EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD

13.10.3 OPERATION

```
VCVTTPS2UDQS (EVEX encoded versions) when src2 operand is a register
(KL, VL) = (2, 128), (4, 256),(8, 512)
S := 31
FOR j := 0 TO KL-1
   i := j * 32
   IF k1[j] OR *no writemask*
   THEN
      DST[i+S:i]:=Convert_Single_Precision_Floating_Point_To_Uinteger_Saturate(SRC[k+S:k])
   ELSE
      IF *merging-masking*     ; merging-masking
      THEN *DST[i+S:i] remains unchanged*
      ELSE
      DST[i+S:i] := 0    ; zeroing-masking
      FI
   FI;
ENDFOR
DST[MAXVL-1:VL/2] := 0

VCVTTPS2UDQS (EVEX encoded versions) when src operand is a memory source
(KL, VL) = (2, 128), (4, 256),(8, 512)
S := 31
FOR j := 0 TO KL-1
   i := j * 32
   IF k1[j] OR *no writemask*:
    IF (EVEX.b = 1):
      DST[i+S:i]:=Convert_Single_Precision_Floating_Point_To_Uinteger_Saturate(SRC[S:0])
    ELSE:
      DST[i+S:i]:=Convert_Single_Precision_Floating_Point_To_Uinteger_Saturate(SRC[k+S:k])
    FI;
   ELSE
    IF *merging-masking*     ; merging-masking
       THEN *DST[i+S:i] remains unchanged*
    ELSE
       DST[i+S:i] := 0    ; zeroing-masking
    FI
   FI;
ENDFOR
DST[MAXVL-1:VL/2] := 0
```

13.10.4 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Invalid, Precision In abbreviated form, this includes: IE, PE

13.10.5 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VCVTTPS2UDQS xmm1, xmm2/m128	E2	IP	AVX10.2
VCVTTPS2UDQS ymm1, ymm2/m256	E2	IP	AVX10.2
VCVTTPS2UDQS zmm1, zmm2/m512	E2	IP	AVX10.2

13.11 VCVTTPS2UQQS

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.128.66.MAP5.W0 6C /r VCVTTPS2UQQS xmm1{k1}{z}, xmm2/m64/m32bcst	A	V/V	AVX10.2
EVEX.256.66.MAP5.W0 6C /r VCVTTPS2UQQS ymm1{k1}{z}, xmm2/m128/m32bcst {sae}	A	V/V	AVX10.2
EVEX.512.66.MAP5.W0 6C /r VCVTTPS2UQQS zmm1{k1}{z}, ymm2/m256/m32bcst {sae}	A	V/V	AVX10.2

13.11.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	HALF	MODRM.REG(w)	MODRM.R/M(r)	N/A	N/A

13.11.2 DESCRIPTION

Note:

For instructions with a CPUID feature flag specifying Intel® AVX10, the programmer must check the available vector options on the processor at run-time via the VL256 and VL512 bits in Converged Vector ISA Leaf 0x24. These bits enumerate whether the corresponding vector lengths are supported (128-bit vectors are always supported) and as such will determine the set of instructions available to the programmer listed in the above opcode table.

VCVTTPS2UQQS: Converts packed single-precision floating-point values in the source operand to unsigned quadword integers in the destination operand with truncation and saturation. The source operand is a YMM/XMM register or a 256/128/64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1. When a conversion is inexact, the source value is first truncated (round toward zero) and then checked against the range of the destination type. If the truncated value is representable in the result type that value is returned, otherwise the floating-point invalid exception is raised, and if this exception is masked, the representable value closest to the truncated value is returned. If the source value is NaN, zero is returned. If some rounding mode other than truncation is desired, it can be applied using the VROUNDPS or VRNDSCALEPS instruction before the conversion. Note: EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

13.11.3 OPERATION

```
VCVTTPS2UQQS (EVEX encoded versions) when src operand is a register
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    k := j * 32
    IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] := convert_SP_to_QW_UnSignedInteger_TruncateSaturate(SRC[k+31:k])
    ELSE
        IF *merging-masking*     ; merging-masking
        THEN *DEST[i+63:i] remains unchanged*
        ELSE
            DEST[i+63:i] := 0    ; zeroing-masking
        FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0

VCVTTPS2UQQS (EVEX encoded versions) when src operand is a memory source
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    k := j * 32
    IF k1[j] OR *no writemask*
    THEN
        IF (EVEX.b == 1)
        THEN
            DEST[i+63:i] := convert_SP_to_QW_UnSignedInteger_TruncateSaturate(SRC[31:0])
        ELSE
            DEST[i+63:i] := convert_SP_to_QW_UnSignedInteger_TruncateSaturate(SRC[k+31:k])
        FI;
    ELSE
        IF *merging-masking*     ; merging-masking
           THEN *DEST[i+63:i] remains unchanged*
        ELSE
           DEST[i+63:i] := 0    ; zeroing-masking
        FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

13.11.4 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Invalid, Precision In abbreviated form, this includes: IE, PE

13.11.5 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VCVTTPS2UQQS xmm1, xmm2/m64	E2	IP	AVX10.2
VCVTTPS2UQQS ymm1, xmm2/m128	E2	IP	AVX10.2
VCVTTPS2UQQS zmm1, ymm2/m256	E2	IP	AVX10.2

13.12 VCVTTSD2SIS

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.LLIG.F2.MAP5.W0 6D /r VCVTTSD2SIS r32, xmm1/m64 {sae}	A	V/V[1]	AVX10.2
EVEX.LLIG.F2.MAP5.W1 6D /r VCVTTSD2SIS r64, xmm1/m64 {sae}	A	V/N.E.	AVX10.2
Notes: 1: Outside of 64b mode, the EVEX.W field is ignored. The instruction behaves as if W=0 were used.			

13.12.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	SCALAR	MODRM.REG(w)	MODRM.R/M(r)	N/A	N/A

13.12.2 DESCRIPTION

Note:

For SCALAR instructions with a CPUID feature flag specifying Intel® AVX10, the programmer can skip the check for the available vector options on the processor at run-time via the VL256 and VL512 bits in Converged Vector ISA Leaf 0x24. These bits enumerate whether the corresponding vector lengths are supported and usually determine the set of instructions available to the programmer. However, scalar instructions are architected without the notion of vector length, so this step does not strictly apply.

VCVTTSD2SIS: Converts a double-precision floating-point value in the source operand (the second operand) to a doubleword integer (or quadword integer if operand size is 64 bits) in the destination operand (the first operand) with truncation and saturation. The source operand can be an XMM register or a 64-bit memory location. The destination operand is a general-purpose register. When the source operand is an XMM register, the double-precision floating-point value is contained in the low quadword of the register. When a conversion is inexact, the source value is first truncated (round toward zero) and then checked against the range of the destination type. If the truncated value is representable in the result type that value is returned, otherwise the floating-point invalid exception is raised, and if this exception is masked, the representable value closest to the truncated value is returned. If the source value is NaN, zero is returned. If some rounding mode other than truncation is desired, it can be applied using the VROUNDSD or VRNDSCALESD instruction before the conversion. EVEX.W1 version: promotes the instruction to produce 64-bit data in 64-bit mode.

13.12.3 OPERATION

```
VCVTTSD2SIS (EVEX encoded version)
IF 64-bit Mode and OperandSize = 64
THEN
    DEST[63:0] := convert_DP_to_QW_SignedInteger_TruncateSaturate(SRC[63:0]);
ELSE
    DEST[31:0] := convert_DP_to_DW_SignedInteger_TruncateSaturate(SRC[63:0]);
FI;

SIMD Floating-Point Exceptions
-Invalid, Precision

Other Exceptions:
-EVEX-encoded instructions, see Table 2-48, "Type E3NF Class Exception
Conditions".

NOTES:

1. For this specific instruction, EVEX.W in non-64 bit is ignored; the
instructions behaves as if the W0 version is used.
```

13.12.4 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Invalid, Precision In abbreviated form, this includes: IE, PE

13.12.5 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VCVTTSD2SIS r32, xmm1/m64	E3NF	IP	AVX10.2
VCVTTSD2SIS r64, xmm1/m64	E3NF	IP	AVX10.2

13.13 VCVTTSD2USIS

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.LLIG.F2.MAP5.W0 6C /r VCVTTSD2USIS r32, xmm1/m64 {sae}	A	V/V[1]	AVX10.2
EVEX.LLIG.F2.MAP5.W1 6C /r VCVTTSD2USIS r64, xmm1/m64 {sae}	A	V/N.E.	AVX10.2
Notes: 1: Outside of 64b mode, the EVEX.W field is ignored. The instruction behaves as if W=0 were used.			

13.13.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	SCALAR	MODRM.REG(w)	MODRM.R/M(r)	N/A	N/A

13.13.2 DESCRIPTION

Note:

For SCALAR instructions with a CPUID feature flag specifying Intel® AVX10, the programmer can skip the check for the available vector options on the processor at run-time via the VL256 and VL512 bits in Converged Vector ISA Leaf 0x24. These bits enumerate whether the corresponding vector lengths are supported and usually determine the set of instructions available to the programmer. However, scalar instructions are architected without the notion of vector length, so this step does not strictly apply.

VCVTTSD2USIS: Converts a double-precision floating-point value in the source operand (the second operand) to an unsigned doubleword integer (or an unsigned quadword integer if operand size is 64 bits) in the destination operand (the first operand) with truncation and saturation. The source operand can be an XMM register or a 64-bit memory location. The destination operand is a general-purpose register. When the source operand is an XMM register, the double-precision floating-point value is contained in the low quadword of the register. When a conversion is inexact, the source value is first truncated (round toward zero) and then checked against the range of the destination type. If the truncated value is representable in the result type that value is returned, otherwise the floating-point invalid exception is raised, and if this exception is masked, the representable value closest to the truncated value is returned. If the source value is NaN, zero is returned. If some rounding mode other than truncation is desired, it can be applied using the VROUNDSD or VRNDSCALESD instruction before the conversion. EVEX.W1 version: promotes the instruction to produce 64-bit data in 64-bit mode.

13.13.3 OPERATION

```
VCVTTSD2USIS (EVEX encoded version)
IF 64-bit Mode and OperandSize = 64
THEN
    DEST[63:0] := convert_DP_to_QW_UnSignedInteger_TruncateSaturate(SRC[63:0]);
ELSE
    DEST[31:0] := convert_DP_to_DW_SignedInteger_TruncateSaturate(SRC[63:0]);
FI;

SIMD Floating-Point Exceptions
-Invalid, Precision
Other Exceptions
-EVEX-encoded instructions, see Table 2-48, "Type E3NF Class Exception Conditions".

NOTES:
1. For this specific instruction, EVEX.W in non-64 bit is ignored; the
instructions behaves as if the W0 version is used.

```

13.13.4 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Invalid, Precision In abbreviated form, this includes: IE, PE

13.13.5 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VCVTTSD2USIS r32, xmm1/m64	E3NF	IP	AVX10.2
VCVTTSD2USIS r64, xmm1/m64	E3NF	IP	AVX10.2

13.14 VCVTTSS2SIS

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.LLIG.F3.MAP5.W0 6D /r VCVTTSS2SIS r32, xmm1/m32 {sae}	A	V/V[1]	AVX10.2
EVEX.LLIG.F3.MAP5.W1 6D /r VCVTTSS2SIS r64, xmm1/m32 {sae}	A	V/N.E.	AVX10.2
Notes: 1: Outside of 64b mode, the EVEX.W field is ignored. The instruction behaves as if W=0 were used.			

13.14.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	SCALAR	MODRM.REG(w)	MODRM.R/M(r)	N/A	N/A

13.14.2 DESCRIPTION

Note:

For SCALAR instructions with a CPUID feature flag specifying Intel® AVX10, the programmer can skip the check for the available vector options on the processor at run-time via the VL256 and VL512 bits in Converged Vector ISA Leaf 0x24. These bits enumerate whether the corresponding vector lengths are supported and usually determine the set of instructions available to the programmer. However, scalar instructions are architected without the notion of vector length, so this step does not strictly apply.

VCVTTSS2SIS: Converts a single-precision floating-point value in the source operand (the second operand) to a doubleword integer (or quadword integer if operand size is 64 bits) in the destination operand (the first operand) with truncation and saturation. The source operand can be an XMM register or a memory location. The destination operand is a general-purpose register. When the source operand is an XMM register, the single-precision floating-point value is contained in the low doubleword of the register. When a conversion is inexact, the source value is first truncated (round toward zero) and then checked against the range of the destination type. If the truncated value is representable in the result type that value is returned, otherwise the floating-point invalid exception is raised, and if this exception is masked, the representable value closest to the truncated value is returned. If the source value is NaN, zero is returned. If some rounding mode other than truncation is desired, it can be applied using the VROUNDSS or VRNDSCALESS instruction before the conversion. EVEX.W1 version: promotes the instruction to produce 64-bit data in 64-bit mode. Note: EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

13.14.3 OPERATION

```
VCVTTSS2SIS (EVEX encoded version)
IF 64-bit Mode and OperandSize = 64
THEN
   DEST[63:0] := convert_SP_to_QW_SignedInteger_TruncateSaturate(SRC[31:0]);
ELSE
   DEST[31:0] := convert_SP_to_DW_SignedInteger_TruncateSaturate(SRC[31:0]);
FI;

SIMD Floating-Point Exceptions
-Invalid, Precision

Other Exceptions
-EVEX-encoded instructions, see Table 2-48, "Type E3NF Class Exception Conditions".

NOTES:

1. For this specific instruction, EVEX.W in non-64 bit is ignored; the
instructions behaves as if the W0 version is used.
```

13.14.4 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Invalid, Precision In abbreviated form, this includes: IE, PE

13.14.5 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VCVTTSS2SIS r32, xmm1/m32	E3NF	IP	AVX10.2
VCVTTSS2SIS r64, xmm1/m32	E3NF	IP	AVX10.2

13.15 VCVTTSS2USIS

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.LLIG.F3.MAP5.W0 6C /r VCVTTSS2USIS r32, xmm1/m32 {sae}	A	V/V[1]	AVX10.2
EVEX.LLIG.F3.MAP5.W1 6C /r VCVTTSS2USIS r64, xmm1/m32 {sae}	A	V/N.E.	AVX10.2
Notes: 1: Outside of 64b mode, the EVEX.W field is ignored. The instruction behaves as if W=0 were used.			

13.15.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	SCALAR	MODRM.REG(w)	MODRM.R/M(r)	N/A	N/A

13.15.2 DESCRIPTION

Note:

For SCALAR instructions with a CPUID feature flag specifying Intel® AVX10, the programmer can skip the check for the available vector options on the processor at run-time via the VL256 and VL512 bits in Converged Vector ISA Leaf 0x24. These bits enumerate whether the corresponding vector lengths are supported and usually determine the set of instructions available to the programmer. However, scalar instructions are architected without the notion of vector length, so this step does not strictly apply.

VCVTTSS2USIS: Converts a single-precision floating-point value in the source operand (the second operand) to a doubleword unsigned integer (or unsigned quadword integer if operand size is 64 bits) in the destination operand (the first operand) with truncation and saturation. The source operand can be an XMM register or a memory location. The destination operand is a general-purpose register. When the source operand is an XMM register, the single-precision floating-point value is contained in the low doubleword of the register. When a conversion is inexact, the source value is first truncated (round toward zero) and then checked against the range of the destination type. If the truncated value is representable in the result type that value is returned, otherwise the floating-point invalid exception is raised, and if this exception is masked, the representable value closest to the truncated value is returned. If the source value is NaN, zero is returned. If some rounding mode other than truncation is desired, it can be applied using the VROUNDSS or VRNDSCALESS instruction before the conversion. EVEX.W1 version: promotes the instruction to produce 64-bit data in 64-bit mode. Note: EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

13.15.3 OPERATION

```
VCVTTSS2USIS (EVEX encoded version)
IF 64-bit Mode and OperandSize = 64
THEN
   DEST[63:0] := convert_SP_to_QW_UnSignedInteger_TruncateSaturate(SRC[31:0]);
ELSE
   DEST[31:0] := convert_SP_to_DW_UnSignedInteger_TruncateSaturate(SRC[31:0]);
FI;

SIMD Floating-Point Exceptions
-Invalid, Precision

Other Exceptions
-EVEX-encoded instructions, see Table 2-48, "Type E3NF Class Exception Conditions".

NOTES:
1. For this specific instruction, EVEX.W in non-64 bit is ignored; the
instructions behaves as if the W0 version is used.

```

13.15.4 SIMD FLOATING-POINT EXCEPTIONS

This instruction can set the following flags in MXCSR: Invalid, Precision In abbreviated form, this includes: IE, PE

13.15.5 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VCVTTSS2USIS r32, xmm1/m32	E3NF	IP	AVX10.2
VCVTTSS2USIS r64, xmm1/m32	E3NF	IP	AVX10.2

Chapter 14

INTEL® AVX10 ZERO-EXTENDING PARTIAL VECTOR COPY INSTRUCTIONS

14.1 VMOVD

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.128.F3.0F.W0 7E /r VMOVD xmm1, xmm2/m32	A	V/V	AVX10.2
EVEX.128.66.0F.W0 D6 /r VMOVD xmm1/m32, xmm2	B	V/V	AVX10.2

14.1.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	TUPLE1	MODRM.REG(w)	MODRM.R/M(r)	N/A	N/A
B	TUPLE1	MODRM.R/M(w)	MODRM.REG(r)	N/A	N/A

14.1.2 DESCRIPTION

Note:

For SCALAR instructions with a CPUID feature flag specifying Intel® AVX10, the programmer can skip the check for the available vector options on the processor at run-time via the VL256 and VL512 bits in Converged Vector ISA Leaf 0x24. These bits enumerate whether the corresponding vector lengths are supported and usually determine the set of instructions available to the programmer. However, scalar instructions are architected without the notion of vector length, so this step does not strictly apply.

VMOVD: Copies a doubleword from the source operand (second operand) to the destination operand (first operand). The source and destination operands can be XMM registers, or 32-bit memory locations. This instruction can be used to move a doubleword between two XMM registers or between an XMM register and a 32-bit memory location. The instruction cannot be used to transfer data between memory locations. When the source operand is an XMM register, the low doubleword is moved; when the destination operand is an XMM register, the doubleword is stored to the low doubleword of the register, and the remaining bits are cleared to all 0s. Note: EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD. EVEX.LL must be 00b, otherwise instructions will #UD.

14.1.3 OPERATION

```
VMOVD (7E) with XMM register source and destination
DEST[31:0] := SRC[31:0]
DEST[MAXVL-1:32] := 0

VMOVD (D6) with XMM register source and destination
DEST[31:0] := SRC[31:0]
DEST[MAXVL-1:32] := 0

VMOVD (7E) with memory source
DEST[31:0] := SRC[31:0]
DEST[:MAXVL-1:32] := 0

VMOVD (D6) with memory dest
DEST[31:0] := SRC2[31:0]

Flags Affected
None.
```

14.1.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VMOVD xmm1, xmm2/m32	E9NF	N/A	AVX10.2
VMOVD xmm1/m32, xmm2	E9NF	N/A	AVX10.2

14.2 VMOVW

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.128.F3.MAP5.W0 6E /r VMOVW xmm1, xmm2/m16	A	V/V	AVX10.2
EVEX.128.F3.MAP5.W0 7E /r VMOVW xmm1/m16, xmm2	B	V/V	AVX10.2

14.2.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	TUPLE1	MODRM.REG(w)	MODRM.R/M(r)	N/A	N/A
B	TUPLE1	MODRM.R/M(w)	MODRM.REG(r)	N/A	N/A

14.2.2 DESCRIPTION

Note:

For SCALAR instructions with a CPUID feature flag specifying Intel® AVX10, the programmer can skip the check for the available vector options on the processor at run-time via the VL256 and VL512 bits in Converged Vector ISA Leaf 0x24. These bits enumerate whether the corresponding vector lengths are supported and usually determine the set of instructions available to the programmer. However, scalar instructions are architected without the notion of vector length, so this step does not strictly apply.

VMOVW: Copies a word from the source operand (second operand) to the destination operand (first operand). The source and destination operands can be XMM registers, or 16-bit memory locations. This instruction can be used to move a word between two XMM registers or between an XMM register and a 16-bit memory location. The instruction cannot be used to transfer data between memory locations. When the source operand is an XMM register, the low word is moved; when the destination operand is an XMM register, the word is stored to the low word of the register, and the remaining bits are cleared to all 0s. Note: EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD. EVEX.LL must be 00b, otherwise instructions will #UD.

14.2.3 OPERATION

```
VMOVW (xx) with XMM register source and destination
DEST[15:0] := SRC[15:0]
DEST[MAXVL-1:16] := 0

VMOVW (xx) with XMM register source and destination
DEST[15:0] := SRC[15:0]
DEST[MAXVL-1:16] := 0

VMOVW (xx) with memory source
DEST[15:0] := SRC[15:0]
DEST[:MAXVL-1:16] := 0

VMOVW (xx) with memory dest
DEST[15:0] := SRC2[15:0]

```

14.2.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VMOVW xmm1, xmm2/m16	E9NF	N/A	AVX10.2
VMOVW xmm1/m16, xmm2	E9NF	N/A	AVX10.2

Chapter 15

INTEL® AVX10-AMX INSTRUCTIONS

15.1 TCVTROWD2PS

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.512.F3.0F38.W0 4A 11:rrr:bbb TCVTROWD2PS zmm1, tmm2, r32	A	V/N.E.	AMX-AVX512 and AVX10.2
EVEX.512.F3.0F3A.W0 07 11:rrr:bbb /ib TCVTROWD2PS zmm1, tmm2, imm8	B	V/N.E.	AMX-AVX512 and AVX10.2

15.1.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	MODRM.REG(w)	MODRM.R/M(r)	VVVV(r)	N/A
B	N/A	MODRM.REG(w)	MODRM.R/M(r)	IMM8(r)	N/A

15.1.2 DESCRIPTION

Note:

For instructions with a CPUID feature flag specifying Intel® AVX10, the programmer must check the available vector options on the processor at run-time via the VL256 and VL512 bits in Converged Vector ISA Leaf 0x24. These bits enumerate whether the corresponding vector lengths are supported (128-bit vectors are always supported) and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Note:

Intel® AMX forms of this instruction are not considered TSX-friendly. Any attempt to execute an AMX instruction inside a TSX transaction will result in a transaction abort.

This instruction moves a row from a tile register to a zmm destination register, converting the int32 source elements to fp32. The row of the tile is selected by an imm8, or a 32b GPR. If the row indicated is out of range, the instruction will #GP.

No SIMD exceptions are generated. Rounding is done as if MXCSR.RC=RNE. Embedded rounding is not supported. MXCSR is neither consulted nor updated.

Any attempt to execute an AMX instruction inside a TSX transaction will result in a transaction abort.

15.1.3 OPERATION

```
TCVTROWD2PS zdest, tsrc, imm8
VL = (512)
VL_bytes =  VL >> 3 //bits to bytes
row_index := imm8&0x3f
row_chunk := (imm8>>6) * VL_bytes

for i in 0 ... (VL_bytes/4)-1:
    if i+row_chunk/4 >= tsrc.colsb/4:
         zdest.dword[i] := 0
    else:
         zdest.f32[i] := CONVERT_INT32_TO_FP32(tsrc.row[row_index].dword[row_chunk/4+i],
                                               RNE)
zdest[MAX_VL-1:VL] := 0
zero_tileconfig_start()
```

```
TCVTROWD2PS zdest, tsrc, r32
VL = (512)
VL_bytes  := VL >> 3 // bits to bytes
row_index := r32&0xffff
row_chunk := ((r32>>16) & 0xffff) * VL_bytes

for i in 0 ... (VL_bytes/4)-1:
    if i + row_chunk/4 >= tsrc.colsb/4:
         zdest.dword[i] := 0
    else:
         zdest.f32[i] := CONVERT_INT32_TO_FP32(tsrc.row[row_index].dword[row_chunk/4+i],
                                               RNE)
zdest[MAX_VL-1:VL] := 0
zero_tileconfig_start()
```

15.1.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
TCVTROWD2PS zmm1, tmm2, r32	AMX-E8-EVEX	N/A	AMX-AVX512, AVX10.2
TCVTROWD2PS zmm1, tmm2, imm8	AMX-E7-EVEX	N/A	AMX-AVX512, AVX10.2

15.2 TCVTROWPS2BF16[H,L]

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.512.F2.0F38.W0 6D 11:rrr:bbb TCVTROWPS2BF16H zmm1, tmm2, r32	A	V/N.E.	AMX-AVX512 and AVX10.2
EVEX.512.F2.0F3A.W0 07 11:rrr:bbb /ib TCVTROWPS2BF16H zmm1, tmm2, imm8	B	V/N.E.	AMX-AVX512 and AVX10.2
EVEX.512.F3.0F38.W0 6D 11:rrr:bbb TCVTROWPS2BF16L zmm1, tmm2, r32	A	V/N.E.	AMX-AVX512 and AVX10.2
EVEX.512.F3.0F3A.W0 77 11:rrr:bbb /ib TCVTROWPS2BF16L zmm1, tmm2, imm8	B	V/N.E.	AMX-AVX512 and AVX10.2

15.2.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	MODRM.REG(w)	MODRM.R/M(r)	VVVV(r)	N/A
B	N/A	MODRM.REG(w)	MODRM.R/M(r)	IMM8(r)	N/A

15.2.2 DESCRIPTION

Note:

For instructions with a CPUID feature flag specifying Intel® AVX10, the programmer must check the available vector options on the processor at run-time via the VL256 and VL512 bits in Converged Vector ISA Leaf 0x24. These bits enumerate whether the corresponding vector lengths are supported (128-bit vectors are always supported) and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Note:

Intel® AMX forms of this instruction are not considered TSX-friendly. Any attempt to execute an AMX instruction inside a TSX transaction will result in a transaction abort.

This instruction moves a row from a tile register to a zmm destination register, converting the fp32 source elements to bf16. The row of the tile is selected by an imm8, or a 32b GPR. If the row indicated is out of range, the instruction will #GP.

TCVTROWPS2BF16H places the resulting bf16 elements in the high 16 bits within each dword, while TCVTROWPS2BF16L places them in the low 16 bits.

No SIMD exceptions are generated. Rounding is done as if MXCSR.RC=RNE. Embedded rounding is not supported. MXCSR is neither consulted nor updated.

Any attempt to execute an AMX instruction inside a TSX transaction will result in a transaction abort.

15.2.3 OPERATION

```
TCVTROWPS2BF16[H,L] zdest, tsrc, imm8
VL = (512)
VL_bytes  := VL>>3 // bits to bytes
row_index := imm8&0x3f
row_chunk := (imm8>>6) * VL_bytes

if instruction is TCVTROWPS2BF16H:
     pos := 1
     zeropos := 0
else:
     pos := 0
     zeropos := 1


for i in 0 ... (VL_bytes/4)-1:
    if i+row_chunk/4 >= tsrc.colsb/4:
         zdest.dword[i] := 0
     else:
         zdest.word[2*i+zeropos] := 0
         zdest.bf16[2*i+pos] := CONVERT_FP32_TO_BF16(
                                          tsrc.row[row_index].fp32[row_chunk/4+i],
                                          RNE)

zdest[MAX_VL-1:VL] := 0
zero_tileconfig_start()
```

```
TCVTROWPS2BF16[H,L] zdest, tsrc, r32
VL = (512)
VL_bytes  := VL >> 3 // bits to bytes
row_index := r32&0xffff
row_chunk := ((r32>>16) & 0xffff) * VL_bytes

if instruction is TCVTROWPS2BF16H:
     pos := 1
     zeropos := 0
else:
     pos := 0
     zeropos := 1

for i in 0 ... (VL_bytes/4)-1:
    if i+row_chunk/4 >= tsrc.colsb/4:
         zdest.dword[i] := 0
     else:
         zdest.word[2*i+zeropos] := 0
         zdest.bf16[2*i+pos] := CONVERT_FP32_TO_BF16(
                                          tsrc.row[row_index].fp32[row_chunk/4+i],
                                          RNE)

zdest[MAX_VL-1:VL] := 0
zero_tileconfig_start()
```

15.2.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
TCVTROWPS2BF16H zmm1, tmm2, r32	AMX-E8-EVEX	N/A	AMX-AVX512, AVX10.2
TCVTROWPS2BF16H zmm1, tmm2, imm8	AMX-E7-EVEX	N/A	AMX-AVX512, AVX10.2
TCVTROWPS2BF16L zmm1, tmm2, r32	AMX-E8-EVEX	N/A	AMX-AVX512, AVX10.2
TCVTROWPS2BF16L zmm1, tmm2, imm8	AMX-E7-EVEX	N/A	AMX-AVX512, AVX10.2

15.3 TCVTROWPS2PH[H,L]

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.512.NP.0F38.W0 6D 11:rrr:bbb TCVTROWPS2PHH zmm1, tmm2, r32	A	V/N.E.	AMX-AVX512 and AVX10.2
EVEX.512.NP.0F3A.W0 07 11:rrr:bbb /ib TCVTROWPS2PHH zmm1, tmm2, imm8	B	V/N.E.	AMX-AVX512 and AVX10.2
EVEX.512.66.0F38.W0 6D 11:rrr:bbb TCVTROWPS2PHL zmm1, tmm2, r32	A	V/N.E.	AMX-AVX512 and AVX10.2
EVEX.512.F2.0F3A.W0 77 11:rrr:bbb /ib TCVTROWPS2PHL zmm1, tmm2, imm8	B	V/N.E.	AMX-AVX512 and AVX10.2

15.3.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	MODRM.REG(w)	MODRM.R/M(r)	VVVV(r)	N/A
B	N/A	MODRM.REG(w)	MODRM.R/M(r)	IMM8(r)	N/A

15.3.2 DESCRIPTION

Note:

For instructions with a CPUID feature flag specifying Intel® AVX10, the programmer must check the available vector options on the processor at run-time via the VL256 and VL512 bits in Converged Vector ISA Leaf 0x24. These bits enumerate whether the corresponding vector lengths are supported (128-bit vectors are always supported) and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Note:

Intel® AMX forms of this instruction are not considered TSX-friendly. Any attempt to execute an AMX instruction inside a TSX transaction will result in a transaction abort.

This instruction moves a row from a tile register to a zmm destination register, converting the fp32 source elements to fp16. The row of the tile is selected by an imm8, or a 32b GPR. If the row indicated is out of range, the instruction will #GP.

TCVTROWPS2PHH places the resulting fp16 elements in the high 16 bits within each dword, while TCVTROWPS2PHL places them in the low 16 bits.

No SIMD exceptions are generated. Rounding is done as if MXCSR.RC=RNE. Embedded rounding is not supported. MXCSR is neither consulted nor updated.

Input FP32 denormals become FP16 zeros on outputs. This instruction can produce FP16 denormal outputs. (MXCSR.FTZ only controls FP32 and FP64 denormal outputs).

Any attempt to execute an AMX instruction inside a TSX transaction will result in a transaction abort.

15.3.3 OPERATION

```
TCVTROWPS2PH[H,L] zdest, tsrc, imm8
VL = (512)
VL_bytes  := VL>>3  // bits to bytes
row_index := imm8&0x3f
row_chunk := (imm8>>6) * VL_bytes

if instruction is TCVTROWPS2PHH:
     pos := 1
     zeropos := 0
else:
     pos := 0
     zeropos := 1

for i in 0 ... (VL_bytes/4)-1:
    if i+row_chunk/4 >= tsrc.colsb/4:
         zdest.dword[i] := 0
     else:
         zdest.word[2*i+zeropos] := 0
         zdest.f16[2*i+pos] := CONVERT_FP32_TO_FP16(
                                          tsrc.row[row_index].fp32[row_chunk/4+i],
                                          RNE)

zdest[MAX_VL-1:VL] := 0
zero_tileconfig_start()
```

```
TCVTROWPS2PH[H,L] zdest, tsrc, r32
VL = (512)
VL_bytes  := VL>>3 // bits to bytes
row_index := r32&0xffff
row_chunk := ((r32>>16) & 0xffff) * VL_bytes

if instruction is TCVTROWPS2PHH:
     pos := 1
     zeropos := 0
else:
     pos := 0
     zeropos := 1

for i in 0 ... (VL_bytes/4)-1:
    if i + row_chunk/4 >= tsrc.colsb/4:
         zdest.dword[i] := 0
     else:
         zdest.word[2*i+zeropos] := 0
         zdest.f16[2*i+pos] := CONVERT_FP32_TO_FP16(
                                          tsrc.row[row_index].fp32[row_chunk/4+i],
                                          RNE)

zdest[MAX_VL-1:VL] := 0
zero_tileconfig_start()
```

15.3.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
TCVTROWPS2PHH zmm1, tmm2, r32	AMX-E8-EVEX	N/A	AMX-AVX512, AVX10.2
TCVTROWPS2PHH zmm1, tmm2, imm8	AMX-E7-EVEX	N/A	AMX-AVX512, AVX10.2
TCVTROWPS2PHL zmm1, tmm2, r32	AMX-E8-EVEX	N/A	AMX-AVX512, AVX10.2
TCVTROWPS2PHL zmm1, tmm2, imm8	AMX-E7-EVEX	N/A	AMX-AVX512, AVX10.2

15.4 TILEMOVROW

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.512.66.0F3A.W0 07 11:rrr:bbb /ib TILEMOVROW zmm1, tmm2, imm8	A	V/N.E.	AMX-AVX512 and AVX10.2
EVEX.512.66.0F38.W0 4A 11:rrr:bbb TILEMOVROW zmm1, tmm2, r32	B	V/N.E.	AMX-AVX512 and AVX10.2

15.4.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	MODRM.REG(w)	MODRM.R/M(r)	IMM8(r)	N/A
B	N/A	MODRM.REG(w)	MODRM.R/M(r)	VVVV(r)	N/A

15.4.2 DESCRIPTION

Note:

For instructions with a CPUID feature flag specifying Intel® AVX10, the programmer must check the available vector options on the processor at run-time via the VL256 and VL512 bits in Converged Vector ISA Leaf 0x24. These bits enumerate whether the corresponding vector lengths are supported (128-bit vectors are always supported) and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Note:

Intel® AMX forms of this instruction are not considered TSX-friendly. Any attempt to execute an AMX instruction inside a TSX transaction will result in a transaction abort.

These instructions move one row of a tile register to a zmm register. The row of the tile is selected by an imm8, or a 32b GPR. If the row indicated is out of range, the instruction will #GP.

Any attempt to execute an AMX instruction inside a TSX transaction will result in a transaction abort.

15.4.3 OPERATION

```
TILEMOVROW zdest, tsrc, imm8
VL = (512)
VL_bytes  := VL>>3 // bits to bytes
row_index := imm8&0x3f
row_chunk := (imm8>>6) * VL_bytes

for i in 0 ... VL_bytes-1:
    if row_chunk + i >= tsrc.colsb:
         zdest.byte[i] := 0
    else:
         zdest.byte[i] := tsrc.row[row_index].byte[row_chunk+i]
zdest[MAX_VL-1:VL] := 0
zero_tileconfig_start()
```

```
TILEMOVROW zdest, tsrc, r32
VL = (512)
VL_bytes  := VL>>3 // bits to bytes
row_index := r32&0xffff
row_chunk := ((r32>>16) & 0xffff) * VL_bytes

for i in 0 ... VL_bytes-1:
    if row_chunk + i >= tsrc.colsb:
         zdest.byte[i] := 0
    else:
         zdest.byte[i] := tsrc.row[row_index].byte[row_chunk+i]
zdest[MAX_VL-1:VL] := 0
zero_tileconfig_start()
```

15.4.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
TILEMOVROW zmm1, tmm2, imm8	AMX-E7-EVEX	N/A	AMX-AVX512, AVX10.2
TILEMOVROW zmm1, tmm2, r32	AMX-E8-EVEX	N/A	AMX-AVX512, AVX10.2

Chapter 16

INTEL® AVX10-DEPENDENT INSTRUCTIONS

16.1 VMOVRS

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.128.F2.MAP5.W0 6F !(11):rrr:bbb VMOVRSB xmm1{k1}{z}, m128	A	V/N.E.	AVX10.2 and MOVRS
EVEX.256.F2.MAP5.W0 6F !(11):rrr:bbb VMOVRSB ymm1{k1}{z}, m256	A	V/N.E.	AVX10.2 and MOVRS
EVEX.512.F2.MAP5.W0 6F !(11):rrr:bbb VMOVRSB zmm1{k1}{z}, m512	A	V/N.E.	AVX10.2 and MOVRS
EVEX.128.F3.MAP5.W0 6F !(11):rrr:bbb VMOVRSD xmm1{k1}{z}, m128	A	V/N.E.	AVX10.2 and MOVRS
EVEX.256.F3.MAP5.W0 6F !(11):rrr:bbb VMOVRSD ymm1{k1}{z}, m256	A	V/N.E.	AVX10.2 and MOVRS
EVEX.512.F3.MAP5.W0 6F !(11):rrr:bbb VMOVRSD zmm1{k1}{z}, m512	A	V/N.E.	AVX10.2 and MOVRS
EVEX.128.F3.MAP5.W1 6F !(11):rrr:bbb VMOVRSQ xmm1{k1}{z}, m128	A	V/N.E.	AVX10.2 and MOVRS
EVEX.256.F3.MAP5.W1 6F !(11):rrr:bbb VMOVRSQ ymm1{k1}{z}, m256	A	V/N.E.	AVX10.2 and MOVRS
EVEX.512.F3.MAP5.W1 6F !(11):rrr:bbb VMOVRSQ zmm1{k1}{z}, m512	A	V/N.E.	AVX10.2 and MOVRS
EVEX.128.F2.MAP5.W1 6F !(11):rrr:bbb VMOVRSW xmm1{k1}{z}, m128	A	V/N.E.	AVX10.2 and MOVRS
EVEX.256.F2.MAP5.W1 6F !(11):rrr:bbb VMOVRSW ymm1{k1}{z}, m256	A	V/N.E.	AVX10.2 and MOVRS
EVEX.512.F2.MAP5.W1 6F !(11):rrr:bbb VMOVRSW zmm1{k1}{z}, m512	A	V/N.E.	AVX10.2 and MOVRS

16.1.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULLMEM	MODRM.REG(w)	MODRM.R/M(r)	N/A	N/A

16.1.2 DESCRIPTION

Note:

For instructions with a CPUID feature flag specifying Intel® AVX10, the programmer must check the available vector options on the processor at run-time via the VL256 and VL512 bits in Converged Vector ISA Leaf 0x24. These bits enumerate whether the corresponding vector lengths are supported (128-bit vectors are always supported) and as such will determine the set of instructions available to the programmer listed in the above opcode table.

Moves data from the source operand (second operand), a memory operand, to the destination operand (first operand), a register. This instruction may be used to load an XMM, YMM, or ZMM register from a 128-bit, 256-bit, or 512-bit memory location. The data may be 16, 32, or 64 bytes; 8, 16, or 32 words; 4, 8, or 16 doublewords; or 2, 4, or 8 quadwords. Additionally, this instruction indicates the source memory location is likely to become read-shared by multiple processors, i.e., read in the future by at least one other processor before it is written, assuming it is ever written in the future. Implementations may optimize the behavior of the caches, especially shared caches, for this data for future reads by multiple processors. A future write to this data before it becomes read-shared will behave as usual, but its performance may be less optimal than if the current read were done via a load without a read-shared hint.

16.1.3 OPERATION

```
VMOVRSB dest, src

(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j := 0 TO KL-1
  i := j * 8
  IF k1[j] OR *no writemask*
      THEN DEST[i+7:i] := SRC[i+7:i]
      ELSE
          IF *merging-masking* ; merging-masking
              THEN *DEST[i+7:i] remains unchanged*
              ELSE DEST[i+7:i] := 0 ; zeroing-masking
          FI
      FI;
ENDFOR
dest[MAXVL-1:VL] := 0
```

```
VMOVRSW dest, src

(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j := 0 TO KL-1
  i := j * 16
  IF k1[j] OR *no writemask*
      THEN DEST[i+15:i] := SRC[i+15:i]
      ELSE
          IF *merging-masking* ; merging-masking
              THEN *DEST[i+15:i] remains unchanged*
              ELSE DEST[i+15:i] := 0 ; zeroing-masking
          FI
      FI;
ENDFOR
dest[MAXVL-1:VL] := 0
```

```
VMOVRSD dest, src

(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
  i := j * 32
  IF k1[j] OR *no writemask*
      THEN DEST[i+31:i] := SRC[i+31:i]
      ELSE
          IF *merging-masking* ; merging-masking
              THEN *DEST[i+31:i] remains unchanged*
              ELSE DEST[i+31:i] := 0 ; zeroing-masking
          FI
      FI;
ENDFOR
dest[MAXVL-1:VL] := 0
```

```
VMOVRSQ dest, src

(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
  i := j * 64
  IF k1[j] OR *no writemask*
      THEN DEST[i+63:i] := SRC[i+63:i]
      ELSE
          IF *merging-masking* ; merging-masking
              THEN *DEST[i+63:i] remains unchanged*
              ELSE DEST[i+63:i] := 0 ; zeroing-masking
          FI
      FI;
ENDFOR
dest[MAXVL-1:VL] := 0
```

16.1.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VMOVRSB xmm1, m128	E4	N/A	AVX10.2, MOVRS
VMOVRSB ymm1, m256	E4	N/A	AVX10.2, MOVRS
VMOVRSB zmm1, m512	E4	N/A	AVX10.2, MOVRS
VMOVRSD xmm1, m128	E4	N/A	AVX10.2, MOVRS
VMOVRSD ymm1, m256	E4	N/A	AVX10.2, MOVRS
VMOVRSD zmm1, m512	E4	N/A	AVX10.2, MOVRS
VMOVRSQ xmm1, m128	E4	N/A	AVX10.2, MOVRS
VMOVRSQ ymm1, m256	E4	N/A	AVX10.2, MOVRS
VMOVRSQ zmm1, m512	E4	N/A	AVX10.2, MOVRS
VMOVRSW xmm1, m128	E4	N/A	AVX10.2, MOVRS
VMOVRSW ymm1, m256	E4	N/A	AVX10.2, MOVRS
VMOVRSW zmm1, m512	E4	N/A	AVX10.2, MOVRS

Chapter 17

INTEL® SM4 INSTRUCTIONS

17.1 VSM4KEY4

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.128.F3.0F38.W0 DA /r VSM4KEY4 xmm1, xmm2, xmm3/m128	A	V/V	AVX10.2 and SM4
EVEX.256.F3.0F38.W0 DA /r VSM4KEY4 ymm1, ymm2, ymm3/m256	A	V/V	AVX10.2 and SM4
EVEX.512.F3.0F38.W0 DA /r VSM4KEY4 zmm1, zmm2, zmm3/m512	A	V/V	AVX10.2 and SM4

17.1.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULLMEM	MODRM.REG(w)	VVVV(r)	MODRM.R/M(r)	N/A

17.1.2 DESCRIPTION

Note:

For instructions with a CPUID feature flag specifying Intel® AVX10, the programmer must check the available vector options on the processor at run-time via the VL256 and VL512 bits in Converged Vector ISA Leaf 0x24. These bits enumerate whether the corresponding vector lengths are supported (128-bit vectors are always supported) and as such will determine the set of instructions available to the programmer listed in the above opcode table.

The VSM4KEY4 instruction performs 4 rounds of SM4 Key Expansion. The instruction operates on independent 128-bit lanes.

“Commercial Cryptography” is a set of algorithms and standards used in the commercial area issued and regulated by the Office of State Commercial Cryptography Administration (OSCCA). SM3 and SM4 algorithms are part of the published OSCCA algorithms.

SM4 is a symmetric block cipher algorithm published in 2012.

More details can be found at: https://tools.ietf.org/html/draft-ribose-cfrg-sm4-10

Both SM4 instructions use a common sbox table:

```
BYTE sbox[256] = {
0xD6, 0x90, 0xE9, 0xFE, 0xCC, 0xE1, 0x3D, 0xB7, 0x16, 0xB6, 0x14, 0xC2, 0x28, 0xFB, 0x2C, 0x05,
0x2B, 0x67, 0x9A, 0x76, 0x2A, 0xBE, 0x04, 0xC3, 0xAA, 0x44, 0x13, 0x26, 0x49, 0x86, 0x06, 0x99,
0x9C, 0x42, 0x50, 0xF4, 0x91, 0xEF, 0x98, 0x7A, 0x33, 0x54, 0x0B, 0x43, 0xED, 0xCF, 0xAC, 0x62,
0xE4, 0xB3, 0x1C, 0xA9, 0xC9, 0x08, 0xE8, 0x95, 0x80, 0xDF, 0x94, 0xFA, 0x75, 0x8F, 0x3F, 0xA6,
0x47, 0x07, 0xA7, 0xFC, 0xF3, 0x73, 0x17, 0xBA, 0x83, 0x59, 0x3C, 0x19, 0xE6, 0x85, 0x4F, 0xA8,
0x68, 0x6B, 0x81, 0xB2, 0x71, 0x64, 0xDA, 0x8B, 0xF8, 0xEB, 0x0F, 0x4B, 0x70, 0x56, 0x9D, 0x35,
0x1E, 0x24, 0x0E, 0x5E, 0x63, 0x58, 0xD1, 0xA2, 0x25, 0x22, 0x7C, 0x3B, 0x01, 0x21, 0x78, 0x87,
0xD4, 0x00, 0x46, 0x57, 0x9F, 0xD3, 0x27, 0x52, 0x4C, 0x36, 0x02, 0xE7, 0xA0, 0xC4, 0xC8, 0x9E,
0xEA, 0xBF, 0x8A, 0xD2, 0x40, 0xC7, 0x38, 0xB5, 0xA3, 0xF7, 0xF2, 0xCE, 0xF9, 0x61, 0x15, 0xA1,
0xE0, 0xAE, 0x5D, 0xA4, 0x9B, 0x34, 0x1A, 0x55, 0xAD, 0x93, 0x32, 0x30, 0xF5, 0x8C, 0xB1, 0xE3,
0x1D, 0xF6, 0xE2, 0x2E, 0x82, 0x66, 0xCA, 0x60, 0xC0, 0x29, 0x23, 0xAB, 0x0D, 0x53, 0x4E, 0x6F,
0xD5, 0xDB, 0x37, 0x45, 0xDE, 0xFD, 0x8E, 0x2F, 0x03, 0xFF, 0x6A, 0x72, 0x6D, 0x6C, 0x5B, 0x51,
0x8D, 0x1B, 0xAF, 0x92, 0xBB, 0xDD, 0xBC, 0x7F, 0x11, 0xD9, 0x5C, 0x41, 0x1F, 0x10, 0x5A, 0xD8,
0x0A, 0xC1, 0x31, 0x88, 0xA5, 0xCD, 0x7B, 0xBD, 0x2D, 0x74, 0xD0, 0x12, 0xB8, 0xE5, 0xB4, 0xB0,
0x89, 0x69, 0x97, 0x4A, 0x0C, 0x96, 0x77, 0x7E, 0x65, 0xB9, 0xF1, 0x09, 0xC5, 0x6E, 0xC6, 0x84,
0x18, 0xF0, 0x7D, 0xEC, 0x3A, 0xDC, 0x4D, 0x20, 0x79, 0xEE, 0x5F, 0x3E, 0xD7, 0xCB, 0x39, 0x48
}
```

17.1.3 OPERATION

```
define ROL32(dword, n):
  count := n % 32
  dest := (dword << count) | (dword >> (32-count))
  return dest

define SBOX_BYTE(dword, i):
     // sbox[] array defined in introduction
     return sbox[dword.byte[i]]

define lower_t(dword):
    tmp.byte[0] := SBOX_BYTE(dword, 0)
    tmp.byte[1] := SBOX_BYTE(dword, 1)
    tmp.byte[2] := SBOX_BYTE(dword, 2)
    tmp.byte[3] := SBOX_BYTE(dword, 3)
    return tmp

define L_KEY(dword):
   return dword ^ ROL32(dword, 13) ^ ROL32(dword, 23)

define T_KEY(dword):
    return L_KEY(lower_t(dword))

define F_KEY(X0, X1, X2, X3, round_key):
    return X0 ^ T_KEY(X1 ^ X2 ^ X3 ^ round_key)
```

```
VSM4KEY4 DEST, SRC1, SRC2

VL = (128,256) // VEX versions
// or
VL = (128,256,512) // EVEX versions

KL := VL/128

for i in 0..KL-1:
    P[0] := SRC1.xmm[i].dword[0]
    P[1] := SRC1.xmm[i].dword[1]
    P[2] := SRC1.xmm[i].dword[2]
    P[3] := SRC1.xmm[i].dword[3]

    C[0] := F_KEY(P[0], P[1], P[2], P[3], SRC2.xmm[i].dword[0])
    C[1] := F_KEY(P[1], P[2], P[3], C[0], SRC2.xmm[i].dword[1])
    C[2] := F_KEY(P[2], P[3], C[0], C[1], SRC2.xmm[i].dword[2])
    C[3] := F_KEY(P[3], C[0], C[1], C[2], SRC2.xmm[i].dword[3])

    DEST.xmm[i].dword[0] := C[0]
    DEST.xmm[i].dword[1] := C[1]
    DEST.xmm[i].dword[2] := C[2]
    DEST.xmm[i].dword[3] := C[3]

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

17.1.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VSM4KEY4 xmm1, xmm2, xmm3/m128	E6	N/A	AVX10.2, SM4
VSM4KEY4 ymm1, ymm2, ymm3/m256	E6	N/A	AVX10.2, SM4
VSM4KEY4 zmm1, zmm2, zmm3/m512	E6	N/A	AVX10.2, SM4

17.2 VSM4RNDS4

Encoding / Instruction	Op/En	64/32-bit mode	CPUID
EVEX.128.F2.0F38.W0 DA /r VSM4RNDS4 xmm1, xmm2, xmm3/m128	A	V/V	AVX10.2 and SM4
EVEX.256.F2.0F38.W0 DA /r VSM4RNDS4 ymm1, ymm2, ymm3/m256	A	V/V	AVX10.2 and SM4
EVEX.512.F2.0F38.W0 DA /r VSM4RNDS4 zmm1, zmm2, zmm3/m512	A	V/V	AVX10.2 and SM4

17.2.1 INSTRUCTION OPERAND ENCODING

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	FULLMEM	MODRM.REG(w)	VVVV(r)	MODRM.R/M(r)	N/A

17.2.2 DESCRIPTION

Note:

For instructions with a CPUID feature flag specifying Intel® AVX10, the programmer must check the available vector options on the processor at run-time via the VL256 and VL512 bits in Converged Vector ISA Leaf 0x24. These bits enumerate whether the corresponding vector lengths are supported (128-bit vectors are always supported) and as such will determine the set of instructions available to the programmer listed in the above opcode table.

The SM4RNDS4 instruction performs 4 rounds of SM4 Encryption. The instruction operates on independent 128-bit lanes.

“Commercial Cryptography” is a set of algorithms and standards used in the commercial area issued and regulated by the Office of State Commercial Cryptography Administration (OSCCA). SM3 and SM4 algorithms are part of the published OSCCA algorithms.

SM4 is a symmetric block cipher algorithm published in 2012.

More details can be found at: https://tools.ietf.org/html/draft-ribose-cfrg-sm4-10

See VSM4KEY4 for the sbox table.

17.2.3 OPERATION

```
// see vsm4key4 for definition of ROL32, lower_t

define L_RND(dword):
  tmp := dword
  tmp := tmp ^ ROL32(dword, 2)
  tmp := tmp ^ ROL32(dword, 10)
  tmp := tmp ^ ROL32(dword, 18)
  tmp := tmp ^ ROL32(dword, 24)
  return tmp

define T_RND(dword):
  return L_RND(lower_t(dword))

define F_RND(X0, X1, X2, X3, round_key):
  return X0 ^ T_RND(X1 ^ X2 ^ X3 ^ round_key)

VSM4RNDS4 DEST, SRC1, SRC2
VL = (128,256) // VEX versions
//or
VL = (128,256,512) // EVEX versions

KL := VL/128
for i in 0..KL-1:
    P[0] := SRC1.xmm[i].dword[0]
    P[1] := SRC1.xmm[i].dword[1]
    P[2] := SRC1.xmm[i].dword[2]
    P[3] := SRC1.xmm[i].dword[3]

    C[0] := F_RND(P[0], P[1], P[2], P[3], SRC2.xmm[i].dword[0])
    C[1] := F_RND(P[1], P[2], P[3], C[0], SRC2.xmm[i].dword[1])
    C[2] := F_RND(P[2], P[3], C[0], C[1], SRC2.xmm[i].dword[2])
    C[3] := F_RND(P[3], C[0], C[1], C[2], SRC2.xmm[i].dword[3])

    DEST.xmm[i].dword[0] := C[0]
    DEST.xmm[i].dword[1] := C[1]
    DEST.xmm[i].dword[2] := C[2]
    DEST.xmm[i].dword[3] := C[3]

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

17.2.4 EXCEPTIONS

Instruction	Exception Type	Arithmetic Flags	CPUID
VSM4RNDS4 xmm1, xmm2, xmm3/m128	E6	N/A	AVX10.2, SM4
VSM4RNDS4 ymm1, ymm2, ymm3/m256	E6	N/A	AVX10.2, SM4
VSM4RNDS4 zmm1, zmm2, zmm3/m512	E6	N/A	AVX10.2, SM4