

# Intel<sup>®</sup> 64 and IA-32 Architectures Software Developer's Manual

## Documentation Changes

---

June 2016

**Notice:** The Intel<sup>®</sup> 64 and IA-32 architectures may contain design defects or errors known as errata that may cause the product to deviate from published specifications. Current characterized errata are documented in the specification updates.

Document Number: 252046-051



Intel technologies features and benefits depend on system configuration and may require enabled hardware, software, or service activation. Learn more at [intel.com](http://intel.com), or from the OEM or retailer.

No computer system can be absolutely secure. Intel does not assume any liability for lost or stolen data or systems or any damages resulting from such losses.

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.

No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document.

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.

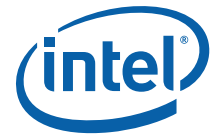
This document contains information on products, services and/or processes in development. All information provided here is subject to change without notice. Contact your Intel representative to obtain the latest Intel product specifications and roadmaps

Copies of documents which have an order number and are referenced in this document, or other Intel literature, may be obtained by calling 1-800-548-4725, or by visiting <http://www.intel.com/design/literature.htm>.

Intel, the Intel logo, Intel Atom, Intel Core, Intel SpeedStep, MMX, Pentium, VTune, and Xeon are trademarks of Intel Corporation in the U.S. and/or other countries.

\*Other names and brands may be claimed as the property of others.

Copyright © 1997-2016, Intel Corporation. All Rights Reserved.



# Contents

---

<b>Revision History</b> . . . . .	4
<b>Preface</b> . . . . .	7
<b>Summary Tables of Changes</b> . . . . .	8
<b>Documentation Changes</b> . . . . .	9



## Revision History

---

Revision	Description	Date
-001	<ul style="list-style-type: none"><li>Initial release</li></ul>	November 2002
-002	<ul style="list-style-type: none"><li>Added 1-10 Documentation Changes.</li><li>Removed old Documentation Changes items that already have been incorporated in the published Software Developer's manual</li></ul>	December 2002
-003	<ul style="list-style-type: none"><li>Added 9 -17 Documentation Changes.</li><li>Removed Documentation Change #6 - References to bits Gen and Len Deleted.</li><li>Removed Documentation Change #4 - VIF Information Added to CLI Discussion</li></ul>	February 2003
-004	<ul style="list-style-type: none"><li>Removed Documentation changes 1-17.</li><li>Added Documentation changes 1-24.</li></ul>	June 2003
-005	<ul style="list-style-type: none"><li>Removed Documentation Changes 1-24.</li><li>Added Documentation Changes 1-15.</li></ul>	September 2003
-006	<ul style="list-style-type: none"><li>Added Documentation Changes 16- 34.</li></ul>	November 2003
-007	<ul style="list-style-type: none"><li>Updated Documentation changes 14, 16, 17, and 28.</li><li>Added Documentation Changes 35-45.</li></ul>	January 2004
-008	<ul style="list-style-type: none"><li>Removed Documentation Changes 1-45.</li><li>Added Documentation Changes 1-5.</li></ul>	March 2004
-009	<ul style="list-style-type: none"><li>Added Documentation Changes 7-27.</li></ul>	May 2004
-010	<ul style="list-style-type: none"><li>Removed Documentation Changes 1-27.</li><li>Added Documentation Changes 1.</li></ul>	August 2004
-011	<ul style="list-style-type: none"><li>Added Documentation Changes 2-28.</li></ul>	November 2004
-012	<ul style="list-style-type: none"><li>Removed Documentation Changes 1-28.</li><li>Added Documentation Changes 1-16.</li></ul>	March 2005
-013	<ul style="list-style-type: none"><li>Updated title.</li><li>There are no Documentation Changes for this revision of the document.</li></ul>	July 2005
-014	<ul style="list-style-type: none"><li>Added Documentation Changes 1-21.</li></ul>	September 2005
-015	<ul style="list-style-type: none"><li>Removed Documentation Changes 1-21.</li><li>Added Documentation Changes 1-20.</li></ul>	March 9, 2006
-016	<ul style="list-style-type: none"><li>Added Documentation changes 21-23.</li></ul>	March 27, 2006
-017	<ul style="list-style-type: none"><li>Removed Documentation Changes 1-23.</li><li>Added Documentation Changes 1-36.</li></ul>	September 2006
-018	<ul style="list-style-type: none"><li>Added Documentation Changes 37-42.</li></ul>	October 2006
-019	<ul style="list-style-type: none"><li>Removed Documentation Changes 1-42.</li><li>Added Documentation Changes 1-19.</li></ul>	March 2007
-020	<ul style="list-style-type: none"><li>Added Documentation Changes 20-27.</li></ul>	May 2007
-021	<ul style="list-style-type: none"><li>Removed Documentation Changes 1-27.</li><li>Added Documentation Changes 1-6</li></ul>	November 2007
-022	<ul style="list-style-type: none"><li>Removed Documentation Changes 1-6</li><li>Added Documentation Changes 1-6</li></ul>	August 2008
-023	<ul style="list-style-type: none"><li>Removed Documentation Changes 1-6</li><li>Added Documentation Changes 1-21</li></ul>	March 2009





Revision	Description	Date
-024	<ul style="list-style-type: none"> <li>Removed Documentation Changes 1-21</li> <li>Added Documentation Changes 1-16</li> </ul>	June 2009
-025	<ul style="list-style-type: none"> <li>Removed Documentation Changes 1-16</li> <li>Added Documentation Changes 1-18</li> </ul>	September 2009
-026	<ul style="list-style-type: none"> <li>Removed Documentation Changes 1-18</li> <li>Added Documentation Changes 1-15</li> </ul>	December 2009
-027	<ul style="list-style-type: none"> <li>Removed Documentation Changes 1-15</li> <li>Added Documentation Changes 1-24</li> </ul>	March 2010
-028	<ul style="list-style-type: none"> <li>Removed Documentation Changes 1-24</li> <li>Added Documentation Changes 1-29</li> </ul>	June 2010
-029	<ul style="list-style-type: none"> <li>Removed Documentation Changes 1-29</li> <li>Added Documentation Changes 1-29</li> </ul>	September 2010
-030	<ul style="list-style-type: none"> <li>Removed Documentation Changes 1-29</li> <li>Added Documentation Changes 1-29</li> </ul>	January 2011
-031	<ul style="list-style-type: none"> <li>Removed Documentation Changes 1-29</li> <li>Added Documentation Changes 1-29</li> </ul>	April 2011
-032	<ul style="list-style-type: none"> <li>Removed Documentation Changes 1-29</li> <li>Added Documentation Changes 1-14</li> </ul>	May 2011
-033	<ul style="list-style-type: none"> <li>Removed Documentation Changes 1-14</li> <li>Added Documentation Changes 1-38</li> </ul>	October 2011
-034	<ul style="list-style-type: none"> <li>Removed Documentation Changes 1-38</li> <li>Added Documentation Changes 1-16</li> </ul>	December 2011
-035	<ul style="list-style-type: none"> <li>Removed Documentation Changes 1-16</li> <li>Added Documentation Changes 1-18</li> </ul>	March 2012
-036	<ul style="list-style-type: none"> <li>Removed Documentation Changes 1-18</li> <li>Added Documentation Changes 1-17</li> </ul>	May 2012
-037	<ul style="list-style-type: none"> <li>Removed Documentation Changes 1-17</li> <li>Added Documentation Changes 1-28</li> </ul>	August 2012
-038	<ul style="list-style-type: none"> <li>Removed Documentation Changes 1-28</li> <li>Add Documentation Changes 1-22</li> </ul>	January 2013
-039	<ul style="list-style-type: none"> <li>Removed Documentation Changes 1-22</li> <li>Add Documentation Changes 1-17</li> </ul>	June 2013
-040	<ul style="list-style-type: none"> <li>Removed Documentation Changes 1-17</li> <li>Add Documentation Changes 1-24</li> </ul>	September 2013
-041	<ul style="list-style-type: none"> <li>Removed Documentation Changes 1-24</li> <li>Add Documentation Changes 1-20</li> </ul>	February 2014
-042	<ul style="list-style-type: none"> <li>Removed Documentation Changes 1-20</li> <li>Add Documentation Changes 1-8</li> </ul>	February 2014
-043	<ul style="list-style-type: none"> <li>Removed Documentation Changes 1-8</li> <li>Add Documentation Changes 1-43</li> </ul>	June 2014
-044	<ul style="list-style-type: none"> <li>Removed Documentation Changes 1-43</li> <li>Add Documentation Changes 1-12</li> </ul>	September 2014
-045	<ul style="list-style-type: none"> <li>Removed Documentation Changes 1-12</li> <li>Add Documentation Changes 1-22</li> </ul>	January 2015
-046	<ul style="list-style-type: none"> <li>Removed Documentation Changes 1-22</li> <li>Add Documentation Changes 1-25</li> </ul>	April 2015
-047	<ul style="list-style-type: none"> <li>Removed Documentation Changes 1-25</li> <li>Add Documentation Changes 1-19</li> </ul>	June 2015



Revision	Description	Date
-048	<ul style="list-style-type: none"><li>Removed Documentation Changes 1-19</li><li>Add Documentation Changes 1-33</li></ul>	September 2015
-049	<ul style="list-style-type: none"><li>Removed Documentation Changes 1-33</li><li>Add Documentation Changes 1-33</li></ul>	December 2015
-050	<ul style="list-style-type: none"><li>Removed Documentation Changes 1-33</li><li>Add Documentation Changes 1-9</li></ul>	April 2016
-051	<ul style="list-style-type: none"><li>Removed Documentation Changes 1-9</li><li>Add Documentation Changes 1-20</li></ul>	June 2016

§

# Preface

---

This document is an update to the specifications contained in the [Affected Documents](#) table below. This document is a compilation of device and documentation errata, specification clarifications and changes. It is intended for hardware system manufacturers and software developers of applications, operating systems, or tools.

## Affected Documents

Document Title	Document Number/ Location
<i>Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture</i>	253665
<i>Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A: Instruction Set Reference, A-L</i>	253666
<i>Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B: Instruction Set Reference, M-U</i>	253667
<i>Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2C: Instruction Set Reference, V-Z</i>	326018
<i>Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2D: Instruction Set Reference</i>	334569
<i>Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A: System Programming Guide, Part 1</i>	253668
<i>Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B: System Programming Guide, Part 2</i>	253669
<i>Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C: System Programming Guide, Part 3</i>	326019
<i>Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3D: System Programming Guide, Part 4</i>	332831

## Nomenclature

**Documentation Changes** include typos, errors, or omissions from the current published specifications. These will be incorporated in any new release of the specification.

# Summary Tables of Changes

---

The following table indicates documentation changes which apply to the Intel® 64 and IA-32 architectures. This table uses the following notations:

## Codes Used in Summary Tables

Change bar to left of table row indicates this erratum is either new or modified from the previous version of the document.

## Documentation Changes

No.	DOCUMENTATION CHANGES
1	Updates to Chapter 1, Volume 1
2	Updates to Chapter 5, Volume 1
3	Updates to Chapter 13, Volume 1
4	Updates to Chapter 15, Volume 1
5	Updates to Chapter 17, Volume 1
6	Updates to Chapter 1, Volume 2A
7	Updates to Chapter 2, Volume 2A
8	Updates to Chapter 3, Volume 2A
9	Updates to Chapter 4, Volume 2B
10	Updates to Chapter 5, Volume 2C
11	Updates to Appendix A, Volume 2D
12	Updates to Appendix C, Volume 2D
13	Updates to Chapter 8, Volume 3A
14	Updates to Chapter 9, Volume 3A
15	Updates to Chapter 18, Volume 3B
16	Updates to Chapter 22, Volume 3B
17	Updates to Chapter 29, Volume 3C
18	Updates to Chapter 34, Volume 3C
19	Updates to Chapter 35, Volume 3C
20	Updates to Chapter 36, Volume 3C

# *Documentation Changes*

---

Changes to the Intel® 64 and IA-32 Architectures Software Developer's Manual volumes follow, and are listed by chapter. Only chapters with changes are included in this document.



## 1. Updates to Chapter 1, Volume 1

Change bars show changes to Chapter 1 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture*.

-----

Changes to this chapter reflect the update to the [Intel® 64 and IA-32 Architectures Software Developer's Manual, Volumes 2A, 2B, 2C & 2D](#). This volume now includes Part D with ordering code.

The *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture* (order number 253665) is part of a set that describes the architecture and programming environment of Intel® 64 and IA-32 architecture processors. Other volumes in this set are:

- The *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volumes 2A, 2B & 2C: Instruction Set Reference* (order numbers 253666, 253667 and 326018).
- The *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volumes 3A, 3B, 3C & 3D: System Programming Guide* (order numbers 253668, 253669, 326019 and 332831).

The *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, describes the basic architecture and programming environment of Intel 64 and IA-32 processors. The *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volumes 2A, 2B & 2C*, describe the instruction set of the processor and the opcode structure. These volumes apply to application programmers and to programmers who write operating systems or executives. The *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volumes 3A, 3B, 3C & 3D*, describe the operating-system support environment of Intel 64 and IA-32 processors. These volumes target operating-system and BIOS designers. In addition, the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*, addresses the programming environment for classes of software that host operating systems.

## 1.1 INTEL® 64 AND IA-32 PROCESSORS COVERED IN THIS MANUAL

This manual set includes information pertaining primarily to the most recent Intel 64 and IA-32 processors, which include:

- Pentium® processors
- P6 family processors
- Pentium® 4 processors
- Pentium® M processors
- Intel® Xeon® processors
- Pentium® D processors
- Pentium® processor Extreme Editions
- 64-bit Intel® Xeon® processors
- Intel® Core™ Duo processor
- Intel® Core™ Solo processor
- Dual-Core Intel® Xeon® processor LV
- Intel® Core™2 Duo processor
- Intel® Core™2 Quad processor Q6000 series
- Intel® Xeon® processor 3000, 3200 series
- Intel® Xeon® processor 5000 series
- Intel® Xeon® processor 5100, 5300 series
- Intel® Core™2 Extreme processor X7000 and X6800 series
- Intel® Core™2 Extreme processor QX6000 series
- Intel® Xeon® processor 7100 series
- Intel® Pentium® Dual-Core processor
- Intel® Xeon® processor 7200, 7300 series
- Intel® Xeon® processor 5200, 5400, 7400 series



## ABOUT THIS MANUAL

- Intel® Core™2 Extreme processor QX9000 and X9000 series
- Intel® Core™2 Quad processor Q9000 series
- Intel® Core™2 Duo processor E8000, T9000 series
- Intel® Atom™ processor family
- Intel® Atom™ processors 200, 300, D400, D500, D2000, N200, N400, N2000, E2000, Z500, Z600, Z2000, C1000 series are built from 45 nm and 32 nm processes
- Intel® Core™ i7 processor
- Intel® Core™ i5 processor
- Intel® Xeon® processor E7-8800/4800/2800 product families
- Intel® Core™ i7-3930K processor
- 2nd generation Intel® Core™ i7-2xxx, Intel® Core™ i5-2xxx, Intel® Core™ i3-2xxx processor series
- Intel® Xeon® processor E3-1200 product family
- Intel® Xeon® processor E5-2400/1400 product family
- Intel® Xeon® processor E5-4600/2600/1600 product family
- 3rd generation Intel® Core™ processors
- Intel® Xeon® processor E3-1200 v2 product family
- Intel® Xeon® processor E5-2400/1400 v2 product families
- Intel® Xeon® processor E5-4600/2600/1600 v2 product families
- Intel® Xeon® processor E7-8800/4800/2800 v2 product families
- 4th generation Intel® Core™ processors
- The Intel® Core™ M processor family
- Intel® Core™ i7-59xx Processor Extreme Edition
- Intel® Core™ i7-49xx Processor Extreme Edition
- Intel® Xeon® processor E3-1200 v3 product family
- Intel® Xeon® processor E5-2600/1600 v3 product families
- 5th generation Intel® Core™ processors
- Intel® Xeon® processor D-1500 product family
- Intel® Xeon® processor E5 v4 family
- Intel® Atom™ processor X7-Z8000 and X5-Z8000 series
- Intel® Atom™ processor Z3400 series
- Intel® Atom™ processor Z3500 series
- 6th generation Intel® Core™ processors
- Intel® Xeon® processor E3-1500m v5 product family

P6 family processors are IA-32 processors based on the P6 family microarchitecture. This includes the Pentium® Pro, Pentium® II, Pentium® III, and Pentium® III Xeon® processors.

The Pentium® 4, Pentium® D, and Pentium® processor Extreme Editions are based on the Intel NetBurst® microarchitecture. Most early Intel® Xeon® processors are based on the Intel NetBurst® microarchitecture. Intel Xeon processor 5000, 7100 series are based on the Intel NetBurst® microarchitecture.

The Intel® Core™ Duo, Intel® Core™ Solo and dual-core Intel® Xeon® processor LV are based on an improved Pentium® M processor microarchitecture.

The Intel® Xeon® processor 3000, 3200, 5100, 5300, 7200 and 7300 series, Intel® Pentium® dual-core, Intel® Core™2 Duo, Intel® Core™2 Quad, and Intel® Core™2 Extreme processors are based on Intel® Core™ microarchitecture.

The Intel® Xeon® processor 5200, 5400, 7400 series, Intel® Core™2 Quad processor Q9000 series, and Intel® Core™2 Extreme processor QX9000, X9000 series, Intel® Core™2 processor E8000 series are based on Enhanced Intel® Core™ microarchitecture.

The Intel® Atom™ processors 200, 300, D400, D500, D2000, N200, N400, N2000, E2000, Z500, Z600, Z2000, C1000 series are based on the Intel® Atom™ microarchitecture and supports Intel 64 architecture.

The Intel® Core™ i7 processor and Intel® Xeon® processor 3400, 5500, 7500 series are based on 45 nm Intel® microarchitecture code name Nehalem. Intel® microarchitecture code name Westmere is a 32 nm version of Intel® microarchitecture code name Nehalem. Intel® Xeon® processor 5600 series, Intel Xeon processor E7 and various Intel Core i7, i5, i3 processors are based on Intel® microarchitecture code name Westmere. These processors support Intel 64 architecture.

The Intel® Xeon® processor E5 family, Intel® Xeon® processor E3-1200 family, Intel® Xeon® processor E7-8800/4800/2800 product families, Intel® Core™ i7-3930K processor, and 2nd generation Intel® Core™ i7-2xxx, Intel® Core™ i5-2xxx, Intel® Core™ i3-2xxx processor series are based on the Intel® microarchitecture code name Sandy Bridge and support Intel 64 architecture.

The Intel® Xeon® processor E7-8800/4800/2800 v2 product families, Intel® Xeon® processor E3-1200 v2 product family and the 3rd generation Intel® Core™ processors are based on the Intel® microarchitecture code name Ivy Bridge and support Intel 64 architecture.

The Intel® Xeon® processor E5-4600/2600/1600 v2 product families, Intel® Xeon® processor E5-2400/1400 v2 product families and Intel® Core™ i7-49xx Processor Extreme Edition are based on the Intel® microarchitecture code name Ivy Bridge-E and support Intel 64 architecture.

The Intel® Xeon® processor E3-1200 v3 product family and 4th Generation Intel® Core™ processors are based on the Intel® microarchitecture code name Haswell and support Intel 64 architecture.

The Intel® Core™ M processor family, 5th generation Intel® Core™ processors, Intel® Xeon® processor D-1500 product family and the Intel® Xeon® processor E5 v4 family are based on the Intel® microarchitecture code name Broadwell and support Intel 64 architecture.

The Intel® Xeon® processor E3-1500m v5 product family and 6th generation Intel® Core™ processors are based on the Intel® microarchitecture code name Skylake and support Intel 64 architecture.

The Intel® Xeon® processor E5-2600/1600 v3 product families and the Intel® Core™ i7-59xx Processor Extreme Edition are based on the Intel® microarchitecture code name Haswell-E and support Intel 64 architecture.

The Intel® Atom™ processor Z8000 series is based on the Intel microarchitecture code name Airmont.

The Intel® Atom™ processor Z3400 series and the Intel® Atom™ processor Z3500 series are based on the Intel microarchitecture code name Silvermont.

P6 family, Pentium® M, Intel® Core™ Solo, Intel® Core™ Duo processors, dual-core Intel® Xeon® processor LV, and early generations of Pentium 4 and Intel Xeon processors support IA-32 architecture. The Intel® Atom™ processor Z5xx series support IA-32 architecture.

The Intel® Xeon® processor 3000, 3200, 5000, 5100, 5200, 5300, 5400, 7100, 7200, 7300, 7400 series, Intel® Core™2 Duo, Intel® Core™2 Extreme processors, Intel Core 2 Quad processors, Pentium® D processors, Pentium® Dual-Core processor, newer generations of Pentium 4 and Intel Xeon processor family support Intel® 64 architecture.

IA-32 architecture is the instruction set architecture and programming environment for Intel's 32-bit microprocessors. Intel® 64 architecture is the instruction set architecture and programming environment which is the superset of Intel's 32-bit and 64-bit architectures. It is compatible with the IA-32 architecture.

## 1.2 OVERVIEW OF VOLUME 1: BASIC ARCHITECTURE

A description of this manual's content follows:

**Chapter 1 — About This Manual.** Gives an overview of all five volumes of the *Intel® 64 and IA-32 Architectures Software Developer's Manual*. It also describes the notational conventions in these manuals and lists related Intel manuals and documentation of interest to programmers and hardware designers.

**Chapter 2 — Intel® 64 and IA-32 Architectures.** Introduces the Intel 64 and IA-32 architectures along with the families of Intel processors that are based on these architectures. It also gives an overview of the common features found in these processors and brief history of the Intel 64 and IA-32 architectures.

**Chapter 3 — Basic Execution Environment.** Introduces the models of memory organization and describes the register set used by applications.

**Chapter 4 — Data Types.** Describes the data types and addressing modes recognized by the processor; provides an overview of real numbers and floating-point formats and of floating-point exceptions.

**Chapter 5 — Instruction Set Summary.** Lists all Intel 64 and IA-32 instructions, divided into technology groups.

**Chapter 6 — Procedure Calls, Interrupts, and Exceptions.** Describes the procedure stack and mechanisms provided for making procedure calls and for servicing interrupts and exceptions.

**Chapter 7 — Programming with General-Purpose Instructions.** Describes basic load and store, program control, arithmetic, and string instructions that operate on basic data types, general-purpose and segment registers; also describes system instructions that are executed in protected mode.

**Chapter 8 — Programming with the x87 FPU.** Describes the x87 floating-point unit (FPU), including floating-point registers and data types; gives an overview of the floating-point instruction set and describes the processor's floating-point exception conditions.

**Chapter 9 — Programming with Intel® MMX™ Technology.** Describes Intel MMX technology, including MMX registers and data types; also provides an overview of the MMX instruction set.

**Chapter 10 — Programming with Intel® Streaming SIMD Extensions (Intel® SSE).** Describes SSE extensions, including XMM registers, the MXCSR register, and packed single-precision floating-point data types; provides an overview of the SSE instruction set and gives guidelines for writing code that accesses the SSE extensions.

**Chapter 11 — Programming with Intel® Streaming SIMD Extensions 2 (Intel® SSE2).** Describes SSE2 extensions, including XMM registers and packed double-precision floating-point data types; provides an overview of the SSE2 instruction set and gives guidelines for writing code that accesses SSE2 extensions. This chapter also describes SIMD floating-point exceptions that can be generated with SSE and SSE2 instructions. It also provides general guidelines for incorporating support for SSE and SSE2 extensions into operating system and applications code.

**Chapter 12 — Programming with Intel® Streaming SIMD Extensions 3 (Intel® SSE3), Supplemental Streaming SIMD Extensions 3 (SSSE3), Intel® Streaming SIMD Extensions 4 (Intel® SSE4) and Intel® AES New Instructions (Intel® AESNI).** Provides an overview of the SSE3 instruction set, Supplemental SSE3, SSE4, AESNI instructions, and guidelines for writing code that accesses these extensions.

**Chapter 13 — Managing State Using the XSAVE Feature Set.** Describes the XSAVE feature set instructions and explains how software can enable the XSAVE feature set and XSAVE-enabled features.

**Chapter 14 — Programming with AVX, FMA and AVX2.** Provides an overview of the Intel® AVX instruction set, FMA and Intel AVX2 extensions and gives guidelines for writing code that accesses these extensions.

**Chapter 15 — Programming with Intel Transactional Synchronization Extensions.** Describes the instruction extensions that support lock elision techniques to improve the performance of multi-threaded software with contended locks.

**Chapter 16 — Input/Output.** Describes the processor's I/O mechanism, including I/O port addressing, I/O instructions, and I/O protection mechanisms.

**Chapter 17 — Processor Identification and Feature Determination.** Describes how to determine the CPU type and features available in the processor.

**Appendix A — EFLAGS Cross-Reference.** Summarizes how the IA-32 instructions affect the flags in the EFLAGS register.

**Appendix B — EFLAGS Condition Codes.** Summarizes how conditional jump, move, and 'byte set on condition code' instructions use condition code flags (OF, CF, ZF, SF, and PF) in the EFLAGS register.

**Appendix C — Floating-Point Exceptions Summary.** Summarizes exceptions raised by the x87 FPU floating-point and SSE/SSE2/SSE3 floating-point instructions.

**Appendix D — Guidelines for Writing x87 FPU Exception Handlers.** Describes how to design and write MS-DOS\* compatible exception handling facilities for FPU exceptions (includes software and hardware requirements

and assembly-language code examples). This appendix also describes general techniques for writing robust FPU exception handlers.

**Appendix E — Guidelines for Writing SIMD Floating-Point Exception Handlers.** Gives guidelines for writing exception handlers for exceptions generated by SSE/SSE2/SSE3 floating-point instructions.

## 1.3 NOTATIONAL CONVENTIONS

This manual uses specific notation for data-structure formats, for symbolic representation of instructions, and for hexadecimal and binary numbers. This notation is described below.

### 1.3.1 Bit and Byte Order

In illustrations of data structures in memory, smaller addresses appear toward the bottom of the figure; addresses increase toward the top. Bit positions are numbered from right to left. The numerical value of a set bit is equal to two raised to the power of the bit position. Intel 64 and IA-32 processors are “little endian” machines; this means the bytes of a word are numbered starting from the least significant byte. See Figure 1-1.

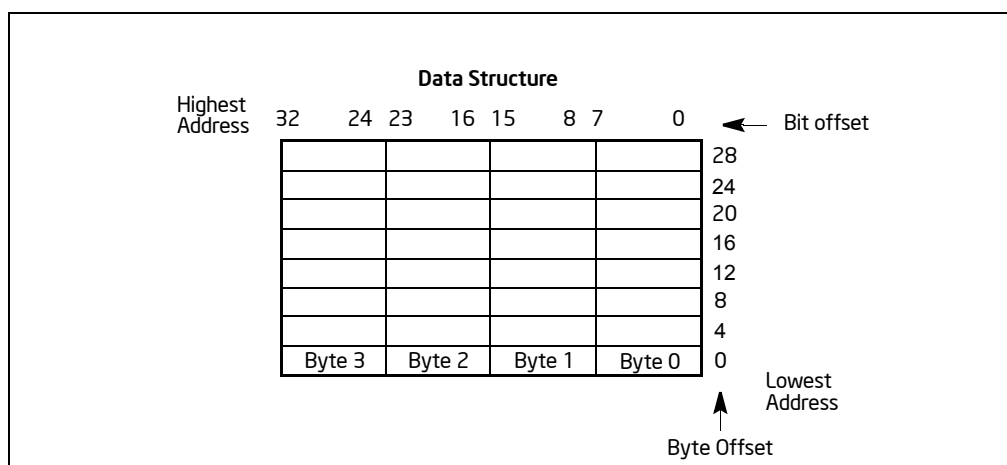


Figure 1-1. Bit and Byte Order

### 1.3.2 Reserved Bits and Software Compatibility

In many register and memory layout descriptions, certain bits are marked as **reserved**. When bits are marked as reserved, it is essential for compatibility with future processors that software treat these bits as having a future, though unknown, effect. The behavior of reserved bits should be regarded as not only undefined, but unpredictable.

Software should follow these guidelines in dealing with reserved bits:

- Do not depend on the states of any reserved bits when testing the values of registers that contain such bits. Mask out the reserved bits before testing.
- Do not depend on the states of any reserved bits when storing to memory or to a register.
- Do not depend on the ability to retain information written into any reserved bits.
- When loading a register, always load the reserved bits with the values indicated in the documentation, if any, or reload them with values previously read from the same register.

#### NOTE

Avoid any software dependence upon the state of reserved bits in Intel 64 and IA-32 registers. Depending upon the values of reserved register bits will make software dependent upon the

unspecified manner in which the processor handles these bits. Programs that depend upon reserved values risk incompatibility with future processors.

### 1.3.2.1 Instruction Operands

When instructions are represented symbolically, a subset of the IA-32 assembly language is used. In this subset, an instruction has the following format:

```
label: mnemonic argument1, argument2, argument3
```

where:

- A **label** is an identifier which is followed by a colon.
- A **mnemonic** is a reserved name for a class of instruction opcodes which have the same function.
- The operands **argument1**, **argument2**, and **argument3** are optional. There may be from zero to three operands, depending on the opcode. When present, they take the form of either literals or identifiers for data items. Operand identifiers are either reserved names of registers or are assumed to be assigned to data items declared in another part of the program (which may not be shown in the example).

When two operands are present in an arithmetic or logical instruction, the right operand is the source and the left operand is the destination.

For example:

```
LOADREG: MOV EAX, SUBTOTAL
```

In this example, LOADREG is a label, MOV is the mnemonic identifier of an opcode, EAX is the destination operand, and SUBTOTAL is the source operand. Some assembly languages put the source and destination in reverse order.

### 1.3.3 Hexadecimal and Binary Numbers

Base 16 (hexadecimal) numbers are represented by a string of hexadecimal digits followed by the character H (for example, 0F82EH). A hexadecimal digit is a character from the following set: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F.

Base 2 (binary) numbers are represented by a string of 1s and 0s, sometimes followed by the character B (for example, 1010B). The "B" designation is only used in situations where confusion as to the type of number might arise.

### 1.3.4 Segmented Addressing

The processor uses byte addressing. This means memory is organized and accessed as a sequence of bytes. Whether one or more bytes are being accessed, a byte address is used to locate the byte or bytes memory. The range of memory that can be addressed is called an **address space**.

The processor also supports segmented addressing. This is a form of addressing where a program may have many independent address spaces, called **segments**. For example, a program can keep its code (instructions) and stack in separate segments. Code addresses would always refer to the code space, and stack addresses would always refer to the stack space. The following notation is used to specify a byte address within a segment:

```
Segment-register:Byte-address
```

For example, the following segment address identifies the byte at address FF79H in the segment pointed by the DS register:

```
DS:FF79H
```

The following segment address identifies an instruction address in the code segment. The CS register points to the code segment and the EIP register contains the address of the instruction.

```
CS:EIP
```

### 1.3.5 A New Syntax for CPUID, CR, and MSR Values

Obtain feature flags, status, and system information by using the CPUID instruction, by checking control register bits, and by reading model-specific registers. We are moving toward a new syntax to represent this information. See Figure 1-2.

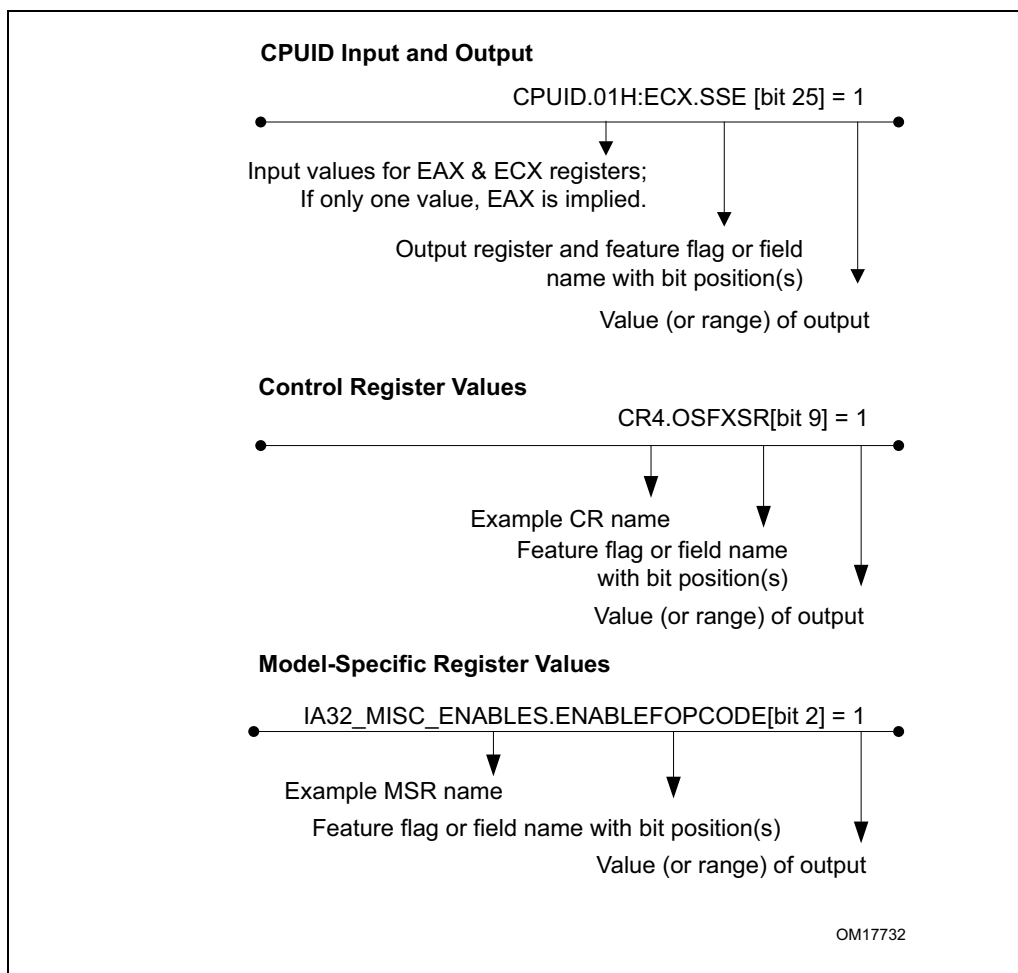


Figure 1-2. Syntax for CPUID, CR, and MSR Data Presentation

### 1.3.6 Exceptions

An exception is an event that typically occurs when an instruction causes an error. For example, an attempt to divide by zero generates an exception. However, some exceptions, such as breakpoints, occur under other conditions. Some types of exceptions may provide error codes. An error code reports additional information about the error. An example of the notation used to show an exception and error code is shown below:

#PF(fault code)

This example refers to a page-fault exception under conditions where an error code naming a type of fault is reported. Under some conditions, exceptions that produce error codes may not be able to report an accurate code. In this case, the error code is zero, as shown below for a general-protection exception:

#GP(0)

## 1.4 RELATED LITERATURE

Literature related to Intel 64 and IA-32 processors is listed and viewable on-line at:

<http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html>

See also:

- The data sheet for a particular Intel 64 or IA-32 processor
- The specification update for a particular Intel 64 or IA-32 processor
- Intel® C++ Compiler documentation and online help:  
<http://software.intel.com/en-us/articles/intel-compilers/>
- Intel® Fortran Compiler documentation and online help:  
<http://software.intel.com/en-us/articles/intel-compilers/>
- Intel® Software Development Tools:  
<http://www.intel.com/cd/software/products/asmo-na/eng/index.htm>
- Intel® 64 and IA-32 Architectures Software Developer's Manual (in three or seven volumes):  
<http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html>
- Intel® 64 and IA-32 Architectures Optimization Reference Manual:  
<http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-optimization-manual.html>
- Intel 64 Architecture x2APIC Specification:  
<http://www.intel.com/content/www/us/en/architecture-and-technology/64-architecture-x2apic-specification.html>
- Intel® Trusted Execution Technology Measured Launched Environment Programming Guide:  
<http://www.intel.com/content/www/us/en/software-developers/intel-txt-software-development-guide.html>
- Developing Multi-threaded Applications: A Platform Consistent Approach:  
<https://software.intel.com/sites/default/files/article/147714/51534-developing-multithreaded-applications.pdf>
- Using Spin-Loops on Intel® Pentium® 4 Processor and Intel® Xeon® Processor:  
<http://software.intel.com/en-us/articles/ap949-using-spin-loops-on-intel-pentiumr-4-processor-and-intel-xeonr-processor/>
- Performance Monitoring Unit Sharing Guide  
<http://software.intel.com/file/30388>

Literature related to selected features in future Intel processors are available at:

- Intel® Architecture Instruction Set Extensions Programming Reference  
<https://software.intel.com/en-us/isa-extensions>
- Intel® Software Guard Extensions (Intel® SGX) Programming Reference  
<https://software.intel.com/en-us/isa-extensions/intel-sgx>

More relevant links are:

- Intel® Developer Zone:  
<https://software.intel.com/en-us>
- Developer centers:  
<http://www.intel.com/content/www/us/en/hardware-developers/developer-centers.html>
- Processor support general link:  
<http://www.intel.com/support/processors/>
- Software products and packages:  
<http://www.intel.com/cd/software/products/asmo-na/eng/index.htm>
- Intel® Hyper-Threading Technology (Intel® HT Technology):  
<http://www.intel.com/technology/platform-technology/hyper-threading/index.htm>

## 2. Updates to Chapter 5, Volume 1

Change bars show changes to Chapter 5 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture*.

-----

Changes include the addition of Intel® SHA Extensions and Intel® AVX-512 Instructions.



## CHAPTER 5 INSTRUCTION SET SUMMARY

This chapter provides an abridged overview of Intel 64 and IA-32 instructions. Instructions are divided into the following groups:

- General purpose
- x87 FPU
- x87 FPU and SIMD state management
- Intel® MMX technology
- SSE extensions
- SSE2 extensions
- SSE3 extensions
- SSSE3 extensions
- SSE4 extensions
- AESNI and PCLMULQDQ
- Intel® AVX extensions
- F16C, RDRAND, RDSEED, FS/GS base access
- FMA extensions
- Intel® AVX2 extensions
- Intel® Transactional Synchronization extensions
- System instructions
- IA-32e mode: 64-bit mode instructions
- VMX instructions
- SMX instructions
- ADCX and ADOX
- Intel® Memory Protection Extensions
- Intel® Security Guard Extensions

Table 5-1 lists the groups and IA-32 processors that support each group. More recent instruction set extensions are listed in Table 5-2. Within these groups, most instructions are collected into functional subgroups.

**Table 5-1. Instruction Groups in Intel 64 and IA-32 Processors**

Instruction Set Architecture	Intel 64 and IA-32 Processor Support
General Purpose	All Intel 64 and IA-32 processors.
x87 FPU	Intel486, Pentium, Pentium with MMX Technology, Celeron, Pentium Pro, Pentium II, Pentium II Xeon, Pentium III, Pentium III Xeon, Pentium 4, Intel Xeon processors, Pentium M, Intel Core Solo, Intel Core Duo, Intel Core 2 Duo processors, Intel Atom processors.
x87 FPU and SIMD State Management	Pentium II, Pentium II Xeon, Pentium III, Pentium III Xeon, Pentium 4, Intel Xeon processors, Pentium M, Intel Core Solo, Intel Core Duo, Intel Core 2 Duo processors, Intel Atom processors.
MMX Technology	Pentium with MMX Technology, Celeron, Pentium II, Pentium II Xeon, Pentium III, Pentium III Xeon, Pentium 4, Intel Xeon processors, Pentium M, Intel Core Solo, Intel Core Duo, Intel Core 2 Duo processors, Intel Atom processors.
SSE Extensions	Pentium III, Pentium III Xeon, Pentium 4, Intel Xeon processors, Pentium M, Intel Core Solo, Intel Core Duo, Intel Core 2 Duo processors, Intel Atom processors.

**Table 5-1. Instruction Groups in Intel 64 and IA-32 Processors (Contd.)**

Instruction Set Architecture	Intel 64 and IA-32 Processor Support
SSE2 Extensions	Pentium 4, Intel Xeon processors, Pentium M, Intel Core Solo, Intel Core Duo, Intel Core 2 Duo processors, Intel Atom processors.
SSE3 Extensions	Pentium 4 supporting HT Technology (built on 90nm process technology), Intel Core Solo, Intel Core Duo, Intel Core 2 Duo processors, Intel Xeon processor 3xxxx, 5xxx, 7xxx Series, Intel Atom processors.
SSSE3 Extensions	Intel Xeon processor 3xxx, 5100, 5200, 5300, 5400, 5500, 5600, 7300, 7400, 7500 series, Intel Core 2 Extreme processors QX6000 series, Intel Core 2 Duo, Intel Core 2 Quad processors, Intel Pentium Dual-Core processors, Intel Atom processors.
IA-32e mode: 64-bit mode instructions	Intel 64 processors.
System Instructions	Intel 64 and IA-32 processors.
VMX Instructions	Intel 64 and IA-32 processors supporting Intel Virtualization Technology.
SMX Instructions	Intel Core 2 Duo processor E6x50, E8xxx; Intel Core 2 Quad processor Q9xxx.

**Table 5-2. Recent Instruction Set Extensions Introduction in Intel 64 and IA-32 Processors**

Instruction Set Architecture	Processor Generation Introduction
SSE4.1 Extensions	Intel Xeon processor 3100, 3300, 5200, 5400, 7400, 7500 series, Intel Core 2 Extreme processors QX9000 series, Intel Core 2 Quad processor Q9000 series, Intel Core 2 Duo processors 8000 series, T9000 series.
SSE4.2 Extensions, CRC32, POPCNT	Intel Core i7 965 processor, Intel Xeon processors X3400, X3500, X5500, X6500, X7500 series.
AESNI, PCLMULQDQ	Intel Xeon processor E7 series, Intel Xeon processors X3600, X5600, Intel Core i7 980X processor; Use CPUID to verify presence of AESNI and PCLMULQDQ across Intel Core processor families.
Intel AVX	Intel Xeon processor E3 and E5 families; 2nd Generation Intel Core i7, i5, i3 processor 2xxx families.
F16C, RDRAND, FS/GS base access	3rd Generation Intel Core processors, Intel Xeon processor E3-1200 v2 product family, Next Generation Intel Xeon processors, Intel Xeon processor E5 v2 and E7 v2 families.
FMA, AVX2, BMI1, BMI2, INVPCID	Intel Xeon processor E3-1200 v3 product family; 4th Generation Intel Core processor family.
TSX	Intel Xeon processor E7 v3 product family.
ADX, RDSEED, CLAC, STAC	Intel Core M processor family; 5th Generation Intel Core processor family.
CLFLUSHOPT, XSAVEC, XSAVES, MPX, SGX1	6th Generation Intel Core processor family.

The following sections list instructions in each major group and subgroup. Given for each instruction is its mnemonic and descriptive names. When two or more mnemonics are given (for example, CMOVA/CMOVNBE), they represent different mnemonics for the same instruction opcode. Assemblers support redundant mnemonics for some instructions to make it easier to read code listings. For instance, CMOVA (Conditional move if above) and CMOVNBE (Conditional move if not below or equal) represent the same condition. For detailed information about specific instructions, see the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volumes 2A, 2B & 2C*.

## 5.1 GENERAL-PURPOSE INSTRUCTIONS

The general-purpose instructions perform basic data movement, arithmetic, logic, program flow, and string operations that programmers commonly use to write application and system software to run on Intel 64 and IA-32 processors. They operate on data contained in memory, in the general-purpose registers (EAX, EBX, ECX, EDX, EDI, ESI, EBP, and ESP) and in the EFLAGS register. They also operate on address information contained in memory, the general-purpose registers, and the segment registers (CS, DS, SS, ES, FS, and GS).

This group of instructions includes the data transfer, binary integer arithmetic, decimal arithmetic, logic operations, shift and rotate, bit and byte operations, program control, string, flag control, segment register operations, and miscellaneous subgroups. The sections that following introduce each subgroup.

For more detailed information on general purpose-instructions, see Chapter 7, "Programming With General-Purpose Instructions."

### 5.1.1 Data Transfer Instructions

The data transfer instructions move data between memory and the general-purpose and segment registers. They also perform specific operations such as conditional moves, stack access, and data conversion.

MOV	Move data between general-purpose registers; move data between memory and general-purpose or segment registers; move immediates to general-purpose registers.
CMOVE/CMOVZ	Conditional move if equal/Conditional move if zero.
CMOVNE/CMOVNZ	Conditional move if not equal/Conditional move if not zero.
CMOVA/CMOVNBE	Conditional move if above/Conditional move if not below or equal.
CMOVAE/CMOVNB	Conditional move if above or equal/Conditional move if not below.
CMOVB/CMOVNAE	Conditional move if below/Conditional move if not above or equal.
CMOVBE/CMOVNA	Conditional move if below or equal/Conditional move if not above.
CMOVG/CMOVNLE	Conditional move if greater/Conditional move if not less or equal.
CMOVGE/CMOVNL	Conditional move if greater or equal/Conditional move if not less.
CMOVL/CMOVNGE	Conditional move if less/Conditional move if not greater or equal.
CMOVLE/CMOVNG	Conditional move if less or equal/Conditional move if not greater.
CMOVC	Conditional move if carry.
CMOVNC	Conditional move if not carry.
CMOVO	Conditional move if overflow.
CMOVNO	Conditional move if not overflow.
CMOVS	Conditional move if sign (negative).
CMOVNS	Conditional move if not sign (non-negative).
CMOVP/CMOVPE	Conditional move if parity/Conditional move if parity even.
CMOVNP/CMOVPO	Conditional move if not parity/Conditional move if parity odd.
XCHG	Exchange.
BSWAP	Byte swap.
XADD	Exchange and add.
CMPXCHG	Compare and exchange.
CMPXCHG8B	Compare and exchange 8 bytes.
PUSH	Push onto stack.
POP	Pop off of stack.
PUSHA/PUSHAD	Push general-purpose registers onto stack.
POPA/POPAD	Pop general-purpose registers from stack.
CWD/CDQ	Convert word to doubleword/Convert doubleword to quadword.
CBW/CWDE	Convert byte to word/Convert word to doubleword in EAX register.
MOVSX	Move and sign extend.

MOVZX                      Move and zero extend.

### 5.1.2      Binary Arithmetic Instructions

The binary arithmetic instructions perform basic binary integer computations on byte, word, and doubleword integers located in memory and/or the general purpose registers.

ADCX	Unsigned integer add with carry.
ADOX	Unsigned integer add with overflow.
ADD	Integer add.
ADC	Add with carry.
SUB	Subtract.
SBB	Subtract with borrow.
IMUL	Signed multiply.
MUL	Unsigned multiply.
IDIV	Signed divide.
DIV	Unsigned divide.
INC	Increment.
DEC	Decrement.
NEG	Negate.
CMP	Compare.

### 5.1.3      Decimal Arithmetic Instructions

The decimal arithmetic instructions perform decimal arithmetic on binary coded decimal (BCD) data.

DAA	Decimal adjust after addition.
DAS	Decimal adjust after subtraction.
AAA	ASCII adjust after addition.
AAS	ASCII adjust after subtraction.
AAM	ASCII adjust after multiplication.
AAD	ASCII adjust before division.

### 5.1.4      Logical Instructions

The logical instructions perform basic AND, OR, XOR, and NOT logical operations on byte, word, and doubleword values.

AND	Perform bitwise logical AND.
OR	Perform bitwise logical OR.
XOR	Perform bitwise logical exclusive OR.
NOT	Perform bitwise logical NOT.

### 5.1.5      Shift and Rotate Instructions

The shift and rotate instructions shift and rotate the bits in word and doubleword operands.

SAR	Shift arithmetic right.
SHR	Shift logical right.
SAL/SHL	Shift arithmetic left/Shift logical left.
SHRD	Shift right double.

SHLD	Shift left double.
ROR	Rotate right.
ROL	Rotate left.
RCR	Rotate through carry right.
RCL	Rotate through carry left.

### 5.1.6 Bit and Byte Instructions

Bit instructions test and modify individual bits in word and doubleword operands. Byte instructions set the value of a byte operand to indicate the status of flags in the EFLAGS register.

BT	Bit test.
BTS	Bit test and set.
BTR	Bit test and reset.
BTC	Bit test and complement.
BSF	Bit scan forward.
BSR	Bit scan reverse.
SETE/SETZ	Set byte if equal/Set byte if zero.
SETNE/SETNZ	Set byte if not equal/Set byte if not zero.
SETA/SETNBE	Set byte if above/Set byte if not below or equal.
SETAE/SETNB/SETNC	Set byte if above or equal/Set byte if not below/Set byte if not carry.
SETB/SETNAE/SETC	Set byte if below/Set byte if not above or equal/Set byte if carry.
SETBE/SETNA	Set byte if below or equal/Set byte if not above.
SETG/SETNLE	Set byte if greater/Set byte if not less or equal.
SETGE/SETNL	Set byte if greater or equal/Set byte if not less.
SETL/SETNGE	Set byte if less/Set byte if not greater or equal.
SETLE/SETNG	Set byte if less or equal/Set byte if not greater.
SETS	Set byte if sign (negative).
SETNS	Set byte if not sign (non-negative).
SETO	Set byte if overflow.
SETNO	Set byte if not overflow.
SETPE/SETP	Set byte if parity even/Set byte if parity.
SETPO/SETNP	Set byte if parity odd/Set byte if not parity.
TEST	Logical compare.
CRC32 <sup>1</sup>	Provides hardware acceleration to calculate cyclic redundancy checks for fast and efficient implementation of data integrity protocols.
POPCNT <sup>2</sup>	This instruction calculates of number of bits set to 1 in the second operand (source) and returns the count in the first operand (a destination register).

### 5.1.7 Control Transfer Instructions

The control transfer instructions provide jump, conditional jump, loop, and call and return operations to control program flow.

JMP	Jump.
JE/JZ	Jump if equal/Jump if zero.
JNE/JNZ	Jump if not equal/Jump if not zero.

1. Processor support of CRC32 is enumerated by CPUID.01:ECX[SSE4.2] = 1
2. Processor support of POPCNT is enumerated by CPUID.01:ECX[POPCNT] = 1

JA/JNBE	Jump if above/Jump if not below or equal.
JAE/JNB	Jump if above or equal/Jump if not below.
JB/JNAE	Jump if below/Jump if not above or equal.
JBE/JNA	Jump if below or equal/Jump if not above.
JG/JNLE	Jump if greater/Jump if not less or equal.
JGE/JNL	Jump if greater or equal/Jump if not less.
JL/JNGE	Jump if less/Jump if not greater or equal.
JLE/JNG	Jump if less or equal/Jump if not greater.
JC	Jump if carry.
JNC	Jump if not carry.
JO	Jump if overflow.
JNO	Jump if not overflow.
JS	Jump if sign (negative).
JNS	Jump if not sign (non-negative).
JPO/JNP	Jump if parity odd/Jump if not parity.
JPE/JP	Jump if parity even/Jump if parity.
JCXZ/JECXZ	Jump register CX zero/Jump register ECX zero.
LOOP	Loop with ECX counter.
LOOPZ/LOOPE	Loop with ECX and zero/Loop with ECX and equal.
LOOPNZ/LOOPNE	Loop with ECX and not zero/Loop with ECX and not equal.
CALL	Call procedure.
RET	Return.
IRET	Return from interrupt.
INT	Software interrupt.
INTO	Interrupt on overflow.
BOUND	Detect value out of range.
ENTER	High-level procedure entry.
LEAVE	High-level procedure exit.

### 5.1.8 String Instructions

The string instructions operate on strings of bytes, allowing them to be moved to and from memory.

MOVS/MOVS	Move string/Move byte string.
MOVS/MOVSW	Move string/Move word string.
MOVS/MOVSD	Move string/Move doubleword string.
CMPS/CMPSB	Compare string/Compare byte string.
CMPS/CMPSW	Compare string/Compare word string.
CMPS/CMPSD	Compare string/Compare doubleword string.
SCAS/SCASB	Scan string/Scan byte string.
SCAS/SCASW	Scan string/Scan word string.
SCAS/SCASD	Scan string/Scan doubleword string.
LODS/LODSB	Load string/Load byte string.
LODS/LODSW	Load string/Load word string.
LODS/LODSD	Load string/Load doubleword string.
STOS/STOSB	Store string/Store byte string.
STOS/STOSW	Store string/Store word string.

STOS/STOSD	Store string/Store doubleword string.
REP	Repeat while ECX not zero.
REPE/REPZ	Repeat while equal/Repeat while zero.
REPNE/REPNZ	Repeat while not equal/Repeat while not zero.

### 5.1.9 I/O Instructions

These instructions move data between the processor's I/O ports and a register or memory.

IN	Read from a port.
OUT	Write to a port.
INS/INSB	Input string from port/Input byte string from port.
INS/INSW	Input string from port/Input word string from port.
INS/INSD	Input string from port/Input doubleword string from port.
OUTS/OUTSB	Output string to port/Output byte string to port.
OUTS/OUTSW	Output string to port/Output word string to port.
OUTS/OUTSD	Output string to port/Output doubleword string to port.

### 5.1.10 Enter and Leave Instructions

These instructions provide machine-language support for procedure calls in block-structured languages.

ENTER	High-level procedure entry.
LEAVE	High-level procedure exit.

### 5.1.11 Flag Control (EFLAG) Instructions

The flag control instructions operate on the flags in the EFLAGS register.

STC	Set carry flag.
CLC	Clear the carry flag.
CMC	Complement the carry flag.
CLD	Clear the direction flag.
STD	Set direction flag.
LAHF	Load flags into AH register.
SAHF	Store AH register into flags.
PUSHF/PUSHFD	Push EFLAGS onto stack.
POPF/POPF	Pop EFLAGS from stack.
STI	Set interrupt flag.
CLI	Clear the interrupt flag.

### 5.1.12 Segment Register Instructions

The segment register instructions allow far pointers (segment addresses) to be loaded into the segment registers.

LDS	Load far pointer using DS.
LES	Load far pointer using ES.
LFS	Load far pointer using FS.
LGS	Load far pointer using GS.
LSS	Load far pointer using SS.

### 5.1.13 Miscellaneous Instructions

The miscellaneous instructions provide such functions as loading an effective address, executing a “no-operation,” and retrieving processor identification information.

LEA	Load effective address.
NOP	No operation.
UD2	Undefined instruction.
XLAT/XLATB	Table lookup translation.
CPUID	Processor identification.
MOVBE <sup>1</sup>	Move data after swapping data bytes.
PREFETCHW	Prefetch data into cache in anticipation of write.
PREFETCHWT1	Prefetch hint T1 with intent to write.
CLFLUSH	Flushes and invalidates a memory operand and its associated cache line from all levels of the processor’s cache hierarchy.
CLFLUSHOPT	Flushes and invalidates a memory operand and its associated cache line from all levels of the processor’s cache hierarchy with optimized memory system throughput.

### 5.1.14 User Mode Extended State Save/Restore Instructions

XSAVE	Save processor extended states to memory.
XSAVEC	Save processor extended states with compaction to memory.
XSAVEOPT	Save processor extended states to memory, optimized.
XRSTOR	Restore processor extended states from memory.
XGETBV	Reads the state of an extended control register.

### 5.1.15 Random Number Generator Instructions

RDRAND	Retrieves a random number generated from hardware.
RDSEED	Retrieves a random number generated from hardware.

### 5.1.16 BMI1, BMI2

ANDN	Bitwise AND of first source with inverted 2nd source operands.
BEXTR	Contiguous bitwise extract.
BLSI	Extract lowest set bit.
BLSMSK	Set all lower bits below first set bit to 1.
BLSR	Reset lowest set bit.
BZHI	Zero high bits starting from specified bit position.
LZCNT	Count the number leading zero bits.
MULX	Unsigned multiply without affecting arithmetic flags.
PDEP	Parallel deposit of bits using a mask.
PEXT	Parallel extraction of bits using a mask.
RORX	Rotate right without affecting arithmetic flags.
SARX	Shift arithmetic right.
SHLX	Shift logic left.
SHRX	Shift logic right.
TZCNT	Count the number trailing zero bits.

1. Processor support of MOVBE is enumerated by CPUID.01:ECX.MOVBE[bit 22] = 1.



### 5.1.16.1 Detection of VEX-encoded GPR Instructions, LZCNT and TZCNT, PREFETCHW

VEX-encoded general-purpose instructions do not operate on any vector registers.

There are separate feature flags for the following subsets of instructions that operate on general purpose registers, and the detection requirements for hardware support are:

CPUID.(EAX=07H, ECX=0H):EBX.BMI1[bit 3]: if 1 indicates the processor supports the first group of advanced bit manipulation extensions (ANDN, BEXTR, BLSI, BLSMSK, BLSR, TZCNT);

CPUID.(EAX=07H, ECX=0H):EBX.BMI2[bit 8]: if 1 indicates the processor supports the second group of advanced bit manipulation extensions (BZHI, MULX, PDEP, PEXT, RORX, SARX, SHLX, SHRX);

CPUID.EAX=8000001H:ECX.LZCNT[bit 5]: if 1 indicates the processor supports the LZCNT instruction.

CPUID.EAX=8000001H:ECX.PREFTEHCHW[bit 8]: if 1 indicates the processor supports the PREFTEHCHW instruction. CPUID.(EAX=07H, ECX=0H):ECX.PREFTEHCHWT1[bit 0]: if 1 indicates the processor supports the PREFTEHCHWT1 instruction.

## 5.2 X87 FPU INSTRUCTIONS

The x87 FPU instructions are executed by the processor's x87 FPU. These instructions operate on floating-point, integer, and binary-coded decimal (BCD) operands. For more detail on x87 FPU instructions, see Chapter 8, "Programming with the x87 FPU."

These instructions are divided into the following subgroups: data transfer, load constants, and FPU control instructions. The sections that follow introduce each subgroup.

### 5.2.1 x87 FPU Data Transfer Instructions

The data transfer instructions move floating-point, integer, and BCD values between memory and the x87 FPU registers. They also perform conditional move operations on floating-point operands.

FLD	Load floating-point value.
FST	Store floating-point value.
FSTP	Store floating-point value and pop.
FILD	Load integer.
FIST	Store integer.
FISTP <sup>1</sup>	Store integer and pop.
FBLD	Load BCD.
FBSTP	Store BCD and pop.
FXCH	Exchange registers.
FCMOVE	Floating-point conditional move if equal.
FCMOVNE	Floating-point conditional move if not equal.
FCMOVB	Floating-point conditional move if below.
FCMOVBE	Floating-point conditional move if below or equal.
FCMOVNB	Floating-point conditional move if not below.
FCMOVNBE	Floating-point conditional move if not below or equal.
FCMOVU	Floating-point conditional move if unordered.
FCMOVNU	Floating-point conditional move if not unordered.

### 5.2.2 x87 FPU Basic Arithmetic Instructions

The basic arithmetic instructions perform basic arithmetic operations on floating-point and integer operands.

1. SSE3 provides an instruction FISTTP for integer conversion.

FADD	Add floating-point
FADDP	Add floating-point and pop
FIADD	Add integer
FSUB	Subtract floating-point
FSUBP	Subtract floating-point and pop
FISUB	Subtract integer
FSUBR	Subtract floating-point reverse
FSUBRP	Subtract floating-point reverse and pop
FISUBR	Subtract integer reverse
FMUL	Multiply floating-point
FMULP	Multiply floating-point and pop
FIMUL	Multiply integer
FDIV	Divide floating-point
FDIVP	Divide floating-point and pop
FIDIV	Divide integer
FDIVR	Divide floating-point reverse
FDIVRP	Divide floating-point reverse and pop
FIDIVR	Divide integer reverse
FPREM	Partial remainder
FPREM1	IEEE Partial remainder
FABS	Absolute value
FCHS	Change sign
FRNDINT	Round to integer
FSCALE	Scale by power of two
FSQRT	Square root
FXTRACT	Extract exponent and significand

### 5.2.3 x87 FPU Comparison Instructions

The compare instructions examine or compare floating-point or integer operands.

FCOM	Compare floating-point.
FCOMP	Compare floating-point and pop.
FCOMPP	Compare floating-point and pop twice.
FUCOM	Unordered compare floating-point.
FUCOMP	Unordered compare floating-point and pop.
FUCOMPP	Unordered compare floating-point and pop twice.
FICOM	Compare integer.
FICOMP	Compare integer and pop.
FCOMI	Compare floating-point and set EFLAGS.
FUCOMI	Unordered compare floating-point and set EFLAGS.
FCOMIP	Compare floating-point, set EFLAGS, and pop.
FUCOMIP	Unordered compare floating-point, set EFLAGS, and pop.
FTST	Test floating-point (compare with 0.0).
FXAM	Examine floating-point.

## 5.2.4 x87 FPU Transcendental Instructions

The transcendental instructions perform basic trigonometric and logarithmic operations on floating-point operands.

FSIN	Sine
FCOS	Cosine
FSINCOS	Sine and cosine
FPTAN	Partial tangent
FPATAN	Partial arctangent
F2XM1	$2^x - 1$
FYL2X	$y * \log_2 x$
FYL2XP1	$y * \log_2(x+1)$

## 5.2.5 x87 FPU Load Constants Instructions

The load constants instructions load common constants, such as  $\pi$ , into the x87 floating-point registers.

FLD1	Load +1.0
FLDZ	Load +0.0
FLDPI	Load $\pi$
FLDL2E	Load $\log_2 e$
FLDLN2	Load $\log_e 2$
FLDL2T	Load $\log_2 10$
FLDLG2	Load $\log_{10} 2$

## 5.2.6 x87 FPU Control Instructions

The x87 FPU control instructions operate on the x87 FPU register stack and save and restore the x87 FPU state.

FINCSTP	Increment FPU register stack pointer.
FDECSTP	Decrement FPU register stack pointer.
FFREE	Free floating-point register.
FINIT	Initialize FPU after checking error conditions.
FNINIT	Initialize FPU without checking error conditions.
FCLEX	Clear floating-point exception flags after checking for error conditions.
FNCLEX	Clear floating-point exception flags without checking for error conditions.
FSTCW	Store FPU control word after checking error conditions.
FNSTCW	Store FPU control word without checking error conditions.
FLDCW	Load FPU control word.
FSTENV	Store FPU environment after checking error conditions.
FNSTENV	Store FPU environment without checking error conditions.
FLDENV	Load FPU environment.
FSAVE	Save FPU state after checking error conditions.
FNSAVE	Save FPU state without checking error conditions.
FRSTOR	Restore FPU state.
FSTSW	Store FPU status word after checking error conditions.
FNSTSW	Store FPU status word without checking error conditions.
WAIT/FWAIT	Wait for FPU.
FNOP	FPU no operation.

## 5.3 X87 FPU AND SIMD STATE MANAGEMENT INSTRUCTIONS

Two state management instructions were introduced into the IA-32 architecture with the Pentium II processor family:

FXSAVE	Save x87 FPU and SIMD state.
FXRSTOR	Restore x87 FPU and SIMD state.

Initially, these instructions operated only on the x87 FPU (and MMX) registers to perform a fast save and restore, respectively, of the x87 FPU and MMX state. With the introduction of SSE extensions in the Pentium III processor family, these instructions were expanded to also save and restore the state of the XMM and MXCSR registers. Intel 64 architecture also supports these instructions.

See Section 10.5, “FXSAVE and FXRSTOR Instructions,” for more detail.

## 5.4 MMX™ INSTRUCTIONS

Four extensions have been introduced into the IA-32 architecture to permit IA-32 processors to perform single-instruction multiple-data (SIMD) operations. These extensions include the MMX technology, SSE extensions, SSE2 extensions, and SSE3 extensions. For a discussion that puts SIMD instructions in their historical context, see Section 2.2.7, “SIMD Instructions.”

MMX instructions operate on packed byte, word, doubleword, or quadword integer operands contained in memory, in MMX registers, and/or in general-purpose registers. For more detail on these instructions, see Chapter 9, “Programming with Intel® MMX™ Technology.”

MMX instructions can only be executed on Intel 64 and IA-32 processors that support the MMX technology. Support for these instructions can be detected with the CPUID instruction. See the description of the CPUID instruction in Chapter 3, “Instruction Set Reference, A-M,” of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A*.

MMX instructions are divided into the following subgroups: data transfer, conversion, packed arithmetic, comparison, logical, shift and rotate, and state management instructions. The sections that follow introduce each subgroup.

### 5.4.1 MMX Data Transfer Instructions

The data transfer instructions move doubleword and quadword operands between MMX registers and between MMX registers and memory.

MOVD	Move doubleword.
MOVQ	Move quadword.

### 5.4.2 MMX Conversion Instructions

The conversion instructions pack and unpack bytes, words, and doublewords

PACKSSWB	Pack words into bytes with signed saturation.
PACKSSDW	Pack doublewords into words with signed saturation.
PACKUSWB	Pack words into bytes with unsigned saturation.
PUNPCKHBW	Unpack high-order bytes.
PUNPCKHWD	Unpack high-order words.
PUNPCKHDQ	Unpack high-order doublewords.
PUNPCKLBW	Unpack low-order bytes.
PUNPCKLWD	Unpack low-order words.
PUNPCKLDQ	Unpack low-order doublewords.

### 5.4.3 MMX Packed Arithmetic Instructions

The packed arithmetic instructions perform packed integer arithmetic on packed byte, word, and doubleword integers.

PADDB	Add packed byte integers.
PADDW	Add packed word integers.
PADDD	Add packed doubleword integers.
PADDSB	Add packed signed byte integers with signed saturation.
PADDSW	Add packed signed word integers with signed saturation.
PADDUSB	Add packed unsigned byte integers with unsigned saturation.
PADDUSW	Add packed unsigned word integers with unsigned saturation.
PSUBB	Subtract packed byte integers.
PSUBW	Subtract packed word integers.
PSUBD	Subtract packed doubleword integers.
PSUBSB	Subtract packed signed byte integers with signed saturation.
PSUBSW	Subtract packed signed word integers with signed saturation.
PSUBUSB	Subtract packed unsigned byte integers with unsigned saturation.
PSUBUSW	Subtract packed unsigned word integers with unsigned saturation.
PMULHW	Multiply packed signed word integers and store high result.
PMULLW	Multiply packed signed word integers and store low result.
PMADDWD	Multiply and add packed word integers.

### 5.4.4 MMX Comparison Instructions

The compare instructions compare packed bytes, words, or doublewords.

PCMPEQB	Compare packed bytes for equal.
PCMPEQW	Compare packed words for equal.
PCMPEQD	Compare packed doublewords for equal.
PCMPGTB	Compare packed signed byte integers for greater than.
PCMPGTW	Compare packed signed word integers for greater than.
PCMPGTD	Compare packed signed doubleword integers for greater than.

### 5.4.5 MMX Logical Instructions

The logical instructions perform AND, AND NOT, OR, and XOR operations on quadword operands.

PAND	Bitwise logical AND.
PANDN	Bitwise logical AND NOT.
POR	Bitwise logical OR.
PXOR	Bitwise logical exclusive OR.

### 5.4.6 MMX Shift and Rotate Instructions

The shift and rotate instructions shift and rotate packed bytes, words, or doublewords, or quadwords in 64-bit operands.

PSLLW	Shift packed words left logical.
PSLLD	Shift packed doublewords left logical.
PSLLQ	Shift packed quadword left logical.
PSRLW	Shift packed words right logical.
PSRLD	Shift packed doublewords right logical.
PSRLQ	Shift packed quadword right logical.

PSRAW	Shift packed words right arithmetic.
PSRAD	Shift packed doublewords right arithmetic.

### 5.4.7 MMX State Management Instructions

The EMMS instruction clears the MMX state from the MMX registers.

EMMS	Empty MMX state.
------	------------------

## 5.5 SSE INSTRUCTIONS

SSE instructions represent an extension of the SIMD execution model introduced with the MMX technology. For more detail on these instructions, see Chapter 10, "Programming with Intel® Streaming SIMD Extensions (Intel® SSE)."

SSE instructions can only be executed on Intel 64 and IA-32 processors that support SSE extensions. Support for these instructions can be detected with the CUID instruction. See the description of the CUID instruction in Chapter 3, "Instruction Set Reference, A-M," of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*.

SSE instructions are divided into four subgroups (note that the first subgroup has subordinate subgroups of its own):

- SIMD single-precision floating-point instructions that operate on the XMM registers.
- MXCSR state management instructions.
- 64-bit SIMD integer instructions that operate on the MMX registers.
- Cacheability control, prefetch, and instruction ordering instructions.

The following sections provide an overview of these groups.

### 5.5.1 SSE SIMD Single-Precision Floating-Point Instructions

These instructions operate on packed and scalar single-precision floating-point values located in XMM registers and/or memory. This subgroup is further divided into the following subordinate subgroups: data transfer, packed arithmetic, comparison, logical, shuffle and unpack, and conversion instructions.

#### 5.5.1.1 SSE Data Transfer Instructions

SSE data transfer instructions move packed and scalar single-precision floating-point operands between XMM registers and between XMM registers and memory.

MOVAPS	Move four aligned packed single-precision floating-point values between XMM registers or between and XMM register and memory.
MOVUPS	Move four unaligned packed single-precision floating-point values between XMM registers or between and XMM register and memory.
MOVHPS	Move two packed single-precision floating-point values to an from the high quadword of an XMM register and memory.
MOVHLPS	Move two packed single-precision floating-point values from the high quadword of an XMM register to the low quadword of another XMM register.
MOVLPS	Move two packed single-precision floating-point values to an from the low quadword of an XMM register and memory.
MOVLHPS	Move two packed single-precision floating-point values from the low quadword of an XMM register to the high quadword of another XMM register.
MOVMSKPS	Extract sign mask from four packed single-precision floating-point values.

**MOVSS** Move scalar single-precision floating-point value between XMM registers or between an XMM register and memory.

### 5.5.1.2 SSE Packed Arithmetic Instructions

SSE packed arithmetic instructions perform packed and scalar arithmetic operations on packed and scalar single-precision floating-point operands.

<b>ADDPS</b>	Add packed single-precision floating-point values.
<b>ADDSS</b>	Add scalar single-precision floating-point values.
<b>SUBPS</b>	Subtract packed single-precision floating-point values.
<b>SUBSS</b>	Subtract scalar single-precision floating-point values.
<b>MULPS</b>	Multiply packed single-precision floating-point values.
<b>MULSS</b>	Multiply scalar single-precision floating-point values.
<b>DIVPS</b>	Divide packed single-precision floating-point values.
<b>DIVSS</b>	Divide scalar single-precision floating-point values.
<b>RCPPS</b>	Compute reciprocals of packed single-precision floating-point values.
<b>RCPSS</b>	Compute reciprocal of scalar single-precision floating-point values.
<b>SQRTPS</b>	Compute square roots of packed single-precision floating-point values.
<b>SQRTSS</b>	Compute square root of scalar single-precision floating-point values.
<b>RSQRTPS</b>	Compute reciprocals of square roots of packed single-precision floating-point values.
<b>RSQRTSS</b>	Compute reciprocal of square root of scalar single-precision floating-point values.
<b>MAXPS</b>	Return maximum packed single-precision floating-point values.
<b>MAXSS</b>	Return maximum scalar single-precision floating-point values.
<b>MINPS</b>	Return minimum packed single-precision floating-point values.
<b>MINSS</b>	Return minimum scalar single-precision floating-point values.

### 5.5.1.3 SSE Comparison Instructions

SSE compare instructions compare packed and scalar single-precision floating-point operands.

<b>CMPPS</b>	Compare packed single-precision floating-point values.
<b>CMPSS</b>	Compare scalar single-precision floating-point values.
<b>COMISS</b>	Perform ordered comparison of scalar single-precision floating-point values and set flags in EFLAGS register.
<b>UCOMISS</b>	Perform unordered comparison of scalar single-precision floating-point values and set flags in EFLAGS register.

### 5.5.1.4 SSE Logical Instructions

SSE logical instructions perform bitwise AND, AND NOT, OR, and XOR operations on packed single-precision floating-point operands.

<b>ANDPS</b>	Perform bitwise logical AND of packed single-precision floating-point values.
<b>ANDNPS</b>	Perform bitwise logical AND NOT of packed single-precision floating-point values.
<b>ORPS</b>	Perform bitwise logical OR of packed single-precision floating-point values.
<b>XORPS</b>	Perform bitwise logical XOR of packed single-precision floating-point values.

### 5.5.1.5 SSE Shuffle and Unpack Instructions

SSE shuffle and unpack instructions shuffle or interleave single-precision floating-point values in packed single-precision floating-point operands.

<b>SHUFPS</b>	Shuffles values in packed single-precision floating-point operands.
---------------	---------------------------------------------------------------------

UNPCKHPS	Unpacks and interleaves the two high-order values from two single-precision floating-point operands.
UNPCKLPS	Unpacks and interleaves the two low-order values from two single-precision floating-point operands.

### 5.5.1.6 SSE Conversion Instructions

SSE conversion instructions convert packed and individual doubleword integers into packed and scalar single-precision floating-point values and vice versa.

CVTPI2PS	Convert packed doubleword integers to packed single-precision floating-point values.
CVTSD2SS	Convert doubleword integer to scalar single-precision floating-point value.
CVTSS2PI	Convert packed single-precision floating-point values to packed doubleword integers.
CVTTSS2PI	Convert with truncation packed single-precision floating-point values to packed doubleword integers.
CVTSS2SI	Convert a scalar single-precision floating-point value to a doubleword integer.
CVTTSS2SI	Convert with truncation a scalar single-precision floating-point value to a scalar doubleword integer.

### 5.5.2 SSE MXCSR State Management Instructions

MXCSR state management instructions allow saving and restoring the state of the MXCSR control and status register.

LDMXCSR	Load MXCSR register.
STMXCSR	Save MXCSR register state.

### 5.5.3 SSE 64-Bit SIMD Integer Instructions

These SSE 64-bit SIMD integer instructions perform additional operations on packed bytes, words, or doublewords contained in MMX registers. They represent enhancements to the MMX instruction set described in Section 5.4, "MMX™ Instructions."

PAVGB	Compute average of packed unsigned byte integers.
PAVGW	Compute average of packed unsigned word integers.
PEXTRW	Extract word.
PINSRW	Insert word.
PMAXUB	Maximum of packed unsigned byte integers.
PMAXSW	Maximum of packed signed word integers.
PMINUB	Minimum of packed unsigned byte integers.
PMINSW	Minimum of packed signed word integers.
PMOVBMSKB	Move byte mask.
PMULHUW	Multiply packed unsigned integers and store high result.
PSADBW	Compute sum of absolute differences.
PSHUFW	Shuffle packed integer word in MMX register.

### 5.5.4 SSE Cacheability Control, Prefetch, and Instruction Ordering Instructions

The cacheability control instructions provide control over the caching of non-temporal data when storing data from the MMX and XMM registers to memory. The `PREFETCHh` allows data to be prefetched to a selected cache level. The `SFENCE` instruction controls instruction ordering on store operations.

MASKMOVQ	Non-temporal store of selected bytes from an MMX register into memory.
----------	------------------------------------------------------------------------



MOVNTQ	Non-temporal store of quadword from an MMX register into memory.
MOVNTPS	Non-temporal store of four packed single-precision floating-point values from an XMM register into memory.
PREFETCH/h	Load 32 or more of bytes from memory to a selected level of the processor's cache hierarchy
SFENCE	Serializes store operations.

## 5.6 SSE2 INSTRUCTIONS

SSE2 extensions represent an extension of the SIMD execution model introduced with MMX technology and the SSE extensions. SSE2 instructions operate on packed double-precision floating-point operands and on packed byte, word, doubleword, and quadword operands located in the XMM registers. For more detail on these instructions, see Chapter 11, "Programming with Intel® Streaming SIMD Extensions 2 (Intel® SSE2)."

SSE2 instructions can only be executed on Intel 64 and IA-32 processors that support the SSE2 extensions. Support for these instructions can be detected with the CPUID instruction. See the description of the CPUID instruction in Chapter 3, "Instruction Set Reference, A-M," of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*.

These instructions are divided into four subgroups (note that the first subgroup is further divided into subordinate subgroups):

- Packed and scalar double-precision floating-point instructions.
- Packed single-precision floating-point conversion instructions.
- 128-bit SIMD integer instructions.
- Cacheability-control and instruction ordering instructions.

The following sections give an overview of each subgroup.

### 5.6.1 SSE2 Packed and Scalar Double-Precision Floating-Point Instructions

SSE2 packed and scalar double-precision floating-point instructions are divided into the following subordinate subgroups: data movement, arithmetic, comparison, conversion, logical, and shuffle operations on double-precision floating-point operands. These are introduced in the sections that follow.

#### 5.6.1.1 SSE2 Data Movement Instructions

SSE2 data movement instructions move double-precision floating-point data between XMM registers and between XMM registers and memory.

MOVAPD	Move two aligned packed double-precision floating-point values between XMM registers or between and XMM register and memory.
MOVUPD	Move two unaligned packed double-precision floating-point values between XMM registers or between and XMM register and memory.
MOVHPD	Move high packed double-precision floating-point value to an from the high quadword of an XMM register and memory.
MOVLPD	Move low packed single-precision floating-point value to an from the low quadword of an XMM register and memory.
MOVMSKPD	Extract sign mask from two packed double-precision floating-point values.
MOVSD	Move scalar double-precision floating-point value between XMM registers or between an XMM register and memory.

### 5.6.1.2 SSE2 Packed Arithmetic Instructions

The arithmetic instructions perform addition, subtraction, multiply, divide, square root, and maximum/minimum operations on packed and scalar double-precision floating-point operands.

ADDPD	Add packed double-precision floating-point values.
ADDSD	Add scalar double precision floating-point values.
SUBPD	Subtract scalar double-precision floating-point values.
SUBSD	Subtract scalar double-precision floating-point values.
MULPD	Multiply packed double-precision floating-point values.
MULSD	Multiply scalar double-precision floating-point values.
DIVPD	Divide packed double-precision floating-point values.
DIVSD	Divide scalar double-precision floating-point values.
SQRTPD	Compute packed square roots of packed double-precision floating-point values.
SQRTSD	Compute scalar square root of scalar double-precision floating-point values.
MAXPD	Return maximum packed double-precision floating-point values.
MAXSD	Return maximum scalar double-precision floating-point values.
MINPD	Return minimum packed double-precision floating-point values.
MINSD	Return minimum scalar double-precision floating-point values.

### 5.6.1.3 SSE2 Logical Instructions

SSE2 logical instructions perform AND, AND NOT, OR, and XOR operations on packed double-precision floating-point values.

ANDPD	Perform bitwise logical AND of packed double-precision floating-point values.
ANDNPD	Perform bitwise logical AND NOT of packed double-precision floating-point values.
ORPD	Perform bitwise logical OR of packed double-precision floating-point values.
XORPD	Perform bitwise logical XOR of packed double-precision floating-point values.

### 5.6.1.4 SSE2 Compare Instructions

SSE2 compare instructions compare packed and scalar double-precision floating-point values and return the results of the comparison either to the destination operand or to the EFLAGS register.

CMPPD	Compare packed double-precision floating-point values.
CMPSD	Compare scalar double-precision floating-point values.
COMISD	Perform ordered comparison of scalar double-precision floating-point values and set flags in EFLAGS register.
UCOMISD	Perform unordered comparison of scalar double-precision floating-point values and set flags in EFLAGS register.

### 5.6.1.5 SSE2 Shuffle and Unpack Instructions

SSE2 shuffle and unpack instructions shuffle or interleave double-precision floating-point values in packed double-precision floating-point operands.

SHUFPD	Shuffles values in packed double-precision floating-point operands.
UNPCKHPD	Unpacks and interleaves the high values from two packed double-precision floating-point operands.
UNPCKLPD	Unpacks and interleaves the low values from two packed double-precision floating-point operands.

### 5.6.1.6 SSE2 Conversion Instructions

SSE2 conversion instructions convert packed and individual doubleword integers into packed and scalar double-precision floating-point values and vice versa. They also convert between packed and scalar single-precision and double-precision floating-point values.

CVTPD2PI	Convert packed double-precision floating-point values to packed doubleword integers.
CVTTPD2PI	Convert with truncation packed double-precision floating-point values to packed doubleword integers.
CVTPI2PD	Convert packed doubleword integers to packed double-precision floating-point values.
CVTPD2DQ	Convert packed double-precision floating-point values to packed doubleword integers.
CVTTPD2DQ	Convert with truncation packed double-precision floating-point values to packed doubleword integers.
CVTDQ2PD	Convert packed doubleword integers to packed double-precision floating-point values.
CVTPS2PD	Convert packed single-precision floating-point values to packed double-precision floating-point values.
CVTPD2PS	Convert packed double-precision floating-point values to packed single-precision floating-point values.
CVTSS2SD	Convert scalar single-precision floating-point values to scalar double-precision floating-point values.
CVTSD2SS	Convert scalar double-precision floating-point values to scalar single-precision floating-point values.
CVTSD2SI	Convert scalar double-precision floating-point values to a doubleword integer.
CVTTSD2SI	Convert with truncation scalar double-precision floating-point values to scalar doubleword integers.
CVTSI2SD	Convert doubleword integer to scalar double-precision floating-point value.

### 5.6.2 SSE2 Packed Single-Precision Floating-Point Instructions

SSE2 packed single-precision floating-point instructions perform conversion operations on single-precision floating-point and integer operands. These instructions represent enhancements to the SSE single-precision floating-point instructions.

CVTDQ2PS	Convert packed doubleword integers to packed single-precision floating-point values.
CVTPS2DQ	Convert packed single-precision floating-point values to packed doubleword integers.
CVTTPS2DQ	Convert with truncation packed single-precision floating-point values to packed doubleword integers.

### 5.6.3 SSE2 128-Bit SIMD Integer Instructions

SSE2 SIMD integer instructions perform additional operations on packed words, doublewords, and quadwords contained in XMM and MMX registers.

MOVDQA	Move aligned double quadword.
MOVDQU	Move unaligned double quadword.
MOVQ2DQ	Move quadword integer from MMX to XMM registers.
MOVDQ2Q	Move quadword integer from XMM to MMX registers.
PMULUDQ	Multiply packed unsigned doubleword integers.
PADDQ	Add packed quadword integers.
PSUBQ	Subtract packed quadword integers.
PSHUFLW	Shuffle packed low words.
PSHUFHW	Shuffle packed high words.
PSHUFD	Shuffle packed doublewords.

PSLLDQ	Shift double quadword left logical.
PSRLDQ	Shift double quadword right logical.
PUNPCKHQDQ	Unpack high quadwords.
PUNPCKLQDQ	Unpack low quadwords.

### 5.6.4 SSE2 Cacheability Control and Ordering Instructions

SSE2 cacheability control instructions provide additional operations for caching of non-temporal data when storing data from XMM registers to memory. LFENCE and MFENCE provide additional control of instruction ordering on store operations.

CLFLUSH	See Section 5.1.13.
LFENCE	Serializes load operations.
MFENCE	Serializes load and store operations.
PAUSE	Improves the performance of “spin-wait loops”.
MASKMOVDQU	Non-temporal store of selected bytes from an XMM register into memory.
MOVNTPD	Non-temporal store of two packed double-precision floating-point values from an XMM register into memory.
MOVNTDQ	Non-temporal store of double quadword from an XMM register into memory.
MOVNTI	Non-temporal store of a doubleword from a general-purpose register into memory.

## 5.7 SSE3 INSTRUCTIONS

The SSE3 extensions offers 13 instructions that accelerate performance of Streaming SIMD Extensions technology, Streaming SIMD Extensions 2 technology, and x87-FP math capabilities. These instructions can be grouped into the following categories:

- One x87FPU instruction used in integer conversion.
- One SIMD integer instruction that addresses unaligned data loads.
- Two SIMD floating-point packed ADD/SUB instructions.
- Four SIMD floating-point horizontal ADD/SUB instructions.
- Three SIMD floating-point LOAD/MOVE/DUPLICATE instructions.
- Two thread synchronization instructions.

SSE3 instructions can only be executed on Intel 64 and IA-32 processors that support SSE3 extensions. Support for these instructions can be detected with the CPUID instruction. See the description of the CPUID instruction in Chapter 3, “Instruction Set Reference, A-M,” of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A*.

The sections that follow describe each subgroup.

### 5.7.1 SSE3 x87-FP Integer Conversion Instruction

FISTTP	Behaves like the FISTP instruction but uses truncation, irrespective of the rounding mode specified in the floating-point control word (FCW).
--------	-----------------------------------------------------------------------------------------------------------------------------------------------

### 5.7.2 SSE3 Specialized 128-bit Unaligned Data Load Instruction

LDDQU	Special 128-bit unaligned load designed to avoid cache line splits.
-------	---------------------------------------------------------------------

### 5.7.3 SSE3 SIMD Floating-Point Packed ADD/SUB Instructions

ADDSUBPS	Performs single-precision addition on the second and fourth pairs of 32-bit data elements within the operands; single-precision subtraction on the first and third pairs.
ADDSUBPD	Performs double-precision addition on the second pair of quadwords, and double-precision subtraction on the first pair.

### 5.7.4 SSE3 SIMD Floating-Point Horizontal ADD/SUB Instructions

HADDPS	Performs a single-precision addition on contiguous data elements. The first data element of the result is obtained by adding the first and second elements of the first operand; the second element by adding the third and fourth elements of the first operand; the third by adding the first and second elements of the second operand; and the fourth by adding the third and fourth elements of the second operand.
HSUBPS	Performs a single-precision subtraction on contiguous data elements. The first data element of the result is obtained by subtracting the second element of the first operand from the first element of the first operand; the second element by subtracting the fourth element of the first operand from the third element of the first operand; the third by subtracting the second element of the second operand from the first element of the second operand; and the fourth by subtracting the fourth element of the second operand from the third element of the second operand.
HADDPD	Performs a double-precision addition on contiguous data elements. The first data element of the result is obtained by adding the first and second elements of the first operand; the second element by adding the first and second elements of the second operand.
HSUBPD	Performs a double-precision subtraction on contiguous data elements. The first data element of the result is obtained by subtracting the second element of the first operand from the first element of the first operand; the second element by subtracting the second element of the second operand from the first element of the second operand.

### 5.7.5 SSE3 SIMD Floating-Point LOAD/MOVE/DUPLICATE Instructions

MOVSHDUP	Loads/moves 128 bits; duplicating the second and fourth 32-bit data elements.
MOVSLDUP	Loads/moves 128 bits; duplicating the first and third 32-bit data elements.
MOVDDUP	Loads/moves 64 bits (bits[63:0] if the source is a register) and returns the same 64 bits in both the lower and upper halves of the 128-bit result register; duplicates the 64 bits from the source.

### 5.7.6 SSE3 Agent Synchronization Instructions

MONITOR	Sets up an address range used to monitor write-back stores.
MWAIT	Enables a logical processor to enter into an optimized state while waiting for a write-back store to the address range set up by the MONITOR instruction.

## 5.8 SUPPLEMENTAL STREAMING SIMD EXTENSIONS 3 (SSSE3) INSTRUCTIONS

SSSE3 provide 32 instructions (represented by 14 mnemonics) to accelerate computations on packed integers. These include:

- Twelve instructions that perform horizontal addition or subtraction operations.
- Six instructions that evaluate absolute values.
- Two instructions that perform multiply and add operations and speed up the evaluation of dot products.
- Two instructions that accelerate packed-integer multiply operations and produce integer values with scaling.
- Two instructions that perform a byte-wise, in-place shuffle according to the second shuffle control operand.

- Six instructions that negate packed integers in the destination operand if the signs of the corresponding element in the source operand is less than zero.
- Two instructions that align data from the composite of two operands.

SSSE3 instructions can only be executed on Intel 64 and IA-32 processors that support SSSE3 extensions. Support for these instructions can be detected with the CPUID instruction. See the description of the CPUID instruction in Chapter 3, “Instruction Set Reference, A-M,” of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A*.

The sections that follow describe each subgroup.

### 5.8.1 Horizontal Addition/Subtraction

PHADDW	Adds two adjacent, signed 16-bit integers horizontally from the source and destination operands and packs the signed 16-bit results to the destination operand.
PHADDSW	Adds two adjacent, signed 16-bit integers horizontally from the source and destination operands and packs the signed, saturated 16-bit results to the destination operand.
PHADDD	Adds two adjacent, signed 32-bit integers horizontally from the source and destination operands and packs the signed 32-bit results to the destination operand.
PHSUBW	Performs horizontal subtraction on each adjacent pair of 16-bit signed integers by subtracting the most significant word from the least significant word of each pair in the source and destination operands. The signed 16-bit results are packed and written to the destination operand.
PHSUBSW	Performs horizontal subtraction on each adjacent pair of 16-bit signed integers by subtracting the most significant word from the least significant word of each pair in the source and destination operands. The signed, saturated 16-bit results are packed and written to the destination operand.
PHSUBD	Performs horizontal subtraction on each adjacent pair of 32-bit signed integers by subtracting the most significant doubleword from the least significant double word of each pair in the source and destination operands. The signed 32-bit results are packed and written to the destination operand.

### 5.8.2 Packed Absolute Values

PABSB	Computes the absolute value of each signed byte data element.
PABSW	Computes the absolute value of each signed 16-bit data element.
PABSD	Computes the absolute value of each signed 32-bit data element.

### 5.8.3 Multiply and Add Packed Signed and Unsigned Bytes

PMADDUBSW	Multiplies each unsigned byte value with the corresponding signed byte value to produce an intermediate, 16-bit signed integer. Each adjacent pair of 16-bit signed values are added horizontally. The signed, saturated 16-bit results are packed to the destination operand.
-----------	--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

### 5.8.4 Packed Multiply High with Round and Scale

PMULHRSW	Multiplies vertically each signed 16-bit integer from the destination operand with the corresponding signed 16-bit integer of the source operand, producing intermediate, signed 32-bit integers. Each intermediate 32-bit integer is truncated to the 18 most significant bits. Rounding is always performed by adding 1 to the least significant bit of the 18-bit intermediate result. The final result is obtained by selecting the 16 bits immediately to the right of the most significant bit of each 18-bit intermediate result and packed to the destination operand.
----------	--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

### 5.8.5 Packed Shuffle Bytes

**PSHUFB** Permutes each byte in place, according to a shuffle control mask. The least significant three or four bits of each shuffle control byte of the control mask form the shuffle index. The shuffle mask is unaffected. If the most significant bit (bit 7) of a shuffle control byte is set, the constant zero is written in the result byte.

### 5.8.6 Packed Sign

**PSIGNB/W/D** Negates each signed integer element of the destination operand if the sign of the corresponding data element in the source operand is less than zero.

### 5.8.7 Packed Align Right

**PALIGNR** Source operand is appended after the destination operand forming an intermediate value of twice the width of an operand. The result is extracted from the intermediate value into the destination operand by selecting the 128 bit or 64 bit value that are right-aligned to the byte offset specified by the immediate value.

## 5.9 SSE4 INSTRUCTIONS

Intel® Streaming SIMD Extensions 4 (SSE4) introduces 54 new instructions. 47 of the SSE4 instructions are referred to as SSE4.1 in this document, 7 new SSE4 instructions are referred to as SSE4.2.

SSE4.1 is targeted to improve the performance of media, imaging, and 3D workloads. SSE4.1 adds instructions that improve compiler vectorization and significantly increase support for packed dword computation. The technology also provides a hint that can improve memory throughput when reading from uncacheable WC memory type.

The 47 SSE4.1 instructions include:

- Two instructions perform packed dword multiplies.
- Two instructions perform floating-point dot products with input/output selects.
- One instruction performs a load with a streaming hint.
- Six instructions simplify packed blending.
- Eight instructions expand support for packed integer MIN/MAX.
- Four instructions support floating-point round with selectable rounding mode and precision exception override.
- Seven instructions improve data insertion and extractions from XMM registers
- Twelve instructions improve packed integer format conversions (sign and zero extensions).
- One instruction improves SAD (sum absolute difference) generation for small block sizes.
- One instruction aids horizontal searching operations.
- One instruction improves masked comparisons.
- One instruction adds qword packed equality comparisons.
- One instruction adds dword packing with unsigned saturation.

The SSE4.2 instructions operating on XMM registers include:

- String and text processing that can take advantage of single-instruction multiple-data programming techniques.
- A SIMD integer instruction that enhances the capability of the 128-bit integer SIMD capability in SSE4.1.



## 5.10 SSE4.1 INSTRUCTIONS

SSE4.1 instructions can use an XMM register as a source or destination. Programming SSE4.1 is similar to programming 128-bit Integer SIMD and floating-point SIMD instructions in SSE/SSE2/SSE3/SSSE3. SSE4.1 does not provide any 64-bit integer SIMD instructions operating on MMX registers. The sections that follow describe each subgroup.

### 5.10.1 Dword Multiply Instructions

PMULLD Returns four lower 32-bits of the 64-bit results of signed 32-bit integer multiplies.  
 PMULDQ Returns two 64-bit signed result of signed 32-bit integer multiplies.

### 5.10.2 Floating-Point Dot Product Instructions

DPPD Perform double-precision dot product for up to 2 elements and broadcast.  
 DPPS Perform single-precision dot products for up to 4 elements and broadcast.

### 5.10.3 Streaming Load Hint Instruction

MOVNTDQA Provides a non-temporal hint that can cause adjacent 16-byte items within an aligned 64-byte region (a streaming line) to be fetched and held in a small set of temporary buffers ("streaming load buffers"). Subsequent streaming loads to other aligned 16-byte items in the same streaming line may be supplied from the streaming load buffer and can improve throughput.

### 5.10.4 Packed Blending Instructions

BLENDPD Conditionally copies specified double-precision floating-point data elements in the source operand to the corresponding data elements in the destination, using an immediate byte control.  
 BLENDPS Conditionally copies specified single-precision floating-point data elements in the source operand to the corresponding data elements in the destination, using an immediate byte control.  
 BLENDVDP Conditionally copies specified double-precision floating-point data elements in the source operand to the corresponding data elements in the destination, using an implied mask.  
 BLENDVPS Conditionally copies specified single-precision floating-point data elements in the source operand to the corresponding data elements in the destination, using an implied mask.  
 PBLENDVB Conditionally copies specified byte elements in the source operand to the corresponding elements in the destination, using an implied mask.  
 PBLENDW Conditionally copies specified word elements in the source operand to the corresponding elements in the destination, using an immediate byte control.

### 5.10.5 Packed Integer MIN/MAX Instructions

PMINUW Compare packed unsigned word integers.  
 PMINUD Compare packed unsigned dword integers.  
 PMINSB Compare packed signed byte integers.  
 PMINSD Compare packed signed dword integers.  
 PMAUW Compare packed unsigned word integers.  
 PMAUD Compare packed unsigned dword integers.  
 PMAUSB Compare packed signed byte integers.



PMAXSD Compare packed signed dword integers.

### 5.10.6 Floating-Point Round Instructions with Selectable Rounding Mode

ROUNDPS Round packed single precision floating-point values into integer values and return rounded floating-point values.

ROUNDPD Round packed double precision floating-point values into integer values and return rounded floating-point values.

ROUNDSS Round the low packed single precision floating-point value into an integer value and return a rounded floating-point value.

ROUNDSD Round the low packed double precision floating-point value into an integer value and return a rounded floating-point value.

### 5.10.7 Insertion and Extractions from XMM Registers

EXTRACTPS Extracts a single-precision floating-point value from a specified offset in an XMM register and stores the result to memory or a general-purpose register.

INSERTPS Inserts a single-precision floating-point value from either a 32-bit memory location or selected from a specified offset in an XMM register to a specified offset in the destination XMM register. In addition, INSERTPS allows zeroing out selected data elements in the destination, using a mask.

PINSRB Insert a byte value from a register or memory into an XMM register.

PINSRD Insert a dword value from 32-bit register or memory into an XMM register.

PINSRQ Insert a qword value from 64-bit register or memory into an XMM register.

PEXTRB Extract a byte from an XMM register and insert the value into a general-purpose register or memory.

PEXTRW Extract a word from an XMM register and insert the value into a general-purpose register or memory.

PEXTRD Extract a dword from an XMM register and insert the value into a general-purpose register or memory.

PEXTRQ Extract a qword from an XMM register and insert the value into a general-purpose register or memory.

### 5.10.8 Packed Integer Format Conversions

PMOVSXBW Sign extend the lower 8-bit integer of each packed word element into packed signed word integers.

PMOVZXBW Zero extend the lower 8-bit integer of each packed word element into packed signed word integers.

PMOVSXBD Sign extend the lower 8-bit integer of each packed dword element into packed signed dword integers.

PMOVZXBW Zero extend the lower 8-bit integer of each packed dword element into packed signed dword integers.

PMOVSXWD Sign extend the lower 16-bit integer of each packed dword element into packed signed dword integers.

PMOVZXWD Zero extend the lower 16-bit integer of each packed dword element into packed signed dword integers.

PMOVSXBQ Sign extend the lower 8-bit integer of each packed qword element into packed signed qword integers.

PMOVZXBQ Zero extend the lower 8-bit integer of each packed qword element into packed signed qword integers.

PMOVSXWQ	Sign extend the lower 16-bit integer of each packed qword element into packed signed qword integers.
PMOVZXWQ	Zero extend the lower 16-bit integer of each packed qword element into packed signed qword integers.
PMOVSXDQ	Sign extend the lower 32-bit integer of each packed qword element into packed signed qword integers.
PMOVZXDQ	Zero extend the lower 32-bit integer of each packed qword element into packed signed qword integers.

### 5.10.9 Improved Sums of Absolute Differences (SAD) for 4-Byte Blocks

MPSADBW	Performs eight 4-byte wide Sum of Absolute Differences operations to produce eight word integers.
---------	---------------------------------------------------------------------------------------------------

### 5.10.10 Horizontal Search

PHMINPOSUW	Finds the value and location of the minimum unsigned word from one of 8 horizontally packed unsigned words. The resulting value and location (offset within the source) are packed into the low dword of the destination XMM register.
------------	----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

### 5.10.11 Packed Test

PTEST	Performs a logical AND between the destination with this mask and sets the ZF flag if the result is zero. The CF flag (zero for TEST) is set if the inverted mask AND'd with the destination is all zeroes.
-------	-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

### 5.10.12 Packed Qword Equality Comparisons

PCMPEQQ	128-bit packed qword equality test.
---------	-------------------------------------

### 5.10.13 Dword Packing With Unsigned Saturation

PACKUSDW	PACKUSDW packs dword to word with unsigned saturation.
----------	--------------------------------------------------------

## 5.11 SSE4.2 INSTRUCTION SET

Five of the SSE4.2 instructions operate on XMM register as a source or destination. These include four text/string processing instructions and one packed quadword compare SIMD instruction. Programming these five SSE4.2 instructions is similar to programming 128-bit Integer SIMD in SSE2/SSSE3. SSE4.2 does not provide any 64-bit integer SIMD instructions.

CRC32 operates on general-purpose registers and is summarized in Section 5.1.6. The sections that follow summarize each subgroup.

### 5.11.1 String and Text Processing Instructions

PCMPESTRI	Packed compare explicit-length strings, return index in ECX/RCX.
PCMPESTRM	Packed compare explicit-length strings, return mask in XMM0.
PCMPISTRI	Packed compare implicit-length strings, return index in ECX/RCX.
PCMPISTRM	Packed compare implicit-length strings, return mask in XMM0.

### 5.11.2 Packed Comparison SIMD integer Instruction

PCMPGTQ Performs logical compare of greater-than on packed integer quadwords.

## 5.12 AESNI AND PCLMULQDQ

Six AESNI instructions operate on XMM registers to provide accelerated primitives for block encryption/decryption using Advanced Encryption Standard (FIPS-197). The PCLMULQDQ instruction performs carry-less multiplication for two binary numbers up to 64-bit wide.

AESDEC	Perform an AES decryption round using an 128-bit state and a round key.
AESDECLAST	Perform the last AES decryption round using an 128-bit state and a round key.
AESENC	Perform an AES encryption round using an 128-bit state and a round key.
AESENCLAST	Perform the last AES encryption round using an 128-bit state and a round key.
AESIMC	Perform an inverse mix column transformation primitive.
AESKEYGENASSIST	Assist the creation of round keys with a key expansion schedule.
PCLMULQDQ	Perform carryless multiplication of two 64-bit numbers.

## 5.13 INTEL® ADVANCED VECTOR EXTENSIONS (INTEL® AVX)

Intel® Advanced Vector Extensions (AVX) promotes legacy 128-bit SIMD instruction sets that operate on XMM register set to use a “vector extension” (VEX) prefix and operates on 256-bit vector registers (YMM). Almost all prior generations of 128-bit SIMD instructions that operates on XMM (but not on MMX registers) are promoted to support three-operand syntax with VEX-128 encoding.

VEX-prefix encoded AVX instructions support 256-bit and 128-bit floating-point operations by extending the legacy 128-bit SIMD floating-point instructions to support three-operand syntax.

Additional functional enhancements are also provided with VEX-encoded AVX instructions.

The list of AVX instructions are listed in the following tables:

- Table 14-2 lists 256-bit and 128-bit floating-point arithmetic instructions promoted from legacy 128-bit SIMD instruction sets.
- Table 14-3 lists 256-bit and 128-bit data movement and processing instructions promoted from legacy 128-bit SIMD instruction sets.
- Table 14-4 lists functional enhancements of 256-bit AVX instructions not available from legacy 128-bit SIMD instruction sets.
- Table 14-5 lists 128-bit integer and floating-point instructions promoted from legacy 128-bit SIMD instruction sets.
- Table 14-6 lists functional enhancements of 128-bit AVX instructions not available from legacy 128-bit SIMD instruction sets.
- Table 14-7 lists 128-bit data movement and processing instructions promoted from legacy instruction sets.

## 5.14 16-BIT FLOATING-POINT CONVERSION

Conversion between single-precision floating-point (32-bit) and half-precision FP (16-bit) data are provided by VCVTSP2PH, VCVTPH2PS:

VCVTPH2PS	Convert eight/four data element containing 16-bit floating-point data into eight/four single-precision floating-point data.
VCVTSP2PH	Convert eight/four data element containing single-precision floating-point data into eight/four 16-bit floating-point data.

## 5.15 FUSED-MULTIPLY-ADD (FMA)

FMA extensions enhances Intel AVX with high-throughput, arithmetic capabilities covering fused multiply-add, fused multiply-subtract, fused multiply add/subtract interleave, signed-reversed multiply on fused multiply-add and multiply-subtract. FMA extensions provide 36 256-bit floating-point instructions to perform computation on 256-bit vectors and additional 128-bit and scalar FMA instructions.

- Table 14-15 lists FMA instruction sets.

## 5.16 INTEL® ADVANCED VECTOR EXTENSIONS 2 (INTEL® AVX2)

Intel® AVX2 extends Intel AVX by promoting most of the 128-bit SIMD integer instructions with 256-bit numeric processing capabilities. Intel AVX2 instructions follow the same programming model as AVX instructions.

In addition, AVX2 provide enhanced functionalities for broadcast/permute operations on data elements, vector shift instructions with variable-shift count per data element, and instructions to fetch non-contiguous data elements from memory.

- Table 14-18 lists promoted vector integer instructions in AVX2.
- Table 14-19 lists new instructions in AVX2 that complements AVX.

## 5.17 INTEL® TRANSACTIONAL SYNCHRONIZATION EXTENSIONS (INTEL® TSX)

XABORT	Abort an RTM transaction execution.
XACQUIRE	Prefix hint to the beginning of an HLE transaction region.
XRELEASE	Prefix hint to the end of an HLE transaction region.
XBEGIN	Transaction begin of an RTM transaction region.
XEND	Transaction end of an RTM transaction region.
XTEST	Test if executing in a transactional region.

## 5.18 INTEL® SHA EXTENSIONS

Intel® SHA extensions provide a set of instructions that target the acceleration of the Secure Hash Algorithm (SHA), specifically the SHA-1 and SHA-256 variants.

SHA1MSG1	Perform an intermediate calculation for the next four SHA1 message dwords from the previous message dwords.
SHA1MSG2	Perform the final calculation for the next four SHA1 message dwords from the intermediate message dwords.
SHA1NEXTE	Calculate SHA1 state E after four rounds.
SHA1RND54	Perform four rounds of SHA1 operations.
SHA256MSG1	Perform an intermediate calculation for the next four SHA256 message dwords.
SHA256MSG2	Perform the final calculation for the next four SHA256 message dwords.
SHA256RND52	Perform two rounds of SHA256 operations.

## 5.19 INTEL® ADVANCED VECTOR EXTENSIONS 512 (INTEL® AVX-512)

The Intel® AVX-512 family comprises a collection of 512-bit SIMD instruction sets to accelerate a diverse range of applications. Intel AVX-512 instructions provide a wide range of functionality that support programming in 512-bit, 256 and 128-bit vector register, plus support for opmask registers and instructions operating on opmask registers.

The collection of 512-bit SIMD instruction sets in Intel AVX-512 include new functionality not available in Intel AVX and Intel AVX2, and promoted instructions similar to equivalent ones in Intel AVX / Intel AVX2 but with enhance-

ment provided by opmask registers not available to VEX-encoded Intel AVX / Intel AVX2. Some instruction mnemonics in AVX / AVX2 that are promoted into AVX-512 can be replaced by new instruction mnemonics that are available only with EVEX encoding, e.g., VBROADCASTF128 into VBROADCASTF32X4. Details of EVEX instruction encoding are discussed in Section 2.6, “Intel® AVX-512 Encoding” of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A*.

512-bit instruction mnemonics in AVX-512F that are not AVX/AVX2 promotions include:

VALIGND/Q	Perform dword/qword alignment of two concatenated source vectors.
VBLENDMPD/PS	Replace the VBLENDVPD/PS instructions (using opmask as select control).
VCOMPRESSPD/PS	Compress packed DP or SP elements of a vector.
VCVT(T)PD2UDQ	Convert packed DP FP elements of a vector to packed unsigned 32-bit integers.
VCVT(T)PS2UDQ	Convert packed SP FP elements of a vector to packed unsigned 32-bit integers.
VCVTQQ2PD/PS	Convert packed signed 64-bit integers to packed DP/SP FP elements.
VCVT(T)SD2USI	Convert the low DP FP element of a vector to an unsigned integer.
VCVT(T)SS2USI	Convert the low SP FP element of a vector to an unsigned integer.
VCVTUDQ2PD/PS	Convert packed unsigned 32-bit integers to packed DP/SP FP elements.
VCVTUSI2USD/S	Convert an unsigned integer to the low DP/SP FP element and merge to a vector.
VEXPANDPD/PS	Expand packed DP or SP elements of a vector.
VEXTRACTF32X4/64X4	Extract a vector from a full-length vector with 32/64-bit granular update.
VEXTRACTI32X4/64X4	Extract a vector from a full-length vector with 32/64-bit granular update.
VFIXUPIIMPD/PS	Perform fix-up to special values in DP/SP FP vectors.
VFIXUPIIMSD/SS	Perform fix-up to special values of the low DP/SP FP element.
VGETEXPPD/PS	Convert the exponent of DP/SP FP elements of a vector into FP values.
VGETEXPSD/SS	Convert the exponent of the low DP/SP FP element in a vector into FP value.
VGETMANTPD/PS	Convert the mantissa of DP/SP FP elements of a vector into FP values.
VGETMANTSD/SS	Convert the mantissa of the low DP/SP FP element of a vector into FP value.
VINSERTF32X4/64X4	Insert a 128/256-bit vector into a full-length vector with 32/64-bit granular update.
VMOVDQA32/64	VMOVDQA with 32/64-bit granular conditional update.
VMOVDQU32/64	VMOVDQU with 32/64-bit granular conditional update.
VPBLENDMD/Q	Blend dword/qword elements using opmask as select control.
VPBROADCASTD/Q	Broadcast from general-purpose register to vector register.
VPCMPD/UD	Compare packed signed/unsigned dwords using specified primitive.
VPCMPQ/UQ	Compare packed signed/unsigned quadwords using specified primitive.
VPCOMPRESSQ/D	Compress packed 64/32-bit elements of a vector.
VPERMI2D/Q	Full permute of two tables of dword/qword elements overwriting the index vector.
VPERMI2PD/PS	Full permute of two tables of DP/SP elements overwriting the index vector.
VPERMT2D/Q	Full permute of two tables of dword/qword elements overwriting one source table.
VPERMT2PD/PS	Full permute of two tables of DP/SP elements overwriting one source table.
VEXPANDD/Q	Expand packed dword/qword elements of a vector.
VPMAXSQ	Compute maximum of packed signed 64-bit integer elements.
VPMAXUD/UQ	Compute maximum of packed unsigned 32/64-bit integer elements.
VPMINSQ	Compute minimum of packed signed 64-bit integer elements.
VPMINUD/UQ	Compute minimum of packed unsigned 32/64-bit integer elements.
VPMOV(S US)QB	Down convert qword elements in a vector to byte elements using truncation (saturation   unsigned saturation).
VPMOV(S US)QW	Down convert qword elements in a vector to word elements using truncation (saturation   unsigned saturation).
VPMOV(S US)QD	Down convert qword elements in a vector to dword elements using truncation (saturation   unsigned saturation).

VPMOV(S US)DB	Down convert dword elements in a vector to byte elements using truncation (saturation   unsigned saturation).
VPMOV(S US)DW	Down convert dword elements in a vector to word elements using truncation (saturation   unsigned saturation).
VPROLD/Q	Rotate dword/qword element left by a constant shift count with conditional update.
VPROLVD/Q	Rotate dword/qword element left by shift counts specified in a vector with conditional update.
VPRORD/Q	Rotate dword/qword element right by a constant shift count with conditional update.
VPRORRD/Q	Rotate dword/qword element right by shift counts specified in a vector with conditional update.
VPSCATTERDD/DQ	Scatter dword/qword elements in a vector to memory using dword indices.
VPSCATTERQD/QQ	Scatter dword/qword elements in a vector to memory using qword indices.
VPSRAQ	Shift qwords right by a constant shift count and shifting in sign bits.
VPSRAVQ	Shift qwords right by shift counts in a vector and shifting in sign bits.
VPTSTNMD/Q	Perform bitwise NAND of dword/qword elements of two vectors and write results to opmask.
VPTERLOGD/Q	Perform bitwise ternary logic operation of three vectors with 32/64 bit granular conditional update.
VPTSTMD/Q	Perform bitwise AND of dword/qword elements of two vectors and write results to opmask.
VRCP14PD/PS	Compute approximate reciprocals of packed DP/SP FP elements of a vector.
VRCP14SD/SS	Compute the approximate reciprocal of the low DP/SP FP element of a vector.
VRNDSCALEPD/PS	Round packed DP/SP FP elements of a vector to specified number of fraction bits.
VRNDSCALESD/SS	Round the low DP/SP FP element of a vector to specified number of fraction bits.
VRSQRT14PD/PS	Compute approximate reciprocals of square roots of packed DP/SP FP elements of a vector.
VRSQRT14SD/SS	Compute the approximate reciprocal of square root of the low DP/SP FP element of a vector.
VSCALEPD/PS	Multiply packed DP/SP FP elements of a vector by powers of two with exponents specified in a second vector.
VSCALESD/SS	Multiply the low DP/SP FP element of a vector by powers of two with exponent specified in the corresponding element of a second vector.
VSCATTERDD/DQ	Scatter SP/DP FP elements in a vector to memory using dword indices.
VSCATTERQD/QQ	Scatter SP/DP FP elements in a vector to memory using qword indices.
VSHUFF32X4/64X2	Shuffle 128-bit lanes of a vector with 32/64 bit granular conditional update.
VSHUFI32X4/64X2	Shuffle 128-bit lanes of a vector with 32/64 bit granular conditional update.

512-bit instruction mnemonics in AVX-512DQ that are not AVX/AVX2 promotions include:

VCVT(T)PD2QQ	Convert packed DP FP elements of a vector to packed signed 64-bit integers.
VCVT(T)PD2UQQ	Convert packed DP FP elements of a vector to packed unsigned 64-bit integers.
VCVT(T)PS2QQ	Convert packed SP FP elements of a vector to packed signed 64-bit integers.
VCVT(T)PS2UQQ	Convert packed SP FP elements of a vector to packed unsigned 64-bit integers.
VCVTUQQ2PD/PS	Convert packed unsigned 64-bit integers to packed DP/SP FP elements.
VEXTRACTF64X2	Extract a vector from a full-length vector with 64-bit granular update.
VEXTRACTI64X2	Extract a vector from a full-length vector with 64-bit granular update.
VFPCLASSPD/PS	Test packed DP/SP FP elements in a vector by numeric/special-value category.
VFPCLASSSD/SS	Test the low DP/SP FP element by numeric/special-value category.
VINSERTF64X2	Insert a 128-bit vector into a full-length vector with 64-bit granular update.
VINSERTI64X2	Insert a 128-bit vector into a full-length vector with 64-bit granular update.
VPMOVM2D/Q	Convert opmask register to vector register in 32/64-bit granularity.



VPMOVB2D/Q2M	Convert a vector register in 32/64-bit granularity to an opmask register.
VPMULLQ	Multiply packed signed 64-bit integer elements of two vectors and store low 64-bit signed result.
VRANGEPD/PS	Perform RANGE operation on each pair of DP/SP FP elements of two vectors using specified range primitive in imm8.
VRANGESD/SS	Perform RANGE operation on the pair of low DP/SP FP element of two vectors using specified range primitive in imm8.
VREDUCEPD/PS	Perform Reduction operation on packed DP/SP FP elements of a vector using specified reduction primitive in imm8.
VREDUCESD/SS	Perform Reduction operation on the low DP/SP FP element of a vector using specified reduction primitive in imm8.

512-bit instruction mnemonics in AVX-512BW that are not AVX/AVX2 promotions include:

VDBPSADBW	Double block packed Sum-Absolute-Differences on unsigned bytes.
VMOVDQU8/16	VMOVDQU with 8/16-bit granular conditional update.
VPBLENDMB	Replaces the VPBLENDVB instruction (using opmask as select control).
VPBLENDMW	Blend word elements using opmask as select control.
VPBROADCASTB/W	Broadcast from general-purpose register to vector register.
VPCMPB/UB	Compare packed signed/unsigned bytes using specified primitive.
VPCMPW/UW	Compare packed signed/unsigned words using specified primitive.
VPERMW	Permute packed word elements.
VPERMI2B/W	Full permute from two tables of byte/word elements overwriting the index vector.
VPMOVM2B/W	Convert opmask register to vector register in 8/16-bit granularity.
VPMOVB2M/W2M	Convert a vector register in 8/16-bit granularity to an opmask register.
VPMOV(S US)WB	Down convert word elements in a vector to byte elements using truncation (saturation   unsigned saturation).
VPSLLVW	Shift word elements in a vector left by shift counts in a vector.
VPSRAVW	Shift words right by shift counts in a vector and shifting in sign bits.
VPSRLVW	Shift word elements in a vector right by shift counts in a vector.
VPTESTNMB/W	Perform bitwise NAND of byte/word elements of two vectors and write results to opmask.
VPTESTMB/W	Perform bitwise AND of byte/word elements of two vectors and write results to opmask.

512-bit instruction mnemonics in AVX-512CD that are not AVX/AVX2 promotions include:

VPBROADCASTM	Broadcast from opmask register to vector register.
VPCONFLICTD/Q	Detect conflicts within a vector of packed 32/64-bit integers.
VPLZCNTD/Q	Count the number of leading zero bits of packed dword/qword elements.

Opmask instructions include:

KADDB/W/D/Q	Add two 8/16/32/64-bit opmasks.
KANDB/W/D/Q	Logical AND two 8/16/32/64-bit opmasks.
KANDNB/W/D/Q	Logical AND NOT two 8/16/32/64-bit opmasks.
KMOVW/W/D/Q	Move from or move to opmask register of 8/16/32/64-bit data.
KNOTB/W/D/Q	Bitwise NOT of two 8/16/32/64-bit opmasks.
KORB/W/D/Q	Logical OR two 8/16/32/64-bit opmasks.
KORTESTB/W/D/Q	Update EFLAGS according to the result of bitwise OR of two 8/16/32/64-bit opmasks.
KSHIFTLB/W/D/Q	Shift left 8/16/32/64-bit opmask by specified count.
KSHIFTRB/W/D/Q	Shift right 8/16/32/64-bit opmask by specified count.

KTESTB/W/D/Q	Update EFLAGS according to the result of bitwise TEST of two 8/16/32/64-bit opmasks.
KUNPCKBW/WD/DQ	Unpack and interleave two 8/16/32-bit opmasks into 16/32/64-bit mask.
KXNORB/W/D/Q	Bitwise logical XNOR of two 8/16/32/64-bit opmasks.
KXORB/W/D/Q	Logical XOR of two 8/16/32/64-bit opmasks.

512-bit instruction mnemonics in AVX-512ER include:

VEXP2PD/PS	Compute approximate base-2 exponential of packed DP/SP FP elements of a vector.
VEXP2SD/SS	Compute approximate base-2 exponential of the low DP/SP FP element of a vector.
VRCP28PD/PS	Compute approximate reciprocals to 28 bits of packed DP/SP FP elements of a vector.
VRCP28SD/SS	Compute the approximate reciprocal to 28 bits of the low DP/SP FP element of a vector.
VRSQRT28PD/PS	Compute approximate reciprocals of square roots to 28 bits of packed DP/SP FP elements of a vector.
VRSQRT28SD/SS	Compute the approximate reciprocal of square root to 28 bits of the low DP/SP FP element of a vector.

512-bit instruction mnemonics in AVX-512PF include:

VGATHERPF0DPD/PS	Sparse prefetch of packed DP/SP FP vector with T0 hint using dword indices.
VGATHERPF0QPD/PS	Sparse prefetch of packed DP/SP FP vector with T0 hint using qword indices.
VGATHERPF1DPD/PS	Sparse prefetch of packed DP/SP FP vector with T1 hint using dword indices.
VGATHERPF1QPD/PS	Sparse prefetch of packed DP/SP FP vector with T1 hint using qword indices.
VSCATTERPF0DPD/PS	Sparse prefetch of packed DP/SP FP vector with T0 hint to write using dword indices.
VSCATTERPF0QPD/PS	Sparse prefetch of packed DP/SP FP vector with T0 hint to write using qword indices.
VSCATTERPF1DPD/PS	Sparse prefetch of packed DP/SP FP vector with T1 hint to write using dword indices.
VSCATTERPF1QPD/PS	Sparse prefetch of packed DP/SP FP vector with T1 hint to write using qword indices.

## 5.20 SYSTEM INSTRUCTIONS

The following system instructions are used to control those functions of the processor that are provided to support for operating systems and executives.

CLAC	Clear AC Flag in EFLAGS register.
STAC	Set AC Flag in EFLAGS register.
LGDT	Load global descriptor table (GDT) register.
SGDT	Store global descriptor table (GDT) register.
LLDT	Load local descriptor table (LDT) register.
SLDT	Store local descriptor table (LDT) register.
LTR	Load task register.
STR	Store task register.
LIDT	Load interrupt descriptor table (IDT) register.
SIDT	Store interrupt descriptor table (IDT) register.
MOV	Load and store control registers.
LMSW	Load machine status word.
SMSW	Store machine status word.
CLTS	Clear the task-switched flag.
ARPL	Adjust requested privilege level.
LAR	Load access rights.
LSL	Load segment limit.



VERR	Verify segment for reading
VERW	Verify segment for writing.
MOV	Load and store debug registers.
INVD	Invalidate cache, no writeback.
WBINVD	Invalidate cache, with writeback.
INVLPG	Invalidate TLB Entry.
INVPCID	Invalidate Process-Context Identifier.
LOCK (prefix)	Lock Bus.
HLT	Halt processor.
RSM	Return from system management mode (SMM).
RDMSR	Read model-specific register.
WRMSR	Write model-specific register.
RDPMC	Read performance monitoring counters.
RDTSC	Read time stamp counter.
RDTSCP	Read time stamp counter and processor ID.
SYSENTER	Fast System Call, transfers to a flat protected mode kernel at CPL = 0.
SYSEXIT	Fast System Call, transfers to a flat protected mode kernel at CPL = 3.
XSAVE	Save processor extended states to memory.
XSAVEC	Save processor extended states with compaction to memory.
XSAVEOPT	Save processor extended states to memory, optimized.
XSAVES	Save processor supervisor-mode extended states to memory.
XRSTOR	Restore processor extended states from memory.
XRSTORS	Restore processor supervisor-mode extended states from memory.
XGETBV	Reads the state of an extended control register.
XSETBV	Writes the state of an extended control register.
RDFSBASE	Reads from FS base address at any privilege level.
RDGSBASE	Reads from GS base address at any privilege level.
WRFSBASE	Writes to FS base address at any privilege level.
WRGSBASE	Writes to GS base address at any privilege level.

## 5.21 64-BIT MODE INSTRUCTIONS

The following instructions are introduced in 64-bit mode. This mode is a sub-mode of IA-32e mode.

CDQE	Convert doubleword to quadword.
CMPSQ	Compare string operands.
CMPXCHG16B	Compare RDX:RAX with m128.
LODSQ	Load qword at address (R)SI into RAX.
MOVSQ	Move qword from address (R)SI to (R)DI.
MOVZX (64-bits)	Move bytes/words to doublewords/quadwords, zero-extension.
STOSQ	Store RAX at address RDI.
SWAPGS	Exchanges current GS base register value with value in MSR address C0000102H.
SYSCALL	Fast call to privilege level 0 system procedures.
SYSRET	Return from fast systemcall.

## 5.22 VIRTUAL-MACHINE EXTENSIONS

The behavior of the VMCS-maintenance instructions is summarized below:

VMPTRLD	Takes a single 64-bit source operand in memory. It makes the referenced VMCS active and current.
VMPTRST	Takes a single 64-bit destination operand that is in memory. Current-VMCS pointer is stored into the destination operand.
VMCLEAR	Takes a single 64-bit operand in memory. The instruction sets the launch state of the VMCS referenced by the operand to “clear”, renders that VMCS inactive, and ensures that data for the VMCS have been written to the VMCS-data area in the referenced VMCS region.
VMREAD	Reads a component from the VMCS (the encoding of that field is given in a register operand) and stores it into a destination operand.
VMWRITE	Writes a component to the VMCS (the encoding of that field is given in a register operand) from a source operand.

The behavior of the VMX management instructions is summarized below:

VMLAUNCH	Launches a virtual machine managed by the VMCS. A VM entry occurs, transferring control to the VM.
VMRESUME	Resumes a virtual machine managed by the VMCS. A VM entry occurs, transferring control to the VM.
VMXOFF	Causes the processor to leave VMX operation.
VMXON	Takes a single 64-bit source operand in memory. It causes a logical processor to enter VMX root operation and to use the memory referenced by the operand to support VMX operation.

The behavior of the VMX-specific TLB-management instructions is summarized below:

INVEPT	Invalidate cached <b>Extended Page Table</b> (EPT) mappings in the processor to synchronize address translation in virtual machines with memory-resident EPT pages.
INVVPID	Invalidate cached mappings of address translation based on the <b>Virtual Processor ID</b> (VPID).

None of the instructions above can be executed in compatibility mode; they generate invalid-opcode exceptions if executed in compatibility mode.

The behavior of the guest-available instructions is summarized below:

VMCALL	Allows a guest in VMX non-root operation to call the VMM for service. A VM exit occurs, transferring control to the VMM.
VMFUNC	This instruction allows software in VMX non-root operation to invoke a VM function, which is processor functionality enabled and configured by software in VMX root operation. No VM exit occurs.

## 5.23 SAFER MODE EXTENSIONS

The behavior of the GETSEC instruction leaves of the Safer Mode Extensions (SMX) are summarized below:

GETSEC[CAPABILITIES]	Returns the available leaf functions of the GETSEC instruction.
GETSEC[ENTERACCS]	Loads an authenticated code chipset module and enters authenticated code execution mode.
GETSEC[EXITAC]	Exits authenticated code execution mode.
GETSEC[SENDER]	Establishes a Measured Launched Environment (MLE) which has its dynamic root of trust anchored to a chipset supporting Intel Trusted Execution Technology.
GETSEC[SEXIT]	Exits the MLE.
GETSEC[PARAMETERS]	Returns SMX related parameter information.
GETSEC[SMCTRL]	SMX mode control.
GETSEC[WAKEUP]	Wakes up sleeping logical processors inside an MLE.

## 5.24 INTEL® MEMORY PROTECTION EXTENSIONS

Intel Memory Protection Extensions (MPX) provides a set of instructions to enable software to add robust bounds checking capability to memory references. Details of Intel MPX are described in Chapter 17, “Intel® MPX”.

BNDMK	Create a LowerBound and a UpperBound in a register.
BNDCL	Check the address of a memory reference against a LowerBound.
BNDCU	Check the address of a memory reference against an UpperBound in 1’s compliment form.
BNDCN	Check the address of a memory reference against an UpperBound not in 1’s compliment form.
BNDMOV	Copy or load from memory of the LowerBound and UpperBound to a register.
BNDMOV	Store to memory of the LowerBound and UpperBound from a register.
BNDLDX	Load bounds using address translation.
BNDSTX	Store bounds using address translation.

## 5.25 INTEL® SECURITY GUARD EXTENSIONS

Intel Security Guard Extensions (SGX) provide two sets of instruction leaf functions to enable application software to instantiate a protected container, referred to as an enclave. The enclave instructions are organized as leaf functions under two instruction mnemonics: ENCLS (ring 0) and ENCLU (ring 3). Details of Intel SGX are described in CHAPTER 37 through CHAPTER 43 of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3D*.

The first implementation of Intel SGX is also referred to as SGX1, it is introduced with the 6th Generation Intel Core Processors. The leaf functions supported in SGX1 is shown in Table 5-3.

**Table 5-3. Supervisor and User Mode Enclave Instruction Leaf Functions in Long-Form of SGX1**

Supervisor Instruction	Description	User Instruction	Description
ENCLS[EADD]	Add a page	ENCLU[EENTER]	Enter an Enclave
ENCLS[EBLOCK]	Block an EPC page	ENCLU[EEXIT]	Exit an Enclave
ENCLS[ECREATE]	Create an enclave	ENCLU[EGETKEY]	Create a cryptographic key
ENCLS[EDBGGRD]	Read data by debugger	ENCLU[EREPORT]	Create a cryptographic report
ENCLS[EDBGWR]	Write data by debugger	ENCLU[ERESUME]	Re-enter an Enclave
ENCLS[EEXTEND]	Extend EPC page measurement		
ENCLS[EINIT]	Initialize an enclave		
ENCLS[ELDB]	Load an EPC page as blocked		
ENCLS[ELDU]	Load an EPC page as unblocked		
ENCLS[EPA]	Add version array		
ENCLS[EREMOVE]	Remove a page from EPC		
ENCLS[ETRACK]	Activate EBLOCK checks		
ENCLS[EWB]	Write back/invalidate an EPC page		



### 3. Updates to Chapter 13, Volume 1

Change bars show changes to Chapter 13 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture*.

-----

Changes include updates to make explicit the layout of each section of the XSAVE area.

# CHAPTER 13

## MANAGING STATE USING THE XSAVE FEATURE SET

---

The XSAVE feature set extends the functionality of the FXSAVE and FXRSTOR instructions (see Section 10.5, “FXSAVE and FXRSTOR Instructions”) by supporting the saving and restoring of processor state in addition to the x87 execution environment (**x87 state**) and the registers used by the streaming SIMD extensions (**SSE state**).

The **XSAVE feature set** comprises eight instructions. XGETBV and XSETBV allow software to read and write the extended control register XCR0, which controls the operation of the XSAVE feature set. XSAVE, XSAVEOPT, XSAVEC, and XSAVES are four instructions that save processor state to memory; XRSTOR and XRSTORS are corresponding instructions that load processor state from memory. XGETBV, XSAVE, XSAVEOPT, XSAVEC, and XRSTOR can be executed at any privilege level; XSETBV, XSAVES, and XRSTORS can be executed only if CPL = 0. In addition to XCR0, the XSAVES and XRSTORS instructions are controlled also by the IA32\_XSS MSR (index DA0H).

The XSAVE feature set organizes the state that manages into **state components**. Operation of the instructions is based on **state-component bitmaps** that have the same format as XCR0 and as the IA32\_XSS MSR: each bit corresponds to a state component. Section 13.1 discusses these state components and bitmaps in more detail.

Section 13.2 describes how the processor enumerates support for the XSAVE feature set and for **XSAVE-enabled features** (those features that require use of the XSAVE feature set for their enabling). Section 13.3 explains how software can enable the XSAVE feature set and XSAVE-enabled features.

The XSAVE feature set allows saving and loading processor state from a region of memory called an **XSAVE area**. Section 13.4 presents details of the XSAVE area and its organization. Each XSAVE-managed state component is associated with a section of the XSAVE area. Section 13.5 describes in detail each of the XSAVE-managed state components.

Section 13.7 through Section 13.12 describe the operation of XSAVE, XRSTOR, XSAVEOPT, XSAVEC, XSAVES, and XRSTORS, respectively.

### 13.1 XSAVE-SUPPORTED FEATURES AND STATE-COMPONENT BITMAPS

The XSAVE feature set supports the saving and restoring of **state components**, each of which is a discrete set of processor registers (or parts of registers). In general, each such state component corresponds to a particular CPU feature. Such a feature is **XSAVE-supported**. Some XSAVE-supported features use registers in multiple XSAVE-managed state components.

The XSAVE feature set organizes the state components of the XSAVE-supported features using **state-component bitmaps**. A state-component bitmap comprises 64 bits; each bit in such a bitmap corresponds to a single state component. The following bits are defined in state-component bitmaps:

- Bit 0 corresponds to the state component used for the x87 FPU execution environment (**x87 state**). See Section 13.5.1.
- Bit 1 corresponds to the state component used for registers used by the streaming SIMD extensions (**SSE state**). See Section 13.5.2.
- Bit 2 corresponds to the state component used for the additional register state used by the Intel® Advanced Vector Extensions (**AVX state**). See Section 13.5.3.
- Bits 4:3 correspond to the two state components used for the additional register state used by Intel® Memory Protection Extensions (**MPX state**):
  - State component 3 is used for the 4 128-bit bounds registers BND0–BND3 (**BNDREGS state**).
  - State component 4 is used for the 64-bit user-mode MPX configuration register BNDCFGU and the 64-bit MPX status register BNDSTATUS (**BNDCSR state**).
- Bits 7:5 correspond to the three state components used for the additional register state used by Intel® Advanced Vector Extensions 512 (**AVX-512 state**):
  - State component 5 is used for the 8 64-bit opmask registers k0–k7 (**opmask state**).

- State component 6 is used for the upper 256 bits of the registers ZMM0–ZMM15. These 16 256-bit values are denoted ZMM0\_H–ZMM15\_H (**ZMM\_Hi256 state**).
- State component 7 is used for the 16 512-bit registers ZMM16–ZMM31 (**Hi16\_ZMM state**).
- Bit 8 corresponds to the state component used for the Intel Processor Trace MSRs (**PT state**).
- Bit 9 corresponds to the state component used for the protection-key feature’s register PKRU (**PKRU state**). See Section 13.5.7.

Bits in the range 62:10 are not currently defined in state-component bitmaps and are reserved for future expansion. As individual state component is defined within bits 62:10, additional sub-sections are updated within Section 13.5 over time. Bit 63 is used for special functionality in some bitmaps and does not correspond to any state component.

The state component corresponding to bit *i* of state-component bitmaps is called **state component *i***. Thus, x87 state is state component 0; SSE state is state component 1; AVX state is state component 2; MPX state comprises state components 3–4; AVX-512 state comprises state components 5–7; PT state is state component 8; and PKRU state is state component 9.

The XSAVE feature set uses state-component bitmaps in multiple ways. Most of the instructions use an implicit operand (in EDX:EAX), called the **instruction mask**, which is the state-component bitmap that specifies the state components on which the instruction operates.

Some state components are **user state components**, and they can be managed by the entire XSAVE feature set. Other state components are **supervisor state components**, and they can be managed only by XSAVES and XRSTORS. All the state components corresponding to bits in the range 9:0 are user state components, except PT state (corresponding to bit 8), which is a supervisor state component.

Extended control register XCR0 contains a state-component bitmap that specifies the user state components that software has enabled the XSAVE feature set to manage. If the bit corresponding to a state component is clear in XCR0, instructions in the XSAVE feature set will not operate on that state component, regardless of the value of the instruction mask.

The IA32\_XSS MSR (index DA0H) contains a state-component bitmap that specifies the supervisor state components that software has enabled XSAVES and XRSTORS to manage (XSAVE, XSAVEC, XSAVEOPT, and XRSTOR cannot manage supervisor state components). If the bit corresponding to a state component is clear in the IA32\_XSS MSR, XSAVES and XRSTORS will not operate on that state component, regardless of the value of the instruction mask.

Some XSAVE-supported features can be used only if XCR0 has been configured so that the features’ state components can be managed by the XSAVE feature set. (This applies only to features with user state components.) Such state components and features are **XSAVE-enabled**. In general, the processor will not modify (or allow modification of) the registers of a state component of an XSAVE-enabled feature if the bit corresponding to that state component is clear in XCR0. (If software clears such a bit in XCR0, the processor preserves the corresponding state component.) If an XSAVE-enabled feature has not been fully enabled in XCR0, execution of any instruction defined for that feature causes an invalid-opcode exception (#UD).

As will be explained in Section 13.3, the XSAVE feature set is enabled only if CR4.OSXSAVE[bit 18] = 1. If CR4.OSXSAVE = 0, the processor treats XSAVE-enabled state features and their state components as if all bits in XCR0 were clear; the state components cannot be modified and the features’ instructions cannot be executed.

The state components for x87 state, for SSE state, for PT state, and for PKRU state are XSAVE-managed but the corresponding features are not XSAVE-enabled. Processors allow modification of this state, as well as execution of x87 FPU instructions and SSE instructions and use of Intel Processor Trace and protection keys, regardless of the value of CR4.OSXSAVE and XCR0.

## 13.2 ENUMERATION OF CPU SUPPORT FOR XSAVE INSTRUCTIONS AND XSAVE-SUPPORTED FEATURES

A processor enumerates support for the XSAVE feature set and for features supported by that feature set using the CPUID instruction. The following items provide specific details:

- CPUID.1:ECX.XSAVE[bit 26] enumerates general support for the XSAVE feature set:

- If this bit is 0, the processor does not support any of the following instructions: XGETBV, XRSTOR, XRSTORS, XSAVE, XSAVEC, XSAVEOPT, XSAVES, and XSETBV; the processor provides no further enumeration through CPUID function 0DH (see below).
- If this bit is 1, the processor supports the following instructions: XGETBV, XRSTOR, XSAVE, and XSETBV.<sup>1</sup> Further enumeration is provided through CPUID function 0DH.

CR4.OSXSAVE can be set to 1 if and only if CPUID.1:ECX.XSAVE[bit 26] is enumerated as 1.

- CPUID function 0DH enumerates details of CPU support through a set of sub-functions. Software selects a specific sub-function by the value placed in the ECX register. The following items provide specific details:
  - CPUID function 0DH, sub-function 0.
    - EDX:EAX is a bitmap of all the user state components that can be managed using the XSAVE feature set. A bit can be set in XCR0 if and only if the corresponding bit is set in this bitmap. Every processor that supports the XSAVE feature set will set EAX[0] (x87 state) and EAX[1] (SSE state).  
If EAX[*i*] = 1 (for 1 < *i* < 32) or EDX[*i*-32] = 1 (for 32 ≤ *i* < 63), sub-function *i* enumerates details for state component *i* (see below).
    - ECX enumerates the size (in bytes) required by the XSAVE instruction for an XSAVE area containing all the user state components supported by this processor.
    - EBX enumerates the size (in bytes) required by the XSAVE instruction for an XSAVE area containing all the user state components corresponding to bits currently set in XCR0.
  - CPUID function 0DH, sub-function 1.
    - EAX[0] enumerates support for the XSAVEOPT instruction. The instruction is supported if and only if this bit is 1. If EAX[0] = 0, execution of XSAVEOPT causes an invalid-opcode exception (#UD).
    - EAX[1] enumerates support for **compaction extensions** to the XSAVE feature set. The following are supported if this bit is 1:
      - The compacted format of the extended region of XSAVE areas (see Section 13.4.3).
      - The XSAVEC instruction. If EAX[1] = 0, execution of XSAVEC causes a #UD.
      - Execution of the compacted form of XRSTOR (see Section 13.8).
    - EAX[2] enumerates support for execution of XGETBV with ECX = 1. This allows software to determine the state of the init optimization. See Section 13.6.
    - EAX[3] enumerates support for XSAVES, XRSTORS, and the IA32\_XSS MSR. If EAX[3] = 0, execution of XSAVES or XRSTORS causes a #UD; an attempt to access the IA32\_XSS MSR using RDMSR or WRMSR causes a general-protection exception (#GP). Every processor that supports a supervisor state component sets EAX[3]. Every processor that sets EAX[3] (XSAVES, XRSTORS, IA32\_XSS) will also set EAX[1] (the compaction extensions).
    - EAX[31:4] are reserved.
    - EBX enumerates the size (in bytes) required by the XSAVES instruction for an XSAVE area containing all the state components corresponding to bits currently set in XCR0 | IA32\_XSS.
    - EDX:ECX is a bitmap of all the supervisor state components that can be managed by XSAVES and XRSTORS. A bit can be set in the IA32\_XSS MSR if and only if the corresponding bit is set in this bitmap.

## NOTE

In summary, the XSAVE feature set supports state component *i* ( $0 \leq i < 63$ ) if one of the following is true: (1)  $i < 32$  and CPUID.(EAX=0DH,ECX=0):EAX[*i*] = 1; (2)  $i \geq 32$  and CPUID.(EAX=0DH,ECX=0):EAX[*i*-32] = 1; (3)  $i < 32$  and CPUID.(EAX=0DH,ECX=1):ECX[*i*] = 1; or (4)  $i \geq 32$  and CPUID.(EAX=0DH,ECX=1):EDX[*i*-32] = 1. The XSAVE feature set supports user state component *i* if (1) or (2) holds; if (3) or (4) holds, state component *i* is a supervisor state component and support is limited to XSAVES and XRSTORS.

---

1. If CPUID.1:ECX.XSAVE[bit 26] = 1, XGETBV and XSETBV may be executed with ECX = 0 (to read and write XCR0). Any support for execution of these instructions with other values of ECX is enumerated separately.



- CPUID function 0DH, sub-function  $i$  ( $i > 1$ ). This sub-function enumerates details for state component  $i$ . If the XSAVE feature set supports state component  $i$  (see note above), the following items provide specific details:
  - EAX enumerates the size (in bytes) required for state component  $i$ .
  - If state component  $i$  is a user state component, EBX enumerates the offset (in bytes, from the base of the XSAVE area) of the section used for state component  $i$ . (This offset applies only when the standard format for the extended region of the XSAVE area is being used; see Section 13.4.3.)
  - If state component  $i$  is a supervisor state component, EBX returns 0.
  - If state component  $i$  is a user state component, ECX[0] return 0; if state component  $i$  is a supervisor state component, ECX[0] returns 1.
  - The value returned by ECX[1] indicates the alignment of state component  $i$  when the compacted format of the extended region of an XSAVE area is used (see Section 13.4.3). If ECX[1] returns 0, state component  $i$  is located immediately following the preceding state component; if ECX[1] returns 1, state component  $i$  is located on the next 64-byte boundary following the preceding state component.
  - ECX[31:2] and EDX return 0.

If the XSAVE feature set does not support state component  $i$ , sub-function  $i$  returns 0 in EAX, EBX, ECX, and EDX.

### 13.3 ENABLING THE XSAVE FEATURE SET AND XSAVE-ENABLED FEATURES

Software enables the XSAVE feature set by setting CR4.OSXSAVE[bit 18] to 1 (e.g., with the MOV to CR4 instruction). If this bit is 0, execution of any of XGETBV, XRSTOR, XRSTORS, XSAVE, XSAVEC, XSAVEOPT, XSAVES, and XSETBV causes an invalid-opcode exception (#UD).

When CR4.OSXSAVE = 1 and CPL = 0, executing the XSETBV instruction with ECX = 0 writes the 64-bit value in EDX:EAX to XCR0 (EAX is written to XCR0[31:0] and EDX to XCR0[63:32]). (Execution of the XSETBV instruction causes a general-protection fault — #GP — if CPL > 0.) The following items provide details regarding individual bits in XCR0:

- XCR0[0] is associated with x87 state (see Section 13.5.1). XCR0[0] is always 1. It has that value coming out of RESET. Executing the XSETBV instruction causes a general-protection fault (#GP) if ECX = 0 and EAX[0] is 0.
- XCR0[1] is associated with SSE state (see Section 13.5.2). Software can use the XSAVE feature set to manage SSE state only if XCR0[1] = 1. The value of XCR0[1] in no way determines whether software can execute SSE instructions (these instructions can be executed even if XCR0[1] = 0).

XCR0[1] is 0 coming out of RESET. As noted in Section 13.2, every processor that supports the XSAVE feature set allows software to set XCR0[1].

- XCR0[2] is associated with AVX state (see Section 13.5.3). Software can use the XSAVE feature set to manage AVX state only if XCR0[2] = 1. In addition, software can execute AVX instructions only if CR4.OSXSAVE = XCR0[2] = 1. Otherwise, any execution of an AVX instruction causes an invalid-opcode exception (#UD).

XCR0[2] is 0 coming out of RESET. As noted in Section 13.2, a processor allows software to set XCR0[2] if and only if CPUID.(EAX=0DH,ECX=0):EAX[2] = 1. In addition, executing the XSETBV instruction causes a general-protection fault (#GP) if ECX = 0 and EAX[2:1] has the value 10b; that is, software cannot enable the XSAVE feature set for AVX state but not for SSE state.

As noted in Section 13.1, the processor will preserve AVX state unmodified if software clears XCR0[2].

However, clearing XCR0[2] while AVX state is not in its initial configuration may cause SSE instructions to incur a power and performance penalty. See Section 13.5.3, "Enable the Use Of XSAVE Feature Set And XSAVE State Components" of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*, for how system software can avoid this penalty.

- XCR0[4:3] are associated with MPX state (see Section 13.5.4). Software can use the XSAVE feature set to manage MPX state only if XCR0[4:3] = 11b. In addition, software can execute MPX instructions only if CR4.OSXSAVE = 1 and XCR0[4:3] = 11b. Otherwise, any execution of an MPX instruction causes an invalid-opcode exception (#UD).<sup>1</sup>

XCR0[4:3] have value 00b coming out of RESET. As noted in Section 13.2, a processor allows software to set XCR0[4:3] to 11b if and only if CPUID.(EAX=0DH,ECX=0):EAX[4:3] = 11b. In addition, executing the XSETBV instruction causes a general-protection fault (#GP) if ECX = 0, EAX[4:3] is neither 00b nor 11b; that is, software can enable the XSAVE feature set for MPX state only if it does so for both state components.

As noted in Section 13.1, the processor will preserve MPX state unmodified if software clears XCR0[4:3].

- XCR0[7:5] are associated with AVX-512 state (see Section 13.5.5). Software can use the XSAVE feature set to manage AVX-512 state only if XCR0[7:5] = 111b. In addition, software can execute AVX-512 instructions only if CR4.OSXSAVE = 1 and XCR0[7:5] = 111b. Otherwise, any execution of an AVX-512 instruction causes an invalid-opcode exception (#UD).

XCR0[7:5] have value 000b coming out of RESET. As noted in Section 13.2, a processor allows software to set XCR0[7:5] to 111b if and only if CPUID.(EAX=0DH,ECX=0):EAX[7:5] = 111b. In addition, executing the XSETBV instruction causes a general-protection fault (#GP) if ECX = 0, EAX[7:5] is not 000b, and any bit is clear in EAX[2:1] or EAX[7:5]; that is, software can enable the XSAVE feature set for AVX-512 state only if it does so for all three state components, and only if it also does so for AVX state and SSE state. This implies that the value of XCR0[7:5] is always either 000b or 111b.

As noted in Section 13.1, the processor will preserve AVX-512 state unmodified if software clears XCR0[7:5]. However, clearing XCR0[7:5] while AVX-512 state is not in its initial configuration may cause SSE and AVX instructions to incur a power and performance penalty. See Section 13.5.3, “Enable the Use Of XSAVE Feature Set And XSAVE State Components” of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*, for how system software can avoid this penalty.

- XCR0[9] is associated with PKRU state (see Section 13.5.7). Software can use the XSAVE feature set to manage PKRU state only if XCR0[9] = 1. The value of XCR0[9] in no way determines whether software can use protection keys or execute other instructions that access PKRU state (these instructions can be executed even if XCR0[9] = 0).

XCR0[9] is 0 coming out of RESET. As noted in Section 13.2, a processor allows software to set XCR0[9] if and only if CPUID.(EAX=0DH,ECX=0):EAX[9] = 1.

- XCR0[63:10] and XCR0[8] are reserved.<sup>1</sup> Executing the XSETBV instruction causes a general-protection fault (#GP) if ECX = 0 and any corresponding bit in EDX:EAX is not 0. These bits in XCR0 are all 0 coming out of RESET.

Software operating with CPL > 0 may need to determine whether the XSAVE feature set and certain XSAVE-enabled features have been enabled. If CPL > 0, execution of the MOV from CR4 instruction causes a general-protection fault (#GP). The following alternative mechanisms allow software to discover the enabling of the XSAVE feature set regardless of CPL:

- The value of CR4.OSXSAVE is returned in CPUID.1:ECX.OSXSAVE[bit 27]. If software determines that CPUID.1:ECX.OSXSAVE = 1, the processor supports the XSAVE feature set and the feature set has been enabled in CR4.
- Executing the XGETBV instruction with ECX = 0 returns the value of XCR0 in EDX:EAX. XGETBV can be executed if CR4.OSXSAVE = 1 (if CPUID.1:ECX.OSXSAVE = 1), regardless of CPL.

Thus, software can use the following algorithm to determine the support and enabling for the XSAVE feature set:

1. Use CPUID to discover the value of CPUID.1:ECX.OSXSAVE.
  - If the bit is 0, either the XSAVE feature set is not supported by the processor or has not been enabled by software. Either way, the XSAVE feature set is not available, nor are XSAVE-enabled features such as AVX.
  - If the bit is 1, the processor supports the XSAVE feature set — including the XGETBV instruction — and it has been enabled by software. The XSAVE feature set can be used to manage x87 state (because XCR0[0] is always 1). Software requiring more detailed information can go on to the next step.
2. Execute XGETBV with ECX = 0 to discover the value of XCR0. If XCR0[1] = 1, the XSAVE feature set can be used to manage SSE state. If XCR0[2] = 1, the XSAVE feature set can be used to manage AVX state and software can execute AVX instructions. If XCR0[4:3] is 11b, the XSAVE feature set can be used to manage MPX

---

1. If XCR0[3] = 0, executions of CALL, RET, JMP, and Jcc do not initialize the bounds registers.

1. Bit 8 corresponds to a supervisor state component. Since bits can be set in XCR0 only for user state components, that bit of XCR0 must be 0.

state and software can execute MPX instructions. If XCR0[7:5] is 111b, the XSAVE feature set can be used to manage AVX-512 state and software can execute AVX-512 instructions. If XCR0[9] = 1, the XSAVE feature set can be used to manage PKRU state.

The IA32\_XSS MSR (with MSR index DA0H) is zero coming out of RESET. If CR4.OSXSAVE = 1, CPUID.(EAX=0DH,ECX=1):EAX[3] = 1, and CPL = 0, executing the WRMSR instruction with ECX = DA0H writes the 64-bit value in EDX:EAX to the IA32\_XSS MSR (EAX is written to IA32\_XSS[31:0] and EDX to IA32\_XSS[63:32]). The following items provide details regarding individual bits in the IA32\_XSS MSR:

- IA32\_XSS[8] is associated with PT state (see Section 13.5.6). Software can use XSAVES and XRSTORS to manage PT state only if IA32\_XSS[8] = 1. The value of IA32\_XSS[8] does not determine whether software can use Intel Processor Trace (the feature can be used even if IA32\_XSS[8] = 0).
- IA32\_XSS[63:9] and IA32\_XSS[7:0] are reserved.<sup>1</sup> Executing the WRMSR instruction causes a general-protection fault (#GP) if ECX = DA0H and any corresponding bit in EDX:EAX is not 0. These bits in XCR0 are all 0 coming out of RESET.

The IA32\_XSS MSR is 0 coming out of RESET.

There is no mechanism by which software operating with CPL > 0 can discover the value of the IA32\_XSS MSR.

## 13.4 XSAVE AREA

The XSAVE feature set includes instructions that save and restore the XSAVE-managed state components to and from memory: XSAVE, XSAVEOPT, XSAVEC, and XSAVES (for saving); and XRSTOR and XRSTORS (for restoring). The processor organizes the state components in a region of memory called an **XSAVE area**. Each of the save and restore instructions takes a memory operand that specifies the 64-byte aligned base address of the XSAVE area on which it operates.

Every XSAVE area has the following format:

- The **legacy region**. The legacy region of an XSAVE area comprises the 512 bytes starting at the area’s base address. It is used to manage the state components for x87 state and SSE state. The legacy region is described in more detail in Section 13.4.1.
- The **XSAVE header**. The XSAVE header of an XSAVE area comprises the 64 bytes starting at an offset of 512 bytes from the area’s base address. The XSAVE header is described in more detail in Section 13.4.2.
- The **extended region**. The extended region of an XSAVE area starts at an offset of 576 bytes from the area’s base address. It is used to manage the state components other than those for x87 state and SSE state. The extended region is described in more detail in Section 13.4.3. The size of the extended region is determined by which state components the processor supports and which bits have been set in XCR0 and IA32\_XSS (see Section 13.3).

### 13.4.1 Legacy Region of an XSAVE Area

The legacy region of an XSAVE area comprises the 512 bytes starting at the area’s base address. It has the same format as the FXSAVE area (see Section 10.5.1). The XSAVE feature set uses the legacy area for x87 state (state component 0) and SSE state (state component 1). Table 13-1 illustrates the format of the first 416 bytes of the legacy region of an XSAVE area.

**Table 13-1. Format of the Legacy Region of an XSAVE Area**

15 14	13 12	11 10	9 8	7 6	5	4	3 2	1 0	
FIP[63:48] or reserved	FCS or FIP[47:32]	FIP[31:0]		FOP	Rsvd.	FTW	FSW	FCW	<b>0</b>
MXCSR_MASK		MXCSR		FDP[63:48] or reserved	FDS or FDP[47:32]		FDP[31:0]		<b>16</b>

1. Bit 9 and bits 7:0 correspond to user state components. Since bits can be set in the IA32\_XSS MSR only for supervisor state components, those bits of the MSR must be 0.

**Table 13-1. Format of the Legacy Region of an XSAVE Area (Contd.) (Contd.)**

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Reserved						ST0/MM0						32				
Reserved						ST1/MM1						48				
Reserved						ST2/MM2						64				
Reserved						ST3/MM3						80				
Reserved						ST4/MM4						96				
Reserved						ST5/MM5						112				
Reserved						ST6/MM6						128				
Reserved						ST7/MM7						144				
						XMM0						160				
						XMM1						176				
						XMM2						192				
						XMM3						208				
						XMM4						224				
						XMM5						240				
						XMM6						256				
						XMM7						272				
						XMM8						288				
						XMM9						304				
						XMM10						320				
						XMM11						336				
						XMM12						352				
						XMM13						368				
						XMM14						384				
						XMM15						400				

The x87 state component comprises bytes 23:0 and bytes 159:32. The SSE state component comprises bytes 31:24 and bytes 415:160. The XSAVE feature set does not use bytes 511:416; bytes 463:416 are reserved.

Section 13.7 through Section 13.9 provide details of how instructions in the XSAVE feature set use the legacy region of an XSAVE area.

### 13.4.2 XSAVE Header

The XSAVE header of an XSAVE area comprises the 64 bytes starting at offset 512 from the area's base address:

- Bytes 7:0 of the XSAVE header is a state-component bitmap (see Section 13.1) called **XSTATE\_BV**. It identifies the state components in the XSAVE area.
- Bytes 15:8 of the XSAVE header is a state-component bitmap called **XCOMP\_BV**. It is used as follows:
  - XCOMP\_BV[63] indicates the format of the extended region of the XSAVE area (see Section 13.4.3). If it is clear, the standard format is used. If it is set, the compacted format is used; XCOMP\_BV[62:0] provide format specifics as specified in Section 13.4.3.
  - XCOMP\_BV[63] determines which form of the XRSTOR instruction is used. If the bit is set, the compacted form is used; otherwise, the standard form is used. See Section 13.8.

- All bits in XCOMP\_BV should be 0 if the processor does not support the compaction extensions to the XSAVE feature set.
- Bytes 63:16 of the XSAVE header are reserved.

Section 13.7 through Section 13.9 provide details of how instructions in the XSAVE feature set use the XSAVE header of an XSAVE area.

### 13.4.3 Extended Region of an XSAVE Area

The extended region of an XSAVE area starts at byte offset 576 from the area's base address. The size of the extended region is determined by which state components the processor supports and which bits have been set in XCR0 | IA32\_XSS (see Section 13.3).

The XSAVE feature set uses the extended area for each state component  $i$ , where  $i \geq 2$ . The following state components are currently supported in the extended area: state component 2 contains AVX state; state components 5–7 contain AVX-512 state; and state component 9 contains PKRU state.

The extended region of the an XSAVE area may have one of two formats. The **standard format** is supported by all processors that support the XSAVE feature set; the **compacted format** is supported by those processors that support the compaction extensions to the XSAVE feature set (see Section 13.2). Bit 63 of the XCOMP\_BV field in the XSAVE header (see Section 13.4.2) indicates which format is used.

The following items describe the two possible formats of the extended region:

- **Standard format.** Each state component  $i$  ( $i \geq 2$ ) is located at the byte offset from the base address of the XSAVE area enumerated in CPUID.(EAX=0DH,ECX= $i$ ):EBX. (CPUID.(EAX=0DH,ECX= $i$ ):EAX enumerates the number of bytes required for state component  $i$ .)
- **Compacted format.** Each state component  $i$  ( $i \geq 2$ ) is located at a byte offset from the base address of the XSAVE area based on the XCOMP\_BV field in the XSAVE header:
  - If XCOMP\_BV[ $i$ ] = 0, state component  $i$  is not in the XSAVE area.
  - If XCOMP\_BV[ $i$ ] = 1, state component  $i$  is located at a byte offset  $location_i$  from the base address of the XSAVE area, where  $location_i$  is determined by the following items:
    - If XCOMP\_BV[ $j$ ] = 0 for every  $j$ ,  $2 \leq j < i$ ,  $location_i$  is 576. (This item applies if  $i$  is the first bit set in bits 62:2 of the XCOMP\_BV; it implies that state component  $i$  is located at the beginning of the extended region.)
    - Otherwise, let  $j$ ,  $2 \leq j < i$ , be the greatest value such that XCOMP\_BV[ $j$ ] = 1. Then  $location_i$  is determined by the following values:  $location_j$ ;  $size_j$ , as enumerated in CPUID.(EAX=0DH,ECX= $j$ ):EAX; and the value of  $align_i$ , as enumerated in CPUID.(EAX=0DH,ECX= $i$ ):ECX[1]:
      - If  $align_i = 0$ ,  $location_i = location_j + size_j$ . (This item implies that state component  $i$  is located immediately following the preceding state component whose bit is set in XCOMP\_BV.)
      - If  $align_i = 1$ ,  $location_i = \text{ceiling}(location_j + size_j, 64)$ . (This item implies that state component  $i$  is located on the next 64-byte boundary following the preceding state component whose bit is set in XCOMP\_BV.)

## 13.5 XSAVE-MANAGED STATE

The section provides details regarding how the XSAVE feature set interacts with the various XSAVE-managed state components.

Unless otherwise state, the state pertaining to a particular state component is saved beginning at byte 0 of the section of the XSAVE are corresponding to that state component.

### 13.5.1 x87 State

Instructions in the XSAVE feature set can manage the same state of the x87 FPU execution environment (**x87 state**) that can be managed using the FXSAVE and FXRSTOR instructions. They organize all x87 state as a user state component in the legacy region of the XSAVE area (see Section 13.4.1). This region is illustrated in Table 13-1; the x87 state is listed below, along with details of its interactions with the XSAVE feature set:

- Bytes 1:0, 3:2, 7:6. These are used for the x87 FPU Control Word (FCW), the x87 FPU Status Word (FSW), and the x87 FPU Opcode (FOP), respectively.
- Byte 4 is used for an abridged version of the x87 FPU Tag Word (FTW). The following items describe its usage:
  - For each  $j$ ,  $0 \leq j \leq 7$ , XSAVE, XSAVEOPT, XSAVEC, and XSAVES save a 0 into bit  $j$  of byte 4 if x87 FPU data register  $ST_j$  has an empty tag; otherwise, XSAVE, XSAVEOPT, XSAVEC, and XSAVES save a 1 into bit  $j$  of byte 4.
  - For each  $j$ ,  $0 \leq j \leq 7$ , XRSTOR and XRSTORS establish the tag value for x87 FPU data register  $ST_j$  as follows. If bit  $j$  of byte 4 is 0, the tag for  $ST_j$  in the tag register for that data register is marked empty (11B); otherwise, the x87 FPU sets the tag for  $ST_j$  based on the value being loaded into that register (see below).
- Bytes 15:8 are used as follows:
  - If the instruction has no REX prefix, or if  $REX.W = 0$ :
    - Bytes 11:8 are used for bits 31:0 of the x87 FPU Instruction Pointer Offset (FIP).
    - If  $CPUID.(EAX=07H,ECX=0H):EBX[\text{bit } 13] = 0$ , bytes 13:12 are used for x87 FPU Instruction Pointer Selector (FCS). Otherwise, XSAVE, XSAVEOPT, XSAVEC, and XSAVES save these bytes as 0000H, and XRSTOR and XRSTORS ignore them.
    - Bytes 15:14 are not used.
  - If the instruction has a REX prefix with  $REX.W = 1$ , bytes 15:8 are used for the full 64 bits of FIP.
- Bytes 23:16 are used as follows:
  - If the instruction has no REX prefix, or if  $REX.W = 0$ :
    - Bytes 19:16 are used for bits 31:0 of the x87 FPU Data Pointer Offset (FDP).
    - If  $CPUID.(EAX=07H,ECX=0H):EBX[\text{bit } 13] = 0$ , bytes 21:20 are used for x87 FPU Data Pointer Selector (FDS). Otherwise, XSAVE, XSAVEOPT, XSAVEC, and XSAVES save these bytes as 0000H; and XRSTOR and XRSTORS ignore them.
    - Bytes 23:22 are not used.
  - If the instruction has a REX prefix with  $REX.W = 1$ , bytes 23:16 are used for the full 64 bits of FDP.
- Bytes 31:24 are used for SSE state (see Section 13.5.2).
- Bytes 159:32 are used for the registers  $ST_0$ – $ST_7$  (MM0–MM7). Each of the 8 registers is allocated a 128-bit region, with the low 80 bits used for the register and the upper 48 bits unused.

x87 state is XSAVE-managed but the x87 FPU feature is not XSAVE-enabled. The XSAVE feature set can operate on x87 state only if the feature set is enabled ( $CR4.OSXSAVE = 1$ ).<sup>1</sup> Software can otherwise use x87 state even if the XSAVE feature set is not enabled.

### 13.5.2 SSE State

Instructions in the XSAVE feature set can manage the registers used by the streaming SIMD extensions (**SSE state**) just as the FXSAVE and FXRSTOR instructions do. They organize all SSE state as a user state component in the legacy region of the XSAVE area (see Section 13.4.1). This region is illustrated in Table 13-1; the SSE state is listed below, along with details of its interactions with the XSAVE feature set:

- Bytes 23:0 are used for x87 state (see Section 13.5.1).
- Bytes 27:24 are used for the MXCSR register. XRSTOR and XRSTORS generate general-protection faults (#GP) in response to attempts to set any of the reserved bits of the MXCSR register.<sup>2</sup>

1. The processor ensures that  $XCRO[0]$  is always 1.



- Bytes 31:28 are used for the MXCSR\_MASK value. XRSTOR and XRSTORS ignore this field.
- Bytes 159:32 are used for x87 state.
- Bytes 287:160 are used for the registers XMM0–XMM7.
- Bytes 415:288 are used for the registers XMM8–XMM15. These fields are used only in 64-bit mode. Executions of XSAVE, XSAVEOPT, XSAVEC, and XSAVES outside 64-bit mode do not modify these bytes; executions of XRSTOR and XRSTORS outside 64-bit mode do not update XMM8–XMM15. See Section 13.13.

SSE state is XSAVE-managed but the SSE feature is not XSAVE-enabled. The XSAVE feature set can operate on SSE state only if the feature set is enabled (CR4.OSXSAVE = 1) and has been configured to manage SSE state (XCR0[1] = 1). Software can otherwise use SSE state even if the XSAVE feature set is not enabled or has not been configured to manage SSE state.

### 13.5.3 AVX State

The register state used by the Intel® Advanced Vector Extensions (AVX) comprises the MXCSR register and 16 256-bit vector registers called YMM0–YMM15. The low 128 bits of each register YMM*i* is identical to the SSE register XMM*i*. Thus, the new state register state added by AVX comprises the upper 128 bits of the registers YMM0–YMM15. These 16 128-bit values are denoted YMM0\_H–YMM15\_H and are collectively called **AVX state**.

As noted in Section 13.1, the XSAVE feature set manages AVX state as user state component 2. Thus, AVX state is located in the extended region of the XSAVE area (see Section 13.4.3).

As noted in Section 13.2, CPUID.(EAX=0DH,ECX=2):EBX enumerates the offset (in bytes, from the base of the XSAVE area) of the section of the extended region of the XSAVE area used for AVX state (when the standard format of the extended region is used). CPUID.(EAX=0DH,ECX=2):EAX enumerates the size (in bytes) required for AVX state.

The XSAVE feature set partitions YMM0\_H–YMM15\_H in a manner similar to that used for the XMM registers (see Section 13.5.2). Bytes 127:0 of the AVX-state section are used for YMM0\_H–YMM7\_H. Bytes 255:128 are used for YMM8\_H–YMM15\_H, but they are used only in 64-bit mode. Executions of XSAVE, XSAVEOPT, XSAVEC, and XSAVES outside 64-bit mode do not modify bytes 255:128; executions of XRSTOR and XRSTORS outside 64-bit mode do not update YMM8\_H–YMM15\_H. See Section 13.13. In general, bytes 16*i*+15:16*i* are used for YMM*i*\_H (for 0 ≤ *i* ≤ 15).

AVX state is XSAVE-managed and the AVX feature is XSAVE-enabled. The XSAVE feature set can operate on AVX state only if the feature set is enabled (CR4.OSXSAVE = 1) and has been configured to manage AVX state (XCR0[2] = 1). AVX instructions cannot be used unless the XSAVE feature set is enabled and has been configured to manage AVX state.

### 13.5.4 MPX State

The register state used by the Intel® Memory Protection Extensions (MPX) comprises the 4 128-bit bounds registers BND0–BND3 (**BNDREG state**); and the 64-bit user-mode configuration register BNDCFGU and the 64-bit MPX status register BNDSTATUS (collectively, **BNDCSR state**). Together, these two user state components compose **MPX state**.

As noted in Section 13.1, the XSAVE feature set manages MPX state as state components 3–4. Thus, MPX state is located in the extended region of the XSAVE area (see Section 13.4.3). The following items detail how these state components are organized in this region:

- **BNDREG state.**  
As noted in Section 13.2, CPUID.(EAX=0DH,ECX=3):EBX enumerates the offset (in bytes, from the base of the XSAVE area) of the section of the extended region of the XSAVE area used for BNDREG state (when the standard format of the extended region is used). CPUID.(EAX=0DH,ECX=5):EAX enumerates the size (in bytes) required for BNDREG state. The BNDREG section is used for the 4 128-bit bound registers BND0–BND3, with bytes 16*i*+15:16*i* being used for BND*i*.

---

2. While MXCSR and MXCSR\_MASK are part of SSE state, their treatment by the XSAVE feature set is not the same as that of the XMM registers. See Section 13.7 through Section 13.11 for details.

- **BNDCSR state.**

As noted in Section 13.2, CPUID.(EAX=0DH,ECX=4):EBX enumerates the offset of the section of the extended region of the XSAVE area used for BNDCSR state (when the standard format of the extended region is used). CPUID.(EAX=0DH,ECX=6):EAX enumerates the size (in bytes) required for BNDCSR state. In the BNDSCR section, bytes 7:0 are used for BNDCFGU and bytes 15:8 are used for BNDSTATUS.

Both components of MPX state are XSAVE-managed and the MPX feature is XSAVE-enabled. The XSAVE feature set can operate on MPX state only if the feature set is enabled (CR4.OSXSAVE = 1) and has been configured to manage MPX state (XCR0[4:3] = 11b). MPX instructions cannot be used unless the XSAVE feature set is enabled and has been configured to manage MPX state.

### 13.5.5 AVX-512 State

The register state used by the Intel® Advanced Vector Extensions 512 (AVX-512) comprises the MXCSR register, the 8 64-bit opmask registers k0–k7, and 32 512-bit vector registers called ZMM0–ZMM31. For each  $i$ ,  $0 \leq i \leq 15$ , the low 256 bits of register ZMM*i* is identical to the AVX register YMM*i*. Thus, the new state register state added by AVX comprises the following user state components:

- The opmask registers, collectively called **opmask state**.
- The upper 256 bits of the registers ZMM0–ZMM15. These 16 256-bit values are denoted ZMM0\_H–ZMM15\_H and are collectively called **ZMM\_Hi256 state**.
- The 16 512-bit registers ZMM16–ZMM31, collectively called **Hi16\_ZMM state**.

Together, these three state components compose **AVX-512 state**.

As noted in Section 13.1, the XSAVE feature set manages AVX-512 state as state components 5–7. Thus, AVX-512 state is located in the extended region of the XSAVE area (see Section 13.4.3). The following items detail how these state components are organized in this region:

- **Opmask state.**

As noted in Section 13.2, CPUID.(EAX=0DH,ECX=5):EBX enumerates the offset (in bytes, from the base of the XSAVE area) of the section of the extended region of the XSAVE area used for opmask state (when the standard format of the extended region is used). CPUID.(EAX=0DH,ECX=5):EAX enumerates the size (in bytes) required for opmask state. The opmask section is used for the 8 64-bit bound registers k0–k7, with bytes  $8i+7:8i$  being used for  $ki$ .

- **ZMM\_Hi256 state.**

As noted in Section 13.2, CPUID.(EAX=0DH,ECX=6):EBX enumerates the offset of the section of the extended region of the XSAVE area used for ZMM\_Hi256 state (when the standard format of the extended region is used). CPUID.(EAX=0DH,ECX=6):EAX enumerates the size (in bytes) required for ZMM\_Hi256 state.

The XSAVE feature set partitions ZMM0\_H–ZMM15\_H in a manner similar to that used for the XMM registers (see Section 13.5.2). Bytes 255:0 of the ZMM\_Hi256-state section are used for ZMM0\_H–ZMM7\_H. Bytes 511:256 are used for ZMM8\_H–ZMM15\_H, but they are used only in 64-bit mode. Executions of XSAVE, XSAVEOPT, XSAVEC, and XSAVES outside 64-bit mode do not modify bytes 511:256; executions of XRSTOR and XRSTORS outside 64-bit mode do not update ZMM8\_H–ZMM15\_H. See Section 13.13. In general, bytes  $32i+31:32i$  are used for ZMM*i*\_H (for  $0 \leq i \leq 15$ ).

- **Hi16\_ZMM state.**

As noted in Section 13.2, CPUID.(EAX=0DH,ECX=7):EBX enumerates the offset of the section of the extended region of the XSAVE area used for Hi16\_ZMM state (when the standard format of the extended region is used). CPUID.(EAX=0DH,ECX=7):EAX enumerates the size (in bytes) required for Hi16\_ZMM state.

The XSAVE feature set accesses Hi16\_ZMM state only in 64-bit mode. Executions of XSAVE, XSAVEOPT, XSAVEC, and XSAVES outside 64-bit mode do not modify the Hi16\_ZMM section; executions of XRSTOR and XRSTORS outside 64-bit mode do not update ZMM16–ZMM31. See Section 13.13. In general, bytes  $64(i-16)+63:64(i-16)$  are used for ZMM*i* (for  $16 \leq i \leq 31$ ).

All three components of AVX-512 state are XSAVE-managed and the AVX-512 feature is XSAVE-enabled. The XSAVE feature set can operate on AVX-512 state only if the feature set is enabled (CR4.OSXSAVE = 1) and has been configured to manage AVX-512 state (XCR0[7:5] = 111b). AVX-512 instructions cannot be used unless the XSAVE feature set is enabled and has been configured to manage AVX-512 state.



### 13.5.6 PT State

The register state used by Intel Processor Trace (**PT state**) comprises the following 9 MSRs: IA32\_RTIT\_CTL, IA32\_RTIT\_OUTPUT\_BASE, IA32\_RTIT\_OUTPUT\_MASK\_PTRS, IA32\_RTIT\_STATUS, IA32\_RTIT\_CR3\_MATCH, IA32\_RTIT\_ADDR0\_A, IA32\_RTIT\_ADDR0\_B, IA32\_RTIT\_ADDR1\_A, and IA32\_RTIT\_ADDR1\_B.<sup>1</sup>

As noted in Section 13.1, the XSAVE feature set manages PT state as supervisor state component 8. Thus, PT state is located in the extended region of the XSAVE area (see Section 13.4.3). As noted in Section 13.2, CPUID.(EAX=0DH,ECX=8):EAX enumerates the size (in bytes) required for PT state. The MSRs are each allocated 8 bytes in the state component in the order given above. Thus, IA32\_RTIT\_CTL is at byte offset 0, IA32\_RTIT\_OUTPUT\_BASE at byte offset 8, etc. Any locations in the state component at or beyond byte offset 72 are reserved.

PT state is XSAVE-managed but Intel Processor Trace is not XSAVE-enabled. The XSAVE feature set can operate on PT state only if the feature set is enabled (CR4.OSXSAVE = 1) and has been configured to manage PT state (IA32\_XSS[8] = 1). Software can otherwise use Intel Processor Trace and access its MSRs (using RDMSR and WRMSR) even if the XSAVE feature set is not enabled or has not been configured to manage PT state.

The following items describe special treatment of PT state by the XSAVES and XRSTORS instructions:

- If XSAVES saves PT state, the instruction clears IA32\_RTIT\_CTL.TraceEn (bit 0) after saving the value of the IA32\_RTIT\_CTL MSR and before saving any other PT state. If XSAVES causes a fault or a VM exit, it restores IA32\_RTIT\_CTL.TraceEn to its original value.
- If XSAVES saves PT state, the instruction saves zeroes in the reserved portions of the state component.
- If XRSTORS would restore (or initialize) PT state and IA32\_RTIT\_CTL.TraceEn = 1, the instruction causes a general-protection exception (#GP) before modifying PT state.
- If XRSTORS causes an exception or a VM exit, it does so before any modification to IA32\_RTIT\_CTL.TraceEn (even if it has loaded other PT state).

### 13.5.7 PKRU State

The register state used by the protection-key feature (**PKRU state**) is the 32-bit PKRU register. As noted in Section 13.1, the XSAVE feature set manages PKRU state as user state component 9. Thus, PKRU state is located in the extended region of the XSAVE area (see Section 13.4.3).

As noted in Section 13.2, CPUID.(EAX=0DH,ECX=9):EBX enumerates the offset (in bytes, from the base of the XSAVE area) of the section of the extended region of the XSAVE area used for PKRU state (when the standard format of the extended region is used). CPUID.(EAX=0DH,ECX=9):EAX enumerates the size (in bytes) required for PKRU state. The XSAVE feature set uses bytes 3:0 of the PK-state section for the PKRU register.

PKRU state is XSAVE-managed but the protection-key feature is not XSAVE-enabled. The XSAVE feature set can operate on PKRU state only if the feature set is enabled (CR4.OSXSAVE = 1) and has been configured to manage PKRU state (XCR0[9] = 1). Software can otherwise use protection keys and access PKRU state even if the XSAVE feature set is not enabled or has not been configured to manage PKRU state.

The value of the PKRU register determines the access rights for user-mode linear addresses. (See Section 4.6, "Access Rights," of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.) The access rights that pertain to an execution of the XRSTOR and XRSTORS instructions are determined by the value of the register before the execution and not by any value that the execution might load into the PKRU register.

## 13.6 PROCESSOR TRACKING OF XSAVE-MANAGED STATE

The XSAVEOPT, XSAVEC, and XSAVES instructions use two optimizations to reduce the amount of data that they write to memory. They avoid writing data for any state component known to be in its initial configuration (the **init optimization**). In addition, if either XSAVEOPT or XSAVES is using the same XSAVE area as that used by the most

1. These MSRs might not be supported by every processor that supports Intel Processor Trace. Software can use the CPUID instruction to discover which are supported; see Section 36.3.1, "Detection of Intel Processor Trace and Capability Enumeration," of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C*.

recent execution of XRSTOR or XRSTORS, it may avoid writing data for any state component whose configuration is known not to have been modified since then (the **modified optimization**). (XSAVE does not use these optimizations, and XSAVEC does not use the modified optimization.) The operation of XSAVEOPT, XSAVEC, and XSAVES are described in more detail in Section 13.9 through Section 13.11.

A processor can support the init and modified optimizations with special hardware that tracks the state components that might benefit from those optimizations. Other implementations might not include such hardware; such a processor would always consider each such state component as not in its initial configuration and as modified since the last execution of XRSTOR or XRSTORS.

The following notation describes the state of the init and modified optimizations:

- **XINUSE** denotes the state-component bitmap corresponding to the init optimization. If  $XINUSE[i] = 0$ , state component  $i$  is known to be in its initial configuration; otherwise  $XINUSE[i] = 1$ . It is possible for  $XINUSE[i]$  to be 1 even when state component  $i$  is in its initial configuration. On a processor that does not support the init optimization,  $XINUSE[i]$  is always 1 for every value of  $i$ .

Executing XGETBV with  $ECX = 1$  returns in  $EDX:EAX$  the logical-AND of  $XCR0$  and the current value of the  $XINUSE$  state-component bitmap. Such an execution of XGETBV always sets  $EAX[1]$  to 1 if  $XCR0[1] = 1$  and  $MXCSR$  does not have its RESET value of 1F80H. Section 13.2 explains how software can determine whether a processor supports this use of XGETBV.

- **XMODIFIED** denotes the state-component bitmap corresponding to the modified optimization. If  $XMODIFIED[i] = 0$ , state component  $i$  is known not to have been modified since the most recent execution of XRSTOR or XRSTORS; otherwise  $XMODIFIED[i] = 1$ . It is possible for  $XMODIFIED[i]$  to be 1 even when state component  $i$  has not been modified since the most recent execution of XRSTOR or XRSTORS. On a processor that does not support the modified optimization,  $XMODIFIED[i]$  is always 1 for every value of  $i$ .

A processor that implements the modified optimization saves information about the most recent execution of XRSTOR or XRSTORS in a quantity called **XRSTOR\_INFO**, a 4-tuple containing the following: (1) the CPL; (2) whether the logical processor was in VMX non-root operation; (3) the linear address of the XSAVE area; and (4) the  $XCOMP\_BV$  field in the XSAVE area. An execution of XSAVEOPT or XSAVES uses the modified optimization only if that execution corresponds to XRSTOR\_INFO on these four parameters.

This mechanism implies that, depending on details of the operating system, the processor might determine that an execution of XSAVEOPT by one user application corresponds to an earlier execution of XRSTOR by a different application. For this reason, Intel recommends the application software not use the XSAVEOPT instruction.

The following items specify the initial configuration each state component (for the purposes of defining the  $XINUSE$  bitmap):

- **x87 state.** x87 state is in its initial configuration if the following all hold:  $FCW$  is 037FH;  $FSW$  is 0000H;  $FTW$  is FFFFH;  $FCS$  and  $FDS$  are each 0000H;  $FIP$  and  $FDP$  are each 00000000\_00000000H; each of  $ST0$ – $ST7$  is 0000\_00000000\_00000000H.
- **SSE state.** In 64-bit mode, SSE state is in its initial configuration if each of  $XMM0$ – $XMM15$  is 0. Outside 64-bit mode, SSE state is in its initial configuration if each of  $XMM0$ – $XMM7$  is 0.  $XINUSE[1]$  pertains only to the state of the XMM registers and not to  $MXCSR$ . An execution of XRSTOR or XRSTORS outside 64-bit mode does not update  $XMM8$ – $XMM15$ . (See Section 13.13.)
- **AVX state.** In 64-bit mode, AVX state is in its initial configuration if each of  $YMM0\_H$ – $YMM15\_H$  is 0. Outside 64-bit mode, AVX state is in its initial configuration if each of  $YMM0\_H$ – $YMM7\_H$  is 0. An execution of XRSTOR or XRSTORS outside 64-bit mode does not update  $YMM8\_H$ – $YMM15\_H$ . (See Section 13.13.)
- **BNDREG state.** BNDREG state is in its initial configuration if the value of each of  $BND0$ – $BND3$  is 0.
- **BNDCSR state.** BNDCSR state is in its initial configuration if  $BNDCFGU$  and  $BNDCSR$  each has value 0.
- **Opmask state.** Opmask state is in its initial configuration if each of the opmask registers  $k0$ – $k7$  is 0.
- **ZMM\_Hi256 state.** In 64-bit mode,  $ZMM\_Hi256$  state is in its initial configuration if each of  $ZMM0\_H$ – $ZMM15\_H$  is 0. Outside 64-bit mode,  $ZMM\_Hi256$  state is in its initial configuration if each of  $ZMM0\_H$ – $ZMM7\_H$  is 0. An execution of XRSTOR or XRSTORS outside 64-bit mode does not update  $ZMM8\_H$ – $ZMM15\_H$ . (See Section 13.13.)
- **Hi16\_ZMM state.** In 64-bit mode,  $Hi16\_ZMM$  state is in its initial configuration if each of  $ZMM16$ – $ZMM31$  is 0. Outside 64-bit mode,  $Hi16\_ZMM$  state is always in its initial configuration. An execution of XRSTOR or XRSTORS outside 64-bit mode does not update  $ZMM31$ – $ZMM31$ . (See Section 13.13.)

- **PT state.** PT state is in its initial configuration if each of the 9 MSR is 0.
- **PKRU state.** PKRU state is in its initial configuration if the value of the PKRU is 0.

## 13.7 OPERATION OF XSAVE

The XSAVE instruction takes a single memory operand, which is an XSAVE area. In addition, the register pair EDX:EAX is an implicit operand used as a state-component bitmap (see Section 13.1) called the **instruction mask**. The logical-AND of XCR0 and the instruction mask is the **requested-feature bitmap (RFBM)** of the user state components to be saved.

The following conditions cause execution of the XSAVE instruction to generate a fault:

- If the XSAVE feature set is not enabled ( $CR4.OSXSAVE = 0$ ), an invalid-opcode exception (#UD) occurs.
- If  $CR0.TS[\text{bit } 3]$  is 1, a device-not-available exception (#NM) occurs.
- If the address of the XSAVE area is not 64-byte aligned, a general-protection exception (#GP) occurs.<sup>1</sup>

If none of these conditions cause a fault, execution of XSAVE reads the XSTATE\_BV field of the XSAVE header (see Section 13.4.2) and writes it back to memory, setting  $XSTATE\_BV[i]$  ( $0 \leq i \leq 63$ ) as follows:

- If  $RFBM[i] = 0$ ,  $XSTATE\_BV[i]$  is not changed.
- If  $RFBM[i] = 1$ ,  $XSTATE\_BV[i]$  is set to the value of  $XINUSE[i]$ . Section 13.6 defines XINUSE to describe the processor init optimization and specifies the initial configuration of each state component. The nature of that optimization implies the following:
  - If state component  $i$  is in its initial configuration,  $XINUSE[i]$  may be either 0 or 1, and  $XSTATE\_BV[i]$  may be written with either 0 or 1.  
 $XINUSE[1]$  pertains only to the state of the XMM registers and not to MXCSR. Thus,  $XSTATE\_BV[1]$  may be written with 0 even if MXCSR does not have its RESET value of 1F80H.
  - If state component  $i$  is not in its initial configuration,  $XINUSE[i] = 1$  and  $XSTATE\_BV[i]$  is written with 1.  
 (As explained in Section 13.6, the initial configurations of some state components may depend on whether the processor is in 64-bit mode.)

The XSAVE instruction does not write any part of the XSAVE header other than the XSTATE\_BV field; in particular, it does **not** write to the XCOMP\_BV field.

Execution of XSAVE saves into the XSAVE area those state components corresponding to bits that are set in RFBM. State components 0 and 1 are located in the legacy region of the XSAVE area (see Section 13.4.1). Each state component  $i$ ,  $2 \leq i \leq 62$ , is located in the extended region; the XSAVE instruction always uses the standard format for the extended region (see Section 13.4.3).

The MXCSR register and MXCSR\_MASK are part of SSE state (see Section 13.5.2) and are thus associated with  $RFBM[1]$ . However, the XSAVE instruction also saves these values when  $RFBM[2] = 1$  (even if  $RFBM[1] = 0$ ).

See Section 13.5 for specifics for each state component and for details regarding mode-specific operation and operation determined by instruction prefixes. See Section 13.13 for details regarding faults caused by memory accesses.

## 13.8 OPERATION OF XRSTOR

The XRSTOR instruction takes a single memory operand, which is an XSAVE area. In addition, the register pair EDX:EAX is an implicit operand used as a state-component bitmap (see Section 13.1) called the **instruction mask**. The logical-AND of XCR0 and the instruction mask is the **requested-feature bitmap (RFBM)** of the user state components to be restored.

The following conditions cause execution of the XRSTOR instruction to generate a fault:

- If the XSAVE feature set is not enabled ( $CR4.OSXSAVE = 0$ ), an invalid-opcode exception (#UD) occurs.

1. If  $CR0.AM = 1$ ,  $CPL = 3$ , and  $EFLAGS.AC = 1$ , an alignment-check exception (#AC) may occur instead of #GP.

- If CR0.TS[bit 3] is 1, a device-not-available exception (#NM) occurs.
- If the address of the XSAVE area is not 64-byte aligned, a general-protection exception (#GP) occurs.<sup>1</sup>

After checking for these faults, the XRSTOR instruction reads the XCOMP\_BV field in the XSAVE area's XSAVE header (see Section 13.4.2). If XCOMP\_BV[63] = 0, the **standard form of XRSTOR** is executed (see Section 13.8.1); otherwise, the **compacted form of XRSTOR** is executed (see Section 13.8.2).<sup>2</sup>

See Section 13.2 for details of how to determine whether the compacted form of XRSTOR is supported.

### 13.8.1 Standard Form of XRSTOR

The standard form of XRSTOR performs additional fault checking. Either of the following conditions causes a general-protection exception (#GP):

- The XSTATE\_BV field of the XSAVE header sets a bit that is not set in XCR0.
- Bytes 23:8 of the XSAVE header are not all 0 (this implies that all bits in XCOMP\_BV are 0).<sup>3</sup>

If none of these conditions cause a fault, the processor updates each state component  $i$  for which RFBM[ $i$ ] = 1. XRSTOR updates state component  $i$  based on the value of bit  $i$  in the XSTATE\_BV field of the XSAVE header:

- If XSTATE\_BV[ $i$ ] = 0, the state component is set to its initial configuration. Section 13.6 specifies the initial configuration of each state component.

The initial configuration of state component 1 pertains only to the XMM registers and not to MXCSR. See below for the treatment of MXCSR

- If XSTATE\_BV[ $i$ ] = 1, the state component is loaded with data from the XSAVE area. See Section 13.5 for specifics for each state component and for details regarding mode-specific operation and operation determined by instruction prefixes. See Section 13.13 for details regarding faults caused by memory accesses.

State components 0 and 1 are located in the legacy region of the XSAVE area (see Section 13.4.1). Each state component  $i$ ,  $2 \leq i \leq 62$ , is located in the extended region; the standard form of XRSTOR uses the standard format for the extended region (see Section 13.4.3).

The MXCSR register is part of state component 1, SSE state (see Section 13.5.2). However, the standard form of XRSTOR loads the MXCSR register from memory whenever the RFBM[1] (SSE) or RFBM[2] (AVX) is set, regardless of the values of XSTATE\_BV[1] and XSTATE\_BV[2]. The standard form of XRSTOR causes a general-protection exception (#GP) if it would load MXCSR with an illegal value.

### 13.8.2 Compacted Form of XRSTOR

The compacted form of XRSTOR performs additional fault checking. Any of the following conditions causes a #GP:

- The XCOMP\_BV field of the XSAVE header sets a bit in the range 62:0 that is not set in XCR0.
- The XSTATE\_BV field of the XSAVE header sets a bit (including bit 63) that is not set in XCOMP\_BV.
- Bytes 63:16 of the XSAVE header are not all 0.

If none of these conditions cause a fault, the processor updates each state component  $i$  for which RFBM[ $i$ ] = 1. XRSTOR updates state component  $i$  based on the value of bit  $i$  in the XSTATE\_BV field of the XSAVE header:

- If XSTATE\_BV[ $i$ ] = 0, the state component is set to its initial configuration. Section 13.6 specifies the initial configuration of each state component.

---

1. If CR0.AM = 1, CPL = 3, and EFLAGS.AC = 1, an alignment-check exception (#AC) may occur instead of #GP.

2. If the processor does not support the compacted form of XRSTOR, it may execute the standard form of XRSTOR without first reading the XCOMP\_BV field. A processor supports the compacted form of XRSTOR only if it enumerates CPUID.(EAX=0DH,ECX=1):EAX[1] as 1.

3. Bytes 63:24 of the XSAVE header are also reserved. Software should ensure that bytes 63:16 of the XSAVE header are all 0 in any XSAVE area. (Bytes 15:8 should also be 0 if the XSAVE area is to be used on a processor that does not support the compaction extensions to the XSAVE feature set.)

If `XSTATE_BV[1] = 0`, the compacted form `XRSTOR` initializes `MXCSR` to `1F80H`. (This differs from the standard form of `XRSTOR`, which loads `MXCSR` from the `XSAVE` area whenever either `RFBM[1]` or `RFBM[2]` is set.)

State component  $i$  is set to its initial configuration as indicated above if `RFBM[ $i$ ] = 1` and `XSTATE_BV[ $i$ ] = 0` — even if `XCOMP_BV[ $i$ ] = 0`. This is true for all values of  $i$ , including 0 (x87 state) and 1 (SSE state).

- If `XSTATE_BV[ $i$ ] = 1`, the state component is loaded with data from the `XSAVE` area.<sup>1</sup> See Section 13.5 for specifics for each state component and for details regarding mode-specific operation and operation determined by instruction prefixes. See Section 13.13 for details regarding faults caused by memory accesses.

State components 0 and 1 are located in the legacy region of the `XSAVE` area (see Section 13.4.1). Each state component  $i$ ,  $2 \leq i \leq 62$ , is located in the extended region; the compacted form of the `XRSTOR` instruction uses the compacted format for the extended region (see Section 13.4.3).

The `MXCSR` register is part of SSE state (see Section 13.5.2) and is thus loaded from memory if `RFBM[1] = XSTATE_BV[1] = 1`. The compacted form of `XRSTOR` does not consider `RFBM[2]` (AVX) when determining whether to update `MXCSR`. (This is a difference from the standard form of `XRSTOR`.) The compacted form of `XRSTOR` causes a general-protection exception (`#GP`) if it would load `MXCSR` with an illegal value.

### 13.8.3 XRSTOR and the Init and Modified Optimizations

Execution of the `XRSTOR` instruction causes the processor to update its tracking for the init and modified optimizations (see Section 13.6). The following items provide details:

- The processor updates its tracking for the init optimization as follows:
  - If `RFBM[ $i$ ] = 0`, `XINUSE[ $i$ ]` is not changed.
  - If `RFBM[ $i$ ] = 1` and `XSTATE_BV[ $i$ ] = 0`, state component  $i$  may be tracked as init; `XINUSE[ $i$ ]` may be set to 0 or 1. (As noted in Section 13.6, a processor need not implement the init optimization for state component  $i$ ; a processor that does not do so implicitly maintains `XINUSE[ $i$ ] = 1` at all times.)
  - If `RFBM[ $i$ ] = 1` and `XSTATE_BV[ $i$ ] = 1`, state component  $i$  is tracked as not init; `XINUSE[ $i$ ]` is set to 1.
- The processor updates its tracking for the modified optimization and records information about the `XRSTOR` execution for future interaction with the `XSAVEOPT` and `XSAVES` instructions (see Section 13.9 and Section 13.11) as follows:
  - If `RFBM[ $i$ ] = 0`, state component  $i$  is tracked as modified; `XMODIFIED[ $i$ ]` is set to 1.
  - If `RFBM[ $i$ ] = 1`, state component  $i$  may be tracked as unmodified; `XMODIFIED[ $i$ ]` may be set to 0 or 1. (As noted in Section 13.6, a processor need not implement the modified optimization for state component  $i$ ; a processor that does not do so implicitly maintains `XMODIFIED[ $i$ ] = 1` at all times.)
  - `XRSTOR_INFO` is set to the 4-tuple  $\langle w, x, y, z \rangle$ , where  $w$  is the CPL (0);  $x$  is 1 if the logical processor is in VMX non-root operation and 0 otherwise;  $y$  is the linear address of the `XSAVE` area; and  $z$  is `XCOMP_BV`. In particular, the standard form of `XRSTOR` always sets  $z$  to all zeroes, while the compacted form of `XRSTORS` never does so (because it sets at least bit 63 to 1).

## 13.9 OPERATION OF XSAVEOPT

The operation of `XSAVEOPT` is similar to that of `XSAVE`. Unlike `XSAVE`, `XSAVEOPT` uses the init optimization (by which it may omit saving state components that are in their initial configuration) and the modified optimization (by which it may omit saving state components that have not been modified since the last execution of `XRSTOR`); see Section 13.6. See Section 13.2 for details of how to determine whether `XSAVEOPT` is supported.

The `XSAVEOPT` instruction takes a single memory operand, which is an `XSAVE` area. In addition, the register pair `EDX:EAX` is an implicit operand used as a state-component bitmap (see Section 13.1) called the **instruction mask**. The logical (bitwise) AND of `XCR0` and the instruction mask is the **requested-feature bitmap (RFBM)** of the user state components to be saved.

---

1. Earlier fault checking ensured that, if the instruction has reached this point in execution and `XSTATE_BV[ $i$ ] = 1`, then `XCOMP_BV[ $i$ ] = 1` also 1.



The following conditions cause execution of the XSAVEOPT instruction to generate a fault:

- If the XSAVE feature set is not enabled ( $CR4.OSXSAVE = 0$ ), an invalid-opcode exception (#UD) occurs.
- If  $CR0.TS[\text{bit } 3]$  is 1, a device-not-available exception (#NM) occurs.
- If the address of the XSAVE area is not 64-byte aligned, a general-protection exception (#GP) occurs.<sup>1</sup>

If none of these conditions cause a fault, execution of XSAVEOPT reads the  $XSTATE\_BV$  field of the XSAVE header (see Section 13.4.2) and writes it back to memory, setting  $XSTATE\_BV[i]$  ( $0 \leq i \leq 63$ ) as follows:

- If  $RFBM[i] = 0$ ,  $XSTATE\_BV[i]$  is not changed.
- If  $RFBM[i] = 1$ ,  $XSTATE\_BV[i]$  is set to the value of  $XINUSE[i]$ . Section 13.6 defines  $XINUSE$  to describe the processor init optimization and specifies the initial configuration of each state component. The nature of that optimization implies the following:
  - If the state component is in its initial configuration,  $XINUSE[i]$  may be either 0 or 1, and  $XSTATE\_BV[i]$  may be written with either 0 or 1.  
 $XINUSE[1]$  pertains only to the state of the XMM registers and not to MXCSR. Thus,  $XSTATE\_BV[1]$  may be written with 0 even if MXCSR does not have its RESET value of 1F80H.
  - If the state component is not in its initial configuration,  $XSTATE\_BV[i]$  is written with 1.  
 (As explained in Section 13.6, the initial configurations of some state components may depend on whether the processor is in 64-bit mode.)

The XSAVEOPT instruction does not write any part of the XSAVE header other than the  $XSTATE\_BV$  field; in particular, it does not write to the  $XCOMP\_BV$  field.

Execution of XSAVEOPT saves into the XSAVE area those state components corresponding to bits that are set in  $RFBM$  (subject to the optimizations described below). State components 0 and 1 are located in the legacy region of the XSAVE area (see Section 13.4.1). Each state component  $i$ ,  $2 \leq i \leq 62$ , is located in the extended region; the XSAVEOPT instruction always uses the standard format for the extended region (see Section 13.4.3).

See Section 13.5 for specifics for each state component and for details regarding mode-specific operation and operation determined by instruction prefixes. See Section 13.13 for details regarding faults caused by memory accesses.

Execution of XSAVEOPT performs two optimizations that reduce the amount of data written to memory:

- **Init optimization.**  
 If  $XINUSE[i] = 0$ , state component  $i$  is not saved to the XSAVE area (even if  $RFBM[i] = 1$ ). (See below for exceptions made for MXCSR.)
- **Modified optimization.**  
 Each execution of  $XRSTOR$  and  $XRSTORS$  establishes  $XRSTOR\_INFO$  as a 4-tuple  $\langle w, x, y, z \rangle$  (see Section 13.8.3 and Section 13.12). Execution of XSAVEOPT uses the modified optimization only if the following all hold for the current value of  $XRSTOR\_INFO$ :
  - $w = CPL$ ;
  - $x = 1$  if and only if the logical processor is in VMX non-root operation;
  - $y$  is the linear address of the XSAVE area being used by XSAVEOPT; and
  - $z$  is 00000000\_00000000H. (This last item implies that XSAVEOPT does not use the modified optimization if the last execution of  $XRSTOR$  used the compacted form, or if an execution of  $XRSTORS$  followed the last execution of  $XRSTOR$ .)

If XSAVEOPT uses the modified optimization and  $XMODIFIED[i] = 0$  (see Section 13.6), state component  $i$  is not saved to the XSAVE area.

(In practice, the benefit of the modified optimization for state component  $i$  depends on how the processor is tracking state component  $i$ ; see Section 13.6. Limitations on the tracking ability may result in state component  $i$  being saved even though it is in the same configuration that was loaded by the previous execution of  $XRSTOR$ .)

---

1. If  $CR0.AM = 1$ ,  $CPL = 3$ , and  $EFLAGS.AC = 1$ , an alignment-check exception (#AC) may occur instead of #GP.

Depending on details of the operating system, an execution of XSAVEOPT by a user application might use the modified optimization when the most recent execution of XRSTOR was by a different application. Because of this, Intel recommends the application software not use the XSAVEOPT instruction.

The MXCSR register and MXCSR\_MASK are part of SSE state (see Section 13.5.2) and are thus associated with bit 1 of RFBM. However, the XSAVEOPT instruction also saves these values when RFBM[2] = 1 (even if RFBM[1] = 0). The init and modified optimizations do not apply to the MXCSR register and MXCSR\_MASK.

## 13.10 OPERATION OF XSAVEC

The operation of XSAVEC is similar to that of XSAVE. Two main differences are (1) XSAVEC uses the compacted format for the extended region of the XSAVE area; and (2) XSAVEC uses the init optimization (see Section 13.6). Unlike XSAVEOPT, XSAVEC does not use the modified optimization. See Section 13.2 for details of how to determine whether XSAVEC is supported.

The XSAVEC instruction takes a single memory operand, which is an XSAVE area. In addition, the register pair EDX:EAX is an implicit operand used as a state-component bitmap (see Section 13.1) called the **instruction mask**. The logical (bitwise) AND of XCR0 and the instruction mask is the **requested-feature bitmap (RFBM)** of the user state components to be saved.

The following conditions cause execution of the XSAVEC instruction to generate a fault:

- If the XSAVE feature set is not enabled (CR4.OSXSAVE = 0), an invalid-opcode exception (#UD) occurs.
- If CR0.TS[bit 3] is 1, a device-not-available exception (#NM) occurs.
- If the address of the XSAVE area is not 64-byte aligned, a general-protection exception (#GP) occurs.<sup>1</sup>

If none of these conditions cause a fault, execution of XSAVEC writes the XSTATE\_BV field of the XSAVE header (see Section 13.4.2), setting XSTATE\_BV[*i*] ( $0 \leq i \leq 63$ ) as follows:<sup>2</sup>

- If RFBM[*i*] = 0, XSTATE\_BV[*i*] is written as 0.
- If RFBM[*i*] = 1, XSTATE\_BV[*i*] is set to the value of XINUSE[*i*] (see below for an exception made for XSTATE\_BV[1]). Section 13.6 defines XINUSE to describe the processor init optimization and specifies the initial configuration of each state component. The nature of that optimization implies the following:
  - If state component *i* is in its initial configuration, XSTATE\_BV[*i*] may be written with either 0 or 1.
  - If state component *i* is not in its initial configuration, XSTATE\_BV[*i*] is written with 1.

XINUSE[1] pertains only to the state of the XMM registers and not to MXCSR. However, if RFBM[1] = 1 and MXCSR does not have the value 1F80H, XSAVEC writes XSTATE\_BV[1] as 1 even if XINUSE[1] = 0.

(As explained in Section 13.6, the initial configurations of some state components may depend on whether the processor is in 64-bit mode.)

The XSAVEC instructions sets bit 63 of the XCOMP\_BV field of the XSAVE header while writing RFBM[62:0] to XCOMP\_BV[62:0]. The XSAVEC instruction does not write any part of the XSAVE header other than the XSTATE\_BV and XCOMP\_BV fields.

Execution of XSAVEC saves into the XSAVE area those state components corresponding to bits that are set in RFBM (subject to the init optimization described below). State components 0 and 1 are located in the legacy region of the XSAVE area (see Section 13.4.1). Each state component *i*,  $2 \leq i \leq 62$ , is located in the extended region; the XSAVEC instruction always uses the compacted format for the extended region (see Section 13.4.3).

See Section 13.5 for specifics for each state component and for details regarding mode-specific operation and operation determined by instruction prefixes. See Section 13.13 for details regarding faults caused by memory accesses.

Execution of XSAVEC performs the init optimization to reduce the amount of data written to memory. If XINUSE[*i*] = 0, state component *i* is not saved to the XSAVE area (even if RFBM[*i*] = 1). However, if RFBM[1] = 1 and MXCSR does not have the value 1F80H, XSAVEC writes saves all of state component 1 (SSE — including the

1. If CR0.AM = 1, CPL = 3, and EFLAGS.AC = 1, an alignment-check exception (#AC) may occur instead of #GP.

2. Unlike the XSAVE and XSAVEOPT instructions, the XSAVEC instruction does **not** read the XSTATE\_BV field of the XSAVE header.

XMM registers) even if `XINUSE[1] = 0`. Unlike the XSAVE instruction, `RFBM[2]` does not determine whether XSAVEC saves `MXCSR` and `MXCSR_MASK`.

## 13.11 OPERATION OF XSAVES

The operation of XSAVES is similar to that of XSAVEC. The main differences are (1) XSAVES can be executed only if `CPL = 0`; (2) XSAVES can operate on the state components whose bits are set in `XCR0 | IA32_XSS` and can thus operate on supervisor state components; and (3) XSAVES uses the modified optimization (see Section 13.6). See Section 13.2 for details of how to determine whether XSAVES is supported.

The XSAVES instruction takes a single memory operand, which is an XSAVE area. In addition, the register pair `EDX:EAX` is an implicit operand used as a state-component bitmap (see Section 13.1) called the **instruction mask**. `EDX:EAX & (XCR0 | IA32_XSS)` (the logical AND the instruction mask with the logical OR of `XCR0` and `IA32_XSS`) is the **requested-feature bitmap (RFBM)** of the state components to be saved.

The following conditions cause execution of the XSAVES instruction to generate a fault:

- If the XSAVE feature set is not enabled (`CR4.OSXSAVE = 0`), an invalid-opcode exception (`#UD`) occurs.
- If `CR0.TS[bit 3]` is 1, a device-not-available exception (`#NM`) occurs.
- If `CPL > 0` or if the address of the XSAVE area is not 64-byte aligned, a general-protection exception (`#GP`) occurs.<sup>1</sup>

If none of these conditions cause a fault, execution of XSAVES writes the `XSTATE_BV` field of the XSAVE header (see Section 13.4.2), setting `XSTATE_BV[i]` ( $0 \leq i \leq 63$ ) as follows:

- If `RFBM[i] = 0`, `XSTATE_BV[i]` is written as 0.
- If `RFBM[i] = 1`, `XSTATE_BV[i]` is set to the value of `XINUSE[i]` (see below for an exception made for `XSTATE_BV[1]`). Section 13.6 defines `XINUSE` to describe the processor init optimization and specifies the initial configuration of each state component. The nature of that optimization implies the following:
  - If state component  $i$  is in its initial configuration, `XSTATE_BV[i]` may be written with either 0 or 1.
  - If state component  $i$  is not in its initial configuration, `XSTATE_BV[i]` is written with 1.

`XINUSE[1]` pertains only to the state of the XMM registers and not to `MXCSR`. However, if `RFBM[1] = 1` and `MXCSR` does not have the value `1F80H`, XSAVES writes `XSTATE_BV[1]` as 1 even if `XINUSE[1] = 0`.

(As explained in Section 13.6, the initial configurations of some state components may depend on whether the processor is in 64-bit mode.)

The XSAVES instructions sets bit 63 of the `XCOMP_BV` field of the XSAVE header while writing `RFBM[62:0]` to `XCOMP_BV[62:0]`. The XSAVES instruction does not write any part of the XSAVE header other than the `XSTATE_BV` and `XCOMP_BV` fields.

Execution of XSAVES saves into the XSAVE area those state components corresponding to bits that are set in `RFBM` (subject to the optimizations described below). State components 0 and 1 are located in the legacy region of the XSAVE area (see Section 13.4.1). Each state component  $i$ ,  $2 \leq i \leq 62$ , is located in the extended region; the XSAVES instruction always uses the compacted format for the extended region (see Section 13.4.3).

See Section 13.5 for specifics for each state component and for details regarding mode-specific operation and operation determined by instruction prefixes; in particular, see Section 13.5.6 for some special treatment of PT state by XSAVES. See Section 13.13 for details regarding faults caused by memory accesses.

Execution of XSAVES performs the init optimization to reduce the amount of data written to memory. If `XINUSE[i] = 0`, state component  $i$  is not saved to the XSAVE area (even if `RFBM[i] = 1`). However, if `RFBM[1] = 1` and `MXCSR` does not have the value `1F80H`, XSAVES writes saves all of state component 1 (SSE — including the XMM registers) even if `XINUSE[1] = 0`.

Like XSAVEOPT, XSAVES may perform the modified optimization. Each execution of `XRSTOR` and `XRSTORS` establishes `XRSTOR_INFO` as a 4-tuple  $\langle w, x, y, z \rangle$  (see Section 13.8.3 and Section 13.12). Execution of XSAVES uses the modified optimization only if the following all hold:

1. If `CR0.AM = 1`, `CPL = 3`, and `EFLAGS.AC = 1`, an alignment-check exception (`#AC`) may occur instead of `#GP`.



- $w = \text{CPL}$ ;
- $x = 1$  if and only if the logical processor is in VMX non-root operation;
- $y$  is the linear address of the XSAVE area being used by XSAVEOPT; and
- $z[63]$  is 1 and  $z[62:0] = \text{RFBM}[62:0]$ . (This last item implies that XSAVES does not use the modified optimization if the last execution of XRSTOR used the standard form and followed the last execution of XRSTORS.)

If XSAVES uses the modified optimization and  $\text{XMODIFIED}[i] = 0$  (see Section 13.6), state component  $i$  is not saved to the XSAVE area.

## 13.12 OPERATION OF XRSTORS

The operation of XRSTORS is similar to that of XRSTOR. Three main differences are (1) XRSTORS can be executed only if  $\text{CPL} = 0$ ; (2) XRSTORS can operate on the state components whose bits are set in  $\text{XCR0} \mid \text{IA32\_XSS}$  and can thus operate on supervisor state components; and (3) XRSTORS has only a compacted form (no standard form; see Section 13.8). See Section 13.2 for details of how to determine whether XRSTORS is supported.

The XRSTORS instruction takes a single memory operand, which is an XSAVE area. In addition, the register pair  $\text{EDX:EAX}$  is an implicit operand used as a state-component bitmap (see Section 13.1) called the **instruction mask**.  $\text{EDX:EAX} \& (\text{XCR0} \mid \text{IA32\_XSS})$  (the logical AND the instruction mask with the logical OR of  $\text{XCR0}$  and  $\text{IA32\_XSS}$ ) is the **requested-feature bitmap (RFBM)** of the state components to be restored.

The following conditions cause execution of the XRSTOR instruction to generate a fault:

- If the XSAVE feature set is not enabled ( $\text{CR4.OSXSAVE} = 0$ ), an invalid-opcode exception ( $\#UD$ ) occurs.
- If  $\text{CR0.TS}[\text{bit } 3]$  is 1, a device-not-available exception ( $\#NM$ ) occurs.
- If  $\text{CPL} > 0$  or if the address of the XSAVE area is not 64-byte aligned, a general-protection exception ( $\#GP$ ) occurs.<sup>1</sup>

After checking for these faults, the XRSTORS instruction reads the first 64 bytes of the XSAVE header, including the  $\text{XSTATE\_BV}$  and  $\text{XCOMP\_BV}$  fields (see Section 13.4.2). A  $\#GP$  occurs if any of the following conditions hold for the values read:

- $\text{XCOMP\_BV}[63] = 0$ .
- $\text{XCOMP\_BV}$  sets a bit in the range 62:0 that is not set in  $\text{XCR0} \mid \text{IA32\_XSS}$ .
- $\text{XSTATE\_BV}$  sets a bit (including bit 63) that is not set in  $\text{XCOMP\_BV}$ .
- Bytes 63:16 of the XSAVE header are not all 0.

If none of these conditions cause a fault, the processor updates each state component  $i$  for which  $\text{RFBM}[i] = 1$ . XRSTORS updates state component  $i$  based on the value of bit  $i$  in the  $\text{XSTATE\_BV}$  field of the XSAVE header:

- If  $\text{XSTATE\_BV}[i] = 0$ , the state component is set to its initial configuration. Section 13.6 specifies the initial configuration of each state component. If  $\text{XSTATE\_BV}[1] = 0$ , XRSTORS initializes  $\text{MXCSR}$  to 1F80H.  
State component  $i$  is set to its initial configuration as indicated above if  $\text{RFBM}[i] = 1$  and  $\text{XSTATE\_BV}[i] = 0$  — **even if  $\text{XCOMP\_BV}[i] = 0$** . This is true for all values of  $i$ , including 0 (x87 state) and 1 (SSE state).
- If  $\text{XSTATE\_BV}[i] = 1$ , the state component is loaded with data from the XSAVE area.<sup>2</sup> See Section 13.5 for specifics for each state component and for details regarding mode-specific operation and operation determined by instruction prefixes; in particular, see Section 13.5.6 for some special treatment of PT state by XRSTORS. See Section 13.13 for details regarding faults caused by memory accesses.

If XRSTORS is restoring a supervisor state component, the instruction causes a general-protection exception ( $\#GP$ ) if it would load any element of that component with an unsupported value (e.g., by setting a reserved bit in an MSR) or if a bit is set in any reserved portion of the state component in the XSAVE area.

1. If  $\text{CR0.AM} = 1$ ,  $\text{CPL} = 3$ , and  $\text{EFLAGS.AC} = 1$ , an alignment-check exception ( $\#AC$ ) may occur instead of  $\#GP$ .

2. Earlier fault checking ensured that, if the instruction has reached this point in execution and  $\text{XSTATE\_BV}[i] = 1$ , then  $\text{XCOMP\_BV}[i]$  is also 1.

State components 0 and 1 are located in the legacy region of the XSAVE area (see Section 13.4.1). Each state component  $i$ ,  $2 \leq i \leq 62$ , is located in the extended region; XRSTORS uses the compacted format for the extended region (see Section 13.4.3).

The MXCSR register is part of SSE state (see Section 13.5.2) and is thus loaded from memory if  $RFBM[1] = XSTATE\_BV[i] = 1$ . XRSTORS causes a general-protection exception (#GP) if it would load MXCSR with an illegal value.

If an execution of XRSTORS causes an exception or a VM exit during or after restoring a supervisor state component, each element of that state component may have the value it held before the XRSTORS execution, the value loaded from the XSAVE area, or the element's initial value (as defined in Section 13.6). See Section 13.5.6 for some special treatment of PT state for the case in which XRSTORS causes an exception or a VM exit.

Like XRSTOR, execution of XRSTORS causes the processor to update its tracking for the init and modified optimizations (see Section 13.6 and Section 13.8.3). The following items provide details:

- The processor updates its tracking for the init optimization as follows:
  - If  $RFBM[i] = 0$ ,  $XINUSE[i]$  is not changed.
  - If  $RFBM[i] = 1$  and  $XSTATE\_BV[i] = 0$ , state component  $i$  may be tracked as init;  $XINUSE[i]$  may be set to 0 or 1.
  - If  $RFBM[i] = 1$  and  $XSTATE\_BV[i] = 1$ , state component  $i$  is tracked as not init;  $XINUSE[i]$  is set to 1.
- The processor updates its tracking for the modified optimization and records information about the XRSTORS execution for future interaction with the XSAVEOPT and XSAVES instructions as follows:
  - If  $RFBM[i] = 0$ , state component  $i$  is tracked as modified;  $XMODIFIED[i]$  is set to 1.
  - If  $RFBM[i] = 1$ , state component  $i$  may be tracked as unmodified;  $XMODIFIED[i]$  may be set to 0 or 1.
  - $XRSTOR\_INFO$  is set to the 4-tuple  $\langle w, x, y, z \rangle$ , where  $w$  is the CPL;  $x$  is 1 if the logical processor is in VMX non-root operation and 0 otherwise;  $y$  is the linear address of the XSAVE area; and  $z$  is  $XCOMP\_BV$  (this implies that  $z[63] = 1$ ).

## 13.13 MEMORY ACCESSSES BY THE XSAVE FEATURE SET

Each instruction in the XSAVE feature set operates on a set of XSAVE-managed state components. The specific set of components on which an instruction operates is determined by the values of XCR0, the IA32\_XSS MSR, EDX:EAX, and (for XRSTOR and XRSTORS) the XSAVE header.

Section 13.4 provides the details necessary to determine the location of each state component for any execution of an instruction in the XSAVE feature set. An execution of an instruction in the XSAVE feature set may access any byte of any state component on which that execution operates.

Section 13.5 provides details of the different XSAVE-managed state components. Some portions of some of these components are accessible only in 64-bit mode. Executions of XRSTOR and XRSTORS outside 64-bit mode will not update those portions; executions of XSAVE, XSAVEC, XSAVEOPT, and XSAVES will not modify the corresponding locations in memory.

Despite this fact, any execution of these instructions outside 64-bit mode may access any byte in any state component on which that execution operates — even those at addresses corresponding to registers that are accessible only in 64-bit mode. As result, such an execution may incur a fault due to an attempt to access such an address.

For example, an execution of XSAVE outside 64-bit mode may incur a page fault if paging does not map as read/write the section of the XSAVE area containing state component 7 (Hi16\_ZMM state) — despite the fact that state component 7 can be accessed only in 64-bit mode.



#### 4. Updates to Chapter 15, Volume 1

Change bars show changes to Chapter 15 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture*.

-----  
This chapter is new and covers programming with Intel® AVX-512.

### 15.1 OVERVIEW

The Intel AVX-512 family comprises of a collection of instruction set extensions, including AVX-512 Foundation, AVX-512 Exponential and Reciprocal instructions, AVX-512 Conflict, AVX-512 Prefetch, and additional 512-bit SIMD instruction extensions. Intel AVX-512 instructions are natural extensions to Intel AVX and Intel AVX2. Intel AVX-512 introduces the following architectural enhancements:

- Support for 512-bit wide vectors and SIMD register set. 512-bit register state is managed by the operating system using XSAVE/XRSTOR instructions introduced in 45 nm Intel 64 processors (see *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B*, and *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*).
- Support for 16 new, 512-bit SIMD registers (for a total of 32 SIMD registers, ZMM0 through ZMM31) in 64-bit mode. The extra 16 registers state is managed by the operating system using XSAVE/XRSTOR/XSAVEOPT.
- Support for 8 new opmask registers (k0 through k7) used for conditional execution and efficient merging of destination operands. The opmask register state is managed by the operating system using the XSAVE/XRSTOR/XSAVEOPT instructions.
- A new encoding prefix (referred to as EVEX) to support additional vector length encoding up to 512 bits. The EVEX prefix builds upon the foundations of the VEX prefix to provide compact, efficient encoding for functionality available to VEX encoding plus the following enhanced vector capabilities:
  - Opmasks.
  - Embedded broadcast.
  - Instruction prefix-embedded rounding control.
  - Compressed address displacements.

#### 15.1.1 512-Bit Wide SIMD Register Support

Intel AVX-512 instructions support 512-bit wide SIMD registers (ZMM0-ZMM31). The lower 256-bits of the ZMM registers are aliased to the respective 256-bit YMM registers and the lower 128-bit are aliased to the respective 128-bit XMM registers.

#### 15.1.2 32 SIMD Register Support

Intel AVX-512 instructions also support 32 SIMD registers in 64-bit mode (XMM0-XMM31, YMM0-YMM31 and ZMM0-ZMM31). The number of available vector registers in 32-bit mode is still 8.

#### 15.1.3 Eight Opmask Register Support

Intel AVX-512 instructions support 8 opmask registers (k0-k7). The width of each opmask register is architecturally defined as size MAX\_KL (64 bits). Seven of the eight opmask registers (k1-k7) can be used in conjunction with EVEX-encoded AVX-512 Foundation instructions to provide conditional execution and efficient merging of data elements in the destination operand. The encoding of opmask register k0 is typically used when all data elements (unconditional processing) are desired. Additionally, the opmask registers are also used as vector flags/element-level vector sources to introduce novel SIMD functionality as seen in new instructions such as VCOMPRESSPS.

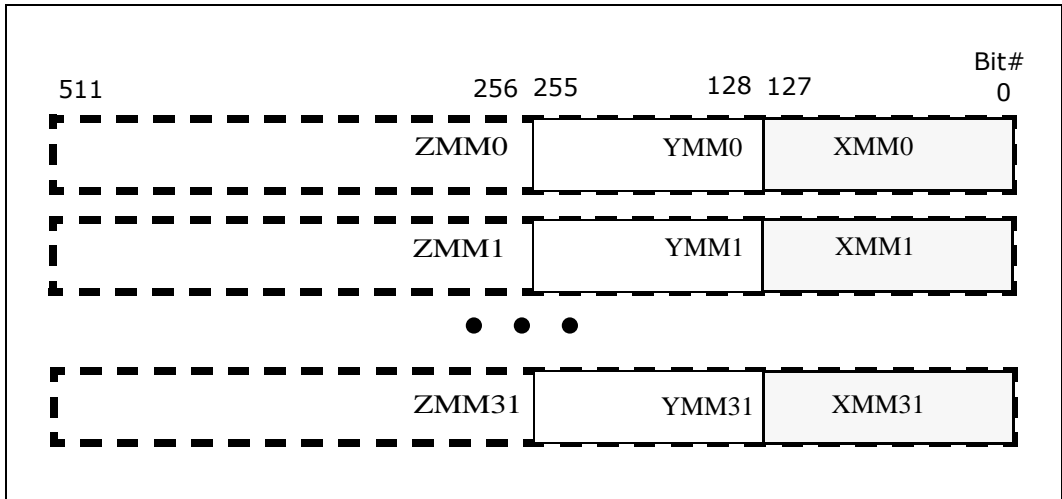


Figure 15-1. 512-Bit Wide Vectors and SIMD Register Set

### 15.1.4 Instruction Syntax Enhancement

The architecture of EVEX encoding enhances the vector instruction encoding scheme in the following way:

- 512-bit vector-length, up to 32 ZMM registers, and enhanced vector programming environment are supported using the enhanced VEX (EVEX).

The EVEX prefix provides more encodable bit fields than the VEX prefix. In addition to encoding 32 ZMM registers in 64-bit mode, instruction encoding using the EVEX prefix can directly encode 7 (out of 8) opmask register operands to provide conditional processing in vector instruction programming. The enhanced vector programming environment can be explicitly expressed in the instruction syntax to include the following elements:

- An opmask operand: the opmask registers are expressed using the notation "k1" through "k7". An EVEX-encoded instruction supporting conditional vector operation using the opmask register k1 is expressed by attaching the notation {k1} next to the destination operand. The use of this feature is optional for most instructions. There are two types of masking (merging and zeroing) differentiated using the EVEX.z bit ({z} in instruction signature).
- Embedded broadcast may be supported for some instructions on the source operand that can be encoded as a memory vector. Data elements of a memory vector may be conditionally fetched or written to.
- For instruction syntax that operates only on floating-point data in SIMD registers with rounding semantics, the EVEX encoding can provide explicit rounding control within the EVEX bit fields at either scalar or 512-bit vector length.

In AVX-512 instructions, vector addition of all elements of the source operands can be expressed in the same syntax as AVX instruction:

```
VADDPS zmm1, zmm2, zmm3
```

Additionally, the EVEX encoding scheme of AVX-512 Foundation can express conditional vector addition as:

```
VADDPS zmm1 {k1}{z}, zmm2, zmm3
```

where:

- Conditional processing and updates to destination are expressed with an opmask register.
- Zeroing behavior of the opmask selected destination element is expressed by the {z} modifier (with merging as the default if no modifier is specified).

Note that some SIMD instructions supporting three-operand syntax but processing only less than or equal to 128-bits of data are considered part of the 512-bit SIMD instruction set extensions, because bits MAX\_VL-1:128 of the destination register are zeroed by the processor. The same rule applies to instructions operating on 256-bits of data where bits MAX\_VL-1:256 of the destination register are zeroed.

### 15.1.5 EVEX Instruction Encoding Support

Intel AVX-512 instructions employ a new encoding prefix, referred to as EVEX, in the Intel 64 and IA-32 instruction encoding format. Instruction encoding using the EVEX prefix provides the following capabilities:

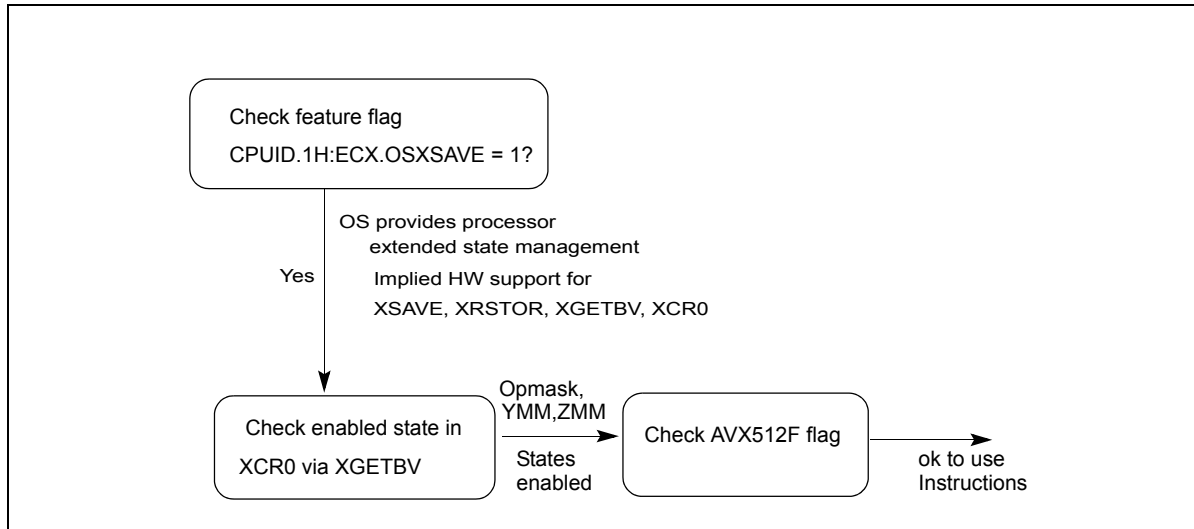
- Direct encoding of a SIMD register operand within EVEX (similar to VEX). This provides instruction syntax support for three source operands.
- Compaction of REX prefix functionality and extended SIMD register encoding: the equivalent REX-prefix compaction functionality offered by the VEX prefix is provided within EVEX. Furthermore, EVEX extends the operand encoding capability to allow direct addressing of up to 32 ZMM registers in 64-bit mode.
- Compaction of SIMD prefix functionality and escape byte encoding: the functionality of a SIMD prefix (66H, F2H, F3H) on opcode is equivalent to an opcode extension field to introduce new processing primitives. This functionality is provided in the VEX prefix encoding scheme and employed within the EVEX prefix. Similarly, the functionality of the escape opcode byte (0FH) and two-byte escape (0F38H, 0F3AH) are also compacted within the EVEX prefix encoding.
- Most EVEX-encoded SIMD numeric and data processing instruction semantics with memory operands have more relaxed memory alignment requirements than instructions encoded using SIMD prefixes (see Section 2.6, “Memory Alignment” in the *Intel® Architecture Instruction Set Extensions Programming Reference*).
- Direct encoding of an opmask operand within the EVEX prefix. This provides instruction syntax support for conditional vector-element operation and merging of destination operand using an opmask register (k1-k7).
- Direct encoding of a broadcast attribute for instructions with a memory operand source. This provides instruction syntax support for elements broadcasting the second operand before being used in the actual operation.
- Compressed memory address displacements for a more compact instruction encoding byte sequence.

EVEX encoding applies to SIMD instructions operating on XMM, YMM and ZMM registers. EVEX is not supported for instructions operating on MMX or x87 registers. Details of EVEX instruction encoding are discussed in Section 2.6, “Intel® AVX-512 Encoding” of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A*.

## 15.2 DETECTION OF AVX-512 FOUNDATION INSTRUCTIONS

The majority of AVX-512 Foundation instructions are encoded using the EVEX encoding scheme. EVEX-encoded instructions can operate on the 512-bit ZMM register state plus 8 opmask registers. The opmask instructions in AVX-512 Foundation instructions operate only on opmask registers or with a general purpose register. System software requirements to support the ZMM state and opmask instructions are described in Chapter 3, “System Programming For Intel® AVX-512” of the *Intel® Architecture Instruction Set Extensions Programming Reference*.

Processor support of AVX-512 Foundation instructions is indicated by CPUID.(EAX=07H, ECX=0):EBX.AVX512F[bit 16] = 1. Detection of AVX-512 Foundation instructions operating on ZMM states and opmask registers needs to follow the general procedural flow in Figure 15-2.



**Figure 15-2. Procedural Flow for Application Detection of AVX-512 Foundation Instructions**

Prior to using AVX-512 Foundation instructions, the application must identify that the operating system supports the XGETBV instruction and the ZMM register state, in addition to confirming the processor’s support for ZMM state management using XSAVE/XRSTOR and AVX-512 Foundation instructions. The following simplified sequence accomplishes both and is strongly recommended.

1. Detect CPUID.1:ECX.OSXSAVE[bit 27] = 1 (XGETBV enabled for application use<sup>1</sup>).
2. Execute XGETBV and verify that XCR0[7:5] = ‘111b’ (OPMASK state, upper 256-bit of ZMM0-ZMM15 and ZMM16-ZMM31 state are enabled by OS) and that XCR0[2:1] = ‘11b’ (XMM state and YMM state are enabled by OS).
3. Detect CPUID.0x7.0:EBX.AVX512F[bit 16] = 1.

### 15.2.1 Additional 512-bit Instruction Extensions of the Intel AVX-512 Family

Processor support of the Intel AVX-512 Exponential and Reciprocal instructions are indicated by querying the feature flag:

- If CPUID.(EAX=07H, ECX=0):EBX.AVX512ER[bit 27] = 1, the collection of VEXP2PD/VEXP2PS/VRCP28xx/VRSQRT28xx instructions are supported.

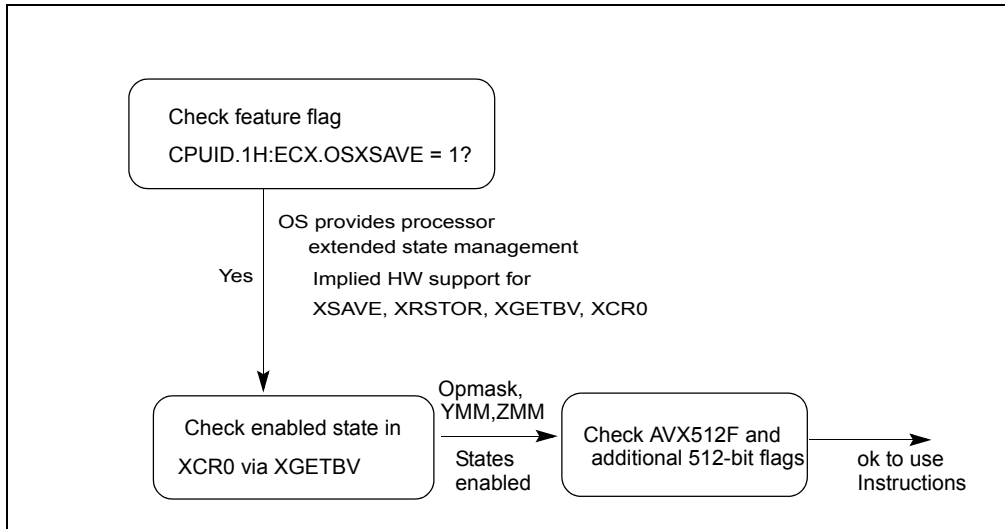
Processor support of the Intel AVX-512 Prefetch instructions are indicated by querying the feature flag:

- If CPUID.(EAX=07H, ECX=0):EBX.AVX512PF[bit 26] = 1, a collection of VGATHERPF0xxx/VGATHERPF1xxx/VSCATTERPF0xxx/VSCATTERPF1xxx instructions are supported.

Detection of 512-bit instructions operating on ZMM states and opmask registers, outside of AVX-512 Foundation, needs to follow the general procedural flow in Figure 15-3.

1. If CPUID.01H:ECX.OSXSAVE reports 1, it also indirectly implies the processor supports XSAVE, XRSTOR, XGETBV, processor extended state bit vector XCR0 register. Thus an application may streamline the checking of CPUID feature flags for XSAVE and OSXSAVE. XSETBV is a privileged instruction.





**Figure 15-3. Procedural Flow for Application Detection of 512-bit Instructions**

PREFETCH1W does not require OS support for XMM/YMM/ZMM/k-reg, SIMD FP exception support.

Procedural Flow of Application Detection of other 512-bit extensions:

Prior to using the Intel AVX-512 Exponential and Reciprocal instructions, the application must identify that the operating system supports the XGETBV instruction and the ZMM register state, in addition to confirming the processor's support for ZMM state management using XSAVE/XRSTOR and AVX-512 Foundation instructions. The following simplified sequence accomplishes both and is strongly recommended.

1. Detect  $\text{CPUID.1:ECX.OSXSAVE}[\text{bit } 27] = 1$  (XGETBV enabled for application use).
2. Execute XGETBV and verify that  $\text{XCR0}[7:5] = '111b'$  (OPMASK state, upper 256-bit of ZMM0-ZMM15 and ZMM16-ZMM31 state are enabled by OS) and that  $\text{XCR0}[2:1] = '11b'$  (XMM state and YMM state are enabled by OS).
3. Verify both  $\text{CPUID.0x7.0:EBX.AVX512F}[\text{bit } 16] = 1$ , and  $\text{CPUID.0x7.0:EBX.AVX512ER}[\text{bit } 27] = 1$ .

Prior to using the Intel AVX-512 Prefetch instructions, the application must identify that the operating system supports the XGETBV instruction and the ZMM register state, in addition to confirming the processor's support for ZMM state management using XSAVE/XRSTOR and AVX-512 Foundation instructions. The following simplified sequence accomplishes both and is strongly recommended.

1. Detect  $\text{CPUID.1:ECX.OSXSAVE}[\text{bit } 27] = 1$  (XGETBV enabled for application use).
2. Execute XGETBV and verify that  $\text{XCR0}[7:5] = '111b'$  (OPMASK state, upper 256-bit of ZMM0-ZMM15 and ZMM16-ZMM31 state are enabled by OS) and that  $\text{XCR0}[2:1] = '11b'$  (XMM state and YMM state are enabled by OS).
3. Verify both  $\text{CPUID.0x7.0:EBX.AVX512F}[\text{bit } 16] = 1$ , and  $\text{CPUID.0x7.0:EBX.AVX512PF}[\text{bit } 26] = 1$ .

## 15.3 DETECTION OF 512-BIT INSTRUCTION GROUPS OF INTEL® AVX-512 FAMILY

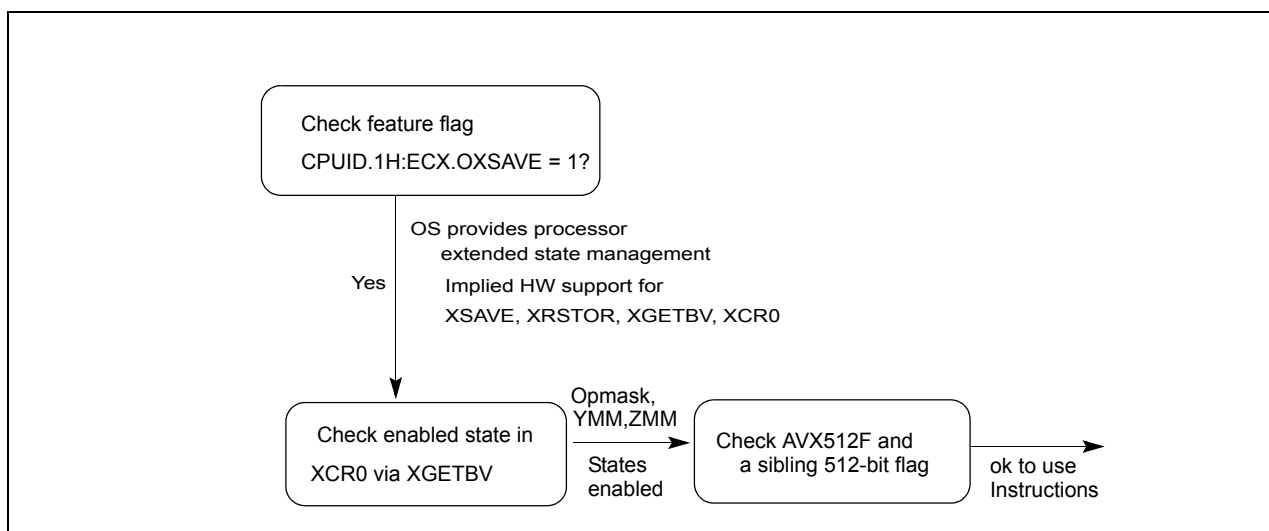
In addition to the Intel AVX-512 Foundation instructions, Intel AVX-512 family provides several groups of instruction extensions that can operate in vector lengths of 512/256/128 bits. Each group is enumerated by a CPUID leaf 7 feature flag and can be encoded via the EVEX.L'L field to support operation at vector lengths smaller than 512 bits. These instruction groups are listed in Table 15-1.

**Table 15-1. 512-bit Instruction Groups in the Intel AVX-512 Family**

CPUID Leaf 7 Feature Flag Bit	Feature Flag abbreviation of 512-bit Instruction Group	SW Detection Flow
CPUID.(EAX=07H, ECX=0):EBX[bit 16]	AVX512F (AVX-512 Foundation)	Figure 15-2
CPUID.(EAX=07H, ECX=0):EBX[bit 28]	AVX512CD	Figure 15-4
CPUID.(EAX=07H, ECX=0):EBX[bit 17]	AVX512DQ	Figure 15-4
CPUID.(EAX=07H, ECX=0):EBX[bit 30]	AVX512BW	Figure 15-4

Software must follow the detection procedure for the 512-bit AVX-512 Foundation instructions as described in Section 15.2.

Detection of other 512-bit sibling instruction groups listed in Table 15-1 (excluding AVX512F) follows the procedure described in Figure 15-4:



**Figure 15-4. Procedural Flow for Application Detection of 512-bit Instruction Groups**

To detect 512-bit instructions enumerated by AVX512CD, the following sequence is strongly recommended.

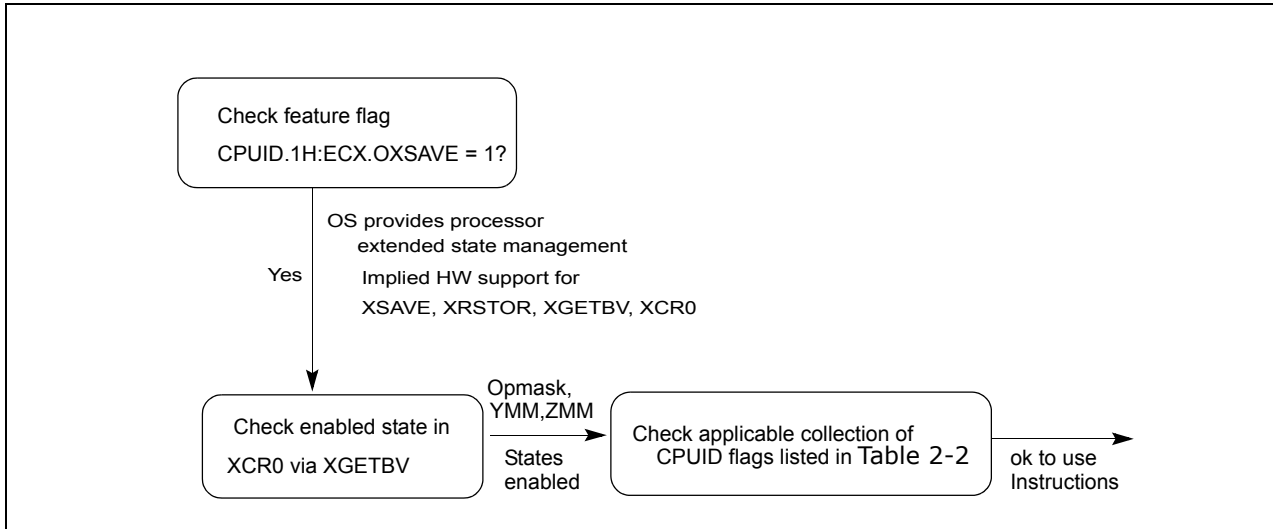
1. Detect CPUID.1:ECX.OSXSAVE[bit 27] = 1 (XGETBV enabled for application use).
2. Execute XGETBV and verify that XCR0[7:5] = '111b' (OPMASK state, upper 256-bit of ZMM0-ZMM15 and ZMM16-ZMM31 state are enabled by OS) and that XCR0[2:1] = '11b' (XMM state and YMM state are enabled by OS).
3. Verify both CPUID.0x7.0:EBX.AVX512F[bit 16] = 1, CPUID.0x7.0:EBX.AVX512CD[bit 28] = 1.

Similarly, the detection procedure for enumerating 512-bit instructions reported by AVX512DW follows the same flow.

## 15.4 DETECTION OF INTEL AVX-512 INSTRUCTION GROUPS OPERATING AT 256 AND 128-BIT VECTOR LENGTHS

For each of the 512-bit instruction groups in the Intel AVX-512 family listed in Table 15-1, the EVEX encoding scheme may support a vast majority of these instructions operating at 256-bit or 128-bit (if applicable) vector lengths. Encoding support for vector lengths smaller than 512-bits is indicated by CPUID.(EAX=07H, ECX=0):EBX[bit 31], abbreviated as AVX512VL.

The AVX512VL flag alone is never sufficient to determine a given Intel AVX-512 instruction may be encoded at vector lengths smaller than 512 bits. Software must use the procedure described in Figure 15-5 and Table 15-2.



**Figure 15-5. Procedural Flow for Detection of Intel AVX-512 Instructions Operating at Vector Lengths < 512**

To illustrate the procedure described in Figure 15-5 and Table 15-2 for software to use EVEX.256 encoded VPCONFLICT, the following sequence is provided. It is strongly recommended this sequence is followed.

- 1) Detect CPUID.1:ECX.OSXSAVE[bit 27] = 1 (XGETBV enabled for application use).
- 2) Execute XGETBV and verify that XCR0[7:5] = '111b' (OPMASK state, upper 256-bit of ZMM0-ZMM15 and ZMM16-ZMM31 state are enabled by OS) and that XCR0[2:1] = '11b' (XMM state and YMM state are enabled by OS).
- 3) Verify CPUID.0x7.0:EBX.AVX512F[bit 16] = 1, CPUID.0x7.0:EBX.AVX512CD[bit 28] = 1, and CPUID.0x7.0:EBX.AVX512VL[bit 31] = 1.

**Table 15-2. Feature flag Collection Required of 256/128 Bit Vector Lengths for Each Instruction Group**

Usage of 256/128 Vector Lengths	Feature Flag Collection to Verify
AVX512F	AVX512F & AVX512VL
AVX512CD	AVX512F & AVX512CD & AVX512VL
AVX512DQ	AVX512F & AVX512DQ & AVX512VL
AVX512BW	AVX512F & AVX512BW & AVX512VL

In some specific cases, AVX512VL may only support EVEX.256 encoding but not EVEX.128. These cases are listed in Table 15-3.

**Table 15-3. Instruction Mnemonics That Do Not Support EVEX.128 Encoding**

Instruction Group	Instruction Mnemonics Supporting EVEX.256 Only Using AVX512VL
AVX512F	VBROADCASTSD, VBROADCASTF32X4, VEXTRACTI32X4, VINSERTF32X4, VINSERTI32X4, VPERMD, VPERMPD, VPERMPS, VPERMQ, VSHUFF32X4, VSHUFF64X2, VSHUFI32X4, VSHUFI64X2
AVX512CD	
AVX512DQ	VBROADCASTF32X2, VBROADCASTF64X2, VBROADCASTI32X4, VBROADCASTI64X2, VEXTRACTI64X2, VINSERTF64X2, VINSERTI64X2,
AVX512BW	

## 15.5 ACCESSING XMM, YMM AND ZMM REGISTERS

The lower 128 bits of a YMM register is aliased to the corresponding XMM register. Legacy SSE instructions (i.e., SIMD instructions operating on XMM state but not using the VEX prefix, also referred to non-VEX encoded SIMD instructions) will not access the upper bits (MAX\_VL-1:128) of the YMM registers. AVX and FMA instructions with a VEX prefix and vector length of 128-bits zeroes the upper 128 bits of the YMM register.

Upper bits of YMM registers (255:128) can be read and written to by many instructions with a VEX.256 prefix. XSAVE and XRSTOR may be used to save and restore the upper bits of the YMM registers.

The lower 256 bits of a ZMM register are aliased to the corresponding YMM register. Legacy SSE instructions (i.e., SIMD instructions operating on XMM state but not using the VEX prefix, also referred to non-VEX encoded SIMD instructions) will not access the upper bits (MAX\_VL-1:128) of the ZMM registers, where MAX\_VL is maximum vector length (currently 512 bits). AVX and FMA instructions with a VEX prefix and vector length of 128-bits zero the upper 384 bits of the ZMM register, while the VEX prefix and vector length of 256-bits zeroes the upper 256 bits of the ZMM register.

Upper bits of ZMM registers (511:256) can be read and written to by instructions with an EVEX.512 prefix.

## 15.6 ENHANCED VECTOR PROGRAMMING ENVIRONMENT USING EVEX ENCODING

EVEX-encoded AVX-512 instructions support an enhanced vector programming environment. The enhanced vector programming environment uses the combination of EVEX bit-field encodings and a set of eight opmask registers to provide the following capabilities:

- Conditional vector processing of an EVEX-encoded instruction. Opmask registers k1 through k7 can be used to conditionally govern the per-data-element computational operation and the per-element updates to the destination operand of an AVX-512 Foundation instruction. Each bit of the opmask register governs one vector element operation (a vector element can be 8 bits, 16 bits, 32 bits or 64 bits).
- In addition to providing predication control on vector instructions via EVEX bit-field encoding, the opmask registers can also be used similarly on general-purpose registers as source/destination operands using modR/M encoding for non-mask-related instructions. In this case, an opmask register k0 through k7 can be selected.
- In 64-bit mode, 32 vector registers can be encoded using the EVEX prefix.
- Broadcast may be supported for some instructions on the operand that can be encoded as a memory vector. The data elements of a memory vector may be conditionally fetched or written to, and the vector size is dependent on the data transformation function.
- Flexible rounding control for the register-to-register flavor of EVEX encoded 512-bit and scalar instructions. Four rounding modes are supported by direct encoding within the EVEX prefix, overriding MXCSR settings.
- Broadcast of one element to the rest of the destination vector register.
- Compressed 8-bit displacement encoding scheme to increase the instruction encoding density for instructions that normally require disp32 syntax.

## 15.6.1 OPMASK Register to Predicate Vector Data Processing

AVX-512 instructions using EVEX encode a predicate operand to conditionally control per-element computational operation and updating of the result to the destination operand. The predicate operand is known as the opmask register. The opmask is a set of eight architectural registers of size MAX\_KL (64-bit). Note that from this set of eight architectural registers, only k1 through k7 can be addressed as a predicate operand. k0 can be used as a regular source or destination but cannot be encoded as a predicate operand. Note also that a predicate operand can be used to enable memory fault-suppression for some instructions with a memory operand (source or destination).

As a predicate operand, the opmask registers contain one bit to govern the operation/update to each data element of a vector register. In general, opmask registers can support instructions with all element sizes: byte (int8), word (int16), single-precision floating-point (float32), integer doubleword(int32), double-precision floating-point (float64), integer quadword (int64). Therefore, a ZMM vector register can hold 8, 16, 32, or 64 elements in principle. The length of an opmask register, MAX\_KL, is sufficient to handle up to 64 elements with one bit per element, i.e., 64 bits. Masking is supported in most of the AVX-512 instructions. For a given vector length, each instruction accesses only the number of least significant mask bits that are needed based on its data type. For example, AVX-512 Foundation instructions operating on 64-bit data elements with a 512-bit vector length, only use the 8 least significant bits of the opmask register.

An opmask register affects an AVX-512 instruction at per-element granularity. Any numeric or non-numeric operation of each data element and per-element updates of intermediate results to the destination operand are predicated on the corresponding bit of the opmask register.

An opmask serving as a predicate operand in AVX-512 obeys the following properties:

- The instruction's operation is not performed for an element if the corresponding opmask bit is not set. This implies that no exception or violation can be caused by an operation on a masked-off element. Consequently, no MXCSR exception flag is updated as a result of a masked-off operation.
- A destination element is not updated with the result of the operation if the corresponding writemask bit is not set. Instead, the destination element value must be preserved (merging-masking) or it must be zeroed out (zeroing-masking).
- For some instructions with a memory operand, memory faults are suppressed for elements with a mask bit of 0.

Note that this feature provides a versatile construct to implement control-flow predication as the mask in effect provides a merging behavior for AVX-512 vector register destinations. As an alternative the masking can be used for zeroing instead of merging, so that the masked out elements are updated with 0 instead of preserving the old value. The zeroing behavior is provided to remove the implicit dependency on the old value when it is not needed.

Most instructions with masking enabled accept both forms of masking. Instructions that must have EVEX.aaa bits different than 0 (gather and scatter) and instructions that write to memory only accept merging-masking.

It's important to note that the per-element destination update rule also applies when the destination operand is a memory location. Vectors are written on a per element basis, based on the opmask register used as a predicate operand.

The value of an opmask register can be:

- Generated as a result of a vector instruction (e.g., CMP, FPCLASS, etc.).
- Loaded from memory.
- Loaded from a GPR register.
- Modified by mask-to-mask operations.

Opmask registers can be used for purposes outside of predication. For example, they can be used to manipulate sparse sets of elements from a vector, or used to set the EFLAGS based on the 0/0xFFFFFFFFFFFFFFFF/other status of the OR of two opmask registers.

### 15.6.1.1 Opmask Register K0

The only exception to the opmask rules described above is that opmask k0 can not be used as a predicate operand. Opmask k0 cannot be encoded as a predicate operand for a vector operation; the encoding value that would select opmask k0 will instead select an implicit opmask value of 0xFFFFFFFFFFFFFFFF, thereby effectively disabling

masking. Opmask register k0 can still be used for any instruction that takes opmask register(s) as operand(s) (either source or destination).

Note that certain instructions implicitly use the opmask as an extra destination operand. In such cases, trying to use the “no mask” feature will translate into a #UD fault being raised.

### 15.6.1.2 Example of Opmask Usages

The example below illustrates the predicated vector add operation and predicated updates of added results into the destination operand. The initial state of vector registers zmm0, zmm1, and zmm2 and k3 are:

```

MSB.....LSB

zmm0 =
[ 0x00000003 0x00000002 0x00000001 0x00000000 ] (bytes 15 through 0)
[ 0x00000007 0x00000006 0x00000005 0x00000004 ] (bytes 31 through 16)
[ 0x0000000B 0x0000000A 0x00000009 0x00000008 ] (bytes 47 through 32)
[ 0x0000000F 0x0000000E 0x0000000D 0x0000000C ] (bytes 63 through 48)

zmm1 =
[ 0x0000000F 0x0000000F 0x0000000F 0x0000000F ] (bytes 15 through 0)
[ 0x0000000F 0x0000000F 0x0000000F 0x0000000F ] (bytes 31 through 16)
[ 0x0000000F 0x0000000F 0x0000000F 0x0000000F ] (bytes 47 through 32)
[ 0x0000000F 0x0000000F 0x0000000F 0x0000000F ] (bytes 63 through 48)

zmm2 =
[ 0xAAAAAAAA 0xAAAAAAAA 0xAAAAAAAA 0xAAAAAAAA ] (bytes 15 through 0)
[ 0xBBBBBBBB 0xBBBBBBBB 0xBBBBBBBB 0xBBBBBBBB ] (bytes 31 through 16)
[ 0xCCCCCCCC 0xCCCCCCCC 0xCCCCCCCC 0xCCCCCCCC ] (bytes 47 through 32)
[ 0xDDDDDDDD 0xDDDDDDDD 0xDDDDDDDD 0xDDDDDDDD ] (bytes 63 through 48)

k3 = 0x8F03 (1000 1111 0000 0011)
    
```

An opmask register serving as a predicate operand is expressed as a curly-braces-enclosed decorator following the first operand in the Intel assembly syntax. Given this state, we will execute the following instruction:

```
vpaddq zmm2 {k3}, zmm0, zmm1
```

The vpaddq instruction performs 32-bit integer additions on each data element conditionally based on the corresponding bit value in the predicate operand k3. Since per-element operations are not operated if the corresponding bit of the predicate mask is not set, the intermediate result is:

```

[ ***** ***** 0x00000010 0x0000000F ] (bytes 15 through 0)
[ ***** ***** ***** ***** ] (bytes 31 through 16)
[ 0x0000001A 0x00000019 0x00000018 0x00000017 ] (bytes 47 through 32)
[ 0x0000001E ***** ***** ***** ] (bytes 63 through 48)
    
```

where “\*\*\*\*\*” indicates that no operation is performed.

This intermediate result is then written into the destination vector register, zmm2, using the opmask register k3 as the writemask, producing the following final result:

```

zmm2 =
[ 0xAAAAAAAA 0xAAAAAAAA 0x00000010 0x0000000F ] (bytes 15 through 0)
[ 0xBBBBBBBBB 0xBBBBBBBBB 0xBBBBBBBBB 0xBBBBBBBBB ] (bytes 31 through 16)
[ 0x0000001A 0x00000019 0x00000018 0x00000017 ] (bytes 47 through 32)
[ 0x0000001E 0xDDDDDDDD 0xDDDDDDDD 0xDDDDDDDD ] (bytes 63 through 48)

```

Note that for a 64-bit instruction (for example, `vaddpd`), only the 8 LSB of mask `k3` (`0x03`) would be used to identify the predicate operation on each one of the 8 elements of the source/destination vectors.

## 15.6.2 OpMask Instructions

AVX-512 Foundation instructions provide a collection of opmask instructions that allow programmers to set, copy, or operate on the contents of a given opmask register. There are three types of opmask instructions:

- **Mask read/write instructions:** These instructions move data between a general-purpose integer register or memory and an opmask mask register, or between two opmask registers. For example:
  - `kmovw k1, ebx`; move lower 16 bits of `ebx` to `k1`.
- **Flag instructions:** This category consists of instructions that modify EFLAGS based on the content of opmask registers.
  - `kortestw k1, k2`; OR registers `k1` and `k2` and updated EFLAGS accordingly.
- **Mask logical instructions:** These instructions perform standard bitwise logical operations between opmask registers.
  - `kandw k1, k2, k3`; AND lowest 16 bits of registers `k2` and `k3`, leaving the result in `k1`.

## 15.6.3 Broadcast

EVEX encoding provides a bit-field to encode data broadcast for some load-op instructions, i.e., instructions that load data from memory and perform some computational or data movement operation. A source element from memory can be broadcasted (repeated) across all the elements of the effective source operand (up to 16 times for a 32-bit data element, up to 8 times for a 64-bit data element). This is useful when we want to reuse the same scalar operand for all the operations in a vector instruction. Broadcast is only enabled on instructions with an element size of 32 bits or 64 bits. Byte and word instructions do not support embedded broadcast. The functionality of data broadcast is expressed as a curly-braces-enclosed decorator following the last register/memory operand in the Intel assembly syntax.

For instance:

```
vmulps zmm1, zmm2, [rax] {1to16}
```

The `{1to16}` primitive loads one float32 (single precision) element from memory, replicates it 16 times to form a vector of 16 32-bit floating-point elements, multiplies the 16 float32 elements with the corresponding elements in the first source operand vector, and puts each of the 16 results into the destination operand.

AVX-512 instructions with store semantics and pure load instructions do not support broadcast primitives.

```
vmovaps [rax] {k3}, zmm19
```

In contrast, the `k3` opmask register is used as the predicate operand in the above example. Only the store operation on data elements corresponding to the non-zero bits in `k3` will be performed.

### 15.6.4 STATIC ROUNDING MODE AND SUPPRESS ALL EXCEPTIONS

In previous SIMD instruction extensions (up to AVX and AVX2), rounding control is generally specified in MXCSR, with a handful of instructions providing per-instruction rounding override via encoding fields within the imm8 operand. AVX-512 offers a more flexible encoding attribute to override MXCSR-based rounding control for floating-pointing instructions with rounding semantics. This rounding attribute embedded in the EVEX prefix is called Static (per instruction) Rounding Mode or Rounding Mode override. This attribute allows programmers to statically apply a specific arithmetic rounding mode irrespective of the value of RM bits in MXCSR. It is available only to register-to-register flavors of EVEX-encoded floating-point instructions with rounding semantic. The differences between these three rounding control interfaces are summarized in Table 15-4.

**Table 15-4. Characteristics of Three Rounding Control Interfaces**

Rounding Interface	Static Rounding Override	Imm8 Embedded Rounding Override	MXCSR Rounding Control
Semantic Requirement	FP rounding	FP rounding	FP rounding
Prefix Requirement	EVEX.B = 1	NA	NA
Rounding Control	EVEX.L'L	IMM8[1:0] or MXCSR.RC (depending on IMM8[2])	MXCSR.RC
Suppress All Exceptions (SAE)	Implied	no	no
SIMD FP Exception #XF	All suppressed	Can raise #I, #P (unless SPE is set)	MXCSR masking controls
MXCSR flag update	No	yes (except PE if SPE is set)	Yes
Precedence	Above MXCSR.RC	Above EVEX.L'L	Default
Scope	512-bit, reg-reg, Scalar reg-reg	ROUNDPx, ROUNDSx, VCVTPS2PH, VRNDSCALExx	All SIMD operands, vector lengths

The static rounding-mode override in AVX-512 also implies the “suppress-all-exceptions” (SAE) attribute. The SAE effect is as if all the MXCSR mask bits are set, and none of the MXCSR flags will be updated. Using static rounding-mode via EVEX without SAE is not supported.

Static Rounding Mode and SAE control can be enabled in the encoding of the instruction by setting the EVEX.b bit to 1 in a register-register vector instruction. In such a case, vector length is assumed to be MAX\_VL (512-bit in case of AVX-512 packed vector instructions) or 128-bit for scalar instructions. Table 15-5 summarizes the possible static rounding-mode assignments in AVX-512 instructions.

Note that some instructions already allow specifying the rounding mode statically via immediate bits. In such cases, the immediate bits take precedence over the embedded rounding mode (in the same vein that they take precedence over whatever MXCSR.RM says).

**Table 15-5. Static Rounding Mode**

Function	Description
{rn-sae}	Round to nearest (even) + SAE
{rd-sae}	Round down (toward -inf) + SAE
{ru-sae}	Round up (toward +inf) + SAE
{rz-sae}	Round toward zero (Truncate) + SAE

An example of use would be as follows:

```
vaddps zmm7 {k6}, zmm2, zmm4, {rd-sae}
```

This would perform the single-precision floating-point addition of vectors zmm2 and zmm4 with round-towards-minus-infinity, leaving the result in vector zmm7 using k6 as conditional writemask.



Note that MXCSR.RM bits are ignored and unaffected by the outcome of this instruction.

Examples of instruction instances where the static rounding-mode is not allowed are shown below:

```
; rounding-mode already specified in the instruction immediate
vrndscaleps zmm7 {k6}, zmm2, 0x00

; instructions with memory operands
vmulps zmm7 {k6}, zmm2, [rax], {rd-sae}

; instructions with vector length different than MAX_VL (512-bit)
vaddps ymm7 {k6}, ymm2, ymm4, {rd-sae}
```

### 15.6.5 Compressed Disp8\*N Encoding

EVEX encoding supports a new displacement representation that allows for a more compact encoding of memory addressing commonly used in unrolled code, where an 8-bit displacement can address a range exceeding the dynamic range of an 8-bit value. This compressed displacement encoding is referred to as `disp8*N`, where `N` is a constant implied by the memory operation characteristic of each instruction.

The compressed displacement is based on the assumption that the effective displacement (of a memory operand occurring in a loop) is a multiple of the granularity of the memory access of each iteration. Since the base register in memory addressing already provides byte-granular resolution, the lower bits of the traditional `disp8` operand become redundant, and can be implied from the memory operation characteristic.

The memory operation characteristics depend on the following:

- The destination operand is updated as a full vector, a single element, or multi-element tuples.
- The memory source operand (or vector source operand if the destination operand is memory) is fetched (or treated) as a full vector, a single element, or multi-element tuples.

For example:

```
vaddps zmm7, zmm2, disp8[membase + index*8]
```

The destination `zmm7` is updated as a full 512-bit vector, and 64-bytes of data are fetched from memory as a full vector; the next unrolled iteration may fetch from memory in 64-byte granularity per iteration. There are 6 bits of lowest address that can be compressed, hence  $N = 2^6 = 64$ . The contribution of “`disp8`” to effective address calculation is  $64 * \text{disp8}$ .

```
vbroadcastf32x4 zmm7, disp8[membase + index*8]
```

In `VBROADCASTF32X4`, memory is fetched as a 4tuple of 4 32-bit entities. Hence the common lowest address bits that can be compressed are 4, corresponding to the 4tuple width of  $2^4 = 16$  bytes (4x32 bits). Therefore,  $N = 2^4$ .

For EVEX encoded instructions that update only one element in the destination, or the source element is fetched individually, the number of lowest address bits that can be compressed is generally the width in bytes of the data element, hence  $N = 2^{(\text{width})}$ .

## 15.7 MEMORY ALIGNMENT

Memory alignment requirements on EVEX-encoded SIMD instructions are similar to VEX-encoded SIMD instructions. Memory alignment applies to EVEX-encoded SIMD instructions in three categories:

- Explicitly-aligned SIMD load and store instructions accessing 64 bytes of memory with EVEX prefix encoded vector length of 512 bits (e.g., `VMOVAPD`, `VMOVAPS`, `VMOVDQA`, etc.). These instructions always require the memory address to be aligned on a 64-byte boundary.

- Explicitly-unaligned SIMD load and store instructions accessing 64 bytes or less of data from memory (e.g., VMOVUPD, VMOVUPS, VMOVDQU, VMOVQ, VMOVD, etc.). These instructions do not require the memory address to be aligned on a natural vector-length byte boundary.
- Most arithmetic and data processing instructions encoded using EVEX support memory access semantics. When these instructions access from memory, there are no alignment restrictions.

Software may see performance penalties when unaligned accesses cross cacheline boundaries or vector-length naturally-aligned boundaries, so reasonable attempts to align commonly used data sets should continue to be pursued.

Atomic memory operation in Intel 64 and IA-32 architecture is guaranteed only for a subset of memory operand sizes and alignment scenarios. The guaranteed atomic operations are described in Section 7.1.1, “Task Structure” of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*. AVX and FMA instructions do not introduce any new guaranteed atomic memory operations.

AVX-512 instructions may generate an #AC(0) fault on misaligned 4 or 8-byte memory references in Ring-3 when CR0.AM=1. 16, 32 and 64-byte memory references will not generate an #AC(0) fault. See Table 15-7 for details.

Certain AVX-512 Foundation instructions always require 64-byte alignment (see the complete list of VEX and EVEX encoded instructions in Table 15-6). These instructions will #GP(0) if not aligned to 64-byte boundaries.

**Table 15-6. SIMD Instructions Requiring Explicitly Aligned Memory**

Require 16-byte alignment	Require 32-byte alignment	Require 64-byte alignment*
(V)MOVDQA xmm, m128	VMOVDQA ymm, m256	VMOVDQA zmm, m512
(V)MOVDQA m128, xmm	VMOVDQA m256, ymm	VMOVDQA m512, zmm
(V)MOVAPS xmm, m128	VMOVAPS ymm, m256	VMOVAPS zmm, m512
(V)MOVAPS m128, xmm	VMOVAPS m256, ymm	VMOVAPS m512, zmm
(V)MOVAPD xmm, m128	VMOVAPD ymm, m256	VMOVAPD zmm, m512
(V)MOVAPD m128, xmm	VMOVAPD m256, ymm	VMOVAPD m512, zmm
(V)MOVNTDQA xmm, m128	VMOVNTPS m256, ymm	VMOVNTPS m512, zmm
(V)MOVNTPS m128, xmm	VMOVNTPD m256, ymm	VMOVNTPD m512, zmm
(V)MOVNTPD m128, xmm	VMOVNTDQ m256, ymm	VMOVNTDQ m512, zmm
(V)MOVNTDQ m128, xmm	VMOVNTDQA ymm, m256	VMOVNTDQA zmm, m512

**Table 15-7. Instructions Not Requiring Explicit Memory Alignment**

(V)MOVDQU xmm, m128	VMOVDQU ymm, m256	VMOVDQU zmm, m512
(V)MOVDQU m128, m128	VMOVDQU m256, ymm	VMOVDQU m512, zmm
(V)MOVUPS xmm, m128	VMOVUPS ymm, m256	VMOVUPS zmm, m512
(V)MOVUPS m128, xmm	VMOVUPS m256, ymm	VMOVUPS m512, zmm
(V)MOVUPD xmm, m128	VMOVUPD ymm, m256	VMOVUPD zmm, m512
(V)MOVUPD m128, xmm	VMOVUPD m256, ymm	VMOVUPD m512, zmm

## 15.8 SIMD FLOATING-POINT EXCEPTIONS

AVX-512 instructions can generate SIMD floating-point exceptions (#XM) if embedded “suppress all exceptions” (SAE) in EVEX is not set. When SAE is not set, these instructions will respond to exception masks of MXCSR in the same way as VEX-encoded AVX instructions. When CR4.OSXMMEXCPT=0, any unmasked FP exceptions generate an Undefined Opcode exception (#UD).

## 15.9 INSTRUCTION EXCEPTION SPECIFICATION

Exception behavior of VEX-encoded AVX / AVX2 instructions are described in *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*. Exception behavior of AVX-512 Foundation instructions and additional 512-bit extensions are described in Section 2.7, "Exception Classifications of EVEX-Encoded instructions" and Section 2.8, "Exception Classifications of Opmask instructions".

## 15.10 EMULATION

Setting the CR0.EM bit to 1 provides a technique to emulate legacy SSE floating-point instruction sets in software. This technique is not supported with AVX instructions, nor FMA instructions.

If an operating system wishes to emulate AVX instructions, set XCR0[2:1] to zero. This will cause AVX instructions to #UD. Emulation of FMA by the operating system can be done similarly as with emulating AVX instructions.

## 15.11 WRITING FLOATING-POINT EXCEPTION HANDLERS

AVX-512, AVX and FMA floating-point exceptions are handled in an entirely analogous way to legacy SSE floating-point exceptions. To handle unmasked SIMD floating-point exceptions, the operating system or executive must provide an exception handler. Section 11.5.1, "SIMD Floating-Point Exceptions", describes the SIMD floating-point exception classes and gives suggestions for writing an exception handler to handle them.

To indicate that the operating system provides a handler for SIMD floating-point exceptions (#XM), the CR4.OSXMMEXCPT flag (bit 10) must be set.



## 5. Updates to Chapter 17, Volume 1

Change bars show changes to Chapter 17 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture*.

-----  
Changes include the addition of MPX address-width adjust (MAWA) information, and details on IA32\_BNDCFGS. Also updated Intel MPX and System Manage Mode details.

### 17.1 INTEL® MEMORY PROTECTION EXTENSIONS (INTEL® MPX)

Intel® Memory Protection Extensions (Intel® MPX) is a new capability introduced into Intel Architecture. Intel MPX can increase the robustness of software when it is used in conjunction with compiler changes to check memory references, for those references whose compile-time normal intentions are usurped at runtime due to buffer overflow or underflow. Two of the most important goals of Intel MPX are to provide this capability at low performance overhead for newly compiled code, and to provide compatibility mechanisms with legacy software components. A direct benefit Intel MPX provides is hardening software against malicious attacks designed to cause or exploit buffer overruns. This chapter describes the software visible interfaces of this extension.

### 17.2 INTRODUCTION

Intel MPX is designed to allow a system (i.e., the logical processor(s) and the OS software) to run both Intel MPX enabled software and legacy software (written for processors without Intel MPX). When executing software containing a mixture of Intel MPX-unaware code (legacy code) and Intel MPX-enabled code, the legacy code does not benefit from Intel MPX, but it also does not experience any change in functionality or reduction in performance. The performance of Intel MPX-enabled code running on processors that do not support Intel MPX may be similar to the use of embedding NOPs in the instruction stream.

Intel MPX is designed such that an Intel MPX enabled application can link with, call into, or be called from legacy software (libraries, etc.) while maintaining existing application binary interfaces (ABIs). And in most cases, the benefit of Intel MPX requires minimal changes to the source code at the application programming interfaces (APIs) to legacy library/applications. As described later, Intel MPX associates **bounds** with pointers in a novel manner, and the Intel MPX hardware uses **bounds** to check that the pointer based accesses are suitably constrained. Intel MPX enabled software is not required to uniformly or universally utilize the new hardware capabilities over all memory references. Specifically, programmers can selectively use Intel MPX to protect a subset of pointers.

The code enabled for Intel MPX benefits from memory protection against vulnerability such as buffer overrun. Therefore there is a heightened incentive for software vendors to adopt this technology. At the same time, the security benefit of Intel MPX-protection can be implemented according to the business priorities of software vendors. A software vendor can choose to adopt Intel MPX in some modules to realize partial benefit from Intel MPX quickly, and introduce Intel MPX in other modules in phases (e.g. some programmer intervention might be required at the interface to legacy calls). This adaptive property of Intel MPX is designed to give software vendors control on their schedule and modularity of adoption. It also allows a software vendor to secure defense for higher priority or more attack-prone software first; and allows the use of Intel MPX features in one phase of software engineering (e.g., testing) and not in another (e.g., general release) as dictated by business realities.

The initial goal of Intel MPX is twofold: (1) provide means to defend a system against attacks that originate external to some trust perimeter where the trust perimeter subsumes the system memory and integral data repositories, and (2) provide means to pinpoint accidental logic defects in pointer usage, by undergirding memory references with hardware based pointer validation.

As with any instruction set extensions, Intel MPX can be used by application developers beyond detecting buffer overflow, the processor does not limit the use of Intel MPX for buffer overflow detection.

### 17.3 INTEL MPX PROGRAMMING ENVIRONMENT

Intel MPX introduces new **bounds registers** and new instructions that operate on bounds registers. Intel MPX allows an OS to support user mode software (operating at CPL=3) and supervisor mode software (CPL < 3) to add memory protection capability against buffer overrun. It provides controls to enable Intel MPX extensions for user mode and supervisor mode independently. Intel MPX extensions are designed to allow software to associate bounds with pointers, and allow software to check memory references against the bounds associated with the

pointer to prevent out of bound memory access (thus preventing buffer overflow). The bounds registers hold lower bound and upper bound that can be checked when referencing memory. An out-of-bounds memory reference then causes a #BR exception. Intel MPX also introduces configuration facilities that the OS must manage to support enabling of user-mode (and/or supervisor-mode) software operations using bounds registers.

### 17.3.1 Detection and Enumeration of Intel MPX Interfaces

Detection of hardware support for processor extended state component is provided by the main CPUID leaf function 0DH with index ECX = 0. Specifically, the return value in EDX:EAX of CPUID.(EAX=0DH, ECX=0) provides a 64-bit wide bit vector of hardware support of processor state components.

If CPUID.(EAX=07H,ECX=0H):EBX.MPX[bit 14] = 1 (the processor supports Intel MPX), CPUID.(EAX=0DH,ECX=0):EAX[bits 4:3] will enumerate the XSAVE state components associated with Intel MPX. These two component states of Intel MPX are the following:

- **BNDREGS:** CPUID.(EAX=0DH,ECX=0):EAX[3] indicates XCR0.BNDREGS[bit 3] is supported. This bit indicates bound register component of Intel MPX state, comprised of four bounds registers, BND0-BND3 (see Section 17.3.2).
- **BNDCSR:** CPUID.(EAX=0DH,ECX=0):EAX[4] indicates XCR0.BNDCSR[bit 4] is supported. This bit indicates bounds configuration and status component of Intel MPX comprised of BNDCFGU and BNDSTATUS. OS must enable both BNDCSR and BNDREGS bits in XCR0 to ensure full Intel MPX support to applications.
- The size of the processor state component, enabled by XCR0.BNDREGS, is enumerated by CPUID.(EAX=0DH,ECX=03H).EAX[31:0] and the byte offset of this component relative to the beginning of the XSAVE/XRSTOR area is reported by CPUID.(EAX=0DH, ECX=03H).EBX[31:0].
- The size of the processor state component, enabled by XCR0.BNDCSR, is enumerated by CPUID.(EAX=0DH,ECX=04H).EAX[31:0] and the byte offset of this component relative to the beginning of the XSAVE/XRSTOR area is reported by CPUID.(EAX=0DH, ECX=04H).EBX[31:0].

On processors that support Intel MPX, CPUID.(EAX=0DH,ECX=0):EAX[3] and CPUID.(EAX=0DH,ECX=0):EAX[4] will both be 1. On processors that do not support Intel MPX, CPUID.(EAX=0DH,ECX=0):EAX[3] and CPUID.(EAX=0DH,ECX=0):EAX[4] will both be 0.

The layout of XCR0 for extended processor state components defined in Intel Architecture is shown in Figure 2-8 of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Enabling Intel MPX requires an OS to manage bits [4:3] of XCR0; see Section 13.5.

The BNDLDX and BNDSTX instructions (Section 17.4.3) each take an operand whose bits are used to traverse data structures in memory. In 64-bit mode, these instructions operate only on the lower bits in the supplied 64-bit addresses. The number of bits used is 48 plus a value called the **MPX address-width adjust (MAWA)**. The MAWA value depends on CPL:

- If CPL < 3, the supervisor MAWA (**MAWAS**) is used. This value is 0.
- If CPL = 3, the user MAWA (**MAWAU**) is used. The value of MAWAU is enumerated in CPUID.(EAX=07H,ECX=0H):ECX.MAWAU[bits 21:17].

(Outside of 64-bit mode, BNDLDX and BNDSTX use the entire 32 bits of the supplied linear-address operands.)

### 17.3.2 Bounds Registers

Intel MPX Architecture defines four new registers, BND0-BND3, which Intel MPX instructions operate on. Each bounds register stores a pair of 64-bit values which are the lower bound (LB) and upper bound (UB) of a buffer, see Figure 17-1.

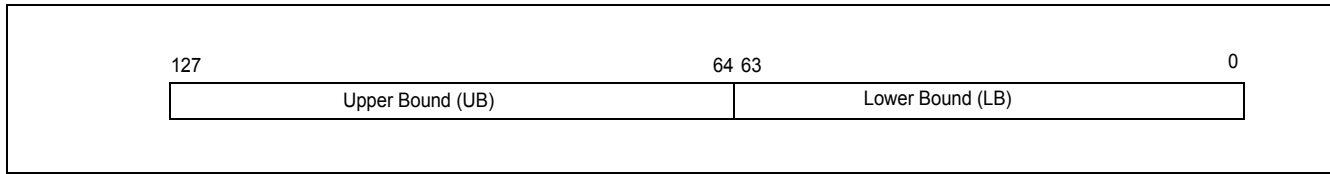


Figure 17-1. Layout of the Bounds Registers BND0-BND3

The bounds are unsigned effective addresses, and are inclusive. The upper bounds are architecturally represented in 1's complement form. Lower bound = 0, and upper bound = 0 (1's complement of all 1s) will allow access to the entire address space. The bounds are considered as INIT when both lower and upper bounds are 0 (cover the entire address space). The two Intel MPX instructions which operate on the upper bound (BNDMK and BNDCU) account for the 1's complement representation of the upper bounds.

The instruction set does not impose any conventions on the use of bounds registers. Software has full flexibility associating pointers to bounds registers including sharing them for multiple pointers.

RESET or INIT# will initialize (write zero to) BND0–BND3.

### 17.3.3 Configuration and Status Registers

Intel MPX defines two configuration registers and one status register. The two configuration registers are defined for user mode (CPL = 3) and supervisor mode (CPL < 3). The user-mode configuration register BNDCFGU is accessible only with the XSAVE feature set instructions.

The supervisor mode configuration register is an MSR, referred to as IA32\_BNDCFGS (MSR 0D90H). Because both configuration registers share a common layout (see Figure 17-2), when describing the common behavior, these configuration registers are often denoted as BNDCFGx, where x can be U or S, for user and supervisor mode respectively.

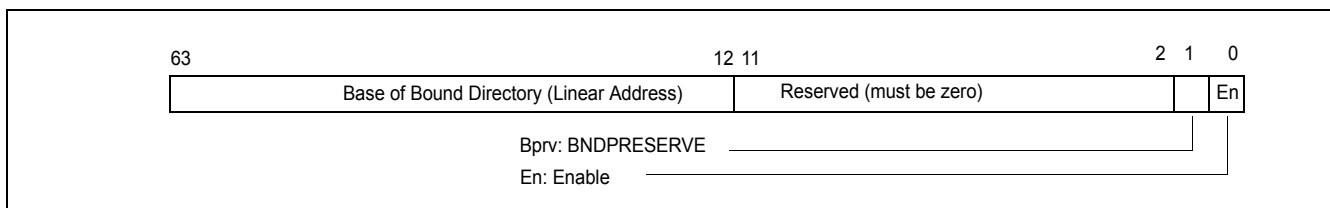


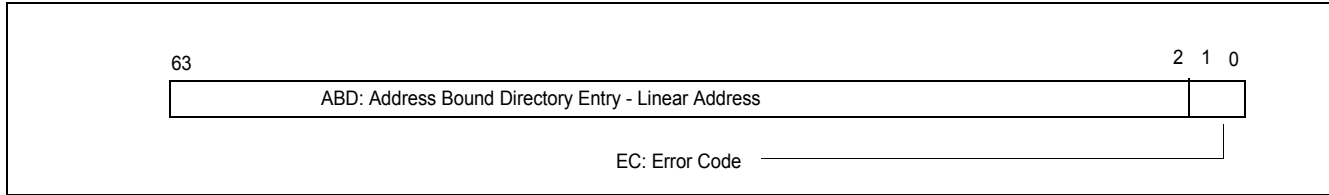
Figure 17-2. Common Layout of the Bound Configuration Registers BNDCFGU and BNDCFGS

The Enable bit in BNDCFGU enables Intel MPX in user mode (CPL = 3), and the Enable bit in BNDCFGS enables Intel MPX in supervisor mode (CPL < 3). The BNDPRESERVE bit controls the initialization behavior of CALL/RET/JMP/Jcc instructions without the BND (F2H) prefix -- see Section 17.5.3.

WRMSR to BNDCFGS will #GP if any of the reserved bits of BNDCFGS is not zero or if the base address of the bound directory is not canonical. XRSTOR of BNDCFGU ignores the reserved bits and does not fault if any is non-zero; similarly, it ignores the upper bits of the base address of the bound directory and sign-extends the highest implemented bit of the linear address to guarantee the canonicity of this address.

Intel MPX also defines a status register (BNDSTATUS) primarily used to communicate status information for #BR exception. The layout of the status register is shown in Figure 17-3.





**Figure 17-3. Layout of the Bound Status Registers BNDSTATUS**

The BNDSTATUS register provides two fields to communicate the status of Intel MPX operations:

- EC (bits 1:0): The error code field communicates status information of a bound range exception #BR or operation involving bound directory.
- ABD: (bits 63:2): The address field of a bound directory entry can provide information when operation on the bound directory caused a #BR.

The valid error codes are defined in Table 17-1.

**Table 17-1. Error Code Definition of BNDSTATUS**

EC	Description	Meaning
00b <sup>1</sup>	No Intel MPX exception	No exception caused by Intel MPX operations.
01b	Bounds violation	#BR caused by BNDCL, BNDCU or BNDCN instructions; ABD is 0.
10b	Invalid BD entry	#BR caused by BNDLDX or BNDSTX instructions, ABD will be set to the linear address of the invalid bound-directory entry
11b	Reserved	Reserved

**NOTES:**

1. When legacy BOUND instruction cause a #BR with Intel MPX enabled (see Section 17.5.4), EC is written with Zero.

RESET or INIT# will set BNDCFGx and BNDSTATUS registers to zero.

### 17.3.4 Read and Write of IA32\_BNDCFGS

The RDMSR and WRMSR instructions can be used to read and write the IA32\_BNDCFGS MSR. (The XSAVE state does not include IA32\_BNDCFGS, and instructions in the XSAVE feature set do not access that register). Attempts to write to IA32\_BNDCFGS check for canonicity of the addresses being loaded into IA32\_BNDCFGS (regardless of mode at the time of execution) and will #GP if the address is not canonical or if reserved bits would be set.

Software can use RDMSR and WRMSR to read and write IA32\_BNDCFGS as long as the processor implements Intel MPX, i.e. CPUID.(EAX=07H, ECX=0H).EBX.MPX = 1. The states of CR4 and XCR0 have no impact on the ability to access IA32\_BNDCFGS.

## 17.4 INTEL MPX INSTRUCTION SUMMARY

When Intel MPX is not enabled or not present, all Intel MPX instructions behave as NOP. There are eight Intel MPX instructions, Table 17-2 provides a summary.

A C/C++ compiler can implement intrinsic support for Intel MPX instructions to facilitate pointer operation with capability of checking for valid bounds on pointers. Typically, Intel MPX intrinsics are implemented by compiler via inline code generation where bounds register allocations are handled by the compiler without requiring the

programmer to directly manipulate any bounds registers. Therefore no new data type for a bounds register is needed in the syntax of Intel MPX intrinsics.

**Table 17-2. Intel MPX Instruction Summary**

Intel MPX Instruction	Description
BNDMK b, m	Create LowerBound (LB) and UpperBound (UB) in the bounds register b
BNDCL b, r/m	Checks the address of a memory reference or address in r against the lower bound
BNDUCU b, r/m	Checks the address of a memory reference or address in r against the upper bound in 1's complement form
BNDUCN b, r/m	Checks the address of a memory reference or address in r against the upper bound not in 1's complement form
BNDMOV b, b/m	Copy/load LB and UB bounds from memory or a bounds register
BNDMOV b/m, b	Store LB and UB bounds in a bounds register to memory or another register
BNDLDX b, mib	Load bounds using address translation using an sib-addressing expression mib
BNDSTX mib, b	Store bounds using address translation using an sib-addressing expression mib

### 17.4.1 Instruction Encoding

All Intel MPX instructions are NOP on processors that report CPUID.(EAX=07H, ECX=0H).EBX.MPX [bit 14] = 0, or if Intel MPX is not enabled by the operating system (see Section 13.5). Applications can selectively opt-in to use Intel MPX instructions.

All Intel MPX opcodes encoded to operate on BND0-BND3 are valid Intel MPX instructions. All Intel MPX opcodes encoded to operate on bound registers beyond BND3 will #UD if Intel MPX is enabled.

BNDLDX/BNDSTX opcodes require 66H as a mandatory prefix with its operand size tied to the address size attribute of the supported operating modes. Attempt to override operand size attribute with 66H or with REX.W in 64-bit mode is ignored.

### 17.4.2 Usage and Examples

BNDMK is typically used after memory is allocated for a buffer, e.g., by functions such as malloc, calloc, or when the memory is allocated on the stack. However, many other usages are possible such as when accessing an array member of a structure.

#### Example 17-1. BNDMK Example Usage in Application and Library Code

<pre>int A[100]; //assume the array A is allocated on the stack at 'offset'            // from RBP.            // the instruction to store starting address of array will be:            LEA RAX, [RBP+offset]            // the instruction to create the bounds for array A will be:            BNDMK BND0, [RAX+399]            // Store RAX into BND0.LB, and ~(RAX+399) into BND0.UB.</pre>	<pre>// similarly, for a library implementation of dynamic allocated // memory            int * k = malloc(100);            // assuming that malloc returns pointer k in RAX and holds (size            // - 1) in RCX            // the malloc implementation will execute the following            // instruction before returning:            BNDMK BND0, [RAX+RCX]            // BND0.LB stores RAX, and BND0.UB stores ~(RAX+RCX)</pre>
--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

BNDMOV is typically used to copy bounds from one bound register to another when a pointer is copied from one general purpose register to another, or to spill/fill bounds into memory corresponding to a spill/fill of a pointer.

#### Example 17-2. BNDMOV Example

<pre>Spilling or caller save of bound register would use BNDMOV [RBP+ offset], BNDx.</pre>
<pre>Assuming that the calling convention is that bound of first pointer is passed in BND0, and that bound happens to be in BND3 before the call, the software will add instruction BNDMOV BND0, BND3 prior to the call.</pre>

BNDCL/BNDCU/BNDCN are typically used before writing to a buffer but can be used in other instances as well. If there are no bounds violations as a result of bound check instruction, the processor will proceed to execute the next instruction. However, if the bound check fails, it will signal #BR exception (fault).

Typically, the pointer used to write to memory will be compared against lower bound. However, for upper bound check, the software must add the (operand size - 1) to the pointer before upper bound checking.

For example, the software intend to write 32-bit integer in 64-bit mode into a buffer at address specified in RAX, and the bounds are in register BND0, the instruction sequence will be:

```
BNDCL BND0, [RAX]
BNDCU BND0, [RAX+3] ; operand size is 4
MOV Dword ptr [RAX], RBX ; RBX has the data to be written to the buffer.
```

Software may move one of the two bound checks out of a loop if it can determine that memory is accessed strictly in ascending or descending order. For string instructions of the form REP MOVS, the software may choose to do check lower bound against first access and upper bound against last access to memory. However, if software wants to also check for wrap around conditions as part of address computation, it should check for both upper and lower bound for first and last instructions (total of four bound checks).

BNDSTX is used to store the bounds associated with a buffer and the “pointer value” of the pointer to that buffer onto a bound table entry via address translation using a two-level structure, see Section 17.4.3.

For example, the software has a buffer with bounds stored in BND0, the pointer to the buffer is in ESI, the following sequence will store the “pointer value” (the buffer) and the bounds into a configured bound table entry using address translation from the linear address associated with the base of a SIB-addressing form consisting of a base register and a index register:

```
MOV ECX, Dword ptr [ESI] ; store the pointer value in the index register ECX
MOV EAX, ESI ; store the pointer in the base register EAX
BNDSTX Dword ptr [EAX+ECX], BND0 ; perform address translation from the linear address of the base
EAX and store bounds and pointer value ECX onto a bound table entry.
```

Similarly to retrieve a buffer and its associated bounds from a bound table entry:

```
MOV EAX, dword ptr [EBX] ;
BNDLDX BND0, dword ptr [EBX+EAX]; perform address translation from the linear address of the base EBX,
and loads bounds and pointer value from a bound table entry
```

### 17.4.3 Loading and Storing Bounds in Memory

Intel MPX defines two instructions to load and store of the linear address of a pointer to a buffer, along with the bounds of the buffer into a data structure of extended bounds. When storing these extended bounds, the processor parses the address of the pointer (where it is stored) to locate an entry in a **bound table** in which to store the extended bounds. Loading of an extended bounds performs the reverse sequence.

The memory representation of an extended bound is a 4-tuple consisting of lower bound, upper bound, pointer value and a reserved field (for use by future versions of Intel MPX; software must not use this field). Accesses to these extended bounds use 32-bit or 64-bit operands according to the current paging mode. Thus, a bound table entry is 4\*64 bits (32 bytes) in 64-bit mode and 4\*32 bits (16 bytes) outside 64-bit mode. The linear address of a bound table is stored in a bound-directory entry (BDE). The linear address of the **bound directory** is derived from either BNDCFGU (CPL = 3) or BNDCFGS (CPL < 3).

The bound directory and bound tables are stored in application memory and are allocated by the application (in case of kernel use, the structures will be in kernel memory). The bound directory and each bound table are in contiguous linear memory.

Software should take care to allocate sufficient memory for the bound directory and the bound tables. The amount of memory required depends on the current operating mode and, in some cases, on CPL:

- In 64-bit mode:
  - Each bound table comprises of  $2^{17}$  32-byte entries thus, the size of a bound table in 64-bit mode is 4 MBytes.

- The size of the bound directory depends on the value of MAWA. Specifically, the bound directory comprises of  $2^{28+MAWA}$  64-bit entries; thus, the size of a bound directory in 64-bit mode is  $2^{1+MAWA}$  GBytes. The value of MAWA depends on CPL:
  - If  $CPL < 3$ , the supervisor MAWA (MAWAS) is used. This value is 0. Thus, when  $CPL < 3$ , a bound directory comprises of  $2^{28}$  64-bit entries and the size of a bound directory is 2 GBytes.
  - If  $CPL = 3$ , the user MAWA (MAWAU) is used. The value of MAWAU is enumerated in CPUID.(EAX=07H,ECX=0H):ECX.MAWAU[bits 21:17]. When  $CPL = 3$ , a bound directory comprises of  $2^{28+MAWAU}$  64-bit entries and the size of a bound directory is  $2^{1+MAWAU}$  GBytes.

**NOTE**

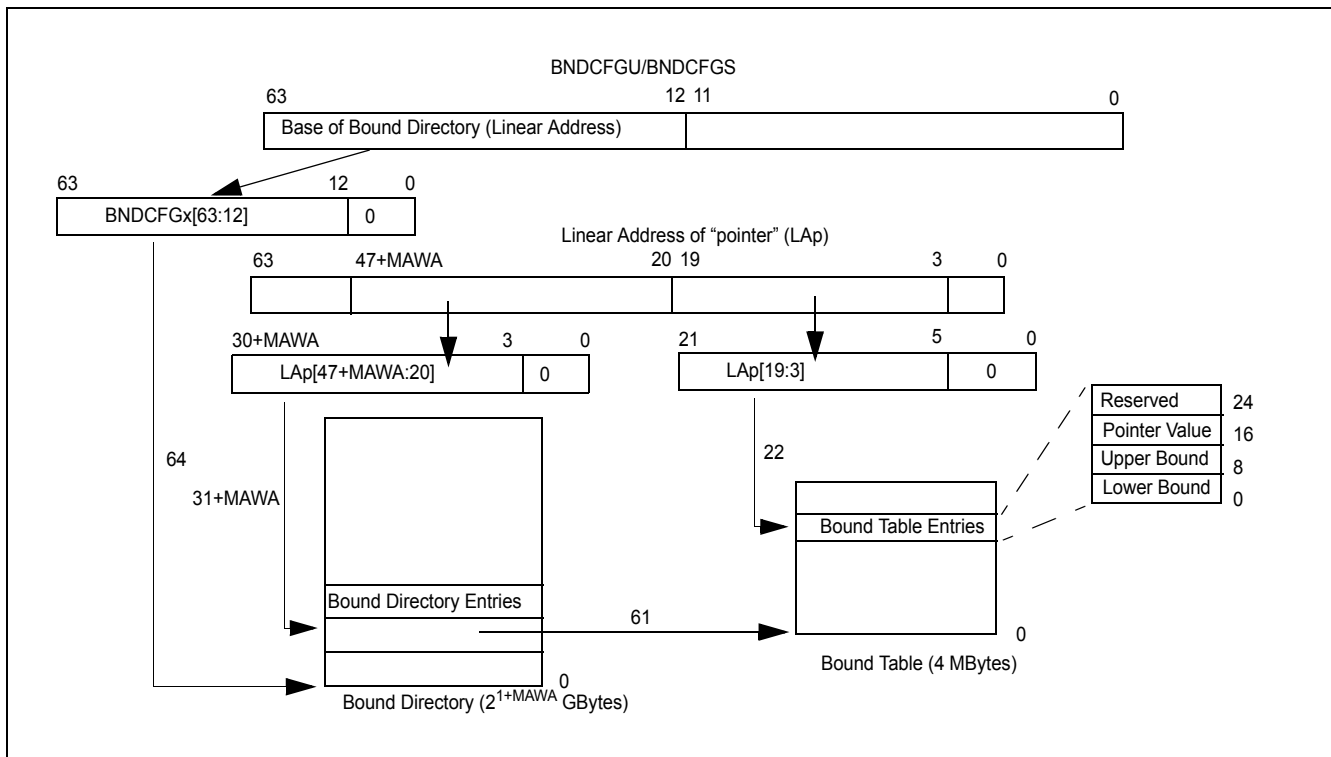
Software operating with  $CPL = 3$  in 64-bit mode should use CPUID to determine the proper amount of memory to allocate for the bound directory.

- Outside 64-bit mode:
  - Each bound table comprises of  $2^{10}$  16-byte entries; thus, the size of a bound table outside 64-bit mode is 16 KBytes.
  - The bound directory comprises of  $2^{20}$  32-bit entries; thus, the size of a bound directory outside 64-bit mode is 4 MBytes. This size is independent of MAWA and CPL.

Bounds in memory are associated with the memory address where the pointer is stored, i.e., Ap. A linear address LAp is computed by adding the appropriate segment base to Ap. (Note: for these instructions, the segment override applies only to the computation.) Section 17.4.3.1 and Section 17.4.3.2 describe how BNDLDX and BNDSTX parse LAp to locate a bound-directory entry (BDE), which contains the address of a bound table, and then a bound-table entry (BTE), which contains the extended bounds for the pointer.

**17.4.3.1 BNDLDX and BNDSTX in 64-Bit Mode**

Figure 17-4 shows the two-level structures for address translation of extended bounds in 64-bit mode.



**Figure 17-4. Bound Paging Structure and Address Translation in 64-Bit Mode**

As noted earlier, the linear address of the bound directory is derived from either BNDCFGU (CPL = 3) or BNDCFGS (CPL < 3). In 64-bit mode, each bound-directory entry (BDE) is 8 bytes. The number of entries in the bound directory is determined by the MPX address-width adjust (MAWA; see Section 17.3.1). Specifically, the number of entries is  $2^{28+MAWA}$ .

In 64-bit mode, the processor uses the two-level structures to access extended bounds as follows:

- A bound directory is located at the 4-KByte aligned linear address specified in bits 63:12 of BNDCFGx (see Figure 17-2). A bound directory comprises of  $2^{28+MAWA}$  64-bit entries (BDEs); thus, the size of a bound directory in 64-bit mode is  $2^{1+MAWA}$  GBytes. A BDE is selected using the LAp (linear address of pointer to a buffer) to construct a 64-bit offset as follows:
  - bits 63:31+MAWA are 0;
  - bits 30+MAWA:3 are LAp[47+MAWA:20]; and
  - bits 2:0 are 0.

The address of the BDE is the sum of the bound-directory base address (from BNDCFGx) plus this 64-bit offset.

- Bit 0 of a BDE is a valid bit. If this bit is 0, use of the BDE by BNDLDX or BNDSTX causes #BR, sets BNDSTATUS[1:0] to 10b (the error code), and loads BNDSTATUS[63:2] with bits 63:2 of the linear address of the BDE. Otherwise, the processor uses bits 63:3 of the BDE as the 8-byte aligned address of a bound table (BT); the processor ignores bits 2:1 of a BDE.

A bound table comprises of  $2^{17}$  32-byte entries (BTEs); thus, the size of a bound table in 64-bit mode is 4 MBytes. A BTE is selected using the LAp (linear address of pointer to a buffer) to construct an offset as follows:

- bits 21:5 are LAp[19:3]; and
- bits 4:0 are 0.

The address of the BTE is the sum of the bound-table base address (from the BDE) plus this offset.

- Each BTE comprises the following:
  - a 64-bit lower bound (LB) field;
  - a 64-bit upper bound (UB) field;
  - a 64-bit pointer value; and
  - a 64-bit reserved field. This field is reserved for future Intel MPX; software must not use it.

### 17.4.3.2 BNDLDX and BNDSTX Outside 64-Bit Mode

Figure 17-5 shows the two-level structures for address translation of extended bounds outside 64-bit mode.

As noted earlier, the linear address of the bound directory is derived from either BNDCFGU (CPL = 3) or BNDCFGS (CPL < 3). Outside 64-bit mode, each bound-directory entry (BDE) is 4 bytes. The number of entries in the bound directory is  $2^{20}$ .

Outside 64-bit mode, the processor uses the two-level structures to access extended bounds as follows:

- A bound directory is located at the 4-KByte aligned linear address specified in bits 31:12 of BNDCFGx (see Figure 17-2). A bound directory comprises of  $2^{20}$  32-bit entries (BDEs); thus, the size of a bound directory outside 64-bit mode is 4 MBytes. A BDE is selected using the LAp (linear address of pointer to a buffer) to construct an offset as follows:
  - bits 21:2 are LAp[31:12]; and
  - bits 1:0 are 0.

The address of the BDE is the sum of the bound-directory base address (from BNDCFGx) plus this offset.

- Bit 0 of a BDE is a valid bit. If this bit is 0, use of the BDE by BNDLDX or BNDSTX causes #BR, sets BNDSTATUS[1:0] to 10b (the error code), and loads BNDSTATUS[31:2] with bits 31:2 of the linear address of the BDE. Otherwise, the processor uses bits 31:2 of the BDE as the 4-byte aligned address of a bound table (BT); the processor ignores bit 1 of a BDE.

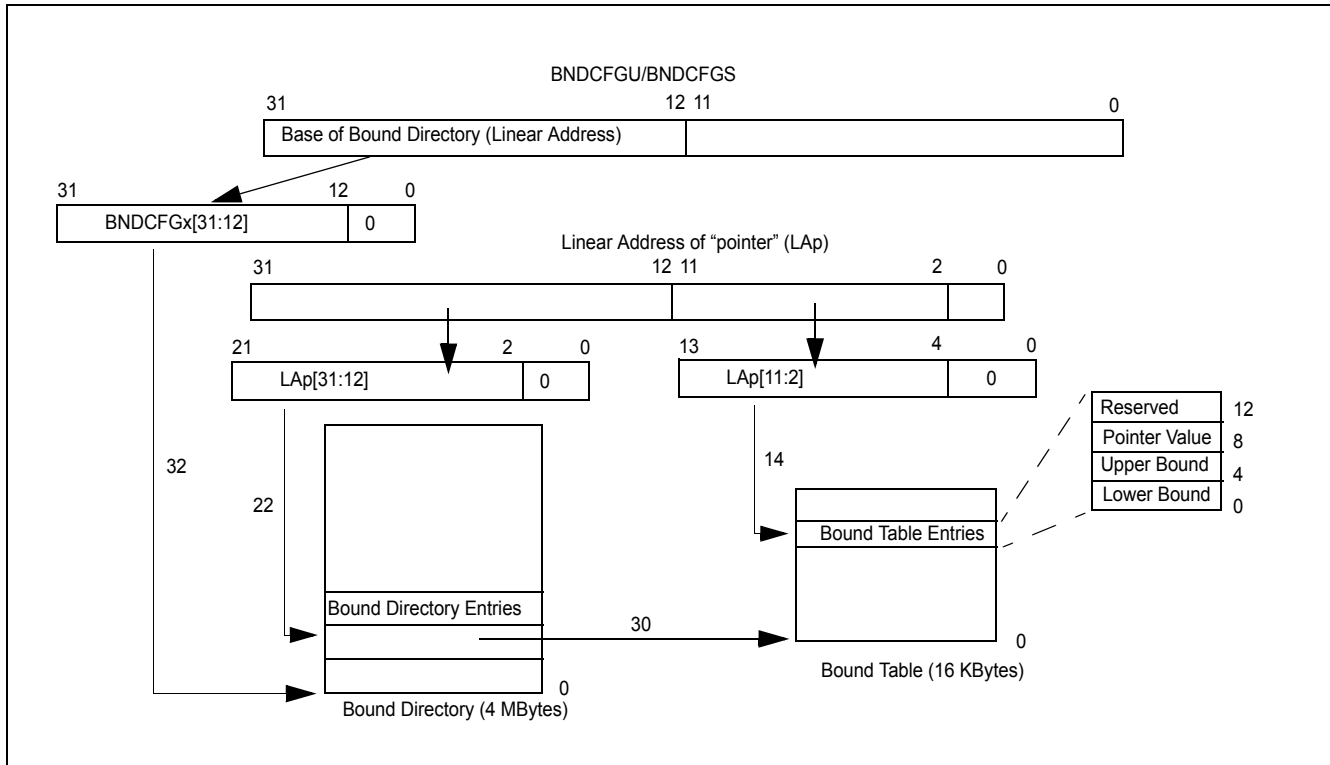


Figure 17-5. Bound Paging Structure and Address Translation Outside 64-Bit Mode

A bound table comprises of  $2^{10}$  16-byte entries (BTEs); thus, the size of a bound table outside 64-bit mode is 16 KBytes. A BTE is selected using the LAp (linear address of pointer to a buffer) to construct an offset as follows:

- bits 13:4 are LAp[11:2]; and
- bits 3:0 are 0.

The address of the BTE is the sum of the bound-table base address (from the BDE) plus this offset. This address is used as an offset into the DS segment to determine the linear address of the BTE.

- Each BTE comprises the following:
  - a 32-bit lower bound (LB) field;
  - a 32-bit upper bound (UB) field;
  - a 32-bit pointer value; and
  - a 32-bit reserved field. This field is reserved for future Intel MPX; software must not use it.

## 17.5 INTERACTIONS WITH INTEL MPX

### 17.5.1 Intel MPX and Operating Modes

In 64-bit Mode, all Intel MPX instructions use 64-bit operands for bounds and 64-bit addressing, i.e. REX.W & 67H have no effect on data or address size.

XSAVE, XSAVEOPT and XRSTOR load/store 64-bit values in all modes, as these state-management instructions are not Intel MPX instructions.

In compatibility and legacy modes (including 16-bit code segments, real and virtual 8086 modes) all Intel MPX instructions use 32-bit operands for bounds and 32 bit addressing. The upper 32-bits of destination bound register are cleared (consistent with behavior of integer registers)

In 32-bit and compatibility mode, the bounds are 32-bit, and are treated same as 32-bit integer registers. Therefore, when 32-bit bound is updated in a bound register, the upper 32-bits are undefined. When switching from 64-bit, the behavior of content of bounds register will be similar to that of general purpose registers.

Table 17-3 describes the impact of 67H prefix on memory forms of Intel MPX instructions (register-only forms ignore 67H prefix) when Intel MPX is enabled:

**Table 17-3. Effective Address Size of Intel MPX Instructions with 67H Prefix**

Addressing Mode	67H Prefix	Effective Address Size used for Intel MPX instructions when Intel MPX is enabled
64-bit Mode	Y	64 bit addressing used
64-bit Mode	N	64 bit addressing used
32-bit Mode	Y	#UD
32-bit Mode	N	32 bit addressing used
16-bit Mode	Y	32 bit addressing used
16-bit Mode	N	#UD

### 17.5.2 Intel MPX Support for Pointer Operations with Branching

Intel MPX provides flexibility in supporting pointer operation across control flow changes. Intel MPX allows

- compatibility with legacy code that may perform pointer operation across control flow changes and are unaware of Intel MPX, along with
- Intel MPX-aware code that adds bounds checking protection to pointer operation across control flow changes.

The interface to provide such flexibility consists of:

- Using a prefix, referred to as BND prefix, to relevant branch instructions: CALL, RET, JMP and Jcc
- BNDCFGU and BNDCFGs provides the bit field, BNDPRESERVE (bit 1).

The value of BNDPRESERVE in conjunction with the presence/absence the BND prefix with those branching instruction will determine whether the values in BND0-BND3 will be initialized or unchanged.

### 17.5.3 CALL, RET, JMP and All Jcc

An application compiled to use Intel MPX will use the REPNE (F2H) prefix (denoted by BND) for all forms of near CALL, near RET, near JMP, short & near Jcc instructions (BND+CALL, BND+RET, BND+JMP, BND+Jcc). See Table 17-4 for specific opcodes. All far CALL, RET and JMP instructions plus short JMP (JMP rel 8, opcode EB) instructions will never cause bound registers to be initialized.

If BNDPRESERVE bit is one, above instructions will NOT INIT the bounds registers when BND prefix is not present for above instructions (legacy behavior). However, If BNDPRESERVE is zero, above instructions will INIT ALL bound registers (BND0-BND3) when BND prefix is not present for above instructions. If BND prefix is present for above instructions, the BND registers will NOT INIT any bound registers (BND0-BND3).

The legacy code will continue to use non-prefixed forms of these instructions, so if BNDPRESERVE is zero, all the bound registers will INIT by legacy code. This allows the legacy function to execute and return to callee with all bound registers initialized (legacy code by definition cannot make or load bounds in bound registers because it does not have Intel MPX instructions). This will eliminate compatibility concerns when legacy function might have changed the pointer in registers but did not update the value of the bounds registers associated with these pointers.

If BNDCFGx.BNDPRESERVE is clear then non-prefixed forms of these instructions will initialize all the bound registers. If this bit is set then non-prefixed and prefixed forms of these instructions will preserve the contents of bound registers as shown in Table 17-4.

**Table 17-4. Bounds Register INIT Behavior Due to BND Prefix with Branch Instructions**

Instruction	Branch Instruction Opcodes	BNDPRESERVE = 0	BNDPRESERVE = 1
CALL	E8, FF/2	Init BND0-BND3	BND0-BND3 unchanged
BND + CALL	F2 E8, F2 FF/2	BND0-BND3 unchanged	BND0-BND3 unchanged
RET	C2, C3	Init BND0-BND3	BND0-BND3 unchanged
BND + RET	F2 C2, F2 C3	BND0-BND3 unchanged	BND0-BND3 unchanged
JMP	E9, FF/4	Init BND0-BND3	BND0-BND3 unchanged
BND + JMP	F2 E9, F2 FF/4	BND0-BND3 unchanged	BND0-BND3 unchanged
Jcc	70 through 7F, 0F 80 through 0F 8F	Init BND0-BND3	BND0-BND3 unchanged
BND + Jcc	F2 70 through F2 7F, F2 0F 80 through F2 0F 8F	BND0-BND3 unchanged	BND0-BND3 unchanged

### 17.5.4 BOUND Instruction and Intel MPX

If Intel MPX is enabled (see Section 13.5) and a #BR was caused due to a BOUND instruction, then BOUND instruction will write zero to the BNDSTATUS register. In all other situations, BOUND instruction will not modify BNDSTATUS. Specifically, the operation of the BOUND instruction can be described as:

```
IF ( ( BOUND instruction caused #BR) AND ( CR4.OXSSAVE = 1 AND XCRO.BNDREGS=1 AND XCRO.BNDCSR = 1) AND
  ((CPL=3 AND BNDCFGU.ENABLE = 1) OR (CPL < 3 AND BNDCFGS.ENABLE = 1)) ) THEN
  BNDSTATUS ← 0;
ELSE
  BNDSTATUS is not modified;
FI;
```

### 17.5.5 Programming Considerations

Intel MPX instruction set does not dictate any calling convention, but allows the calling convention extensions to be interoperable with legacy code by making use of the of the bound registers and the bound tables to convey arguments and return values.

### 17.5.6 Intel MPX and System Manage Mode

Upon delivery of an SMI to a processor supporting Intel MPX, the contents of IA32\_BNDCFGS is saved to SMM state save map (at offset 7ED0H) and the register is then cleared when entering into SMM. RSM restores IA32\_BNDCFGS from the SMM state save map. The instruction forces the reserved bits (11:2) to 0 and sign-extends the highest implemented bit of the linear address to guarantee the canonicity of this address (regardless of what is in SMM state save map).

The content of IA32\_BNDCFGS is cleared after entering into SMM. Thus, Intel MPX is disabled inside an SMM handler until SMM code enables it explicitly. This will prevent the side-effect of INIT-ing bound registers by legacy CALL/RET/JMP/Jcc in SMM code.

### 17.5.7 Support of Intel MPX in VMCS

A new guest-state field for IA32\_BNDCFGS is added to the VMCS. In addition, two new controls are added:

- a VM-exit control called "clear BNDCFGS"
- a VM-entry control called "load BNDCFGS."



VM exits always save IA32\_BNDCFGS into BNDCFGS field of VMCS; if “clear BNDCFGS” is 1, VM exits clear IA32\_BNDCFGS. If “load BNDCFGS” is 1, VM entry loads IA32\_BNDCFGS from VMCS. If loading IA32\_BNDCFGS, VM entry should check the value of that register in the guest-state area of the VMCS and cause the VM entry to fail (late) if the value is one that would cause WRMSR to fault if executed in ring 0.

### 17.5.8 Support of Intel MPX in Intel TSX

For some processor implementations, the following Intel MPX instructions may always cause transactional aborts:

- An Intel TSX transaction abort will occur in case of legacy branch (that causes bounds registers INIT) when at least one bounds register was in a NON-INIT state.
- An Intel TSX transaction abort will occur in case of a BNDLDX & BNDSTX instruction on non-flat segment.

Intel MPX Instructions (including BND prefix + branch instructions) not enumerated above as causing transactional abort when used inside a transaction will typically not cause an Intel TSX transaction to abort.

## 6. Updates to Chapter 1, Volume 2A

Change bars show changes to Chapter 1 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A: Instruction Set Reference, A-L*.

-----  
Changes include minor updates to reflect volume addition, arrangement of information and document order numbers.

The *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volumes 2A, 2B, 2C & 2D: Instruction Set Reference* (order numbers 253666, 253667, 326018 and 334569) are part of a set that describes the architecture and programming environment of all Intel 64 and IA-32 architecture processors. Other volumes in this set are:

- The *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture* (Order Number 253665).
- The *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volumes 3A, 3B, 3C & 3D: System Programming Guide* (order numbers 253668, 253669, 326019 and 332831).

The *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, describes the basic architecture and programming environment of Intel 64 and IA-32 processors. The *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volumes 2A, 2B, 2C & 2D*, describe the instruction set of the processor and the opcode structure. These volumes apply to application programmers and to programmers who write operating systems or executives. The *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volumes 3A, 3B, 3C & 3D*, describe the operating-system support environment of Intel 64 and IA-32 processors. These volumes target operating-system and BIOS designers. In addition, the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*, addresses the programming environment for classes of software that host operating systems.

## 1.1 INTEL® 64 AND IA-32 PROCESSORS COVERED IN THIS MANUAL

This manual set includes information pertaining primarily to the most recent Intel 64 and IA-32 processors, which include:

- Pentium® processors
- P6 family processors
- Pentium® 4 processors
- Pentium® M processors
- Intel® Xeon® processors
- Pentium® D processors
- Pentium® processor Extreme Editions
- 64-bit Intel® Xeon® processors
- Intel® Core™ Duo processor
- Intel® Core™ Solo processor
- Dual-Core Intel® Xeon® processor LV
- Intel® Core™2 Duo processor
- Intel® Core™2 Quad processor Q6000 series
- Intel® Xeon® processor 3000, 3200 series
- Intel® Xeon® processor 5000 series
- Intel® Xeon® processor 5100, 5300 series
- Intel® Core™2 Extreme processor X7000 and X6800 series
- Intel® Core™2 Extreme processor QX6000 series
- Intel® Xeon® processor 7100 series
- Intel® Pentium® Dual-Core processor
- Intel® Xeon® processor 7200, 7300 series
- Intel® Xeon® processor 5200, 5400, 7400 series

## ABOUT THIS MANUAL

- Intel® Core™2 Extreme processor QX9000 and X9000 series
- Intel® Core™2 Quad processor Q9000 series
- Intel® Core™2 Duo processor E8000, T9000 series
- Intel® Atom™ processor family
- Intel® Atom™ processors 200, 300, D400, D500, D2000, N200, N400, N2000, E2000, Z500, Z600, Z2000, C1000 series are built from 45 nm and 32 nm processes
- Intel® Core™ i7 processor
- Intel® Core™ i5 processor
- Intel® Xeon® processor E7-8800/4800/2800 product families
- Intel® Core™ i7-3930K processor
- 2nd generation Intel® Core™ i7-2xxx, Intel® Core™ i5-2xxx, Intel® Core™ i3-2xxx processor series
- Intel® Xeon® processor E3-1200 product family
- Intel® Xeon® processor E5-2400/1400 product family
- Intel® Xeon® processor E5-4600/2600/1600 product family
- 3rd generation Intel® Core™ processors
- Intel® Xeon® processor E3-1200 v2 product family
- Intel® Xeon® processor E5-2400/1400 v2 product families
- Intel® Xeon® processor E5-4600/2600/1600 v2 product families
- Intel® Xeon® processor E7-8800/4800/2800 v2 product families
- 4th generation Intel® Core™ processors
- The Intel® Core™ M processor family
- Intel® Core™ i7-59xx Processor Extreme Edition
- Intel® Core™ i7-49xx Processor Extreme Edition
- Intel® Xeon® processor E3-1200 v3 product family
- Intel® Xeon® processor E5-2600/1600 v3 product families
- 5th generation Intel® Core™ processors
- Intel® Xeon® processor D-1500 product family
- Intel® Xeon® processor E5 v4 family
- Intel® Atom™ processor X7-Z8000 and X5-Z8000 series
- Intel® Atom™ processor Z3400 series
- Intel® Atom™ processor Z3500 series
- 6th generation Intel® Core™ processors
- Intel® Xeon® processor E3-1500m v5 product family

P6 family processors are IA-32 processors based on the P6 family microarchitecture. This includes the Pentium® Pro, Pentium® II, Pentium® III, and Pentium® III Xeon® processors.

The Pentium® 4, Pentium® D, and Pentium® processor Extreme Editions are based on the Intel NetBurst® microarchitecture. Most early Intel® Xeon® processors are based on the Intel NetBurst® microarchitecture. Intel Xeon processor 5000, 7100 series are based on the Intel NetBurst® microarchitecture.

The Intel® Core™ Duo, Intel® Core™ Solo and dual-core Intel® Xeon® processor LV are based on an improved Pentium® M processor microarchitecture.

The Intel® Xeon® processor 3000, 3200, 5100, 5300, 7200, and 7300 series, Intel® Pentium® dual-core, Intel® Core™2 Duo, Intel® Core™2 Quad, and Intel® Core™2 Extreme processors are based on Intel® Core™ microarchitecture.

The Intel® Xeon® processor 5200, 5400, 7400 series, Intel® Core™2 Quad processor Q9000 series, and Intel® Core™2 Extreme processors QX9000, X9000 series, Intel® Core™2 processor E8000 series are based on Enhanced Intel® Core™ microarchitecture.

The Intel® Atom™ processors 200, 300, D400, D500, D2000, N200, N400, N2000, E2000, Z500, Z600, Z2000, C1000 series are based on the Intel® Atom™ microarchitecture and supports Intel 64 architecture.

The Intel® Core™ i7 processor and Intel® Xeon® processor 3400, 5500, 7500 series are based on 45 nm Intel® microarchitecture code name Nehalem. Intel® microarchitecture code name Westmere is a 32 nm version of Intel® microarchitecture code name Nehalem. Intel® Xeon® processor 5600 series, Intel Xeon processor E7 and various Intel Core i7, i5, i3 processors are based on Intel® microarchitecture code name Westmere. These processors support Intel 64 architecture.

The Intel® Xeon® processor E5 family, Intel® Xeon® processor E3-1200 family, Intel® Xeon® processor E7-8800/4800/2800 product families, Intel® Core™ i7-3930K processor, and 2nd generation Intel® Core™ i7-2xxx, Intel® Core™ i5-2xxx, Intel® Core™ i3-2xxx processor series are based on the Intel® microarchitecture code name Sandy Bridge and support Intel 64 architecture.

The Intel® Xeon® processor E7-8800/4800/2800 v2 product families, Intel® Xeon® processor E3-1200 v2 product family and 3rd generation Intel® Core™ processors are based on the Intel® microarchitecture code name Ivy Bridge and support Intel 64 architecture.

The Intel® Xeon® processor E5-4600/2600/1600 v2 product families, Intel® Xeon® processor E5-2400/1400 v2 product families and Intel® Core™ i7-49xx Processor Extreme Edition are based on the Intel® microarchitecture code name Ivy Bridge-E and support Intel 64 architecture.

The Intel® Xeon® processor E3-1200 v3 product family and 4th Generation Intel® Core™ processors are based on the Intel® microarchitecture code name Haswell and support Intel 64 architecture.

The Intel® Core™ M processor family, 5th generation Intel® Core™ processors, Intel® Xeon® processor D-1500 product family and the Intel® Xeon® processor E5 v4 family are based on the Intel® microarchitecture code name Broadwell and support Intel 64 architecture.

The Intel® Xeon® processor E3-1500m v5 product family and 6th generation Intel® Core™ processors are based on the Intel® microarchitecture code name Skylake and support Intel 64 architecture.

The Intel® Xeon® processor E5-2600/1600 v3 product families and the Intel® Core™ i7-59xx Processor Extreme Edition are based on the Intel® microarchitecture code name Haswell-E and support Intel 64 architecture.

The Intel® Atom™ processor Z8000 series is based on the Intel microarchitecture code name Airmont.

The Intel® Atom™ processor Z3400 series and the Intel® Atom™ processor Z3500 series are based on the Intel microarchitecture code name Silvermont.

P6 family, Pentium® M, Intel® Core™ Solo, Intel® Core™ Duo processors, dual-core Intel® Xeon® processor LV, and early generations of Pentium 4 and Intel Xeon processors support IA-32 architecture. The Intel® Atom™ processor Z5xx series support IA-32 architecture.

The Intel® Xeon® processor 3000, 3200, 5000, 5100, 5200, 5300, 5400, 7100, 7200, 7300, 7400 series, Intel® Core™2 Duo, Intel® Core™2 Extreme, Intel® Core™2 Quad processors, Pentium® D processors, Pentium® Dual-Core processor, newer generations of Pentium 4 and Intel Xeon processor family support Intel® 64 architecture.

IA-32 architecture is the instruction set architecture and programming environment for Intel's 32-bit microprocessors. Intel® 64 architecture is the instruction set architecture and programming environment which is the superset of Intel's 32-bit and 64-bit architectures. It is compatible with the IA-32 architecture.

## 1.2 OVERVIEW OF VOLUME 2A, 2B, 2C AND 2D: INSTRUCTION SET REFERENCE

A description of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volumes 2A, 2B, 2C & 2D* content follows:

**Chapter 1 — About This Manual.** Gives an overview of all seven volumes of the *Intel® 64 and IA-32 Architectures Software Developer's Manual*. It also describes the notational conventions in these manuals and lists related Intel® manuals and documentation of interest to programmers and hardware designers.

**Chapter 2 — Instruction Format.** Describes the machine-level instruction format used for all IA-32 instructions and gives the allowable encodings of prefixes, the operand-identifier byte (ModR/M byte), the addressing-mode specifier byte (SIB byte), and the displacement and immediate bytes.

**Chapter 3 — Instruction Set Reference, A-L.** Describes Intel 64 and IA-32 instructions in detail, including an algorithmic description of operations, the effect on flags, the effect of operand- and address-size attributes, and the exceptions that may be generated. The instructions are arranged in alphabetical order. General-purpose, x87 FPU, Intel MMX™ technology, SSE/SSE2/SSE3/SSSE3/SSE4 extensions, and system instructions are included.

**Chapter 4 — Instruction Set Reference, M-U.** Continues the description of Intel 64 and IA-32 instructions started in Chapter 3. It starts *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B*.

**Chapter 5 — Instruction Set Reference, V-Z.** Continues the description of Intel 64 and IA-32 instructions started in chapters 3 and 4. It provides the balance of the alphabetized list of instructions and starts *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2C*.

**Chapter 6— Safer Mode Extensions Reference.** Describes the safer mode extensions (SMX). SMX is intended for a system executive to support launching a measured environment in a platform where the identity of the software controlling the platform hardware can be measured for the purpose of making trust decisions. This chapter starts *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2D*.

**Appendix A — Opcode Map.** Gives an opcode map for the IA-32 instruction set.

**Appendix B — Instruction Formats and Encodings.** Gives the binary encoding of each form of each IA-32 instruction.

**Appendix C — Intel® C/C++ Compiler Intrinsic and Functional Equivalents.** Lists the Intel® C/C++ compiler intrinsics and their assembly code equivalents for each of the IA-32 MMX and SSE/SSE2/SSE3 instructions.

## 1.3 NOTATIONAL CONVENTIONS

This manual uses specific notation for data-structure formats, for symbolic representation of instructions, and for hexadecimal and binary numbers. A review of this notation makes the manual easier to read.

### 1.3.1 Bit and Byte Order

In illustrations of data structures in memory, smaller addresses appear toward the bottom of the figure; addresses increase toward the top. Bit positions are numbered from right to left. The numerical value of a set bit is equal to two raised to the power of the bit position. IA-32 processors are “little endian” machines; this means the bytes of a word are numbered starting from the least significant byte. Figure 1-1 illustrates these conventions.

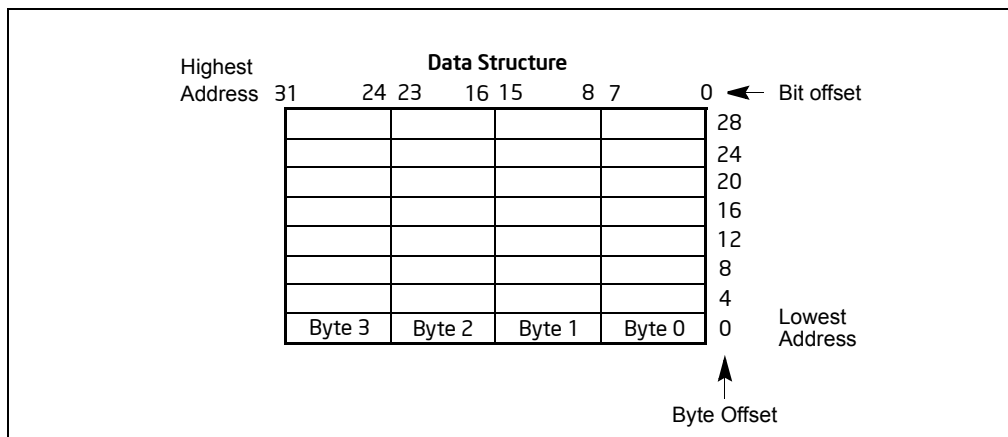


Figure 1-1. Bit and Byte Order

### 1.3.2 Reserved Bits and Software Compatibility

In many register and memory layout descriptions, certain bits are marked as **reserved**. When bits are marked as reserved, it is essential for compatibility with future processors that software treat these bits as having a future, though unknown, effect. The behavior of reserved bits should be regarded as not only undefined, but unpredictable. Software should follow these guidelines in dealing with reserved bits:

- Do not depend on the states of any reserved bits when testing the values of registers which contain such bits. Mask out the reserved bits before testing.
- Do not depend on the states of any reserved bits when storing to memory or to a register.
- Do not depend on the ability to retain information written into any reserved bits.
- When loading a register, always load the reserved bits with the values indicated in the documentation, if any, or reload them with values previously read from the same register.

#### NOTE

Avoid any software dependence upon the state of reserved bits in IA-32 registers. Depending upon the values of reserved register bits will make software dependent upon the unspecified manner in which the processor handles these bits. Programs that depend upon reserved values risk incompatibility with future processors.

### 1.3.3 Instruction Operands

When instructions are represented symbolically, a subset of the IA-32 assembly language is used. In this subset, an instruction has the following format:

```
label: mnemonic argument1, argument2, argument3
```

where:

- A **label** is an identifier which is followed by a colon.
- A **mnemonic** is a reserved name for a class of instruction opcodes which have the same function.
- The operands *argument1*, *argument2*, and *argument3* are optional. There may be from zero to three operands, depending on the opcode. When present, they take the form of either literals or identifiers for data items. Operand identifiers are either reserved names of registers or are assumed to be assigned to data items declared in another part of the program (which may not be shown in the example).

When two operands are present in an arithmetic or logical instruction, the right operand is the source and the left operand is the destination.

For example:

```
LOADREG: MOV EAX, SUBTOTAL
```

In this example, LOADREG is a label, MOV is the mnemonic identifier of an opcode, EAX is the destination operand, and SUBTOTAL is the source operand. Some assembly languages put the source and destination in reverse order.

### 1.3.4 Hexadecimal and Binary Numbers

Base 16 (hexadecimal) numbers are represented by a string of hexadecimal digits followed by the character H (for example, F82EH). A hexadecimal digit is a character from the following set: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F.

Base 2 (binary) numbers are represented by a string of 1s and 0s, sometimes followed by the character B (for example, 1010B). The "B" designation is only used in situations where confusion as to the type of number might arise.

### 1.3.5 Segmented Addressing

The processor uses byte addressing. This means memory is organized and accessed as a sequence of bytes. Whether one or more bytes are being accessed, a byte address is used to locate the byte or bytes in memory. The range of memory that can be addressed is called an **address space**.

The processor also supports segmented addressing. This is a form of addressing where a program may have many independent address spaces, called **segments**. For example, a program can keep its code (instructions) and stack in separate segments. Code addresses would always refer to the code space, and stack addresses would always refer to the stack space. The following notation is used to specify a byte address within a segment:

Segment-register:Byte-address

For example, the following segment address identifies the byte at address FF79H in the segment pointed by the DS register:

DS:FF79H

The following segment address identifies an instruction address in the code segment. The CS register points to the code segment and the EIP register contains the address of the instruction.

CS:EIP

### 1.3.6 Exceptions

An exception is an event that typically occurs when an instruction causes an error. For example, an attempt to divide by zero generates an exception. However, some exceptions, such as breakpoints, occur under other conditions. Some types of exceptions may provide error codes. An error code reports additional information about the error. An example of the notation used to show an exception and error code is shown below:

#PF(fault code)

This example refers to a page-fault exception under conditions where an error code naming a type of fault is reported. Under some conditions, exceptions which produce error codes may not be able to report an accurate code. In this case, the error code is zero, as shown below for a general-protection exception:

#GP(0)

### 1.3.7 A New Syntax for CPUID, CR, and MSR Values

Obtain feature flags, status, and system information by using the CPUID instruction, by checking control register bits, and by reading model-specific registers. We are moving toward a new syntax to represent this information. See Figure 1-2.



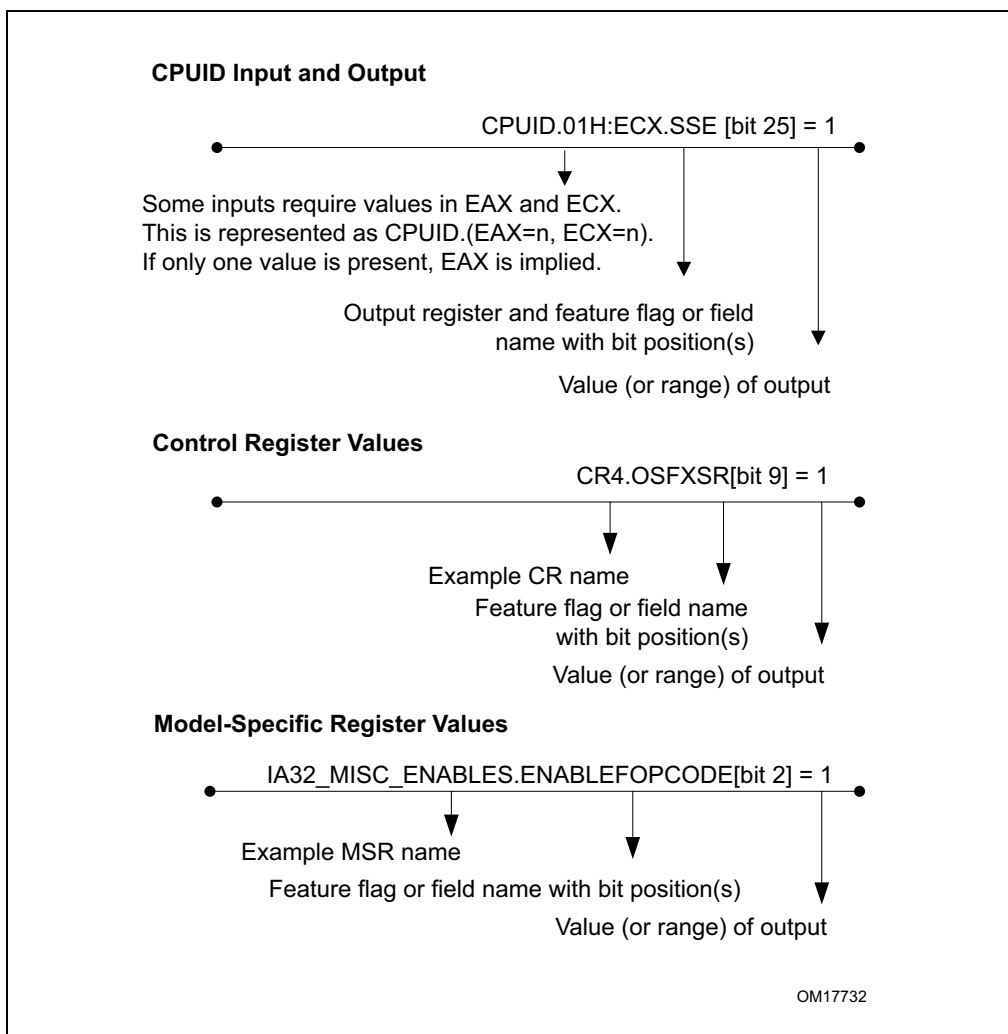


Figure 1-2. Syntax for CPUID, CR, and MSR Data Presentation

## 1.4 RELATED LITERATURE

Literature related to Intel 64 and IA-32 processors is listed and viewable on-line at:

<http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html>

See also:

- The data sheet for a particular Intel 64 or IA-32 processor
- The specification update for a particular Intel 64 or IA-32 processor
- Intel® C++ Compiler documentation and online help:  
<http://software.intel.com/en-us/articles/intel-compilers/>
- Intel® Fortran Compiler documentation and online help:  
<http://software.intel.com/en-us/articles/intel-compilers/>
- Intel® Software Development Tools:  
<http://www.intel.com/cd/software/products/asmo-na/eng/index.htm>
- Intel® 64 and IA-32 Architectures Software Developer's Manual (in three or seven volumes):  
<http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html>

## ABOUT THIS MANUAL

- Intel® 64 and IA-32 Architectures Optimization Reference Manual:  
<http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-optimization-manual.html>
- Intel 64 Architecture x2APIC Specification:  
<http://www.intel.com/content/www/us/en/architecture-and-technology/64-architecture-x2apic-specification.html>
- Intel® Trusted Execution Technology Measured Launched Environment Programming Guide:  
<http://www.intel.com/content/www/us/en/software-developers/intel-txt-software-development-guide.html>
- Developing Multi-threaded Applications: A Platform Consistent Approach:  
<https://software.intel.com/sites/default/files/article/147714/51534-developing-multithreaded-applications.pdf>
- Using Spin-Loops on Intel® Pentium® 4 Processor and Intel® Xeon® Processor:  
<http://software.intel.com/en-us/articles/ap949-using-spin-loops-on-intel-pentiumr-4-processor-and-intel-xeonr-processor/>
- Performance Monitoring Unit Sharing Guide  
<http://software.intel.com/file/30388>

Literature related to selected features in future Intel processors are available at:

- Intel® Architecture Instruction Set Extensions Programming Reference  
<https://software.intel.com/en-us/isa-extensions>
- Intel® Software Guard Extensions (Intel® SGX) Programming Reference  
<https://software.intel.com/en-us/isa-extensions/intel-sgx>

More relevant links are:

- Intel® Developer Zone:  
<https://software.intel.com/en-us>
- Developer centers:  
<http://www.intel.com/content/www/us/en/hardware-developers/developer-centers.html>
- Processor support general link:  
<http://www.intel.com/support/processors/>
- Software products and packages:  
<http://www.intel.com/cd/software/products/asmo-na/eng/index.htm>
- Intel® Hyper-Threading Technology (Intel® HT Technology):  
<http://www.intel.com/technology/platform-technology/hyper-threading/index.htm>

## 7. Updates to Chapter 2, Volume 2A

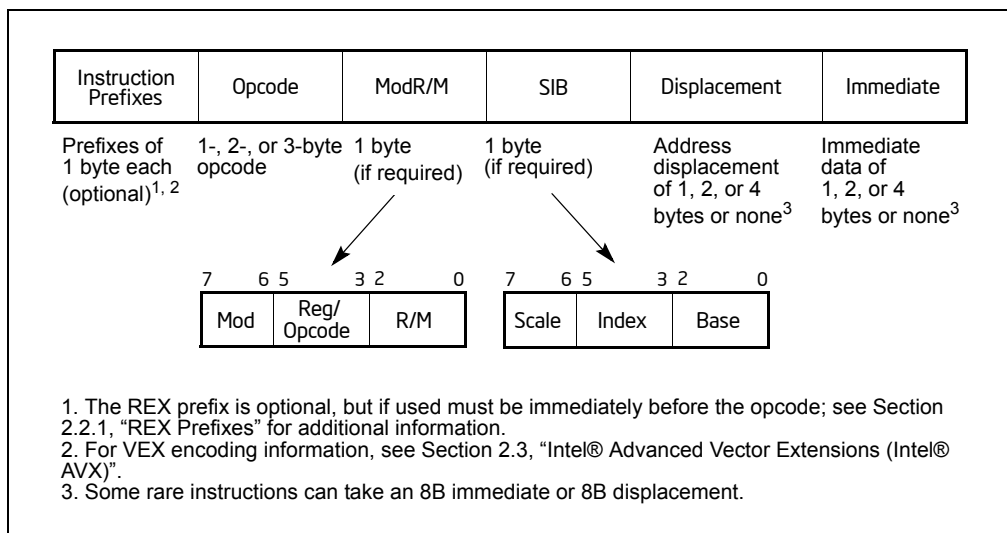
Change bars show changes to Chapter 2 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A: Instruction Set Reference, A-L*.

-----  
Changes include the addition of Intel® AVX-512 encoding details.

This chapter describes the instruction format for all Intel 64 and IA-32 processors. The instruction format for protected mode, real-address mode and virtual-8086 mode is described in Section 2.1. Increments provided for IA-32e mode and its sub-modes are described in Section 2.2.

## 2.1 INSTRUCTION FORMAT FOR PROTECTED MODE, REAL-ADDRESS MODE, AND VIRTUAL-8086 MODE

The Intel 64 and IA-32 architectures instruction encodings are subsets of the format shown in Figure 2-1. Instructions consist of optional instruction prefixes (in any order), primary opcode bytes (up to three bytes), an addressing-form specifier (if required) consisting of the ModR/M byte and sometimes the SIB (Scale-Index-Base) byte, a displacement (if required), and an immediate data field (if required).



**Figure 2-1. Intel 64 and IA-32 Architectures Instruction Format**

### 2.1.1 Instruction Prefixes

Instruction prefixes are divided into four groups, each with a set of allowable prefix codes. For each instruction, it is only useful to include up to one prefix code from each of the four groups (Groups 1, 2, 3, 4). Groups 1 through 4 may be placed in any order relative to each other.

- Group 1
  - Lock and repeat prefixes:
    - LOCK prefix is encoded using F0H.
    - REPNE/REPZ prefix is encoded using F2H. Repeat-Not-Zero prefix applies only to string and input/output instructions. (F2H is also used as a mandatory prefix for some instructions.)
    - REP or REPE/REPZ is encoded using F3H. The repeat prefix applies only to string and input/output instructions. F3H is also used as a mandatory prefix for POPCNT, LZCNT and ADOX instructions.
  - Bound prefix is encoded using F2H if the following conditions are true:
    - CPUID.(EAX=07H, ECX=0):EBX.MPX[bit 14] is set.

- BNDCFGU.EN and/or IA32\_BNDCFGS.EN is set.
- When the F2 prefix precedes a near CALL, a near RET, a near JMP, or a near Jcc instruction (see Chapter 17, “Intel® MPX,” of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*).
- Group 2
  - Segment override prefixes:
    - 2EH—CS segment override (use with any branch instruction is reserved).
    - 36H—SS segment override prefix (use with any branch instruction is reserved).
    - 3EH—DS segment override prefix (use with any branch instruction is reserved).
    - 26H—ES segment override prefix (use with any branch instruction is reserved).
    - 64H—FS segment override prefix (use with any branch instruction is reserved).
    - 65H—GS segment override prefix (use with any branch instruction is reserved).
  - Branch hints<sup>1</sup>:
    - 2EH—Branch not taken (used only with Jcc instructions).
    - 3EH—Branch taken (used only with Jcc instructions).
- Group 3
  - Operand-size override prefix is encoded using 66H (66H is also used as a mandatory prefix for some instructions).
- Group 4
  - 67H—Address-size override prefix.

The LOCK prefix (F0H) forces an operation that ensures exclusive use of shared memory in a multiprocessor environment. See “LOCK—Assert LOCK# Signal Prefix” in Chapter 3, “Instruction Set Reference, A-L,” for a description of this prefix.

Repeat prefixes (F2H, F3H) cause an instruction to be repeated for each element of a string. Use these prefixes only with string and I/O instructions (MOVS, CMPS, SCAS, LODS, STOS, INS, and OUTS). Use of repeat prefixes and/or undefined opcodes with other Intel 64 or IA-32 instructions is reserved; such use may cause unpredictable behavior.

Some instructions may use F2H, F3H as a mandatory prefix to express distinct functionality.

Branch hint prefixes (2EH, 3EH) allow a program to give a hint to the processor about the most likely code path for a branch. Use these prefixes only with conditional branch instructions (Jcc). Other use of branch hint prefixes and/or other undefined opcodes with Intel 64 or IA-32 instructions is reserved; such use may cause unpredictable behavior.

The operand-size override prefix allows a program to switch between 16- and 32-bit operand sizes. Either size can be the default; use of the prefix selects the non-default size.

Some SSE2/SSE3/SSSE3/SSE4 instructions and instructions using a three-byte sequence of primary opcode bytes may use 66H as a mandatory prefix to express distinct functionality.

Other use of the 66H prefix is reserved; such use may cause unpredictable behavior.

The address-size override prefix (67H) allows programs to switch between 16- and 32-bit addressing. Either size can be the default; the prefix selects the non-default size. Using this prefix and/or other undefined opcodes when operands for the instruction do not reside in memory is reserved; such use may cause unpredictable behavior.

---

1. Some earlier microarchitectures used these as branch hints, but recent generations have not and they are reserved for future hint usage.

## 2.1.2 Opcodes

A primary opcode can be 1, 2, or 3 bytes in length. An additional 3-bit opcode field is sometimes encoded in the ModR/M byte. Smaller fields can be defined within the primary opcode. Such fields define the direction of operation, size of displacements, register encoding, condition codes, or sign extension. Encoding fields used by an opcode vary depending on the class of operation.

Two-byte opcode formats for general-purpose and SIMD instructions consist of one of the following:

- An escape opcode byte 0FH as the primary opcode and a second opcode byte.
- A mandatory prefix (66H, F2H, or F3H), an escape opcode byte, and a second opcode byte (same as previous bullet).

For example, CVTQ2PD consists of the following sequence: F3 0F E6. The first byte is a mandatory prefix (it is not considered as a repeat prefix).

Three-byte opcode formats for general-purpose and SIMD instructions consist of one of the following:

- An escape opcode byte 0FH as the primary opcode, plus two additional opcode bytes.
- A mandatory prefix (66H, F2H, or F3H), an escape opcode byte, plus two additional opcode bytes (same as previous bullet).

For example, PHADDW for XMM registers consists of the following sequence: 66 0F 38 01. The first byte is the mandatory prefix.

Valid opcode expressions are defined in Appendix A and Appendix B.

## 2.1.3 ModR/M and SIB Bytes

Many instructions that refer to an operand in memory have an addressing-form specifier byte (called the ModR/M byte) following the primary opcode. The ModR/M byte contains three fields of information:

- The *mod* field combines with the *r/m* field to form 32 possible values: eight registers and 24 addressing modes.
- The *reg/opcode* field specifies either a register number or three more bits of opcode information. The purpose of the *reg/opcode* field is specified in the primary opcode.
- The *r/m* field can specify a register as an operand or it can be combined with the *mod* field to encode an addressing mode. Sometimes, certain combinations of the *mod* field and the *r/m* field are used to express opcode information for some instructions.

Certain encodings of the ModR/M byte require a second addressing byte (the SIB byte). The base-plus-index and scale-plus-index forms of 32-bit addressing require the SIB byte. The SIB byte includes the following fields:

- The *scale* field specifies the scale factor.
- The *index* field specifies the register number of the index register.
- The *base* field specifies the register number of the base register.

See Section 2.1.5 for the encodings of the ModR/M and SIB bytes.

## 2.1.4 Displacement and Immediate Bytes

Some addressing forms include a displacement immediately following the ModR/M byte (or the SIB byte if one is present). If a displacement is required, it can be 1, 2, or 4 bytes.

If an instruction specifies an immediate operand, the operand always follows any displacement bytes. An immediate operand can be 1, 2 or 4 bytes.

## 2.1.5 Addressing-Mode Encoding of ModR/M and SIB Bytes

The values and corresponding addressing forms of the ModR/M and SIB bytes are shown in Table 2-1 through Table 2-3: 16-bit addressing forms specified by the ModR/M byte are in Table 2-1 and 32-bit addressing forms are in Table 2-2. Table 2-3 shows 32-bit addressing forms specified by the SIB byte. In cases where the reg/opcode field in the ModR/M byte represents an extended opcode, valid encodings are shown in Appendix B.

In Table 2-1 and Table 2-2, the Effective Address column lists 32 effective addresses that can be assigned to the first operand of an instruction by using the Mod and R/M fields of the ModR/M byte. The first 24 options provide ways of specifying a memory location; the last eight (Mod = 11B) provide ways of specifying general-purpose, MMX technology and XMM registers.

The Mod and R/M columns in Table 2-1 and Table 2-2 give the binary encodings of the Mod and R/M fields required to obtain the effective address listed in the first column. For example: see the row indicated by Mod = 11B, R/M = 000B. The row identifies the general-purpose registers EAX, AX or AL; MMX technology register MM0; or XMM register XMM0. The register used is determined by the opcode byte and the operand-size attribute.

Now look at the seventh row in either table (labeled "REG ="). This row specifies the use of the 3-bit Reg/Opcode field when the field is used to give the location of a second operand. The second operand must be a general-purpose, MMX technology, or XMM register. Rows one through five list the registers that may correspond to the value in the table. Again, the register used is determined by the opcode byte along with the operand-size attribute. If the instruction does not require a second operand, then the Reg/Opcode field may be used as an opcode extension. This use is represented by the sixth row in the tables (labeled "/digit (Opcode)"). Note that values in row six are represented in decimal form.

The body of Table 2-1 and Table 2-2 (under the label "Value of ModR/M Byte (in Hexadecimal)") contains a 32 by 8 array that presents all of 256 values of the ModR/M byte (in hexadecimal). Bits 3, 4 and 5 are specified by the column of the table in which a byte resides. The row specifies bits 0, 1 and 2; and bits 6 and 7. The figure below demonstrates interpretation of one table value.

	Mod	11	
	RM	000	
/digit (Opcode);	REG =	001	
	C8H	11001000	

**Figure 2-2. Table Interpretation of ModR/M Byte (C8H)**

Table 2-1. 16-Bit Addressing Forms with the ModR/M Byte

			AL AX EAX MM0 XMM0 0 000	CL CX ECX MM1 XMM1 1 001	DL DX EDX MM2 XMM2 2 010	BL BX EBX MM3 XMM3 3 011	AH SP ESP MM4 XMM4 4 100	CH BP <sup>1</sup> EBP MM5 XMM5 5 101	DH SI ESI MM6 XMM6 6 110	BH DI EDI MM7 XMM7 7 111
Effective Address	Mod	R/M	Value of ModR/M Byte (in Hexadecimal)							
[BX+SI] [BX+DI] [BP+SI] [BP+DI] [SI] [DI] disp16 <sup>2</sup> [BX]	00	000 001 010 011 100 101 110 111	00 01 02 03 04 05 06 07	08 09 0A 0B 0C 0D 0E 0F	10 11 12 13 14 15 16 17	18 19 1A 1B 1C 1D 1E 1F	20 21 22 23 24 25 26 27	28 29 2A 2B 2C 2D 2E 2F	30 31 32 33 34 35 36 37	38 39 3A 3B 3C 3D 3E 3F
[BX+SI]+disp8 <sup>3</sup> [BX+DI]+disp8 [BP+SI]+disp8 [BP+DI]+disp8 [SI]+disp8 [DI]+disp8 [BP]+disp8 [BX]+disp8	01	000 001 010 011 100 101 110 111	40 41 42 43 44 45 46 47	48 49 4A 4B 4C 4D 4E 4F	50 51 52 53 54 55 56 57	58 59 5A 5B 5C 5D 5E 5F	60 61 62 63 64 65 66 67	68 69 6A 6B 6C 6D 6E 6F	70 71 72 73 74 75 76 77	78 79 7A 7B 7C 7D 7E 7F
[BX+SI]+disp16 [BX+DI]+disp16 [BP+SI]+disp16 [BP+DI]+disp16 [SI]+disp16 [DI]+disp16 [BP]+disp16 [BX]+disp16	10	000 001 010 011 100 101 110 111	80 81 82 83 84 85 86 87	88 89 8A 8B 8C 8D 8E 8F	90 91 92 93 94 95 96 97	98 99 9A 9B 9C 9D 9E 9F	A0 A1 A2 A3 A4 A5 A6 A7	A8 A9 AA AB AC AD AE AF	B0 B1 B2 B3 B4 B5 B6 B7	B8 B9 BA BB BC BD BE BF
EAX/AX/AL/MM0/XMM0 ECX/CX/CL/MM1/XMM1 EDX/DX/DL/MM2/XMM2 EBX/BX/BL/MM3/XMM3 ESP/SP/AHMM4/XMM4 EBP/BP/CH/MM5/XMM5 ESI/SI/DH/MM6/XMM6 EDI/DI/BH/MM7/XMM7	11	000 001 010 011 100 101 110 111	C0 C1 C2 C3 C4 C5 C6 C7	C8 C9 CA CB CC CD CE CF	D0 D1 D2 D3 D4 D5 D6 D7	D8 D9 DA DB DC DD DE DF	E0 E1 E2 E3 E4 E5 E6 E7	E8 E9 EA EB EC ED EE EF	F0 F1 F2 F3 F4 F5 F6 F7	F8 F9 FA FB FC FD FE FF

**NOTES:**

1. The default segment register is SS for the effective addresses containing a BP index, DS for other effective addresses.
2. The disp16 nomenclature denotes a 16-bit displacement that follows the ModR/M byte and that is added to the index.
3. The disp8 nomenclature denotes an 8-bit displacement that follows the ModR/M byte and that is sign-extended and added to the index.



Table 2-2. 32-Bit Addressing Forms with the ModR/M Byte

			AL	CL	DL	BL	AH	CH	DH	BH
			AX	CX	DX	BX	SP	BP	SI	DI
			EAX	ECX	EDX	EBX	ESP	EBP	ESI	EDI
			MM0	MM1	MM2	MM3	MM4	MM5	MM6	MM7
			XMM0	XMM1	XMM2	XMM3	XMM4	XMM5	XMM6	XMM7
			0	1	2	3	4	5	6	7
			000	001	010	011	100	101	110	111
Effective Address	Mod	R/M	Value of ModR/M Byte (in Hexadecimal)							
[EAX]	00	000	00	08	10	18	20	28	30	38
[ECX]		001	01	09	11	19	21	29	31	39
[EDX]		010	02	0A	12	1A	22	2A	32	3A
[EBX]		011	03	0B	13	1B	23	2B	33	3B
[--][--] <sup>1</sup>		100	04	0C	14	1C	24	2C	34	3C
disp32 <sup>2</sup>		101	05	0D	15	1D	25	2D	35	3D
[ESI]		110	06	0E	16	1E	26	2E	36	3E
[EDI]		111	07	0F	17	1F	27	2F	37	3F
[EAX]+disp8 <sup>3</sup>	01	000	40	48	50	58	60	68	70	78
[ECX]+disp8		001	41	49	51	59	61	69	71	79
[EDX]+disp8		010	42	4A	52	5A	62	6A	72	7A
[EBX]+disp8		011	43	4B	53	5B	63	6B	73	7B
[--][--]+disp8		100	44	4C	54	5C	64	6C	74	7C
[EBP]+disp8		101	45	4D	55	5D	65	6D	75	7D
[ESI]+disp8		110	46	4E	56	5E	66	6E	76	7E
[EDI]+disp8		111	47	4F	57	5F	67	6F	77	7F
[EAX]+disp32	10	000	80	88	90	98	A0	A8	B0	B8
[ECX]+disp32		001	81	89	91	99	A1	A9	B1	B9
[EDX]+disp32		010	82	8A	92	9A	A2	AA	B2	BA
[EBX]+disp32		011	83	8B	93	9B	A3	AB	B3	BB
[--][--]+disp32		100	84	8C	94	9C	A4	AC	B4	BC
[EBP]+disp32		101	85	8D	95	9D	A5	AD	B5	BD
[ESI]+disp32		110	86	8E	96	9E	A6	AE	B6	BE
[EDI]+disp32		111	87	8F	97	9F	A7	AF	B7	BF
EAX/AX/AL/MM0/XMM0	11	000	C0	C8	D0	D8	E0	E8	F0	F8
ECX/CX/CL/MM1/XMM1		001	C1	C9	D1	D9	E1	E9	F1	F9
EDX/DX/DL/MM2/XMM2		010	C2	CA	D2	DA	E2	EA	F2	FA
EBX/BX/BL/MM3/XMM3		011	C3	CB	D3	DB	E3	EB	F3	FB
ESP/SP/AH/MM4/XMM4		100	C4	CC	D4	DC	E4	EC	F4	FC
EBP/BP/CH/MM5/XMM5		101	C5	CD	D5	DD	E5	ED	F5	FD
ESI/SI/DH/MM6/XMM6		110	C6	CE	D6	DE	E6	EE	F6	FE
EDI/DI/BH/MM7/XMM7		111	C7	CF	D7	DF	E7	EF	F7	FF

**NOTES:**

1. The [--][--] nomenclature means a SIB follows the ModR/M byte.
2. The disp32 nomenclature denotes a 32-bit displacement that follows the ModR/M byte (or the SIB byte if one is present) and that is added to the index.
3. The disp8 nomenclature denotes an 8-bit displacement that follows the ModR/M byte (or the SIB byte if one is present) and that is sign-extended and added to the index.

Table 2-3 is organized to give 256 possible values of the SIB byte (in hexadecimal). General purpose registers used as a base are indicated across the top of the table, along with corresponding values for the SIB byte’s base field. Table rows in the body of the table indicate the register used as the index (SIB byte bits 3, 4 and 5) and the scaling factor (determined by SIB byte bits 6 and 7).

Table 2-3. 32-Bit Addressing Forms with the SIB Byte

r32 (In decimal) Base = (In binary) Base =			EAX 0 000	ECX 1 001	EDX 2 010	EBX 3 011	ESP 4 100	[*] 5 101	ESI 6 110	EDI 7 111
Scaled Index	SS	Index	Value of SIB Byte (in Hexadecimal)							
[EAX] [ECX] [EDX] [EBX] none [EBP] [ESI] [EDI]	00	000 001 010 011 100 101 110 111	00 08 10 18 20 28 30 38	01 09 11 19 21 29 31 39	02 0A 12 1A 22 2A 32 3A	03 0B 13 1B 23 2B 33 3B	04 0C 14 1C 24 2C 34 3C	05 0D 15 1D 25 2D 35 3D	06 0E 16 1E 26 2E 36 3E	07 0F 17 1F 27 2F 37 3F
[EAX*2] [ECX*2] [EDX*2] [EBX*2] none [EBP*2] [ESI*2] [EDI*2]	01	000 001 010 011 100 101 110 111	40 48 50 58 60 68 70 78	41 49 51 59 61 69 71 79	42 4A 52 5A 62 6A 72 7A	43 4B 53 5B 63 6B 73 7B	44 4C 54 5C 64 6C 74 7C	45 4D 55 5D 65 6D 75 7D	46 4E 56 5E 66 6E 76 7E	47 4F 57 5F 67 6F 77 7F
[EAX*4] [ECX*4] [EDX*4] [EBX*4] none [EBP*4] [ESI*4] [EDI*4]	10	000 001 010 011 100 101 110 111	80 88 90 98 A0 A8 B0 B8	81 89 91 99 A1 A9 B1 B9	82 8A 92 9A A2 AA B2 BA	83 8B 93 9B A3 AB B3 BB	84 8C 94 9C A4 AC B4 BC	85 8D 95 9D A5 AD B5 BD	86 8E 96 9E A6 AE B6 BE	87 8F 97 9F A7 AF B7 BF
[EAX*8] [ECX*8] [EDX*8] [EBX*8] none [EBP*8] [ESI*8] [EDI*8]	11	000 001 010 011 100 101 110 111	C0 C8 D0 D8 E0 E8 F0 F8	C1 C9 D1 D9 E1 E9 F1 F9	C2 CA D2 DA E2 EA F2 FA	C3 CB D3 DB E3 EB F3 FB	C4 CC D4 DC E4 EC F4 FC	C5 CD D5 DD E5 ED F5 FD	C6 CE D6 DE E6 EE F6 FE	C7 CF D7 DF E7 EF F7 FF

**NOTES:**

- The [\*] nomenclature means a disp32 with no base if the MOD is 00B. Otherwise, [\*] means disp8 or disp32 + [EBP]. This provides the following address modes:

MOD bits    Effective Address

- 00            [scaled index] + disp32  
01            [scaled index] + disp8 + [EBP]  
10            [scaled index] + disp32 + [EBP]

## 2.2 IA-32E MODE

IA-32e mode has two sub-modes. These are:

- Compatibility Mode.** Enables a 64-bit operating system to run most legacy protected mode software unmodified.
- 64-Bit Mode.** Enables a 64-bit operating system to run applications written to access 64-bit address space.

## 2.2.1 REX Prefixes

REX prefixes are instruction-prefix bytes used in 64-bit mode. They do the following:

- Specify GPRs and SSE registers.
- Specify 64-bit operand size.
- Specify extended control registers.

Not all instructions require a REX prefix in 64-bit mode. A prefix is necessary only if an instruction references one of the extended registers or uses a 64-bit operand. If a REX prefix is used when it has no meaning, it is ignored.

Only one REX prefix is allowed per instruction. If used, the REX prefix byte must immediately precede the opcode byte or the escape opcode byte (0FH). When a REX prefix is used in conjunction with an instruction containing a mandatory prefix, the mandatory prefix must come before the REX so the REX prefix can be immediately preceding the opcode or the escape byte. For example, CVTDQ2PD with a REX prefix should have REX placed between F3 and 0F E6. Other placements are ignored. The instruction-size limit of 15 bytes still applies to instructions with a REX prefix. See Figure 2-3.

Legacy Prefixes	REX Prefix	Opcode	ModR/M	SIB	Displacement	Immediate
Grp 1, Grp 2, Grp 3, Grp 4 (optional)	(optional)	1-, 2-, or 3-byte opcode	1 byte (if required)	1 byte (if required)	Address displacement of 1, 2, or 4 bytes	Immediate data of 1, 2, or 4 bytes or none

Figure 2-3. Prefix Ordering in 64-bit Mode

### 2.2.1.1 Encoding

Intel 64 and IA-32 instruction formats specify up to three registers by using 3-bit fields in the encoding, depending on the format:

- ModR/M: the reg and r/m fields of the ModR/M byte.
- ModR/M with SIB: the reg field of the ModR/M byte, the base and index fields of the SIB (scale, index, base) byte.
- Instructions without ModR/M: the reg field of the opcode.

In 64-bit mode, these formats do not change. Bits needed to define fields in the 64-bit context are provided by the addition of REX prefixes.

### 2.2.1.2 More on REX Prefix Fields

REX prefixes are a set of 16 opcodes that span one row of the opcode map and occupy entries 40H to 4FH. These opcodes represent valid instructions (INC or DEC) in IA-32 operating modes and in compatibility mode. In 64-bit mode, the same opcodes represent the instruction prefix REX and are not treated as individual instructions.

The single-byte-opcode forms of the INC/DEC instructions are not available in 64-bit mode. INC/DEC functionality is still available using ModR/M forms of the same instructions (opcodes FF/0 and FF/1).

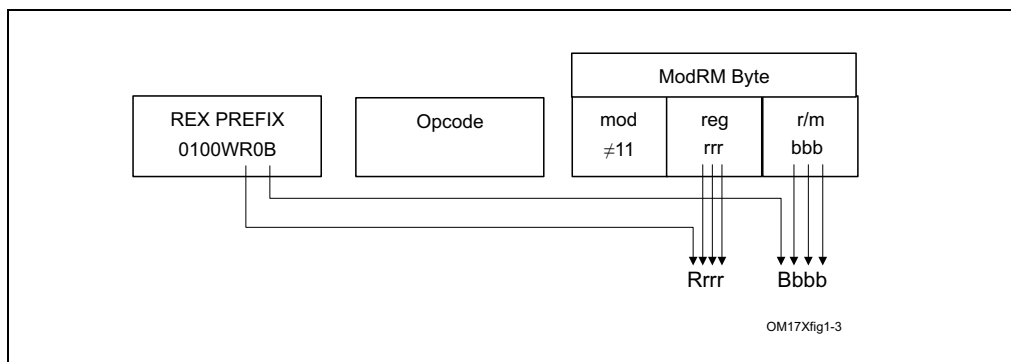
See Table 2-4 for a summary of the REX prefix format. Figure 2-4 through Figure 2-7 show examples of REX prefix fields in use. Some combinations of REX prefix fields are invalid. In such cases, the prefix is ignored. Some additional information follows:

- Setting REX.W can be used to determine the operand size but does not solely determine operand width. Like the 66H size prefix, 64-bit operand size override has no effect on byte-specific operations.
- For non-byte operations: if a 66H prefix is used with prefix (REX.W = 1), 66H is ignored.
- If a 66H override is used with REX and REX.W = 0, the operand size is 16 bits.

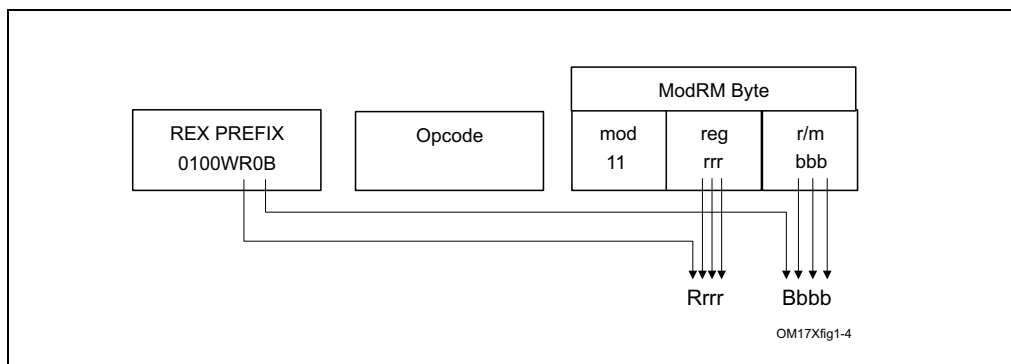
- REX.R modifies the ModR/M reg field when that field encodes a GPR, SSE, control or debug register. REX.R is ignored when ModR/M specifies other registers or defines an extended opcode.
- REX.X bit modifies the SIB index field.
- REX.B either modifies the base in the ModR/M r/m field or SIB base field; or it modifies the opcode reg field used for accessing GPRs.

**Table 2-4. REX Prefix Fields [BITS: 0100WRXB]**

Field Name	Bit Position	Definition
-	7:4	0100
W	3	0 = Operand size determined by CS.D 1 = 64 Bit Operand Size
R	2	Extension of the ModR/M reg field
X	1	Extension of the SIB index field
B	0	Extension of the ModR/M r/m field, SIB base field, or Opcode reg field



**Figure 2-4. Memory Addressing Without an SIB Byte; REX.X Not Used**



**Figure 2-5. Register-Register Addressing (No Memory Operand); REX.X Not Used**

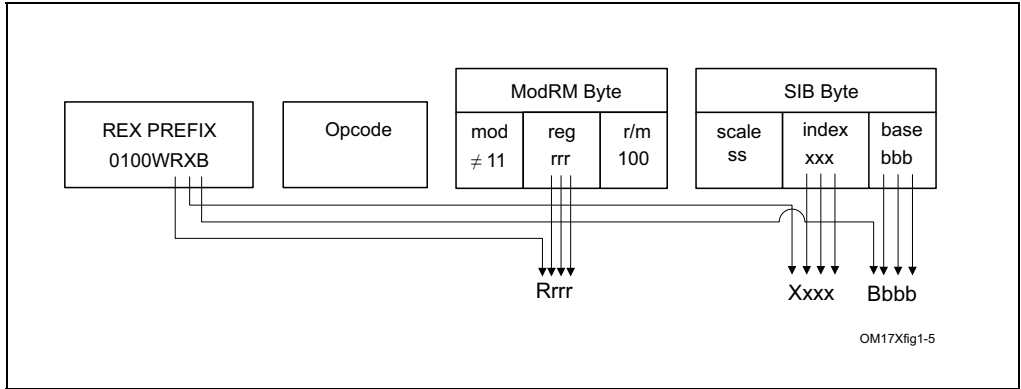


Figure 2-6. Memory Addressing With a SIB Byte

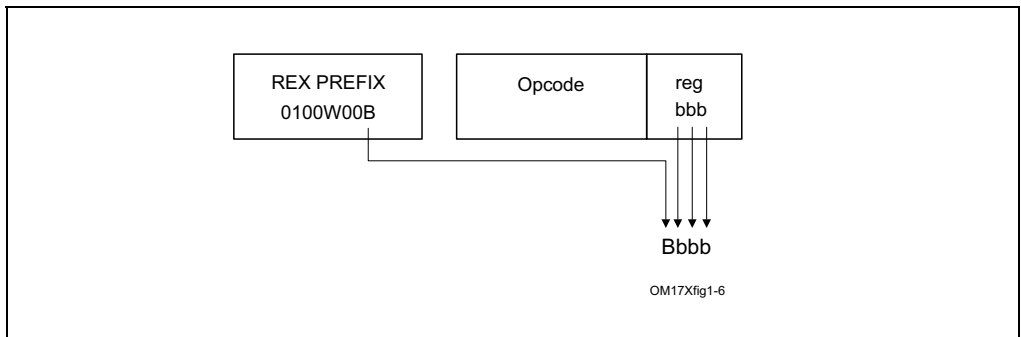


Figure 2-7. Register Operand Coded in Opcode Byte; REX.X & REX.R Not Used

In the IA-32 architecture, byte registers (AH, AL, BH, BL, CH, CL, DH, and DL) are encoded in the ModR/M byte's reg field, the r/m field or the opcode reg field as registers 0 through 7. REX prefixes provide an additional addressing capability for byte-registers that makes the least-significant byte of GPRs available for byte operations. Certain combinations of the fields of the ModR/M byte and the SIB byte have special meaning for register encodings. For some combinations, fields expanded by the REX prefix are not decoded. Table 2-5 describes how each case behaves.

Table 2-5. Special Cases of REX Encodings

ModR/M or SIB	Sub-field Encodings	Compatibility Mode Operation	Compatibility Mode Implications	Additional Implications
ModR/M Byte	mod ≠ 11 r/m = b*100(ESP)	SIB byte present.	SIB byte required for ESP-based addressing.	REX prefix adds a fourth bit (b) which is not decoded (don't care). SIB byte also required for R12-based addressing.
ModR/M Byte	mod = 0 r/m = b*101(EBP)	Base register not used.	EBP without a displacement must be done using mod = 01 with displacement of 0.	REX prefix adds a fourth bit (b) which is not decoded (don't care). Using RBP or R13 without displacement must be done using mod = 01 with a displacement of 0.
SIB Byte	index = 0100(ESP)	Index register not used.	ESP cannot be used as an index register.	REX prefix adds a fourth bit (b) which is decoded. There are no additional implications. The expanded index field allows distinguishing RSP from R12, therefore R12 can be used as an index.
SIB Byte	base = 0101(EBP)	Base register is unused if mod = 0.	Base register depends on mod encoding.	REX prefix adds a fourth bit (b) which is not decoded. This requires explicit displacement to be used with EBP/RBP or R13.

**NOTES:**

\* Don't care about value of REX.B

### 2.2.1.3 Displacement

Addressing in 64-bit mode uses existing 32-bit ModR/M and SIB encodings. The ModR/M and SIB displacement sizes do not change. They remain 8 bits or 32 bits and are sign-extended to 64 bits.

### 2.2.1.4 Direct Memory-Offset MOVs

In 64-bit mode, direct memory-offset forms of the MOV instruction are extended to specify a 64-bit immediate absolute address. This address is called a moffset. No prefix is needed to specify this 64-bit memory offset. For these MOV instructions, the size of the memory offset follows the address-size default (64 bits in 64-bit mode). See Table 2-6.

Table 2-6. Direct Memory Offset Form of MOV

Opcode	Instruction
A0	MOV AL, moffset
A1	MOV EAX, moffset
A2	MOV moffset, AL
A3	MOV moffset, EAX

### 2.2.1.5 Immediates

In 64-bit mode, the typical size of immediate operands remains 32 bits. When the operand size is 64 bits, the processor sign-extends all immediates to 64 bits prior to their use.

Support for 64-bit immediate operands is accomplished by expanding the semantics of the existing move (MOV reg, imm16/32) instructions. These instructions (opcodes B8H – BFH) move 16-bits or 32-bits of immediate data (depending on the effective operand size) into a GPR. When the effective operand size is 64 bits, these instructions can be used to load an immediate into a GPR. A REX prefix is needed to override the 32-bit default operand size to a 64-bit operand size.

For example:

```
48 B8 8877665544332211 MOV RAX,1122334455667788H
```

### 2.2.1.6 RIP-Relative Addressing

A new addressing form, RIP-relative (relative instruction-pointer) addressing, is implemented in 64-bit mode. An effective address is formed by adding displacement to the 64-bit RIP of the next instruction.

In IA-32 architecture and compatibility mode, addressing relative to the instruction pointer is available only with control-transfer instructions. In 64-bit mode, instructions that use ModR/M addressing can use RIP-relative addressing. Without RIP-relative addressing, all ModR/M modes address memory relative to zero.

RIP-relative addressing allows specific ModR/M modes to address memory relative to the 64-bit RIP using a signed 32-bit displacement. This provides an offset range of  $\pm 2\text{GB}$  from the RIP. Table 2-7 shows the ModR/M and SIB encodings for RIP-relative addressing. Redundant forms of 32-bit displacement-addressing exist in the current ModR/M and SIB encodings. There is one ModR/M encoding and there are several SIB encodings. RIP-relative addressing is encoded using a redundant form.

In 64-bit mode, the ModR/M Disp32 (32-bit displacement) encoding is re-defined to be RIP+Disp32 rather than displacement-only. See Table 2-7.

**Table 2-7. RIP-Relative Addressing**

ModR/M and SIB Sub-field Encodings		Compatibility Mode Operation	64-bit Mode Operation	Additional Implications in 64-bit mode
ModR/M Byte	mod = 00	Disp32	RIP + Disp32	Must use SIB form with normal (zero-based) displacement addressing
	r/m = 101 (none)			
SIB Byte	base = 101 (none)	if mod = 00, Disp32	Same as legacy	None
	index = 100 (none)			
	scale = 0, 1, 2, 4			

The ModR/M encoding for RIP-relative addressing does not depend on using a prefix. Specifically, the r/m bit field encoding of 101B (used to select RIP-relative addressing) is not affected by the REX prefix. For example, selecting R13 (REX.B = 1, r/m = 101B) with mod = 00B still results in RIP-relative addressing. The 4-bit r/m field of REX.B combined with ModR/M is not fully decoded. In order to address R13 with no displacement, software must encode R13 + 0 using a 1-byte displacement of zero.

RIP-relative addressing is enabled by 64-bit mode, not by a 64-bit address-size. The use of the address-size prefix does not disable RIP-relative addressing. The effect of the address-size prefix is to truncate and zero-extend the computed effective address to 32 bits.

### 2.2.1.7 Default 64-Bit Operand Size

In 64-bit mode, two groups of instructions have a default operand size of 64 bits (do not need a REX prefix for this operand size). These are:

- Near branches.
- All instructions, except far branches, that implicitly reference the RSP.

## 2.2.2 Additional Encodings for Control and Debug Registers

In 64-bit mode, more encodings for control and debug registers are available. The REX.R bit is used to modify the ModR/M reg field when that field encodes a control or debug register (see Table 2-4). These encodings enable the processor to address CR8-CR15 and DR8-DR15. An additional control register (CR8) is defined in 64-bit mode. CR8 becomes the Task Priority Register (TPR).

In the first implementation of IA-32e mode, CR9-CR15 and DR8-DR15 are not implemented. Any attempt to access unimplemented registers results in an invalid-opcode exception (#UD).

## 2.3 INTEL® ADVANCED VECTOR EXTENSIONS (INTEL® AVX)

Intel AVX instructions are encoded using an encoding scheme that combines prefix bytes, opcode extension field, operand encoding fields, and vector length encoding capability into a new prefix, referred to as VEX. In the VEX encoding scheme, the VEX prefix may be two or three bytes long, depending on the instruction semantics. Despite the two-byte or three-byte length of the VEX prefix, the VEX encoding format provides a more compact representation/packing of the components of encoding an instruction in Intel 64 architecture. The VEX encoding scheme also allows more headroom for future growth of Intel 64 architecture.

### 2.3.1 Instruction Format

Instruction encoding using VEX prefix provides several advantages:

- Instruction syntax support for three operands and up-to four operands when necessary. For example, the third source register used by VBLENDVPD is encoded using bits 7:4 of the immediate byte.
- Encoding support for vector length of 128 bits (using XMM registers) and 256 bits (using YMM registers).
- Encoding support for instruction syntax of non-destructive source operands.
- Elimination of escape opcode byte (0FH), SIMD prefix byte (66H, F2H, F3H) via a compact bit field representation within the VEX prefix.
- Elimination of the need to use REX prefix to encode the extended half of general-purpose register sets (R8-R15) for direct register access, memory addressing, or accessing XMM8-XMM15 (including YMM8-YMM15).
- Flexible and more compact bit fields are provided in the VEX prefix to retain the full functionality provided by REX prefix. REX.W, REX.X, REX.B functionalities are provided in the three-byte VEX prefix only because only a subset of SIMD instructions need them.
- Extensibility for future instruction extensions without significant instruction length increase.

Figure 2-8 shows the Intel 64 instruction encoding format with VEX prefix support. Legacy instruction without a VEX prefix is fully supported and unchanged. The use of VEX prefix in an Intel 64 instruction is optional, but a VEX prefix is required for Intel 64 instructions that operate on YMM registers or support three and four operand syntax. VEX prefix is not a constant-valued, “single-purpose” byte like 0FH, 66H, F2H, F3H in legacy SSE instructions. VEX prefix provides substantially richer capability than the REX prefix.

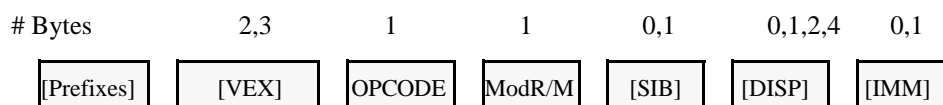


Figure 2-8. Instruction Encoding Format with VEX Prefix

### 2.3.2 VEX and the LOCK prefix

Any VEX-encoded instruction with a LOCK prefix preceding VEX will #UD.

### 2.3.3 VEX and the 66H, F2H, and F3H prefixes

Any VEX-encoded instruction with a 66H, F2H, or F3H prefix preceding VEX will #UD.

### 2.3.4 VEX and the REX prefix

Any VEX-encoded instruction with a REX prefix preceding VEX will #UD.



### 2.3.5 The VEX Prefix

The VEX prefix is encoded in either the two-byte form (the first byte must be C5H) or in the three-byte form (the first byte must be C4H). The two-byte VEX is used mainly for 128-bit, scalar, and the most common 256-bit AVX instructions; while the three-byte VEX provides a compact replacement of REX and 3-byte opcode instructions (including AVX and FMA instructions). Beyond the first byte of the VEX prefix, it consists of a number of bit fields providing specific capability, they are shown in Figure 2-9.

The bit fields of the VEX prefix can be summarized by its functional purposes:

- Non-destructive source register encoding (applicable to three and four operand syntax): This is the first source operand in the instruction syntax. It is represented by the notation, VEX.vvvv. This field is encoded using 1's complement form (inverted form), i.e. XMM0/YMM0/R0 is encoded as 1111B, XMM15/YMM15/R15 is encoded as 0000B.
- Vector length encoding: This 1-bit field represented by the notation VEX.L. L= 0 means vector length is 128 bits wide, L=1 means 256 bit vector. The value of this field is written as VEX.128 or VEX.256 in this document to distinguish encoded values of other VEX bit fields.
- REX prefix functionality: Full REX prefix functionality is provided in the three-byte form of VEX prefix. However the VEX bit fields providing REX functionality are encoded using 1's complement form, i.e. XMM0/YMM0/R0 is encoded as 1111B, XMM15/YMM15/R15 is encoded as 0000B.
  - Two-byte form of the VEX prefix only provides the equivalent functionality of REX.R, using 1's complement encoding. This is represented as VEX.R.
  - Three-byte form of the VEX prefix provides REX.R, REX.X, REX.B functionality using 1's complement encoding and three dedicated bit fields represented as VEX.R, VEX.X, VEX.B.
  - Three-byte form of the VEX prefix provides the functionality of REX.W only to specific instructions that need to override default 32-bit operand size for a general purpose register to 64-bit size in 64-bit mode. For those applicable instructions, VEX.W field provides the same functionality as REX.W. VEX.W field can provide completely different functionality for other instructions.

Consequently, the use of REX prefix with VEX encoded instructions is not allowed. However, the intent of the REX prefix for expanding register set is reserved for future instruction set extensions using VEX prefix encoding format.

- Compaction of SIMD prefix: Legacy SSE instructions effectively use SIMD prefixes (66H, F2H, F3H) as an opcode extension field. VEX prefix encoding allows the functional capability of such legacy SSE instructions (operating on XMM registers, bits 255:128 of corresponding YMM unmodified) to be encoded using the VEX.pp field without the presence of any SIMD prefix. The VEX-encoded 128-bit instruction will zero-out bits 255:128 of the destination register. VEX-encoded instruction may have 128 bit vector length or 256 bits length.
- Compaction of two-byte and three-byte opcode: More recently introduced legacy SSE instructions employ two and three-byte opcode. The one or two leading bytes are: 0FH, and 0FH 3AH/0FH 38H. The one-byte escape (0FH) and two-byte escape (0FH 3AH, 0FH 38H) can also be interpreted as an opcode extension field. The VEX.mmmmm field provides compaction to allow many legacy instruction to be encoded without the constant byte sequence, 0FH, 0FH 3AH, 0FH 38H. These VEX-encoded instruction may have 128 bit vector length or 256 bits length.

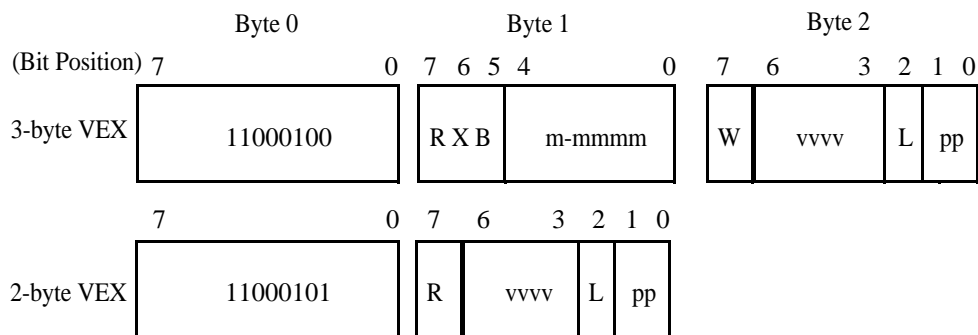
The VEX prefix is required to be the last prefix and immediately precedes the opcode bytes. It must follow any other prefixes. If VEX prefix is present a REX prefix is not supported.

The 3-byte VEX leaves room for future expansion with 3 reserved bits. REX and the 66h/F2h/F3h prefixes are reclaimed for future use.

VEX prefix has a two-byte form and a three byte form. If an instruction syntax can be encoded using the two-byte form, it can also be encoded using the three byte form of VEX. The latter increases the length of the instruction by one byte. This may be helpful in some situations for code alignment.

The VEX prefix supports 256-bit versions of floating-point SSE, SSE2, SSE3, and SSE4 instructions. Note, certain new instruction functionality can only be encoded with the VEX prefix.

The VEX prefix will #UD on any instruction containing MMX register sources or destinations.



R: REX.R in 1's complement (inverted) form  
 1: Same as REX.R=0 (must be 1 in 32-bit mode)  
 0: Same as REX.R=1 (64-bit mode only)

X: REX.X in 1's complement (inverted) form  
 1: Same as REX.X=0 (must be 1 in 32-bit mode)  
 0: Same as REX.X=1 (64-bit mode only)

B: REX.B in 1's complement (inverted) form  
 1: Same as REX.B=0 (Ignored in 32-bit mode).  
 0: Same as REX.B=1 (64-bit mode only)

W: opcode specific (use like REX.W, or used for opcode extension, or ignored, depending on the opcode byte)

m-mmmm:

00000: Reserved for future use (will #UD)  
 00001: implied 0F leading opcode byte  
 00010: implied 0F 38 leading opcode bytes  
 00011: implied 0F 3A leading opcode bytes  
 00100-11111: Reserved for future use (will #UD)

vvvv: a register specifier (in 1's complement form) or 1111 if unused.

L: Vector Length

0: scalar or 128-bit vector  
 1: 256-bit vector

pp: opcode extension providing equivalent functionality of a SIMD prefix

00: None  
 01: 66  
 10: F3  
 11: F2

**Figure 2-9. VEX bit fields**

The following subsections describe the various fields in two or three-byte VEX prefix.

### 2.3.5.1 VEX Byte 0, bits[7:0]

VEX Byte 0, bits [7:0] must contain the value 11000101b (C5h) or 11000100b (C4h). The 3-byte VEX uses the C4h first byte, while the 2-byte VEX uses the C5h first byte.

### 2.3.5.2 VEX Byte 1, bit [7] - 'R'

VEX Byte 1, bit [7] contains a bit analogous to a bit inverted REX.R. In protected and compatibility modes the bit must be set to '1' otherwise the instruction is LES or LDS.

This bit is present in both 2- and 3-byte VEX prefixes.

The usage of WRXB bits for legacy instructions is explained in detail section 2.2.1.2 of Intel 64 and IA-32 Architectures Software developer's manual, Volume 2A.

This bit is stored in bit inverted format.

### 2.3.5.3 3-byte VEX byte 1, bit[6] - 'X'

Bit[6] of the 3-byte VEX byte 1 encodes a bit analogous to a bit inverted REX.X. It is an extension of the SIB Index field in 64-bit modes. In 32-bit modes, this bit must be set to '1' otherwise the instruction is LES or LDS.

This bit is available only in the 3-byte VEX prefix.

This bit is stored in bit inverted format.

### 2.3.5.4 3-byte VEX byte 1, bit[5] - 'B'

Bit[5] of the 3-byte VEX byte 1 encodes a bit analogous to a bit inverted REX.B. In 64-bit modes, it is an extension of the ModR/M r/m field, or the SIB base field. In 32-bit modes, this bit is ignored.

This bit is available only in the 3-byte VEX prefix.

This bit is stored in bit inverted format.

### 2.3.5.5 3-byte VEX byte 2, bit[7] - 'W'

Bit[7] of the 3-byte VEX byte 2 is represented by the notation VEX.W. It can provide following functions, depending on the specific opcode.

- For AVX instructions that have equivalent legacy SSE instructions (typically these SSE instructions have a general-purpose register operand with its operand size attribute promotable by REX.W), if REX.W promotes the operand size attribute of the general-purpose register operand in legacy SSE instruction, VEX.W has same meaning in the corresponding AVX equivalent form. In 32-bit modes, VEX.W is silently ignored.
- For AVX instructions that have equivalent legacy SSE instructions (typically these SSE instructions have operands with their operand size attribute fixed and not promotable by REX.W), if REX.W is don't care in legacy SSE instruction, VEX.W is ignored in the corresponding AVX equivalent form irrespective of mode.
- For new AVX instructions where VEX.W has no defined function (typically these meant the combination of the opcode byte and VEX.mmmmm did not have any equivalent SSE functions), VEX.W is reserved as zero and setting to other than zero will cause instruction to #UD.

### 2.3.5.6 2-byte VEX Byte 1, bits[6:3] and 3-byte VEX Byte 2, bits [6:3]- 'vvvv' the Source or Dest Register Specifier

In 32-bit mode the VEX first byte C4 and C5 alias onto the LES and LDS instructions. To maintain compatibility with existing programs the VEX 2nd byte, bits [7:6] must be 11b. To achieve this, the VEX payload bits are selected to place only inverted, 64-bit valid fields (extended register selectors) in these upper bits.

The 2-byte VEX Byte 1, bits [6:3] and the 3-byte VEX, Byte 2, bits [6:3] encode a field (shorthand VEX.vvvv) that for instructions with 2 or more source registers and an XMM or YMM or memory destination encodes the first source register specifier stored in inverted (1's complement) form.

VEX.vvvv is not used by the instructions with one source (except certain shifts, see below) or on instructions with no XMM or YMM or memory destination. If an instruction does not use VEX.vvvv then it should be set to 1111b otherwise instruction will #UD.

In 64-bit mode all 4 bits may be used. See Table 2-8 for the encoding of the XMM or YMM registers. In 32-bit and 16-bit modes bit 6 must be 1 (if bit 6 is not 1, the 2-byte VEX version will generate LDS instruction and the 3-byte VEX version will ignore this bit).

**Table 2-8. VEX.vvvv to register name mapping**

VEX.vvvv	Dest Register	Valid in Legacy/Compatibility 32-bit modes?
1111B	XMM0/YMM0	Valid
1110B	XMM1/YMM1	Valid
1101B	XMM2/YMM2	Valid
1100B	XMM3/YMM3	Valid
1011B	XMM4/YMM4	Valid
1010B	XMM5/YMM5	Valid
1001B	XMM6/YMM6	Valid
1000B	XMM7/YMM7	Valid
0111B	XMM8/YMM8	Invalid
0110B	XMM9/YMM9	Invalid
0101B	XMM10/YMM10	Invalid
0100B	XMM11/YMM11	Invalid
0011B	XMM12/YMM12	Invalid
0010B	XMM13/YMM13	Invalid
0001B	XMM14/YMM14	Invalid
0000B	XMM15/YMM15	Invalid

The VEX.vvvv field is encoded in bit inverted format for accessing a register operand.

### 2.3.6 Instruction Operand Encoding and VEX.vvvv, ModR/M

VEX-encoded instructions support three-operand and four-operand instruction syntax. Some VEX-encoded instructions have syntax with less than three operands, e.g. VEX-encoded pack shift instructions support one source operand and one destination operand).

The roles of VEX.vvvv, reg field of ModR/M byte (ModR/M.reg), r/m field of ModR/M byte (ModR/M.r/m) with respect to encoding destination and source operands vary with different type of instruction syntax.

The role of VEX.vvvv can be summarized to three situations:

- VEX.vvvv encodes the first source register operand, specified in inverted (1's complement) form and is valid for instructions with 2 or more source operands.
- VEX.vvvv encodes the destination register operand, specified in 1's complement form for certain vector shifts. The instructions where VEX.vvvv is used as a destination are listed in Table 2-9. The notation in the "Opcode" column in Table 2-9 is described in detail in section 3.1.1.
- VEX.vvvv does not encode any operand, the field is reserved and should contain 1111b.

**Table 2-9. Instructions with a VEX.vvvv destination**

Opcode	Instruction mnemonic
VEX.NDD.128.66.0F 73 /7 ib	VPSLLDQ xmm1, xmm2, imm8
VEX.NDD.128.66.0F 73 /3 ib	VPSRLDQ xmm1, xmm2, imm8
VEX.NDD.128.66.0F 71 /2 ib	VPSRLW xmm1, xmm2, imm8
VEX.NDD.128.66.0F 72 /2 ib	VPSRLD xmm1, xmm2, imm8
VEX.NDD.128.66.0F 73 /2 ib	VPSRLQ xmm1, xmm2, imm8
VEX.NDD.128.66.0F 71 /4 ib	VPSRAW xmm1, xmm2, imm8
VEX.NDD.128.66.0F 72 /4 ib	VPSRAD xmm1, xmm2, imm8
VEX.NDD.128.66.0F 71 /6 ib	VPSLLW xmm1, xmm2, imm8
VEX.NDD.128.66.0F 72 /6 ib	VPSLLD xmm1, xmm2, imm8
VEX.NDD.128.66.0F 73 /6 ib	VPSLLQ xmm1, xmm2, imm8

The role of ModR/M.r/m field can be summarized to two situations:

- ModR/M.r/m encodes the instruction operand that references a memory address.
- For some instructions that do not support memory addressing semantics, ModR/M.r/m encodes either the destination register operand or a source register operand.

The role of ModR/M.reg field can be summarized to two situations:

- ModR/M.reg encodes either the destination register operand or a source register operand.
- For some instructions, ModR/M.reg is treated as an opcode extension and not used to encode any instruction operand.

For instruction syntax that support four operands, VEX.vvvv, ModR/M.r/m, ModR/M.reg encodes three of the four operands. The role of bits 7:4 of the immediate byte serves the following situation:

- Imm8[7:4] encodes the third source register operand.

### 2.3.6.1 3-byte VEX byte 1, bits[4:0] - “m-mmmm”

Bits[4:0] of the 3-byte VEX byte 1 encode an implied leading opcode byte (0F, 0F 38, or 0F 3A). Several bits are reserved for future use and will #UD unless 0.

**Table 2-10. VEX.m-mmmm interpretation**

VEX.m-mmmm	Implied Leading Opcode Bytes
00000B	Reserved
00001B	0F
00010B	0F 38
00011B	0F 3A
00100-11111B	Reserved
(2-byte VEX)	0F

VEX.m-mmmm is only available on the 3-byte VEX. The 2-byte VEX implies a leading 0Fh opcode byte.

### 2.3.6.2 2-byte VEX byte 1, bit[2], and 3-byte VEX byte 2, bit [2]- “L”

The vector length field, VEX.L, is encoded in bit[2] of either the second byte of 2-byte VEX, or the third byte of 3-byte VEX. If “VEX.L = 1”, it indicates 256-bit vector operation. “VEX.L = 0” indicates scalar and 128-bit vector operations.

The instruction VZEROUPPER is a special case that is encoded with VEX.L = 0, although its operation zero’s bits 255:128 of all YMM registers accessible in the current operating mode.

See the following table.

**Table 2-11. VEX.L interpretation**

VEX.L	Vector Length
0	128-bit (or 32/64-bit scalar)
1	256-bit

### 2.3.6.3 2-byte VEX byte 1, bits[1:0], and 3-byte VEX byte 2, bits [1:0]- “pp”

Up to one implied prefix is encoded by bits[1:0] of either the 2-byte VEX byte 1 or the 3-byte VEX byte 2. The prefix behaves as if it was encoded prior to VEX, but after all other encoded prefixes.

See the following table.

Table 2-12. VEX.pp interpretation

pp	Implies this prefix after other prefixes but before VEX
00B	None
01B	66
10B	F3
11B	F2

### 2.3.7 The Opcode Byte

One (and only one) opcode byte follows the 2 or 3 byte VEX. Legal opcodes are specified in Appendix B, in color. Any instruction that uses illegal opcode will #UD.

### 2.3.8 The MODRM, SIB, and Displacement Bytes

The encodings are unchanged but the interpretation of reg\_field or rm\_field differs (see above).

### 2.3.9 The Third Source Operand (Immediate Byte)

VEX-encoded instructions can support instruction with a four operand syntax. VBLENDVPD, VBLENDVPS, and PBLENDVB use imm8[7:4] to encode one of the source registers.

### 2.3.10 AVX Instructions and the Upper 128-bits of YMM registers

If an instruction with a destination XMM register is encoded with a VEX prefix, the processor zeroes the upper bits (above bit 128) of the equivalent YMM register. Legacy SSE instructions without VEX preserve the upper bits.

#### 2.3.10.1 Vector Length Transition and Programming Considerations

An instruction encoded with a VEX.128 prefix that loads a YMM register operand operates as follows:

- Data is loaded into bits 127:0 of the register
- Bits above bit 127 in the register are cleared.

Thus, such an instruction clears bits 255:128 of a destination YMM register on processors with a maximum vector-register width of 256 bits. In the event that future processors extend the vector registers to greater widths, an instruction encoded with a VEX.128 or VEX.256 prefix will also clear any bits beyond bit 255. (This is in contrast with legacy SSE instructions, which have no VEX prefix; these modify only bits 127:0 of any destination register operand.)

Programmers should bear in mind that instructions encoded with VEX.128 and VEX.256 prefixes will clear any future extensions to the vector registers. A calling function that uses such extensions should save their state before calling legacy functions. This is not possible for involuntary calls (e.g., into an interrupt-service routine). It is recommended that software handling involuntary calls accommodate this by not executing instructions encoded with VEX.128 and VEX.256 prefixes. In the event that it is not possible or desirable to restrict these instructions, then software must take special care to avoid actions that would, on future processors, zero the upper bits of vector registers.

Processors that support further vector-register extensions (defining bits beyond bit 255) will also extend the XSAVE and XRSTOR instructions to save and restore these extensions. To ensure forward compatibility, software that handles involuntary calls and that uses instructions encoded with VEX.128 and VEX.256 prefixes should first save and then restore the vector registers (with any extensions) using the XSAVE and XRSTOR instructions with save/restore masks that set bits that correspond to all vector-register extensions. Ideally, software should rely on a mechanism that is cognizant of which bits to set. (E.g., an OS mechanism that sets the save/restore mask bits for all vector-register extensions that are enabled in XCR0.) Saving and restoring state with instructions other than XSAVE and XRSTOR will, on future processors with wider vector registers, corrupt the extended state of the vector registers - even if doing so functions correctly on processors supporting 256-bit vector registers. (The same is true

if XSAVE and XRSTOR are used with a save/restore mask that does not set bits corresponding to all supported extensions to the vector registers.)

### 2.3.11 AVX Instruction Length

The AVX instructions described in this document (including VEX and ignoring other prefixes) do not exceed 11 bytes in length, but may increase in the future. The maximum length of an Intel 64 and IA-32 instruction remains 15 bytes.

### 2.3.12 Vector SIB (VSIB) Memory Addressing

In Intel® Advanced Vector Extensions 2 (Intel® AVX2), an SIB byte that follows the ModR/M byte can support VSIB memory addressing to an array of linear addresses. VSIB addressing is only supported in a subset of Intel AVX2 instructions. VSIB memory addressing requires 32-bit or 64-bit effective address. In 32-bit mode, VSIB addressing is not supported when address size attribute is overridden to 16 bits. In 16-bit protected mode, VSIB memory addressing is permitted if address size attribute is overridden to 32 bits. Additionally, VSIB memory addressing is supported only with VEX prefix.

In VSIB memory addressing, the SIB byte consists of:

- The scale field (bit 7:6) specifies the scale factor.
- The index field (bits 5:3) specifies the register number of the vector index register, each element in the vector register specifies an index.
- The base field (bits 2:0) specifies the register number of the base register.

Table 2-3 shows the 32-bit VSIB addressing form. It is organized to give 256 possible values of the SIB byte (in hexadecimal). General purpose registers used as a base are indicated across the top of the table, along with corresponding values for the SIB byte's base field. The register names also include R8L-R15L applicable only in 64-bit mode (when address size override prefix is used, but the value of VEX.B is not shown in Table 2-3). In 32-bit mode, R8L-R15L does not apply.

Table rows in the body of the table indicate the vector index register used as the index field and each supported scaling factor shown separately. Vector registers used in the index field can be XMM or YMM registers. The left-most column includes vector registers VR8-VR15 (i.e. XMM8/YMM8-XMM15/YMM15), which are only available in 64-bit mode and does not apply if encoding in 32-bit mode.

**Table 2-13. 32-Bit VSIB Addressing Forms of the SIB Byte**

r32			EAX/ R8L	ECX/ R9L	EDX/ R10L	EBX/ R11L	ESP/ R12L	EBP/ R13L <sup>1</sup>	ESI/ R14L	EDI/ R15L	
(In decimal) Base =			0	1	2	3	4	5	6	7	
(In binary) Base =			000	001	010	011	100	101	110	111	
Scaled Index	SS	Index	Value of SIB Byte (in Hexadecimal)								
VR0/VR8	*1	00	000	00	01	02	03	04	05	06	07
VR1/VR9			001	08	09	0A	0B	0C	0D	0E	0F
VR2/VR10			010	10	11	12	13	14	15	16	17
VR3/VR11			011	18	19	1A	1B	1C	1D	1E	1F
VR4/VR12			100	20	21	22	23	24	25	26	27
VR5/VR13			101	28	29	2A	2B	2C	2D	2E	2F
VR6/VR14			110	30	31	32	33	34	35	36	37
VR7/VR15			111	38	39	3A	3B	3C	3D	3E	3F
VR0/VR8	*2	01	000	40	41	42	43	44	45	46	47
VR1/VR9			001	48	49	4A	4B	4C	4D	4E	4F
VR2/VR10			010	50	51	52	53	54	55	56	57
VR3/VR11			011	58	59	5A	5B	5C	5D	5E	5F
VR4/VR12			100	60	61	62	63	64	65	66	67
VR5/VR13			101	68	69	6A	6B	6C	6D	6E	6F
VR6/VR14			110	70	71	72	73	74	75	76	77
VR7/VR15			111	78	79	7A	7B	7C	7D	7E	7F

Table 2-13. 32-Bit VSIB Addressing Forms of the SIB Byte (Contd.)

VR0/VR8	*4	10	000	80	81	82	83	84	85	86	87
VR1/VR9			001	88	89	8A	8B	8C	8D	8E	8F
VR2/VR10			010	90	91	92	93	94	95	96	97
VR3/VR11			011	98	99	9A	9B	9C	9D	9E	9F
VR4/VR12			100	A0	A1	A2	A3	A4	A5	A6	A7
VR5/VR13			101	A8	A9	AA	AB	AC	AD	AE	AF
VR6/VR14			110	B0	B1	B2	B3	B4	B5	B6	B7
VR7/VR15			111	B8	B9	BA	BB	BC	BD	BE	BF
VR0/VR8	*8	11	000	C0	C1	C2	C3	C4	C5	C6	C7
VR1/VR9			001	C8	C9	CA	CB	CC	CD	CE	CF
VR2/VR10			010	D0	D1	D2	D3	D4	D5	D6	D7
VR3/VR11			011	D8	D9	DA	DB	DC	DD	DE	DF
VR4/VR12			100	E0	E1	E2	E3	E4	E5	E6	E7
VR5/VR13			101	E8	E9	EA	EB	EC	ED	EE	EF
VR6/VR14			110	F0	F1	F2	F3	F4	F5	F6	F7
VR7/VR15			111	F8	F9	FA	FB	FC	FD	FE	FF

**NOTES:**

1. If ModR/M.mod = 00b, the base address is zero, then effective address is computed as [scaled vector index] + disp32. Otherwise the base address is computed as [EBP/R13]+ disp, the displacement is either 8 bit or 32 bit depending on the value of ModR/M.mod:

MOD	Effective Address
00b	[Scaled Vector Register] + Disp32
01b	[Scaled Vector Register] + Disp8 + [EBP/R13]
10b	[Scaled Vector Register] + Disp32 + [EBP/R13]

**2.3.12.1 64-bit Mode VSIB Memory Addressing**

In 64-bit mode VSIB memory addressing uses the VEX.B field and the base field of the SIB byte to encode one of the 16 general-purpose register as the base register. The VEX.X field and the index field of the SIB byte encode one of the 16 vector registers as the vector index register.

In 64-bit mode the top row of Table 2-13 base register should be interpreted as the full 64-bit of each register.

**2.4 AVX AND SSE INSTRUCTION EXCEPTION SPECIFICATION**

To look up the exceptions of legacy 128-bit SIMD instruction, 128-bit VEX-encoded instructions, and 256-bit VEX-encoded instruction, Table 2-14 summarizes the exception behavior into separate classes, with detailed exception conditions defined in sub-sections 2.4.1 through 2.5.1. For example, ADDPS contains the entry:

“See Exceptions Type 2”

In this entry, “Type2” can be looked up in Table 2-14.

The instruction’s corresponding CPUID feature flag can be identified in the fourth column of the Instruction summary table.

Note: #UD on CPUID feature flags=0 is not guaranteed in a virtualized environment if the hardware supports the feature flag.

**NOTE**

Instructions that operate only with MMX, X87, or general-purpose registers are not covered by the exception classes defined in this section. For instructions that operate on MMX registers, see Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B*.



**Table 2-14. Exception class description**

Exception Class	Instruction set	Mem arg	Floating-Point Exceptions (#XM)
Type 1	AVX, Legacy SSE	16/32 byte explicitly aligned	None
Type 2	AVX, Legacy SSE	16/32 byte not explicitly aligned	Yes
Type 3	AVX, Legacy SSE	< 16 byte	Yes
Type 4	AVX, Legacy SSE	16/32 byte not explicitly aligned	No
Type 5	AVX, Legacy SSE	< 16 byte	No
Type 6	AVX (no Legacy SSE)	Varies	(At present, none do)
Type 7	AVX, Legacy SSE	None	None
Type 8	AVX	None	None
Type 11	F16C	8 or 16 byte, Not explicitly aligned, no AC#	Yes
Type 12	AVX2	Not explicitly aligned, no AC#	No

See Table 2-15 for lists of instructions in each exception class.

Table 2-15. Instructions in each Exception Class

Exception Class	Instruction
Type 1	(V)MOVAPD, (V)MOVAPS, (V)MOVDDQA, (V)MOVNTDQ, (V)MOVNTDQA, (V)MOVNTPD, (V)MOVNTPS
Type 2	(V)ADDPD, (V)ADDPs, (V)ADDSUBPD, (V)ADDSUBPS, (V)CMPPD, (V)CMPPS, (V)CVTDQ2PS, (V)CVTPD2DQ, (V)CVTPD2PS, (V)CVTPS2DQ, (V)CVTTPD2DQ, (V)CVTTPS2DQ, (V)DIVPD, (V)DIVPS, (V)DPPD*, (V)DPPS*, (V)FMADD132PD, (V)FMADD213PD, (V)FMADD231PD, (V)FMADD132PS, (V)FMADD213PS, (V)FMADD231PS, (V)FMADDSUB132PD, (V)FMADDSUB213PD, (V)FMADDSUB231PD, (V)FMADDSUB132PS, (V)FMADDSUB213PS, (V)FMADDSUB231PS, (V)FMSUBADD132PD, (V)FMSUBADD213PD, (V)FMSUBADD231PD, (V)FMSUBADD132PS, (V)FMSUBADD213PS, (V)FMSUBADD231PS, (V)FMSUB132PD, (V)FMSUB213PD, (V)FMSUB231PD, (V)FMSUB132PS, (V)FMSUB213PS, (V)FMSUB231PS, (V)FNMADD132PD, (V)FNMADD213PD, (V)FNMADD231PD, (V)FNMADD132PS, (V)FNMADD213PS, (V)FNMADD231PS, (V)FNMMSUB132PD, (V)FNMMSUB213PD, (V)FNMMSUB231PD, (V)FNMMSUB132PS, (V)FNMMSUB213PS, (V)FNMMSUB231PS, (V)HADDPD, (V)HADDPs, (V)HSUBPD, (V)HSUBPS, (V)MAXPD, (V)MAXPS, (V)MINPD, (V)MINPS, (V)MULPD, (V)MULPS, (V)ROUNDPS, (V)SQRTPD, (V)SQRTPS, (V)SUBPD, (V)SUBPS
Type 3	(V)ADDS, (V)ADDSs, (V)CMPD, (V)CMPs, (V)COMISD, (V)COMISS, (V)CVTSD2PS, (V)CVTSD2SI, (V)CVTSD2SS, (V)CVTSD2SD, (V)CVTSD2SS, (V)CVTSS2SD, (V)CVTSS2SI, (V)CVTSS2SI, (V)CVTSS2SI, (V)DIVSD, (V)DIVSS, (V)FMADD132SD, (V)FMADD213SD, (V)FMADD231SD, (V)FMADD132SS, (V)FMADD213SS, (V)FMADD231SS, (V)FMSUB132SD, (V)FMSUB213SD, (V)FMSUB231SD, (V)FMSUB132SS, (V)FMSUB213SS, (V)FMSUB231SS, (V)FNMADD132SD, (V)FNMADD213SD, (V)FNMADD231SD, (V)FNMADD132SS, (V)FNMADD213SS, (V)FNMADD231SS, (V)FNMMSUB132SD, (V)FNMMSUB213SD, (V)FNMMSUB231SD, (V)FNMMSUB132SS, (V)FNMMSUB213SS, (V)FNMMSUB231SS, (V)MAXSD, (V)MAXSS, (V)MINS, (V)MINSs, (V)MULSD, (V)MULSS, (V)ROUNDSD, (V)ROUNDSS, (V)SQRTSD, (V)SQRTSS, (V)SUBSD, (V)SUBSS, (V)UCOMISD, (V)UCOMISS
Type 4	(V)AESDEC, (V)AESDECLAST, (V)AESENC, (V)AESENCLAST, (V)AESIMC, (V)AESKEYGENASSIST, (V)ANDPD, (V)ANDPS, (V)ANDNPD, (V)ANDNPS, (V)BLENDPD, (V)BLENDPS, (V)BLENDVPD, (V)BLENDVPS, (V)LDDQU***, (V)MASKMOVDQU, (V)PTEST, (V)TESTPS, (V)TESTPD, (V)MOVDQU*, (V)MOVSHDUP, (V)MOVSLDUP, (V)MOVUPD*, (V)MOVUPS*, (V)MPSADBW, (V)ORPD, (V)ORPS, (V)PABSB, (V)PABSW, (V)PABSD, (V)PACKSSWB, (V)PACKSSDW, (V)PACKUSWB, (V)PACKUSDW, (V)PADDB, (V)PADDW, (V)PADDD, (V)PADDQ, (V)PADDSB, (V)PADDSW, (V)PADDUSB, (V)PADDUSW, (V)PALIGNR, (V)PAND, (V)PANDN, (V)PAVGB, (V)PAVGW, (V)PBLENDVB, (V)PBLENDW, (V)PCMP(E/I)STRI/M***, (V)PCMPEQB, (V)PCMPEQW, (V)PCMPEQD, (V)PCMPEQQ, (V)PCMPGTB, (V)PCMPGTW, (V)PCMPGTD, (V)PCMPGTQ, (V)PCLMULQDQ, (V)PHADDW, (V)PHADD, (V)PHADDSW, (V)PHMINPOSUW, (V)PHSUBD, (V)PHSUBW, (V)PHSUBSW, (V)PMADDWD, (V)PMADDUBSW, (V)PMASXB, (V)PMASXW, (V)PMASXD, (V)PMASXUB, (V)PMASXUW, (V)PMASXUD, (V)PMINSB, (V)PMINSW, (V)PMINSD, (V)PMINUB, (V)PMINUW, (V)PMINUD, (V)PMULHUW, (V)PMULHRW, (V)PMULHW, (V)PMULLW, (V)PMULLD, (V)PMULUDQ, (V)PMULDQ, (V)POR, (V)PSADBW, (V)PSHUFB, (V)PSHUFD, (V)PSHUFW, (V)PSHUFLW, (V)PSIGNB, (V)PSIGNW, (V)PSIGND, (V)PSLLW, (V)PSLLD, (V)PSLLQ, (V)PSRAW, (V)PSRAD, (V)PSRLW, (V)PSRLD, (V)PSRLQ, (V)PSUBB, (V)PSUBW, (V)PSUBD, (V)PSUBQ, (V)PSUBSB, (V)PSUBSW, (V)PUNPCKHBW, (V)PUNPCKHWD, (V)PUNPCKHDQ, (V)PUNPCKHQDQ, (V)PUNPCKLBW, (V)PUNPCKLWD, (V)PUNPCKLDQ, (V)PUNPCKLQDQ, (V)PXOR, (V)RCPPS, (V)RSQRTPS, (V)SHUFPD, (V)SHUFPS, (V)UNPCKHPD, (V)UNPCKHPS, (V)UNPCKLPD, (V)UNPCKLPS, (V)XORPD, (V)XORPS, (V)BLEND, (V)PERMD, (V)PERMPS, (V)PERMPD, (V)PERMQ, (V)PSLLVD, (V)PSLLVQ, (V)PSRAVD, (V)PSRLVD, (V)PSRLVQ, (V)PERMILPD, (V)PERMILPS, (V)PERM2F128
Type 5	(V)CVTDQ2PD, (V)EXTRACTPS, (V)INSERTPS, (V)MOVD, (V)MOVQ, (V)MOVDDUP, (V)MOVLPD, (V)MOVLPS, (V)MOVHPD, (V)MOVHPS, (V)MOVSD, (V)MOVSS, (V)PEXTRB, (V)PEXTRD, (V)PEXTRW, (V)PEXTRQ, (V)PINSRB, (V)PINSRD, (V)PINSRW, (V)PINSRQ, (V)RCPPS, (V)RSQRTSS, (V)PMOVSX/ZX, (V)LDMXCSR*, (V)STMXCSR
Type 6	VEXTRACTF128, VBROADCASTSS, VBROADCASTSD, VBROADCASTF128, VINSERTF128, VMASKMOVPS**, (V)MASKMOVPD**, (V)MASKMOVQ, (V)MASKMOVQ, VBROADCASTI128, VPBROADCASTB, VPBROADCASTD, VPBROADCASTW, VPBROADCASTQ, VEXTRACTI128, VINSERTI128, VPERM2I128
Type 7	(V)MOVLHPS, (V)MOVHLPS, (V)MOVMSKPD, (V)MOVMSKPS, (V)PMOVMASKB, (V)PSLLDQ, (V)PSRLDQ, (V)PSLLW, (V)PSLLD, (V)PSLLQ, (V)PSRAW, (V)PSRAD, (V)PSRLW, (V)PSRLD, (V)PSRLQ
Type 8	VZEROALL, VZERoupper
Type 11	VCVTPH2PS, VCVTPS2PH
Type 12	VGATHERDPS, VGATHERDPD, VGATHERQPS, VGATHERQPD, VPGATHERDD, VPGATHERDQ, VPGATHERQD, VPGATHERQQ

(\*) - Additional exception restrictions are present - see the Instruction description for details

(\*\*) - Instruction behavior on alignment check reporting with mask bits of less than all 1s are the same as with mask bits of all 1s, i.e. no alignment checks are performed.

(\*\*\*) - PCMPSTRI, PCMPSTRM, PCMPISTRI, PCMPISTRM and LDDQU instructions do not cause #GP if the memory operand is not aligned to 16-Byte boundary.

Table 2-15 classifies exception behaviors for AVX instructions. Within each class of exception conditions that are listed in Table 2-18 through Table 2-27, certain subsets of AVX instructions may be subject to #UD exception depending on the encoded value of the VEX.L field. Table 2-17 provides supplemental information of AVX instructions that may be subject to #UD exception if encoded with incorrect values in the VEX.W or VEX.L field.

**Table 2-16. #UD Exception and VEX.W=1 Encoding**

Exception Class	#UD If VEX.W = 1 in all modes	#UD If VEX.W = 1 in non-64-bit modes
Type 1		
Type 2		
Type 3		
Type 4	VBLENDVPD, VBLENDVPS, VPBLENDVB, VTESTPD, VTESTPS, VPBLEND, VPERMD, VPERMPS, VPERM2I128, VPSRAVD, VPERMILPD, VPERMILPS, VPERM2F128	
Type 5		VPEXTRQ, VPINSRQ,
Type 6	VEXTRACTF128, VBROADCASTSS, VBROADCASTSD, VBROADCASTF128, VINSERTF128, VMASKMOVPS, VMASKMOVPD, VBROADCASTI128, VPBROADCASTB/W/D, VEXTRACTI128, VINSERTI128	
Type 7		
Type 8		
Type 11	VCVTPH2PS, VCVTPS2PH	
Type 12		

Table 2-17. #UD Exception and VEX.L Field Encoding

Exception Class	#UD If VEX.L = 0	#UD If (VEX.L = 1 && AVX2 not present && AVX present)	#UD If (VEX.L = 1 && AVX2 present)
Type 1		VMOVNTDQA	
Type 2		VDPPD	VDPPD
Type 3			
Type 4		VMASKMOVDQU, VMPSADBW, VPABSB/W/D, VPACKSSWB/DW, VPACKUSWB/DW, VPADDB/W/D, VPADDQ, VPADDSB/W, VPADDUSB/W, VPALIGNR, VPAND, VPANDN, VPAVGB/W, VPBLENDVB, VPBLENDW, VPCMP(E/I)STRI/M, VPCMPEQB/W/D/Q, VPCMPGTB/W/D/Q, VPHADDW/D, VPHADDSW, VPHMINPOSUW, VPHSUBD/W, VPHSUBSW, VPMADDWD, VPMADDUBSW, VPMAXSB/W/D, VPMAXUB/W/D, VPMINSB/W/D, VPMINUB/W/D, VPMULHUW, VPMULHRW, VPMULHW/LW, VPMULLD, VPMULLDQ, VPMULDQ, VPOR, VPSADBW, VPSHUF/D, VPSHUFHW/LW, VPSIGNB/W/D, VPSLLW/D/Q, VPSRAW/D, VPSRLW/D/Q, VPSUBB/W/D/Q, VPSUBSB/W, VPUNPCKHBW/W/D/DQ, VPUNPCKHQDQ, VPUNPCKLBW/W/D/DQ, VPUNPCKLQDQ, VPXOR	VPCMP(E/I)STRI/M, PHMINPOSUW
Type 5		VEXTRACTPS, VINSERTPS, VMOVD, VMOVQ, VMOVLPD, VMOVLPS, VMOVHPD, VMOVHPS, VPEXTRB, VPEXTRD, VPEXTRW, VPEXTRQ, VPINSRB, VPINSRD, VPINSRW, VPINSRQ, VPMOVSX/ZX, VLDMXCSR, VSTMXCSR	Same as column 3
Type 6	VEXTRACTF128, VPERM2F128, VBROADCASTSD, VBROADCASTF128, VINSERTF128,		
Type 7		VMOVLHPS, VMOVHLPS, VPMOVMASKB, VPSLLDQ, VPSRLDQ, VPSLLW, VPSLLD, VPSLLQ, VPSRAW, VPSRAD, VPSRLW, VPSRLD, VPSRLQ	VMOVLHPS, VMOVHLPS
Type 8			
Type 11			
Type 12			

## 2.4.1 Exceptions Type 1 (Aligned memory reference)

**Table 2-18. Type 1 Class Exception Conditions**

Exception	Real	Virtual-8086	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			VEX prefix.
			X	X	VEX prefix: If XCRO[2:1] ≠ '11b'. If CR4.OSXSAVE[bit 18]=0.
	X	X	X	X	Legacy SSE instruction: If CR0.EM[bit 2] = 1. If CR4.OSFXSR[bit 9] = 0.
	X	X	X	X	If preceded by a LOCK prefix (FOH).
			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	X	VEX.256: Memory operand is not 32-byte aligned. VEX.128: Memory operand is not 16-byte aligned.
	X	X	X	X	Legacy SSE: Memory operand is not 16-byte aligned.
			X		For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If the memory address is in a non-canonical form.
	X	X			If any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF(fault-code)		X	X	X	For a page fault.

## 2.4.2 Exceptions Type 2 (>=16 Byte Memory Reference, Unaligned)

Table 2-19. Type 2 Class Exception Conditions

Exception	Real	Virtual 8086	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			VEX prefix.
	X	X	X	X	If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0.
			X	X	VEX prefix: If XCRO[2:1] ≠ '11b'. If CR4.OSXSAVE[bit 18]=0.
	X	X	X	X	Legacy SSE instruction: If CRO.EM[bit 2] = 1. If CR4.OSFXSR[bit 9] = 0.
	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 CRO.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	X	X	X	Legacy SSE: Memory operand is not 16-byte aligned.
			X		For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If the memory address is in a non-canonical form.
	X	X			If any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF(fault-code)		X	X	X	For a page fault.
SIMD Floating-point Exception, #XM	X	X	X	X	If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1.

## 2.4.3 Exceptions Type 3 (<16 Byte memory argument)

Table 2-20. Type 3 Class Exception Conditions

Exception	Real	Virtual-8086	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			VEX prefix.
	X	X	X	X	If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0.
			X	X	VEX prefix: If XCRO[2:1] ≠ '11b'. If CR4.OSXSAVE[bit 18]=0.
	X	X	X	X	Legacy SSE instruction: If CR0.EM[bit 2] = 1. If CR4.OSFXSR[bit 9] = 0.
	X	X	X	X	If preceded by a LOCK prefix (FOH).
			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.
				X	If the memory address is in a non-canonical form.
	X	X			If any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF(fault-code)		X	X	X	For a page fault.
Alignment Check #AC(0)		X	X	X	If alignment checking is enabled and an unaligned memory reference of 8 Bytes or less is made while the current privilege level is 3.
SIMD Floating-point Exception, #XM	X	X	X	X	If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1.

## 2.4.4 Exceptions Type 4 (>=16 Byte mem arg no alignment, no floating-point exceptions)

Table 2-21. Type 4 Class Exception Conditions

Exception	Real	Virtual-8086	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			VEX prefix.
			X	X	VEX prefix: If XCRO[2:1] ≠ '11b'. If CR4.OSXSAVE[bit 18]=0.
	X	X	X	X	Legacy SSE instruction: If CRO.EM[bit 2] = 1. If CR4.OSFXSR[bit 9] = 0.
	X	X	X	X	If preceded by a LOCK prefix (FOH).
			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 CRO.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	X	X	X	Legacy SSE: Memory operand is not 16-byte aligned. <sup>1</sup>
			X		For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If the memory address is in a non-canonical form.
	X	X			If any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF(fault-code)		X	X	X	For a page fault.

### NOTES:

1. PCMPSTRI, PCMPSTRM, PCMPISTRI, PCMPISTRM and LDDQU instructions do not cause #GP if the memory operand is not aligned to 16-Byte boundary.



## 2.4.5 Exceptions Type 5 (<16 Byte mem arg and no FP exceptions)

Table 2-22. Type 5 Class Exception Conditions

Exception	Real	Virtual-8086	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			VEX prefix.
			X	X	VEX prefix: If XCRO[2:1] ≠ '11b'. If CR4.OSXSAVE[bit 18]=0.
	X	X	X	X	Legacy SSE instruction: If CRO.EM[bit 2] = 1. If CR4.OSFXSR[bit 9] = 0.
	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 CRO.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.
				X	If the memory address is in a non-canonical form.
	X	X			If any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF(fault-code)		X	X	X	For a page fault.
Alignment Check #AC(0)		X	X	X	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

## 2.4.6 Exceptions Type 6 (VEX-Encoded Instructions Without Legacy SSE Analogues)

Note: At present, the AVX instructions in this category do not generate floating-point exceptions.

**Table 2-23. Type 6 Class Exception Conditions**

Exception	Real	Virtual-8086	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			VEX prefix.
			X	X	If XCRO[2:1] ≠ '11b'. If CR4.OSXSAVE[bit 18]=0.
			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	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM			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.
				X	If the memory address is in a non-canonical form.
Page Fault #PF(fault-code)			X	X	For a page fault.
Alignment Check #AC(0)			X	X	For 4 or 8 byte memory references if alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

## 2.4.7 Exceptions Type 7 (No FP exceptions, no memory arg)

Table 2-24. Type 7 Class Exception Conditions

Exception	Real	Virtual-8086	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			VEX prefix.
			X	X	VEX prefix: If XCRO[2:1] ≠ '11b'. If CR4.OSXSAVE[bit 18]=0.
	X	X	X	X	Legacy SSE instruction: If CR0.EM[bit 2] = 1. If CR4.OSFXSR[bit 9] = 0.
	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	If CR0.TS[bit 3]=1.

## 2.4.8 Exceptions Type 8 (AVX and no memory argument)

Table 2-25. Type 8 Class Exception Conditions

Exception	Real	Virtual-8086	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			Always in Real or Virtual-8086 mode.
			X	X	If XCRO[2:1] ≠ '11b'. If CR4.OSXSAVE[bit 18]=0. If CPUID.01H.ECX.AVX[bit 28]=0. If VEX.vvvv ≠ 1111B.
	X	X	X	X	If preceded by a LOCK prefix (F0H).
Device Not Available, #NM			X	X	If CR0.TS[bit 3]=1.

## 2.4.9 Exception Type 11 (VEX-only, mem arg no AC, floating-point exceptions)

Table 2-26. Type 11 Class Exception Conditions

Exception	Real	Virtual-8086	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			VEX prefix.
			X	X	VEX prefix: If XCRO[2:1] ≠ '11b'. If CR4.OSXSAVE[bit 18]=0.
	X	X	X	X	If preceded by a LOCK prefix (FOH).
			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.
				X	If the memory address is in a non-canonical form.
	X	X			If any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF (fault-code)		X	X	X	For a page fault.
SIMD Floating-Point Exception, #XM	X	X	X	X	If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1.

## 2.4.10 Exception Type 12 (VEX-only, VSIB mem arg, no AC, no floating-point exceptions)

Table 2-27. Type 12 Class Exception Conditions

Exception	Real	Virtual-8086	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			VEX prefix.
			X	X	VEX prefix: If XCRO[2:1] ≠ '11b'. If CR4.OSXSAVE[bit 18]=0.
	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	NA	If address size attribute is 16 bit.
	X	X	X	X	If ModR/M.mod = '11b'.
	X	X	X	X	If ModR/M.rm ≠ '100b'.
	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.
			X		For an illegal address in the SS segment.
Stack, SS(0)				X	If a memory address referencing the SS segment is in a non-canonical form.
			X		For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
General Protection, #GP(0)				X	If the memory address is in a non-canonical form.
	X	X			If any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF (fault-code)		X	X	X	For a page fault.

## 2.5 VEX ENCODING SUPPORT FOR GPR INSTRUCTIONS

VEX prefix may be used to encode instructions that operate on neither YMM nor XMM registers. VEX-encoded general-purpose-register instructions have the following properties:

- Instruction syntax support for three encodable operands.
- Encoding support for instruction syntax of non-destructive source operand, destination operand encoded via VEX.vvvv, and destructive three-operand syntax.
- Elimination of escape opcode byte (0FH), two-byte escape via a compact bit field representation within the VEX prefix.
- Elimination of the need to use REX prefix to encode the extended half of general-purpose register sets (R8-R15) for direct register access or memory addressing.
- Flexible and more compact bit fields are provided in the VEX prefix to retain the full functionality provided by REX prefix. REX.W, REX.X, REX.B functionalities are provided in the three-byte VEX prefix only.
- VEX-encoded GPR instructions are encoded with VEX.L=0.

Any VEX-encoded GPR instruction with a 66H, F2H, or F3H prefix preceding VEX will #UD.  
 Any VEX-encoded GPR instruction with a REX prefix proceeding VEX will #UD.  
 VEX-encoded GPR instructions are not supported in real and virtual 8086 modes.

## 2.5.1 Exception Conditions for VEX-Encoded GPR Instructions

The exception conditions applicable to VEX-encoded GPR instruction differs from those of legacy GPR instructions. Table 2-28 lists VEX-encoded GPR instructions. The exception conditions for VEX-encoded GPR instructions are found in Table 2-29 for those instructions which have a default operand size of 32 bits and 16-bit operand size is not encodable.

**Table 2-28. VEX-Encoded GPR Instructions**

Exception Class	Instruction
See Table 2-29	ANDN, BLSI, BLSMSK, BLSR, BZHI, MULX, PDEP, PEXT, RORX, SARX, SHLX, SHRX

(\*) - Additional exception restrictions are present - see the Instruction description for details.

**Table 2-29. Exception Definition (VEX-Encoded GPR Instructions)**

Exception	Real	Virtual-8086	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X	X	X	If BMI1/BMI2 CPUID feature flag is '0'.
	X	X			If a VEX prefix is present.
			X	X	If any REX, F2, F3, or 66 prefixes precede a VEX prefix.
Stack, SS(0)	X	X	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(fault-code)		X	X	X	For a page fault.
Alignment Check #AC(0)		X	X	X	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

## 2.6 INTEL® AVX-512 ENCODING

The majority of the Intel AVX-512 family of instructions (operating on 512/256/128-bit vector register operands) are encoded using a new prefix (called EVEX). Opmask instructions (operating on opmask register operands) are encoded using the VEX prefix. The EVEX prefix has some parts resembling the instruction encoding scheme using the VEX prefix, and many other capabilities not available with the VEX prefix.

The significant feature differences between EVEX and VEX are summarized below.

- EVEX is a 4-Byte prefix (the first byte must be 62H); VEX is either a 2-Byte (C5H is the first byte) or 3-Byte (C4H is the first byte) prefix.
- EVEX prefix can encode 32 vector registers (XMM/YMM/ZMM) in 64-bit mode.
- EVEX prefix can encode an opmask register for conditional processing or selection control in EVEX-encoded vector instructions. Opmask instructions, whose source/destination operands are opmask registers and treat the content of an opmask register as a single value, are encoded using the VEX prefix.
- EVEX memory addressing with disp8 form uses a compressed disp8 encoding scheme to improve the encoding density of the instruction byte stream.
- EVEX prefix can encode functionality that are specific to instruction classes (e.g., packed instruction with “load+op” semantic can support embedded broadcast functionality, floating-point instruction with rounding semantic can support static rounding functionality, floating-point instruction with non-rounding arithmetic semantic can support “suppress all exceptions” functionality).

### 2.6.1 Instruction Format and EVEX

The placement of the EVEX prefix in an IA instruction is represented in Figure 2-10.

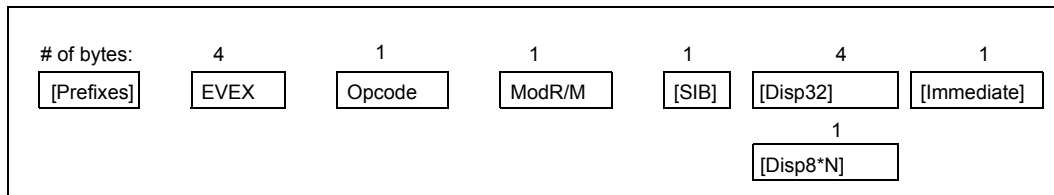


Figure 2-10. AVX-512 Instruction Format and the EVEX Prefix

The EVEX prefix is a 4-byte prefix, with the first two bytes derived from unused encoding form of the 32-bit-mode-only BOUND instruction. The layout of the EVEX prefix is shown in Figure 2-11. The first byte must be 62H, followed by three payload bytes, denoted as P0, P1, and P2 individually or collectively as P[23:0] (see Figure 2-11).

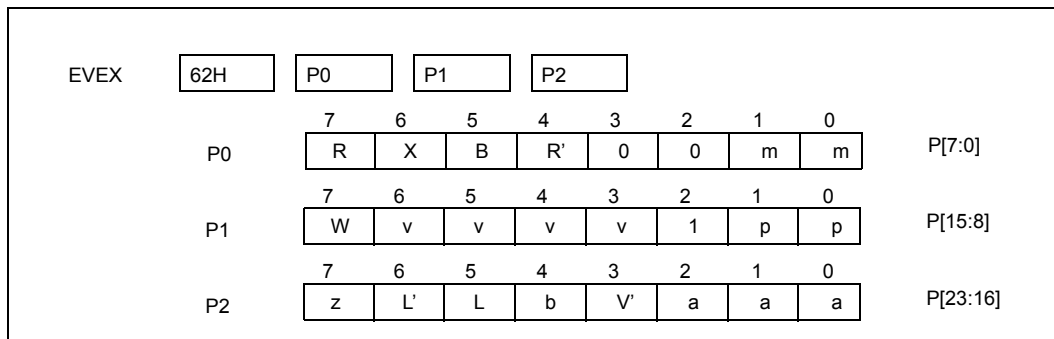


Figure 2-11. Bit Field Layout of the EVEX Prefix

Table 2-30. EVEX Prefix Bit Field Functional Grouping

Notation	Bit field Group	Position	Comment
--	Reserved	P[3 : 2]	Must be 0.
--	Fixed Value	P[10]	Must be 1.
EVEX.mm	Compressed legacy escape	P[1: 0]	Identical to low two bits of VEX.mmmmm.
EVEX.pp	Compressed legacy prefix	P[9 : 8]	Identical to VEX.pp.
EVEX.RXB	Next-8 register specifier modifier	P[7 : 5]	Combine with ModR/M.reg, ModR/M.rm (base, index/vidx).
EVEXR'	High-16 register specifier modifier	P[4]	Combine with EVEX.R and ModR/M.reg.
EVEXX	High-16 register specifier modifier	P[6]	Combine with EVEX.B and ModR/M.rm, when SIB/VSIB absent.
EVEX.vvvv	NDS register specifier	P[14 : 11]	Same as VEX.vvvv.
EVEXV'	High-16 NDS/VIDX register specifier	P[19]	Combine with EVEX.vvvv or when VSIB present.
EVEX.aaa	Embedded opmask register specifier	P[18 : 16]	
EVEX.W	Osize promotion/Opcode extension	P[15]	
EVEX.z	Zeroing/Merging	P[23]	
EVEX.b	Broadcast/RC/SAE Context	P[20]	
EVEX.L'L	Vector length/RC	P[22 : 21]	

The bit fields in P[23:0] are divided into the following functional groups (Table 2-30 provides a tabular summary):

- Reserved bits: P[3:2] must be 0, otherwise #UD.
- Fixed-value bit: P[10] must be 1, otherwise #UD.
- Compressed legacy prefix/escape bytes: P[1:0] is identical to the lowest 2 bits of VEX.mmmmm; P[9:8] is identical to VEX.pp.
- Operand specifier modifier bits for vector register, general purpose register, memory addressing: P[7:5] allows access to the next set of 8 registers beyond the low 8 registers when combined with ModR/M register specifiers.
- Operand specifier modifier bit for vector register: P[4] (or EVEX.R') allows access to the high 16 vector register set when combined with P[7] and ModR/M.reg specifier; P[6] can also provide access to a high 16 vector register when SIB or VSIB addressing are not needed.
- Non-destructive source /vector index operand specifier: P[19] and P[14:11] encode the second source vector register operand in a non-destructive source syntax, vector index register operand can access an upper 16 vector register using P[19].
- Op-mask register specifiers: P[18:16] encodes op-mask register set k0-k7 in instructions operating on vector registers.
- EVEX.W: P[15] is similar to VEX.W which serves either as opcode extension bit or operand size promotion to 64-bit in 64-bit mode.
- Vector destination merging/zeroing: P[23] encodes the destination result behavior which either zeroes the masked elements or leave masked element unchanged.
- Broadcast/Static-rounding/SAE context bit: P[20] encodes multiple functionality, which differs across different classes of instructions and can affect the meaning of the remaining field (EVEX.L'L). The functionality for the following instruction classes are:
  - Broadcasting a single element across the destination vector register: this applies to the instruction class with Load+Op semantic where one of the source operand is from memory.
  - Redirect L'L field (P[22:21]) as static rounding control for floating-point instructions with rounding semantic. Static rounding control overrides MXCSR.RC field and implies "Suppress all exceptions" (SAE).
  - Enable SAE for floating -point instructions with arithmetic semantic that is not rounding.
  - For instruction classes outside of the afore-mentioned three classes, setting EVEX.b will cause #UD.



- Vector length/rounding control specifier: P[22:21] can serve one of three options.
  - Vector length information for packed vector instructions.
  - Ignored for instructions operating on vector register content as a single data element.
  - Rounding control for floating-point instructions that have a rounding semantic and whose source and destination operands are all vector registers.

## 2.6.2 Register Specifier Encoding and EVEX

EVEX-encoded instruction can access 8 opmask registers, 16 general-purpose registers and 32 vector registers in 64-bit mode (8 general-purpose registers and 8 vector registers in non-64-bit modes). EVEX-encoding can support instruction syntax that access up to 4 instruction operands. Normal memory addressing modes and VSIB memory addressing are supported with EVEX prefix encoding. The mapping of register operands used by various instruction syntax and memory addressing in 64-bit mode are shown in Table 2-31. Opmask register encoding is described in Section 2.6.3.

**Table 2-31. 32-Register Support in 64-bit Mode Using EVEX with Embedded REX Bits**

	4 <sup>1</sup>	3	[2:0]	Reg. Type	Common Usages
<b>REG</b>	EVEX.R'	REX.R	modrm.reg	GPR, Vector	Destination or Source
<b>NDS/NDD</b>	EVEX.V'	EVEX.vvvv		GPR, Vector	2ndSource or Destination
<b>RM</b>	EVEX.X	EVEX.B	modrm.r/m	GPR, Vector	1st Source or Destination
<b>BASE</b>	0	EVEX.B	modrm.r/m	GPR	memory addressing
<b>INDEX</b>	0	EVEX.X	sib.index	GPR	memory addressing
<b>VIDX</b>	EVEX.V'	EVEX.X	sib.index	Vector	VSIB memory addressing
<b>IS4</b>	Imm8[3]	Imm8[7:4]		Vector	3rd Source

### NOTES:

1. Not applicable for accessing general purpose registers.

The mapping of register operands used by various instruction syntax and memory addressing in 32-bit modes are shown in Table 2-32.

**Table 2-32. EVEX Encoding Register Specifiers in 32-bit Mode**

	[2:0]	Reg. Type	Common Usages
<b>REG</b>	modrm.reg	GPR, Vector	Destination or Source
<b>NDS/NDD</b>	EVEX.vvv	GPR, Vector	2nd Source or Destination
<b>RM</b>	modrm.r/m	GPR, Vector	1st Source or Destination
<b>BASE</b>	modrm.r/m	GPR	Memory Addressing
<b>INDEX</b>	sib.index	GPR	Memory Addressing
<b>VIDX</b>	sib.index	Vector	VSIB Memory Addressing
<b>IS4</b>	Imm8[7:5]	Vector	3rd Source

## 2.6.3 Opmask Register Encoding

There are eight opmask registers, k0-k7. Opmask register encoding falls into two categories:

- Opmask registers that are the source or destination operands of an instruction treating the content of opmask register as a scalar value, are encoded using the VEX prefix scheme. It can support up to three operands using

standard modR/M byte's reg field and rm field and VEX.vvvv. Such a scalar opmask instruction does not support conditional update of the destination operand.

- An opmask register providing conditional processing and/or conditional update of the destination register of a vector instruction is encoded using EVEX.aaa field (see Section 2.6.4).
- An opmask register serving as the destination or source operand of a vector instruction is encoded using standard modR/M byte's reg field and rm fields.

**Table 2-33. Opmask Register Specifier Encoding**

	[2:0]	Register Access	Common Usages
REG	modrm.reg	k0-k7	Source
NDS	VEX.vvvv	k0-k7	2nd Source
RM	modrm.r/m	k0-7	1st Source
{k1}	EVEX.aaa	k0 <sup>1</sup> -k7	Opmask

**NOTES:**

1. Instructions that overwrite the conditional mask in opmask do not permit using k0 as the embedded mask.

## 2.6.4 Masking Support in EVEX

EVEX can encode an opmask register to conditionally control per-element computational operation and updating of result of an instruction to the destination operand. The predicate operand is known as the opmask register. The EVEX.aaa field, P[18:16] of the EVEX prefix, is used to encode one out of a set of eight 64-bit architectural registers. Note that from this set of 8 architectural registers, only k1 through k7 can be addressed as predicate operands. k0 can be used as a regular source or destination but cannot be encoded as a predicate operand.

AVX-512 instructions support two types of masking with EVEX.z bit (P[23]) controlling the type of masking:

- Merging-masking, which is the default type of masking for EVEX-encoded vector instructions, preserves the old value of each element of the destination where the corresponding mask bit has a 0. It corresponds to the case of EVEX.z = 0.
- Zeroing-masking, is enabled by having the EVEX.z bit set to 1. In this case, an element of the destination is set to 0 when the corresponding mask bit has a 0 value.

AVX-512 Foundation instructions can be divided into the following groups:

- Instructions which support "zeroing-masking".
  - Also allow merging-masking.
- Instructions which require aaa = 000.
  - Do not allow any form of masking.
- Instructions which allow merging-masking but do not allow zeroing-masking.
  - Require EVEX.z to be set to 0.
  - This group is mostly composed of instructions that write to memory.
- Instructions which require aaa <> 000 do not allow EVEX.z to be set to 1.
  - Allow merging-masking and do not allow zeroing-masking, e.g., gather instructions.

## 2.6.5 Compressed Displacement (disp8\*N) Support in EVEX

For memory addressing using disp8 form, EVEX-encoded instructions always use a compressed displacement scheme by multiplying disp8 in conjunction with a scaling factor N that is determined based on the vector length, the value of EVEX.b bit (embedded broadcast) and the input element size of the instruction. In general, the factor N corresponds to the number of bytes characterizing the internal memory operation of the input operand (e.g., 64 when the accessing a full 512-bit memory vector). The scale factor N is listed in Table 2-34 and Table 2-35 below,

where EVEX encoded instructions are classified using the **tupletype** attribute. The scale factor N of each tupletype is listed based on the vector length (VL) and other factors affecting it.

Table 2-34 covers EVEX-encoded instructions which has a load semantic in conjunction with additional computational or data element movement operation, operating either on the full vector or half vector (due to conversion of numerical precision from a wider format to narrower format). EVEX.b is supported for such instructions for data element sizes which are either dword or qword (see Section 2.6.11).

EVEX-encoded instruction that are pure load/store, and "Load+op" instruction semantic that operate on data element size less than dword do not support broadcasting using EVEX.b. These are listed in Table 2-35. Table 2-35 also includes many broadcast instructions which perform broadcast using a subset of data elements without using EVEX.b. These instructions and a few data element size conversion instruction are covered in Table 2-35. Instruction classified in Table 2-35 do not use EVEX.b and EVEX.b must be 0, otherwise #UD will occur.

The tupletype abbreviation will be referenced in the instruction operand encoding table in the reference page of each instruction, providing the cross reference for the scaling factor N to encoding memory addressing operand. Note that the disp8\*N rules still apply when using 16b addressing.

**Table 2-34. Compressed Displacement (DISP8\*N) Affected by Embedded Broadcast**

TupleType	EVEX.b	InputSize	EVEX.W	Broadcast	N (VL=128)	N (VL=256)	N (VL= 512)	Comment
Full Vector (FV)	0	32bit	0	none	16	32	64	Load+Op (Full Vector Dword/Qword)
	1	32bit	0	{1 tox}	4	4	4	
	0	64bit	1	none	16	32	64	
	1	64bit	1	{1 tox}	8	8	8	
Half Vector (HV)	0	32bit	0	none	8	16	32	Load+Op (Half Vector)
	1	32bit	0	{1 tox}	4	4	4	

**Table 2-35. EVEX DISP8\*N for Instructions Not Affected by Embedded Broadcast**

TupleType	InputSize	EVEX.W	N (VL= 128)	N (VL= 256)	N (VL= 512)	Comment
Full Vector Mem (FVM)	N/A	N/A	16	32	64	Load/store or subDword full vector
Tuple1 Scalar (T1S)	8bit	N/A	1	1	1	1 Tuple less than Full Vector
	16bit	N/A	2	2	2	
	32bit	0	4	4	4	
	64bit	1	8	8	8	
Tuple1 Fixed (T1F)	32bit	N/A	4	4	4	1 Tuple memsize not affected by EVEX.W
	64bit	N/A	8	8	8	
Tuple2 (T2)	32bit	0	8	8	8	Broadcast (2 elements)
	64bit	1	NA	16	16	
Tuple4 (T4)	32bit	0	NA	16	16	Broadcast (4 elements)
	64bit	1	NA	NA	32	
Tuple8 (T8)	32bit	0	NA	NA	32	Broadcast (8 elements)
Half Mem (HVM)	N/A	N/A	8	16	32	SubQword Conversion
QuarterMem (QVM)	N/A	N/A	4	8	16	SubDword Conversion
OctMem (OVM)	N/A	N/A	2	4	8	SubWord Conversion
Mem128 (M128)	N/A	N/A	16	16	16	Shift count from memory
MOVDDUP (DUP)	N/A	N/A	8	32	64	VMOVDDUP

## 2.6.6 EVEX Encoding of Broadcast/Rounding/SAE Support

EVEX.b can provide three types of encoding context, depending on the instruction classes:

- Embedded broadcasting of one data element from a source memory operand to the destination for vector instructions with “load+op” semantic.
- Static rounding control overriding MXCSR.RC for floating-point instructions with rounding semantic.
- “Suppress All exceptions” (SAE) overriding MXCSR mask control for floating-point arithmetic instructions that do not have rounding semantic.

## 2.6.7 Embedded Broadcast Support in EVEX

EVEX encodes an embedded broadcast functionality that is supported on many vector instructions with 32-bit (double word or single-precision floating-point) and 64-bit data elements, and when the source operand is from memory. EVEX.b (P[20]) bit is used to enable broadcast on load-op instructions. When enabled, only one element is loaded from memory and broadcasted to all other elements instead of loading the full memory size.

The following instruction classes do not support embedded broadcasting:

- Instructions with only one scalar result is written to the vector destination.
- Instructions with explicit broadcast functionality provided by its opcode.
- Instruction semantic is a pure load or a pure store operation.

## 2.6.8 Static Rounding Support in EVEX

Static rounding control embedded in the EVEX encoding system applies only to register-to-register flavor of floating-point instructions with rounding semantic at two distinct vector lengths: (i) scalar, (ii) 512-bit. In both cases, the field EVEX.L'L expresses rounding mode control overriding MXCSR.RC if EVEX.b is set. When EVEX.b is set, “suppress all exceptions” is implied. The processor behave as if all MXCSR masking controls are set.

## 2.6.9 SAE Support in EVEX

The EVEX encoding system allows arithmetic floating-point instructions without rounding semantic to be encoded with the SAE attribute. This capability applies to scalar and 512-bit vector lengths, register-to-register only, by setting EVEX.b. When EVEX.b is set, “suppress all exceptions” is implied. The processor behaves as if all MXCSR masking controls are set.

## 2.6.10 Vector Length Orthogonality

The architecture of EVEX encoding scheme can support SIMD instructions operating at multiple vector lengths. Many AVX-512 Foundation instructions operate at 512-bit vector length. The vector length of EVEX encoded vector instructions are generally determined using the L'L field in EVEX prefix, except for 512-bit floating-point, reg-reg instructions with rounding semantic. The table below shows the vector length corresponding to various values of the L'L bits. When EVEX is used to encode scalar instructions, L'L is generally ignored.

When EVEX.b bit is set for a register-register instructions with floating-point rounding semantic, the same two bits P2[6:5] specifies rounding mode for the instruction, with implied SAE behavior. The mapping of different instruction classes relative to the embedded broadcast/rounding/SAE control and the EVEX.L'L fields are summarized in Table 2-36.

**Table 2-36. EVEX Embedded Broadcast/Rounding/SAE and Vector Length on Vector Instructions**

Position	P2[4]	P2[6:5]	P2[6:5]
Broadcast/Rounding/SAE Context	EVEX.b	EVEX.L'L	EVEX.RC
Reg-reg, FP Instructions w/ rounding semantic	Enable static rounding control (SAE implied)	Vector length Implied (512 bit or scalar)	00b: SAE + RNE 01b: SAE + RD 10b: SAE + RU 11b: SAE + RZ
FP Instructions w/o rounding semantic, can cause #XF	SAE control	00b: 128-bit 01b: 256-bit 10b: 512-bit 11b: Reserved (#UD)	NA
Load+op Instructions w/ memory source	Broadcast Control		NA
Other Instructions ( Explicit Load/Store/Broadcast/Gather/Scatter)	Must be 0 (otherwise #UD)		NA

## 2.6.11 #UD Equations for EVEX

Instructions encoded using EVEX can face three types of UD conditions: state dependent, opcode independent and opcode dependent.

### 2.6.11.1 State Dependent #UD

In general, attempts to execute an instruction, which required OS support for incremental extended state component, will #UD if required state components were not enabled by OS. Table 2-37 lists instruction categories with respect to required processor state components. Attempts to execute a given category of instructions while enabled states were less than the required bit vector in XCR0 shown in Table 2-37 will cause #UD.

**Table 2-37. OS XSAVE Enabling Requirements of Instruction Categories**

Instruction Categories	Vector Register State Access	Required XCR0 Bit Vector [7:0]
Legacy SIMD prefix encoded Instructions (e.g SSE)	XMM	xxxxxx11b
VEX-encoded instructions operating on YMM	YMM	xxxxx111b
EVEX-encoded 128-bit instructions	ZMM	111xx111b
EVEX-encoded 256-bit instructions	ZMM	111xx111b
EVEX-encoded 512-bit instructions	ZMM	111xx111b
VEX-encoded instructions operating on opmask	k-reg	xx1xxx11b

### 2.6.11.2 Opcode Independent #UD

A number of bit fields in EVEX encoded instruction must obey mode-specific but opcode-independent patterns listed in Table 2-38.

**Table 2-38. Opcode Independent, State Dependent EVEX Bit Fields**

Position	Notation	64-bit #UD	Non-64-bit #UD
P[3 : 2]	--	if > 0	if > 0
P[10]	--	if 0	if 0
P[1 : 0]	EVEX.mm	if 00b	if 00b
P[7 : 6]	EVEX.RX	None (valid)	None (BOUND if EVEX.RX != 11b)

### 2.6.11.3 Opcode Dependent #UD

This section describes legal values for the rest of the EVEX bit fields. Table 2-39 lists the #UD conditions of EVEX prefix bit fields which encodes or modifies register operands.

**Table 2-39. #UD Conditions of Operand-Encoding EVEX Prefix Bit Fields**

Notation	Position	Operand Encoding	64-bit #UD	Non-64-bit #UD
EVEX.R	P[7]	ModRM.reg encodes k-reg	if EVEX.R = 0	None (BOUND if EVEX.RX != 11b)
		ModRM.reg is opcode extension	None (ignored)	
		ModRM.reg encodes all other registers	None (valid)	
EVEX.X	P[6]	ModRM.r/m encodes ZMM/YMM/XMM	None (valid)	
		ModRM.r/m encodes k-reg or GPR	None (ignored)	
		ModRM.r/m without SIB/VSIB	None (ignored)	
		ModRM.r/m with SIB/VSIB	None (valid)	
EVEX.B	P[5]	ModRM.r/m encodes k-reg	None (ignored)	None (ignored)
		ModRM.r/m encodes other registers	None (valid)	
		ModRM.r/m base present	None (valid)	
		ModRM.r/m base not present	None (ignored)	
EVEXR'	P[4]	ModRM.reg encodes k-reg or GPR	if 0	None (ignored)
		ModRM.reg is opcode extension	None (ignored)	
		ModRM.reg encodes ZMM/YMM/XMM	None (valid)	
EVEX.vvvv	P[14 : 11]	vvvv encodes ZMM/YMM/XMM	None (valid)	None (valid) P[14] ignored
		Otherwise	if != 1111b	if != 1111b
EVEXV'	P[19]	Encodes ZMM/YMM/XMM	None (valid)	if 0
		Otherwise	if 0	if 0

Table 2-40 lists the #UD conditions of instruction encoding of opmask register using EVEX.aaa and EVEX.z

**Table 2-40. #UD Conditions of Opmask Related Encoding Field**

Notation	Position	Operand Encoding	64-bit #UD	Non-64-bit #UD
EVEX.aaa	P[18 : 16]	Instructions do not use opmask for conditional processing <sup>1</sup> .	if aaa != 000b	if aaa != 000b
		Opmask used as conditional processing mask and updated at completion <sup>2</sup> .	if aaa = 000b	if aaa = 000b;
		Opmask used as conditional processing.	None (valid <sup>3</sup> )	None (valid <sup>1</sup> )
EVEX.z	P[23]	Vector instruction using opmask as source or destination <sup>4</sup> .	if EVEX.z != 0	if EVEX.z != 0
		Store instructions or gather/scatter instructions.	if EVEX.z != 0	if EVEX.z != 0
		Instruction supporting conditional processing mask with EVEX.aaa = 000b.	if EVEX.z != 0	if EVEX.z != 0

#### NOTES:

1. E.g., VBROADCASTMxxx, VPMOVM2x, VPMOVx2M.

2. E.g., Gather/Scatter family.

3. aaa can take any value. A value of 000 indicates that there is no masking on the instruction; in this case, all elements will be processed as if there was a mask of 'all ones' regardless of the actual value in KO.

4. E.g., VFPClassPD/PS, VCMPB/D/Q/W family, VPMOVM2x, VPMOVx2M.

Table 2-41 lists the #UD conditions of EVEX bit fields that depends on the context of EVEX.b.

**Table 2-41. #UD Conditions Dependent on EVEX.b Context**

Notation	Position	Operand Encoding	64-bit #UD	Non-64-bit #UD
EVEX.L'Lb	P[22 : 20]	Reg-reg, FP instructions with rounding semantic.	None (valid <sup>1</sup> )	None (valid <sup>1</sup> )
		Other reg-reg, FP instructions that can cause #XF.	None (valid <sup>2</sup> )	None (valid <sup>2</sup> )
		Other reg-mem instructions in Table 2-34.	None (valid <sup>3</sup> )	None (valid <sup>3</sup> )
		Other instruction classes <sup>4</sup> in Table 2-35.	If EVEX.b > 0	If EVEX.b > 0

**NOTES:**

1. L'L specifies rounding control, see Table 2-36, supports {er} syntax.
2. L'L specifies vector length, see Table 2-36, supports {sae} syntax.
3. L'L specifies vector length, see Table 2-36, supports embedded broadcast syntax
4. L'L specifies either vector length or ignored.

## 2.6.12 Device Not Available

EVEX-encoded instructions follow the same rules when it comes to generating #NM (Device Not Available) exception. In particular, it is generated when CR0.TS[bit 3]= 1.

## 2.6.13 Scalar Instructions

EVEX-encoded scalar SIMD instructions can access up to 32 registers in 64-bit mode. Scalar instructions support masking (using the least significant bit of the opmask register), but broadcasting is not supported.

## 2.7 EXCEPTION CLASSIFICATIONS OF EVEX-ENCODED INSTRUCTIONS

The exception behavior of EVEX-encoded instructions can be classified into the classes shown in the rest of this section. The classification of EVEX-encoded instructions follow a similar framework as those of AVX and AVX2 instructions using the VEX prefix. Exception types for EVEX-encoded instructions are named in the style of "E##" or with a suffix "E##XX". The "##" designation generally follows that of AVX/AVX2 instructions. The majority of EVEX encoded instruction with "Load+op" semantic supports memory fault suppression, which is represented by E##. The instructions with "Load+op" semantic but do not support fault suppression are named "E##NF". A summary table of exception classes by class names are shown below.

**Table 2-42. EVEX-Encoded Instruction Exception Class Summary**

Exception Class	Instruction set	Mem arg	(#XM)
Type E1	Vector Moves/Load/Stores	Explicitly aligned, w/ fault suppression	None
Type E1NF	Vector Non-temporal Stores	Explicitly aligned, no fault suppression	None
Type E2	FP Vector Load+op	Support fault suppression	Yes
Type E2NF	FP Vector Load+op	No fault suppression	Yes
Type E3	FP Scalar/Partial Vector, Load+Op	Support fault suppression	Yes
Type E3NF	FP Scalar/Partial Vector, Load+Op	No fault suppression	Yes
Type E4	Integer Vector Load+op	Support fault suppression	No
Type E4NF	Integer Vector Load+op	No fault suppression	No
Type E5	Legacy-like Promotion	Varies, Support fault suppression	No
Type E5NF	Legacy-like Promotion	Varies, No fault suppression	No

**Table 2-42. EVEX-Encoded Instruction Exception Class Summary**

Exception Class	Instruction set	Mem arg	(#XM)
Type E6	Post AVX Promotion	Varies, w/ fault suppression	No
Type E6NF	Post AVX Promotion	Varies, no fault suppression	No
Type E7NM	Register-to-register op	None	None
Type E9NF	Miscellaneous 128-bit	Vector-length Specific, no fault suppression	None
Type E10	Non-XF Scalar	Vector Length ignored, w/ fault suppression	None
Type E10NF	Non-XF Scalar	Vector Length ignored, no fault suppression	None
Type E11	VCVTPH2PS	Half Vector Length, w/ fault suppression	Yes
Type E11NF	VCVTPS2PH	Half Vector Length, no fault suppression	Yes
Type E12	Gather and Scatter Family	VSIB addressing, w/ fault suppression	None
Type E12NP	Gather and Scatter Prefetch Family	VSIB addressing, w/o page fault	None

Table 2-43 lists EVEX-encoded instruction mnemonic by exception classes.

**Table 2-43. EVEX Instructions in each Exception Class**

Exception Class	Instruction
Type E1	VMOVAPD, VMOVAPS, VMOVDQA32, VMOVDQA64
Type E1NF	VMOVNTDQ, VMOVNTDQA, VMOVNTPD, VMOVNTPS
Type E2	VADDPD, VADDPs, VCMPPD, VCMPPS, VCVTDQ2PS, VCVTPD2DQ, VCVTPD2PS, VCVTPS2DQ, VCVTTPD2DQ, VCVTTPS2DQ, VDIVPD, VDIVPS, VFMADDxxxPD, VFMADDxxxPS, VFMSUBADDxxxPD, VFMSUBADDxxxPS, VFMSUBxxxPD, VFMSUBxxxPS, VFNMADDxxxPD, VFNMADDxxxPS, VFNMSUBxxxPD, VFNMSUBxxxPS, VMAXPD, VMAXPS, VMINPD, VMINPS, VMULPD, VMULPS, VSQRTPD, VSQRTPS, VSUBPD, VSUBPS VCVTPD2QQ, VCVTPD2UQQ, VCVTPD2UDQ, VCVTPS2UDQS, VCVTQQ2PD, VCVTQQ2PS, VCVTTPD2DQ, VCVTTPD2QQ, VCVTTPD2UDQ, VCVTTPD2UQQ, VCVTTPS2DQ, VCVTTPS2UDQ, VCVTUDQ2PS, VCVTUQQ2PD, VCVTUQQ2PS, VFIXUPIMMPD, VFIXUPIMMPS, VGETEXPPD, VGETEXPPS, VGETMANTPD, VGETMANTPS, VRANGEPD, VRANGEPS, VREDUCEPD, VREDUCEPS, VRNDSCALEPD, VRNDSCALEPS, VSCALEFPD, VSCALEFPS, VRCP28PD, VRCP28PS, VRSQRT28PD, VRSQRT28PS
Type E3	VADDSd, VADDSs, VCMPSD, VCMPSs, VCVTPS2PD, VCVTSD2SS, VCVTSS2SD, VDIVSD, VDIVSS, VMAXSD, VMAXSS, VMINSD, VMINSS, VMULSD, VMULSS, VSQRSD, VSQRSS, VSUBSD, VSUBSS VCVTPS2QQ, VCVTPS2UQQ, VCVTTPS2QQ, VCVTTPS2UQQ, VFMADDxxxSD, VFMADDxxxSS, VFMSUBxxxSD, VFMSUBxxxSS, VFNMADDxxxSD, VFNMADDxxxSS, VFNMSUBxxxSD, VFNMSUBxxxSS, VFIXUPIMMSD, VFIXUPIMMSS, VGETEXPSD, VGETEXPPS, VGETMANTSD, VGETMANTSS, VRANGESD, VRANGESS, VREDUCESD, VREDUCESS, VRNDSCALESD, VRNDSCALESS, VSCALEFSD, VSCALEFSS, VRCP28SD, VRCP28SS, VRSQRT28SD, VRSQRT28SS
Type E3NF	VCOMISD, VCOMISS, VCVTSD2SI, VCVTSI2SD, VCVTSI2SS, VCVTSS2SI, VCVTSS2SI, VCVTSS2SI, VUCOMISD, VUCOMISS VCVTSD2USI, VCVTSS2USI, VCVTSS2USI, VCVTSS2USI, VCVTUSI2SD, VCVTUSI2SS



Table 2-43. EVEX Instructions in each Exception Class (Contd.)

Exception Class	Instruction
Type E4	VANDPD, VANDPS, VANDNPD, VANDNPS, VORPD, VORPS, VPABSD, VPABSQ, VPADDD, VPADDQ, VPANDD, VPANDQ, VPANDND, VPANDNQ, VPCMPEQD, VPCMPEQQ, VPCMPGTD, VPCMPGTQ, VPMAXSD, VPMAXSQ, VPMAXUD, VPMAXUQ, VPMINSQ, VPMINSQ, VPMINUD, VPMINUQ, VPMULLD, VPMULLQ, VPMULUDQ, VPMULDQ, VPORD, VPORQ, VPSUBD, VPSUBQ, VPXORD, VPXORQ, VXORPD, VXORPS, VPSLLVD, VPSLLVQ, VBLENDMPD, VBLENDMPS, VPBLENDMD, VPBLENDMQ, VFPCLASSPD, VFPCLASSPS, VPCMPD, VPCMPQ, VPCMPUD, VPCMPUQ, VPLZCNTD, VPLZCNTQ, VPROLD, VPROLQ, (VPSLLD, VPSLLQ, VPSRAD, VPSRAQ, VPSRLD, VPSRLQ) <sup>1</sup> , VPTERNLOGD, VPTERNLOGQ, VPTESTMD, VPTESTMQ, VPTESTNMD, VPTESTNMQ, VRCP14PD, VRCP14PS, VRSQRT14PD, VRSQRT14PS, VPCONFLICTD, VPCONFLICTQ, VPSRAVW, VPSRAVD, VPSRAVW, VPSRAVQ, VPMADD52LUQ, VPMADD52HUQ
E4.nb <sup>2</sup>	VMOVUPD, VMOVUPS, VMOVDQU8, VMOVDQU16, VMOVDQU32, VMOVDQU64, VPCMPB, VPCMPW, VPCMPUB, VPCMPUW, VEXPANDPD, VEXPANDPS, VPCOMPRESSD, VPCOMPRESSQ, VPEXPANDD, VPEXPANDQ, VCOMPRESSPD, VCOMPRESSPS, VPABSB, VPABSW, VPADDB, VPADDW, VPADDSB, VPADDSW, VPADDUSB, VPADDUSW, VPAVGB, VPAVGW, VPCMPEQB, VPCMPEQW, VPCMPGTB, VPCMPGTW, VPMAXSB, VPMAXSW, VPMAXUB, VPMAXUW, VPMINSB, VPMINSW, VPMINUB, VPMINUW, VPMULHRW, VPMULHUW, VPMULHW, VPMULLW, VPSUBB, VPSUBW, VPSUBSB, VPSUBSW, VPTESTMB, VPTESTMW, VPTESTNMB, VPTESTNMW, VPSLLW, VPSRAW, VPSRLW, VPSLLW, VPSRLW
Type E4NF	VPACKSSDW, VPACKUSDW, VPSHUFD, VPUNPCKHDQ, VPUNPCKHQDQ, VPUNPCKLDQ, VPUNPCKLQDQ, VSHUFPD, VSHUFPS, VUNPCKHPD, VUNPCKHPS, VUNPCKLPD, VUNPCKLPS, VPERMD, VPERMPS, VPERMPD, VPERMQ, VALIGND, VALIGNQ, VPERMI2D, VPERMI2PS, VPERMI2PD, VPERMI2Q, VPERMT2D, VPERMT2PS, VPERMT2Q, VPERMT2PD, VPERMILPD, VPERMILPS, VSHUFI32X4, VSHUFI64X2, VSHUFF32X4, VSHUFF64X2, VPMULTISHIFTQB
E4NF.nb <sup>2</sup>	VDBPSADBW, VPACKSSWB, VPACKUSWB, VPALIGNR, VPMADDWD, VPMADDUBSW, VMOVSHDUP, VMOVSLDUP, VPSADBW, VPSHUFB, VPSHUFBW, VPSHUFLW, VPSLLDQ, VPSRLDQ, VPSLLW, VPSRAW, VPSRLW, (VPSLLD, VPSLLQ, VPSRAD, VPSRAQ, VPSRLD, VPSRLQ) <sup>3</sup> , VPUNPCKHBW, VPUNPCKHWD, VPUNPCKLBW, VPUNPCKLWD, VPERMW, VPERMI2W, VPERMT2W, VPERMB, VPERMI2B, VPERMT2B
Type E5	VCVTDQ2PD, PMOVSBW, PMOVSBW, PMOVXBD, PMOVXBQ, PMOVXWD, PMOVXWQ, PMOVXQD, PMOVZXBW, PMOVZXBQ, PMOVZXBQ, PMOVZXWD, PMOVZXWQ, PMOVZXDQ VCVTUDQ2PD
Type E5NF	VMOVDDUP
Type E6	VBROADCASTSS, VBROADCASTSD, VBROADCASTF32X4, VBROADCASTI32X4, VPBROADCASTB, VPBROADCASTD, VPBROADCASTW, VPBROADCASTQ, VBROADCASTF32X2, VBROADCASTF32X4, VBROADCASTF64X2, VBROADCASTF32X8, VBROADCASTF64X4, VBROADCASTI32X2, VBROADCASTI32X4, VBROADCASTI64X2, VBROADCASTI32X8, VBROADCASTI64X4, VFPCLASSSD, VFPCLASSSS, VPMOVQB, VPMOVQB, VPMOVUSQB, VPMOVQW, VPMOVSQW, VPMOVUSQW, VPMOVQD, VPMOVSD, VPMOVUSQD, VPMOVDB, VPMOVSD, VPMOVUSDB, VPMOVDW, VPMOVSDW, VPMOVUSDW
Type E6NF	VEXTRACTF32X4, VEXTRACTF64X2, VEXTRACTF32X8, VINSERTF32X4, VINSERTF64X2, VINSERTF64X4, VINSERTF32X8, VINSERTI32X4, VINSERTI64X2, VINSERTI64X4, VINSERTI32X8, VEXTRACTI32X4, VEXTRACTI64X2, VEXTRACTI32X8, VEXTRACTI64X4, VPBROADCASTMB2Q, VPBROADCASTMW2D, VPMOVWB, VPMOVSWB, VPMOVUSWB
Type E7NM.128 <sup>4</sup>	VMOVLHPS, VMOVHLPS
Type E7NM.	(VPBROADCASTD, VPBROADCASTQ, VPBROADCASTB, VPBROADCASTW) <sup>5</sup> , VPMOVM2B, VPMOVM2D, VPMOVM2Q, VPMOVM2W, VPMOVB2M, VPMOVD2M, VPMOVQ2M, VPMOVW2M

Table 2-43. EVEX Instructions in each Exception Class (Contd.)

Exception Class	Instruction
Type E9NF	VEXTRACTPS, VINSERTPS, VMOVHPD, VMOVHPS, VMOVLPD, VMOVLPS, VMOVD, VMOVQ, VPEXTRB, VPEXTRD, VPEXTRW, VPEXTRQ, VPINSRB, VPINSRD, VPINSRW, VPINSRQ
Type E10	VMOVSD, VMOVSS, VRCP14SD, VRCP14SS, VRSQRT14SD, VRSQRT14SS,
Type E10NF	(VCVTSI2SD, VCVTUSI2SD) <sup>6</sup>
Type E11	VCVTPH2PS, VCVTPS2PH
Type E12	VGATHERDPS, VGATHERDPD, VGATHERQPS, VGATHERQPD, VPGATHERDD, VPGATHERDQ, VPGATHERQD, VPGATHERQQ, VPSCATTERDD, VPSCATTERDQ, VPSCATTERQD, VPSCATTERQQ, VSCATTERDPD, VSCATTERDPS, VSCATTERQPD, VSCATTERQPS
Type E12NP	VGATHERPFODPD, VGATHERPFODPS, VGATHERPFOQPD, VGATHERPFOQPS, VGATHERPF1DPD, VGATHERPF1DPS, VGATHERPF1QPD, VGATHERPF1QPS, VSCATTERPFODPD, VSCATTERPFODPS, VSCATTERPFOQPD, VSCATTERPFOQPS, VSCATTERPF1DPD, VSCATTERPF1DPS, VSCATTERPF1QPD, VSCATTERPF1QPS

**NOTES:**

1. Operand encoding FVI tupletype with immediate.
2. Embedded broadcast is not supported with the “.nb” suffix.
3. Operand encoding M128 tupletype.
4. #UD raised if EVEX.L'L !=00b (VL=128).
5. The source operand is a general purpose register.
6. W0 encoding only.

## 2.7.1 Exceptions Type E1 and E1NF of EVEX-Encoded Instructions

EVEX-encoded instructions with memory alignment restrictions, and supporting memory fault suppression follow exception class E1.

**Table 2-44. Type E1 Class Exception Conditions**

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			If EVEX prefix present.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: <ul style="list-style-type: none"> <li>State requirement, Table 2-37 not met.</li> <li>Opcode independent #UD condition in Table 2-38.</li> <li>Operand encoding #UD conditions in Table 2-39.</li> <li>Opmask encoding #UD condition of Table 2-40.</li> <li>If EVEX.b != 0.</li> <li>If EVEX.L'L != 10b (VL=512).</li> </ul>
	X	X	X	X	If preceded by a LOCK prefix (FOH).
			X	X	If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM	X	X	X	X	If CR0.TS[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	X	EVEX.512: Memory operand is not 64-byte aligned. EVEX.256: Memory operand is not 32-byte aligned. EVEX.128: Memory operand is not 16-byte aligned.
			X		If fault suppression not set, and an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If fault suppression not set, and the memory address is in a non-canonical form.
	X	X			If fault suppression not set, and any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF(fault-code)		X	X	X	If fault suppression not set, and a page fault.

EVEX-encoded instructions with memory alignment restrictions, but do not support memory fault suppression follow exception class E1NF.

**Table 2-45. Type E1NF Class Exception Conditions**

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			If EVEX prefix present.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: <ul style="list-style-type: none"> <li>State requirement, Table 2-37 not met.</li> <li>Opcode independent #UD condition in Table 2-38.</li> <li>Operand encoding #UD conditions in Table 2-39.</li> <li>Opmask encoding #UD condition of Table 2-40.</li> <li>If EVEX.b != 0.</li> <li>If EVEX.L'L != 10b (VL=512).</li> </ul>
	X	X	X	X	If preceded by a LOCK prefix (F0H).
			X	X	If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM	X	X	X	X	If CR0.TS[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	X	EVEX.512: Memory operand is not 64-byte aligned. EVEX.256: Memory operand is not 32-byte aligned. EVEX.128: Memory operand is not 16-byte aligned.
			X		For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If the memory address is in a non-canonical form.
	X	X			If any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF(fault-code)		X	X	X	For a page fault.

## 2.7.2 Exceptions Type E2 of EVEX-Encoded Instructions

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

**Table 2-46. Type E2 Class Exception Conditions**

Exception	Real	Virtual 8086	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			If EVEX prefix present.
	X	X	X	X	If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: <ul style="list-style-type: none"> <li>▪ State requirement, Table 2-37 not met.</li> <li>▪ Opcode independent #UD condition in Table 2-38.</li> <li>▪ Operand encoding #UD conditions in Table 2-39.</li> <li>▪ Opmask encoding #UD condition of Table 2-40.</li> <li>▪ If EVEX.L'L != 10b (VL=512).</li> </ul>
	X	X	X	X	If preceded by a LOCK prefix (F0H).
			X	X	If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM	X	X	X	X	If CR0.TS[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.
				X	If fault suppression not set, and the memory address is in a non-canonical form.
	X	X			If fault suppression not set, and any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF(fault-code)		X	X	X	If fault suppression not set, and a page fault.
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.

## 2.7.3 Exceptions Type E3 and E3NF of EVEX-Encoded Instructions

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

**Table 2-47. Type E3 Class Exception Conditions**

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			If EVEX prefix present.
	X	X	X	X	If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: <ul style="list-style-type: none"> <li>▪ State requirement, Table 2-37 not met.</li> <li>▪ Opcode independent #UD condition in Table 2-38.</li> <li>▪ Operand encoding #UD conditions in Table 2-39.</li> <li>▪ Opmask encoding #UD condition of Table 2-40.</li> <li>▪ If EVEX.b != 0.</li> </ul>
	X	X	X	X	If preceded by a LOCK prefix (F0H).
			X	X	If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM	X	X	X	X	If CR0.TS[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.
				X	If fault suppression not set, and the memory address is in a non-canonical form.
	X	X			If fault suppression not set, and any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF(fault-code)		X	X	X	If fault suppression not set, and a page fault.
Alignment Check #AC(0)		X	X	X	If alignment checking is enabled and an unaligned memory reference of 8 bytes or less is made while the current privilege level is 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.

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

**Table 2-48. Type E3NF Class Exception Conditions**

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			EVEX prefix.
	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 one of following conditions applies: <ul style="list-style-type: none"> <li>State requirement, Table 2-37 not met.</li> <li>Opcode independent #UD condition in Table 2-38.</li> <li>Operand encoding #UD conditions in Table 2-39.</li> <li>Opmask encoding #UD condition of Table 2-40.</li> <li>If EVEX.b != 0.</li> </ul>
	X	X	X	X	If preceded by a LOCK prefix (F0H).
			X	X	If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM	X	X	X	X	If CR0.TS[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.
				X	If the memory address is in a non-canonical form.
	X	X			If any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF(fault-code)		X	X	X	For a page fault.
Alignment Check #AC(0)		X	X	X	If alignment checking is enabled and an unaligned memory reference of 8 bytes or less is made while the current privilege level is 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.

## 2.7.4 Exceptions Type E4 and E4NF of EVEX-Encoded Instructions

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

**Table 2-49. Type E4 Class Exception Conditions**

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			If EVEX prefix present.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: <ul style="list-style-type: none"> <li>State requirement, Table 2-37 not met.</li> <li>Opcode independent #UD condition in Table 2-38.</li> <li>Operand encoding #UD conditions in Table 2-39.</li> <li>Opmask encoding #UD condition of Table 2-40.</li> <li>If EVEX.b != 0 and in E4.nb subclass (see E4.nb entries in Table 2-43).</li> <li>If EVEX.L'L != 10b (VL=512).</li> </ul>
	X	X	X	X	If preceded by a LOCK prefix (F0H).
			X	X	If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM	X	X	X	X	If CR0.TS[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.
				X	If fault suppression not set, and the memory address is in a non-canonical form.
	X	X			If fault suppression not set, and any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF(fault-code)		X	X	X	If fault suppression not set, and a page fault.



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

**Table 2-50. Type E4NF Class Exception Conditions**

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			If EVEX prefix present.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: <ul style="list-style-type: none"> <li>▪ State requirement, Table 2-37 not met.</li> <li>▪ Opcode independent #UD condition in Table 2-38.</li> <li>▪ Operand encoding #UD conditions in Table 2-39.</li> <li>▪ Opmask encoding #UD condition of Table 2-40.</li> <li>▪ If EVEX.b != 0 and in E4NF.nb subclass (see E4NF.nb entries in Table 2-43).</li> <li>▪ If EVEX.L'L != 10b (VL=512).</li> </ul>
	X	X	X	X	If preceded by a LOCK prefix (F0H).
			X	X	If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM	X	X	X	X	If CR0.TS[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.
				X	If the memory address is in a non-canonical form.
	X	X			If any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF(fault-code)		X	X	X	For a page fault.

## 2.7.5 Exceptions Type E5 and E5NF

EVEX-encoded scalar/partial-vector instructions that cause no SIMD FP exception and support memory fault suppression follow exception class E5.

**Table 2-51. Type E5 Class Exception Conditions**

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			If EVEX prefix present.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: <ul style="list-style-type: none"> <li>State requirement, Table 2-37 not met.</li> <li>Opcode independent #UD condition in Table 2-38.</li> <li>Operand encoding #UD conditions in Table 2-39.</li> <li>Opmask encoding #UD condition of Table 2-40.</li> <li>If EVEX.b != 0.</li> <li>If EVEX.L'L != 10b (VL=512).</li> </ul>
	X	X	X	X	If preceded by a LOCK prefix (FOH).
			X	X	If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM	X	X	X	X	If CR0.TS[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.
				X	If fault suppression not set, and the memory address is in a non-canonical form.
	X	X			If fault suppression not set, and any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF(fault-code)		X	X	X	If fault suppression not set, and a page fault.
Alignment Check #AC(0)		X	X	X	If alignment checking is enabled and an unaligned memory reference of 8 bytes or less is made while the current privilege level is 3.

EVEX-encoded scalar/partial vector instructions that do not cause SIMD FP exception nor support memory fault suppression follow exception class E5NF.

Table 2-52. Type E5NF Class Exception Conditions

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			If EVEX prefix present.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: <ul style="list-style-type: none"> <li>▪ State requirement, Table 2-37 not met.</li> <li>▪ Opcode independent #UD condition in Table 2-38.</li> <li>▪ Operand encoding #UD conditions in Table 2-39.</li> <li>▪ Opmask encoding #UD condition of Table 2-40.</li> <li>▪ If EVEX.b != 0.</li> <li>▪ If EVEX.L'L != 10b (VL=512).</li> </ul>
	X	X	X	X	If preceded by a LOCK prefix (FOH).
			X	X	If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1.
Stack, SS(0)			X		If an illegal address in the SS segment.
				X	If a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)			X		If an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If the memory address is in a non-canonical form.
	X	X			If any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF(fault-code)		X	X	X	For a page fault.
Alignment Check #AC(0)		X	X	X	If alignment checking is enabled and an unaligned memory reference of 8 bytes or less is made while the current privilege level is 3.

## 2.7.6 Exceptions Type E6 and E6NF

Table 2-53. Type E6 Class Exception Conditions

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			If EVEX prefix present.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: <ul style="list-style-type: none"> <li>▪ State requirement, Table 2-37 not met.</li> <li>▪ Opcode independent #UD condition in Table 2-38.</li> <li>▪ Operand encoding #UD conditions in Table 2-39.</li> <li>▪ Opmask encoding #UD condition of Table 2-40.</li> <li>▪ If EVEX.b != 0.</li> <li>▪ If EVEX.L'L != 10b (VL=512).</li> </ul>
			X	X	If preceded by a LOCK prefix (FOH).
			X	X	If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.
			X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM			X	X	If CR0.TS[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.
				X	If fault suppression not set, and the memory address is in a non-canonical form.
Page Fault #PF(fault-code)			X	X	If fault suppression not set, and a page fault.
Alignment Check #AC(0)			X	X	For 4 or 8 byte memory references if alignment checking is enabled and an unaligned memory reference of 8 bytes or less is made while the current privilege level is 3.

EVEX-encoded instructions that do not cause SIMD FP exception nor support memory fault suppression follow exception class E6NF.

**Table 2-54. Type E6NF Class Exception Conditions**

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			If EVEX prefix present.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: <ul style="list-style-type: none"> <li>▪ State requirement, Table 2-37 not met.</li> <li>▪ Opcode independent #UD condition in Table 2-38.</li> <li>▪ Operand encoding #UD conditions in Table 2-39.</li> <li>▪ Opmask encoding #UD condition of Table 2-40.</li> <li>▪ If EVEX.b != 0.</li> <li>▪ If EVEX.L'L != 10b (VL=512).</li> </ul>
			X	X	If preceded by a LOCK prefix (FOH).
			X	X	If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.
			X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM			X	X	If CR0.TS[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.
				X	If the memory address is in a non-canonical form.
Page Fault #PF(fault-code)			X	X	For a page fault.
Alignment Check #AC(0)			X	X	For 4 or 8 byte memory references if alignment checking is enabled and an unaligned memory reference of 8 bytes or less is made while the current privilege level is 3.

## 2.7.7 Exceptions Type E7NM

EVEX-encoded instructions that cause no SIMD FP exception and do not reference memory follow exception class E7NM.

**Table 2-55. Type E7NM Class Exception Conditions**

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			If EVEX prefix present.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: <ul style="list-style-type: none"> <li>▪ State requirement, Table 2-37 not met.</li> <li>▪ Opcode independent #UD condition in Table 2-38.</li> <li>▪ Operand encoding #UD conditions in Table 2-39.</li> <li>▪ Opmask encoding #UD condition of Table 2-40.</li> <li>▪ If EVEX.b != 0.</li> <li>▪ Instruction specific EVEX.L'L restriction not met.</li> </ul>
	X	X	X	X	If preceded by a LOCK prefix (FOH).
			X	X	If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM			X	X	If CR0.TS[bit 3]=1.

## 2.7.8 Exceptions Type E9 and E9NF

EVEX-encoded vector or partial-vector instructions that do not cause no SIMD FP exception and support memory fault suppression follow exception class E9.

**Table 2-56. Type E9 Class Exception Conditions**

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			If EVEX prefix present.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: <ul style="list-style-type: none"> <li>State requirement, Table 2-37 not met.</li> <li>Opcode independent #UD condition in Table 2-38.</li> <li>Operand encoding #UD conditions in Table 2-39.</li> <li>Opmask encoding #UD condition of Table 2-40.</li> <li>If EVEX.b != 0.</li> <li>If EVEX.L'L != 00b (VL=128).</li> </ul>
	X	X	X	X	If preceded by a LOCK prefix (FOH).
			X	X	If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM	X	X	X	X	If CR0.TS[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.
				X	If fault suppression not set, and the memory address is in a non-canonical form.
	X	X			If fault suppression not set, and any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF(fault-code)		X	X	X	If fault suppression not set, and a page fault.
Alignment Check #AC(0)		X	X	X	If alignment checking is enabled and an unaligned memory reference of 8 bytes or less is made while the current privilege level is 3.

EVEX-encoded vector or partial-vector instructions that must be encoded with VEX.L'L = 0, do not cause SIMD FP exception nor support memory fault suppression follow exception class E9NF.

**Table 2-57. Type E9NF Class Exception Conditions**

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			If EVEX prefix present.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: <ul style="list-style-type: none"> <li>▪ State requirement, Table 2-37 not met.</li> <li>▪ Opcode independent #UD condition in Table 2-38.</li> <li>▪ Operand encoding #UD conditions in Table 2-39.</li> <li>▪ Opmask encoding #UD condition of Table 2-40.</li> <li>▪ If EVEX.b != 0.</li> <li>▪ If EVEX.L'L != 00b (VL=128).</li> </ul>
	X	X	X	X	If preceded by a LOCK prefix (FOH).
			X	X	If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1.
Stack, SS(0)			X		If an illegal address in the SS segment.
				X	If a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)			X		If an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If the memory address is in a non-canonical form.
	X	X			If any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF(fault-code)		X	X	X	For a page fault.
Alignment Check #AC(0)		X	X	X	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.



## 2.7.9 Exceptions Type E10

EVEX-encoded scalar instructions that ignore EVEX.L'L vector length encoding and do not cause no SIMD FP exception, support memory fault suppression follow exception class E10.

**Table 2-58. Type E10 Class Exception Conditions**

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			If EVEX prefix present.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: <ul style="list-style-type: none"> <li>State requirement, Table 2-37 not met.</li> <li>Opcode independent #UD condition in Table 2-38.</li> <li>Operand encoding #UD conditions in Table 2-39.</li> <li>Opmask encoding #UD condition of Table 2-40.</li> <li>If EVEX.b != 0.</li> </ul>
	X	X	X	X	If preceded by a LOCK prefix (FOH).
			X	X	If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM	X	X	X	X	If CR0.TS[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.
				X	If fault suppression not set, and the memory address is in a non-canonical form.
	X	X			If fault suppression not set, and any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF(fault-code)		X	X	X	If fault suppression not set, and a page fault.
Alignment Check #AC(0)		X	X	X	If alignment checking is enabled and an unaligned memory reference of 8 bytes or less is made while the current privilege level is 3.

EVEX-encoded scalar instructions that must be encoded with VEX.L'L = 0, do not cause SIMD FP exception nor support memory fault suppression follow exception class E10NF.

**Table 2-59. Type E10NF Class Exception Conditions**

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			If EVEX prefix present.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: <ul style="list-style-type: none"> <li>▪ State requirement, Table 2-37 not met.</li> <li>▪ Opcode independent #UD condition in Table 2-38.</li> <li>▪ Operand encoding #UD conditions in Table 2-39.</li> <li>▪ Opmask encoding #UD condition of Table 2-40.</li> <li>▪ If EVEX.b != 0.</li> </ul>
	X	X	X	X	If preceded by a LOCK prefix (F0H).
			X	X	If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM	X	X	X	X	If CR0.TS[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.
				X	If fault suppression not set, and the memory address is in a non-canonical form.
	X	X			If fault suppression not set, and any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF(fault-code)		X	X	X	If fault suppression not set, and a page fault.
Alignment Check #AC(0)		X	X	X	If alignment checking is enabled and an unaligned memory reference of 8 bytes or less is made while the current privilege level is 3.

## 2.7.10 Exception Type E11 (EVEX-only, mem arg no AC, floating-point exceptions)

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

**Table 2-60. Type E11 Class Exception Conditions**

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			If EVEX prefix present.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: <ul style="list-style-type: none"> <li>State requirement, Table 2-37 not met.</li> <li>Opcode independent #UD condition in Table 2-38.</li> <li>Operand encoding #UD conditions in Table 2-39.</li> <li>Opmask encoding #UD condition of Table 2-40.</li> <li>If EVEX.b != 0.</li> <li>If EVEX.L'L != 10b (VL=512).</li> </ul>
	X	X	X	X	If preceded by a LOCK prefix (FOH).
			X	X	If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM	X	X	X	X	If CR0.TS[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.
				X	If fault suppression not set, and the memory address is in a non-canonical form.
	X	X			If fault suppression not set, and any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF (fault-code)		X	X	X	If fault suppression not set, and a page fault.
SIMD Floating-Point Exception, #XM	X	X	X	X	If an unmasked SIMD floating-point exception, {sae} not set, and CR4.OSXMMEX-CPT[bit 10] = 1.

## 2.7.11 Exception Type E12 and E12NP (VSIB mem arg, no AC, no floating-point exceptions)

Table 2-61. Type E12 Class Exception Conditions

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			If EVEX prefix present.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: <ul style="list-style-type: none"> <li>State requirement, Table 2-37 not met.</li> <li>Opcode independent #UD condition in Table 2-38.</li> <li>Operand encoding #UD conditions in Table 2-39.</li> <li>Opmask encoding #UD condition of Table 2-40.</li> <li>If EVEX.b != 0.</li> <li>If EVEX.L'L != 10b (VL=512).</li> <li>If vvvv != 1111b.</li> </ul>
	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	NA	If address size attribute is 16 bit.
	X	X	X	X	If ModR/M.mod = '11b'.
	X	X	X	X	If ModR/M.rm != '100b'.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
	X	X	X	X	If k0 is used (gather or scatter operation).
X	X	X	X	If index = destination register (gather operation).	
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.
				X	If the memory address is in a non-canonical form.
	X	X			If any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF (fault-code)		X	X	X	For a page fault.

EVEX-encoded prefetch instructions that do not cause #PF follow exception class E12NP.

**Table 2-62. Type E12NP Class Exception Conditions**

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			If EVEX prefix present.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: <ul style="list-style-type: none"> <li>▪ State requirement, Table 2-37 not met.</li> <li>▪ Opcode independent #UD condition in Table 2-38.</li> <li>▪ Operand encoding #UD conditions in Table 2-39.</li> <li>▪ Opmask encoding #UD condition of Table 2-40.</li> <li>▪ If EVEX.b != 0.</li> <li>▪ If EVEX.L'L != 10b (VL=512).</li> </ul>
	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	NA	If address size attribute is 16 bit.
	X	X	X	X	If ModR/M.mod = '11b'.
	X	X	X	X	If ModR/M.rm != '100b'.
	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.
			X		For an illegal address in the SS segment.
Stack, SS(0)				X	If a memory address referencing the SS segment is in a non-canonical form.
			X		For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
General Protection, #GP(0)				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.

## 2.8 EXCEPTION CLASSIFICATIONS OF OPMASK INSTRUCTIONS

The exception behavior of VEX-encoded opmask instructions are listed below.

Exception conditions of Opmask instructions that do not address memory are listed as Type K20.

**Table 2-63. TYPE K20 Exception Definition (VEX-Encoded OpMask Instructions w/o Memory Arg)**

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X	X	X	If relevant CPUID feature flag is '0'.
	X	X			If a VEX prefix is present.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: <ul style="list-style-type: none"> <li>▪ State requirement, Table 2-37 not met.</li> <li>▪ Opcode independent #UD condition in Table 2-38.</li> <li>▪ Operand encoding #UD conditions in Table 2-39.</li> </ul>
			X	X	If any REX, F2, F3, or 66 prefixes precede a VEX prefix.
			X	X	If ModRM:[7:6] != 11b.
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1.

Exception conditions of Opmask instructions that address memory are listed as Type K21.

**Table 2-64. TYPE K21 Exception Definition (VEX-Encoded OpMask Instructions Addressing Memory)**

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X	X	X	If relevant CPUID feature flag is '0'.
	X	X			If a VEX prefix is present.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: <ul style="list-style-type: none"> <li>▪ State requirement, Table 2-37 not met.</li> <li>▪ Opcode independent #UD condition in Table 2-38.</li> <li>▪ Operand encoding #UD conditions in Table 2-39.</li> </ul>
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1.
			X	X	If any REX, F2, F3, or 66 prefixes precede a VEX prefix.
Stack, SS(0)	X	X	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(fault-code)		X	X	X	For a page fault.
Alignment Check #AC(0)		X	X	X	If alignment checking is enabled and an unaligned memory reference of 8 bytes or less is made while the current privilege level is 3.

## 8. Updates to Chapter 3, Volume 2A

Change bars show changes to Chapter 3 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A: Instruction Set Reference, A-L*.

-----

Changes include minor updates to reflect volume addition, arrangement of information and document order numbers. EVEX encoding details added to instruction format section at the beginning of the chapter. Addition of Intel® AVX-512 instructions. The following instructions have been added and/or updated: ADDPD, ADDPS, ADDSD, ADDSS, ANDPD, ANDPS, ANDNPD, ANDNPS, BNDLDX, BNDSTX, CMPPD, CMPPS, CMPSD, CMPSS, COMISD, COMISS, CPUID, CVTDQ2PD, CVTDQ2PS, CVTPD2DQ, CVTPS2PD, CVTSD2SI, CVTSD2SS, CVTSI2SD, CVTSI2SS, CVTSS2SD, CVTSS2SI, CVTTPD2DQ, CVTTPS2DQ, CVTTSD2SI, CVTTSS2SI, CVTPD2PS, CVTPS2DQ, DIVPD, DIVSD, DIVSS, EXTRACTPS, FXAM, IMUL, INSERTPS, KADDW/KADDB/KADDQ/KADDD, KANDW/KANDB/KANDQ/KANDD, KANDNW/KANDNB/KANDNQ/KANDND, KMOVW/KMOVB/KMOVQ/KMOVD, KNOTW/KNOTB/KNOTQ/KNOTD, KORW/KORB/KORQ/KORD, KORTESW/KORTESB/KORTESQ/KORTESD, KSHIFTLW/KSHIFTLB/KSHIFTLQ/KSHIFTLD, KSHIFTRW/KSHIFTRB/KSHIFTRQ/KSHIFTRD, KTESTW/KTESTB/KTESTQ/KTESTD, KUNPCKBW/KUNPCKWD/KUNPCKDQ, KXNORW/KXNORB/KXNORQ/KXNORD, KXORW/KXORB/KXORQ/KXORD.



## CHAPTER 3 INSTRUCTION SET REFERENCE, A-L

This chapter describes the instruction set for the Intel 64 and IA-32 architectures (A-L) in IA-32e, protected, virtual-8086, and real-address modes of operation. The set includes general-purpose, x87 FPU, MMX, SSE/SSE2/SSE3/SSSE3/SSE4, AESNI/PCLMULQDQ, AVX and system instructions. See also Chapter 4, “Instruction Set Reference, M-U,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2B*, and Chapter 5, “Instruction Set Reference, V-Z,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2C*.

For each instruction, each operand combination is described. A description of the instruction and its operand, an operational description, a description of the effect of the instructions on flags in the EFLAGS register, and a summary of exceptions that can be generated are also provided.

### 3.1 INTERPRETING THE INSTRUCTION REFERENCE PAGES

This section describes the format of information contained in the instruction reference pages in this chapter. It explains notational conventions and abbreviations used in these sections.

#### 3.1.1 Instruction Format

The following is an example of the format used for each instruction description in this chapter. The heading below introduces the example. The table below provides an example summary table.

#### CMC—Complement Carry Flag [this is an example]

Opcode	Instruction	Op/En	64/32-bit Mode	CPUID Feature Flag	Description
F5	CMC	A	V/V	NP	Complement carry flag.

#### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

### 3.1.1.1 Opcode Column in the Instruction Summary Table (Instructions without VEX Prefix)

The “Opcode” column in the table above shows the object code produced for each form of the instruction. When possible, codes are given as hexadecimal bytes in the same order in which they appear in memory. Definitions of entries other than hexadecimal bytes are as follows:

- **REX.W** — Indicates the use of a REX prefix that affects operand size or instruction semantics. The ordering of the REX prefix and other optional/mandatory instruction prefixes are discussed Chapter 2. Note that REX prefixes that promote legacy instructions to 64-bit behavior are not listed explicitly in the opcode column.
- **/digit** — A digit between 0 and 7 indicates that the ModR/M byte of the instruction uses only the r/m (register or memory) operand. The reg field contains the digit that provides an extension to the instruction's opcode.
- **/r** — Indicates that the ModR/M byte of the instruction contains a register operand and an r/m operand.
- **cb, cw, cd, cp, co, ct** — A 1-byte (cb), 2-byte (cw), 4-byte (cd), 6-byte (cp), 8-byte (co) or 10-byte (ct) value following the opcode. This value is used to specify a code offset and possibly a new value for the code segment register.
- **ib, iw, id, io** — A 1-byte (ib), 2-byte (iw), 4-byte (id) or 8-byte (io) immediate operand to the instruction that follows the opcode, ModR/M bytes or scale-indexing bytes. The opcode determines if the operand is a signed value. All words, doublewords and quadwords are given with the low-order byte first.
- **+rb, +rw, +rd, +ro** — Indicated the lower 3 bits of the opcode byte is used to encode the register operand without a modR/M byte. The instruction lists the corresponding hexadecimal value of the opcode byte with low 3 bits as 000b. In non-64-bit mode, a register code, from 0 through 7, is added to the hexadecimal value of the opcode byte. In 64-bit mode, indicates the four bit field of REX.b and opcode[2:0] field encodes the register operand of the instruction. “+ro” is applicable only in 64-bit mode. See Table 3-1 for the codes.
- **+i** — A number used in floating-point instructions when one of the operands is ST(i) from the FPU register stack. The number i (which can range from 0 to 7) is added to the hexadecimal byte given at the left of the plus sign to form a single opcode byte.

**Table 3-1. Register Codes Associated With +rb, +rw, +rd, +ro**

byte register			word register			dword register			quadword register (64-Bit Mode only)		
Register	REX.B	Reg Field	Register	REX.B	Reg Field	Register	REX.B	Reg Field	Register	REX.B	Reg Field
AL	None	0	AX	None	0	EAX	None	0	RAX	None	0
CL	None	1	CX	None	1	ECX	None	1	RCX	None	1
DL	None	2	DX	None	2	EDX	None	2	RDX	None	2
BL	None	3	BX	None	3	EBX	None	3	RBX	None	3
AH	Not encodable (N.E.)	4	SP	None	4	ESP	None	4	N/A	N/A	N/A
CH	N.E.	5	BP	None	5	EBP	None	5	N/A	N/A	N/A
DH	N.E.	6	SI	None	6	ESI	None	6	N/A	N/A	N/A
BH	N.E.	7	DI	None	7	EDI	None	7	N/A	N/A	N/A
SPL	Yes	4	SP	None	4	ESP	None	4	RSP	None	4
BPL	Yes	5	BP	None	5	EBP	None	5	RBP	None	5
SIL	Yes	6	SI	None	6	ESI	None	6	RSI	None	6
DIL	Yes	7	DI	None	7	EDI	None	7	RDI	None	7
Registers R8 - R15 (see below): Available in 64-Bit Mode Only											
R8L	Yes	0	R8W	Yes	0	R8D	Yes	0	R8	Yes	0
R9L	Yes	1	R9W	Yes	1	R9D	Yes	1	R9	Yes	1
R10L	Yes	2	R10W	Yes	2	R10D	Yes	2	R10	Yes	2

Table 3-1. Register Codes Associated With +rb, +rw, +rd, +ro (Contd.)

byte register			word register			dword register			quadword register (64-Bit Mode only)		
Register	REX.B	Reg Field	Register	REX.B	Reg Field	Register	REX.B	Reg Field	Register	REX.B	Reg Field
R11L	Yes	3	R11W	Yes	3	R11D	Yes	3	R11	Yes	3
R12L	Yes	4	R12W	Yes	4	R12D	Yes	4	R12	Yes	4
R13L	Yes	5	R13W	Yes	5	R13D	Yes	5	R13	Yes	5
R14L	Yes	6	R14W	Yes	6	R14D	Yes	6	R14	Yes	6
R15L	Yes	7	R15W	Yes	7	R15D	Yes	7	R15	Yes	7

### 3.1.1.2 Opcode Column in the Instruction Summary Table (Instructions with VEX prefix)

In the Instruction Summary Table, the Opcode column presents each instruction encoded using the VEX prefix in following form (including the modR/M byte if applicable, the immediate byte if applicable):

**VEX.[NDS].[128,256].[66,F2,F3].OF/OF3A/OF38.[W0,W1] opcode [/r] [/ib,/is4]**

- **VEX** — Indicates the presence of the VEX prefix is required. The VEX prefix can be encoded using the three-byte form (the first byte is C4H), or using the two-byte form (the first byte is C5H). The two-byte form of VEX only applies to those instructions that do not require the following fields to be encoded: VEX.mmmmm, VEX.W, VEX.X, VEX.B. Refer to Section 2.3 for more detail on the VEX prefix.

The encoding of various sub-fields of the VEX prefix is described using the following notations:

- **NDS, NDD, DDS**: Specifies that VEX.vvvv field is valid for the encoding of a register operand:
  - VEX.NDS: VEX.vvvv encodes the first source register in an instruction syntax where the content of source registers will be preserved.
  - VEX.NDD: VEX.vvvv encodes the destination register that cannot be encoded by ModR/M:reg field.
  - VEX.DDS: VEX.vvvv encodes the second source register in a three-operand instruction syntax where the content of first source register will be overwritten by the result.
  - If none of NDS, NDD, and DDS is present, VEX.vvvv must be 1111b (i.e. VEX.vvvv does not encode an operand). The VEX.vvvv field can be encoded using either the 2-byte or 3-byte form of the VEX prefix.
- **128,256**: VEX.L field can be 0 (denoted by VEX.128 or VEX.LZ) or 1 (denoted by VEX.256). The VEX.L field can be encoded using either the 2-byte or 3-byte form of the VEX prefix. The presence of the notation VEX.256 or VEX.128 in the opcode column should be interpreted as follows:
  - If VEX.256 is present in the opcode column: The semantics of the instruction must be encoded with VEX.L = 1. An attempt to encode this instruction with VEX.L = 0 can result in one of two situations: (a) if VEX.128 version is defined, the processor will behave according to the defined VEX.128 behavior; (b) an #UD occurs if there is no VEX.128 version defined.
  - If VEX.128 is present in the opcode column but there is no VEX.256 version defined for the same opcode byte: Two situations apply: (a) For VEX-encoded, 128-bit SIMD integer instructions, software must encode the instruction with VEX.L = 0. The processor will treat the opcode byte encoded with VEX.L = 1 by causing an #UD exception; (b) For VEX-encoded, 128-bit packed floating-point instructions, software must encode the instruction with VEX.L = 0. The processor will treat the opcode byte encoded with VEX.L = 1 by causing an #UD exception (e.g. VMOVLPS).
  - If VEX.LIG is present in the opcode column: The VEX.L value is ignored. This generally applies to VEX-encoded scalar SIMD floating-point instructions. Scalar SIMD floating-point instruction can be distinguished from the mnemonic of the instruction. Generally, the last two letters of the instruction mnemonic would be either "SS", "SD", or "SI" for SIMD floating-point conversion instructions.
  - If VEX.LZ is present in the opcode column: The VEX.L must be encoded to be 0B, an #UD occurs if VEX.L is not zero.

- **66,F2,F3**: The presence or absence of these values map to the VEX.pp field encodings. If absent, this corresponds to VEX.pp=00B. If present, the corresponding VEX.pp value affects the “opcode” byte in the same way as if a SIMD prefix (66H, F2H or F3H) does to the ensuing opcode byte. Thus a non-zero encoding of VEX.pp may be considered as an implied 66H/F2H/F3H prefix. The VEX.pp field may be encoded using either the 2-byte or 3-byte form of the VEX prefix.
- **0F,0F3A,0F38**: The presence maps to a valid encoding of the VEX.mmmmm field. Only three encoded values of VEX.mmmmm are defined as valid, corresponding to the escape byte sequence of 0FH, 0F3AH and 0F38H. The effect of a valid VEX.mmmmm encoding on the ensuing opcode byte is same as if the corresponding escape byte sequence on the ensuing opcode byte for non-VEX encoded instructions. Thus a valid encoding of VEX.mmmmm may be considered as an implied escape byte sequence of either 0FH, 0F3AH or 0F38H. The VEX.mmmmm field must be encoded using the 3-byte form of VEX prefix.
- **0F,0F3A,0F38 and 2-byte/3-byte VEX**: The presence of 0F3A and 0F38 in the opcode column implies that opcode can only be encoded by the three-byte form of VEX. The presence of 0F in the opcode column does not preclude the opcode to be encoded by the two-byte of VEX if the semantics of the opcode does not require any subfield of VEX not present in the two-byte form of the VEX prefix.
- **W0**: VEX.W=0.
- **W1**: VEX.W=1.
- The presence of W0/W1 in the opcode column applies to two situations: (a) it is treated as an extended opcode bit, (b) the instruction semantics support an operand size promotion to 64-bit of a general-purpose register operand or a 32-bit memory operand. The presence of W1 in the opcode column implies the opcode must be encoded using the 3-byte form of the VEX prefix. The presence of W0 in the opcode column does not preclude the opcode to be encoded using the C5H form of the VEX prefix, if the semantics of the opcode does not require other VEX subfields not present in the two-byte form of the VEX prefix. Please see Section 2.3 on the subfield definitions within VEX.
- **WIG**: can use C5H form (if not requiring VEX.mmmmm) or VEX.W value is ignored in the C4H form of VEX prefix.
- If WIG is present, the instruction may be encoded using either the two-byte form or the three-byte form of VEX. When encoding the instruction using the three-byte form of VEX, the value of VEX.W is ignored.
- **opcode** — Instruction opcode.
- **/is4** — An 8-bit immediate byte is present containing a source register specifier in imm[7:4] and instruction-specific payload in imm[3:0].
- In general, the encoding of VEX.R, VEX.X, VEX.B field are not shown explicitly in the opcode column. The encoding scheme of VEX.R, VEX.X, VEX.B fields must follow the rules defined in Section 2.3.

#### **EVEX.[NDS/NDD/DDS].[128,256,512,LIG].[66,F2,F3].0F/0F3A/0F38.[W0,W1,WIG] opcode [/r] [ib,/is4]**

- **EVEX** — The EVEX prefix is encoded using the four-byte form (the first byte is 62H). Refer to Section 4.2 for more detail on the EVEX prefix.

The encoding of various sub-fields of the EVEX prefix is described using the following notations:

- **NDS, NDD, DDS**: implies that EVEX.vvvv (and EVEX.v’) field is valid for the encoding of an operand. It may specify either the source register (NDS) or the destination register (NDD). DDS expresses a syntax where vvvv encodes the second source register in a three-operand instruction syntax where the content of first source register will be overwritten by the result. If both NDS and NDD absent (i.e. EVEX.vvvv does not encode an operand), EVEX.vvvv must be 1111b (and EVEX.v’ must be 1b).
- **128, 256, 512, LIG**: This corresponds to the vector length; three values are allowed by EVEX: 512-bit, 256-bit and 128-bit. Alternatively, vector length is ignored (LIG) for certain instructions; this typically applies to scalar instructions operating on one data element of a vector register.
- **66,F2,F3**: The presence of these value maps to the EVEX.pp field encodings. The corresponding VEX.pp value affects the “opcode” byte in the same way as if a SIMD prefix (66H, F2H or F3H) does to the ensuing opcode byte. Thus a non-zero encoding of VEX.pp may be considered as an implied 66H/F2H/F3H prefix.

- **0F,0F3A,0F38**: The presence maps to a valid encoding of the EVEX.mmm field. Only three encoded values of EVEX.mmm are defined as valid, corresponding to the escape byte sequence of 0FH, 0F3AH and 0F38H. The effect of a valid EVEX.mmm encoding on the ensuing opcode byte is the same as if the corresponding escape byte sequence on the ensuing opcode byte for non-EVEX encoded instructions. Thus a valid encoding of EVEX.mmm may be considered as an implied escape byte sequence of either 0FH, 0F3AH or 0F38H.
- **W0**: EVEX.W=0.
- **W1**: EVEX.W=1.
- **WIG**: EVEX.W bit ignored
- **opcode** — Instruction opcode.
- **/is4** — A relative address in the range from 128 bytes before the end of the instruction to 127 bytes after the end of the instruction. An 8-bit immediate byte is present containing a source register specifier in imm[7:4] and instruction-specific payload in imm[3:0].
- In general, the encoding of EVEX.R and R', EVEX.X and X', and EVEX.B and B' fields are not shown explicitly in the opcode column.

### 3.1.1.3 Instruction Column in the Opcode Summary Table

The “Instruction” column gives the syntax of the instruction statement as it would appear in an ASM386 program. The following is a list of the symbols used to represent operands in the instruction statements:

- **rel8** — A relative address in the range from 128 bytes before the end of the instruction to 127 bytes after the end of the instruction.
- **rel16, rel32** — A relative address within the same code segment as the instruction assembled. The rel16 symbol applies to instructions with an operand-size attribute of 16 bits; the rel32 symbol applies to instructions with an operand-size attribute of 32 bits.
- **ptr16:16, ptr16:32** — A far pointer, typically to a code segment different from that of the instruction. The notation *16:16* indicates that the value of the pointer has two parts. The value to the left of the colon is a 16-bit selector or value destined for the code segment register. The value to the right corresponds to the offset within the destination segment. The ptr16:16 symbol is used when the instruction's operand-size attribute is 16 bits; the ptr16:32 symbol is used when the operand-size attribute is 32 bits.
- **r8** — One of the byte general-purpose registers: AL, CL, DL, BL, AH, CH, DH, BH, BPL, SPL, DIL and SIL; or one of the byte registers (R8L - R15L) available when using REX.R and 64-bit mode.
- **r16** — One of the word general-purpose registers: AX, CX, DX, BX, SP, BP, SI, DI; or one of the word registers (R8-R15) available when using REX.R and 64-bit mode.
- **r32** — One of the doubleword general-purpose registers: EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI; or one of the doubleword registers (R8D - R15D) available when using REX.R in 64-bit mode.
- **r64** — One of the quadword general-purpose registers: RAX, RBX, RCX, RDX, RDI, RSI, RBP, RSP, R8-R15. These are available when using REX.R and 64-bit mode.
- **imm8** — An immediate byte value. The imm8 symbol is a signed number between -128 and +127 inclusive. For instructions in which imm8 is combined with a word or doubleword operand, the immediate value is sign-extended to form a word or doubleword. The upper byte of the word is filled with the topmost bit of the immediate value.
- **imm16** — An immediate word value used for instructions whose operand-size attribute is 16 bits. This is a number between -32,768 and +32,767 inclusive.
- **imm32** — An immediate doubleword value used for instructions whose operand-size attribute is 32 bits. It allows the use of a number between +2,147,483,647 and -2,147,483,648 inclusive.
- **imm64** — An immediate quadword value used for instructions whose operand-size attribute is 64 bits. The value allows the use of a number between +9,223,372,036,854,775,807 and -9,223,372,036,854,775,808 inclusive.
- **r/m8** — A byte operand that is either the contents of a byte general-purpose register (AL, CL, DL, BL, AH, CH, DH, BH, BPL, SPL, DIL and SIL) or a byte from memory. Byte registers R8L - R15L are available using REX.R in 64-bit mode.

- **r/m16** — A word general-purpose register or memory operand used for instructions whose operand-size attribute is 16 bits. The word general-purpose registers are: AX, CX, DX, BX, SP, BP, SI, DI. The contents of memory are found at the address provided by the effective address computation. Word registers R8W - R15W are available using REX.R in 64-bit mode.
- **r/m32** — A doubleword general-purpose register or memory operand used for instructions whose operand-size attribute is 32 bits. The doubleword general-purpose registers are: EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI. The contents of memory are found at the address provided by the effective address computation. Doubleword registers R8D - R15D are available when using REX.R in 64-bit mode.
- **r/m64** — A quadword general-purpose register or memory operand used for instructions whose operand-size attribute is 64 bits when using REX.W. Quadword general-purpose registers are: RAX, RBX, RCX, RDX, RDI, RSI, RBP, RSP, R8-R15; these are available only in 64-bit mode. The contents of memory are found at the address provided by the effective address computation.
- **m** — A 16-, 32- or 64-bit operand in memory.
- **m8** — A byte operand in memory, usually expressed as a variable or array name, but pointed to by the DS:(E)SI or ES:(E)DI registers. In 64-bit mode, it is pointed to by the RSI or RDI registers.
- **m16** — A word operand in memory, usually expressed as a variable or array name, but pointed to by the DS:(E)SI or ES:(E)DI registers. This nomenclature is used only with the string instructions.
- **m32** — A doubleword operand in memory, usually expressed as a variable or array name, but pointed to by the DS:(E)SI or ES:(E)DI registers. This nomenclature is used only with the string instructions.
- **m64** — A memory quadword operand in memory.
- **m128** — A memory double quadword operand in memory.
- **m16:16, m16:32 & m16:64** — A memory operand containing a far pointer composed of two numbers. The number to the left of the colon corresponds to the pointer's segment selector. The number to the right corresponds to its offset.
- **m16&32, m16&16, m32&32, m16&64** — A memory operand consisting of data item pairs whose sizes are indicated on the left and the right side of the ampersand. All memory addressing modes are allowed. The m16&16 and m32&32 operands are used by the BOUND instruction to provide an operand containing an upper and lower bounds for array indices. The m16&32 operand is used by LIDT and LGDT to provide a word with which to load the limit field, and a doubleword with which to load the base field of the corresponding GDTR and IDTR registers. The m16&64 operand is used by LIDT and LGDT in 64-bit mode to provide a word with which to load the limit field, and a quadword with which to load the base field of the corresponding GDTR and IDTR registers.
- **moffs8, moffs16, moffs32, moffs64** — A simple memory variable (memory offset) of type byte, word, or doubleword used by some variants of the MOV instruction. The actual address is given by a simple offset relative to the segment base. No ModR/M byte is used in the instruction. The number shown with moffs indicates its size, which is determined by the address-size attribute of the instruction.
- **Sreg** — A segment register. The segment register bit assignments are ES = 0, CS = 1, SS = 2, DS = 3, FS = 4, and GS = 5.
- **m32fp, m64fp, m80fp** — A single-precision, double-precision, and double extended-precision (respectively) floating-point operand in memory. These symbols designate floating-point values that are used as operands for x87 FPU floating-point instructions.
- **m16int, m32int, m64int** — A word, doubleword, and quadword integer (respectively) operand in memory. These symbols designate integers that are used as operands for x87 FPU integer instructions.
- **ST or ST(0)** — The top element of the FPU register stack.
- **ST(i)** — The  $i^{\text{th}}$  element from the top of the FPU register stack ( $i \leftarrow 0$  through 7).
- **mm** — An MMX register. The 64-bit MMX registers are: MM0 through MM7.
- **mm/m32** — The low order 32 bits of an MMX register or a 32-bit memory operand. The 64-bit MMX registers are: MM0 through MM7. The contents of memory are found at the address provided by the effective address computation.
- **mm/m64** — An MMX register or a 64-bit memory operand. The 64-bit MMX registers are: MM0 through MM7. The contents of memory are found at the address provided by the effective address computation.



- **xmm** — An XMM register. The 128-bit XMM registers are: XMM0 through XMM7; XMM8 through XMM15 are available using REX.R in 64-bit mode.
- **xmm/m32** — An XMM register or a 32-bit memory operand. The 128-bit XMM registers are XMM0 through XMM7; XMM8 through XMM15 are available using REX.R in 64-bit mode. The contents of memory are found at the address provided by the effective address computation.
- **xmm/m64** — An XMM register or a 64-bit memory operand. The 128-bit SIMD floating-point registers are XMM0 through XMM7; XMM8 through XMM15 are available using REX.R in 64-bit mode. The contents of memory are found at the address provided by the effective address computation.
- **xmm/m128** — An XMM register or a 128-bit memory operand. The 128-bit XMM registers are XMM0 through XMM7; XMM8 through XMM15 are available using REX.R in 64-bit mode. The contents of memory are found at the address provided by the effective address computation.
- **<XMM0>** — Indicates implied use of the XMM0 register.

When there is ambiguity, **xmm1** indicates the first source operand using an XMM register and **xmm2** the second source operand using an XMM register.

Some instructions use the XMM0 register as the third source operand, indicated by **<XMM0>**. The use of the third XMM register operand is implicit in the instruction encoding and does not affect the ModR/M encoding.

- **ymm** — A YMM register. The 256-bit YMM registers are: YMM0 through YMM7; YMM8 through YMM15 are available in 64-bit mode.
- **m256** — A 32-byte operand in memory. This nomenclature is used only with AVX instructions.
- **ymm/m256** — A YMM register or 256-bit memory operand.
- **<YMM0>** — Indicates use of the YMM0 register as an implicit argument.
- **bnd** — A 128-bit bounds register. BND0 through BND3.
- **mib** — A memory operand using SIB addressing form, where the index register is not used in address calculation, Scale is ignored. Only the base and displacement are used in effective address calculation.
- **m512** — A 64-byte operand in memory.
- **zmm/m512** — A ZMM register or 512-bit memory operand.
- **{k1}{z}** — A mask register used as instruction writemask. The 64-bit k registers are: k1 through k7. Writemask specification is available exclusively via EVEX prefix. The masking can either be done as a merging-masking, where the old values are preserved for masked out elements or as a zeroing masking. The type of masking is determined by using the EVEX.z bit.
- **{k1}** — Without {z}: a mask register used as instruction writemask for instructions that do not allow zeroing-masking but support merging-masking. This corresponds to instructions that require the value of the aaa field to be different than 0 (e.g., gather) and store-type instructions which allow only merging-masking.
- **k1** — A mask register used as a regular operand (either destination or source). The 64-bit k registers are: k0 through k7.
- **mV** — A vector memory operand; the operand size is dependent on the instruction.
- **vm32{x,y,z}** — A vector array of memory operands specified using VSIB memory addressing. The array of memory addresses are specified using a common base register, a constant scale factor, and a vector index register with individual elements of 32-bit index value in an XMM register (vm32x), a YMM register (vm32y) or a ZMM register (vm32z).
- **vm64{x,y,z}** — A vector array of memory operands specified using VSIB memory addressing. The array of memory addresses are specified using a common base register, a constant scale factor, and a vector index register with individual elements of 64-bit index value in an XMM register (vm64x), a YMM register (vm64y) or a ZMM register (vm64z).
- **zmm/m512/m32bcst** — An operand that can be a ZMM register, a 512-bit memory location or a 512-bit vector loaded from a 32-bit memory location.
- **zmm/m512/m64bcst** — An operand that can be a ZMM register, a 512-bit memory location or a 512-bit vector loaded from a 64-bit memory location.
- **<ZMM0>** — Indicates use of the ZMM0 register as an implicit argument.

- **{er}** — Indicates support for embedded rounding control, which is only applicable to the register-register form of the instruction. This also implies support for SAE (Suppress All Exceptions).
- **{sae}** — Indicates support for SAE (Suppress All Exceptions). This is used for instructions that support SAE, but do not support embedded rounding control.
- **SRC1** — Denotes the first source operand in the instruction syntax of an instruction encoded with the VEX/EVEX prefix and having two or more source operands.
- **SRC2** — Denotes the second source operand in the instruction syntax of an instruction encoded with the VEX/EVEX prefix and having two or more source operands.
- **SRC3** — Denotes the third source operand in the instruction syntax of an instruction encoded with the VEX/EVEX prefix and having three source operands.
- **SRC** — The source in a single-source instruction.
- **DST** — the destination in an instruction. This field is encoded by `reg_field`.

### 3.1.1.4 Operand Encoding Column in the Instruction Summary Table

The “operand encoding” column is abbreviated as Op/En in the Instruction Summary table heading. Instruction operand encoding information is provided for each assembly instruction syntax using a letter to cross reference to a row entry in the operand encoding definition table that follows the instruction summary table. The operand encoding table in each instruction reference page lists each instruction operand (according to each instruction syntax and operand ordering shown in the instruction column) relative to the ModRM byte, VEX.vvvv field or additional operand encoding placement.

EVEX encoded instructions employ compressed `disp8*N` encoding of the displacement bytes, where N is defined in Table 2-34 and Table 2-35, according to tuple types. The Op/En column of an EVEX encoded instruction uses an abbreviation that corresponds to the tuple type abbreviation (and may include an additional abbreviation related to ModR/M and vvvv encoding). Most EVEX encoded instructions with VEX encoded equivalent have the ModR/M and vvvv encoding order. In such cases, the Tuple abbreviation is shown and the ModR/M, vvvv encoding abbreviation may be omitted.

## NOTES

- The letters in the Op/En column of an instruction apply ONLY to the encoding definition table immediately following the instruction summary table.
- In the encoding definition table, the letter ‘r’ within a pair of parenthesis denotes the content of the operand will be read by the processor. The letter ‘w’ within a pair of parenthesis denotes the content of the operand will be updated by the processor.

### 3.1.1.5 64/32-bit Mode Column in the Instruction Summary Table

The “64/32-bit Mode” column indicates whether the opcode sequence is supported in (a) 64-bit mode or (b) the Compatibility mode and other IA-32 modes that apply in conjunction with the CPUID feature flag associated specific instruction extensions.

The 64-bit mode support is to the left of the ‘slash’ and has the following notation:

- **V** — Supported.
- **I** — Not supported.
- **N.E.** — Indicates an instruction syntax is not encodable in 64-bit mode (it may represent part of a sequence of valid instructions in other modes).
- **N.P.** — Indicates the REX prefix does not affect the legacy instruction in 64-bit mode.
- **N.I.** — Indicates the opcode is treated as a new instruction in 64-bit mode.
- **N.S.** — Indicates an instruction syntax that requires an address override prefix in 64-bit mode and is not supported. Using an address override prefix in 64-bit mode may result in model-specific execution behavior.



The Compatibility/Legacy Mode support is to the right of the 'slash' and has the following notation:

- **V** — Supported.
- **I** — Not supported.
- **N.E.** — Indicates an Intel 64 instruction mnemonics/syntax that is not encodable; the opcode sequence is not applicable as an individual instruction in compatibility mode or IA-32 mode. The opcode may represent a valid sequence of legacy IA-32 instructions.

### 3.1.1.6 CPUID Support Column in the Instruction Summary Table

The fourth column holds abbreviated CPUID feature flags (e.g., appropriate bit in CPUID.1.ECX, CPUID.1.EDX for SSE/SSE2/SSE3/SSSE3/SSE4.1/SSE4.2/AESNI/PCLMULQDQ/AVX/RDRAND support) that indicate processor support for the instruction. If the corresponding flag is '0', the instruction will #UD.

### 3.1.1.7 Description Column in the Instruction Summary Table

The "Description" column briefly explains forms of the instruction.

### 3.1.1.8 Description Section

Each instruction is then described by number of information sections. The "Description" section describes the purpose of the instructions and required operands in more detail.

Summary of terms that may be used in the description section:

- **Legacy SSE** — Refers to SSE, SSE2, SSE3, SSSE3, SSE4, AESNI, PCLMULQDQ and any future instruction sets referencing XMM registers and encoded without a VEX prefix.
- **VEX.vvvv** — The VEX bit field specifying a source or destination register (in 1's complement form).
- **rm\_field** — shorthand for the ModR/M *r/m* field and any REX.B
- **reg\_field** — shorthand for the ModR/M *reg* field and any REX.R

### 3.1.1.9 Operation Section

The "Operation" section contains an algorithm description (frequently written in pseudo-code) for the instruction. Algorithms are composed of the following elements:

- Comments are enclosed within the symbol pairs "(" and ")".
- Compound statements are enclosed in keywords, such as: IF, THEN, ELSE and FI for an if statement; DO and OD for a do statement; or CASE... OF for a case statement.
- A register name implies the contents of the register. A register name enclosed in brackets implies the contents of the location whose address is contained in that register. For example, ES:[DI] indicates the contents of the location whose ES segment relative address is in register DI. [SI] indicates the contents of the address contained in register SI relative to the SI register's default segment (DS) or the overridden segment.
- Parentheses around the "E" in a general-purpose register name, such as (E)SI, indicates that the offset is read from the SI register if the address-size attribute is 16, from the ESI register if the address-size attribute is 32. Parentheses around the "R" in a general-purpose register name, (R)SI, in the presence of a 64-bit register definition such as (R)SI, indicates that the offset is read from the 64-bit RSI register if the address-size attribute is 64.
- Brackets are used for memory operands where they mean that the contents of the memory location is a segment-relative offset. For example, [SRC] indicates that the content of the source operand is a segment-relative offset.
- $A \leftarrow B$  indicates that the value of B is assigned to A.
- The symbols =, ≠, >, <, ≥, and ≤ are relational operators used to compare two values: meaning equal, not equal, greater or equal, less or equal, respectively. A relational expression such as  $A = B$  is TRUE if the value of A is equal to B; otherwise it is FALSE.

- The expression “« COUNT” and “» COUNT” indicates that the destination operand should be shifted left or right by the number of bits indicated by the count operand.

The following identifiers are used in the algorithmic descriptions:

- **OperandSize and AddressSize** — The OperandSize identifier represents the operand-size attribute of the instruction, which is 16, 32 or 64-bits. The AddressSize identifier represents the address-size attribute, which is 16, 32 or 64-bits. For example, the following pseudo-code indicates that the operand-size attribute depends on the form of the MOV instruction used.

```

IF Instruction = MOVW
    THEN OperandSize ← 16;
ELSE
    IF Instruction = MOVD
        THEN OperandSize ← 32;
    ELSE
        IF Instruction = MOVQ
            THEN OperandSize ← 64;
        FI;
    FI;
FI;
    
```

See “Operand-Size and Address-Size Attributes” in Chapter 3 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for guidelines on how these attributes are determined.

- **StackAddrSize** — Represents the stack address-size attribute associated with the instruction, which has a value of 16, 32 or 64-bits. See “Address-Size Attribute for Stack” in Chapter 6, “Procedure Calls, Interrupts, and Exceptions,” of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*.
- **SRC** — Represents the source operand.
- **DEST** — Represents the destination operand.
- **VLMAX** — The maximum vector register width pertaining to the instruction. This is not the vector-length encoding in the instruction’s prefix but is instead determined by the current value of XCR0. For existing processors, VLMAX is 256 whenever XCR0.YMM[bit 2] is 1. Future processors may defined new bits in XCR0 whose setting may imply other values for VLMAX.

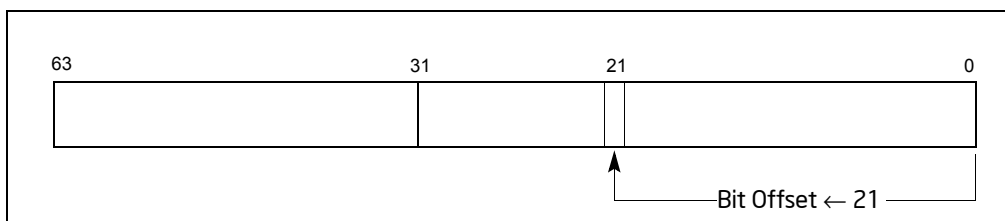
**VLMAX Definition**

XCR0 Component	VLMAX
XCR0.YMM	256

The following functions are used in the algorithmic descriptions:

- **ZeroExtend(value)** — Returns a value zero-extended to the operand-size attribute of the instruction. For example, if the operand-size attribute is 32, zero extending a byte value of -10 converts the byte from F6H to a doubleword value of 00000F6H. If the value passed to the ZeroExtend function and the operand-size attribute are the same size, ZeroExtend returns the value unaltered.
- **SignExtend(value)** — Returns a value sign-extended to the operand-size attribute of the instruction. For example, if the operand-size attribute is 32, sign extending a byte containing the value -10 converts the byte from F6H to a doubleword value of FFFFFFF6H. If the value passed to the SignExtend function and the operand-size attribute are the same size, SignExtend returns the value unaltered.
- **SaturateSignedWordToSignedByte** — Converts a signed 16-bit value to a signed 8-bit value. If the signed 16-bit value is less than -128, it is represented by the saturated value -128 (80H); if it is greater than 127, it is represented by the saturated value 127 (7FH).
- **SaturateSignedDwordToSignedWord** — Converts a signed 32-bit value to a signed 16-bit value. If the signed 32-bit value is less than -32768, it is represented by the saturated value -32768 (8000H); if it is greater than 32767, it is represented by the saturated value 32767 (7FFFH).

- **SaturateSignedWordToUnsignedByte** — Converts a signed 16-bit value to an unsigned 8-bit value. If the signed 16-bit value is less than zero, it is represented by the saturated value zero (00H); if it is greater than 255, it is represented by the saturated value 255 (FFH).
- **SaturateToSignedByte** — Represents the result of an operation as a signed 8-bit value. If the result is less than -128, it is represented by the saturated value -128 (80H); if it is greater than 127, it is represented by the saturated value 127 (7FH).
- **SaturateToSignedWord** — Represents the result of an operation as a signed 16-bit value. If the result is less than -32768, it is represented by the saturated value -32768 (8000H); if it is greater than 32767, it is represented by the saturated value 32767 (7FFFH).
- **SaturateToUnsignedByte** — Represents the result of an operation as a signed 8-bit value. If the result is less than zero it is represented by the saturated value zero (00H); if it is greater than 255, it is represented by the saturated value 255 (FFH).
- **SaturateToUnsignedWord** — Represents the result of an operation as a signed 16-bit value. If the result is less than zero it is represented by the saturated value zero (00H); if it is greater than 65535, it is represented by the saturated value 65535 (FFFFH).
- **LowOrderWord(DEST \* SRC)** — Multiplies a word operand by a word operand and stores the least significant word of the doubleword result in the destination operand.
- **HighOrderWord(DEST \* SRC)** — Multiplies a word operand by a word operand and stores the most significant word of the doubleword result in the destination operand.
- **Push(value)** — Pushes a value onto the stack. The number of bytes pushed is determined by the operand-size attribute of the instruction. See the “Operation” subsection of the “PUSH—Push Word, Doubleword or Quadword Onto the Stack” section in Chapter 4 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2B*.
- **Pop()** — removes the value from the top of the stack and returns it. The statement  $EAX \leftarrow \text{Pop}()$ ; assigns to EAX the 32-bit value from the top of the stack. Pop will return either a word, a doubleword or a quadword depending on the operand-size attribute. See the “Operation” subsection in the “POP—Pop a Value from the Stack” section of Chapter 4 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2B*.
- **PopRegisterStack** — Marks the FPU ST(0) register as empty and increments the FPU register stack pointer (TOP) by 1.
- **Switch-Tasks** — Performs a task switch.
- **Bit(BitBase, BitOffset)** — Returns the value of a bit within a bit string. The bit string is a sequence of bits in memory or a register. Bits are numbered from low-order to high-order within registers and within memory bytes. If the BitBase is a register, the BitOffset can be in the range 0 to [15, 31, 63] depending on the mode and register size. See Figure 3-1: the function  $\text{Bit}[RAX, 21]$  is illustrated.



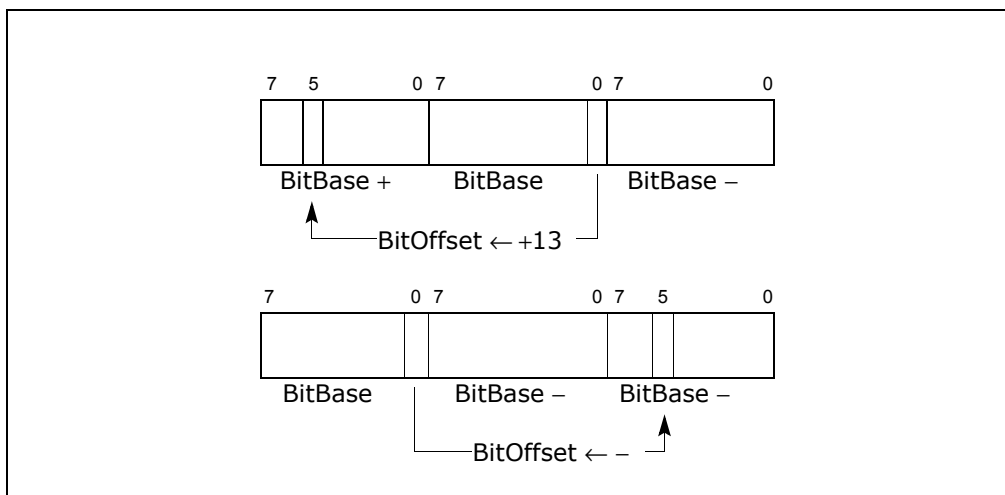
**Figure 3-1. Bit Offset for BIT[RAX, 21]**

If BitBase is a memory address, the BitOffset can range has different ranges depending on the operand size (see Table 3-2).

**Table 3-2. Range of Bit Positions Specified by Bit Offset Operands**

Operand Size	Immediate BitOffset	Register BitOffset
16	0 to 15	$-2^{15}$ to $2^{15} - 1$
32	0 to 31	$-2^{31}$ to $2^{31} - 1$
64	0 to 63	$-2^{63}$ to $2^{63} - 1$

The addressed bit is numbered (Offset MOD 8) within the byte at address (BitBase + (BitOffset DIV 8)) where DIV is signed division with rounding towards negative infinity and MOD returns a positive number (see Figure 3-2).



**Figure 3-2. Memory Bit Indexing**

### 3.1.1.10 Intel® C/C++ Compiler Intrinsics Equivalents Section

The Intel C/C++ compiler intrinsic functions give access to the full power of the Intel Architecture Instruction Set, while allowing the compiler to optimize register allocation and instruction scheduling for faster execution. Most of these functions are associated with a single IA instruction, although some may generate multiple instructions or different instructions depending upon how they are used. In particular, these functions are used to invoke instructions that perform operations on vector registers that can hold multiple data elements. These SIMD instructions use the following data types.

- `__m128`, `__m256` and `__m512` can represent 4, 8 or 16 packed single-precision floating-point values, and are used with the vector registers and SSE, AVX, or AVX-512 instruction set extension families. The `__m128` data type is also used with various single-precision floating-point scalar instructions that perform calculations using only the lowest 32 bits of a vector register; the remaining bits of the result come from one of the sources or are set to zero depending upon the instruction.
- `__m128d`, `__m256d` and `__m512d` can represent 2, 4 or 8 packed double-precision floating-point values, and are used with the vector registers and SSE, AVX, or AVX-512 instruction set extension families. The `__m128d` data type is also used with various double-precision floating-point scalar instructions that perform calculations using only the lowest 64 bits of a vector register; the remaining bits of the result come from one of the sources or are set to zero depending upon the instruction.
- `__m128i`, `__m256i` and `__m512i` can represent integer data in bytes, words, doublewords, quadwords, and occasionally larger data types.

Each of these data types incorporates in its name the number of bits it can hold. For example, the `__m128` type holds 128 bits, and because each single-precision floating-point value is 32 bits long the `__m128` type holds (128/32) or four values. Normally the compiler will allocate memory for these data types on an even multiple of the size of the type. Such aligned memory locations may be faster to read and write than locations at other addresses.

These SIMD data types are not basic Standard C data types or C++ objects, so they may be used only with the assignment operator, passed as function arguments, and returned from a function call. If you access the internal members of these types directly, or indirectly by using them in a union, there may be side effects affecting optimization, so it is recommended to use them only with the SIMD instruction intrinsic functions described in this manual or the Intel C/C++ compiler documentation.

Many intrinsic functions names are prefixed with an indicator of the vector length and suffixed by an indicator of the vector element data type, although some functions do not follow the rules below. The prefixes are:

- `_mm_` indicates that the function operates on 128-bit (or sometimes 64-bit) vectors.
- `_mm256_` indicates the function operates on 256-bit vectors.
- `_mm512_` indicates that the function operates on 512-bit vectors.

The suffixes include:

- `_ps`, which indicates a function that operates on packed single-precision floating-point data. Packed single-precision floating-point data corresponds to arrays of the C/C++ type `float` with either 4, 8 or 16 elements. Values of this type can be loaded from an array using the `_mm_loadu_ps`, `_mm256_loadu_ps`, or `_mm512_loadu_ps` functions, or created from individual values using `_mm_set_ps`, `_mm256_set_ps`, or `_mm512_set_ps` functions, and they can be stored in an array using `_mm_storeu_ps`, `_mm256_storeu_ps`, or `_mm512_storeu_ps`.
- `_ss`, which indicates a function that operates on scalar single-precision floating-point data. Single-precision floating-point data corresponds to the C/C++ type `float`, and values of type `float` can be converted to type `__m128` for use with these functions using the `_mm_set_ss` function, and converted back using the `_mm_cvtss_f32` function. When used with functions that operate on packed single-precision floating-point data the scalar element corresponds with the first packed value.
- `_pd`, which indicates a function that operates on packed double-precision floating-point data. Packed double-precision floating-point data corresponds to arrays of the C/C++ type `double` with either 2, 4, or 8 elements. Values of this type can be loaded from an array using the `_mm_loadu_pd`, `_mm256_loadu_pd`, or `_mm512_loadu_pd` functions, or created from individual values using `_mm_set_pd`, `_mm256_set_pd`, or `_mm512_set_pd` functions, and they can be stored in an array using `_mm_storeu_pd`, `_mm256_storeu_pd`, or `_mm512_storeu_pd`.
- `_sd`, which indicates a function that operates on scalar double-precision floating-point data. Double-precision floating-point data corresponds to the C/C++ type `double`, and values of type `double` can be converted to type `__m128d` for use with these functions using the `_mm_set_sd` function, and converted back using the `_mm_cvtsd_f64` function. When used with functions that operate on packed double-precision floating-point data the scalar element corresponds with the first packed value.
- `_epi8`, which indicates a function that operates on packed 8-bit signed integer values. Packed 8-bit signed integers correspond to an array of `signed char` with 16, 32 or 64 elements. Values of this type can be created from individual elements using `_mm_set_epi8`, `_mm256_set_epi8`, or `_mm512_set_epi8` functions.
- `_epi16`, which indicates a function that operates on packed 16-bit signed integer values. Packed 16-bit signed integers correspond to an array of `short` with 8, 16 or 32 elements. Values of this type can be created from individual elements using `_mm_set_epi16`, `_mm256_set_epi16`, or `_mm512_set_epi16` functions.
- `_epi32`, which indicates a function that operates on packed 32-bit signed integer values. Packed 32-bit signed integers correspond to an array of `int` with 4, 8 or 16 elements. Values of this type can be created from individual elements using `_mm_set_epi32`, `_mm256_set_epi32`, or `_mm512_set_epi32` functions.
- `_epi64`, which indicates a function that operates on packed 64-bit signed integer values. Packed 64-bit signed integers correspond to an array of `long long` (or `long` if it is a 64-bit data type) with 2, 4 or 8 elements. Values of this type can be created from individual elements using `_mm_set_epi32`, `_mm256_set_epi32`, or `_mm512_set_epi32` functions.
- `_epu8`, which indicates a function that operates on packed 8-bit unsigned integer values. Packed 8-bit unsigned integers correspond to an array of `unsigned char` with 16, 32 or 64 elements.

- `_epu16`, which indicates a function that operates on packed 16-bit unsigned integer values. Packed 16-bit unsigned integers correspond to an array of *unsigned short* with 8, 16 or 32 elements.
- `_epu32`, which indicates a function that operates on packed 32-bit unsigned integer values. Packed 32-bit unsigned integers correspond to an array of *unsigned* with 4, 8 or 16 elements.
- `_epu64`, which indicates a function that operates on packed 64-bit unsigned integer values. Packed 64-bit unsigned integers correspond to an array of *unsigned long long* (or *unsigned long* if it is a 64-bit data type) with 2, 4 or 8 elements.
- `_si128`, which indicates a function that operates on a single 128-bit value of type `__m128i`.
- `_si256`, which indicates a function that operates on a single a 256-bit value of type `__m256i`.
- `_si512`, which indicates a function that operates on a single a 512-bit value of type `__m512i`.

Values of any packed integer type can be loaded from an array using the `_mm_loadu_si128`, `_mm256_loadu_si256`, or `_mm512_loadu_si512` functions, and they can be stored in an array using `_mm_storeu_si128`, `_mm256_storeu_si256`, or `_mm512_storeu_si512`.

These functions and data types are used with the SSE, AVX, and AVX-512 instruction set extension families. In addition there are similar functions that correspond to MMX instructions. These are less frequently used because they require additional state management, and only operate on 64-bit packed integer values.

The declarations of Intel C/C++ compiler intrinsic functions may reference some non-standard data types, such as `__int64`. The C Standard header `stdint.h` defines similar platform-independent types, and the documentation for that header gives characteristics that apply to corresponding non-standard types according to the following table.

**Table 3-3. Standard and Non-standard Data Types**

Non-standard Type	Standard Type (from <code>stdint.h</code> )
<code>__int64</code>	<code>int64_t</code>
<code>unsigned __int64</code>	<code>uint64_t</code>
<code>__int32</code>	<code>int32_t</code>
<code>unsigned __int32</code>	<code>uint32_t</code>
<code>__int16</code>	<code>int16_t</code>
<code>unsigned __int16</code>	<code>uint16_t</code>

For a more detailed description of each intrinsic function and additional information related to its usage, refer to the online Intel Intrinsics Guide, <https://software.intel.com/sites/landingpage/IntrinsicsGuide>.

### 3.1.1.11 Flags Affected Section

The “Flags Affected” section lists the flags in the EFLAGS register that are affected by the instruction. When a flag is cleared, it is equal to 0; when it is set, it is equal to 1. The arithmetic and logical instructions usually assign values to the status flags in a uniform manner (see Appendix A, “EFLAGS Cross-Reference,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*). Non-conventional assignments are described in the “Operation” section. The values of flags listed as **undefined** may be changed by the instruction in an indeterminate manner. Flags that are not listed are unchanged by the instruction.

### 3.1.1.12 FPU Flags Affected Section

The floating-point instructions have an “FPU Flags Affected” section that describes how each instruction can affect the four condition code flags of the FPU status word.

### 3.1.1.13 Protected Mode Exceptions Section

The “Protected Mode Exceptions” section lists the exceptions that can occur when the instruction is executed in protected mode and the reasons for the exceptions. Each exception is given a mnemonic that consists of a pound

sign (#) followed by two letters and an optional error code in parentheses. For example, #GP(0) denotes a general protection exception with an error code of 0. Table 3-4 associates each two-letter mnemonic with the corresponding exception vector and name. See Chapter 6, “Procedure Calls, Interrupts, and Exceptions,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*, for a detailed description of the exceptions.

Application programmers should consult the documentation provided with their operating systems to determine the actions taken when exceptions occur.

**Table 3-4. Intel 64 and IA-32 General Exceptions**

Vector	Name	Source	Protected Mode <sup>1</sup>	Real Address Mode	Virtual 8086 Mode
0	#DE—Divide Error	DIV and IDIV instructions.	Yes	Yes	Yes
1	#DB—Debug	Any code or data reference.	Yes	Yes	Yes
3	#BP—Breakpoint	INT 3 instruction.	Yes	Yes	Yes
4	#OF—Overflow	INTO instruction.	Yes	Yes	Yes
5	#BR—BOUND Range Exceeded	BOUND instruction.	Yes	Yes	Yes
6	#UD—Invalid Opcode (Undefined Opcode)	UD2 instruction or reserved opcode.	Yes	Yes	Yes
7	#NM—Device Not Available (No Math Coprocessor)	Floating-point or WAIT/FWAIT instruction.	Yes	Yes	Yes
8	#DF—Double Fault	Any instruction that can generate an exception, an NMI, or an INTR.	Yes	Yes	Yes
10	#TS—Invalid TSS	Task switch or TSS access.	Yes	Reserved	Yes
11	#NP—Segment Not Present	Loading segment registers or accessing system segments.	Yes	Reserved	Yes
12	#SS—Stack Segment Fault	Stack operations and SS register loads.	Yes	Yes	Yes
13	#GP—General Protection <sup>2</sup>	Any memory reference and other protection checks.	Yes	Yes	Yes
14	#PF—Page Fault	Any memory reference.	Yes	Reserved	Yes
16	#MF—Floating-Point Error (Math Fault)	Floating-point or WAIT/FWAIT instruction.	Yes	Yes	Yes
17	#AC—Alignment Check	Any data reference in memory.	Yes	Reserved	Yes
18	#MC—Machine Check	Model dependent machine check errors.	Yes	Yes	Yes
19	#XM—SIMD Floating-Point Numeric Error	SSE/SSE2/SSE3 floating-point instructions.	Yes	Yes	Yes

**NOTES:**

1. Apply to protected mode, compatibility mode, and 64-bit mode.
2. In the real-address mode, vector 13 is the segment overrun exception.

### 3.1.1.14 Real-Address Mode Exceptions Section

The “Real-Address Mode Exceptions” section lists the exceptions that can occur when the instruction is executed in real-address mode (see Table 3-4).

### 3.1.1.15 Virtual-8086 Mode Exceptions Section

The “Virtual-8086 Mode Exceptions” section lists the exceptions that can occur when the instruction is executed in virtual-8086 mode (see Table 3-4).



### 3.1.1.16 Floating-Point Exceptions Section

The “Floating-Point Exceptions” section lists exceptions that can occur when an x87 FPU floating-point instruction is executed. All of these exception conditions result in a floating-point error exception (#MF, exception 16) being generated. Table 3-5 associates a one- or two-letter mnemonic with the corresponding exception name. See “Floating-Point Exception Conditions” in Chapter 8 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for a detailed description of these exceptions.

**Table 3-5. x87 FPU Floating-Point Exceptions**

Mnemonic	Name	Source
#IS #IA	Floating-point invalid operation: - Stack overflow or underflow - Invalid arithmetic operation	- x87 FPU stack overflow or underflow - Invalid FPU arithmetic operation
#Z	Floating-point divide-by-zero	Divide-by-zero
#D	Floating-point denormal operand	Source operand that is a denormal number
#O	Floating-point numeric overflow	Overflow in result
#U	Floating-point numeric underflow	Underflow in result
#P	Floating-point inexact result (precision)	Inexact result (precision)

### 3.1.1.17 SIMD Floating-Point Exceptions Section

The “SIMD Floating-Point Exceptions” section lists exceptions that can occur when an SSE/SSE2/SSE3 floating-point instruction is executed. All of these exception conditions result in a SIMD floating-point error exception (#XM, exception 19) being generated. Table 3-6 associates a one-letter mnemonic with the corresponding exception name. For a detailed description of these exceptions, refer to “SSE and SSE2 Exceptions”, in Chapter 11 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*.

**Table 3-6. SIMD Floating-Point Exceptions**

Mnemonic	Name	Source
#I	Floating-point invalid operation	Invalid arithmetic operation or source operand
#Z	Floating-point divide-by-zero	Divide-by-zero
#D	Floating-point denormal operand	Source operand that is a denormal number
#O	Floating-point numeric overflow	Overflow in result
#U	Floating-point numeric underflow	Underflow in result
#P	Floating-point inexact result	Inexact result (precision)

### 3.1.1.18 Compatibility Mode Exceptions Section

This section lists exceptions that occur within compatibility mode.

### 3.1.1.19 64-Bit Mode Exceptions Section

This section lists exceptions that occur within 64-bit mode.



## 3.2 INSTRUCTIONS (A-L)

The remainder of this chapter provides descriptions of Intel 64 and IA-32 instructions (A-L). See also: Chapter 4, “Instruction Set Reference, M-U,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2B*, and Chapter 5, “Instruction Set Reference, V-Z,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2C*.

## ADDPD—Add Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 58 /r ADDPD xmm1, xmm2/m128	RM	V/V	SSE2	Add packed double-precision floating-point values from xmm2/mem to xmm1 and store result in xmm1.
VEEX.NDS.128.66.0F.WIG 58 /r VADDPD xmm1,xmm2, xmm3/m128	RVM	V/V	AVX	Add packed double-precision floating-point values from xmm3/mem to xmm2 and store result in xmm1.
VEEX.NDS.256.66.0F.WIG 58 /r VADDPD ymm1, ymm2, ymm3/m256	RVM	V/V	AVX	Add packed double-precision floating-point values from ymm3/mem to ymm2 and store result in ymm1.
EVEX.NDS.128.66.0F.W1 58 /r VADDPD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	FV	V/V	AVX512VL AVX512F	Add packed double-precision floating-point values from xmm3/m128/m64bcst to xmm2 and store result in xmm1 with writemask k1.
EVEX.NDS.256.66.0F.W1 58 /r VADDPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	FV	V/V	AVX512VL AVX512F	Add packed double-precision floating-point values from ymm3/m256/m64bcst to ymm2 and store result in ymm1 with writemask k1.
EVEX.NDS.512.66.0F.W1 58 /r VADDPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	FV	V/V	AVX512F	Add packed double-precision floating-point values from zmm3/m512/m64bcst to zmm2 and store result in zmm1 with writemask k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEEX.vvvv	ModRM:r/m (r)	NA
FV-RVM	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

### Description

Add two, four or eight packed double-precision floating-point values from the first source operand to the second source operand, and stores the packed double-precision floating-point results in the destination operand.

EVEX encoded versions: The first source 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 64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

VEEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAX\_VL-1:256) of the corresponding ZMM register destination are zeroed.

VEEX.128 encoded version: the first source operand is a XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAX\_VL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper Bits (MAX\_VL-1:128) of the corresponding ZMM register destination are unmodified.

### Operation

**VADDPD (EVEX encoded versions) when src2 operand is a vector register**

(KL, VL) = (2, 128), (4, 256), (8, 512)

IF (VL = 512) AND (EVEX.b = 1)

THEN

SET\_RM(EVEX.RC);

ELSE

```

    SET_RM(MXCSR.RM);
FI;
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← SRC1[i+63:i] + SRC2[i+63:i]
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+63:i] remains unchanged*
        ELSE ; zeroing-masking
            DEST[i+63:i] ← 0
        FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VADDPD (EVEX encoded versions) when src2 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] ← SRC1[i+63:i] + SRC2[63:0]
                ELSE
                    DEST[i+63:i] ← SRC1[i+63:i] + SRC2[i+63:i]
            FI;
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+63:i] ← 0
            FI
        FI;
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VADDPD (VEX.256 encoded version)**

```

DEST[63:0] ← SRC1[63:0] + SRC2[63:0]
DEST[127:64] ← SRC1[127:64] + SRC2[127:64]
DEST[191:128] ← SRC1[191:128] + SRC2[191:128]
DEST[255:192] ← SRC1[255:192] + SRC2[255:192]
DEST[MAX_VL-1:256] ← 0

```

**VADDPD (VEX.128 encoded version)**

DEST[63:0] ← SRC1[63:0] + SRC2[63:0]  
 DEST[127:64] ← SRC1[127:64] + SRC2[127:64]  
 DEST[MAX\_VL-1:128] ← 0

**ADDPD (128-bit Legacy SSE version)**

DEST[63:0] ← DEST[63:0] + SRC[63:0]  
 DEST[127:64] ← DEST[127:64] + SRC[127:64]  
 DEST[MAX\_VL-1:128] (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**

VADDPD \_\_m512d \_\_mm512\_add\_pd (\_\_m512d a, \_\_m512d b);  
 VADDPD \_\_m512d \_\_mm512\_mask\_add\_pd (\_\_m512d s, \_\_mmask8 k, \_\_m512d a, \_\_m512d b);  
 VADDPD \_\_m512d \_\_mm512\_maskz\_add\_pd (\_\_mmask8 k, \_\_m512d a, \_\_m512d b);  
 VADDPD \_\_m256d \_\_mm256\_mask\_add\_pd (\_\_m256d s, \_\_mmask8 k, \_\_m256d a, \_\_m256d b);  
 VADDPD \_\_m256d \_\_mm256\_maskz\_add\_pd (\_\_mmask8 k, \_\_m256d a, \_\_m256d b);  
 VADDPD \_\_m128d \_\_mm\_mask\_add\_pd (\_\_m128d s, \_\_mmask8 k, \_\_m128d a, \_\_m128d b);  
 VADDPD \_\_m128d \_\_mm\_maskz\_add\_pd (\_\_mmask8 k, \_\_m128d a, \_\_m128d b);  
 VADDPD \_\_m512d \_\_mm512\_add\_round\_pd (\_\_m512d a, \_\_m512d b, int);  
 VADDPD \_\_m512d \_\_mm512\_mask\_add\_round\_pd (\_\_m512d s, \_\_mmask8 k, \_\_m512d a, \_\_m512d b, int);  
 VADDPD \_\_m512d \_\_mm512\_maskz\_add\_round\_pd (\_\_mmask8 k, \_\_m512d a, \_\_m512d b, int);  
 ADDPD \_\_m256d \_\_mm256\_add\_pd (\_\_m256d a, \_\_m256d b);  
 ADDPD \_\_m128d \_\_mm\_add\_pd (\_\_m128d a, \_\_m128d b);

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal

**Other Exceptions**

VEX-encoded instruction, see Exceptions Type 2.

EVEX-encoded instruction, see Exceptions Type E2.

## ADDPS—Add Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
OF 58 /r ADDPS xmm1, xmm2/m128	RM	V/V	SSE	Add packed single-precision floating-point values from xmm2/m128 to xmm1 and store result in xmm1.
VEX.NDS.128.OF.WIG 58 /r VADDPS xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Add packed single-precision floating-point values from xmm3/m128 to xmm2 and store result in xmm1.
VEX.NDS.256.OF.WIG 58 /r VADDPS ymm1, ymm2, ymm3/m256	RVM	V/V	AVX	Add packed single-precision floating-point values from ymm3/m256 to ymm2 and store result in ymm1.
EVEX.NDS.128.OF.W0 58 /r VADDPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	FV	V/V	AVX512VL AVX512F	Add packed single-precision floating-point values from xmm3/m128/m32bcst to xmm2 and store result in xmm1 with writemask k1.
EVEX.NDS.256.OF.W0 58 /r VADDPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	FV	V/V	AVX512VL AVX512F	Add packed single-precision floating-point values from ymm3/m256/m32bcst to ymm2 and store result in ymm1 with writemask k1.
EVEX.NDS.512.OF.W0 58 /r VADDPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst {er}	FV	V/V	AVX512F	Add packed single-precision floating-point values from zmm3/m512/m32bcst to zmm2 and store result in zmm1 with writemask k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
FV-RVM	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

### Description

Add four, eight or sixteen packed single-precision floating-point values from the first source operand with the second source operand, and stores the packed single-precision floating-point results in the destination operand.

EVEX encoded versions: The first source 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 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAX\_VL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: the first source operand is a XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAX\_VL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper Bits (MAX\_VL-1:128) of the corresponding ZMM register destination are unmodified.

### Operation

**VADDPS (EVEX encoded versions) when src2 operand is a register**

(KL, VL) = (4, 128), (8, 256), (16, 512)

IF (VL = 512) AND (EVEX.b = 1)

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

```

FI;
FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ← SRC1[i+31:i] + SRC2[i+31:i]
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+31:i] remains unchanged*
    ELSE ; zeroing-masking
      DEST[i+31:i] ← 0
    FI
  FI;
ENDFOR;
DEST[MAX_VL-1:VL] ← 0

```

**VADDPS (EVEX encoded versions) when src2 operand is a memory source**  
 (KL, VL) = (4, 128), (8, 256), (16, 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] ← SRC1[i+31:i] + SRC2[31:0]
        ELSE
          DEST[i+31:i] ← SRC1[i+31:i] + SRC2[i+31:i]
        FI;
      ELSE
        IF *merging-masking* ; merging-masking
          THEN *DEST[i+31:i] remains unchanged*
        ELSE ; zeroing-masking
          DEST[i+31:i] ← 0
        FI
      FI;
    FI;
  FI;
ENDFOR;
DEST[MAX_VL-1:VL] ← 0

```

**VADDPS (VEX.256 encoded version)**

$\text{DEST}[31:0] \leftarrow \text{SRC1}[31:0] + \text{SRC2}[31:0]$   
 $\text{DEST}[63:32] \leftarrow \text{SRC1}[63:32] + \text{SRC2}[63:32]$   
 $\text{DEST}[95:64] \leftarrow \text{SRC1}[95:64] + \text{SRC2}[95:64]$   
 $\text{DEST}[127:96] \leftarrow \text{SRC1}[127:96] + \text{SRC2}[127:96]$   
 $\text{DEST}[159:128] \leftarrow \text{SRC1}[159:128] + \text{SRC2}[159:128]$   
 $\text{DEST}[191:160] \leftarrow \text{SRC1}[191:160] + \text{SRC2}[191:160]$   
 $\text{DEST}[223:192] \leftarrow \text{SRC1}[223:192] + \text{SRC2}[223:192]$   
 $\text{DEST}[255:224] \leftarrow \text{SRC1}[255:224] + \text{SRC2}[255:224]$   
 $\text{DEST}[\text{MAX\_VL}-1:256] \leftarrow 0$

**VADDPS (VEX.128 encoded version)**

$\text{DEST}[31:0] \leftarrow \text{SRC1}[31:0] + \text{SRC2}[31:0]$   
 $\text{DEST}[63:32] \leftarrow \text{SRC1}[63:32] + \text{SRC2}[63:32]$   
 $\text{DEST}[95:64] \leftarrow \text{SRC1}[95:64] + \text{SRC2}[95:64]$   
 $\text{DEST}[127:96] \leftarrow \text{SRC1}[127:96] + \text{SRC2}[127:96]$   
 $\text{DEST}[\text{MAX\_VL}-1:128] \leftarrow 0$

**ADDPS (128-bit Legacy SSE version)**

$\text{DEST}[31:0] \leftarrow \text{SRC1}[31:0] + \text{SRC2}[31:0]$   
 $\text{DEST}[63:32] \leftarrow \text{SRC1}[63:32] + \text{SRC2}[63:32]$   
 $\text{DEST}[95:64] \leftarrow \text{SRC1}[95:64] + \text{SRC2}[95:64]$   
 $\text{DEST}[127:96] \leftarrow \text{SRC1}[127:96] + \text{SRC2}[127:96]$   
 $\text{DEST}[\text{MAX\_VL}-1:128]$  (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**

**VADDPS** `__m512 _mm512_add_ps (__m512 a, __m512 b);`  
**VADDPS** `__m512 _mm512_mask_add_ps (__m512 s, __mmask16 k, __m512 a, __m512 b);`  
**VADDPS** `__m512 _mm512_maskz_add_ps (__mmask16 k, __m512 a, __m512 b);`  
**VADDPS** `__m256 _mm256_mask_add_ps (__m256 s, __mmask8 k, __m256 a, __m256 b);`  
**VADDPS** `__m256 _mm256_maskz_add_ps (__mmask8 k, __m256 a, __m256 b);`  
**VADDPS** `__m128 _mm_mask_add_ps (__m128d s, __mmask8 k, __m128 a, __m128 b);`  
**VADDPS** `__m128 _mm_maskz_add_ps (__mmask8 k, __m128 a, __m128 b);`  
**VADDPS** `__m512 _mm512_add_round_ps (__m512 a, __m512 b, int);`  
**VADDPS** `__m512 _mm512_mask_add_round_ps (__m512 s, __mmask16 k, __m512 a, __m512 b, int);`  
**VADDPS** `__m512 _mm512_maskz_add_round_ps (__mmask16 k, __m512 a, __m512 b, int);`  
**ADDPS** `__m256 _mm256_add_ps (__m256 a, __m256 b);`  
**ADDPS** `__m128 _mm_add_ps (__m128 a, __m128 b);`

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal

**Other Exceptions**

VEX-encoded instruction, see Exceptions Type 2.

EVEX-encoded instruction, see Exceptions Type E2.

## ADDSD—Add Scalar Double-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 58 /r ADDSD xmm1, xmm2/m64	RM	V/V	SSE2	Add the low double-precision floating-point value from xmm2/mem to xmm1 and store the result in xmm1.
VEX.NDS.128.F2.0F.WIG 58 /r VADDSD xmm1, xmm2, xmm3/m64	RVM	V/V	AVX	Add the low double-precision floating-point value from xmm3/mem to xmm2 and store the result in xmm1.
EVEX.NDS.LIG.F2.0F.W1 58 /r VADDSD xmm1 {k1}{z}, xmm2, xmm3/m64{er}	T1S	V/V	AVX512F	Add the low double-precision floating-point value from xmm3/m64 to xmm2 and store the result in xmm1 with writemask k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
T1S-RVM	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

### Description

Adds the low double-precision floating-point values from the second source operand and the first source operand and stores the double-precision floating-point result in the destination operand.

The second source operand can be an XMM register or a 64-bit memory location. The first source and destination operands are XMM registers.

128-bit Legacy SSE version: The first source and destination operands are the same. Bits (MAX\_VL-1:64) of the corresponding destination register remain unchanged.

EVEX and VEX.128 encoded version: The first source operand is encoded by EVEX.vvvv/VEX.vvvv. Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (MAX\_VL-1:128) of the destination register are zeroed.

EVEX version: The low quadword element of the destination is updated according to the writemask.

Software should ensure VADDSD is encoded with VEX.L=0. Encoding VADDSD with VEX.L=1 may encounter unpredictable behavior across different processor generations.

### Operation

#### VADDSD (EVEX encoded version)

IF (EVEX.b = 1) AND SRC2 \*is a register\*

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

IF k1[0] or \*no writemask\*

THEN DEST[63:0] ← SRC1[63:0] + SRC2[63:0]

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[63:0] remains unchanged\*

ELSE ; zeroing-masking

THEN DEST[63:0] ← 0

FI;

FI;

DEST[127:64] ← SRC1[127:64]



DEST[MAX\_VL-1:128] ← 0

**VADDSD (VEX.128 encoded version)**

DEST[63:0] ← SRC1[63:0] + SRC2[63:0]

DEST[127:64] ← SRC1[127:64]

DEST[MAX\_VL-1:128] ← 0

**ADDSD (128-bit Legacy SSE version)**

DEST[63:0] ← DEST[63:0] + SRC[63:0]

DEST[MAX\_VL-1:64] (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**

VADDSD \_\_m128d \_mm\_mask\_add\_sd (\_\_m128d s, \_\_mmask8 k, \_\_m128d a, \_\_m128d b);

VADDSD \_\_m128d \_mm\_maskz\_add\_sd (\_\_mmask8 k, \_\_m128d a, \_\_m128d b);

VADDSD \_\_m128d \_mm\_add\_round\_sd (\_\_m128d a, \_\_m128d b, int);

VADDSD \_\_m128d \_mm\_mask\_add\_round\_sd (\_\_m128d s, \_\_mmask8 k, \_\_m128d a, \_\_m128d b, int);

VADDSD \_\_m128d \_mm\_maskz\_add\_round\_sd (\_\_mmask8 k, \_\_m128d a, \_\_m128d b, int);

ADDSD \_\_m128d \_mm\_add\_sd (\_\_m128d a, \_\_m128d b);

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal

**Other Exceptions**

VEX-encoded instruction, see Exceptions Type 3.

EVEX-encoded instruction, see Exceptions Type E3.

## ADDSS—Add Scalar Single-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 58 /r ADDSS xmm1, xmm2/m32	RM	V/V	SSE	Add the low single-precision floating-point value from xmm2/mem to xmm1 and store the result in xmm1.
VEX.NDS.128.F3.0F.WIG 58 /r VADDSS xmm1,xmm2, xmm3/m32	RVM	V/V	AVX	Add the low single-precision floating-point value from xmm3/mem to xmm2 and store the result in xmm1.
EVEX.NDS.LIG.F3.0F.W0 58 /r VADDSS xmm1{k1}{z}, xmm2, xmm3/m32{er}	T1S	V/V	AVX512F	Add the low single-precision floating-point value from xmm3/m32 to xmm2 and store the result in xmm1 with writemask k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
T1S	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

### Description

Adds the low single-precision floating-point values from the second source operand and the first source operand, and stores the double-precision floating-point result in the destination operand.

The second source operand can be an XMM register or a 64-bit memory location. The first source and destination operands are XMM registers.

128-bit Legacy SSE version: The first source and destination operands are the same. Bits (MAX\_VL-1:32) of the corresponding the destination register remain unchanged.

EVEX and VEX.128 encoded version: The first source operand is encoded by EVEX.vvvv/VEX.vvvv. Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (MAX\_VL-1:128) of the destination register are zeroed.

EVEX version: The low doubleword element of the destination is updated according to the writemask.

Software should ensure VADDSS is encoded with VEX.L=0. Encoding VADDSS with VEX.L=1 may encounter unpredictable behavior across different processor generations.

### Operation

#### VADDSS (EVEX encoded versions)

IF (EVEX.b = 1) AND SRC2 \*is a register\*

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

IF k1[0] or \*no writemask\*

THEN DEST[31:0] ← SRC1[31:0] + SRC2[31:0]

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[31:0] remains unchanged\*

ELSE ; zeroing-masking

THEN DEST[31:0] ← 0

FI;

FI;

DEST[127:32] ← SRC1[127:32]

DEST[MAX\_VL-1:128] ← 0

**VADDSS DEST, SRC1, SRC2 (VEX.128 encoded version)**

DEST[31:0] ← SRC1[31:0] + SRC2[31:0]

DEST[127:32] ← SRC1[127:32]

DEST[MAX\_VL-1:128] ← 0

**ADDSS DEST, SRC (128-bit Legacy SSE version)**

DEST[31:0] ← DEST[31:0] + SRC[31:0]

DEST[MAX\_VL-1:32] (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**

VADDSS \_\_m128 \_mm\_mask\_add\_ss (\_\_m128 s, \_\_mmask8 k, \_\_m128 a, \_\_m128 b);

VADDSS \_\_m128 \_mm\_maskz\_add\_ss (\_\_mmask8 k, \_\_m128 a, \_\_m128 b);

VADDSS \_\_m128 \_mm\_add\_round\_ss (\_\_m128 a, \_\_m128 b, int);

VADDSS \_\_m128 \_mm\_mask\_add\_round\_ss (\_\_m128 s, \_\_mmask8 k, \_\_m128 a, \_\_m128 b, int);

VADDSS \_\_m128 \_mm\_maskz\_add\_round\_ss (\_\_mmask8 k, \_\_m128 a, \_\_m128 b, int);

ADDSS \_\_m128 \_mm\_add\_ss (\_\_m128 a, \_\_m128 b);

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal

**Other Exceptions**

VEX-encoded instruction, see Exceptions Type 3.

EVEX-encoded instruction, see Exceptions Type E3.

## ANDPD—Bitwise Logical AND of Packed Double Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 54 /r ANDPD xmm1, xmm2/m128	RM	V/V	SSE2	Return the bitwise logical AND of packed double-precision floating-point values in xmm1 and xmm2/mem.
VEX.NDS.128.66.0F 54 /r VANDPD xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Return the bitwise logical AND of packed double-precision floating-point values in xmm2 and xmm3/mem.
VEX.NDS.256.66.0F 54 /r VANDPD ymm1, ymm2, ymm3/m256	RVM	V/V	AVX	Return the bitwise logical AND of packed double-precision floating-point values in ymm2 and ymm3/mem.
EVEX.NDS.128.66.0F.W1 54 /r VANDPD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	FV	V/V	AVX512VL AVX512DQ	Return the bitwise logical AND of packed double-precision floating-point values in xmm2 and xmm3/m128/m64bcst subject to writemask k1.
EVEX.NDS.256.66.0F.W1 54 /r VANDPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	FV	V/V	AVX512VL AVX512DQ	Return the bitwise logical AND of packed double-precision floating-point values in ymm2 and ymm3/m256/m64bcst subject to writemask k1.
EVEX.NDS.512.66.0F.W1 54 /r VANDPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	FV	V/V	AVX512DQ	Return the bitwise logical AND of packed double-precision floating-point values in zmm2 and zmm3/m512/m64bcst subject to writemask k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
FV	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

### Description

Performs a bitwise logical AND of the two, four or eight packed double-precision floating-point values from the first source operand and the second source operand, and stores the result in the destination operand.

EVEX encoded versions: The first source 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 64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAX\_VL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAX\_VL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAX\_VL-1:128) of the corresponding register destination are unmodified.

## Operation

### VANDPD (EVEX encoded versions)

(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) AND (SRC2 *is memory*)
        THEN
          DEST[i+63:i] ← SRC1[i+63:i] BITWISE AND SRC2[63:0]
        ELSE
          DEST[i+63:i] ← SRC1[i+63:i] BITWISE AND SRC2[i+63:i]
        FI;
      ELSE
        IF *merging-masking*           ; merging-masking
          THEN *DEST[i+63:i] remains unchanged*
        ELSE                             ; zeroing-masking
          DEST[i+63:i] = 0
        FI;
      FI;
    ENDFOR
  DEST[MAX_VL-1:VL] ← 0

```

### VANDPD (VEX.256 encoded version)

```

DEST[63:0] ← SRC1[63:0] BITWISE AND SRC2[63:0]
DEST[127:64] ← SRC1[127:64] BITWISE AND SRC2[127:64]
DEST[191:128] ← SRC1[191:128] BITWISE AND SRC2[191:128]
DEST[255:192] ← SRC1[255:192] BITWISE AND SRC2[255:192]
DEST[MAX_VL-1:256] ← 0

```

### VANDPD (VEX.128 encoded version)

```

DEST[63:0] ← SRC1[63:0] BITWISE AND SRC2[63:0]
DEST[127:64] ← SRC1[127:64] BITWISE AND SRC2[127:64]
DEST[MAX_VL-1:128] ← 0

```

### ANDPD (128-bit Legacy SSE version)

```

DEST[63:0] ← DEST[63:0] BITWISE AND SRC[63:0]
DEST[127:64] ← DEST[127:64] BITWISE AND SRC[127:64]
DEST[MAX_VL-1:128] (Unmodified)

```

## Intel C/C++ Compiler Intrinsic Equivalent

```

VANDPD __m512d __mm512_and_pd (__m512d a, __m512d b);
VANDPD __m512d __mm512_mask_and_pd (__m512d s, __mmask8 k, __m512d a, __m512d b);
VANDPD __m512d __mm512_maskz_and_pd (__mmask8 k, __m512d a, __m512d b);
VANDPD __m256d __mm256_mask_and_pd (__m256d s, __mmask8 k, __m256d a, __m256d b);
VANDPD __m256d __mm256_maskz_and_pd (__mmask8 k, __m256d a, __m256d b);
VANDPD __m128d __mm_mask_and_pd (__m128d s, __mmask8 k, __m128d a, __m128d b);
VANDPD __m128d __mm_maskz_and_pd (__mmask8 k, __m128d a, __m128d b);
VANDPD __m256d __mm256_and_pd (__m256d a, __m256d b);
ANDPD __m128d __mm_and_pd (__m128d a, __m128d b);

```

## SIMD Floating-Point Exceptions

None



### Other Exceptions

VEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.

## ANDPS—Bitwise Logical AND of Packed Single Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
OF 54 /r ANDPS xmm1, xmm2/m128	RM	V/V	SSE	Return the bitwise logical AND of packed single-precision floating-point values in xmm1 and xmm2/mem.
VEX.NDS.128.0F 54 /r VANDPS xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Return the bitwise logical AND of packed single-precision floating-point values in xmm2 and xmm3/mem.
VEX.NDS.256.0F 54 /r VANDPS ymm1, ymm2, ymm3/m256	RVM	V/V	AVX	Return the bitwise logical AND of packed single-precision floating-point values in ymm2 and ymm3/mem.
EVEX.NDS.128.0F.W0 54 /r VANDPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	FV	V/V	AVX512VL AVX512DQ	Return the bitwise logical AND of packed single-precision floating-point values in xmm2 and xmm3/m128/m32bcst subject to writemask k1.
EVEX.NDS.256.0F.W0 54 /r VANDPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	FV	V/V	AVX512VL AVX512DQ	Return the bitwise logical AND of packed single-precision floating-point values in ymm2 and ymm3/m256/m32bcst subject to writemask k1.
EVEX.NDS.512.0F.W0 54 /r VANDPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	FV	V/V	AVX512DQ	Return the bitwise logical AND of packed single-precision floating-point values in zmm2 and zmm3/m512/m32bcst subject to writemask k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
FV	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

### Description

Performs a bitwise logical AND of the four, eight or sixteen packed single-precision floating-point values from the first source operand and the second source operand, and stores the result in the destination operand.

EVEX encoded versions: The first source 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 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAX\_VL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAX\_VL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAX\_VL-1:128) of the corresponding ZMM register destination are unmodified.

**Operation****VANDPS (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\*

IF (EVEX.b == 1) AND (SRC2 \*is memory\*)

THEN

DEST[i+63:i] ← SRC1[i+31:i] BITWISE AND SRC2[31:0]

ELSE

DEST[i+31:i] ← SRC1[i+31:i] BITWISE AND SRC2[i+31:i]

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+31:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+31:i] ← 0

FI;

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0;

**VANDPS (VEX.256 encoded version)**

DEST[31:0] ← SRC1[31:0] BITWISE AND SRC2[31:0]

DEST[63:32] ← SRC1[63:32] BITWISE AND SRC2[63:32]

DEST[95:64] ← SRC1[95:64] BITWISE AND SRC2[95:64]

DEST[127:96] ← SRC1[127:96] BITWISE AND SRC2[127:96]

DEST[159:128] ← SRC1[159:128] BITWISE AND SRC2[159:128]

DEST[191:160] ← SRC1[191:160] BITWISE AND SRC2[191:160]

DEST[223:192] ← SRC1[223:192] BITWISE AND SRC2[223:192]

DEST[255:224] ← SRC1[255:224] BITWISE AND SRC2[255:224].

DEST[MAX\_VL-1:256] ← 0;

**VANDPS (VEX.128 encoded version)**

DEST[31:0] ← SRC1[31:0] BITWISE AND SRC2[31:0]

DEST[63:32] ← SRC1[63:32] BITWISE AND SRC2[63:32]

DEST[95:64] ← SRC1[95:64] BITWISE AND SRC2[95:64]

DEST[127:96] ← SRC1[127:96] BITWISE AND SRC2[127:96]

DEST[MAX\_VL-1:128] ← 0;

**ANDPS (128-bit Legacy SSE version)**

DEST[31:0] ← DEST[31:0] BITWISE AND SRC[31:0]

DEST[63:32] ← DEST[63:32] BITWISE AND SRC[63:32]

DEST[95:64] ← DEST[95:64] BITWISE AND SRC[95:64]

DEST[127:96] ← DEST[127:96] BITWISE AND SRC[127:96]

DEST[MAX\_VL-1:128] (Unmodified)



### Intel C/C++ Compiler Intrinsic Equivalent

VANDPS \_\_m512 \_\_mm512\_and\_ps (\_\_m512 a, \_\_m512 b);  
 VANDPS \_\_m512 \_\_mm512\_mask\_and\_ps (\_\_m512 s, \_\_mmask16 k, \_\_m512 a, \_\_m512 b);  
 VANDPS \_\_m512 \_\_mm512\_maskz\_and\_ps (\_\_mmask16 k, \_\_m512 a, \_\_m512 b);  
 VANDPS \_\_m256 \_\_mm256\_mask\_and\_ps (\_\_m256 s, \_\_mmask8 k, \_\_m256 a, \_\_m256 b);  
 VANDPS \_\_m256 \_\_mm256\_maskz\_and\_ps (\_\_mmask8 k, \_\_m256 a, \_\_m256 b);  
 VANDPS \_\_m128 \_\_mm\_mask\_and\_ps (\_\_m128 s, \_\_mmask8 k, \_\_m128 a, \_\_m128 b);  
 VANDPS \_\_m128 \_\_mm\_maskz\_and\_ps (\_\_mmask8 k, \_\_m128 a, \_\_m128 b);  
 VANDPS \_\_m256 \_\_mm256\_and\_ps (\_\_m256 a, \_\_m256 b);  
 ANDPS \_\_m128 \_\_mm\_and\_ps (\_\_m128 a, \_\_m128 b);

### SIMD Floating-Point Exceptions

None

### Other Exceptions

VEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.

## ANDNPD—Bitwise Logical AND NOT of Packed Double Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 55 /r ANDNPD xmm1, xmm2/m128	RM	V/V	SSE2	Return the bitwise logical AND NOT of packed double-precision floating-point values in xmm1 and xmm2/mem.
VEX.NDS.128.66.0F 55 /r VANDNPD xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Return the bitwise logical AND NOT of packed double-precision floating-point values in xmm2 and xmm3/mem.
VEX.NDS.256.66.0F 55/r VANDNPD ymm1, ymm2, ymm3/m256	RVM	V/V	AVX	Return the bitwise logical AND NOT of packed double-precision floating-point values in ymm2 and ymm3/mem.
EVEX.NDS.128.66.0F.W1 55 /r VANDNPD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	FV	V/V	AVX512VL AVX512DQ	Return the bitwise logical AND NOT of packed double-precision floating-point values in xmm2 and xmm3/m128/m64bcst subject to writemask k1.
EVEX.NDS.256.66.0F.W1 55 /r VANDNPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	FV	V/V	AVX512VL AVX512DQ	Return the bitwise logical AND NOT of packed double-precision floating-point values in ymm2 and ymm3/m256/m64bcst subject to writemask k1.
EVEX.NDS.512.66.0F.W1 55 /r VANDNPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	FV	V/V	AVX512DQ	Return the bitwise logical AND NOT of packed double-precision floating-point values in zmm2 and zmm3/m512/m64bcst subject to writemask k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
FV	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

### Description

Performs a bitwise logical AND NOT of the two, four or eight packed double-precision floating-point values from the first source operand and the second source operand, and stores the result in the destination operand.

EVEX encoded versions: The first source 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 64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAX\_VL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAX\_VL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAX\_VL-1:128) of the corresponding register destination are unmodified.

## Operation

### VANDNPD (EVEX encoded versions)

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

  i ← j \* 64

  IF k1[j] OR \*no writemask\*

    IF (EVEX.b == 1) AND (SRC2 \*is memory\*)

      THEN

        DEST[i+63:i] ← (NOT(SRC1[i+63:i])) BITWISE AND SRC2[63:0]

      ELSE

        DEST[i+63:i] ← (NOT(SRC1[i+63:i])) BITWISE AND SRC2[i+63:i]

    FI;

  ELSE

    IF \*merging-masking\* ; merging-masking

      THEN \*DEST[i+63:i] remains unchanged\*

      ELSE ; zeroing-masking

        DEST[i+63:i] = 0

    FI;

  FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

### VANDNPD (VEX.256 encoded version)

DEST[63:0] ← (NOT(SRC1[63:0])) BITWISE AND SRC2[63:0]

DEST[127:64] ← (NOT(SRC1[127:64])) BITWISE AND SRC2[127:64]

DEST[191:128] ← (NOT(SRC1[191:128])) BITWISE AND SRC2[191:128]

DEST[255:192] ← (NOT(SRC1[255:192])) BITWISE AND SRC2[255:192]

DEST[MAX\_VL-1:256] ← 0

### VANDNPD (VEX.128 encoded version)

DEST[63:0] ← (NOT(SRC1[63:0])) BITWISE AND SRC2[63:0]

DEST[127:64] ← (NOT(SRC1[127:64])) BITWISE AND SRC2[127:64]

DEST[MAX\_VL-1:128] ← 0

### ANDNPD (128-bit Legacy SSE version)

DEST[63:0] ← (NOT(DEST[63:0])) BITWISE AND SRC[63:0]

DEST[127:64] ← (NOT(DEST[127:64])) BITWISE AND SRC[127:64]

DEST[MAX\_VL-1:128] (Unmodified)

## Intel C/C++ Compiler Intrinsic Equivalent

VANDNPD \_\_m512d \_\_mm512\_andnot\_pd (\_\_m512d a, \_\_m512d b);

VANDNPD \_\_m512d \_\_mm512\_mask\_andnot\_pd (\_\_m512d s, \_\_mmask8 k, \_\_m512d a, \_\_m512d b);

VANDNPD \_\_m512d \_\_mm512\_maskz\_andnot\_pd (\_\_mmask8 k, \_\_m512d a, \_\_m512d b);

VANDNPD \_\_m256d \_\_mm256\_mask\_andnot\_pd (\_\_m256d s, \_\_mmask8 k, \_\_m256d a, \_\_m256d b);

VANDNPD \_\_m256d \_\_mm256\_maskz\_andnot\_pd (\_\_mmask8 k, \_\_m256d a, \_\_m256d b);

VANDNPD \_\_m128d \_\_mm\_mask\_andnot\_pd (\_\_m128d s, \_\_mmask8 k, \_\_m128d a, \_\_m128d b);

VANDNPD \_\_m128d \_\_mm\_maskz\_andnot\_pd (\_\_mmask8 k, \_\_m128d a, \_\_m128d b);

VANDNPD \_\_m256d \_\_mm256\_andnot\_pd (\_\_m256d a, \_\_m256d b);

ANDNPD \_\_m128d \_\_mm\_andnot\_pd (\_\_m128d a, \_\_m128d b);

## SIMD Floating-Point Exceptions

None



### Other Exceptions

VEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.

## ANDNPS—Bitwise Logical AND NOT of Packed Single Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
OF 55 /r ANDNPS xmm1, xmm2/m128	RM	V/V	SSE	Return the bitwise logical AND NOT of packed single-precision floating-point values in xmm1 and xmm2/mem.
VEX.NDS.128.OF 55 /r VANDNPS xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Return the bitwise logical AND NOT of packed single-precision floating-point values in xmm2 and xmm3/mem.
VEX.NDS.256.OF 55 /r VANDNPS ymm1, ymm2, ymm3/m256	RVM	V/V	AVX	Return the bitwise logical AND NOT of packed single-precision floating-point values in ymm2 and ymm3/mem.
EVEX.NDS.128.OF.W0 55 /r VANDNPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	FV	V/V	AVX512VL AVX512DQ	Return the bitwise logical AND of packed single-precision floating-point values in xmm2 and xmm3/m128/m32bcst subject to writemask k1.
EVEX.NDS.256.OF.W0 55 /r VANDNPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	FV	V/V	AVX512VL AVX512DQ	Return the bitwise logical AND of packed single-precision floating-point values in ymm2 and ymm3/m256/m32bcst subject to writemask k1.
EVEX.NDS.512.OF.W0 55 /r VANDNPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	FV	V/V	AVX512DQ	Return the bitwise logical AND of packed single-precision floating-point values in zmm2 and zmm3/m512/m32bcst subject to writemask k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
FV	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

### Description

Performs a bitwise logical AND NOT of the four, eight or sixteen packed single-precision floating-point values from the first source operand and the second source operand, and stores the result in the destination operand.

EVEX encoded versions: The first source 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 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAX\_VL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAX\_VL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAX\_VL-1:128) of the corresponding ZMM register destination are unmodified.

**Operation****VANDNPS (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\*

IF (EVEX.b == 1) AND (SRC2 \*is memory\*)

THEN

DEST[i+31:i] ← (NOT(SRC1[i+31:i])) BITWISE AND SRC2[31:0]

ELSE

DEST[i+31:i] ← (NOT(SRC1[i+31:i])) BITWISE AND SRC2[i+31:i]

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+31:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+31:i] = 0

FI;

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VANDNPS (VEX.256 encoded version)**

DEST[31:0] ← (NOT(SRC1[31:0])) BITWISE AND SRC2[31:0]

DEST[63:32] ← (NOT(SRC1[63:32])) BITWISE AND SRC2[63:32]

DEST[95:64] ← (NOT(SRC1[95:64])) BITWISE AND SRC2[95:64]

DEST[127:96] ← (NOT(SRC1[127:96])) BITWISE AND SRC2[127:96]

DEST[159:128] ← (NOT(SRC1[159:128])) BITWISE AND SRC2[159:128]

DEST[191:160] ← (NOT(SRC1[191:160])) BITWISE AND SRC2[191:160]

DEST[223:192] ← (NOT(SRC1[223:192])) BITWISE AND SRC2[223:192]

DEST[255:224] ← (NOT(SRC1[255:224])) BITWISE AND SRC2[255:224].

DEST[MAX\_VL-1:256] ← 0

**VANDNPS (VEX.128 encoded version)**

DEST[31:0] ← (NOT(SRC1[31:0])) BITWISE AND SRC2[31:0]

DEST[63:32] ← (NOT(SRC1[63:32])) BITWISE AND SRC2[63:32]

DEST[95:64] ← (NOT(SRC1[95:64])) BITWISE AND SRC2[95:64]

DEST[127:96] ← (NOT(SRC1[127:96])) BITWISE AND SRC2[127:96]

DEST[MAX\_VL-1:128] ← 0

**ANDNPS (128-bit Legacy SSE version)**

DEST[31:0] ← (NOT(DEST[31:0])) BITWISE AND SRC[31:0]

DEST[63:32] ← (NOT(DEST[63:32])) BITWISE AND SRC[63:32]

DEST[95:64] ← (NOT(DEST[95:64])) BITWISE AND SRC[95:64]

DEST[127:96] ← (NOT(DEST[127:96])) BITWISE AND SRC[127:96]

DEST[MAX\_VL-1:128] (Unmodified)

### Intel C/C++ Compiler Intrinsic Equivalent

VANDNPS \_\_m512 \_\_mm512\_andnot\_ps (\_\_m512 a, \_\_m512 b);  
VANDNPS \_\_m512 \_\_mm512\_mask\_andnot\_ps (\_\_m512 s, \_\_mmask16 k, \_\_m512 a, \_\_m512 b);  
VANDNPS \_\_m512 \_\_mm512\_maskz\_andnot\_ps (\_\_mmask16 k, \_\_m512 a, \_\_m512 b);  
VANDNPS \_\_m256 \_\_mm256\_mask\_andnot\_ps (\_\_m256 s, \_\_mmask8 k, \_\_m256 a, \_\_m256 b);  
VANDNPS \_\_m256 \_\_mm256\_maskz\_andnot\_ps (\_\_mmask8 k, \_\_m256 a, \_\_m256 b);  
VANDNPS \_\_m128 \_\_mm\_mask\_andnot\_ps (\_\_m128 s, \_\_mmask8 k, \_\_m128 a, \_\_m128 b);  
VANDNPS \_\_m128 \_\_mm\_maskz\_andnot\_ps (\_\_mmask8 k, \_\_m128 a, \_\_m128 b);  
VANDNPS \_\_m256 \_\_mm256\_andnot\_ps (\_\_m256 a, \_\_m256 b);  
ANDNPS \_\_m128 \_\_mm\_andnot\_ps (\_\_m128 a, \_\_m128 b);

### SIMD Floating-Point Exceptions

None

### Other Exceptions

VEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.

**BNDLDX—Load Extended Bounds Using Address Translation**

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
OF 1A /r BNDLDX bnd, mib	RM	V/V	MPX	Load the bounds stored in a bound table entry (BTE) into bnd with address translation using the base of mib and conditional on the index of mib matching the pointer value in the BTE.

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3
RM	ModRM:reg (w)	SIB.base (r): Address of pointer SIB.index(r)	NA

**Description**

BNDLDX uses the linear address constructed from the base register and displacement of the SIB-addressing form of the memory operand (mib) to perform address translation to access a bound table entry and conditionally load the bounds in the BTE to the destination. The destination register is updated with the bounds in the BTE, if the content of the index register of mib matches the pointer value stored in the BTE.

If the pointer value comparison fails, the destination is updated with INIT bounds (lb = 0x0, ub = 0x0) (note: as articulated earlier, the upper bound is represented using 1's complement, therefore, the 0x0 value of upper bound allows for access to full memory).

This instruction does not cause memory access to the linear address of mib nor the effective address referenced by the base, and does not read or write any flags.

Segment overrides apply to the linear address computation with the base of mib, and are used during address translation to generate the address of the bound table entry. By default, the address of the BTE is assumed to be linear address. There are no segmentation checks performed on the base of mib.

The base of mib will not be checked for canonical address violation as it does not access memory.

Any encoding of this instruction that does not specify base or index register will treat those registers as zero (constant). The reg-reg form of this instruction will remain a NOP.

The scale field of the SIB byte has no effect on these instructions and is ignored.

The bound register may be partially updated on memory faults. The order in which memory operands are loaded is implementation specific.

**Operation**

$$\text{base} \leftarrow \text{mib.SIB.base} ? \text{mib.SIB.base} + \text{Disp} : 0;$$

$$\text{ptr\_value} \leftarrow \text{mib.SIB.index} ? \text{mib.SIB.index} : 0;$$
**Outside 64-bit mode**

$$\text{A\_BDE}[31:0] \leftarrow (\text{Zero\_extend32}(\text{base}[31:12] \ll 2) + (\text{BNDCFG}[31:12] \ll 12));$$

$$\text{A\_BT}[31:0] \leftarrow \text{LoadFrom}(\text{A\_BDE});$$

IF A\_BT[0] equal 0 Then

$$\text{BNDSTATUS} \leftarrow \text{A\_BDE} | 02\text{H};$$

$$\#BR;$$

FI;

$$\text{A\_BTE}[31:0] \leftarrow (\text{Zero\_extend32}(\text{base}[11:2] \ll 4) + (\text{A\_BT}[31:2] \ll 2));$$

$$\text{Temp\_lb}[31:0] \leftarrow \text{LoadFrom}(\text{A\_BTE});$$

$$\text{Temp\_ub}[31:0] \leftarrow \text{LoadFrom}(\text{A\_BTE} + 4);$$

$$\text{Temp\_ptr}[31:0] \leftarrow \text{LoadFrom}(\text{A\_BTE} + 8);$$

IF Temp\_ptr equal ptr\_value Then

$$\text{BND.LB} \leftarrow \text{Temp\_lb};$$

$$\text{BND.UB} \leftarrow \text{Temp\_ub};$$



```

ELSE
    BND.LB ← 0;
    BND.UB ← 0;
FI;

```

#### In 64-bit mode

```

A_BDE[63:0] ← (Zero_extend64(base[47+MAWA:20] << 3) + (BNDCFG[63:20] << 12));1
A_BT[63:0] ← LoadFrom(A_BDE);
IF A_BT[0] equal 0 Then
    BNDSTATUS ← A_BDE | 02H;
    #BR;
FI;
A_BTE[63:0] ← (Zero_extend64(base[19:3] << 5) + (A_BT[63:3] << 3));
Temp_lb[63:0] ← LoadFrom(A_BTE);
Temp_ub[63:0] ← LoadFrom(A_BTE + 8);
Temp_ptr[63:0] ← LoadFrom(A_BTE + 16);
IF Temp_ptr equal ptr_value Then
    BND.LB ← Temp_lb;
    BND.UB ← Temp_ub;
ELSE
    BND.LB ← 0;
    BND.UB ← 0;
FI;

```

#### Intel C/C++ Compiler Intrinsic Equivalent

BNDLDX: Generated by compiler as needed.

#### Flags Affected

None

#### Protected Mode Exceptions

#BR	If the bound directory entry is invalid.
#UD	If the LOCK prefix is used. If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled. If 67H prefix is not used and CS.D=0. If 67H prefix is used and CS.D=1.
#GP(0)	If a destination effective address of the Bound Table entry is outside the DS segment limit. If DS register contains a NULL segment selector.
#PF(fault code)	If a page fault occurs.

#### Real-Address Mode Exceptions

#UD	If the LOCK prefix is used. If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled. If 16-bit addressing is used.
#GP(0)	If a destination effective address of the Bound Table entry is outside the DS segment limit.

1. If CPL < 3, the supervisor MAWA (MAWAS) is used; this value is 0. If CPL = 3, the user MAWA (MAWAU) is used; this value is enumerated in CPUID.(EAX=07H,ECX=0H):ECX.MAWAU[bits 21:17]. See Section 17.3.1 of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*.

### Virtual-8086 Mode Exceptions

- #UD                    If the LOCK prefix is used.  
                         If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled.  
                         If 16-bit addressing is used.
- #GP(0)                If a destination effective address of the Bound Table entry is outside the DS segment limit.
- #PF(fault code)      If a page fault occurs.

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

- #BR                    If the bound directory entry is invalid.
- #UD                    If ModRM is RIP relative.  
                         If the LOCK prefix is used.  
                         If ModRM.r/m and REX encodes BND4-BND15 when Intel MPX is enabled.
- #GP(0)                If the memory address (A\_BDE or A\_BTE) is in a non-canonical form.
- #PF(fault code)      If a page fault occurs.

## BNDSTX—Store Extended Bounds Using Address Translation

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
OF 1B /r BNDSTX mib, bnd	MR	V/V	MPX	Store the bounds in bnd and the pointer value in the index register of mib to a bound table entry (BTE) with address translation using the base of mib.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3
MR	SIB.base (r): Address of pointer SIB.index(r)	ModRM:reg (r)	NA

### Description

BNDSTX uses the linear address constructed from the displacement and base register of the SIB-addressing form of the memory operand (mib) to perform address translation to store to a bound table entry. The bounds in the source operand bnd are written to the lower and upper bounds in the BTE. The content of the index register of mib is written to the pointer value field in the BTE.

This instruction does not cause memory access to the linear address of mib nor the effective address referenced by the base, and does not read or write any flags.

Segment overrides apply to the linear address computation with the base of mib, and are used during address translation to generate the address of the bound table entry. By default, the address of the BTE is assumed to be linear address. There are no segmentation checks performed on the base of mib.

The base of mib will not be checked for canonical address violation as it does not access memory.

Any encoding of this instruction that does not specify base or index register will treat those registers as zero (constant). The reg-reg form of this instruction will remain a NOP.

The scale field of the SIB byte has no effect on these instructions and is ignored.

The bound register may be partially updated on memory faults. The order in which memory operands are loaded is implementation specific.

### Operation

```
base ← mib.SIB.base ? mib.SIB.base + Disp : 0;
ptr_value ← mib.SIB.index ? mib.SIB.index : 0;
```

#### Outside 64-bit mode

```
A_BDE[31:0] ← (Zero_extend32(base[31:12] << 2) + (BNDCFG[31:12] << 12));
```

```
A_BT[31:0] ← LoadFrom(A_BDE);
```

```
IF A_BT[0] equal 0 Then
```

```
    BNDSTATUS ← A_BDE | 02H;
```

```
    #BR;
```

```
FI;
```

```
A_DEST[31:0] ← (Zero_extend32(base[11:2] << 4) + (A_BT[31:2] << 2)); // address of Bound table entry
```

```
A_DEST[8][31:0] ← ptr_value;
```

```
A_DEST[0][31:0] ← BND.LB;
```

```
A_DEST[4][31:0] ← BND.UB;
```

**In 64-bit mode**

```

A_BDE[63:0] ← (Zero_extend64(base[47+MAWA:20] << 3) + (BNDCFG[63:20] << 12));1
A_BT[63:0] ← LoadFrom(A_BDE);
IF A_BT[0] equal 0 Then
    BNDSTATUS ← A_BDE | 02H;
    #BR;
FI;
A_DEST[63:0] ← (Zero_extend64(base[19:3] << 5) + (A_BT[63:3] << 3)); // address of Bound table entry
A_DEST[16][63:0] ← ptr_value;
A_DEST[0][63:0] ← BND.LB;
A_DEST[8][63:0] ← BND.UB;

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```
BNDSTX: _bnd_store_ptr_bounds(const void **ptr_addr, const void *ptr_val);
```

**Flags Affected**

None

**Protected Mode Exceptions**

#BR	If the bound directory entry is invalid.
#UD	If the LOCK prefix is used. If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled. If 67H prefix is not used and CS.D=0. If 67H prefix is used and CS.D=1.
#GP(0)	If a destination effective address of the Bound Table entry is outside the DS segment limit. If DS register contains a NULL segment selector. If the destination operand points to a non-writable segment
#PF(fault code)	If a page fault occurs.

**Real-Address Mode Exceptions**

#UD	If the LOCK prefix is used. If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled. If 16-bit addressing is used.
#GP(0)	If a destination effective address of the Bound Table entry is outside the DS segment limit.

**Virtual-8086 Mode Exceptions**

#UD	If the LOCK prefix is used. If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled. If 16-bit addressing is used.
#GP(0)	If a destination effective address of the Bound Table entry is outside the DS segment limit.
#PF(fault code)	If a page fault occurs.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

1. If CPL < 3, the supervisor MAWA (MAWAS) is used; this value is 0. If CPL = 3, the user MAWA (MAWAU) is used; this value is enumerated in CPUID.(EAX=07H,ECX=0H):ECX.MAWAU[bits 21:17]. See Section 17.3.1 of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*.

## 64-Bit Mode Exceptions

#BR	If the bound directory entry is invalid.
#UD	If ModRM is RIP relative. If the LOCK prefix is used. If ModRM.r/m and REX encodes BND4-BND15 when Intel MPX is enabled.
#GP(0)	If the memory address (A_BDE or A_BTE) is in a non-canonical form. If the destination operand points to a non-writable segment
#PF(fault code)	If a page fault occurs.

## CMPPD—Compare Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F C2 /r ib CMPPD xmm1, xmm2/m128, imm8	RMI	V/V	SSE2	Compare packed double-precision floating-point values in xmm2/m128 and xmm1 using bits 2:0 of imm8 as a comparison predicate.
VEX.NDS.128.66.0F.WIG C2 /r ib VCMPD xmm1, xmm2, xmm3/m128, imm8	RVMI	V/V	AVX	Compare packed double-precision floating-point values in xmm3/m128 and xmm2 using bits 4:0 of imm8 as a comparison predicate.
VEX.NDS.256.66.0F.WIG C2 /r ib VCMPD ymm1, ymm2, ymm3/m256, imm8	RVMI	V/V	AVX	Compare packed double-precision floating-point values in ymm3/m256 and ymm2 using bits 4:0 of imm8 as a comparison predicate.
EVEX.NDS.128.66.0F.W1 C2 /r ib VCMPD k1 {k2}, xmm2, xmm3/m128/m64bcst, imm8	FV	V/V	AVX512VL AVX512F	Compare packed double-precision floating-point values in xmm3/m128/m64bcst and xmm2 using bits 4:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.
EVEX.NDS.256.66.0F.W1 C2 /r ib VCMPD k1 {k2}, ymm2, ymm3/m256/m64bcst, imm8	FV	V/V	AVX512VL AVX512F	Compare packed double-precision floating-point values in ymm3/m256/m64bcst and ymm2 using bits 4:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.
EVEX.NDS.512.66.0F.W1 C2 /r ib VCMPD k1 {k2}, zmm2, zmm3/m512/m64bcst{sae}, imm8	FV	V/V	AVX512F	Compare packed double-precision floating-point values in zmm3/m512/m64bcst and zmm2 using bits 4:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (r, w)	ModRM:r/m (r)	Imm8	NA
RVMI	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	Imm8
FV	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	Imm8

### Description

Performs a SIMD compare of the packed double-precision floating-point values in the second source operand and the first source operand and returns the results of the comparison to the destination operand. The comparison predicate operand (immediate byte) specifies the type of comparison performed on each pair of packed values in the two source operands.

EVEX encoded versions: The first source operand (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 64-bit memory location. The destination operand (first operand) is an opmask register. Comparison results are written to the destination operand under the writemask k2. Each comparison result is a single mask bit of 1 (comparison true) or 0 (comparison false).

VEX.256 encoded version: The first source operand (second operand) is a YMM register. The second source operand (third operand) can be a YMM register or a 256-bit memory location. The destination operand (first operand) is a YMM register. Four comparisons are performed with results written to the destination operand. The result of each comparison is a quadword mask of all 1s (comparison true) or all 0s (comparison false).

128-bit Legacy SSE version: The first source and destination operand (first operand) is an XMM register. The second source operand (second operand) can be an XMM register or 128-bit memory location. Bits (MAX\_VL-1:128) of the corresponding ZMM destination register remain unchanged. Two comparisons are performed with results written to bits 127:0 of the destination operand. The result of each comparison is a quadword mask of all 1s (comparison true) or all 0s (comparison false).

VEX.128 encoded version: The first source operand (second operand) is an XMM register. The second source operand (third operand) can be an XMM register or a 128-bit memory location. Bits (MAX\_VL-1:128) of the destination ZMM register are zeroed. Two comparisons are performed with results written to bits 127:0 of the destination operand.

The comparison predicate operand is an 8-bit immediate:

- For instructions encoded using the VEX or EVEX prefix, bits 4:0 define the type of comparison to be performed (see Table 3-1). Bits 5 through 7 of the immediate are reserved.
- For instruction encodings that do not use VEX prefix, bits 2:0 define the type of comparison to be made (see the first 8 rows of Table 3-1). Bits 3 through 7 of the immediate are reserved.

**Table 3-1. Comparison Predicate for CMPDD and CMPPS Instructions**

Predicate	imm8 Value	Description	Result: A Is 1st Operand, B Is 2nd Operand				Signals #IA on QNAN
			A > B	A < B	A = B	Unordered <sup>1</sup>	
EQ_OQ (EQ)	0H	Equal (ordered, non-signaling)	False	False	True	False	No
LT_OS (LT)	1H	Less-than (ordered, signaling)	False	True	False	False	Yes
LE_OS (LE)	2H	Less-than-or-equal (ordered, signaling)	False	True	True	False	Yes
UNORD_Q (UNORD)	3H	Unordered (non-signaling)	False	False	False	True	No
NEQ_UQ (NEQ)	4H	Not-equal (unordered, non-signaling)	True	True	False	True	No
NLT_US (NLT)	5H	Not-less-than (unordered, signaling)	True	False	True	True	Yes
NLE_US (NLE)	6H	Not-less-than-or-equal (unordered, signaling)	True	False	False	True	Yes
ORD_Q (ORD)	7H	Ordered (non-signaling)	True	True	True	False	No
EQ_UQ	8H	Equal (unordered, non-signaling)	False	False	True	True	No
NGE_US (NGE)	9H	Not-greater-than-or-equal (unordered, signaling)	False	True	False	True	Yes
NGT_US (NGT)	AH	Not-greater-than (unordered, signaling)	False	True	True	True	Yes
FALSE_OQ (FALSE)	BH	False (ordered, non-signaling)	False	False	False	False	No
NEQ_OQ	CH	Not-equal (ordered, non-signaling)	True	True	False	False	No
GE_OS (GE)	DH	Greater-than-or-equal (ordered, signaling)	True	False	True	False	Yes
GT_OS (GT)	EH	Greater-than (ordered, signaling)	True	False	False	False	Yes
TRUE_UQ (TRUE)	FH	True (unordered, non-signaling)	True	True	True	True	No
EQ_OS	10H	Equal (ordered, signaling)	False	False	True	False	Yes
LT_OQ	11H	Less-than (ordered, nonsignaling)	False	True	False	False	No
LE_OQ	12H	Less-than-or-equal (ordered, nonsignaling)	False	True	True	False	No
UNORD_S	13H	Unordered (signaling)	False	False	False	True	Yes
NEQ_US	14H	Not-equal (unordered, signaling)	True	True	False	True	Yes
NLT_UQ	15H	Not-less-than (unordered, nonsignaling)	True	False	True	True	No
NLE_UQ	16H	Not-less-than-or-equal (unordered, nonsignaling)	True	False	False	True	No
ORD_S	17H	Ordered (signaling)	True	True	True	False	Yes
EQ_US	18H	Equal (unordered, signaling)	False	False	True	True	Yes
NGE_UQ	19H	Not-greater-than-or-equal (unordered, non-signaling)	False	True	False	True	No

**Table 3-1. Comparison Predicate for CMPPD and CMPPS Instructions (Contd.)**

Predicate	imm8 Value	Description	Result: A Is 1st Operand, B Is 2nd Operand				Signals #IA on QNaN
			A > B	A < B	A = B	Unordered <sup>1</sup>	
NGT_UQ	1AH	Not-greater-than (unordered, nonsignaling)	False	True	True	True	No
FALSE_OS	1BH	False (ordered, signaling)	False	False	False	False	Yes
NEQ_OS	1CH	Not-equal (ordered, signaling)	True	True	False	False	Yes
GE_OQ	1DH	Greater-than-or-equal (ordered, nonsignaling)	True	False	True	False	No
GT_OQ	1EH	Greater-than (ordered, nonsignaling)	True	False	False	False	No
TRUE_US	1FH	True (unordered, signaling)	True	True	True	True	Yes

**NOTES:**

1. If either operand A or B is a NaN.

The unordered relationship is true when at least one of the two source operands being compared is a NaN; the ordered relationship is true when neither source operand is a NaN.

A subsequent computational instruction that uses the mask result in the destination operand as an input operand will not generate an exception, because a mask of all 0s corresponds to a floating-point value of +0.0 and a mask of all 1s corresponds to a QNaN.

Note that processors with "CPUID.1H:ECX.AVX = 0" do not implement the "greater-than", "greater-than-or-equal", "not-greater than", and "not-greater-than-or-equal relations" predicates. These comparisons can be made either by using the inverse relationship (that is, use the "not-less-than-or-equal" to make a "greater-than" comparison) or by using software emulation. When using software emulation, the program must swap the operands (copying registers when necessary to protect the data that will now be in the destination), and then perform the compare using a different predicate. The predicate to be used for these emulations is listed in the first 8 rows of Table 3-7 (*Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2A*) under the heading Emulation.

Compilers and assemblers may implement the following two-operand pseudo-ops in addition to the three-operand CMPPD instruction, for processors with "CPUID.1H:ECX.AVX = 0". See Table 3-2. Compiler should treat reserved Imm8 values as illegal syntax.

**Table 3-2. Pseudo-Op and CMPPD Implementation**

Pseudo-Op	CMPPD Implementation
CMPEQPD <i>xmm1, xmm2</i>	CMPPD <i>xmm1, xmm2, 0</i>
CMPLTPD <i>xmm1, xmm2</i>	CMPPD <i>xmm1, xmm2, 1</i>
CMPLDPD <i>xmm1, xmm2</i>	CMPPD <i>xmm1, xmm2, 2</i>
CMPUNORDPD <i>xmm1, xmm2</i>	CMPPD <i>xmm1, xmm2, 3</i>
CMPNEQPD <i>xmm1, xmm2</i>	CMPPD <i>xmm1, xmm2, 4</i>
CMPNLTPD <i>xmm1, xmm2</i>	CMPPD <i>xmm1, xmm2, 5</i>
CMPNLEPD <i>xmm1, xmm2</i>	CMPPD <i>xmm1, xmm2, 6</i>
CMPORDPD <i>xmm1, xmm2</i>	CMPPD <i>xmm1, xmm2, 7</i>

The greater-than relations that the processor does not implement require more than one instruction to emulate in software and therefore should not be implemented as pseudo-ops. (For these, the programmer should reverse the operands of the corresponding less than relations and use move instructions to ensure that the mask is moved to the correct destination register and that the source operand is left intact.)

Processors with "CPUID.1H:ECX.AVX = 1" implement the full complement of 32 predicates shown in Table 3-3, software emulation is no longer needed. Compilers and assemblers may implement the following three-operand pseudo-ops in addition to the four-operand VCMPPD instruction. See Table 3-3, where the notations of reg1 reg2, and reg3 represent either XMM registers or YMM registers. Compiler should treat reserved Imm8 values as illegal



syntax. Alternately, intrinsics can map the pseudo-ops to pre-defined constants to support a simpler intrinsic interface. Compilers and assemblers may implement three-operand pseudo-ops for EVEX encoded VCMPPD instructions in a similar fashion by extending the syntax listed in Table 3-3.

**Table 3-3. Pseudo-Op and VCMPPD Implementation**

<b>Pseudo-Op</b>	<b>CMPPD Implementation</b>
VCMPEQPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 0</i>
VCMPLTPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 1</i>
VCMLEPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 2</i>
VCMUNORDPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 3</i>
VCMNEQPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 4</i>
VCMNLTDP <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 5</i>
VCMNLEPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 6</i>
VCMORDPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 7</i>
VCMPEQ_UQPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 8</i>
VCMNGEPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 9</i>
VCMNGTPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 0AH</i>
VCMFALSEPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 0BH</i>
VCMNEQ_OQPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 0CH</i>
VCMGEPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 0DH</i>
VCMGTPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 0EH</i>
VCMTRUEPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 0FH</i>
VCMPEQ_OSPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 10H</i>
VCMPLT_OQPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 11H</i>
VCMLE_OQPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 12H</i>
VCMUNORD_SPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 13H</i>
VCMNEQ_USPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 14H</i>
VCMNLT_UQPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 15H</i>
VCMNLE_UQPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 16H</i>
VCMORD_SPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 17H</i>
VCMPEQ_USPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 18H</i>
VCMNGE_UQPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 19H</i>
VCMNGT_UQPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 1AH</i>
VCMFALSE_OSPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 1BH</i>
VCMNEQ_OSPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 1CH</i>
VCMGE_OQPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 1DH</i>
VCMGT_OQPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 1EH</i>
VCMTRUE_USPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 1FH</i>

## Operation

CASE (COMPARISON PREDICATE) OF

0: OP3  $\leftarrow$  EQ\_OQ; OP5  $\leftarrow$  EQ\_OQ;  
1: OP3  $\leftarrow$  LT\_OS; OP5  $\leftarrow$  LT\_OS;  
2: OP3  $\leftarrow$  LE\_OS; OP5  $\leftarrow$  LE\_OS;  
3: OP3  $\leftarrow$  UNORD\_Q; OP5  $\leftarrow$  UNORD\_Q;  
4: OP3  $\leftarrow$  NEQ\_UQ; OP5  $\leftarrow$  NEQ\_UQ;  
5: OP3  $\leftarrow$  NLT\_US; OP5  $\leftarrow$  NLT\_US;  
6: OP3  $\leftarrow$  NLE\_US; OP5  $\leftarrow$  NLE\_US;  
7: OP3  $\leftarrow$  ORD\_Q; OP5  $\leftarrow$  ORD\_Q;  
8: OP5  $\leftarrow$  EQ\_UQ;  
9: OP5  $\leftarrow$  NGE\_US;  
10: OP5  $\leftarrow$  NGT\_US;  
11: OP5  $\leftarrow$  FALSE\_OQ;  
12: OP5  $\leftarrow$  NEQ\_OQ;  
13: OP5  $\leftarrow$  GE\_OS;  
14: OP5  $\leftarrow$  GT\_OS;  
15: OP5  $\leftarrow$  TRUE\_UQ;  
16: OP5  $\leftarrow$  EQ\_OS;  
17: OP5  $\leftarrow$  LT\_OQ;  
18: OP5  $\leftarrow$  LE\_OQ;  
19: OP5  $\leftarrow$  UNORD\_S;  
20: OP5  $\leftarrow$  NEQ\_US;  
21: OP5  $\leftarrow$  NLT\_UQ;  
22: OP5  $\leftarrow$  NLE\_UQ;  
23: OP5  $\leftarrow$  ORD\_S;  
24: OP5  $\leftarrow$  EQ\_US;  
25: OP5  $\leftarrow$  NGE\_UQ;  
26: OP5  $\leftarrow$  NGT\_UQ;  
27: OP5  $\leftarrow$  FALSE\_OS;  
28: OP5  $\leftarrow$  NEQ\_OS;  
29: OP5  $\leftarrow$  GE\_OQ;  
30: OP5  $\leftarrow$  GT\_OQ;  
31: OP5  $\leftarrow$  TRUE\_US;  
DEFAULT: Reserved;  
ESAC;

**VCMPDP (EVEX encoded versions)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 64
  IF k2[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1) AND (SRC2 *is memory*)
        THEN
          CMP ← SRC1[i+63:i] OP5 SRC2[63:0]
        ELSE
          CMP ← SRC1[i+63:i] OP5 SRC2[j+63:i]
      FI;
      IF CMP = TRUE
        THEN DEST[j] ← 1;
        ELSE DEST[j] ← 0; FI;
    ELSE DEST[j] ← 0 ; zeroing-masking only
  FI;
ENDFOR
DEST[MAX_KL-1:KL] ← 0

```

**VCMPDP (VEX.256 encoded version)**

```

CMP0 ← SRC1[63:0] OP5 SRC2[63:0];
CMP1 ← SRC1[127:64] OP5 SRC2[127:64];
CMP2 ← SRC1[191:128] OP5 SRC2[191:128];
CMP3 ← SRC1[255:192] OP5 SRC2[255:192];
IF CMP0 = TRUE
  THEN DEST[63:0] ← FFFFFFFFFFFFFFFFH;
  ELSE DEST[63:0] ← 0000000000000000H; FI;
IF CMP1 = TRUE
  THEN DEST[127:64] ← FFFFFFFFFFFFFFFFH;
  ELSE DEST[127:64] ← 0000000000000000H; FI;
IF CMP2 = TRUE
  THEN DEST[191:128] ← FFFFFFFFFFFFFFFFH;
  ELSE DEST[191:128] ← 0000000000000000H; FI;
IF CMP3 = TRUE
  THEN DEST[255:192] ← FFFFFFFFFFFFFFFFH;
  ELSE DEST[255:192] ← 0000000000000000H; FI;
DEST[MAX_VL-1:256] ← 0

```

**VCMPDP (VEX.128 encoded version)**

```

CMP0 ← SRC1[63:0] OP5 SRC2[63:0];
CMP1 ← SRC1[127:64] OP5 SRC2[127:64];
IF CMP0 = TRUE
  THEN DEST[63:0] ← FFFFFFFFFFFFFFFFH;
  ELSE DEST[63:0] ← 0000000000000000H; FI;
IF CMP1 = TRUE
  THEN DEST[127:64] ← FFFFFFFFFFFFFFFFH;
  ELSE DEST[127:64] ← 0000000000000000H; FI;
DEST[MAX_VL-1:128] ← 0

```

**CMPPD (128-bit Legacy SSE version)**

```

CMP0 ← SRC1[63:0] OP3 SRC2[63:0];
CMP1 ← SRC1[127:64] OP3 SRC2[127:64];
IF CMP0 = TRUE
    THEN DEST[63:0] ← FFFFFFFFFFFFFFFFH;
    ELSE DEST[63:0] ← 0000000000000000H; FI;
IF CMP1 = TRUE
    THEN DEST[127:64] ← FFFFFFFFFFFFFFFFH;
    ELSE DEST[127:64] ← 0000000000000000H; FI;
DEST[MAX_VL-1:128] (Unmodified)

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VCMPD __mmask8 __mm512_cmp_pd_mask( __m512d a, __m512d b, int imm);
VCMPD __mmask8 __mm512_cmp_round_pd_mask( __m512d a, __m512d b, int imm, int sae);
VCMPD __mmask8 __mm512_mask_cmp_pd_mask( __mmask8 k1, __m512d a, __m512d b, int imm);
VCMPD __mmask8 __mm512_mask_cmp_round_pd_mask( __mmask8 k1, __m512d a, __m512d b, int imm, int sae);
VCMPD __mmask8 __mm256_cmp_pd_mask( __m256d a, __m256d b, int imm);
VCMPD __mmask8 __mm256_mask_cmp_pd_mask( __mmask8 k1, __m256d a, __m256d b, int imm);
VCMPD __mmask8 __mm_cmp_pd_mask( __m128d a, __m128d b, int imm);
VCMPD __mmask8 __mm_mask_cmp_pd_mask( __mmask8 k1, __m128d a, __m128d b, int imm);
VCMPD __m256 __mm256_cmp_pd(__m256d a, __m256d b, int imm)
(V)VCMPD __m128 __mm_cmp_pd(__m128d a, __m128d b, int imm)

```

**SIMD Floating-Point Exceptions**

Invalid if SNaN operand and invalid if QNaN and predicate as listed in Table 3-1.

Denormal

**Other Exceptions**

VEX-encoded instructions, see Exceptions Type 2.

EVEX-encoded instructions, see Exceptions Type E2.

## CMPPS—Compare Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
OF C2 /r ib CMPPS xmm1, xmm2/m128, imm8	RMI	V/V	SSE	Compare packed single-precision floating-point values in xmm2/m128 and xmm1 using bits 2:0 of imm8 as a comparison predicate.
VEEX.NDS.128.OF.WIG C2 /r ib VCMPPS xmm1, xmm2, xmm3/m128, imm8	RVMI	V/V	AVX	Compare packed single-precision floating-point values in xmm3/m128 and xmm2 using bits 4:0 of imm8 as a comparison predicate.
VEEX.NDS.256.OF.WIG C2 /r ib VCMPPS ymm1, ymm2, ymm3/m256, imm8	RVMI	V/V	AVX	Compare packed single-precision floating-point values in ymm3/m256 and ymm2 using bits 4:0 of imm8 as a comparison predicate.
EVEX.NDS.128.OF.WO C2 /r ib VCMPPS k1 {k2}, xmm2, xmm3/m128/m32bcst, imm8	FV	V/V	AVX512VL AVX512F	Compare packed single-precision floating-point values in xmm3/m128/m32bcst and xmm2 using bits 4:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.
EVEX.NDS.256.OF.WO C2 /r ib VCMPPS k1 {k2}, ymm2, ymm3/m256/m32bcst, imm8	FV	V/V	AVX512VL AVX512F	Compare packed single-precision floating-point values in ymm3/m256/m32bcst and ymm2 using bits 4:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.
EVEX.NDS.512.OF.WO C2 /r ib VCMPPS k1 {k2}, zmm2, zmm3/m512/m32bcst{sae}, imm8	FV	V/V	AVX512F	Compare packed single-precision floating-point values in zmm3/m512/m32bcst and zmm2 using bits 4:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (r, w)	ModRM:r/m (r)	Imm8	NA
RVMI	ModRM:reg (w)	VEEX.vvvv	ModRM:r/m (r)	Imm8
FV	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	Imm8

### Description

Performs a SIMD compare of the packed single-precision floating-point values in the second source operand and the first source operand and returns the results of the comparison to the destination operand. The comparison predicate operand (immediate byte) specifies the type of comparison performed on each of the pairs of packed values.

EVEX encoded versions: The first source operand (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 32-bit memory location. The destination operand (first operand) is an opmask register. Comparison results are written to the destination operand under the writemask k2. Each comparison result is a single mask bit of 1 (comparison true) or 0 (comparison false).

VEEX.256 encoded version: The first source operand (second operand) is a YMM register. The second source operand (third operand) can be a YMM register or a 256-bit memory location. The destination operand (first operand) is a YMM register. Eight comparisons are performed with results written to the destination operand. The result of each comparison is a doubleword mask of all 1s (comparison true) or all 0s (comparison false).

128-bit Legacy SSE version: The first source and destination operand (first operand) is an XMM register. The second source operand (second operand) can be an XMM register or 128-bit memory location. Bits (MAX\_VL-1:128) of the corresponding ZMM destination register remain unchanged. Four comparisons are performed with results written to bits 127:0 of the destination operand. The result of each comparison is a doubleword mask of all 1s (comparison true) or all 0s (comparison false).

VEX.128 encoded version: The first source operand (second operand) is an XMM register. The second source operand (third operand) can be an XMM register or a 128-bit memory location. Bits (MAX\_VL-1:128) of the destination ZMM register are zeroed. Four comparisons are performed with results written to bits 127:0 of the destination operand.

The comparison predicate operand is an 8-bit immediate:

- For instructions encoded using the VEX prefix and EVEX prefix, bits 4:0 define the type of comparison to be performed (see Table 3-1). Bits 5 through 7 of the immediate are reserved.
- For instruction encodings that do not use VEX prefix, bits 2:0 define the type of comparison to be made (see the first 8 rows of Table 3-1). Bits 3 through 7 of the immediate are reserved.

The unordered relationship is true when at least one of the two source operands being compared is a NaN; the ordered relationship is true when neither source operand is a NaN.

A subsequent computational instruction that uses the mask result in the destination operand as an input operand will not generate an exception, because a mask of all 0s corresponds to a floating-point value of +0.0 and a mask of all 1s corresponds to a QNaN.

Note that processors with "CPUID.1H:ECX.AVX = 0" do not implement the "greater-than", "greater-than-or-equal", "not-greater than", and "not-greater-than-or-equal" predicates. These comparisons can be made either by using the inverse relationship (that is, use the "not-less-than-or-equal" to make a "greater-than" comparison) or by using software emulation. When using software emulation, the program must swap the operands (copying registers when necessary to protect the data that will now be in the destination), and then perform the compare using a different predicate. The predicate to be used for these emulations is listed in the first 8 rows of Table 3-7 (*Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2A*) under the heading Emulation.

Compilers and assemblers may implement the following two-operand pseudo-ops in addition to the three-operand CMPPS instruction, for processors with "CPUID.1H:ECX.AVX = 0". See Table 3-4. Compiler should treat reserved Imm8 values as illegal syntax.

**Table 3-4. Pseudo-Op and CMPPS Implementation**

Pseudo-Op	CMPPS Implementation
CMPEQPS <i>xmm1, xmm2</i>	CMPPS <i>xmm1, xmm2, 0</i>
CMPLTPS <i>xmm1, xmm2</i>	CMPPS <i>xmm1, xmm2, 1</i>
CMPLEPS <i>xmm1, xmm2</i>	CMPPS <i>xmm1, xmm2, 2</i>
CMPUNORDPS <i>xmm1, xmm2</i>	CMPPS <i>xmm1, xmm2, 3</i>
CMPNEQPS <i>xmm1, xmm2</i>	CMPPS <i>xmm1, xmm2, 4</i>
CMPNLTPS <i>xmm1, xmm2</i>	CMPPS <i>xmm1, xmm2, 5</i>
CMPNLEPS <i>xmm1, xmm2</i>	CMPPS <i>xmm1, xmm2, 6</i>
CMPORDPS <i>xmm1, xmm2</i>	CMPPS <i>xmm1, xmm2, 7</i>

The greater-than relations that the processor does not implement require more than one instruction to emulate in software and therefore should not be implemented as pseudo-ops. (For these, the programmer should reverse the operands of the corresponding less than relations and use move instructions to ensure that the mask is moved to the correct destination register and that the source operand is left intact.)

Processors with "CPUID.1H:ECX.AVX = 1" implement the full complement of 32 predicates shown in Table 3-5, software emulation is no longer needed. Compilers and assemblers may implement the following three-operand pseudo-ops in addition to the four-operand VCMPPS instruction. See Table 3-5, where the notation of reg1 and reg2 represent either XMM registers or YMM registers. Compiler should treat reserved Imm8 values as illegal syntax. Alternately, intrinsics can map the pseudo-ops to pre-defined constants to support a simpler intrinsic interface. Compilers and assemblers may implement three-operand pseudo-ops for EVEX encoded VCMPPS instructions in a similar fashion by extending the syntax listed in Table 3-5.

Table 3-5. Pseudo-Op and VCMPPS Implementation

<b>Pseudo-Op</b>	<b>CMPPS Implementation</b>
<i>VCMPPEQPS reg1, reg2, reg3</i>	<i>VCMPPS reg1, reg2, reg3, 0</i>
<i>VCMPPLTPS reg1, reg2, reg3</i>	<i>VCMPPS reg1, reg2, reg3, 1</i>
<i>VCMPLEPS reg1, reg2, reg3</i>	<i>VCMPPS reg1, reg2, reg3, 2</i>
<i>VCMPUNORDPS reg1, reg2, reg3</i>	<i>VCMPPS reg1, reg2, reg3, 3</i>
<i>VCMPNEQPS reg1, reg2, reg3</i>	<i>VCMPPS reg1, reg2, reg3, 4</i>
<i>VCMPNLTPS reg1, reg2, reg3</i>	<i>VCMPPS reg1, reg2, reg3, 5</i>
<i>VCMPNLEPS reg1, reg2, reg3</i>	<i>VCMPPS reg1, reg2, reg3, 6</i>
<i>VCMPORDPS reg1, reg2, reg3</i>	<i>VCMPPS reg1, reg2, reg3, 7</i>
<i>VCMPPEQ_UQPS reg1, reg2, reg3</i>	<i>VCMPPS reg1, reg2, reg3, 8</i>
<i>VCMPNGEPS reg1, reg2, reg3</i>	<i>VCMPPS reg1, reg2, reg3, 9</i>
<i>VCMPNGTPS reg1, reg2, reg3</i>	<i>VCMPPS reg1, reg2, reg3, 0AH</i>
<i>VCMPFALSEPS reg1, reg2, reg3</i>	<i>VCMPPS reg1, reg2, reg3, 0BH</i>
<i>VCMPNEQ_OQPS reg1, reg2, reg3</i>	<i>VCMPPS reg1, reg2, reg3, 0CH</i>
<i>VCMPGEPs reg1, reg2, reg3</i>	<i>VCMPPS reg1, reg2, reg3, 0DH</i>
<i>VCMPGTPS reg1, reg2, reg3</i>	<i>VCMPPS reg1, reg2, reg3, 0EH</i>
<i>VCMPTRUEPS reg1, reg2, reg3</i>	<i>VCMPPS reg1, reg2, reg3, 0FH</i>
<i>VCMPPEQ_OSPS reg1, reg2, reg3</i>	<i>VCMPPS reg1, reg2, reg3, 10H</i>
<i>VCMPPLT_OQPS reg1, reg2, reg3</i>	<i>VCMPPS reg1, reg2, reg3, 11H</i>
<i>VCMPLE_OQPS reg1, reg2, reg3</i>	<i>VCMPPS reg1, reg2, reg3, 12H</i>
<i>VCMPUNORD_SPS reg1, reg2, reg3</i>	<i>VCMPPS reg1, reg2, reg3, 13H</i>
<i>VCMPNEQ_USPS reg1, reg2, reg3</i>	<i>VCMPPS reg1, reg2, reg3, 14H</i>
<i>VCMPNLT_UQPS reg1, reg2, reg3</i>	<i>VCMPPS reg1, reg2, reg3, 15H</i>
<i>VCMPNLE_UQPS reg1, reg2, reg3</i>	<i>VCMPPS reg1, reg2, reg3, 16H</i>
<i>VCMPORD_SPS reg1, reg2, reg3</i>	<i>VCMPPS reg1, reg2, reg3, 17H</i>
<i>VCMPPEQ_USPS reg1, reg2, reg3</i>	<i>VCMPPS reg1, reg2, reg3, 18H</i>
<i>VCMPNGE_UQPS reg1, reg2, reg3</i>	<i>VCMPPS reg1, reg2, reg3, 19H</i>
<i>VCMPNGT_UQPS reg1, reg2, reg3</i>	<i>VCMPPS reg1, reg2, reg3, 1AH</i>
<i>VCMPFALSE_OSPS reg1, reg2, reg3</i>	<i>VCMPPS reg1, reg2, reg3, 1BH</i>
<i>VCMPNEQ_OSPS reg1, reg2, reg3</i>	<i>VCMPPS reg1, reg2, reg3, 1CH</i>
<i>VCMPGE_OQPS reg1, reg2, reg3</i>	<i>VCMPPS reg1, reg2, reg3, 1DH</i>
<i>VCMPGT_OQPS reg1, reg2, reg3</i>	<i>VCMPPS reg1, reg2, reg3, 1EH</i>
<i>VCMPTRUE_USPS reg1, reg2, reg3</i>	<i>VCMPPS reg1, reg2, reg3, 1FH</i>

**Operation**

CASE (COMPARISON PREDICATE) OF

- 0: OP3  $\leftarrow$  EQ\_OQ; OP5  $\leftarrow$  EQ\_OQ;
- 1: OP3  $\leftarrow$  LT\_OS; OP5  $\leftarrow$  LT\_OS;
- 2: OP3  $\leftarrow$  LE\_OS; OP5  $\leftarrow$  LE\_OS;
- 3: OP3  $\leftarrow$  UNORD\_Q; OP5  $\leftarrow$  UNORD\_Q;
- 4: OP3  $\leftarrow$  NEQ\_UQ; OP5  $\leftarrow$  NEQ\_UQ;
- 5: OP3  $\leftarrow$  NLT\_US; OP5  $\leftarrow$  NLT\_US;
- 6: OP3  $\leftarrow$  NLE\_US; OP5  $\leftarrow$  NLE\_US;
- 7: OP3  $\leftarrow$  ORD\_Q; OP5  $\leftarrow$  ORD\_Q;
- 8: OP5  $\leftarrow$  EQ\_UQ;
- 9: OP5  $\leftarrow$  NGE\_US;
- 10: OP5  $\leftarrow$  NGT\_US;
- 11: OP5  $\leftarrow$  FALSE\_OQ;
- 12: OP5  $\leftarrow$  NEQ\_OQ;
- 13: OP5  $\leftarrow$  GE\_OS;
- 14: OP5  $\leftarrow$  GT\_OS;
- 15: OP5  $\leftarrow$  TRUE\_UQ;
- 16: OP5  $\leftarrow$  EQ\_OS;
- 17: OP5  $\leftarrow$  LT\_OQ;
- 18: OP5  $\leftarrow$  LE\_OQ;
- 19: OP5  $\leftarrow$  UNORD\_S;
- 20: OP5  $\leftarrow$  NEQ\_US;
- 21: OP5  $\leftarrow$  NLT\_UQ;
- 22: OP5  $\leftarrow$  NLE\_UQ;
- 23: OP5  $\leftarrow$  ORD\_S;
- 24: OP5  $\leftarrow$  EQ\_US;
- 25: OP5  $\leftarrow$  NGE\_UQ;
- 26: OP5  $\leftarrow$  NGT\_UQ;
- 27: OP5  $\leftarrow$  FALSE\_OS;
- 28: OP5  $\leftarrow$  NEQ\_OS;
- 29: OP5  $\leftarrow$  GE\_OQ;
- 30: OP5  $\leftarrow$  GT\_OQ;
- 31: OP5  $\leftarrow$  TRUE\_US;

DEFAULT: Reserved

ESAC;



**VCMPSS (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

i ← j \* 32

IF k2[j] OR \*no writemask\*

THEN

IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

THEN

CMP ← SRC1[i+31:i] OP5 SRC2[31:0]

ELSE

CMP ← SRC1[i+31:i] OP5 SRC2[j+31:i]

FI;

IF CMP = TRUE

THEN DEST[j] ← 1;

ELSE DEST[j] ← 0; FI;

ELSE DEST[j] ← 0 ; zeroing-masking only FI;

FI;

ENDFOR

DEST[MAX\_KL-1:KL] ← 0

**VCMPSS (VEX.256 encoded version)**

CMP0 ← SRC1[31:0] OP5 SRC2[31:0];

CMP1 ← SRC1[63:32] OP5 SRC2[63:32];

CMP2 ← SRC1[95:64] OP5 SRC2[95:64];

CMP3 ← SRC1[127:96] OP5 SRC2[127:96];

CMP4 ← SRC1[159:128] OP5 SRC2[159:128];

CMP5 ← SRC1[191:160] OP5 SRC2[191:160];

CMP6 ← SRC1[223:192] OP5 SRC2[223:192];

CMP7 ← SRC1[255:224] OP5 SRC2[255:224];

IF CMP0 = TRUE

THEN DEST[31:0] ← FFFFFFFFH;

ELSE DEST[31:0] ← 00000000H; FI;

IF CMP1 = TRUE

THEN DEST[63:32] ← FFFFFFFFH;

ELSE DEST[63:32] ← 00000000H; FI;

IF CMP2 = TRUE

THEN DEST[95:64] ← FFFFFFFFH;

ELSE DEST[95:64] ← 00000000H; FI;

IF CMP3 = TRUE

THEN DEST[127:96] ← FFFFFFFFH;

ELSE DEST[127:96] ← 00000000H; FI;

IF CMP4 = TRUE

THEN DEST[159:128] ← FFFFFFFFH;

ELSE DEST[159:128] ← 00000000H; FI;

IF CMP5 = TRUE

THEN DEST[191:160] ← FFFFFFFFH;

ELSE DEST[191:160] ← 00000000H; FI;

IF CMP6 = TRUE

THEN DEST[223:192] ← FFFFFFFFH;

ELSE DEST[223:192] ← 00000000H; FI;

IF CMP7 = TRUE

THEN DEST[255:224] ← FFFFFFFFH;

ELSE DEST[255:224] ← 00000000H; FI;

DEST[MAX\_VL-1:256] ← 0

**VCMPSS (VEX.128 encoded version)**

```

CMP0 ← SRC1[31:0] OP5 SRC2[31:0];
CMP1 ← SRC1[63:32] OP5 SRC2[63:32];
CMP2 ← SRC1[95:64] OP5 SRC2[95:64];
CMP3 ← SRC1[127:96] OP5 SRC2[127:96];
IF CMP0 = TRUE
    THEN DEST[31:0] ← FFFFFFFFH;
    ELSE DEST[31:0] ← 00000000H; FI;
IF CMP1 = TRUE
    THEN DEST[63:32] ← FFFFFFFFH;
    ELSE DEST[63:32] ← 00000000H; FI;
IF CMP2 = TRUE
    THEN DEST[95:64] ← FFFFFFFFH;
    ELSE DEST[95:64] ← 00000000H; FI;
IF CMP3 = TRUE
    THEN DEST[127:96] ← FFFFFFFFH;
    ELSE DEST[127:96] ← 00000000H; FI;
DEST[MAX_VL-1:128] ← 0

```

**CMPPS (128-bit Legacy SSE version)**

```

CMP0 ← SRC1[31:0] OP3 SRC2[31:0];
CMP1 ← SRC1[63:32] OP3 SRC2[63:32];
CMP2 ← SRC1[95:64] OP3 SRC2[95:64];
CMP3 ← SRC1[127:96] OP3 SRC2[127:96];
IF CMP0 = TRUE
    THEN DEST[31:0] ← FFFFFFFFH;
    ELSE DEST[31:0] ← 00000000H; FI;
IF CMP1 = TRUE
    THEN DEST[63:32] ← FFFFFFFFH;
    ELSE DEST[63:32] ← 00000000H; FI;
IF CMP2 = TRUE
    THEN DEST[95:64] ← FFFFFFFFH;
    ELSE DEST[95:64] ← 00000000H; FI;
IF CMP3 = TRUE
    THEN DEST[127:96] ← FFFFFFFFH;
    ELSE DEST[127:96] ← 00000000H; FI;
DEST[MAX_VL-1:128] (Unmodified)

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VCMPSS __mmask16 __mm512_cmp_ps_mask( __m512 a, __m512 b, int imm);
VCMPSS __mmask16 __mm512_cmp_round_ps_mask( __m512 a, __m512 b, int imm, int sae);
VCMPSS __mmask16 __mm512_mask_cmp_ps_mask( __mmask16 k1, __m512 a, __m512 b, int imm);
VCMPSS __mmask16 __mm512_mask_cmp_round_ps_mask( __mmask16 k1, __m512 a, __m512 b, int imm, int sae);
VCMPSS __mmask8 __mm256_cmp_ps_mask( __m256 a, __m256 b, int imm);
VCMPSS __mmask8 __mm256_mask_cmp_ps_mask( __mmask8 k1, __m256 a, __m256 b, int imm);
VCMPSS __mmask8 __mm_cmp_ps_mask( __m128 a, __m128 b, int imm);
VCMPSS __mmask8 __mm_mask_cmp_ps_mask( __mmask8 k1, __m128 a, __m128 b, int imm);
VCMPSS __m256 __mm256_cmp_ps(__m256 a, __m256 b, int imm)
CMPPS __m128 __mm_cmp_ps(__m128 a, __m128 b, int imm)

```

**SIMD Floating-Point Exceptions**

Invalid if SNaN operand and invalid if QNaN and predicate as listed in Table 3-1.

Denormal



## Other Exceptions

VEX-encoded instructions, see Exceptions Type 2.

EVEX-encoded instructions, see Exceptions Type E2.

## CMPSD—Compare Scalar Double-Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F C2 /r ib CMPSD xmm1, xmm2/m64, imm8	RMI	V/V	SSE2	Compare low double-precision floating-point value in xmm2/m64 and xmm1 using bits 2:0 of imm8 as comparison predicate.
VEX.NDS.128.F2.0F.WIG C2 /r ib VCMPSSD xmm1, xmm2, xmm3/m64, imm8	RVMI	V/V	AVX	Compare low double-precision floating-point value in xmm3/m64 and xmm2 using bits 4:0 of imm8 as comparison predicate.
EVEX.NDS.LIG.F2.0F.W1 C2 /r ib VCMPSSD k1 {k2}, xmm2, xmm3/m64{sae}, imm8	T1S	V/V	AVX512F	Compare low double-precision floating-point value in xmm3/m64 and xmm2 using bits 4:0 of imm8 as comparison predicate with writemask k2 and leave the result in mask register k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (r, w)	ModRM:r/m (r)	Imm8	NA
RVMI	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	Imm8
T1S	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	Imm8

### Description

Compares the low double-precision floating-point values in the second source operand and the first source operand and returns the results in of the comparison to the destination operand. The comparison predicate operand (immediate operand) specifies the type of comparison performed.

128-bit Legacy SSE version: The first source and destination operand (first operand) is an XMM register. The second source operand (second operand) can be an XMM register or 64-bit memory location. Bits (MAX\_VL-1:64) of the corresponding YMM destination register remain unchanged. The comparison result is a quadword mask of all 1s (comparison true) or all 0s (comparison false).

VEX.128 encoded version: The first source operand (second operand) is an XMM register. The second source operand (third operand) can be an XMM register or a 64-bit memory location. The result is stored in the low quadword of the destination operand; the high quadword is filled with the contents of the high quadword of the first source operand. Bits (MAX\_VL-1:128) of the destination ZMM register are zeroed. The comparison result is a quadword mask of all 1s (comparison true) or all 0s (comparison false).

EVEX encoded version: The first source operand (second operand) is an XMM register. The second source operand can be a XMM register or a 64-bit memory location. The destination operand (first operand) is an opmask register. The comparison result is a single mask bit of 1 (comparison true) or 0 (comparison false), written to the destination starting from the LSB according to the writemask k2. Bits (MAX\_KL-1:128) of the destination register are cleared.

The comparison predicate operand is an 8-bit immediate:

- For instructions encoded using the VEX prefix, bits 4:0 define the type of comparison to be performed (see Table 3-1). Bits 5 through 7 of the immediate are reserved.
- For instruction encodings that do not use VEX prefix, bits 2:0 define the type of comparison to be made (see the first 8 rows of Table 3-1). Bits 3 through 7 of the immediate are reserved.

The unordered relationship is true when at least one of the two source operands being compared is a NaN; the ordered relationship is true when neither source operand is a NaN.

A subsequent computational instruction that uses the mask result in the destination operand as an input operand will not generate an exception, because a mask of all 0s corresponds to a floating-point value of +0.0 and a mask of all 1s corresponds to a QNaN.

Note that processors with "CPUID.1H:ECX.AVX =0" do not implement the "greater-than", "greater-than-or-equal", "not-greater than", and "not-greater-than-or-equal relations" predicates. These comparisons can be made either by using the inverse relationship (that is, use the "not-less-than-or-equal" to make a "greater-than" comparison)

or by using software emulation. When using software emulation, the program must swap the operands (copying registers when necessary to protect the data that will now be in the destination), and then perform the compare using a different predicate. The predicate to be used for these emulations is listed in the first 8 rows of Table 3-7 (*Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2A*) under the heading Emulation.

Compilers and assemblers may implement the following two-operand pseudo-ops in addition to the three-operand CMPSD instruction, for processors with "CPUID.1H:ECX.AVX =0". See Table 3-6. Compiler should treat reserved Imm8 values as illegal syntax.

**Table 3-6. Pseudo-Op and CMPSD Implementation**

Pseudo-Op	CMPSD Implementation
CMPEQSD <i>xmm1, xmm2</i>	CMPSD <i>xmm1, xmm2, 0</i>
CMPLTSD <i>xmm1, xmm2</i>	CMPSD <i>xmm1, xmm2, 1</i>
CMPLSD <i>xmm1, xmm2</i>	CMPSD <i>xmm1, xmm2, 2</i>
CMPUNORDSD <i>xmm1, xmm2</i>	CMPSD <i>xmm1, xmm2, 3</i>
CMPNEQSD <i>xmm1, xmm2</i>	CMPSD <i>xmm1, xmm2, 4</i>
CMPNLTSD <i>xmm1, xmm2</i>	CMPSD <i>xmm1, xmm2, 5</i>
CMPNLESD <i>xmm1, xmm2</i>	CMPSD <i>xmm1, xmm2, 6</i>
CMPODSD <i>xmm1, xmm2</i>	CMPSD <i>xmm1, xmm2, 7</i>

The greater-than relations that the processor does not implement require more than one instruction to emulate in software and therefore should not be implemented as pseudo-ops. (For these, the programmer should reverse the operands of the corresponding less than relations and use move instructions to ensure that the mask is moved to the correct destination register and that the source operand is left intact.)

Processors with "CPUID.1H:ECX.AVX =1" implement the full complement of 32 predicates shown in Table 3-7, software emulation is no longer needed. Compilers and assemblers may implement the following three-operand pseudo-ops in addition to the four-operand VCMPSD instruction. See Table 3-7, where the notations of *reg1*, *reg2*, and *reg3* represent either XMM registers or YMM registers. Compiler should treat reserved Imm8 values as illegal syntax. Alternately, intrinsics can map the pseudo-ops to pre-defined constants to support a simpler intrinsic interface. Compilers and assemblers may implement three-operand pseudo-ops for EVEX encoded VCMPSD instructions in a similar fashion by extending the syntax listed in Table 3-7.

**Table 3-7. Pseudo-Op and VCMPSD Implementation**

Pseudo-Op	CMPSD Implementation
VCMPEQSD <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 0</i>
VCMPLTSD <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 1</i>
VCMPLSD <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 2</i>
VCMPUNORDSD <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 3</i>
VCMPNEQSD <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 4</i>
VCMNLTSD <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 5</i>
VCMNLESD <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 6</i>
VCMPODSD <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 7</i>
VCMPEQ_UQSD <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 8</i>
VCMPNGESD <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 9</i>
VCMPNGTSD <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 0AH</i>
VCMPFALSESD <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 0BH</i>
VCMPNEQ_OQSD <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 0CH</i>
VCMPGESD <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 0DH</i>

Table 3-7. Pseudo-Op and VCMPSD Implementation

Pseudo-Op	CMPSD Implementation
VCMPGTSD <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 0EH</i>
VCMPTRUESD <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 0FH</i>
VCMPEQ_OSSD <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 10H</i>
VCMPLT_OQSD <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 11H</i>
VCMPLT_OQSD <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 12H</i>
VCMUNORD_SSD <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 13H</i>
VCMNEQ_USSD <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 14H</i>
VCMNLT_UQSD <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 15H</i>
VCMNLE_UQSD <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 16H</i>
VCMORD_SSD <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 17H</i>
VCMPEQ_USSD <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 18H</i>
VCMNGE_UQSD <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 19H</i>
VCMNGT_UQSD <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 1AH</i>
VCMFALSE_OSSD <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 1BH</i>
VCMNEQ_OSSD <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 1CH</i>
VCMNGE_OQSD <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 1DH</i>
VCMNGT_OQSD <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 1EH</i>
VCMPTTRUE_USSD <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 1FH</i>

Software should ensure VCMPSD is encoded with VEX.L=0. Encoding VCMPSD with VEX.L=1 may encounter unpredictable behavior across different processor generations.

### Operation

CASE (COMPARISON PREDICATE) OF

- 0: OP3 ←EQ\_OQ; OP5 ←EQ\_OQ;
- 1: OP3 ←LT\_OS; OP5 ←LT\_OS;
- 2: OP3 ←LE\_OS; OP5 ←LE\_OS;
- 3: OP3 ←UNORD\_Q; OP5 ←UNORD\_Q;
- 4: OP3 ←NEQ\_UQ; OP5 ←NEQ\_UQ;
- 5: OP3 ←NLT\_US; OP5 ←NLT\_US;
- 6: OP3 ←NLE\_US; OP5 ←NLE\_US;
- 7: OP3 ←ORD\_Q; OP5 ←ORD\_Q;
- 8: OP5 ←EQ\_UQ;
- 9: OP5 ←NGE\_US;
- 10: OP5 ←NGT\_US;
- 11: OP5 ←FALSE\_OQ;
- 12: OP5 ←NEQ\_OQ;
- 13: OP5 ←GE\_OS;
- 14: OP5 ←GT\_OS;
- 15: OP5 ←TRUE\_UQ;
- 16: OP5 ←EQ\_OS;
- 17: OP5 ←LT\_OQ;
- 18: OP5 ←LE\_OQ;
- 19: OP5 ←UNORD\_S;
- 20: OP5 ←NEQ\_US;
- 21: OP5 ←NLT\_UQ;

22: OP5 ←NLE\_UQ;  
 23: OP5 ←ORD\_S;  
 24: OP5 ←EQ\_US;  
 25: OP5 ←NGE\_UQ;  
 26: OP5 ←NGT\_UQ;  
 27: OP5 ←FALSE\_OS;  
 28: OP5 ←NEQ\_OS;  
 29: OP5 ←GE\_OQ;  
 30: OP5 ←GT\_OQ;  
 31: OP5 ←TRUE\_US;  
 DEFAULT: Reserved

ESAC;

#### VCMPSD (EVEX encoded version)

CMPO ← SRC1[63:0] OP5 SRC2[63:0];

IF k2[0] or \*no writemask\*

THEN IF CMPO = TRUE

THEN DEST[0] ← 1;

ELSE DEST[0] ← 0; FI;

ELSE DEST[0] ← 0 ; zeroing-masking only

FI;

DEST[MAX\_KL-1:1] ← 0

#### CMPSD (128-bit Legacy SSE version)

CMPO ←DEST[63:0] OP3 SRC[63:0];

IF CMPO = TRUE

THEN DEST[63:0] ←FFFFFFFFFFFFFFFFH;

ELSE DEST[63:0] ←0000000000000000H; FI;

DEST[MAX\_VL-1:64] (Unmodified)

#### VCMPSD (VEX.128 encoded version)

CMPO ←SRC1[63:0] OP5 SRC2[63:0];

IF CMPO = TRUE

THEN DEST[63:0] ←FFFFFFFFFFFFFFFFH;

ELSE DEST[63:0] ←0000000000000000H; FI;

DEST[127:64] ←SRC1[127:64]

DEST[MAX\_VL-1:128] ←0

#### Intel C/C++ Compiler Intrinsic Equivalent

VCMPSD \_\_mmask8 \_mm\_cmp\_sd\_mask( \_\_m128d a, \_\_m128d b, int imm);

VCMPSD \_\_mmask8 \_mm\_cmp\_round\_sd\_mask( \_\_m128d a, \_\_m128d b, int imm, int sae);

VCMPSD \_\_mmask8 \_mm\_mask\_cmp\_sd\_mask( \_\_mmask8 k1, \_\_m128d a, \_\_m128d b, int imm);

VCMPSD \_\_mmask8 \_mm\_mask\_cmp\_round\_sd\_mask( \_\_mmask8 k1, \_\_m128d a, \_\_m128d b, int imm, int sae);

(V)CMPSD \_\_m128d \_mm\_cmp\_sd( \_\_m128d a, \_\_m128d b, const int imm)

#### SIMD Floating-Point Exceptions

Invalid if SNaN operand, Invalid if QNaN and predicate as listed in Table 3-1 Denormal.

#### Other Exceptions

VEX-encoded instructions, see Exceptions Type 3.

EVEX-encoded instructions, see Exceptions Type E3.

## CMPSS—Compare Scalar Single-Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F C2 /r ib CMPSS xmm1, xmm2/m32, imm8	RMI	V/V	SSE	Compare low single-precision floating-point value in xmm2/m32 and xmm1 using bits 2:0 of imm8 as comparison predicate.
VEX.NDS.128.F3.0F.WIG C2 /r ib VCMPSS xmm1, xmm2, xmm3/m32, imm8	RVMI	V/V	AVX	Compare low single-precision floating-point value in xmm3/m32 and xmm2 using bits 4:0 of imm8 as comparison predicate.
EVEX.NDS.LIG.F3.0F.WO C2 /r ib VCMPSS k1 {k2}, xmm2, xmm3/m32{sae}, imm8	T1S	V/V	AVX512F	Compare low single-precision floating-point value in xmm3/m32 and xmm2 using bits 4:0 of imm8 as comparison predicate with writemask k2 and leave the result in mask register k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (r, w)	ModRM:r/m (r)	Imm8	NA
RVMI	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	Imm8
T1S	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	Imm8

### Description

Compares the low single-precision floating-point values in the second source operand and the first source operand and returns the results of the comparison to the destination operand. The comparison predicate operand (immediate operand) specifies the type of comparison performed.

128-bit Legacy SSE version: The first source and destination operand (first operand) is an XMM register. The second source operand (second operand) can be an XMM register or 32-bit memory location. Bits (MAX\_VL-1:32) of the corresponding YMM destination register remain unchanged. The comparison result is a doubleword mask of all 1s (comparison true) or all 0s (comparison false).

VEX.128 encoded version: The first source operand (second operand) is an XMM register. The second source operand (third operand) can be an XMM register or a 32-bit memory location. The result is stored in the low 32 bits of the destination operand; bits 128:32 of the destination operand are copied from the first source operand. Bits (MAX\_VL-1:128) of the destination ZMM register are zeroed. The comparison result is a doubleword mask of all 1s (comparison true) or all 0s (comparison false).

EVEX encoded version: The first source operand (second operand) is an XMM register. The second source operand can be a XMM register or a 32-bit memory location. The destination operand (first operand) is an opmask register. The comparison result is a single mask bit of 1 (comparison true) or 0 (comparison false), written to the destination starting from the LSB according to the writemask k2. Bits (MAX\_KL-1:128) of the destination register are cleared.

The comparison predicate operand is an 8-bit immediate:

- For instructions encoded using the VEX prefix, bits 4:0 define the type of comparison to be performed (see Table 3-1). Bits 5 through 7 of the immediate are reserved.
- For instruction encodings that do not use VEX prefix, bits 2:0 define the type of comparison to be made (see the first 8 rows of Table 3-1). Bits 3 through 7 of the immediate are reserved.

The unordered relationship is true when at least one of the two source operands being compared is a NaN; the ordered relationship is true when neither source operand is a NaN.

A subsequent computational instruction that uses the mask result in the destination operand as an input operand will not generate an exception, because a mask of all 0s corresponds to a floating-point value of +0.0 and a mask of all 1s corresponds to a QNaN.

Note that processors with "CPUID.1H:ECX.AVX =0" do not implement the "greater-than", "greater-than-or-equal", "not-greater than", and "not-greater-than-or-equal relations" predicates. These comparisons can be made either



by using the inverse relationship (that is, use the “not-less-than-or-equal” to make a “greater-than” comparison) or by using software emulation. When using software emulation, the program must swap the operands (copying registers when necessary to protect the data that will now be in the destination), and then perform the compare using a different predicate. The predicate to be used for these emulations is listed in the first 8 rows of Table 3-7 (*Intel 64 and IA-32 Architectures Software Developer’s Manual Volume 2A*) under the heading Emulation.

Compilers and assemblers may implement the following two-operand pseudo-ops in addition to the three-operand CMPSS instruction, for processors with “CPUID.1H:ECX.AVX =0”. See Table 3-8. Compiler should treat reserved Imm8 values as illegal syntax.

**Table 3-8. Pseudo-Op and CMPSS Implementation**

Pseudo-Op	CMPSS Implementation
CMPEQSS <i>xmm1, xmm2</i>	CMPSS <i>xmm1, xmm2, 0</i>
CMPLTSS <i>xmm1, xmm2</i>	CMPSS <i>xmm1, xmm2, 1</i>
CMPLSS <i>xmm1, xmm2</i>	CMPSS <i>xmm1, xmm2, 2</i>
CMPUNORDSS <i>xmm1, xmm2</i>	CMPSS <i>xmm1, xmm2, 3</i>
CMPNEQSS <i>xmm1, xmm2</i>	CMPSS <i>xmm1, xmm2, 4</i>
CMPNLTSS <i>xmm1, xmm2</i>	CMPSS <i>xmm1, xmm2, 5</i>
CMPNLESS <i>xmm1, xmm2</i>	CMPSS <i>xmm1, xmm2, 6</i>
CMPORDSS <i>xmm1, xmm2</i>	CMPSS <i>xmm1, xmm2, 7</i>

The greater-than relations that the processor does not implement require more than one instruction to emulate in software and therefore should not be implemented as pseudo-ops. (For these, the programmer should reverse the operands of the corresponding less than relations and use move instructions to ensure that the mask is moved to the correct destination register and that the source operand is left intact.)

Processors with “CPUID.1H:ECX.AVX =1” implement the full complement of 32 predicates shown in Table 3-7, software emulation is no longer needed. Compilers and assemblers may implement the following three-operand pseudo-ops in addition to the four-operand VCMPS instruction. See Table 3-9, where the notations of *reg1*, *reg2*, and *reg3* represent either XMM registers or YMM registers. Compiler should treat reserved Imm8 values as illegal syntax. Alternately, intrinsics can map the pseudo-ops to pre-defined constants to support a simpler intrinsic interface. Compilers and assemblers may implement three-operand pseudo-ops for EVEX encoded VCMPS instructions in a similar fashion by extending the syntax listed in Table 3-9.

**Table 3-9. Pseudo-Op and VCMPS Implementation**

Pseudo-Op	VCMPS Implementation
VCMPEQSS <i>reg1, reg2, reg3</i>	VCMPS <i>reg1, reg2, reg3, 0</i>
VCMPLTSS <i>reg1, reg2, reg3</i>	VCMPS <i>reg1, reg2, reg3, 1</i>
VCMPLSS <i>reg1, reg2, reg3</i>	VCMPS <i>reg1, reg2, reg3, 2</i>
VCMPUNORDSS <i>reg1, reg2, reg3</i>	VCMPS <i>reg1, reg2, reg3, 3</i>
VCMPNEQSS <i>reg1, reg2, reg3</i>	VCMPS <i>reg1, reg2, reg3, 4</i>
VCMNLTSS <i>reg1, reg2, reg3</i>	VCMPS <i>reg1, reg2, reg3, 5</i>
VCMNLESS <i>reg1, reg2, reg3</i>	VCMPS <i>reg1, reg2, reg3, 6</i>
VCMPODSS <i>reg1, reg2, reg3</i>	VCMPS <i>reg1, reg2, reg3, 7</i>
VCMPEQ_UQSS <i>reg1, reg2, reg3</i>	VCMPS <i>reg1, reg2, reg3, 8</i>
VCMNGESS <i>reg1, reg2, reg3</i>	VCMPS <i>reg1, reg2, reg3, 9</i>
VCMNGTSS <i>reg1, reg2, reg3</i>	VCMPS <i>reg1, reg2, reg3, 0AH</i>
VCMFALSESS <i>reg1, reg2, reg3</i>	VCMPS <i>reg1, reg2, reg3, 0BH</i>
VCMPEQ_OQSS <i>reg1, reg2, reg3</i>	VCMPS <i>reg1, reg2, reg3, 0CH</i>
VCMNGESS <i>reg1, reg2, reg3</i>	VCMPS <i>reg1, reg2, reg3, 0DH</i>

Table 3-9. Pseudo-Op and VCM PSS Implementation

Pseudo-Op	VCM PSS Implementation
VCM PG TSS <i>reg1, reg2, reg3</i>	VCM PSS <i>reg1, reg2, reg3, 0EH</i>
VCM PTRUESS <i>reg1, reg2, reg3</i>	VCM PSS <i>reg1, reg2, reg3, 0FH</i>
VCMPEQ_OSSS <i>reg1, reg2, reg3</i>	VCM PSS <i>reg1, reg2, reg3, 10H</i>
VCMPLT_OQSS <i>reg1, reg2, reg3</i>	VCM PSS <i>reg1, reg2, reg3, 11H</i>
VCMPLD_OQSS <i>reg1, reg2, reg3</i>	VCM PSS <i>reg1, reg2, reg3, 12H</i>
VCM PUNORD_SSS <i>reg1, reg2, reg3</i>	VCM PSS <i>reg1, reg2, reg3, 13H</i>
VCM PNEQ_USSS <i>reg1, reg2, reg3</i>	VCM PSS <i>reg1, reg2, reg3, 14H</i>
VCM PNL T_UQSS <i>reg1, reg2, reg3</i>	VCM PSS <i>reg1, reg2, reg3, 15H</i>
VCM PNL E_UQSS <i>reg1, reg2, reg3</i>	VCM PSS <i>reg1, reg2, reg3, 16H</i>
VCM PORD_SSS <i>reg1, reg2, reg3</i>	VCM PSS <i>reg1, reg2, reg3, 17H</i>
VCMPEQ_USSS <i>reg1, reg2, reg3</i>	VCM PSS <i>reg1, reg2, reg3, 18H</i>
VCM PNGE_UQSS <i>reg1, reg2, reg3</i>	VCM PSS <i>reg1, reg2, reg3, 19H</i>
VCM PNG T_UQSS <i>reg1, reg2, reg3</i>	VCM PSS <i>reg1, reg2, reg3, 1AH</i>
VCM PFALSE_OSSS <i>reg1, reg2, reg3</i>	VCM PSS <i>reg1, reg2, reg3, 1BH</i>
VCM PNEQ_OSSS <i>reg1, reg2, reg3</i>	VCM PSS <i>reg1, reg2, reg3, 1CH</i>
VCM PGE_OQSS <i>reg1, reg2, reg3</i>	VCM PSS <i>reg1, reg2, reg3, 1DH</i>
VCM PG T_OQSS <i>reg1, reg2, reg3</i>	VCM PSS <i>reg1, reg2, reg3, 1EH</i>
VCM PTRUE_USSS <i>reg1, reg2, reg3</i>	VCM PSS <i>reg1, reg2, reg3, 1FH</i>

Software should ensure VCM PSS is encoded with VEX.L=0. Encoding VCM PSS with VEX.L=1 may encounter unpredictable behavior across different processor generations.

### Operation

CASE (COMPARISON PREDICATE) OF

0: OP3 ←EQ\_OQ; OP5 ←EQ\_OQ;  
 1: OP3 ←LT\_OS; OP5 ←LT\_OS;  
 2: OP3 ←LE\_OS; OP5 ←LE\_OS;  
 3: OP3 ←UNORD\_Q; OP5 ←UNORD\_Q;  
 4: OP3 ←NEQ\_UQ; OP5 ←NEQ\_UQ;  
 5: OP3 ←NLT\_US; OP5 ←NLT\_US;  
 6: OP3 ←NLE\_US; OP5 ←NLE\_US;  
 7: OP3 ←ORD\_Q; OP5 ←ORD\_Q;  
 8: OP5 ←EQ\_UQ;  
 9: OP5 ←NGE\_US;  
 10: OP5 ←NGT\_US;  
 11: OP5 ←FALSE\_OQ;  
 12: OP5 ←NEQ\_OQ;  
 13: OP5 ←GE\_OS;  
 14: OP5 ←GT\_OS;  
 15: OP5 ←TRUE\_UQ;  
 16: OP5 ←EQ\_OS;  
 17: OP5 ←LT\_OQ;  
 18: OP5 ←LE\_OQ;  
 19: OP5 ←UNORD\_S;  
 20: OP5 ←NEQ\_US;  
 21: OP5 ←NLT\_UQ;

22: OP5 ← NLE\_UQ;  
 23: OP5 ← ORD\_S;  
 24: OP5 ← EQ\_US;  
 25: OP5 ← NGE\_UQ;  
 26: OP5 ← NGT\_UQ;  
 27: OP5 ← FALSE\_OS;  
 28: OP5 ← NEQ\_OS;  
 29: OP5 ← GE\_OQ;  
 30: OP5 ← GT\_OQ;  
 31: OP5 ← TRUE\_US;  
 DEFAULT: Reserved

ESAC;

#### VCMPS (EVEX encoded version)

CMPO ← SRC1[31:0] OP5 SRC2[31:0];

IF k2[0] or \*no writemask\*

THEN IF CMPO = TRUE

THEN DEST[0] ← 1;

ELSE DEST[0] ← 0; FI;

ELSE DEST[0] ← 0 ; zeroing-masking only

FI;

DEST[MAX\_KL-1:1] ← 0

#### CMPS (128-bit Legacy SSE version)

CMPO ← DEST[31:0] OP3 SRC[31:0];

IF CMPO = TRUE

THEN DEST[31:0] ← FFFFFFFFH;

ELSE DEST[31:0] ← 00000000H; FI;

DEST[MAX\_VL-1:32] (Unmodified)

#### VCMPS (VEX.128 encoded version)

CMPO ← SRC1[31:0] OP5 SRC2[31:0];

IF CMPO = TRUE

THEN DEST[31:0] ← FFFFFFFFH;

ELSE DEST[31:0] ← 00000000H; FI;

DEST[127:32] ← SRC1[127:32]

DEST[MAX\_VL-1:128] ← 0

#### Intel C/C++ Compiler Intrinsic Equivalent

VCMPS \_\_mmask8 \_\_mm\_cmp\_ss\_mask( \_\_m128 a, \_\_m128 b, int imm);

VCMPS \_\_mmask8 \_\_mm\_cmp\_round\_ss\_mask( \_\_m128 a, \_\_m128 b, int imm, int sae);

VCMPS \_\_mmask8 \_\_mm\_mask\_cmp\_ss\_mask( \_\_mmask8 k1, \_\_m128 a, \_\_m128 b, int imm);

VCMPS \_\_mmask8 \_\_mm\_mask\_cmp\_round\_ss\_mask( \_\_mmask8 k1, \_\_m128 a, \_\_m128 b, int imm, int sae);

(V)CMPS \_\_m128 \_\_mm\_cmp\_ss(\_\_m128 a, \_\_m128 b, const int imm)

#### SIMD Floating-Point Exceptions

Invalid if SNaN operand, Invalid if QNaN and predicate as listed in Table 3-1, Denormal.

#### Other Exceptions

VEX-encoded instructions, see Exceptions Type 3.

EVEX-encoded instructions, see Exceptions Type E3.

## COMISD—Compare Scalar Ordered Double-Precision Floating-Point Values and Set EFLAGS

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 2F /r COMISD xmm1, xmm2/m64	RM	V/V	SSE2	Compare low double-precision floating-point values in xmm1 and xmm2/mem64 and set the EFLAGS flags accordingly.
VEX.128.66.0F.WIG 2F /r VCOMISD xmm1, xmm2/m64	RM	V/V	AVX	Compare low double-precision floating-point values in xmm1 and xmm2/mem64 and set the EFLAGS flags accordingly.
EVEX.LIG.66.0F.W1 2F /r VCOMISD xmm1, xmm2/m64{sae}	T1S	V/V	AVX512F	Compare low double-precision floating-point values in xmm1 and xmm2/mem64 and set the EFLAGS flags accordingly.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
T1S	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Compares the double-precision floating-point values in the low quadwords 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).

Operand 1 is an XMM register; operand 2 can be an XMM register or a 64 bit memory

location. The COMISD instruction differs from the UCOMISD instruction in that it signals a SIMD floating-point invalid operation exception (#I) when a source operand is either a QNaN or SNaN. The UCOMISD instruction signals an invalid numeric exception only if a source operand is an SNaN.

The EFLAGS register is not updated if an unmasked SIMD floating-point exception is generated.

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

Software should ensure VCOMISD is encoded with VEX.L=0. Encoding VCOMISD with VEX.L=1 may encounter unpredictable behavior across different processor generations.

### Operation

#### COMISD (all versions)

```
RESULT ← OrderedCompare(DEST[63:0] <> SRC[63:0]) {
```

```
(* Set EFLAGS *) CASE (RESULT) OF
```

```
  UNORDERED: ZF,PF,CF ← 111;
```

```
  GREATER_THAN: ZF,PF,CF ← 000;
```

```
  LESS_THAN: ZF,PF,CF ← 001;
```

```
  EQUAL: ZF,PF,CF ← 100;
```

```
ESAC;
```

```
OF, AF, SF ← 0; }
```

### Intel C/C++ Compiler Intrinsic Equivalent

VCOMISD int \_\_mm\_comi\_round\_sd(\_\_m128d a, \_\_m128d b, int imm, int sae);  
 VCOMISD int \_\_mm\_comieq\_sd (\_\_m128d a, \_\_m128d b)  
 VCOMISD int \_\_mm\_comilt\_sd (\_\_m128d a, \_\_m128d b)  
 VCOMISD int \_\_mm\_comile\_sd (\_\_m128d a, \_\_m128d b)  
 VCOMISD int \_\_mm\_comigt\_sd (\_\_m128d a, \_\_m128d b)  
 VCOMISD int \_\_mm\_comige\_sd (\_\_m128d a, \_\_m128d b)  
 VCOMISD int \_\_mm\_comineq\_sd (\_\_m128d a, \_\_m128d b)

### SIMD Floating-Point Exceptions

Invalid (if SNaN or QNaN operands), Denormal.

### Other Exceptions

VEX-encoded instructions, see Exceptions Type 3;

EVEX-encoded instructions, see Exceptions Type E3NF.

#UD                    If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

## COMISS—Compare Scalar Ordered Single-Precision Floating-Point Values and Set EFLAGS

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
OF 2F /r COMISS xmm1, xmm2/m32	RM	V/V	SSE	Compare low single-precision floating-point values in xmm1 and xmm2/mem32 and set the EFLAGS flags accordingly.
VEX.128.OF.WIG 2F /r VCOMISS xmm1, xmm2/m32	RM	V/V	AVX	Compare low single-precision floating-point values in xmm1 and xmm2/mem32 and set the EFLAGS flags accordingly.
EVEX.LIG.OF.WO 2F /r VCOMISS xmm1, xmm2/m32{sae}	T1S	V/V	AVX512F	Compare low single-precision floating-point values in xmm1 and xmm2/mem32 and set the EFLAGS flags accordingly.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
T1S	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Compares the single-precision floating-point values in the low quadwords 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).

Operand 1 is an XMM register; operand 2 can be an XMM register or a 32 bit memory location.

The COMISS instruction differs from the UCOMISS instruction in that it signals a SIMD floating-point invalid operation exception (#I) when a source operand is either a QNaN or SNaN. The UCOMISS instruction signals an invalid numeric exception only if a source operand is an SNaN.

The EFLAGS register is not updated if an unmasked SIMD floating-point exception is generated.

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

Software should ensure VCOMISS is encoded with VEX.L=0. Encoding VCOMISS with VEX.L=1 may encounter unpredictable behavior across different processor generations.

### Operation

#### COMISS (all versions)

```
RESULT ← OrderedCompare(DEST[31:0] <> SRC[31:0]) {
```

```
(* Set EFLAGS *) CASE (RESULT) OF
```

```
  UNORDERED: ZF,PF,CF ← 111;
```

```
  GREATER_THAN: ZF,PF,CF ← 000;
```

```
  LESS_THAN: ZF,PF,CF ← 001;
```

```
  EQUAL: ZF,PF,CF ← 100;
```

```
ESAC;
```

```
OF, AF, SF ← 0; }
```

### Intel C/C++ Compiler Intrinsic Equivalent

VCOMISS int \_\_mm\_comi\_round\_ss(\_\_m128 a, \_\_m128 b, int imm, int sae);  
 VCOMISS int \_\_mm\_comieq\_ss (\_\_m128 a, \_\_m128 b)  
 VCOMISS int \_\_mm\_comilt\_ss (\_\_m128 a, \_\_m128 b)  
 VCOMISS int \_\_mm\_comile\_ss (\_\_m128 a, \_\_m128 b)  
 VCOMISS int \_\_mm\_comigt\_ss (\_\_m128 a, \_\_m128 b)  
 VCOMISS int \_\_mm\_comige\_ss (\_\_m128 a, \_\_m128 b)  
 VCOMISS int \_\_mm\_comineq\_ss (\_\_m128 a, \_\_m128 b)

### SIMD Floating-Point Exceptions

Invalid (if SNaN or QNaN operands), Denormal.

### Other Exceptions

VEX-encoded instructions, see Exceptions Type 3;

EVEX-encoded instructions, see Exceptions Type E3NF.

#UD                    If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

## CPUID—CPU Identification

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
0F A2	CPUID	NP	Valid	Valid	Returns processor identification and feature information to the EAX, EBX, ECX, and EDX registers, as determined by input entered in EAX (in some cases, ECX as well).

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

### Description

The ID flag (bit 21) in the EFLAGS register indicates support for the CPUID instruction. If a software procedure can set and clear this flag, the processor executing the procedure supports the CPUID instruction. This instruction operates the same in non-64-bit modes and 64-bit mode.

CPUID returns processor identification and feature information in the EAX, EBX, ECX, and EDX registers.<sup>1</sup> The instruction's output is dependent on the contents of the EAX register upon execution (in some cases, ECX as well). For example, the following pseudocode loads EAX with 00H and causes CPUID to return a Maximum Return Value and the Vendor Identification String in the appropriate registers:

```
MOV EAX, 00H
CPUID
```

Table 3-8 shows information returned, depending on the initial value loaded into the EAX register.

Two types of information are returned: basic and extended function information. If a value entered for CPUID.EAX is higher than the maximum input value for basic or extended function for that processor then the data for the highest basic information leaf is returned. For example, using the Intel Core i7 processor, the following is true:

```
CPUID.EAX = 05H (* Returns MONITOR/MWAIT leaf. *)
CPUID.EAX = 0AH (* Returns Architectural Performance Monitoring leaf. *)
CPUID.EAX = 0BH (* Returns Extended Topology Enumeration leaf. *)
CPUID.EAX = 0CH (* INVALID: Returns the same information as CPUID.EAX = 0BH. *)
CPUID.EAX = 80000008H (* Returns linear/physical address size data. *)
CPUID.EAX = 8000000AH (* INVALID: Returns same information as CPUID.EAX = 0BH. *)
```

If a value entered for CPUID.EAX is less than or equal to the maximum input value and the leaf is not supported on that processor then 0 is returned in all the registers.

When CPUID returns the highest basic leaf information as a result of an invalid input EAX value, any dependence on input ECX value in the basic leaf is honored.

CPUID can be executed at any privilege level to serialize instruction execution. Serializing instruction execution guarantees that any modifications to flags, registers, and memory for previous instructions are completed before the next instruction is fetched and executed.

### See also:

"Serializing Instructions" in Chapter 8, "Multiple-Processor Management," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

"Caching Translation Information" in Chapter 4, "Paging," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

1. On Intel 64 processors, CPUID clears the high 32 bits of the RAX/RBX/RCX/RDX registers in all modes.



**Table 3-7. Information Returned by CPUID Instruction**

Initial EAX Value	Information Provided about the Processor	
<i>Basic CPUID Information</i>		
0H	EAX	Maximum Input Value for Basic CPUID Information.
	EBX	"Genu"
	ECX	"ntel"
	EDX	"inel"
01H	EAX	Version Information: Type, Family, Model, and Stepping ID (see Figure 3-6).
	EBX	Bits 07 - 00: Brand Index. Bits 15 - 08: CLFLUSH line size (Value * 8 = cache line size in bytes; used also by CLFLUSHOPT). Bits 23 - 16: Maximum number of addressable IDs for logical processors in this physical package*. Bits 31 - 24: Initial APIC ID.
	ECX	Feature Information (see Figure 3-7 and Table 3-10).
	EDX	Feature Information (see Figure 3-8 and Table 3-11).
		<b>NOTES:</b> * The nearest power-of-2 integer that is not smaller than EBX[23:16] is the number of unique initial APIC IDs reserved for addressing different logical processors in a physical package. This field is only valid if CPUID.1.EDX.HTT[bit 28]= 1.
02H	EAX	Cache and TLB Information (see Table 3-12).
	EBX	Cache and TLB Information.
	ECX	Cache and TLB Information.
	EDX	Cache and TLB Information.
03H	EAX	Reserved.
	EBX	Reserved.
	ECX	Bits 00 - 31 of 96 bit processor serial number. (Available in Pentium III processor only; otherwise, the value in this register is reserved.)
	EDX	Bits 32 - 63 of 96 bit processor serial number. (Available in Pentium III processor only; otherwise, the value in this register is reserved.)
		<b>NOTES:</b> Processor serial number (PSN) is not supported in the Pentium 4 processor or later. On all models, use the PSN flag (returned using CPUID) to check for PSN support before accessing the feature.
CPUID leaves above 2 and below 80000000H are visible only when IA32_MISC_ENABLE[bit 22] has its default value of 0.		
<i>Deterministic Cache Parameters Leaf</i>		
04H		<b>NOTES:</b> Leaf 04H output depends on the initial value in ECX.* See also: "INPUT EAX = 04H: Returns Deterministic Cache Parameters for Each Level" on page 213.
	EAX	Bits 04 - 00: Cache Type Field. 0 = Null - No more caches. 1 = Data Cache. 2 = Instruction Cache. 3 = Unified Cache. 4-31 = Reserved.

**Table 3-7. Information Returned by CPUID Instruction (Contd.)**

Initial EAX Value	Information Provided about the Processor
	<p>Bits 07 - 05: Cache Level (starts at 1).                      Bit 08: Self Initializing cache level (does not need SW initialization).                      Bit 09: Fully Associative cache.</p> <p>Bits 13 - 10: Reserved.                      Bits 25 - 14: Maximum number of addressable IDs for logical processors sharing this cache**, ***.                      Bits 31 - 26: Maximum number of addressable IDs for processor cores in the physical package**, ****, *****.</p> <p>EBX Bits 11 - 00: L = System Coherency Line Size**.                      Bits 21 - 12: P = Physical Line partitions**.                      Bits 31 - 22: W = Ways of associativity**.</p> <p>ECX Bits 31-00: S = Number of Sets**.</p> <p>EDX Bit 00: Write-Back Invalidate/Invalidate.                      0 = WBINVD/INVD from threads sharing this cache acts upon lower level caches for threads sharing this cache.                      1 = WBINVD/INVD is not guaranteed to act upon lower level caches of non-originating threads sharing this cache.</p> <p>Bit 01: Cache Inclusiveness.                      0 = Cache is not inclusive of lower cache levels.                      1 = Cache is inclusive of lower cache levels.</p> <p>Bit 02: Complex Cache Indexing.                      0 = Direct mapped cache.                      1 = A complex function is used to index the cache, potentially using all address bits.</p> <p>Bits 31 - 03: Reserved = 0.</p> <p><b>NOTES:</b></p> <p>* If ECX contains an invalid sub leaf index, EAX/EBX/ECX/EDX return 0. Sub-leaf index n+1 is invalid if sub-leaf n returns EAX[4:0] as 0.</p> <p>** Add one to the return value to get the result.</p> <p>***The nearest power-of-2 integer that is not smaller than (1 + EAX[25:14]) is the number of unique initial APIC IDs reserved for addressing different logical processors sharing this cache.</p> <p>**** The nearest power-of-2 integer that is not smaller than (1 + EAX[31:26]) is the number of unique Core_IDs reserved for addressing different processor cores in a physical package. Core ID is a subset of bits of the initial APIC ID.</p> <p>***** The returned value is constant for valid initial values in ECX. Valid ECX values start from 0.</p>
	<i>MONITOR/MWAIT Leaf</i>
05H	<p>EAX Bits 15 - 00: Smallest monitor-line size in bytes (default is processor’s monitor granularity).                      Bits 31 - 16: Reserved = 0.</p> <p>EBX Bits 15 - 00: Largest monitor-line size in bytes (default is processor’s monitor granularity).                      Bits 31 - 16: Reserved = 0.</p> <p>ECX Bit 00: Enumeration of Monitor-Mwait extensions (beyond EAX and EBX registers) supported.                      Bit 01: Supports treating interrupts as break-event for MWAIT, even when interrupts disabled.                      Bits 31 - 02: Reserved.</p>

**Table 3-7. Information Returned by CPUID Instruction (Contd.)**

Initial EAX Value	Information Provided about the Processor	
	EDX	Bits 03 - 00: Number of C0* sub C-states supported using MWAIT. Bits 07 - 04: Number of C1* sub C-states supported using MWAIT. Bits 11 - 08: Number of C2* sub C-states supported using MWAIT. Bits 15 - 12: Number of C3* sub C-states supported using MWAIT. Bits 19 - 16: Number of C4* sub C-states supported using MWAIT. Bits 23 - 20: Number of C5* sub C-states supported using MWAIT. Bits 27 - 24: Number of C6* sub C-states supported using MWAIT. Bits 31 - 28: Number of C7* sub C-states supported using MWAIT. <b>NOTE:</b> * The definition of C0 through C7 states for MWAIT extension are processor-specific C-states, not ACPI C-states.
<i>Thermal and Power Management Leaf</i>		
06H	EAX	Bit 00: Digital temperature sensor is supported if set. Bit 01: Intel Turbo Boost Technology Available (see description of IA32_MISC_ENABLE[38]). Bit 02: ARAT. APIC-Timer-always-running feature is supported if set. Bit 03: Reserved. Bit 04: PLN. Power limit notification controls are supported if set. Bit 05: ECMD. Clock modulation duty cycle extension is supported if set. Bit 06: PTM. Package thermal management is supported if set. Bit 07: HWP. HWP base registers (IA32_PM_ENABLE[bit 0], IA32_HWP_CAPABILITIES, IA32_HWP_REQUEST, IA32_HWP_STATUS) are supported if set. Bit 08: HWP_Notification. IA32_HWP_INTERRUPT MSR is supported if set. Bit 09: HWP_Activity_Window. IA32_HWP_REQUEST[bits 41:32] is supported if set. Bit 10: HWP_Energy_Performance_Preference. IA32_HWP_REQUEST[bits 31:24] is supported if set. Bit 11: HWP_Package_Level_Request. IA32_HWP_REQUEST_PKG MSR is supported if set. Bit 12: Reserved. Bit 13: HDC. HDC base registers IA32_PKG_HDC_CTL, IA32_PM_CTL1, IA32_THREAD_STALL MSRs are supported if set. Bits 31 - 15: Reserved.
	EBX	Bits 03 - 00: Number of Interrupt Thresholds in Digital Thermal Sensor. Bits 31 - 04: Reserved.
	ECX	Bit 00: Hardware Coordination Feedback Capability (Presence of IA32_MPERF and IA32_APERF). The capability to provide a measure of delivered processor performance (since last reset of the counters), as a percentage of the expected processor performance when running at the TSC frequency. Bits 02 - 01: Reserved = 0. Bit 03: The processor supports performance-energy bias preference if CPUID.06H:ECX.SETBH[bit 3] is set and it also implies the presence of a new architectural MSR called IA32_ENERGY_PERF_BIAS (1B0H). Bits 31 - 04: Reserved = 0.
	EDX	Reserved = 0.

**Table 3-7. Information Returned by CPUID Instruction (Contd.)**

Initial EAX Value	Information Provided about the Processor
<i>Structured Extended Feature Flags Enumeration Leaf (Output depends on ECX input value)</i>	
07H	<p style="text-align: center;">Sub-leaf 0 (Input ECX = 0). *</p> <p>EAX      Bits 31 - 00: Reports the maximum input value for supported leaf 7 sub-leaves.</p> <p>EBX      Bit 00: FSGSBASE. Supports RDFSBASE/RDGSBASE/WRFSBASE/WRGSBASE if 1.            Bit 01: IA32_TSC_ADJUST MSR is supported if 1.            Bit 02: SGX. Supports Intel® Software Guard Extensions (Intel® SGX Extensions) if 1.            Bit 03: BMI1.            Bit 04: HLE.            Bit 05: AVX2.            Bit 06: FDP_EXCPTN_ONLY. x87 FPU Data Pointer updated only on x87 exceptions if 1.            Bit 07: SMEP. Supports Supervisor-Mode Execution Prevention if 1.            Bit 08: BMI2.            Bit 09: Supports Enhanced REP MOVSB/STOSB if 1.            Bit 10: INVPCID. If 1, supports INVPCID instruction for system software that manages process-context identifiers.            Bit 11: RTM.            Bit 12: RDT-M. Supports Intel® Resource Director Technology (Intel® RDT) Monitoring capability if 1.            Bit 13: Deprecates FPU CS and FPU DS values if 1.            Bit 14: MPX. Supports Intel® Memory Protection Extensions if 1.            Bit 15: RDT-A. Supports Intel® Resource Director Technology (Intel® RDT) Allocation capability if 1.            Bits 17:16: Reserved.            Bit 18: RDSEED.            Bit 19: ADX.            Bit 20: SMAP. Supports Supervisor-Mode Access Prevention (and the CLAC/STAC instructions) if 1.            Bits 22 - 21: Reserved.            Bit 23: CLFLUSHOPT.            Bit 24: Reserved.            Bit 25: Intel Processor Trace.            Bits 28 - 26: Reserved.            Bit 29: SHA. supports Intel® Secure Hash Algorithm Extensions (Intel® SHA Extensions) if 1.            Bits 31 - 30: Reserved.</p> <p>ECX      Bit 00: PREFETCHWT1.            Bit 01: Reserved.            Bit 02: UMIP. Supports user-mode instruction prevention if 1.            Bit 03: PKU. Supports protection keys for user-mode pages if 1.            Bit 04: OSPKE. If 1, OS has set CR4.PKE to enable protection keys (and the RDPKRU/WRPKRU instructions).            Bits 16 - 5: Reserved.            Bits 21 - 17: The value of MAWAU used by the BNDLDX and BNDSTX instructions in 64-bit mode.            Bit 22: RDPID. Supports Read Processor ID if 1.            Bits 29 - 23: Reserved.            Bit 30: SGX_LC. Supports SGX Launch Configuration if 1.            Bit 31: Reserved.</p> <p>EDX      Reserved.</p> <p><b>NOTE:</b>            * If ECX contains an invalid sub-leaf index, EAX/EBX/ECX/EDX return 0. Sub-leaf index n is invalid if n exceeds the value that sub-leaf 0 returns in EAX.</p>

Table 3-7. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value	Information Provided about the Processor	
<i>Direct Cache Access Information Leaf</i>		
09H	EAX	Value of bits [31:0] of IA32_PLATFORM_DCA_CAP MSR (address 1F8H).
	EBX	Reserved.
	ECX	Reserved.
	EDX	Reserved.
<i>Architectural Performance Monitoring Leaf</i>		
0AH	EAX	Bits 07 - 00: Version ID of architectural performance monitoring. Bits 15 - 08: Number of general-purpose performance monitoring counter per logical processor. Bits 23 - 16: Bit width of general-purpose, performance monitoring counter. Bits 31 - 24: Length of EBX bit vector to enumerate architectural performance monitoring events.
	EBX	Bit 00: Core cycle event not available if 1. Bit 01: Instruction retired event not available if 1. Bit 02: Reference cycles event not available if 1. Bit 03: Last-level cache reference event not available if 1. Bit 04: Last-level cache misses event not available if 1. Bit 05: Branch instruction retired event not available if 1. Bit 06: Branch mispredict retired event not available if 1. Bits 31 - 07: Reserved = 0.
	ECX	Reserved = 0.
	EDX	Bits 04 - 00: Number of fixed-function performance counters (if Version ID > 1). Bits 12 - 05: Bit width of fixed-function performance counters (if Version ID > 1). Reserved = 0.
<i>Extended Topology Enumeration Leaf</i>		
0BH	<p><b>NOTES:</b></p> <p>Most of Leaf 0BH output depends on the initial value in ECX.</p> <p>The EDX output of leaf 0BH is always valid and does not vary with input value in ECX.</p> <p>Output value in ECX[7:0] always equals input value in ECX[7:0].</p> <p>For sub-leaves that return an invalid level-type of 0 in ECX[15:8]; EAX and EBX will return 0.</p> <p>If an input value n in ECX returns the invalid level-type of 0 in ECX[15:8], other input values with ECX &gt; n also return 0 in ECX[15:8].</p>	
	EAX	Bits 04 - 00: Number of bits to shift right on x2APIC ID to get a unique topology ID of the next level type*. All logical processors with the same next level ID share current level. Bits 31 - 05: Reserved.
	EBX	Bits 15 - 00: Number of logical processors at this level type. The number reflects configuration as shipped by Intel**. Bits 31 - 16: Reserved.
	ECX	Bits 07 - 00: Level number. Same value in ECX input. Bits 15 - 08: Level type***. Bits 31 - 16: Reserved.
	EDX	Bits 31 - 00: x2APIC ID the current logical processor.
	<p><b>NOTES:</b></p> <p>* Software should use this field (EAX[4:0]) to enumerate processor topology of the system.</p>	

**Table 3-7. Information Returned by CPUID Instruction (Contd.)**

Initial EAX Value	Information Provided about the Processor	
	<p>** Software must not use EBX[15:0] to enumerate processor topology of the system. This value in this field (EBX[15:0]) is only intended for display/diagnostic purposes. The actual number of logical processors available to BIOS/OS/Applications may be different from the value of EBX[15:0], depending on software and platform hardware configurations.</p> <p>*** The value of the "level type" field is not related to level numbers in any way, higher "level type" values do not mean higher levels. Level type field has the following encoding:                      0: Invalid.                      1: SMT.                      2: Core.                      3-255: Reserved.</p>	
<i>Processor Extended State Enumeration Main Leaf (EAX = 0DH, ECX = 0)</i>		
0DH		<p><b>NOTES:</b>                      Leaf 0DH main leaf (ECX = 0).</p> <p>EAX     Bits 31 - 00: Reports the supported bits of the lower 32 bits of XCRO. XCRO[n] can be set to 1 only if EAX[n] is 1.                      Bit 00: x87 state.                      Bit 01: SSE state.                      Bit 02: AVX state.                      Bits 04 - 03: MPX state.                      Bits 07 - 05: AVX-512 state.                      Bit 08: Used for IA32_XSS.                      Bit 09: PKRU state.                      Bits 31 - 10: Reserved.</p> <p>EBX     Bits 31 - 00: Maximum size (bytes, from the beginning of the XSAVE/XRSTOR save area) required by enabled features in XCRO. May be different than ECX if some features at the end of the XSAVE save area are not enabled.</p> <p>ECX     Bit 31 - 00: Maximum size (bytes, from the beginning of the XSAVE/XRSTOR save area) of the XSAVE/XRSTOR save area required by all supported features in the processor, i.e., all the valid bit fields in XCRO.</p> <p>EDX     Bit 31 - 00: Reports the supported bits of the upper 32 bits of XCRO. XCRO[n+32] can be set to 1 only if EDX[n] is 1.                      Bits 31 - 00: Reserved.</p>
<i>Processor Extended State Enumeration Sub-leaf (EAX = 0DH, ECX = 1)</i>		
0DH		<p>EAX     Bit 00: XSAVEOPT is available.                      Bit 01: Supports XSAVEC and the compacted form of XRSTOR if set.                      Bit 02: Supports XGETBV with ECX = 1 if set.                      Bit 03: Supports XSAVEES/XRSTORS and IA32_XSS if set.                      Bits 31 - 04: Reserved.</p> <p>EBX     Bits 31 - 00: The size in bytes of the XSAVE area containing all states enabled by XCRO   IA32_XSS.</p> <p>ECX     Bits 31 - 00: Reports the supported bits of the lower 32 bits of the IA32_XSS MSR. IA32_XSS[n] can be set to 1 only if ECX[n] is 1.                      Bits 07 - 00: Used for XCRO.                      Bit 08: PT state.                      Bit 09: Used for XCRO.                      Bits 31 - 10: Reserved.</p> <p>EDX     Bits 31 - 00: Reports the supported bits of the upper 32 bits of the IA32_XSS MSR. IA32_XSS[n+32] can be set to 1 only if EDX[n] is 1.                      Bits 31 - 00: Reserved.</p>

**Table 3-7. Information Returned by CPUID Instruction (Contd.)**

Initial EAX Value	Information Provided about the Processor
<i>Processor Extended State Enumeration Sub-leaves (EAX = 0DH, ECX = n, n &gt; 1)</i>	
0DH	<p><b>NOTES:</b>  Leaf 0DH output depends on the initial value in ECX.  Each sub-leaf index (starting at position 2) is supported if it corresponds to a supported bit in either the XCRO register or the IA32_XSS MSR.  * If ECX contains an invalid sub-leaf index, EAX/EBX/ECX/EDX return 0. Sub-leaf n (<math>0 \leq n \leq 31</math>) is invalid if sub-leaf 0 returns 0 in EAX[n] and sub-leaf 1 returns 0 in ECX[n]. Sub-leaf n (<math>32 \leq n \leq 63</math>) is invalid if sub-leaf 0 returns 0 in EDX[n-32] and sub-leaf 1 returns 0 in EDX[n-32].</p> <p>EAX    Bits 31 - 0: The size in bytes (from the offset specified in EBX) of the save area for an extended state feature associated with a valid sub-leaf index, <i>n</i>.</p> <p>EBX    Bits 31 - 0: The offset in bytes of this extended state component's save area from the beginning of the XSAVE/XRSTOR area.  This field reports 0 if the sub-leaf index, <i>n</i>, does not map to a valid bit in the XCRO register*.</p> <p>ECX    Bit 00 is set if the bit <i>n</i> (corresponding to the sub-leaf index) is supported in the IA32_XSS MSR; it is clear if bit <i>n</i> is instead supported in XCRO.  Bit 01 is set if, when the compacted format of an XSAVE area is used, this extended state component located on the next 64-byte boundary following the preceding state component (otherwise, it is located immediately following the preceding state component).  Bits 31 - 02 are reserved.  This field reports 0 if the sub-leaf index, <i>n</i>, is invalid*.</p> <p>EDX    This field reports 0 if the sub-leaf index, <i>n</i>, is invalid*; otherwise it is reserved.</p>
<i>Intel Resource Director Technology (Intel RDT) Monitoring Enumeration Sub-leaf (EAX = 0FH, ECX = 0)</i>	
0FH	<p><b>NOTES:</b>  Leaf 0FH output depends on the initial value in ECX.  Sub-leaf index 0 reports valid resource type starting at bit position 1 of EDX.</p> <p>EAX    Reserved.</p> <p>EBX    Bits 31 - 00: Maximum range (zero-based) of RMID within this physical processor of all types.</p> <p>ECX    Reserved.</p> <p>EDX    Bit 00: Reserved.  Bit 01: Supports L3 Cache Intel RDT Monitoring if 1.  Bits 31 - 02: Reserved.</p>
<i>L3 Cache Intel RDT Monitoring Capability Enumeration Sub-leaf (EAX = 0FH, ECX = 1)</i>	
0FH	<p><b>NOTES:</b>  Leaf 0FH output depends on the initial value in ECX.</p> <p>EAX    Reserved.</p> <p>EBX    Bits 31 - 00: Conversion factor from reported IA32_QM_CTR value to occupancy metric (bytes).</p> <p>ECX    Maximum range (zero-based) of RMID of this resource type.</p> <p>EDX    Bit 00: Supports L3 occupancy monitoring if 1.  Bit 01: Supports L3 Total Bandwidth monitoring if 1.  Bit 02: Supports L3 Local Bandwidth monitoring if 1.  Bits 31 - 03: Reserved.</p>

**Table 3-7. Information Returned by CPUID Instruction (Contd.)**

Initial EAX Value	Information Provided about the Processor
<i>Intel Resource Director Technology (Intel RDT) Allocation Enumeration Sub-leaf (EAX = 10H, ECX = 0)</i>	
10H	<p><b>NOTES:</b>            Leaf 10H output depends on the initial value in ECX.            Sub-leaf index 0 reports valid resource identification (ResID) starting at bit position 1 of EBX.</p> <p>EAX      Reserved.            EBX      Bit 00: Reserved.                      Bit 01: Supports L3 Cache Allocation Technology if 1.                      Bit 02: Supports L2 Cache Allocation Technology if 1.                      Bits 31 - 03: Reserved.            ECX      Reserved.            EDX      Reserved.</p>
<i>L3 Cache Allocation Technology Enumeration Sub-leaf (EAX = 10H, ECX = ResID =1)</i>	
10H	<p><b>NOTES:</b>            Leaf 10H output depends on the initial value in ECX.</p> <p>EAX      Bits 4 - 00: Length of the capacity bit mask for the corresponding ResID using minus-one notation.                      Bits 31 - 05: Reserved.            EBX      Bits 31 - 00: Bit-granular map of isolation/contention of allocation units.            ECX      Bit 00: Reserved.                      Bit 01: Updates of COS should be infrequent if 1.                      Bit 02: Code and Data Prioritization Technology supported if 1.                      Bits 31 - 03: Reserved.            EDX      Bits 15 - 00: Highest COS number supported for this ResID.                      Bits 31 - 16: Reserved.</p>
<i>L2 Cache Allocation Technology Enumeration Sub-leaf (EAX = 10H, ECX = ResID =2)</i>	
10H	<p><b>NOTES:</b>            Leaf 10H output depends on the initial value in ECX.</p> <p>EAX      Bits 4 - 00: Length of the capacity bit mask for the corresponding ResID using minus-one notation.                      Bits 31 - 05: Reserved.            EBX      Bits 31 - 00: Bit-granular map of isolation/contention of allocation units.            ECX      Bits 31 - 00: Reserved.            EDX      Bits 15 - 00: Highest COS number supported for this ResID.                      Bits 31 - 16: Reserved.</p>
<i>Intel SGX Capability Enumeration Leaf, sub-leaf 0 (EAX = 12H, ECX = 0)</i>	
12H	<p><b>NOTES:</b>            Leaf 12H sub-leaf 0 (ECX = 0) is supported if CPUID.(EAX=07H, ECX=0H);EBX[SGX] = 1.</p> <p>EAX      Bit 00: SGX1. If 1, Indicates Intel SGX supports the collection of SGX1 leaf functions.                      Bit 01: SGX2. If 1, Indicates Intel SGX supports the collection of SGX2 leaf functions.                      Bit 31 - 02: Reserved.            EBX      Bit 31 - 00: MISCSELECT. Bit vector of supported extended SGX features.            ECX      Bit 31 - 00: Reserved.</p>



**Table 3-7. Information Returned by CPUID Instruction (Contd.)**

Initial EAX Value	Information Provided about the Processor
	EDX Bit 07 - 00: MaxEnclaveSize_Not64. The maximum supported enclave size in non-64-bit mode is $2^{(EDX[7:0])}$ . Bit 15 - 08: MaxEnclaveSize_64. The maximum supported enclave size in 64-bit mode is $2^{(EDX[15:8])}$ . Bits 31 - 16: Reserved.
<i>Intel SGX Attributes Enumeration Leaf, sub-leaf 1 (EAX = 12H, ECX = 1)</i>	
12H	<p><b>NOTES:</b>            Leaf 12H sub-leaf 1 (ECX = 1) is supported if CPUID.(EAX=07H, ECX=0H):EBX[SGX] = 1.</p> EAX Bit 31 - 00: Reports the valid bits of SECS.ATTRIBUTES[31:0] that software can set with ECREATE. EBX Bit 31 - 00: Reports the valid bits of SECS.ATTRIBUTES[63:32] that software can set with ECREATE. ECX Bit 31 - 00: Reports the valid bits of SECS.ATTRIBUTES[95:64] that software can set with ECREATE. EDX Bit 31 - 00: Reports the valid bits of SECS.ATTRIBUTES[127:96] that software can set with ECREATE.
<i>Intel SGX EPC Enumeration Leaf, sub-leaves (EAX = 12H, ECX = 2 or higher)</i>	
12H	<p><b>NOTES:</b>            Leaf 12H sub-leaf 2 or higher (ECX &gt;= 2) is supported if CPUID.(EAX=07H, ECX=0H):EBX[SGX] = 1.            For sub-leaves (ECX = 2 or higher), definition of EDX,ECX,EBX,EAX[31:4] depends on the sub-leaf type listed below.</p> EAX Bit 03 - 00: Sub-leaf Type 0000b: Indicates this sub-leaf is invalid. 0001b: This sub-leaf enumerates an EPC section. EBX:EAX and EDX:ECX provide information on the Enclave Page Cache (EPC) section. All other type encodings are reserved.           Type 0000b. This sub-leaf is invalid. EDX:ECX:EBX:EAX return 0.           Type 0001b. This sub-leaf enumerates an EPC sections with EDX:ECX, EBX:EAX defined as follows. EAX[11:04]: Reserved (enumerate 0). EAX[31:12]: Bits 31:12 of the physical address of the base of the EPC section. EBX[19:00]: Bits 51:32 of the physical address of the base of the EPC section. EBX[31:20]: Reserved. ECX[03:00]: EPC section property encoding defined as follows: If EAX[3:0] 0000b, then all bits of the EDX:ECX pair are enumerated as 0. If EAX[3:0] 0001b, then this section has confidentiality and integrity protection. All other encodings are reserved. ECX[11:04]: Reserved (enumerate 0). ECX[31:12]: Bits 31:12 of the size of the corresponding EPC section within the Processor Reserved Memory. EDX[19:00]: Bits 51:32 of the size of the corresponding EPC section within the Processor Reserved Memory. EDX[31:20]: Reserved.

**Table 3-7. Information Returned by CPUID Instruction (Contd.)**

Initial EAX Value	Information Provided about the Processor	
<i>Intel Processor Trace Enumeration Main Leaf (EAX = 14H, ECX = 0)</i>		
14H		<p><b>NOTES:</b> Leaf 14H main leaf (ECX = 0).</p> <p>EAX Bits 31 - 00: Reports the maximum sub-leaf supported in leaf 14H.</p> <p>EBX Bit 00: If 1, indicates that IA32_RTIT_CTL.CR3Filter can be set to 1, and that IA32_RTIT_CR3_MATCH MSR can be accessed. Bit 01: If 1, indicates support of Configurable PSB and Cycle-Accurate Mode. Bit 02: If 1, indicates support of IP Filtering, TraceStop filtering, and preservation of Intel PT MSRs across warm reset. Bit 03: If 1, indicates support of MTC timing packet and suppression of COFI-based packets. Bit 04: If 1, indicates support of PTWRITE. Writes can set IA32_RTIT_CTL[12] (PTWEn) and IA32_RTIT_CTL[5] (FUPonPTw), and PTWRITE can generate packets. Bit 05: If 1, indicates support of Power Event Trace. Writes can set IA32_RTIT_CTL[4] (PwrEvtEn), enabling Power Event Trace packet generation. Bit 31 - 06: Reserved.</p> <p>ECX Bit 00: If 1, Tracing can be enabled with IA32_RTIT_CTL.ToPA = 1, hence utilizing the ToPA output scheme; IA32_RTIT_OUTPUT_BASE and IA32_RTIT_OUTPUT_MASK_PTRS MSRs can be accessed. Bit 01: If 1, ToPA tables can hold any number of output entries, up to the maximum allowed by the MaskOrTableOffset field of IA32_RTIT_OUTPUT_MASK_PTRS. Bit 02: If 1, indicates support of Single-Range Output scheme. Bit 03: If 1, indicates support of output to Trace Transport subsystem. Bit 30 - 04: Reserved. Bit 31: If 1, generated packets which contain IP payloads have LIP values, which include the CS base component.</p> <p>EDX Bits 31 - 00: Reserved.</p>
<i>Intel Processor Trace Enumeration Sub-leaf (EAX = 14H, ECX = 1)</i>		
14H	EAX	Bits 02 - 00: Number of configurable Address Ranges for filtering. Bits 15 - 03: Reserved. Bits 31 - 16: Bitmap of supported MTC period encodings.
	EBX	Bits 15 - 00: Bitmap of supported Cycle Threshold value encodings. Bit 31 - 16: Bitmap of supported Configurable PSB frequency encodings.
	ECX	Bits 31 - 00: Reserved.
	EDX	Bits 31 - 00: Reserved.
<i>Time Stamp Counter/Core Crystal Clock Information-leaf</i>		
15H		<p><b>NOTES:</b> If EBX[31:0] is 0, the TSC/"core crystal clock" ratio is not enumerated. EBX[31:0]/EAX[31:0] indicates the ratio of the TSC frequency and the core crystal clock frequency. "TSC frequency" = "core crystal clock frequency" * EBX/EAX. The core crystal clock may differ from the reference clock, bus clock, or core clock frequencies.</p> <p>EAX Bits 31 - 00: An unsigned integer which is the denominator of the TSC/"core crystal clock" ratio.</p> <p>EBX Bits 31 - 00: An unsigned integer which is the numerator of the TSC/"core crystal clock" ratio.</p> <p>ECX Bits 31 - 00: Reserved = 0.</p> <p>EDX Bits 31 - 00: Reserved = 0.</p>

Table 3-7. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value	Information Provided about the Processor	
<i>Processor Frequency Information Leaf</i>		
16H	EAX	Bits 15 - 00: Processor Base Frequency (in MHz). Bits 31 - 16: Reserved = 0.
	EBX	Bits 15 - 00: Maximum Frequency (in MHz). Bits 31 - 16: Reserved = 0.
	ECX	Bits 15 - 00: Bus (Reference) Frequency (in MHz). Bits 31 - 16: Reserved = 0.
	EDX	Reserved.
<p><b>NOTES:</b></p> <p>* Data is returned from this interface in accordance with the processor's specification and does not reflect actual values. Suitable use of this data includes the display of processor information in like manner to the processor brand string and for determining the appropriate range to use when displaying processor information e.g. frequency history graphs. The returned information should not be used for any other purpose as the returned information does not accurately correlate to information / counters returned by other processor interfaces.</p> <p>While a processor may support the Processor Frequency Information leaf, fields that return a value of zero are not supported.</p>		
<i>System-On-Chip Vendor Attribute Enumeration Main Leaf (EAX = 17H, ECX = 0)</i>		
17H		<p><b>NOTES:</b></p> <p>Leaf 17H main leaf (ECX = 0). Leaf 17H output depends on the initial value in ECX. Leaf 17H sub-leaves 1 through 3 reports SOC Vendor Brand String. Leaf 17H is valid if MaxSOCID_Index &gt;= 3. Leaf 17H sub-leaves 4 and above are reserved.</p> <p>EAX Bits 31 - 00: MaxSOCID_Index. Reports the maximum input value of supported sub-leaf in leaf 17H.</p> <p>EBX Bits 15 - 00: SOC Vendor ID. Bit 16: IsVendorScheme. If 1, the SOC Vendor ID field is assigned via an industry standard enumeration scheme. Otherwise, the SOC Vendor ID field is assigned by Intel. Bits 31 - 17: Reserved = 0.</p> <p>ECX Bits 31 - 00: Project ID. A unique number an SOC vendor assigns to its SOC projects.</p> <p>EDX Bits 31 - 00: Stepping ID. A unique number within an SOC project that an SOC vendor assigns.</p>
<i>System-On-Chip Vendor Attribute Enumeration Sub-leaf (EAX = 17H, ECX = 1..3)</i>		
17H	EAX	Bit 31 - 00: SOC Vendor Brand String. UTF-8 encoded string.
	EBX	Bit 31 - 00: SOC Vendor Brand String. UTF-8 encoded string.
	ECX	Bit 31 - 00: SOC Vendor Brand String. UTF-8 encoded string.
	EDX	Bit 31 - 00: SOC Vendor Brand String. UTF-8 encoded string.
<p><b>NOTES:</b></p> <p>Leaf 17H output depends on the initial value in ECX. SOC Vendor Brand String is a UTF-8 encoded string padded with trailing bytes of 00H. The complete SOC Vendor Brand String is constructed by concatenating in ascending order of EAX:EBX:ECX:EDX and from the sub-leaf 1 fragment towards sub-leaf 3.</p>		

Table 3-7. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value	Information Provided about the Processor	
<i>System-On-Chip Vendor Attribute Enumeration Sub-leaves (EAX = 17H, ECX &gt; MaxSOCID_Index)</i>		
17H	<p><b>NOTES:</b> Leaf 17H output depends on the initial value in ECX.</p> <p>EAX      Bits 31 - 00: Reserved = 0. EBX      Bits 31 - 00: Reserved = 0. ECX      Bits 31 - 00: Reserved = 0. EDX      Bits 31 - 00: Reserved = 0.</p>	
<i>Unimplemented CPUID Leaf Functions</i>		
40000000H - 4FFFFFFFH	Invalid. No existing or future CPU will return processor identification or feature information if the initial EAX value is in the range 40000000H to 4FFFFFFFH.	
<i>Extended Function CPUID Information</i>		
80000000H	EAX	Maximum Input Value for Extended Function CPUID Information.
	EBX	Reserved.
	ECX	Reserved.
	EDX	Reserved.
80000001H	EAX	Extended Processor Signature and Feature Bits.
	EBX	Reserved.
	ECX	Bit 00: LAHF/SAHF available in 64-bit mode. Bits 04 - 01: Reserved. Bit 05: LZCNT. Bits 07 - 06: Reserved. Bit 08: PREFETCHW. Bits 31 - 09: Reserved.
	EDX	Bits 10 - 00: Reserved. Bit 11: SYSCALL/SYSRET available in 64-bit mode. Bits 19 - 12: Reserved = 0. Bit 20: Execute Disable Bit available. Bits 25 - 21: Reserved = 0. Bit 26: 1-GByte pages are available if 1. Bit 27: RDTSCP and IA32_TSC_AUX are available if 1. Bit 28: Reserved = 0. Bit 29: Intel® 64 Architecture available if 1. Bits 31 - 30: Reserved = 0.
80000002H	EAX	Processor Brand String.
	EBX	Processor Brand String Continued.
	ECX	Processor Brand String Continued.
	EDX	Processor Brand String Continued.
80000003H	EAX	Processor Brand String Continued.
	EBX	Processor Brand String Continued.
	ECX	Processor Brand String Continued.
	EDX	Processor Brand String Continued.

**Table 3-7. Information Returned by CPUID Instruction (Contd.)**

Initial EAX Value	Information Provided about the Processor	
80000004H	EAX EBX ECX EDX	Processor Brand String Continued. Processor Brand String Continued. Processor Brand String Continued. Processor Brand String Continued.
80000005H	EAX EBX ECX EDX	Reserved = 0. Reserved = 0. Reserved = 0. Reserved = 0.
80000006H	EAX EBX ECX EDX	Reserved = 0. Reserved = 0. Bits 07 - 00: Cache Line size in bytes. Bits 11 - 08: Reserved. Bits 15 - 12: L2 Associativity field *. Bits 31 - 16: Cache size in 1K units. Reserved = 0.  <b>NOTES:</b> * L2 associativity field encodings: 00H - Disabled. 01H - Direct mapped. 02H - 2-way. 04H - 4-way. 06H - 8-way. 08H - 16-way. 0FH - Fully associative.
80000007H	EAX EBX ECX EDX	Reserved = 0. Reserved = 0. Reserved = 0. Bits 07 - 00: Reserved = 0. Bit 08: Invariant TSC available if 1. Bits 31 - 09: Reserved = 0.
80000008H	EAX  EBX ECX EDX	Linear/Physical Address size. Bits 07 - 00: #Physical Address Bits*. Bits 15 - 08: #Linear Address Bits. Bits 31 - 16: Reserved = 0.  Reserved = 0. Reserved = 0. Reserved = 0.  <b>NOTES:</b> * If CPUID.80000008H:EAX[7:0] is supported, the maximum physical address number supported should come from this field.

**INPUT EAX = 0: Returns CPUID's Highest Value for Basic Processor Information and the Vendor Identification String**

When CPUID executes with EAX set to 0, the processor returns the highest value the CPUID recognizes for returning basic processor information. The value is returned in the EAX register and is processor specific.

A vendor identification string is also returned in EBX, EDX, and ECX. For Intel processors, the string is “GenuineIntel” and is expressed:

```
EBX ← 756e6547h (* "Genu", with G in the low eight bits of BL *)
EDX ← 49656e69h (* "inel", with i in the low eight bits of DL *)
ECX ← 6c65746eh (* "ntel", with n in the low eight bits of CL *)
```

**INPUT EAX = 80000000H: Returns CPUID’s Highest Value for Extended Processor Information**

When CPUID executes with EAX set to 80000000H, the processor returns the highest value the processor recognizes for returning extended processor information. The value is returned in the EAX register and is processor specific.

**IA32\_BIOS\_SIGN\_ID Returns Microcode Update Signature**

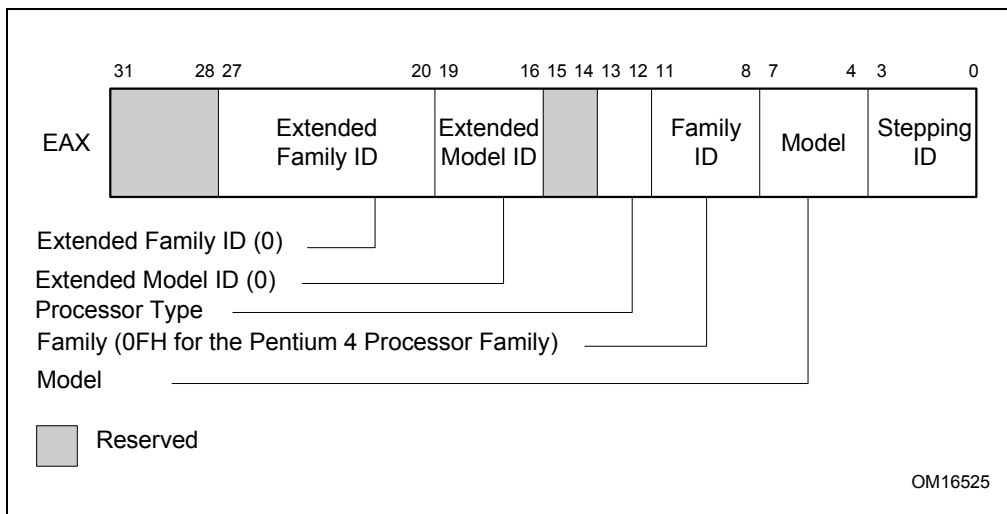
For processors that support the microcode update facility, the IA32\_BIOS\_SIGN\_ID MSR is loaded with the update signature whenever CPUID executes. The signature is returned in the upper DWORD. For details, see Chapter 9 in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

**INPUT EAX = 01H: Returns Model, Family, Stepping Information**

When CPUID executes with EAX set to 01H, version information is returned in EAX (see Figure 3-6). For example: model, family, and processor type for the Intel Xeon processor 5100 series is as follows:

- Model — 1111B
- Family — 0101B
- Processor Type — 00B

See Table 3-9 for available processor type values. Stepping IDs are provided as needed.



**Figure 3-3. Version Information Returned by CPUID in EAX**

**Table 3-8. Processor Type Field**

Type	Encoding
Original OEM Processor	00B
Intel OverDrive™ Processor	01B
Dual processor (not applicable to Intel486 processors)	10B
Intel reserved	11B

**NOTE**

See Chapter 19 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for information on identifying earlier IA-32 processors.

The Extended Family ID needs to be examined only when the Family ID is 0FH. Integrate the fields into a display using the following rule:

```
IF Family_ID ≠ 0FH
  THEN DisplayFamily = Family_ID;
  ELSE DisplayFamily = Extended_Family_ID + Family_ID;
  (* Right justify and zero-extend 4-bit field. *)
FI;
(* Show DisplayFamily as HEX field. *)
```

The Extended Model ID needs to be examined only when the Family ID is 06H or 0FH. Integrate the field into a display using the following rule:

```
IF (Family_ID = 06H or Family_ID = 0FH)
  THEN DisplayModel = (Extended_Model_ID « 4) + Model_ID;
  (* Right justify and zero-extend 4-bit field; display Model_ID as HEX field.*)
  ELSE DisplayModel = Model_ID;
FI;
(* Show DisplayModel as HEX field. *)
```

**INPUT EAX = 01H: Returns Additional Information in EBX**

When CPUID executes with EAX set to 01H, additional information is returned to the EBX register:

- Brand index (low byte of EBX) — this number provides an entry into a brand string table that contains brand strings for IA-32 processors. More information about this field is provided later in this section.
- CLFLUSH instruction cache line size (second byte of EBX) — this number indicates the size of the cache line flushed by the CLFLUSH and CLFLUSHOPT instructions in 8-byte increments. This field was introduced in the Pentium 4 processor.
- Local APIC ID (high byte of EBX) — this number is the 8-bit ID that is assigned to the local APIC on the processor during power up. This field was introduced in the Pentium 4 processor.

**INPUT EAX = 01H: Returns Feature Information in ECX and EDX**

When CPUID executes with EAX set to 01H, feature information is returned in ECX and EDX.

- Figure 3-7 and Table 3-10 show encodings for ECX.
- Figure 3-8 and Table 3-11 show encodings for EDX.

For all feature flags, a 1 indicates that the feature is supported. Use Intel to properly interpret feature flags.

**NOTE**

Software must confirm that a processor feature is present using feature flags returned by CPUID prior to using the feature. Software should not depend on future offerings retaining all features.

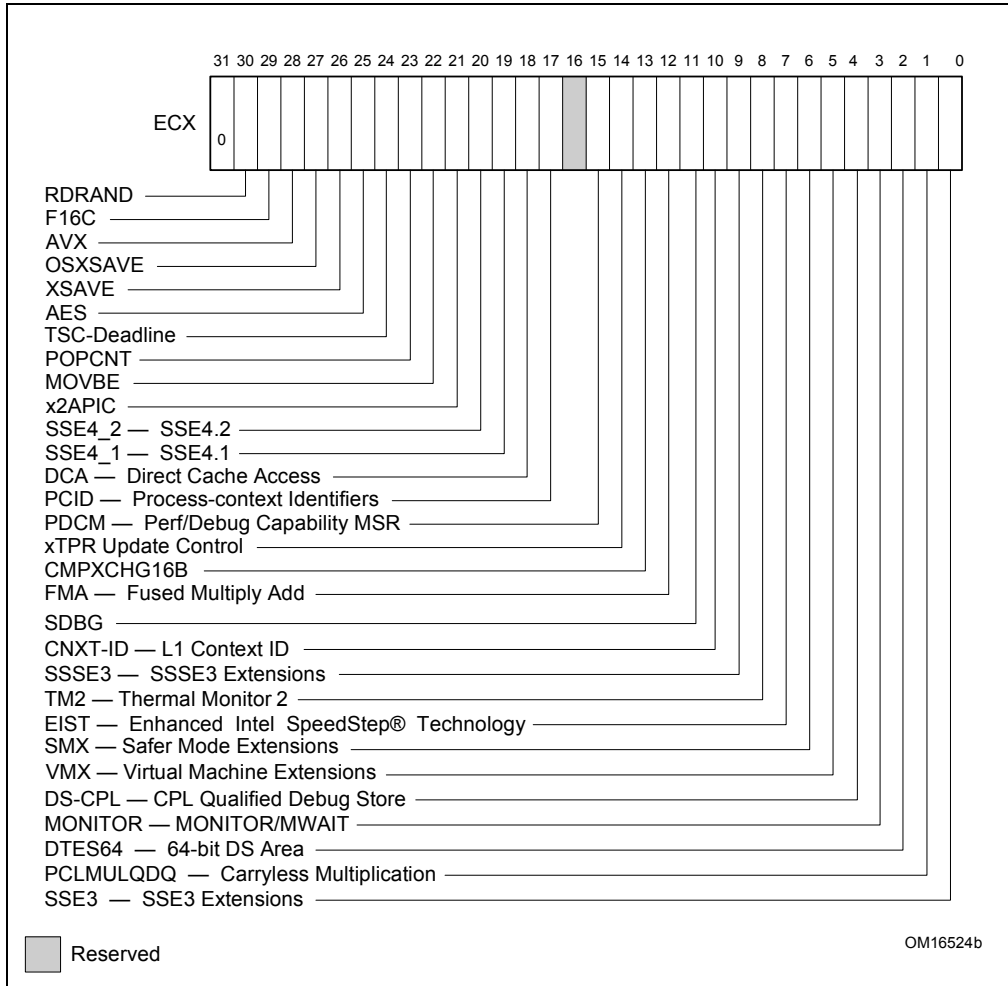


Figure 3-4. Feature Information Returned in the ECX Register

Table 3-9. Feature Information Returned in the ECX Register

Bit #	Mnemonic	Description
0	SSE3	<b>Streaming SIMD Extensions 3 (SSE3).</b> A value of 1 indicates the processor supports this technology.
1	PCLMULQDQ	<b>PCLMULQDQ.</b> A value of 1 indicates the processor supports the PCLMULQDQ instruction.
2	DTES64	<b>64-bit DS Area.</b> A value of 1 indicates the processor supports DS area using 64-bit layout.
3	MONITOR	<b>MONITOR/MWAIT.</b> A value of 1 indicates the processor supports this feature.
4	DS-CPL	<b>CPL Qualified Debug Store.</b> A value of 1 indicates the processor supports the extensions to the Debug Store feature to allow for branch message storage qualified by CPL.
5	VMX	<b>Virtual Machine Extensions.</b> A value of 1 indicates that the processor supports this technology.
6	SMX	<b>Safer Mode Extensions.</b> A value of 1 indicates that the processor supports this technology. See Chapter 6, “Safer Mode Extensions Reference”.
7	EIST	<b>Enhanced Intel SpeedStep® technology.</b> A value of 1 indicates that the processor supports this technology.
8	TM2	<b>Thermal Monitor 2.</b> A value of 1 indicates whether the processor supports this technology.
9	SSSE3	A value of 1 indicates the presence of the Supplemental Streaming SIMD Extensions 3 (SSSE3). A value of 0 indicates the instruction extensions are not present in the processor.



Table 3-9. Feature Information Returned in the ECX Register (Contd.)

Bit #	Mnemonic	Description
10	CNXT-ID	<b>L1 Context ID.</b> A value of 1 indicates the L1 data cache mode can be set to either adaptive mode or shared mode. A value of 0 indicates this feature is not supported. See definition of the IA32_MISC_ENABLE MSR Bit 24 (L1 Data Cache Context Mode) for details.
11	SDBG	A value of 1 indicates the processor supports IA32_DEBUG_INTERFACE MSR for silicon debug.
12	FMA	A value of 1 indicates the processor supports FMA extensions using YMM state.
13	CMPXCHG16B	<b>CMPXCHG16B Available.</b> A value of 1 indicates that the feature is available. See the “CMPXCHG8B/CMPXCHG16B—Compare and Exchange Bytes” section in this chapter for a description.
14	xTPR Update Control	<b>xTPR Update Control.</b> A value of 1 indicates that the processor supports changing IA32_MISC_ENABLE[bit 23].
15	PDCM	<b>Perfmon and Debug Capability:</b> A value of 1 indicates the processor supports the performance and debug feature indication MSR IA32_PERF_CAPABILITIES.
16	Reserved	Reserved
17	PCID	<b>Process-context identifiers.</b> A value of 1 indicates that the processor supports PCIDs and that software may set CR4.PCIDE to 1.
18	DCA	A value of 1 indicates the processor supports the ability to prefetch data from a memory mapped device.
19	SSE4.1	A value of 1 indicates that the processor supports SSE4.1.
20	SSE4.2	A value of 1 indicates that the processor supports SSE4.2.
21	x2APIC	A value of 1 indicates that the processor supports x2APIC feature.
22	MOVBE	A value of 1 indicates that the processor supports MOVBE instruction.
23	POPCNT	A value of 1 indicates that the processor supports the POPCNT instruction.
24	TSC-Deadline	A value of 1 indicates that the processor’s local APIC timer supports one-shot operation using a TSC deadline value.
25	AESNI	A value of 1 indicates that the processor supports the AESNI instruction extensions.
26	XSAVE	A value of 1 indicates that the processor supports the XSAVE/XRSTOR processor extended states feature, the XSETBV/XGETBV instructions, and XCR0.
27	OSXSAVE	A value of 1 indicates that the OS has set CR4.OSXSAVE[bit 18] to enable XSETBV/XGETBV instructions to access XCR0 and to support processor extended state management using XSAVE/XRSTOR.
28	AVX	A value of 1 indicates the processor supports the AVX instruction extensions.
29	F16C	A value of 1 indicates that processor supports 16-bit floating-point conversion instructions.
30	RDRAND	A value of 1 indicates that processor supports RDRAND instruction.
31	Not Used	Always returns 0.

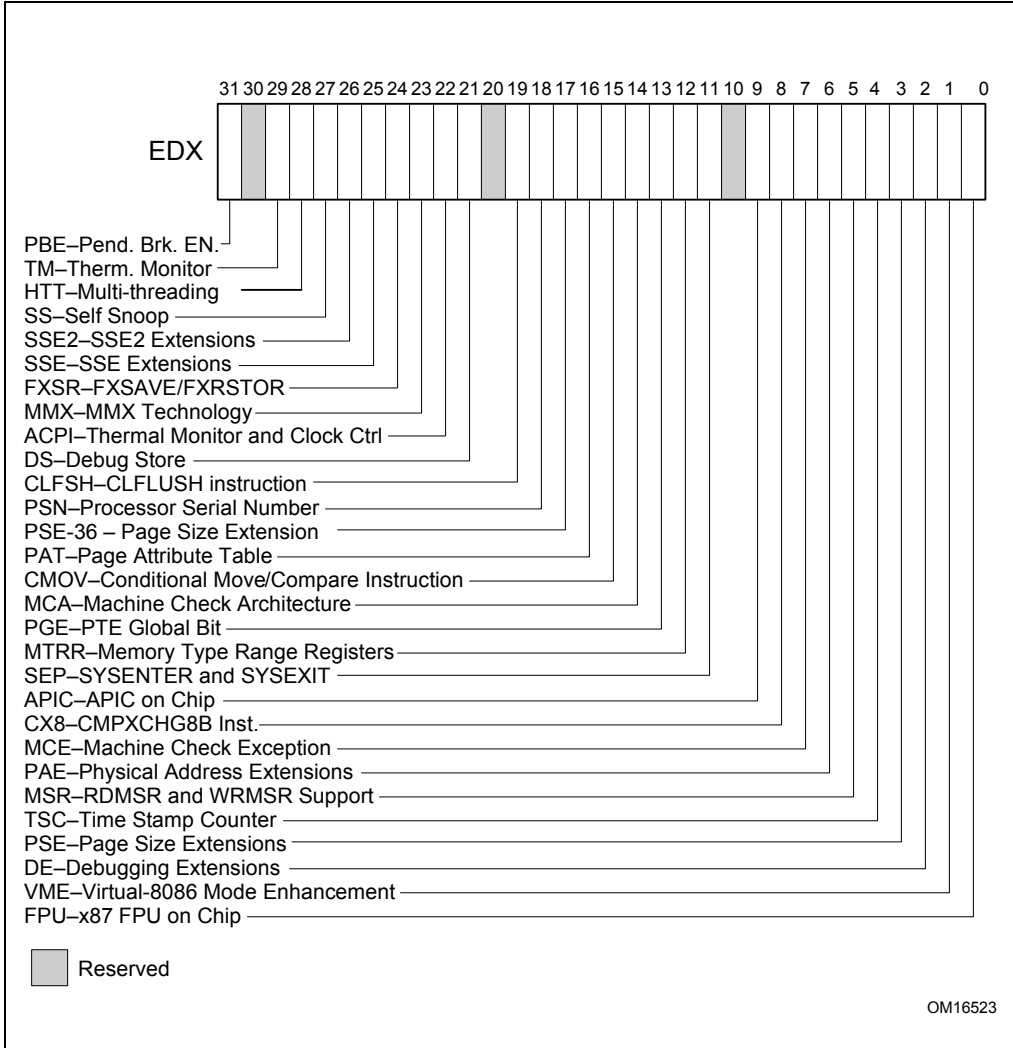


Figure 3-5. Feature Information Returned in the EDX Register

**Table 3-10. More on Feature Information Returned in the EDX Register**

Bit #	Mnemonic	Description
0	FPU	<b>Floating Point Unit On-Chip.</b> The processor contains an x87 FPU.
1	VME	<b>Virtual 8086 Mode Enhancements.</b> Virtual 8086 mode enhancements, including CR4.VME for controlling the feature, CR4.PVI for protected mode virtual interrupts, software interrupt indirection, expansion of the TSS with the software indirection bitmap, and EFLAGS.VIF and EFLAGS.VIP flags.
2	DE	<b>Debugging Extensions.</b> Support for I/O breakpoints, including CR4.DE for controlling the feature, and optional trapping of accesses to DR4 and DR5.
3	PSE	<b>Page Size Extension.</b> Large pages of size 4 MByte are supported, including CR4.PSE for controlling the feature, the defined dirty bit in PDE (Page Directory Entries), optional reserved bit trapping in CR3, PDEs, and PTEs.
4	TSC	<b>Time Stamp Counter.</b> The RDTSC instruction is supported, including CR4.TSD for controlling privilege.
5	MSR	<b>Model Specific Registers RDMSR and WRMSR Instructions.</b> The RDMSR and WRMSR instructions are supported. Some of the MSRs are implementation dependent.
6	PAE	<b>Physical Address Extension.</b> Physical addresses greater than 32 bits are supported: extended page table entry formats, an extra level in the page translation tables is defined, 2-MByte pages are supported instead of 4 Mbyte pages if PAE bit is 1.
7	MCE	<b>Machine Check Exception.</b> Exception 18 is defined for Machine Checks, including CR4.MCE for controlling the feature. This feature does not define the model-specific implementations of machine-check error logging, reporting, and processor shutdowns. Machine Check exception handlers may have to depend on processor version to do model specific processing of the exception, or test for the presence of the Machine Check feature.
8	CX8	<b>CMPXCHG8B Instruction.</b> The compare-and-exchange 8 bytes (64 bits) instruction is supported (implicitly locked and atomic).
9	APIC	<b>APIC On-Chip.</b> The processor contains an Advanced Programmable Interrupt Controller (APIC), responding to memory mapped commands in the physical address range FFFE0000H to FFFE0FFFH (by default - some processors permit the APIC to be relocated).
10	Reserved	Reserved
11	SEP	<b>SYSENTER and SYSEXIT Instructions.</b> The SYSENTER and SYSEXIT and associated MSRs are supported.
12	MTRR	<b>Memory Type Range Registers.</b> MTRRs are supported. The MTRRcap MSR contains feature bits that describe what memory types are supported, how many variable MTRRs are supported, and whether fixed MTRRs are supported.
13	PGE	<b>Page Global Bit.</b> The global bit is supported in paging-structure entries that map a page, indicating TLB entries that are common to different processes and need not be flushed. The CR4.PGE bit controls this feature.
14	MCA	<b>Machine Check Architecture.</b> A value of 1 indicates the Machine Check Architecture of reporting machine errors is supported. The MCG_CAP MSR contains feature bits describing how many banks of error reporting MSRs are supported.
15	CMOV	<b>Conditional Move Instructions.</b> The conditional move instruction CMOV is supported. In addition, if x87 FPU is present as indicated by the CPUID.FPU feature bit, then the FCOMI and FCMOV instructions are supported
16	PAT	<b>Page Attribute Table.</b> Page Attribute Table is supported. This feature augments the Memory Type Range Registers (MTRRs), allowing an operating system to specify attributes of memory accessed through a linear address on a 4KB granularity.
17	PSE-36	<b>36-Bit Page Size Extension.</b> 4-MByte pages addressing physical memory beyond 4 GBytes are supported with 32-bit paging. This feature indicates that upper bits of the physical address of a 4-MByte page are encoded in bits 20:13 of the page-directory entry. Such physical addresses are limited by MAXPHYADDR and may be up to 40 bits in size.
18	PSN	<b>Processor Serial Number.</b> The processor supports the 96-bit processor identification number feature and the feature is enabled.
19	CLFSH	<b>CLFLUSH Instruction.</b> CLFLUSH Instruction is supported.
20	Reserved	Reserved

**Table 3-10. More on Feature Information Returned in the EDX Register (Contd.)**

Bit #	Mnemonic	Description
21	DS	<b>Debug Store.</b> The processor supports the ability to write debug information into a memory resident buffer. This feature is used by the branch trace store (BTS) and precise event-based sampling (PEBS) facilities (see Chapter 23, "Introduction to Virtual-Machine Extensions," in the <i>Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C</i> ).
22	ACPI	<b>Thermal Monitor and Software Controlled Clock Facilities.</b> The processor implements internal MSRs that allow processor temperature to be monitored and processor performance to be modulated in predefined duty cycles under software control.
23	MMX	<b>Intel MMX Technology.</b> The processor supports the Intel MMX technology.
24	FXSR	<b>FXSAVE and FXRSTOR Instructions.</b> The FXSAVE and FXRSTOR instructions are supported for fast save and restore of the floating point context. Presence of this bit also indicates that CR4.OSFXSR is available for an operating system to indicate that it supports the FXSAVE and FXRSTOR instructions.
25	SSE	<b>SSE.</b> The processor supports the SSE extensions.
26	SSE2	<b>SSE2.</b> The processor supports the SSE2 extensions.
27	SS	<b>Self Snoop.</b> The processor supports the management of conflicting memory types by performing a snoop of its own cache structure for transactions issued to the bus.
28	HTT	<b>Max APIC IDs reserved field is Valid.</b> A value of 0 for HTT indicates there is only a single logical processor in the package and software should assume only a single APIC ID is reserved. A value of 1 for HTT indicates the value in CPUID.1.EBX[23:16] (the Maximum number of addressable IDs for logical processors in this package) is valid for the package.
29	TM	<b>Thermal Monitor.</b> The processor implements the thermal monitor automatic thermal control circuitry (TCC).
30	Reserved	Reserved
31	PBE	<b>Pending Break Enable.</b> The processor supports the use of the FERR#/PBE# pin when the processor is in the stop-clock state (STPCLK# is asserted) to signal the processor that an interrupt is pending and that the processor should return to normal operation to handle the interrupt. Bit 10 (PBE enable) in the IA32_MISC_ENABLE MSR enables this capability.

**INPUT EAX = 02H: TLB/Cache/Prefetch Information Returned in EAX, EBX, ECX, EDX**

When CPUID executes with EAX set to 02H, the processor returns information about the processor's internal TLBs, cache and prefetch hardware in the EAX, EBX, ECX, and EDX registers. The information is reported in encoded form and fall into the following categories:

- The least-significant byte in register EAX (register AL) will always return 01H. Software should ignore this value and not interpret it as an informational descriptor.
- The most significant bit (bit 31) of each register indicates whether the register contains valid information (set to 0) or is reserved (set to 1).
- If a register contains valid information, the information is contained in 1 byte descriptors. There are four types of encoding values for the byte descriptor, the encoding type is noted in the second column of Table 3-12. Table 3-12 lists the encoding of these descriptors. Note that the order of descriptors in the EAX, EBX, ECX, and EDX registers is not defined; that is, specific bytes are not designated to contain descriptors for specific cache, prefetch, or TLB types. The descriptors may appear in any order. Note also a processor may report a general descriptor type (FFH) and not report any byte descriptor of "cache type" via CPUID leaf 2.

Table 3-11. Encoding of CPUID Leaf 2 Descriptors

Value	Type	Description
00H	General	Null descriptor, this byte contains no information
01H	TLB	Instruction TLB: 4 KByte pages, 4-way set associative, 32 entries
02H	TLB	Instruction TLB: 4 MByte pages, fully associative, 2 entries
03H	TLB	Data TLB: 4 KByte pages, 4-way set associative, 64 entries
04H	TLB	Data TLB: 4 MByte pages, 4-way set associative, 8 entries
05H	TLB	Data TLB1: 4 MByte pages, 4-way set associative, 32 entries
06H	Cache	1st-level instruction cache: 8 KBytes, 4-way set associative, 32 byte line size
08H	Cache	1st-level instruction cache: 16 KBytes, 4-way set associative, 32 byte line size
09H	Cache	1st-level instruction cache: 32KBytes, 4-way set associative, 64 byte line size
0AH	Cache	1st-level data cache: 8 KBytes, 2-way set associative, 32 byte line size
0BH	TLB	Instruction TLB: 4 MByte pages, 4-way set associative, 4 entries
0CH	Cache	1st-level data cache: 16 KBytes, 4-way set associative, 32 byte line size
0DH	Cache	1st-level data cache: 16 KBytes, 4-way set associative, 64 byte line size
0EH	Cache	1st-level data cache: 24 KBytes, 6-way set associative, 64 byte line size
1DH	Cache	2nd-level cache: 128 KBytes, 2-way set associative, 64 byte line size
21H	Cache	2nd-level cache: 256 KBytes, 8-way set associative, 64 byte line size
22H	Cache	3rd-level cache: 512 KBytes, 4-way set associative, 64 byte line size, 2 lines per sector
23H	Cache	3rd-level cache: 1 MBytes, 8-way set associative, 64 byte line size, 2 lines per sector
24H	Cache	2nd-level cache: 1 MBytes, 16-way set associative, 64 byte line size
25H	Cache	3rd-level cache: 2 MBytes, 8-way set associative, 64 byte line size, 2 lines per sector
29H	Cache	3rd-level cache: 4 MBytes, 8-way set associative, 64 byte line size, 2 lines per sector
2CH	Cache	1st-level data cache: 32 KBytes, 8-way set associative, 64 byte line size
30H	Cache	1st-level instruction cache: 32 KBytes, 8-way set associative, 64 byte line size
40H	Cache	No 2nd-level cache or, if processor contains a valid 2nd-level cache, no 3rd-level cache
41H	Cache	2nd-level cache: 128 KBytes, 4-way set associative, 32 byte line size
42H	Cache	2nd-level cache: 256 KBytes, 4-way set associative, 32 byte line size
43H	Cache	2nd-level cache: 512 KBytes, 4-way set associative, 32 byte line size
44H	Cache	2nd-level cache: 1 MByte, 4-way set associative, 32 byte line size
45H	Cache	2nd-level cache: 2 MByte, 4-way set associative, 32 byte line size
46H	Cache	3rd-level cache: 4 MByte, 4-way set associative, 64 byte line size
47H	Cache	3rd-level cache: 8 MByte, 8-way set associative, 64 byte line size
48H	Cache	2nd-level cache: 3MByte, 12-way set associative, 64 byte line size
49H	Cache	3rd-level cache: 4MB, 16-way set associative, 64-byte line size (Intel Xeon processor MP, Family 0FH, Model 06H); 2nd-level cache: 4 MByte, 16-way set associative, 64 byte line size
4AH	Cache	3rd-level cache: 6MByte, 12-way set associative, 64 byte line size
4BH	Cache	3rd-level cache: 8MByte, 16-way set associative, 64 byte line size
4CH	Cache	3rd-level cache: 12MByte, 12-way set associative, 64 byte line size
4DH	Cache	3rd-level cache: 16MByte, 16-way set associative, 64 byte line size
4EH	Cache	2nd-level cache: 6MByte, 24-way set associative, 64 byte line size
4FH	TLB	Instruction TLB: 4 KByte pages, 32 entries

Table 3-11. Encoding of CPUID Leaf 2 Descriptors (Contd.)

Value	Type	Description
50H	TLB	Instruction TLB: 4 KByte and 2-MByte or 4-MByte pages, 64 entries
51H	TLB	Instruction TLB: 4 KByte and 2-MByte or 4-MByte pages, 128 entries
52H	TLB	Instruction TLB: 4 KByte and 2-MByte or 4-MByte pages, 256 entries
55H	TLB	Instruction TLB: 2-MByte or 4-MByte pages, fully associative, 7 entries
56H	TLB	Data TLB0: 4 MByte pages, 4-way set associative, 16 entries
57H	TLB	Data TLB0: 4 KByte pages, 4-way associative, 16 entries
59H	TLB	Data TLB0: 4 KByte pages, fully associative, 16 entries
5AH	TLB	Data TLB0: 2 MByte or 4 MByte pages, 4-way set associative, 32 entries
5BH	TLB	Data TLB: 4 KByte and 4 MByte pages, 64 entries
5CH	TLB	Data TLB: 4 KByte and 4 MByte pages, 128 entries
5DH	TLB	Data TLB: 4 KByte and 4 MByte pages, 256 entries
60H	Cache	1st-level data cache: 16 KByte, 8-way set associative, 64 byte line size
61H	TLB	Instruction TLB: 4 KByte pages, fully associative, 48 entries
63H	TLB	Data TLB: 2 MByte or 4 MByte pages, 4-way set associative, 32 entries and a separate array with 1 GByte pages, 4-way set associative, 4 entries
64H	TLB	Data TLB: 4 KByte pages, 4-way set associative, 512 entries
66H	Cache	1st-level data cache: 8 KByte, 4-way set associative, 64 byte line size
67H	Cache	1st-level data cache: 16 KByte, 4-way set associative, 64 byte line size
68H	Cache	1st-level data cache: 32 KByte, 4-way set associative, 64 byte line size
6AH	Cache	uTLB: 4 KByte pages, 8-way set associative, 64 entries
6BH	Cache	DTLB: 4 KByte pages, 8-way set associative, 256 entries
6CH	Cache	DTLB: 2M/4M pages, 8-way set associative, 128 entries
6DH	Cache	DTLB: 1 GByte pages, fully associative, 16 entries
70H	Cache	Trace cache: 12 K- $\mu$ op, 8-way set associative
71H	Cache	Trace cache: 16 K- $\mu$ op, 8-way set associative
72H	Cache	Trace cache: 32 K- $\mu$ op, 8-way set associative
76H	TLB	Instruction TLB: 2M/4M pages, fully associative, 8 entries
78H	Cache	2nd-level cache: 1 MByte, 4-way set associative, 64byte line size
79H	Cache	2nd-level cache: 128 KByte, 8-way set associative, 64 byte line size, 2 lines per sector
7AH	Cache	2nd-level cache: 256 KByte, 8-way set associative, 64 byte line size, 2 lines per sector
7BH	Cache	2nd-level cache: 512 KByte, 8-way set associative, 64 byte line size, 2 lines per sector
7CH	Cache	2nd-level cache: 1 MByte, 8-way set associative, 64 byte line size, 2 lines per sector
7DH	Cache	2nd-level cache: 2 MByte, 8-way set associative, 64byte line size
7FH	Cache	2nd-level cache: 512 KByte, 2-way set associative, 64-byte line size
80H	Cache	2nd-level cache: 512 KByte, 8-way set associative, 64-byte line size
82H	Cache	2nd-level cache: 256 KByte, 8-way set associative, 32 byte line size
83H	Cache	2nd-level cache: 512 KByte, 8-way set associative, 32 byte line size
84H	Cache	2nd-level cache: 1 MByte, 8-way set associative, 32 byte line size
85H	Cache	2nd-level cache: 2 MByte, 8-way set associative, 32 byte line size
86H	Cache	2nd-level cache: 512 KByte, 4-way set associative, 64 byte line size
87H	Cache	2nd-level cache: 1 MByte, 8-way set associative, 64 byte line size

**Table 3-11. Encoding of CPUID Leaf 2 Descriptors (Contd.)**

Value	Type	Description
A0H	DTLB	DTLB: 4k pages, fully associative, 32 entries
B0H	TLB	Instruction TLB: 4 KByte pages, 4-way set associative, 128 entries
B1H	TLB	Instruction TLB: 2M pages, 4-way, 8 entries or 4M pages, 4-way, 4 entries
B2H	TLB	Instruction TLB: 4KByte pages, 4-way set associative, 64 entries
B3H	TLB	Data TLB: 4 KByte pages, 4-way set associative, 128 entries
B4H	TLB	Data TLB1: 4 KByte pages, 4-way associative, 256 entries
B5H	TLB	Instruction TLB: 4KByte pages, 8-way set associative, 64 entries
B6H	TLB	Instruction TLB: 4KByte pages, 8-way set associative, 128 entries
BAH	TLB	Data TLB1: 4 KByte pages, 4-way associative, 64 entries
C0H	TLB	Data TLB: 4 KByte and 4 MByte pages, 4-way associative, 8 entries
C1H	STLB	Shared 2nd-Level TLB: 4 KByte/2MByte pages, 8-way associative, 1024 entries
C2H	DTLB	DTLB: 4 KByte/2 MByte pages, 4-way associative, 16 entries
C3H	STLB	Shared 2nd-Level TLB: 4 KByte /2 MByte pages, 6-way associative, 1536 entries. Also 1GByte pages, 4-way, 16 entries.
C4H	DTLB	DTLB: 2M/4M Byte pages, 4-way associative, 32 entries
CAH	STLB	Shared 2nd-Level TLB: 4 KByte pages, 4-way associative, 512 entries
D0H	Cache	3rd-level cache: 512 KByte, 4-way set associative, 64 byte line size
D1H	Cache	3rd-level cache: 1 MByte, 4-way set associative, 64 byte line size
D2H	Cache	3rd-level cache: 2 MByte, 4-way set associative, 64 byte line size
D6H	Cache	3rd-level cache: 1 MByte, 8-way set associative, 64 byte line size
D7H	Cache	3rd-level cache: 2 MByte, 8-way set associative, 64 byte line size
D8H	Cache	3rd-level cache: 4 MByte, 8-way set associative, 64 byte line size
DCH	Cache	3rd-level cache: 1.5 MByte, 12-way set associative, 64 byte line size
DDH	Cache	3rd-level cache: 3 MByte, 12-way set associative, 64 byte line size
DEH	Cache	3rd-level cache: 6 MByte, 12-way set associative, 64 byte line size
E2H	Cache	3rd-level cache: 2 MByte, 16-way set associative, 64 byte line size
E3H	Cache	3rd-level cache: 4 MByte, 16-way set associative, 64 byte line size
E4H	Cache	3rd-level cache: 8 MByte, 16-way set associative, 64 byte line size
EAH	Cache	3rd-level cache: 12MByte, 24-way set associative, 64 byte line size
EBH	Cache	3rd-level cache: 18MByte, 24-way set associative, 64 byte line size
ECH	Cache	3rd-level cache: 24MByte, 24-way set associative, 64 byte line size
F0H	Prefetch	64-Byte prefetching
F1H	Prefetch	128-Byte prefetching
FFH	General	CPUID leaf 2 does not report cache descriptor information, use CPUID leaf 4 to query cache parameters

**Example 3-1. Example of Cache and TLB Interpretation**

The first member of the family of Pentium 4 processors returns the following information about caches and TLBs when the CPUID executes with an input value of 2:

```
EAX    66 5B 50 01H
EBX    0H
ECX    0H
EDX    00 7A 70 00H
```



Which means:

- The least-significant byte (byte 0) of register EAX is set to 01H. This value should be ignored.
- The most-significant bit of all four registers (EAX, EBX, ECX, and EDX) is set to 0, indicating that each register contains valid 1-byte descriptors.
- Bytes 1, 2, and 3 of register EAX indicate that the processor has:
  - 50H - a 64-entry instruction TLB, for mapping 4-KByte and 2-MByte or 4-MByte pages.
  - 5BH - a 64-entry data TLB, for mapping 4-KByte and 4-MByte pages.
  - 66H - an 8-KByte 1st level data cache, 4-way set associative, with a 64-Byte cache line size.
- The descriptors in registers EBX and ECX are valid, but contain NULL descriptors.
- Bytes 0, 1, 2, and 3 of register EDX indicate that the processor has:
  - 00H - NULL descriptor.
  - 70H - Trace cache: 12 K- $\mu$ op, 8-way set associative.
  - 7AH - a 256-KByte 2nd level cache, 8-way set associative, with a sectored, 64-byte cache line size.
  - 00H - NULL descriptor.

### INPUT EAX = 04H: Returns Deterministic Cache Parameters for Each Level

When CPUID executes with EAX set to 04H and ECX contains an index value, the processor returns encoded data that describe a set of deterministic cache parameters (for the cache level associated with the input in ECX). Valid index values start from 0.

Software can enumerate the deterministic cache parameters for each level of the cache hierarchy starting with an index value of 0, until the parameters report the value associated with the cache type field is 0. The architecturally defined fields reported by deterministic cache parameters are documented in Table 3-8.

This Cache Size in Bytes

$$= (\text{Ways} + 1) * (\text{Partitions} + 1) * (\text{Line\_Size} + 1) * (\text{Sets} + 1)$$

$$= (\text{EBX}[31:22] + 1) * (\text{EBX}[21:12] + 1) * (\text{EBX}[11:0] + 1) * (\text{ECX} + 1)$$

The CPUID leaf 04H also reports data that can be used to derive the topology of processor cores in a physical package. This information is constant for all valid index values. Software can query the raw data reported by executing CPUID with EAX=04H and ECX=0 and use it as part of the topology enumeration algorithm described in Chapter 8, "Multiple-Processor Management," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

### INPUT EAX = 05H: Returns MONITOR and MWAIT Features

When CPUID executes with EAX set to 05H, the processor returns information about features available to MONITOR/MWAIT instructions. The MONITOR instruction is used for address-range monitoring in conjunction with MWAIT instruction. The MWAIT instruction optionally provides additional extensions for advanced power management. See Table 3-8.

### INPUT EAX = 06H: Returns Thermal and Power Management Features

When CPUID executes with EAX set to 06H, the processor returns information about thermal and power management features. See Table 3-8.

### INPUT EAX = 07H: Returns Structured Extended Feature Enumeration Information

When CPUID executes with EAX set to 07H and ECX = 0, the processor returns information about the maximum input value for sub-leaves that contain extended feature flags. See Table 3-8.

**When CPUID executes with EAX set to 07H and the input value of ECX is invalid (see leaf 07H entry in Table 3-8), the processor returns 0 in EAX/EBX/ECX/EDX. In subleaf 0, EAX returns the maximum input value of the highest**



leaf 7 sub-leaf, and EBX, ECX & EDX contain information of extended feature flags.

#### INPUT EAX = 09H: Returns Direct Cache Access Information

When CPUID executes with EAX set to 09H, the processor returns information about Direct Cache Access capabilities. See Table 3-8.

#### INPUT EAX = 0AH: Returns Architectural Performance Monitoring Features

When CPUID executes with EAX set to 0AH, the processor returns information about support for architectural performance monitoring capabilities. Architectural performance monitoring is supported if the version ID (see Table 3-8) is greater than Pn 0. See Table 3-8.

For each version of architectural performance monitoring capability, software must enumerate this leaf to discover the programming facilities and the architectural performance events available in the processor. The details are described in Chapter 23, "Introduction to Virtual-Machine Extensions," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C*.

#### INPUT EAX = 0BH: Returns Extended Topology Information

When CPUID executes with EAX set to 0BH, the processor returns information about extended topology enumeration data. Software must detect the presence of CPUID leaf 0BH by verifying (a) the highest leaf index supported by CPUID is  $\geq 0BH$ , and (b) CPUID.0BH:EBX[15:0] reports a non-zero value. See Table 3-8.

#### INPUT EAX = 0DH: Returns Processor Extended States Enumeration Information

When CPUID executes with EAX set to 0DH and ECX = 0, the processor returns information about the bit-vector representation of all processor state extensions that are supported in the processor and storage size requirements of the XSAVE/XRSTOR area. See Table 3-8.

When CPUID executes with EAX set to 0DH and ECX = n (n > 1, and is a valid sub-leaf index), the processor returns information about the size and offset of each processor extended state save area within the XSAVE/XRSTOR area. See Table 3-8. Software can use the forward-extendable technique depicted below to query the valid sub-leaves and obtain size and offset information for each processor extended state save area:

```
For i = 2 to 62 // sub-leaf 1 is reserved
  IF (CPUID.(EAX=0DH, ECX=0):VECTOR[i] = 1) // VECTOR is the 64-bit value of EDX:EAX
    Execute CPUID.(EAX=0DH, ECX = i) to examine size and offset for sub-leaf i;
  FI;
```

#### INPUT EAX = 0FH: Returns Intel Resource Director Technology (Intel RDT) Monitoring Enumeration Information

When CPUID executes with EAX set to 0FH and ECX = 0, the processor returns information about the bit-vector representation of QoS monitoring resource types that are supported in the processor and maximum range of RMID values the processor can use to monitor of any supported resource types. Each bit, starting from bit 1, corresponds to a specific resource type if the bit is set. The bit position corresponds to the sub-leaf index (or ResID) that software must use to query QoS monitoring capability available for that type. See Table 3-8.

When CPUID executes with EAX set to 0FH and ECX = n (n  $\geq$  1, and is a valid ResID), the processor returns information software can use to program IA32\_PQR\_ASSOC, IA32\_QM\_EVTSEL MSRs before reading QoS data from the IA32\_QM\_CTR MSR.

#### INPUT EAX = 10H: Returns Intel Resource Director Technology (Intel RDT) Allocation Enumeration Information

When CPUID executes with EAX set to 10H and ECX = 0, the processor returns information about the bit-vector representation of QoS Enforcement resource types that are supported in the processor. Each bit, starting from bit 1, corresponds to a specific resource type if the bit is set. The bit position corresponds to the sub-leaf index (or ResID) that software must use to query QoS enforcement capability available for that type. See Table 3-8.

When CPUID executes with EAX set to 10H and ECX = n (n  $\geq$  1, and is a valid ResID), the processor returns information about available classes of service and range of QoS mask MSRs that software can use to configure each class of services using capability bit masks in the QoS Mask registers, IA32\_resourceType\_Mask\_n.

**INPUT EAX = 12H: Returns Intel SGX Enumeration Information**

When CPUID executes with EAX set to 12H and ECX = 0H, the processor returns information about Intel SGX capabilities. See Table 3-8.

When CPUID executes with EAX set to 12H and ECX = 1H, the processor returns information about Intel SGX attributes. See Table 3-8.

When CPUID executes with EAX set to 12H and ECX = n (n > 1), the processor returns information about Intel SGX Enclave Page Cache. See Table 3-8.

**INPUT EAX = 14H: Returns Intel Processor Trace Enumeration Information**

When CPUID executes with EAX set to 14H and ECX = 0H, the processor returns information about Intel Processor Trace extensions. See Table 3-8.

When CPUID executes with EAX set to 14H and ECX = n (n > 0 and less than the number of non-zero bits in CPUID.(EAX=14H, ECX= 0H).EAX), the processor returns information about packet generation in Intel Processor Trace. See Table 3-8.

**INPUT EAX = 15H: Returns Time Stamp Counter/Core Crystal Clock Information**

When CPUID executes with EAX set to 15H and ECX = 0H, the processor returns information about Time Stamp Counter/Core Crystal Clock. See Table 3-8.

**INPUT EAX = 16H: Returns Processor Frequency Information**

When CPUID executes with EAX set to 16H, the processor returns information about Processor Frequency Information. See Table 3-8.

**INPUT EAX = 17H: Returns System-On-Chip Information**

When CPUID executes with EAX set to 17H, the processor returns information about the System-On-Chip Vendor Attribute Enumeration. See Table 3-8.

**METHODS FOR RETURNING BRANDING INFORMATION**

Use the following techniques to access branding information:

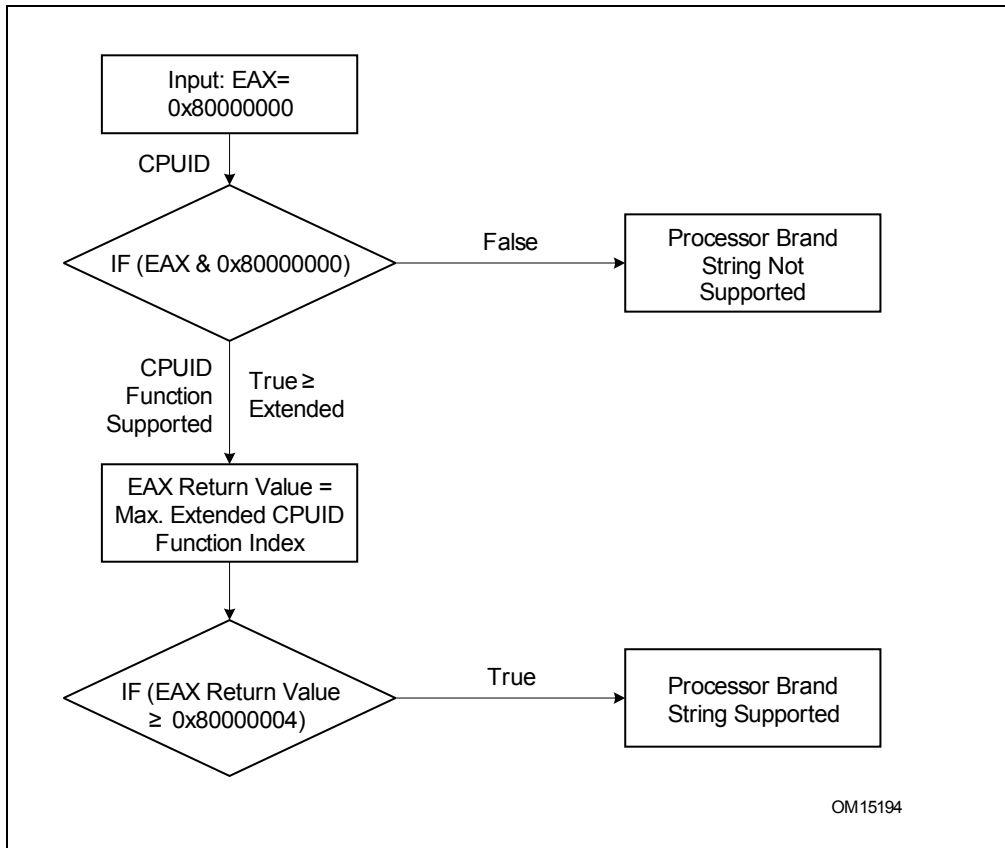
1. Processor brand string method.
2. Processor brand index; this method uses a software supplied brand string table.

These two methods are discussed in the following sections. For methods that are available in early processors, see Section: "Identification of Earlier IA-32 Processors" in Chapter 19 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*.

**The Processor Brand String Method**

Figure 3-9 describes the algorithm used for detection of the brand string. Processor brand identification software should execute this algorithm on all Intel 64 and IA-32 processors.

This method (introduced with Pentium 4 processors) returns an ASCII brand identification string and the Processor Base frequency of the processor to the EAX, EBX, ECX, and EDX registers.



**Figure 3-6. Determination of Support for the Processor Brand String**

### How Brand Strings Work

To use the brand string method, execute CPUID with EAX input of 8000002H through 8000004H. For each input value, CPUID returns 16 ASCII characters using EAX, EBX, ECX, and EDX. The returned string will be NULL-terminated.

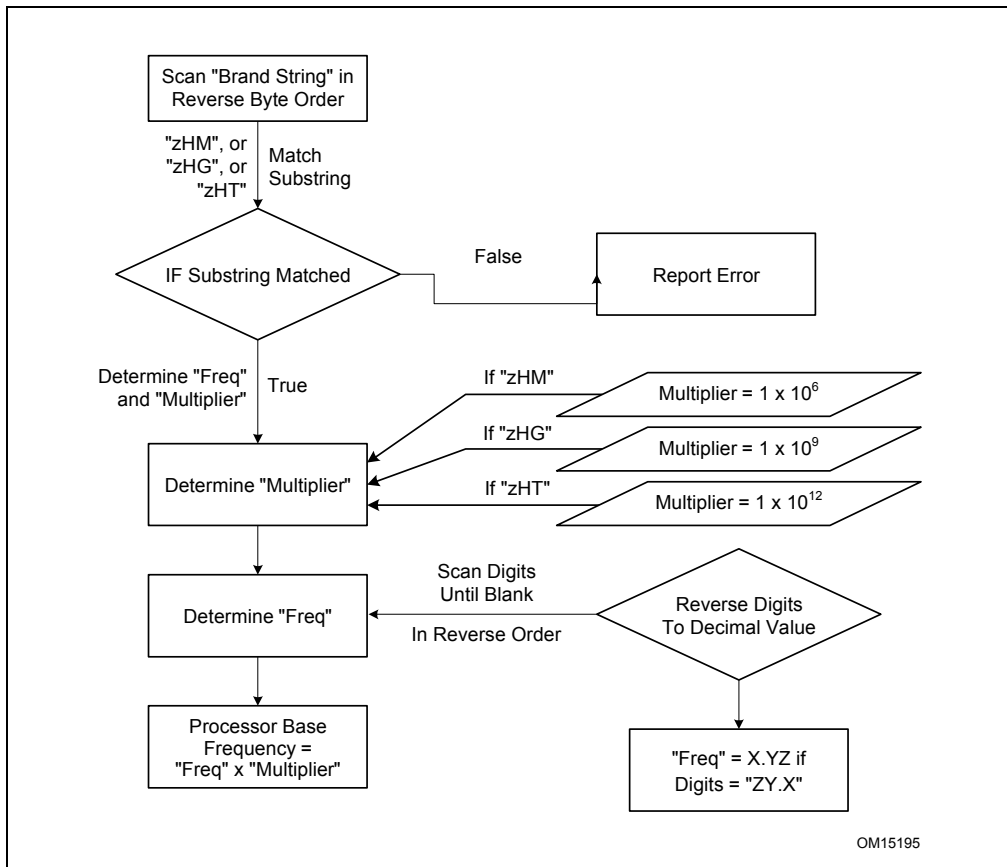
Table 3-13 shows the brand string that is returned by the first processor in the Pentium 4 processor family.

**Table 3-12. Processor Brand String Returned with Pentium 4 Processor**

EAX Input Value	Return Values	ASCII Equivalent
80000002H	EAX = 20202020H EBX = 20202020H ECX = 20202020H EDX = 6E492020H	" " " " " " " " " "nl "
80000003H	EAX = 286C6574H EBX = 50202952H ECX = 69746E65H EDX = 52286D75H	"(let" "P )R" "itne" "R(mu"
80000004H	EAX = 20342029H EBX = 20555043H ECX = 30303531H EDX = 007A484DH	" 4 )" " UPC" "0051" "\0zHM"

**Extracting the Processor Frequency from Brand Strings**

Figure 3-10 provides an algorithm which software can use to extract the Processor Base frequency from the processor brand string.



**Figure 3-7. Algorithm for Extracting Processor Frequency**

## The Processor Brand Index Method

The brand index method (introduced with Pentium® III Xeon® processors) provides an entry point into a brand identification table that is maintained in memory by system software and is accessible from system- and user-level code. In this table, each brand index is associate with an ASCII brand identification string that identifies the official Intel family and model number of a processor.

When CPUID executes with EAX set to 1, the processor returns a brand index to the low byte in EBX. Software can then use this index to locate the brand identification string for the processor in the brand identification table. The first entry (brand index 0) in this table is reserved, allowing for backward compatibility with processors that do not support the brand identification feature. Starting with processor signature family ID = 0FH, model = 03H, brand index method is no longer supported. Use brand string method instead.

Table 3-14 shows brand indices that have identification strings associated with them.

**Table 3-13. Mapping of Brand Indices; and Intel 64 and IA-32 Processor Brand Strings**

Brand Index	Brand String
00H	This processor does not support the brand identification feature
01H	Intel(R) Celeron(R) processor <sup>1</sup>
02H	Intel(R) Pentium(R) III processor <sup>1</sup>
03H	Intel(R) Pentium(R) III Xeon(R) processor; If processor signature = 000006B1h, then Intel(R) Celeron(R) processor
04H	Intel(R) Pentium(R) III processor
06H	Mobile Intel(R) Pentium(R) III processor-M
07H	Mobile Intel(R) Celeron(R) processor <sup>1</sup>
08H	Intel(R) Pentium(R) 4 processor
09H	Intel(R) Pentium(R) 4 processor
0AH	Intel(R) Celeron(R) processor <sup>1</sup>
0BH	Intel(R) Xeon(R) processor; If processor signature = 00000F13h, then Intel(R) Xeon(R) processor MP
0CH	Intel(R) Xeon(R) processor MP
0EH	Mobile Intel(R) Pentium(R) 4 processor-M; If processor signature = 00000F13h, then Intel(R) Xeon(R) processor
0FH	Mobile Intel(R) Celeron(R) processor <sup>1</sup>
11H	Mobile Genuine Intel(R) processor
12H	Intel(R) Celeron(R) M processor
13H	Mobile Intel(R) Celeron(R) processor <sup>1</sup>
14H	Intel(R) Celeron(R) processor
15H	Mobile Genuine Intel(R) processor
16H	Intel(R) Pentium(R) M processor
17H	Mobile Intel(R) Celeron(R) processor <sup>1</sup>
18H - 0FFH	RESERVED

### NOTES:

1. Indicates versions of these processors that were introduced after the Pentium III

## IA-32 Architecture Compatibility

CPUID is not supported in early models of the Intel486 processor or in any IA-32 processor earlier than the Intel486 processor.

**Operation**

IA32\_BIOS\_SIGN\_ID MSR ← Update with installed microcode revision number;

CASE (EAX) OF

EAX = 0:

EAX ← Highest basic function input value understood by CPUID;

EBX ← Vendor identification string;

EDX ← Vendor identification string;

ECX ← Vendor identification string;

BREAK;

EAX = 1H:

EAX[3:0] ← Stepping ID;

EAX[7:4] ← Model;

EAX[11:8] ← Family;

EAX[13:12] ← Processor type;

EAX[15:14] ← Reserved;

EAX[19:16] ← Extended Model;

EAX[27:20] ← Extended Family;

EAX[31:28] ← Reserved;

EBX[7:0] ← Brand Index; (\* Reserved if the value is zero. \*)

EBX[15:8] ← CLFLUSH Line Size;

EBX[16:23] ← Reserved; (\* Number of threads enabled = 2 if MT enable fuse set. \*)

EBX[24:31] ← Initial APIC ID;

ECX ← Feature flags; (\* See Figure 3-7. \*)

EDX ← Feature flags; (\* See Figure 3-8. \*)

BREAK;

EAX = 2H:

EAX ← Cache and TLB information;

EBX ← Cache and TLB information;

ECX ← Cache and TLB information;

EDX ← Cache and TLB information;

BREAK;

EAX = 3H:

EAX ← Reserved;

EBX ← Reserved;

ECX ← ProcessorSerialNumber[31:0];

(\* Pentium III processors only, otherwise reserved. \*)

EDX ← ProcessorSerialNumber[63:32];

(\* Pentium III processors only, otherwise reserved. \*)

BREAK

EAX = 4H:

EAX ← Deterministic Cache Parameters Leaf; (\* See Table 3-8. \*)

EBX ← Deterministic Cache Parameters Leaf;

ECX ← Deterministic Cache Parameters Leaf;

EDX ← Deterministic Cache Parameters Leaf;

BREAK;

EAX = 5H:

EAX ← MONITOR/MWAIT Leaf; (\* See Table 3-8. \*)

EBX ← MONITOR/MWAIT Leaf;

ECX ← MONITOR/MWAIT Leaf;

EDX ← MONITOR/MWAIT Leaf;

BREAK;

EAX = 6H:  
 EAX ← Thermal and Power Management Leaf; (\* See Table 3-8. \*)  
 EBX ← Thermal and Power Management Leaf;  
 ECX ← Thermal and Power Management Leaf;  
 EDX ← Thermal and Power Management Leaf;

BREAK;

EAX = 7H:  
 EAX ← Structured Extended Feature Flags Enumeration Leaf; (\* See Table 3-8. \*)  
 EBX ← Structured Extended Feature Flags Enumeration Leaf;  
 ECX ← Structured Extended Feature Flags Enumeration Leaf;  
 EDX ← Structured Extended Feature Flags Enumeration Leaf;

BREAK;

EAX = 8H:  
 EAX ← Reserved = 0;  
 EBX ← Reserved = 0;  
 ECX ← Reserved = 0;  
 EDX ← Reserved = 0;

BREAK;

EAX = 9H:  
 EAX ← Direct Cache Access Information Leaf; (\* See Table 3-8. \*)  
 EBX ← Direct Cache Access Information Leaf;  
 ECX ← Direct Cache Access Information Leaf;  
 EDX ← Direct Cache Access Information Leaf;

BREAK;

EAX = AH:  
 EAX ← Architectural Performance Monitoring Leaf; (\* See Table 3-8. \*)  
 EBX ← Architectural Performance Monitoring Leaf;  
 ECX ← Architectural Performance Monitoring Leaf;  
 EDX ← Architectural Performance Monitoring Leaf;  
 BREAK

EAX = BH:  
 EAX ← Extended Topology Enumeration Leaf; (\* See Table 3-8. \*)  
 EBX ← Extended Topology Enumeration Leaf;  
 ECX ← Extended Topology Enumeration Leaf;  
 EDX ← Extended Topology Enumeration Leaf;

BREAK;

EAX = CH:  
 EAX ← Reserved = 0;  
 EBX ← Reserved = 0;  
 ECX ← Reserved = 0;  
 EDX ← Reserved = 0;

BREAK;

EAX = DH:  
 EAX ← Processor Extended State Enumeration Leaf; (\* See Table 3-8. \*)  
 EBX ← Processor Extended State Enumeration Leaf;  
 ECX ← Processor Extended State Enumeration Leaf;  
 EDX ← Processor Extended State Enumeration Leaf;

BREAK;

EAX = EH:  
 EAX ← Reserved = 0;  
 EBX ← Reserved = 0;  
 ECX ← Reserved = 0;  
 EDX ← Reserved = 0;

BREAK;

EAX = FH:

EAX ← Intel Resource Director Technology Monitoring Enumeration Leaf; (\* See Table 3-8. \*)  
 EBX ← Intel Resource Director Technology Monitoring Enumeration Leaf;  
 ECX ← Intel Resource Director Technology Monitoring Enumeration Leaf;  
 EDX ← Intel Resource Director Technology Monitoring Enumeration Leaf;

BREAK;

EAX = 10H:

EAX ← Intel Resource Director Technology Allocation Enumeration Leaf; (\* See Table 3-8. \*)  
 EBX ← Intel Resource Director Technology Allocation Enumeration Leaf;  
 ECX ← Intel Resource Director Technology Allocation Enumeration Leaf;  
 EDX ← Intel Resource Director Technology Allocation Enumeration Leaf;

BREAK;

EAX = 12H:

EAX ← Intel SGX Enumeration Leaf; (\* See Table 3-8. \*)  
 EBX ← Intel SGX Enumeration Leaf;  
 ECX ← Intel SGX Enumeration Leaf;  
 EDX ← Intel SGX Enumeration Leaf;

BREAK;

EAX = 14H:

EAX ← Intel Processor Trace Enumeration Leaf; (\* See Table 3-8. \*)  
 EBX ← Intel Processor Trace Enumeration Leaf;  
 ECX ← Intel Processor Trace Enumeration Leaf;  
 EDX ← Intel Processor Trace Enumeration Leaf;

BREAK;

EAX = 15H:

EAX ← Time Stamp Counter/Core Crystal Clock Information Leaf; (\* See Table 3-8. \*)  
 EBX ← Time Stamp Counter/Core Crystal Clock Information Leaf;  
 ECX ← Time Stamp Counter/Core Crystal Clock Information Leaf;  
 EDX ← Time Stamp Counter/Core Crystal Clock Information Leaf;

BREAK;

EAX = 16H:

EAX ← Processor Frequency Information Enumeration Leaf; (\* See Table 3-8. \*)  
 EBX ← Processor Frequency Information Enumeration Leaf;  
 ECX ← Processor Frequency Information Enumeration Leaf;  
 EDX ← Processor Frequency Information Enumeration Leaf;

BREAK;

EAX = 17H:

EAX ← System-On-Chip Vendor Attribute Enumeration Leaf; (\* See Table 3-8. \*)  
 EBX ← System-On-Chip Vendor Attribute Enumeration Leaf;  
 ECX ← System-On-Chip Vendor Attribute Enumeration Leaf;  
 EDX ← System-On-Chip Vendor Attribute Enumeration Leaf;

BREAK;

EAX = 80000000H:

EAX ← Highest extended function input value understood by CPUID;  
 EBX ← Reserved;  
 ECX ← Reserved;  
 EDX ← Reserved;

BREAK;

EAX = 80000001H:

EAX ← Reserved;  
 EBX ← Reserved;  
 ECX ← Extended Feature Bits (\* See Table 3-8.\*);  
 EDX ← Extended Feature Bits (\* See Table 3-8.\*);

BREAK;



EAX = 80000002H:

EAX ← Processor Brand String;  
 EBX ← Processor Brand String, continued;  
 ECX ← Processor Brand String, continued;  
 EDX ← Processor Brand String, continued;

BREAK;

EAX = 80000003H:

EAX ← Processor Brand String, continued;  
 EBX ← Processor Brand String, continued;  
 ECX ← Processor Brand String, continued;  
 EDX ← Processor Brand String, continued;

BREAK;

EAX = 80000004H:

EAX ← Processor Brand String, continued;  
 EBX ← Processor Brand String, continued;  
 ECX ← Processor Brand String, continued;  
 EDX ← Processor Brand String, continued;

BREAK;

EAX = 80000005H:

EAX ← Reserved = 0;  
 EBX ← Reserved = 0;  
 ECX ← Reserved = 0;  
 EDX ← Reserved = 0;

BREAK;

EAX = 80000006H:

EAX ← Reserved = 0;  
 EBX ← Reserved = 0;  
 ECX ← Cache information;  
 EDX ← Reserved = 0;

BREAK;

EAX = 80000007H:

EAX ← Reserved = 0;  
 EBX ← Reserved = 0;  
 ECX ← Reserved = 0;  
 EDX ← Reserved = Misc Feature Flags;

BREAK;

EAX = 80000008H:

EAX ← Reserved = Physical Address Size Information;  
 EBX ← Reserved = Virtual Address Size Information;  
 ECX ← Reserved = 0;  
 EDX ← Reserved = 0;

BREAK;

EAX >= 40000000H and EAX <= 4FFFFFFFH:

DEFAULT: (\* EAX = Value outside of recognized range for CPUID. \*)

(\* If the highest basic information leaf data depend on ECX input value, ECX is honored. \*)

EAX ← Reserved; (\* Information returned for highest basic information leaf. \*)  
 EBX ← Reserved; (\* Information returned for highest basic information leaf. \*)  
 ECX ← Reserved; (\* Information returned for highest basic information leaf. \*)  
 EDX ← Reserved; (\* Information returned for highest basic information leaf. \*)

BREAK;

ESAC;

## Flags Affected

None.

### Exceptions (All Operating Modes)

#UD

If the LOCK prefix is used.

In earlier IA-32 processors that do not support the CPUID instruction, execution of the instruction results in an invalid opcode (#UD) exception being generated.

## CVTDDQ2PD—Convert Packed Doubleword Integers to Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F E6 /r CVTDDQ2PD xmm1, xmm2/m64	RM	V/V	SSE2	Convert two packed signed doubleword integers from xmm2/mem to two packed double-precision floating-point values in xmm1.
VEX.128.F3.0F.WIG E6 /r VCVTDQ2PD xmm1, xmm2/m64	RM	V/V	AVX	Convert two packed signed doubleword integers from xmm2/mem to two packed double-precision floating-point values in xmm1.
VEX.256.F3.0F.WIG E6 /r VCVTDQ2PD ymm1, xmm2/m128	RM	V/V	AVX	Convert four packed signed doubleword integers from xmm2/mem to four packed double-precision floating-point values in ymm1.
EVEX.128.F3.0F.W0 E6 /r VCVTDQ2PD xmm1 {k1}{z}, xmm2/m128/m32bcst	HV	V/V	AVX512VL AVX512F	Convert 2 packed signed doubleword integers from xmm2/m128/m32bcst to eight packed double-precision floating-point values in xmm1 with writemask k1.
EVEX.256.F3.0F.W0 E6 /r VCVTDQ2PD ymm1 {k1}{z}, xmm2/m128/m32bcst	HV	V/V	AVX512VL AVX512F	Convert 4 packed signed doubleword integers from xmm2/m128/m32bcst to 4 packed double-precision floating-point values in ymm1 with writemask k1.
EVEX.512.F3.0F.W0 E6 /r VCVTDQ2PD zmm1 {k1}{z}, ymm2/m256/m32bcst	HV	V/V	AVX512F	Convert eight packed signed doubleword integers from ymm2/m256/m32bcst to eight packed double-precision floating-point values in zmm1 with writemask k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
HV	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Converts two, four or eight packed signed doubleword integers in the source operand (the second operand) to two, four or eight packed double-precision floating-point values in the destination operand (the first operand).

EVEX encoded versions: The source operand can be a YMM/XMM/XMM (low 64 bits) register, a 256/128/64-bit memory location or a 256/128/64-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1. Attempt to encode this instruction with EVEX embedded rounding is ignored.

VEX.256 encoded version: The source operand is an XMM register or 128-bit memory location. The destination operand is a YMM register.

VEX.128 encoded version: The source operand is an XMM register or 64-bit memory location. The destination operand is a XMM register. The upper Bits (MAX\_VL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The source operand is an XMM register or 64-bit memory location. The destination operand is an XMM register. The upper Bits (MAX\_VL-1:128) of the corresponding ZMM register destination are unmodified.

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

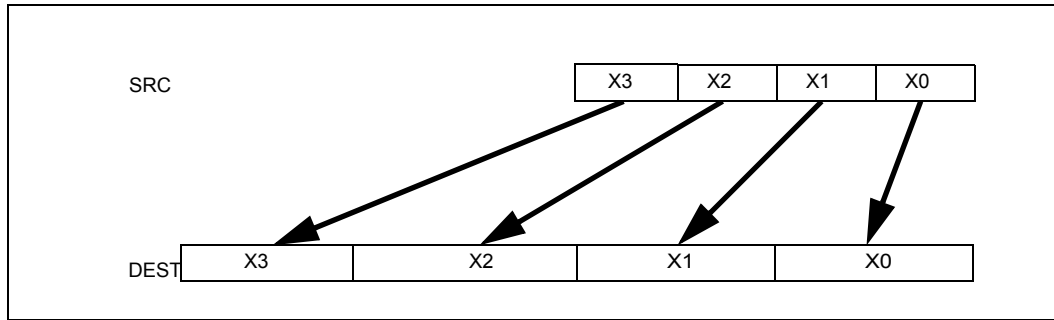


Figure 3-8. CVTQ2PD (VEX.256 encoded version)

### Operation

**VCVTQ2PD (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\_Integer\_To\_Double\_Precision\_Floating\_Point(SRC[k+31:k])

  ELSE

    IF \*merging-masking\* ; merging-masking

      THEN \*DEST[i+63:i] remains unchanged\*

    ELSE ; zeroing-masking

      DEST[i+63:i] ← 0

  FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VCVTDQ2PD (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_Integer_To_Double_Precision_Floating_Point(SRC[31:0])
        ELSE
          DEST[i+63:i] ←
          Convert_Integer_To_Double_Precision_Floating_Point(SRC[k+31:k])
      FI;
    ELSE
      IF *merging-masking*           ; merging-masking
        THEN *DEST[i+63:i] remains unchanged*
      ELSE                             ; zeroing-masking
        DEST[i+63:i] ← 0
      FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VCVTDQ2PD (VEX.256 encoded version)**

```

DEST[63:0] ← Convert_Integer_To_Double_Precision_Floating_Point(SRC[31:0])
DEST[127:64] ← Convert_Integer_To_Double_Precision_Floating_Point(SRC[63:32])
DEST[191:128] ← Convert_Integer_To_Double_Precision_Floating_Point(SRC[95:64])
DEST[255:192] ← Convert_Integer_To_Double_Precision_Floating_Point(SRC[127:96])
DEST[MAX_VL-1:256] ← 0

```

**VCVTDQ2PD (VEX.128 encoded version)**

```

DEST[63:0] ← Convert_Integer_To_Double_Precision_Floating_Point(SRC[31:0])
DEST[127:64] ← Convert_Integer_To_Double_Precision_Floating_Point(SRC[63:32])
DEST[MAX_VL-1:128] ← 0

```

**CVTDQ2PD (128-bit Legacy SSE version)**

```

DEST[63:0] ← Convert_Integer_To_Double_Precision_Floating_Point(SRC[31:0])
DEST[127:64] ← Convert_Integer_To_Double_Precision_Floating_Point(SRC[63:32])
DEST[MAX_VL-1:128] (unmodified)

```

### Intel C/C++ Compiler Intrinsic Equivalent

```

VCVTDQ2PD __m512d __mm512_cvtepi32_pd( __m256i a);
VCVTDQ2PD __m512d __mm512_mask_cvtepi32_pd( __m512d s, __mmask8 k, __m256i a);
VCVTDQ2PD __m512d __mm512_maskz_cvtepi32_pd( __mmask8 k, __m256i a);
VCVTDQ2PD __m256d __mm256_mask_cvtepi32_pd( __m256d s, __mmask8 k, __m256i a);
VCVTDQ2PD __m256d __mm256_maskz_cvtepi32_pd( __mmask8 k, __m256i a);
VCVTDQ2PD __m128d __mm_mask_cvtepi32_pd( __m128d s, __mmask8 k, __m128i a);
VCVTDQ2PD __m128d __mm_maskz_cvtepi32_pd( __mmask8 k, __m128i a);
CVTDQ2PD __m256d __mm256_cvtepi32_pd( __m128i src)
CVTDQ2PD __m128d __mm_cvtepi32_pd( __m128i src)

```

### Other Exceptions

VEX-encoded instructions, see Exceptions Type 5;

EVEX-encoded instructions, see Exceptions Type E5.

#UD                      If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

## CVTDQ2PS—Convert Packed Doubleword Integers to Packed Single-Precision Floating-Point Values

Opcode Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
OF 5B /r CVTDQ2PS xmm1, xmm2/m128	RM	V/V	SSE2	Convert four packed signed doubleword integers from xmm2/mem to four packed single-precision floating-point values in xmm1.
VEX.128.OF.WIG 5B /r VCVTDQ2PS xmm1, xmm2/m128	RM	V/V	AVX	Convert four packed signed doubleword integers from xmm2/mem to four packed single-precision floating-point values in xmm1.
VEX.256.OF.WIG 5B /r VCVTDQ2PS ymm1, ymm2/m256	RM	V/V	AVX	Convert eight packed signed doubleword integers from ymm2/mem to eight packed single-precision floating-point values in ymm1.
EVEX.128.OF.W0 5B /r VCVTDQ2PS xmm1 {k1}{z}, xmm2/m128/m32bcst	FV	V/V	AVX512VL AVX512F	Convert four packed signed doubleword integers from xmm2/m128/m32bcst to four packed single-precision floating-point values in xmm1 with writemask k1.
EVEX.256.OF.W0 5B /r VCVTDQ2PS ymm1 {k1}{z}, ymm2/m256/m32bcst	FV	V/V	AVX512VL AVX512F	Convert eight packed signed doubleword integers from ymm2/m256/m32bcst to eight packed single-precision floating-point values in ymm1 with writemask k1.
EVEX.512.OF.W0 5B /r VCVTDQ2PS zmm1 {k1}{z}, zmm2/m512/m32bcst{er}	FV	V/V	AVX512F	Convert sixteen packed signed doubleword integers from zmm2/m512/m32bcst to sixteen packed single-precision floating-point values in zmm1 with writemask k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
FV	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Converts four, eight or sixteen packed signed doubleword integers in the source operand to four, eight or sixteen packed single-precision floating-point values in the destination operand.

EVEX encoded versions: 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 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

VEX.256 encoded version: The source operand is a YMM register or 256-bit memory location. The destination operand is a YMM register. Bits (MAX\_VL-1:256) of the corresponding register destination are zeroed.

VEX.128 encoded version: The source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAX\_VL-1:128) of the corresponding register destination are zeroed.

128-bit Legacy SSE version: The source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper Bits (MAX\_VL-1:128) of the corresponding register destination are unmodified.

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

**Operation****VCVTDQ2PS (EVEX encoded versions) when SRC operand is a register**

(KL, VL) = (4, 128), (8, 256), (16, 512)

IF (VL = 512) AND (EVEX.b = 1)

THEN

SET\_RM(EVEX.RC); ; refer to Table 2-4

ELSE

SET\_RM(MXCSR.RM); ; refer to Table 2-4

FI;

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\*

THEN DEST[i+31:i] ←

Convert\_Integer\_To\_Single\_Precision\_Floating\_Point(SRC[i+31:i])

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+31:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VCVTDQ2PS (EVEX encoded versions) when SRC operand is a memory source**

(KL, VL) = (4, 128), (8, 256), (16, 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\_Integer\_To\_Single\_Precision\_Floating\_Point(SRC[31:0])

ELSE

DEST[i+31:i] ←

Convert\_Integer\_To\_Single\_Precision\_Floating\_Point(SRC[i+31:i])

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+31:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0



**VCVTDQ2PS (VEX.256 encoded version)**

DEST[31:0] ← Convert\_Integer\_To\_Single\_Precision\_Floating\_Point(SRC[31:0])  
 DEST[63:32] ← Convert\_Integer\_To\_Single\_Precision\_Floating\_Point(SRC[63:32])  
 DEST[95:64] ← Convert\_Integer\_To\_Single\_Precision\_Floating\_Point(SRC[95:64])  
 DEST[127:96] ← Convert\_Integer\_To\_Single\_Precision\_Floating\_Point(SRC[127:96])  
 DEST[159:128] ← Convert\_Integer\_To\_Single\_Precision\_Floating\_Point(SRC[159:128])  
 DEST[191:160] ← Convert\_Integer\_To\_Single\_Precision\_Floating\_Point(SRC[191:160])  
 DEST[223:192] ← Convert\_Integer\_To\_Single\_Precision\_Floating\_Point(SRC[223:192])  
 DEST[255:224] ← Convert\_Integer\_To\_Single\_Precision\_Floating\_Point(SRC[255:224])  
 DEST[MAX\_VL-1:256] ← 0

**VCVTDQ2PS (VEX.128 encoded version)**

DEST[31:0] ← Convert\_Integer\_To\_Single\_Precision\_Floating\_Point(SRC[31:0])  
 DEST[63:32] ← Convert\_Integer\_To\_Single\_Precision\_Floating\_Point(SRC[63:32])  
 DEST[95:64] ← Convert\_Integer\_To\_Single\_Precision\_Floating\_Point(SRC[95:64])  
 DEST[127:96] ← Convert\_Integer\_To\_Single\_Precision\_Floating\_Point(SRC[127:96])  
 DEST[MAX\_VL-1:128] ← 0

**CVTDQ2PS (128-bit Legacy SSE version)**

DEST[31:0] ← Convert\_Integer\_To\_Single\_Precision\_Floating\_Point(SRC[31:0])  
 DEST[63:32] ← Convert\_Integer\_To\_Single\_Precision\_Floating\_Point(SRC[63:32])  
 DEST[95:64] ← Convert\_Integer\_To\_Single\_Precision\_Floating\_Point(SRC[95:64])  
 DEST[127:96] ← Convert\_Integer\_To\_Single\_Precision\_Floating\_Point(SRC[127:96])  
 DEST[MAX\_VL-1:128] (unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**

VCVTDQ2PS \_\_m512 \_\_mm512\_cvtepi32\_ps( \_\_m512i a);  
 VCVTDQ2PS \_\_m512 \_\_mm512\_mask\_cvtepi32\_ps( \_\_m512 s, \_\_mmask16 k, \_\_m512i a);  
 VCVTDQ2PS \_\_m512 \_\_mm512\_maskz\_cvtepi32\_ps( \_\_mmask16 k, \_\_m512i a);  
 VCVTDQ2PS \_\_m512 \_\_mm512\_cvt\_roundepsi32\_ps( \_\_m512i a, int r);  
 VCVTDQ2PS \_\_m512 \_\_mm512\_mask\_cvt\_roundepsi32\_ps( \_\_m512 s, \_\_mmask16 k, \_\_m512i a, int r);  
 VCVTDQ2PS \_\_m512 \_\_mm512\_maskz\_cvt\_roundepsi32\_ps( \_\_mmask16 k, \_\_m512i a, int r);  
 VCVTDQ2PS \_\_m256 \_\_mm256\_mask\_cvtepi32\_ps( \_\_m256 s, \_\_mmask8 k, \_\_m256i a);  
 VCVTDQ2PS \_\_m256 \_\_mm256\_maskz\_cvtepi32\_ps( \_\_mmask8 k, \_\_m256i a);  
 VCVTDQ2PS \_\_m128 \_\_mm\_mask\_cvtepi32\_ps( \_\_m128 s, \_\_mmask8 k, \_\_m128i a);  
 VCVTDQ2PS \_\_m128 \_\_mm\_maskz\_cvtepi32\_ps( \_\_mmask8 k, \_\_m128i a);  
 CVTDQ2PS \_\_m256 \_\_mm256\_cvtepi32\_ps( \_\_m256i src)  
 CVTDQ2PS \_\_m128 \_\_mm\_cvtepi32\_ps( \_\_m128i src)

**SIMD Floating-Point Exceptions**

Precision

**Other Exceptions**

VEX-encoded instructions, see Exceptions Type 2;

EVEX-encoded instructions, see Exceptions Type E2.

#UD If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

## CVTPD2DQ—Convert Packed Double-Precision Floating-Point Values to Packed Doubleword Integers

Opcode Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F E6 /r CVTPD2DQ xmm1, xmm2/m128	RM	V/V	SSE2	Convert two packed double-precision floating-point values in xmm2/mem to two signed doubleword integers in xmm1.
VEX.128.F2.0F.WIG E6 /r VCVTPD2DQ xmm1, xmm2/m128	RM	V/V	AVX	Convert two packed double-precision floating-point values in xmm2/mem to two signed doubleword integers in xmm1.
VEX.256.F2.0F.WIG E6 /r VCVTPD2DQ xmm1, ymm2/m256	RM	V/V	AVX	Convert four packed double-precision floating-point values in ymm2/mem to four signed doubleword integers in xmm1.
EVEX.128.F2.0F.W1 E6 /r VCVTPD2DQ xmm1 {k1}{z}, xmm2/m128/m64bcst	FV	V/V	AVX512VL AVX512F	Convert two packed double-precision floating-point values in xmm2/m128/m64bcst to two signed doubleword integers in xmm1 subject to writemask k1.
EVEX.256.F2.0F.W1 E6 /r VCVTPD2DQ xmm1 {k1}{z}, ymm2/m256/m64bcst	FV	V/V	AVX512VL AVX512F	Convert four packed double-precision floating-point values in ymm2/m256/m64bcst to four signed doubleword integers in xmm1 subject to writemask k1.
EVEX.512.F2.0F.W1 E6 /r VCVTPD2DQ ymm1 {k1}{z}, zmm2/m512/m64bcst{er}	FV	V/V	AVX512F	Convert eight packed double-precision floating-point values in zmm2/m512/m64bcst to eight signed doubleword integers in ymm1 subject to writemask k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
FV	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Converts packed double-precision floating-point values in the source operand (second operand) to packed signed doubleword integers in the destination operand (first operand).

When a conversion is inexact, 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, the indefinite integer value ( $2^{w-1}$ , where  $w$  represents the number of bits in the destination format) is returned.

EVEX encoded versions: The source operand is a ZMM/YMM/XMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1. The upper bits (MAX\_VL-1:256/128/64) of the corresponding destination are zeroed.

VEX.256 encoded version: The source operand is a YMM register or 256-bit memory location. The destination operand is an XMM register. The upper bits (MAX\_VL-1:128) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: The source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAX\_VL-1:64) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. Bits[127:64] of the destination XMM register are zeroed. However, the upper bits (MAX\_VL-1:128) of the corresponding ZMM register destination are unmodified.

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

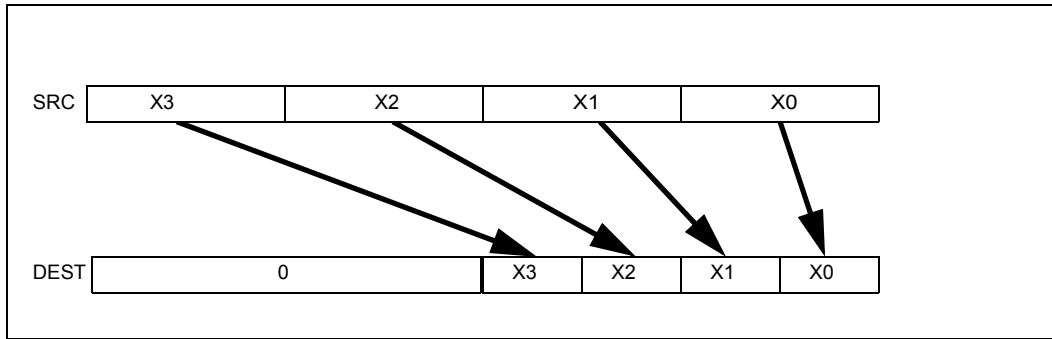


Figure 3-9. VCVTPD2DQ (VEX.256 encoded version)

### Operation

**VCVTPD2DQ (EVEX encoded versions) when src operand is a register**

(KL, VL) = (2, 128), (4, 256), (8, 512)

IF (VL = 512) AND (EVEX.b = 1)

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j \* 32

k ← j \* 64

IF k1[j] OR \*no writemask\*

THEN DEST[i+31:i] ←

Convert\_Double\_Precision\_Floating\_Point\_To\_Integer(SRC[k+63:k])

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+31:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL/2] ← 0

**VCVTPD2DQ (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\_Double\_Precision\_Floating\_Point\_To\_Integer(SRC[63:0])

ELSE

DEST[i+31:i] ←

Convert\_Double\_Precision\_Floating\_Point\_To\_Integer(SRC[k+63:k])

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+31:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL/2] ← 0

**VCVTPD2DQ (VEX.256 encoded version)**

DEST[31:0] ← Convert\_Double\_Precision\_Floating\_Point\_To\_Integer(SRC[63:0])

DEST[63:32] ← Convert\_Double\_Precision\_Floating\_Point\_To\_Integer(SRC[127:64])

DEST[95:64] ← Convert\_Double\_Precision\_Floating\_Point\_To\_Integer(SRC[191:128])

DEST[127:96] ← Convert\_Double\_Precision\_Floating\_Point\_To\_Integer(SRC[255:192])

DEST[MAX\_VL-1:128] ← 0

**VCVTPD2DQ (VEX.128 encoded version)**

DEST[31:0] ← Convert\_Double\_Precision\_Floating\_Point\_To\_Integer(SRC[63:0])

DEST[63:32] ← Convert\_Double\_Precision\_Floating\_Point\_To\_Integer(SRC[127:64])

DEST[MAX\_VL-1:64] ← 0

**CVTPD2DQ (128-bit Legacy SSE version)**

DEST[31:0] ← Convert\_Double\_Precision\_Floating\_Point\_To\_Integer(SRC[63:0])

DEST[63:32] ← Convert\_Double\_Precision\_Floating\_Point\_To\_Integer(SRC[127:64])

DEST[127:64] ← 0

DEST[MAX\_VL-1:128] (unmodified)

### Intel C/C++ Compiler Intrinsic Equivalent

```

VCVTPD2DQ __m256i __mm512_cvtpd_epi32( __m512d a);
VCVTPD2DQ __m256i __mm512_mask_cvtpd_epi32( __m256i s, __mmask8 k, __m512d a);
VCVTPD2DQ __m256i __mm512_maskz_cvtpd_epi32( __mmask8 k, __m512d a);
VCVTPD2DQ __m256i __mm512_cvt_roundpd_epi32( __m512d a, int r);
VCVTPD2DQ __m256i __mm512_mask_cvt_roundpd_epi32( __m256i s, __mmask8 k, __m512d a, int r);
VCVTPD2DQ __m256i __mm512_maskz_cvt_roundpd_epi32( __mmask8 k, __m512d a, int r);
VCVTPD2DQ __m128i __mm256_mask_cvtpd_epi32( __m128i s, __mmask8 k, __m256d a);
VCVTPD2DQ __m128i __mm256_maskz_cvtpd_epi32( __mmask8 k, __m256d a);
VCVTPD2DQ __m128i __mm_mask_cvtpd_epi32( __m128i s, __mmask8 k, __m128d a);
VCVTPD2DQ __m128i __mm_maskz_cvtpd_epi32( __mmask8 k, __m128d a);
VCVTPD2DQ __m128i __mm256_cvtpd_epi32( __m256d src)
CVTPD2DQ __m128i __mm_cvtpd_epi32( __m128d src)

```

### SIMD Floating-Point Exceptions

Invalid, Precision

### Other Exceptions

See Exceptions Type 2; additionally

EVEX-encoded instructions, see Exceptions Type E2.

#UD If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

## CVTPD2PS—Convert Packed Double-Precision Floating-Point Values to Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 5A /r CVTPD2PS xmm1, xmm2/m128	RM	V/V	SSE2	Convert two packed double-precision floating-point values in xmm2/mem to two single-precision floating-point values in xmm1.
VEX.128.66.0F.WIG 5A /r VCVTPD2PS xmm1, xmm2/m128	RM	V/V	AVX	Convert two packed double-precision floating-point values in xmm2/mem to two single-precision floating-point values in xmm1.
VEX.256.66.0F.WIG 5A /r VCVTPD2PS xmm1, ymm2/m256	RM	V/V	AVX	Convert four packed double-precision floating-point values in ymm2/mem to four single-precision floating-point values in xmm1.
EVEX.128.66.0F.W1 5A /r VCVTPD2PS xmm1 {k1}{z}, xmm2/m128/m64bcst	FV	V/V	AVX512VL AVX512F	Convert two packed double-precision floating-point values in xmm2/m128/m64bcst to two single-precision floating-point values in xmm1 with writemask k1.
EVEX.256.66.0F.W1 5A /r VCVTPD2PS xmm1 {k1}{z}, ymm2/m256/m64bcst	FV	V/V	AVX512VL AVX512F	Convert four packed double-precision floating-point values in ymm2/m256/m64bcst to four single-precision floating-point values in xmm1 with writemask k1.
EVEX.512.66.0F.W1 5A /r VCVTPD2PS ymm1 {k1}{z}, zmm2/m512/m64bcst{er}	FV	V/V	AVX512F	Convert eight packed double-precision floating-point values in zmm2/m512/m64bcst to eight single-precision floating-point values in ymm1 with writemask k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
FV	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Converts two, four or eight packed double-precision floating-point values in the source operand (second operand) to two, four or eight packed single-precision floating-point values in the destination operand (first operand).

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits.

EVEX encoded versions: 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 64-bit memory location. The destination operand is a YMM/XMM/XMM (low 64-bits) register conditionally updated with writemask k1. The upper bits (MAX\_VL-1:256/128/64) of the corresponding destination are zeroed.

VEX.256 encoded version: The source operand is a YMM register or 256-bit memory location. The destination operand is an XMM register. The upper bits (MAX\_VL-1:128) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: The source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAX\_VL-1:64) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. Bits[127:64] of the destination XMM register are zeroed. However, the upper Bits (MAX\_VL-1:128) of the corresponding ZMM register destination are unmodified.

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

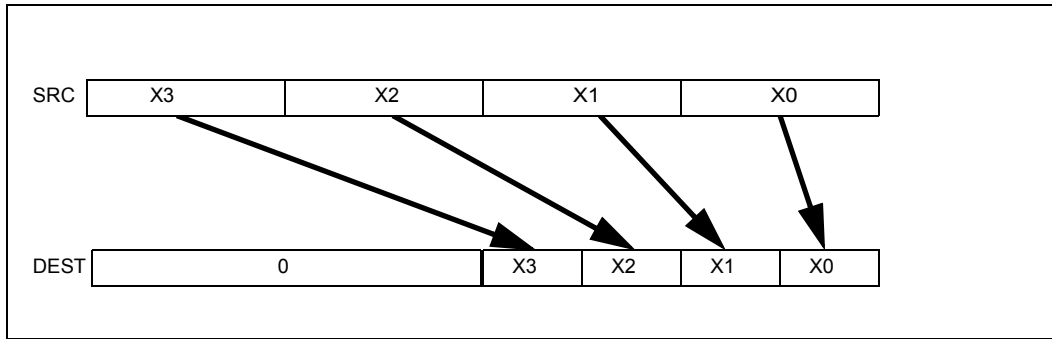


Figure 3-10. VCVTPD2PS (VEX.256 encoded version)

### Operation

**VCVTPD2PS (EVEX encoded version) when src operand is a register**

(KL, VL) = (2, 128), (4, 256), (8, 512)

IF (VL = 512) AND (EVEX.b = 1)

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j \* 32

k ← j \* 64

IF k1[j] OR \*no writemask\*

THEN

DEST[i+31:i] ← Convert\_Double\_Precision\_Floating\_Point\_To\_Single\_Precision\_Floating\_Point(SRC[k+63:k])

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+31:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL/2] ← 0

**VCVTPD2PS (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 * 32
  k ← j * 64
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1)
        THEN
          DEST[i+31:i] ← Convert_Double_Precision_Floating_Point_To_Single_Precision_Floating_Point(SRC[63:0])
        ELSE
          DEST[i+31:i] ← Convert_Double_Precision_Floating_Point_To_Single_Precision_Floating_Point(SRC[k+63:k])
        FI;
      ELSE
        IF *merging-masking*           ; merging-masking
          THEN *DEST[i+31:i] remains unchanged*
        ELSE                             ; zeroing-masking
          DEST[i+31:i] ← 0
        FI
      FI;
    ENDFOR
  DEST[MAX_VL-1:VL/2] ← 0

```

**VCVTPD2PS (VEX.256 encoded version)**

```

DEST[31:0] ← Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC[63:0])
DEST[63:32] ← Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC[127:64])
DEST[95:64] ← Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC[191:128])
DEST[127:96] ← Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC[255:192])
DEST[MAX_VL-1:128] ← 0

```

**VCVTPD2PS (VEX.128 encoded version)**

```

DEST[31:0] ← Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC[63:0])
DEST[63:32] ← Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC[127:64])
DEST[MAX_VL-1:64] ← 0

```

**CVTPD2PS (128-bit Legacy SSE version)**

```

DEST[31:0] ← Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC[63:0])
DEST[63:32] ← Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC[127:64])
DEST[127:64] ← 0
DEST[MAX_VL-1:128] (unmodified)

```



### Intel C/C++ Compiler Intrinsic Equivalent

```

VCVTPD2PS __m256 __mm512_cvtpd_ps( __m512d a);
VCVTPD2PS __m256 __mm512_mask_cvtpd_ps( __m256 s, __mmask8 k, __m512d a);
VCVTPD2PS __m256 __mm512_maskz_cvtpd_ps( __mmask8 k, __m512d a);
VCVTPD2PS __m256 __mm512_cvt_roundpd_ps( __m512d a, int r);
VCVTPD2PS __m256 __mm512_mask_cvt_roundpd_ps( __m256 s, __mmask8 k, __m512d a, int r);
VCVTPD2PS __m256 __mm512_maskz_cvt_roundpd_ps( __mmask8 k, __m512d a, int r);
VCVTPD2PS __m128 __mm256_mask_cvtpd_ps( __m128 s, __mmask8 k, __m256d a);
VCVTPD2PS __m128 __mm256_maskz_cvtpd_ps( __mmask8 k, __m256d a);
VCVTPD2PS __m128 __mm_mask_cvtpd_ps( __m128 s, __mmask8 k, __m128d a);
VCVTPD2PS __m128 __mm_maskz_cvtpd_ps( __mmask8 k, __m128d a);
VCVTPD2PS __m128 __mm256_cvtpd_ps( __m256d a)
CVTPD2PS __m128 __mm_cvtpd_ps( __m128d a)

```

### SIMD Floating-Point Exceptions

Invalid, Precision, Underflow, Overflow, Denormal

### Other Exceptions

VEX-encoded instructions, see Exceptions Type 2;

EVEX-encoded instructions, see Exceptions Type E2.

#UD If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

## CVTPS2DQ—Convert Packed Single-Precision Floating-Point Values to Packed Signed Doubleword Integer Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 5B /r CVTPS2DQ xmm1, xmm2/m128	RM	V/V	SSE2	Convert four packed single-precision floating-point values from xmm2/mem to four packed signed doubleword values in xmm1.
VEX.128.66.0F.WIG 5B /r VCVTPS2DQ xmm1, xmm2/m128	RM	V/V	AVX	Convert four packed single-precision floating-point values from xmm2/mem to four packed signed doubleword values in xmm1.
VEX.256.66.0F.WIG 5B /r VCVTPS2DQ ymm1, ymm2/m256	RM	V/V	AVX	Convert eight packed single-precision floating-point values from ymm2/mem to eight packed signed doubleword values in ymm1.
EVEX.128.66.0F.W0 5B /r VCVTPS2DQ xmm1 {k1}{z}, xmm2/m128/m32bcst	FV	V/V	AVX512VL AVX512F	Convert four packed single precision floating-point values from xmm2/m128/m32bcst to four packed signed doubleword values in xmm1 subject to writemask k1.
EVEX.256.66.0F.W0 5B /r VCVTPS2DQ ymm1 {k1}{z}, ymm2/m256/m32bcst	FV	V/V	AVX512VL AVX512F	Convert eight packed single precision floating-point values from ymm2/m256/m32bcst to eight packed signed doubleword values in ymm1 subject to writemask k1.
EVEX.512.66.0F.W0 5B /r VCVTPS2DQ zmm1 {k1}{z}, zmm2/m512/m32bcst[er]	FV	V/V	AVX512F	Convert sixteen packed single-precision floating-point values from zmm2/m512/m32bcst to sixteen packed signed doubleword values in zmm1 subject to writemask k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
FV	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Converts four, eight or sixteen packed single-precision floating-point values in the source operand to four, eight or sixteen signed doubleword integers in the destination operand.

When a conversion is inexact, 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, the indefinite integer value ( $2^{w-1}$ , where  $w$  represents the number of bits in the destination format) is returned.

EVEX encoded versions: The source operand is a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM register conditionally updated with writemask k1.

VEX.256 encoded version: The source operand is a YMM register or 256-bit memory location. The destination operand is a YMM register. The upper bits (MAX\_VL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: The source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAX\_VL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAX\_VL-1:128) of the corresponding ZMM register destination are unmodified.

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

**Operation****VCVTSP2DQ (encoded versions) when src operand is a register**

(KL, VL) = (4, 128), (8, 256), (16, 512)

IF (VL = 512) AND (EVEX.b = 1)

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\*

THEN DEST[i+31:i] ←

Convert\_Single\_Precision\_Floating\_Point\_To\_Integer(SRC[i+31:i])

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+31:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VCVTSP2DQ (EVEX encoded versions) when src operand is a memory source**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO 15

i ← j \* 32

IF k1[j] OR \*no writemask\*

THEN

IF (EVEX.b = 1)

THEN

DEST[i+31:i] ←

Convert\_Single\_Precision\_Floating\_Point\_To\_Integer(SRC[31:0])

ELSE

DEST[i+31:i] ←

Convert\_Single\_Precision\_Floating\_Point\_To\_Integer(SRC[i+31:i])

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+31:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VCVTPS2DQ (VEX.256 encoded version)**

DEST[31:0] ← Convert\_Single\_Precision\_Floating\_Point\_To\_Integer(SRC[31:0])  
 DEST[63:32] ← Convert\_Single\_Precision\_Floating\_Point\_To\_Integer(SRC[63:32])  
 DEST[95:64] ← Convert\_Single\_Precision\_Floating\_Point\_To\_Integer(SRC[95:64])  
 DEST[127:96] ← Convert\_Single\_Precision\_Floating\_Point\_To\_Integer(SRC[127:96])  
 DEST[159:128] ← Convert\_Single\_Precision\_Floating\_Point\_To\_Integer(SRC[159:128])  
 DEST[191:160] ← Convert\_Single\_Precision\_Floating\_Point\_To\_Integer(SRC[191:160])  
 DEST[223:192] ← Convert\_Single\_Precision\_Floating\_Point\_To\_Integer(SRC[223:192])  
 DEST[255:224] ← Convert\_Single\_Precision\_Floating\_Point\_To\_Integer(SRC[255:224])

**VCVTPS2DQ (VEX.128 encoded version)**

DEST[31:0] ← Convert\_Single\_Precision\_Floating\_Point\_To\_Integer(SRC[31:0])  
 DEST[63:32] ← Convert\_Single\_Precision\_Floating\_Point\_To\_Integer(SRC[63:32])  
 DEST[95:64] ← Convert\_Single\_Precision\_Floating\_Point\_To\_Integer(SRC[95:64])  
 DEST[127:96] ← Convert\_Single\_Precision\_Floating\_Point\_To\_Integer(SRC[127:96])  
 DEST[MAX\_VL-1:128] ← 0

**CVTTPS2DQ (128-bit Legacy SSE version)**

DEST[31:0] ← Convert\_Single\_Precision\_Floating\_Point\_To\_Integer(SRC[31:0])  
 DEST[63:32] ← Convert\_Single\_Precision\_Floating\_Point\_To\_Integer(SRC[63:32])  
 DEST[95:64] ← Convert\_Single\_Precision\_Floating\_Point\_To\_Integer(SRC[95:64])  
 DEST[127:96] ← Convert\_Single\_Precision\_Floating\_Point\_To\_Integer(SRC[127:96])  
 DEST[MAX\_VL-1:128] (unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**

VCVTPS2DQ \_\_m512i \_\_mm512\_cvtps\_epi32( \_\_m512 a);  
 VCVTPS2DQ \_\_m512i \_\_mm512\_mask\_cvtps\_epi32( \_\_m512i s, \_\_mmask16 k, \_\_m512 a);  
 VCVTPS2DQ \_\_m512i \_\_mm512\_maskz\_cvtps\_epi32( \_\_mmask16 k, \_\_m512 a);  
 VCVTPS2DQ \_\_m512i \_\_mm512\_cvt\_roundps\_epi32( \_\_m512 a, int r);  
 VCVTPS2DQ \_\_m512i \_\_mm512\_mask\_cvt\_roundps\_epi32( \_\_m512i s, \_\_mmask16 k, \_\_m512 a, int r);  
 VCVTPS2DQ \_\_m512i \_\_mm512\_maskz\_cvt\_roundps\_epi32( \_\_mmask16 k, \_\_m512 a, int r);  
 VCVTPS2DQ \_\_m256i \_\_mm256\_mask\_cvtps\_epi32( \_\_m256i s, \_\_mmask8 k, \_\_m256 a);  
 VCVTPS2DQ \_\_m256i \_\_mm256\_maskz\_cvtps\_epi32( \_\_mmask8 k, \_\_m256 a);  
 VCVTPS2DQ \_\_m128i \_\_mm\_mask\_cvtps\_epi32( \_\_m128i s, \_\_mmask8 k, \_\_m128 a);  
 VCVTPS2DQ \_\_m128i \_\_mm\_maskz\_cvtps\_epi32( \_\_mmask8 k, \_\_m128 a);  
 VCVTPS2DQ \_\_m256i \_\_mm256\_cvtps\_epi32( \_\_m256 a)  
 CVTTPS2DQ \_\_m128i \_\_mm\_cvtps\_epi32( \_\_m128 a)

**SIMD Floating-Point Exceptions**

Invalid, Precision

**Other Exceptions**

VEX-encoded instructions, see Exceptions Type 2;

EVEX-encoded instructions, see Exceptions Type E2.

#UD If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

## CVTPS2PD—Convert Packed Single-Precision Floating-Point Values to Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
OF 5A /r CVTPS2PD xmm1, xmm2/m64	RM	V/V	SSE2	Convert two packed single-precision floating-point values in xmm2/m64 to two packed double-precision floating-point values in xmm1.
VEX.128.OF.WIG 5A /r VCVTPS2PD xmm1, xmm2/m64	RM	V/V	AVX	Convert two packed single-precision floating-point values in xmm2/m64 to two packed double-precision floating-point values in xmm1.
VEX.256.OF.WIG 5A /r VCVTPS2PD ymm1, xmm2/m128	RM	V/V	AVX	Convert four packed single-precision floating-point values in xmm2/m128 to four packed double-precision floating-point values in ymm1.
EVEX.128.OF.W0 5A /r VCVTPS2PD xmm1 {k1}{z}, xmm2/m64/m32bcst	HV	V/V	AVX512VL AVX512F	Convert two packed single-precision floating-point values in xmm2/m64/m32bcst to packed double-precision floating-point values in xmm1 with writemask k1.
EVEX.256.OF.W0 5A /r VCVTPS2PD ymm1 {k1}{z}, xmm2/m128/m32bcst	HV	V/V	AVX512VL	Convert four packed single-precision floating-point values in xmm2/m128/m32bcst to packed double-precision floating-point values in ymm1 with writemask k1.
EVEX.512.OF.W0 5A /r VCVTPS2PD zmm1 {k1}{z}, ymm2/m256/m32bcst{sae}	HV	V/V	AVX512F	Convert eight packed single-precision floating-point values in ymm2/m256/b32bcst to eight packed double-precision floating-point values in zmm1 with writemask k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
HV	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Converts two, four or eight packed single-precision floating-point values in the source operand (second operand) to two, four or eight packed double-precision floating-point values in the destination operand (first operand).

EVEX encoded versions: The source operand is a YMM/XMM/XMM (low 64-bits) register, a 256/128/64-bit memory location or a 256/128/64-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

VEX.256 encoded version: The source operand is an XMM register or 128-bit memory location. The destination operand is a YMM register. Bits (MAX\_VL-1:256) of the corresponding destination ZMM register are zeroed.

VEX.128 encoded version: The source operand is an XMM register or 64-bit memory location. The destination operand is an XMM register. The upper Bits (MAX\_VL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The source operand is an XMM register or 64-bit memory location. The destination operand is an XMM register. The upper Bits (MAX\_VL-1:128) of the corresponding ZMM register destination are unmodified.

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

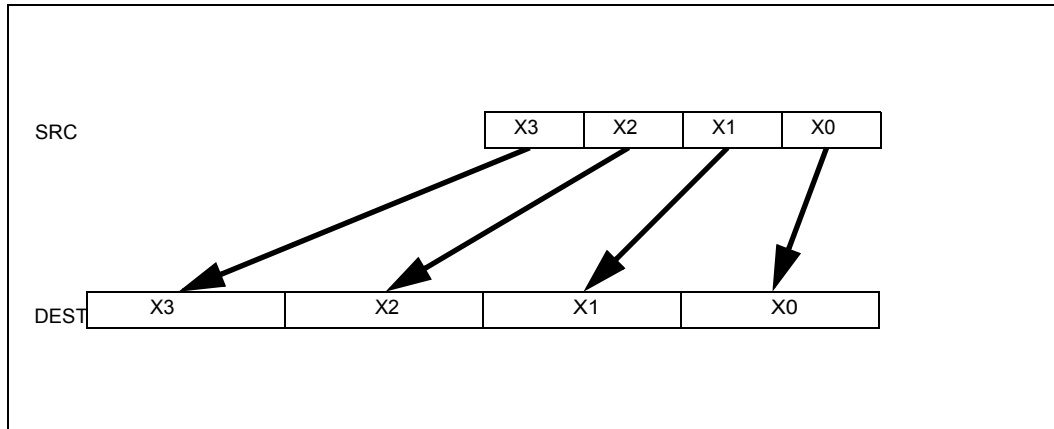


Figure 3-11. CVTSP2PD (VEX.256 encoded version)

### Operation

**VCVTSP2PD (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\_Single\_Precision\_To\_Double\_Precision\_Floating\_Point(SRC[k+31:k])

  ELSE

    IF \*merging-masking\* ; merging-masking

      THEN \*DEST[i+63:i] remains unchanged\*

    ELSE ; zeroing-masking

      DEST[i+63:i] ← 0

  FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VCVTSP2PD (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\_Single\_Precision\_To\_Double\_Precision\_Floating\_Point(SRC[31:0])

        ELSE

          DEST[i+63:i] ←

          Convert\_Single\_Precision\_To\_Double\_Precision\_Floating\_Point(SRC[k+31:k])

      FI;

    ELSE

```

        IF *merging-masking*           ; merging-masking
            THEN *DEST[i+63:i] remains unchanged*
            ELSE                         ; zeroing-masking
                DEST[i+63:i] ← 0
        FI
FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VCVTSP2PD (VEX.256 encoded version)**

```

DEST[63:0] ← Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[31:0])
DEST[127:64] ← Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[63:32])
DEST[191:128] ← Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[95:64])
DEST[255:192] ← Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[127:96])
DEST[MAX_VL-1:256] ← 0

```

**VCVTSP2PD (VEX.128 encoded version)**

```

DEST[63:0] ← Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[31:0])
DEST[127:64] ← Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[63:32])
DEST[MAX_VL-1:128] ← 0

```

**CVTSP2PD (128-bit Legacy SSE version)**

```

DEST[63:0] ← Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[31:0])
DEST[127:64] ← Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[63:32])
DEST[MAX_VL-1:128] (unmodified)

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VCVTSP2PD __m512d __mm512_cvtps_pd( __m256 a);
VCVTSP2PD __m512d __mm512_mask_cvtps_pd( __m512d s, __mmask8 k, __m256 a);
VCVTSP2PD __m512d __mm512_maskz_cvtps_pd( __mmask8 k, __m256 a);
VCVTSP2PD __m512d __mm512_cvt_roundps_pd( __m256 a, int sae);
VCVTSP2PD __m512d __mm512_mask_cvt_roundps_pd( __m512d s, __mmask8 k, __m256 a, int sae);
VCVTSP2PD __m512d __mm512_maskz_cvt_roundps_pd( __mmask8 k, __m256 a, int sae);
VCVTSP2PD __m256d __mm256_mask_cvtps_pd( __m256d s, __mmask8 k, __m128 a);
VCVTSP2PD __m256d __mm256_maskz_cvtps_pd( __mmask8 k, __m128a);
VCVTSP2PD __m128d __mm_mask_cvtps_pd( __m128d s, __mmask8 k, __m128 a);
VCVTSP2PD __m128d __mm_maskz_cvtps_pd( __mmask8 k, __m128 a);
VCVTSP2PD __m256d __mm256_cvtps_pd( __m128 a)
CVTSP2PD __m128d __mm_cvtps_pd( __m128 a)

```

**SIMD Floating-Point Exceptions**

Invalid, Denormal

**Other Exceptions**

VEX-encoded instructions, see Exceptions Type 3;

EVEX-encoded instructions, see Exceptions Type E3.

#UD If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

## CVTSD2SI—Convert Scalar Double-Precision Floating-Point Value to Doubleword Integer

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 OF 2D /r CVTSD2SI r32, xmm1/m64	RM	V/V	SSE2	Convert one double-precision floating-point value from xmm1/m64 to one signed doubleword integer r32.
F2 REX.W OF 2D /r CVTSD2SI r64, xmm1/m64	RM	V/N.E.	SSE2	Convert one double-precision floating-point value from xmm1/m64 to one signed quadword integer sign-extended into r64.
VEX.128.F2.OF.W0 2D /r VCVTSD2SI r32, xmm1/m64	RM	V/V	AVX	Convert one double-precision floating-point value from xmm1/m64 to one signed doubleword integer r32.
VEX.128.F2.OF.W1 2D /r VCVTSD2SI r64, xmm1/m64	RM	V/N.E. <sup>1</sup>	AVX	Convert one double-precision floating-point value from xmm1/m64 to one signed quadword integer sign-extended into r64.
EVEX.LIG.F2.OF.W0 2D /r VCVTSD2SI r32, xmm1/m64{er}	T1F	V/V	AVX512F	Convert one double-precision floating-point value from xmm1/m64 to one signed doubleword integer r32.
EVEX.LIG.F2.OF.W1 2D /r VCVTSD2SI r64, xmm1/m64{er}	T1F	V/N.E. <sup>1</sup>	AVX512F	Convert one double-precision floating-point value from xmm1/m64 to one signed quadword integer sign-extended into r64.

### NOTES:

1. VEX.W1/EVEX.W1 in non-64 bit is ignored; the instructions behaves as if the W0 version is used.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
T1F	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Converts a double-precision floating-point value in the source operand (the second operand) to a signed doubleword integer in the destination operand (first operand). 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 value returned is rounded according to the rounding control bits in the MXCSR register.

If a converted result exceeds the range limits of signed doubleword integer (in non-64-bit modes or 64-bit mode with REX.W/VEX.W/EVEX.W=0), the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value (80000000H) is returned.

If a converted result exceeds the range limits of signed quadword integer (in 64-bit mode and REX.W/VEX.W/EVEX.W = 1), the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value (80000000\_00000000H) is returned.

Legacy SSE instruction: Use of the REX.W prefix promotes the instruction to produce 64-bit data in 64-bit mode. See the summary chart at the beginning of this section for encoding data and limits.

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

Software should ensure VCVTSD2SI is encoded with VEX.L=0. Encoding VCVTSD2SI with VEX.L=1 may encounter unpredictable behavior across different processor generations.



## Operation

### VCVTSD2SI (EVEX encoded version)

```

IF SRC *is register* AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
IF 64-Bit Mode and OperandSize = 64
    THEN DEST[63:0] ← Convert_Double_Precision_Floating_Point_To_Integer(SRC[63:0]);
    ELSE DEST[31:0] ← Convert_Double_Precision_Floating_Point_To_Integer(SRC[63:0]);
FI

```

### (V)CVTSD2SI

```

IF 64-Bit Mode and OperandSize = 64
    THEN
        DEST[63:0] ← Convert_Double_Precision_Floating_Point_To_Integer(SRC[63:0]);
    ELSE
        DEST[31:0] ← Convert_Double_Precision_Floating_Point_To_Integer(SRC[63:0]);
FI;

```

### Intel C/C++ Compiler Intrinsic Equivalent

```

VCVTSD2SI int _mm_cvtsd_i32(__m128d);
VCVTSD2SI int _mm_cvt_roundsd_i32(__m128d, int r);
VCVTSD2SI __int64 _mm_cvtsd_i64(__m128d);
VCVTSD2SI __int64 _mm_cvt_roundsd_i64(__m128d, int r);
CVTSD2SI __int64 _mm_cvtsd_si64(__m128d);
CVTSD2SI int _mm_cvtsd_si32(__m128d a)

```

### SIMD Floating-Point Exceptions

Invalid, Precision

### Other Exceptions

VEX-encoded instructions, see Exceptions Type 3;

EVEX-encoded instructions, see Exceptions Type E3NF.

#UD If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

## CVTSD2SS—Convert Scalar Double-Precision Floating-Point Value to Scalar Single-Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 5A /r CVTSD2SS xmm1, xmm2/m64	RM	V/V	SSE2	Convert one double-precision floating-point value in xmm2/m64 to one single-precision floating-point value in xmm1.
VEX.NDS.128.F2.0F.WIG 5A /r VCVTSD2SS xmm1, xmm2, xmm3/m64	RVM	V/V	AVX	Convert one double-precision floating-point value in xmm3/m64 to one single-precision floating-point value and merge with high bits in xmm2.
EVEX.NDS.LIG.F2.0F.W1 5A /r VCVTSD2SS xmm1 {k1}{z}, xmm2, xmm3/m64{er}	T1S	V/V	AVX512F	Convert one double-precision floating-point value in xmm3/m64 to one single-precision floating-point value and merge with high bits in xmm2 under writemask k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
T1S	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

### Description

Converts a double-precision floating-point value in the “convert-from” source operand (the second operand in SSE2 version, otherwise the third operand) to a single-precision floating-point value in the destination operand.

When the “convert-from” operand is an XMM register, the double-precision floating-point value is contained in the low quadword of the register. The result is stored in the low doubleword of the destination operand. When the conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register.

128-bit Legacy SSE version: The “convert-from” source operand (the second operand) is an XMM register or memory location. Bits (MAX\_VL-1:32) of the corresponding destination register remain unchanged. The destination operand is an XMM register.

VEX.128 and EVEX encoded versions: The “convert-from” source operand (the third operand) can be an XMM register or a 64-bit memory location. The first source and destination operands are XMM registers. Bits (127:32) of the XMM register destination are copied from the corresponding bits in the first source operand. Bits (MAX\_VL-1:128) of the destination register are zeroed.

EVEX encoded version: the converted result is written to the low doubleword element of the destination under the writemask.

Software should ensure VCVTSD2SS is encoded with VEX.L=0. Encoding VCVTSD2SS with VEX.L=1 may encounter unpredictable behavior across different processor generations.

## Operation

### VCVTSD2SS (EVEX encoded version)

```

IF (SRC2 *is register*) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
    THEN DEST[31:0] ← Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC2[63:0]);
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE ; zeroing-masking
                THEN DEST[31:0] ← 0
        FI;
FI;
DEST[127:32] ← SRC1[127:32]
DEST[MAX_VL-1:128] ← 0

```

### VCVTSD2SS (VEX.128 encoded version)

```

DEST[31:0] ← Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC2[63:0]);
DEST[127:32] ← SRC1[127:32]
DEST[MAX_VL-1:128] ← 0

```

### CVTSD2SS (128-bit Legacy SSE version)

```

DEST[31:0] ← Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC[63:0]);
(* DEST[MAX_VL-1:32] Unmodified *)

```

## Intel C/C++ Compiler Intrinsic Equivalent

```

VCVTSD2SS __m128_mm_mask_cvtssd_ss(__m128 s, __mmask8 k, __m128 a, __m128d b);
VCVTSD2SS __m128_mm_maskz_cvtssd_ss(__mmask8 k, __m128 a, __m128d b);
VCVTSD2SS __m128_mm_cvt_roundsd_ss(__m128 a, __m128d b, int r);
VCVTSD2SS __m128_mm_mask_cvt_roundsd_ss(__m128 s, __mmask8 k, __m128 a, __m128d b, int r);
VCVTSD2SS __m128_mm_maskz_cvt_roundsd_ss(__mmask8 k, __m128 a, __m128d b, int r);
CVTSD2SS __m128_mm_cvtssd_ss(__m128 a, __m128d b)

```

## SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

## Other Exceptions

VEX-encoded instructions, see Exceptions Type 3.

EVEX-encoded instructions, see Exceptions Type E3.

## CVTISI2SD—Convert Doubleword Integer to Scalar Double-Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 OF 2A /r CVTISI2SD xmm1, r32/m32	RM	V/V	SSE2	Convert one signed doubleword integer from r32/m32 to one double-precision floating-point value in xmm1.
F2 REX.W OF 2A /r CVTISI2SD xmm1, r/m64	RM	V/N.E.	SSE2	Convert one signed quadword integer from r/m64 to one double-precision floating-point value in xmm1.
VEX.NDS.128.F2.OF.W0 2A /r VCVTISI2SD xmm1, xmm2, r/m32	RVM	V/V	AVX	Convert one signed doubleword integer from r/m32 to one double-precision floating-point value in xmm1.
VEX.NDS.128.F2.OF.W1 2A /r VCVTISI2SD xmm1, xmm2, r/m64	RVM	V/N.E. <sup>1</sup>	AVX	Convert one signed quadword integer from r/m64 to one double-precision floating-point value in xmm1.
EVEX.NDS.LIG.F2.OF.W0 2A /r VCVTISI2SD xmm1, xmm2, r/m32	T1S	V/V	AVX512F	Convert one signed doubleword integer from r/m32 to one double-precision floating-point value in xmm1.
EVEX.NDS.LIG.F2.OF.W1 2A /r VCVTISI2SD xmm1, xmm2, r/m64{er}	T1S	V/N.E. <sup>1</sup>	AVX512F	Convert one signed quadword integer from r/m64 to one double-precision floating-point value in xmm1.

### NOTES:

1. VEX.W1/EVEX.W1 in non-64 bit is ignored; the instructions behaves as if the W0 version is used.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
T1S	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

### Description

Converts a signed doubleword integer (or signed quadword integer if operand size is 64 bits) in the “convert-from” source operand to a double-precision floating-point value in the destination operand. The result is stored in the low quadword of the destination operand, and the high quadword left unchanged. When conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register.

The second source operand can be a general-purpose register or a 32/64-bit memory location. The first source and destination operands are XMM registers.

128-bit Legacy SSE version: Use of the REX.W prefix promotes the instruction to 64-bit operands. The “convert-from” source operand (the second operand) is a general-purpose register or memory location. The destination is an XMM register Bits (MAX\_VL-1:64) of the corresponding destination register remain unchanged.

VEX.128 and EVEX encoded versions: The “convert-from” source operand (the third operand) can be a general-purpose register or a memory location. The first source and destination operands are XMM registers. Bits (127:64) of the XMM register destination are copied from the corresponding bits in the first source operand. Bits (MAX\_VL-1:128) of the destination register are zeroed.

EVEX.W0 version: attempt to encode this instruction with EVEX embedded rounding is ignored.

VEX.W1 and EVEX.W1 versions: promotes the instruction to use 64-bit input value in 64-bit mode.

Software should ensure VCVTISI2SD is encoded with VEX.L=0. Encoding VCVTISI2SD with VEX.L=1 may encounter unpredictable behavior across different processor generations.

## Operation

### VCVTSI2SD (EVEX encoded version)

```

IF (SRC2 *is register*) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
IF 64-Bit Mode And OperandSize = 64
    THEN
        DEST[63:0] ← Convert_Integer_To_Double_Precision_Floating_Point(SRC2[63:0]);
    ELSE
        DEST[63:0] ← Convert_Integer_To_Double_Precision_Floating_Point(SRC2[31:0]);
FI;
DEST[127:64] ← SRC1[127:64]
DEST[MAX_VL-1:128] ← 0

```

### VCVTSI2SD (VEX.128 encoded version)

```

IF 64-Bit Mode And OperandSize = 64
    THEN
        DEST[63:0] ← Convert_Integer_To_Double_Precision_Floating_Point(SRC2[63:0]);
    ELSE
        DEST[63:0] ← Convert_Integer_To_Double_Precision_Floating_Point(SRC2[31:0]);
FI;
DEST[127:64] ← SRC1[127:64]
DEST[MAX_VL-1:128] ← 0

```

### CVTSI2SD

```

IF 64-Bit Mode And OperandSize = 64
    THEN
        DEST[63:0] ← Convert_Integer_To_Double_Precision_Floating_Point(SRC[63:0]);
    ELSE
        DEST[63:0] ← Convert_Integer_To_Double_Precision_Floating_Point(SRC[31:0]);
FI;
DEST[MAX_VL-1:64] (Unmodified)

```

## Intel C/C++ Compiler Intrinsic Equivalent

```

VCVTSI2SD __m128d _mm_cvti32_sd(__m128d s, int a);
VCVTSI2SD __m128d _mm_cvt_roundi32_sd(__m128d s, int a, int r);
VCVTSI2SD __m128d _mm_cvti64_sd(__m128d s, __int64 a);
VCVTSI2SD __m128d _mm_cvt_roundi64_sd(__m128d s, __int64 a, int r);
CVTSI2SD __m128d _mm_cvtsi64_sd(__m128d s, __int64 a);
CVTSI2SD __m128d _mm_cvtsi32_sd(__m128d a, int b)

```

## SIMD Floating-Point Exceptions

Precision

## Other Exceptions

VEX-encoded instructions, see Exceptions Type 3 if W1, else Type 5.

EVEX-encoded instructions, see Exceptions Type E3NF if W1, else Type E10NF.

## CVTSS2SS—Convert Doubleword Integer to Scalar Single-Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 2A /r CVTSS2SS xmm1, r/m32	RM	V/V	SSE	Convert one signed doubleword integer from r/m32 to one single-precision floating-point value in xmm1.
F3 REX.W 0F 2A /r CVTSS2SS xmm1, r/m64	RM	V/N.E.	SSE	Convert one signed quadword integer from r/m64 to one single-precision floating-point value in xmm1.
VEX.NDS.128.F3.0F.W0 2A /r VCVTSS2SS xmm1, xmm2, r/m32	RVM	V/V	AVX	Convert one signed doubleword integer from r/m32 to one single-precision floating-point value in xmm1.
VEX.NDS.128.F3.0F.W1 2A /r VCVTSS2SS xmm1, xmm2, r/m64	RVM	V/N.E. <sup>1</sup>	AVX	Convert one signed quadword integer from r/m64 to one single-precision floating-point value in xmm1.
EVEX.NDS.LIG.F3.0F.W0 2A /r VCVTSS2SS xmm1, xmm2, r/m32{er}	T1S	V/V	AVX512F	Convert one signed doubleword integer from r/m32 to one single-precision floating-point value in xmm1.
EVEX.NDS.LIG.F3.0F.W1 2A /r VCVTSS2SS xmm1, xmm2, r/m64{er}	T1S	V/N.E. <sup>1</sup>	AVX512F	Convert one signed quadword integer from r/m64 to one single-precision floating-point value in xmm1.

### NOTES:

1. VEX.W1/EVEX.W1 in non-64 bit is ignored; the instructions behaves as if the W0 version is used.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
T1S	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

### Description

Converts a signed doubleword integer (or signed quadword integer if operand size is 64 bits) in the “convert-from” source operand to a single-precision floating-point value in the destination operand (first operand). The “convert-from” source operand can be a general-purpose register or a memory location. The destination operand is an XMM register. The result is stored in the low doubleword of the destination operand, and the upper three doublewords are left unchanged. When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits.

128-bit Legacy SSE version: In 64-bit mode, Use of the REX.W prefix promotes the instruction to use 64-bit input value. The “convert-from” source operand (the second operand) is a general-purpose register or memory location. Bits (MAX\_VL-1:32) of the corresponding destination register remain unchanged.

VEX.128 and EVEX encoded versions: The “convert-from” source operand (the third operand) can be a general-purpose register or a memory location. The first source and destination operands are XMM registers. Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (MAX\_VL-1:128) of the destination register are zeroed.

EVEX encoded version: the converted result is written to the low doubleword element of the destination under the writemask.

Software should ensure VCVTSS2SS is encoded with VEX.L=0. Encoding VCVTSS2SS with VEX.L=1 may encounter unpredictable behavior across different processor generations.

## Operation

### VCVTSI2SS (EVEX encoded version)

```

IF (SRC2 *is register*) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
IF 64-Bit Mode And OperandSize = 64
    THEN
        DEST[31:0] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[63:0]);
    ELSE
        DEST[31:0] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[31:0]);
FI;
DEST[127:32] ← SRC1[127:32]
DEST[MAX_VL-1:128] ← 0

```

### VCVTSI2SS (VEX.128 encoded version)

```

IF 64-Bit Mode And OperandSize = 64
    THEN
        DEST[31:0] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[63:0]);
    ELSE
        DEST[31:0] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[31:0]);
FI;
DEST[127:32] ← SRC1[127:32]
DEST[MAX_VL-1:128] ← 0

```

### CVTSI2SS (128-bit Legacy SSE version)

```

IF 64-Bit Mode And OperandSize = 64
    THEN
        DEST[31:0] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[63:0]);
    ELSE
        DEST[31:0] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[31:0]);
FI;
DEST[MAX_VL-1:32] (Unmodified)

```

## Intel C/C++ Compiler Intrinsic Equivalent

```

VCVTSI2SS __m128 _mm_cvtsi32_ss(__m128 s, int a);
VCVTSI2SS __m128 _mm_cvt_roundi32_ss(__m128 s, int a, int r);
VCVTSI2SS __m128 _mm_cvtsi64_ss(__m128 s, __int64 a);
VCVTSI2SS __m128 _mm_cvt_roundi64_ss(__m128 s, __int64 a, int r);
CVTSI2SS __m128 _mm_cvtsi64_ss(__m128 s, __int64 a);
CVTSI2SS __m128 _mm_cvtsi32_ss(__m128 a, int b);

```

## SIMD Floating-Point Exceptions

Precision

## Other Exceptions

VEX-encoded instructions, see Exceptions Type 3.  
 EVEX-encoded instructions, see Exceptions Type E3NF.

## CVTSS2SD—Convert Scalar Single-Precision Floating-Point Value to Scalar Double-Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 5A /r CVTSS2SD xmm1, xmm2/m32	RM	V/V	SSE2	Convert one single-precision floating-point value in xmm2/m32 to one double-precision floating-point value in xmm1.
VEX.NDS.128.F3.0F.WIG 5A /r VCVTSS2SD xmm1, xmm2, xmm3/m32	RVM	V/V	AVX	Convert one single-precision floating-point value in xmm3/m32 to one double-precision floating-point value and merge with high bits of xmm2.
EVEX.NDS.LIG.F3.0F.W0 5A /r VCVTSS2SD xmm1 {k1}{z}, xmm2, xmm3/m32{sae}	T1S	V/V	AVX512F	Convert one single-precision floating-point value in xmm3/m32 to one double-precision floating-point value and merge with high bits of xmm2 under writemask k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
T1S	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

### Description

Converts a single-precision floating-point value in the “convert-from” source operand to a double-precision floating-point value in the destination operand. When the “convert-from” source operand is an XMM register, the single-precision floating-point value is contained in the low doubleword of the register. The result is stored in the low quadword of the destination operand.

128-bit Legacy SSE version: The “convert-from” source operand (the second operand) is an XMM register or memory location. Bits (MAX\_VL-1:64) of the corresponding destination register remain unchanged. The destination operand is an XMM register.

VEX.128 and EVEX encoded versions: The “convert-from” source operand (the third operand) can be an XMM register or a 32-bit memory location. The first source and destination operands are XMM registers. Bits (127:64) of the XMM register destination are copied from the corresponding bits in the first source operand. Bits (MAX\_VL-1:128) of the destination register are zeroed.

Software should ensure VCVTSS2SD is encoded with VEX.L=0. Encoding VCVTSS2SD with VEX.L=1 may encounter unpredictable behavior across different processor generations.

### Operation

#### VCVTSS2SD (EVEX encoded version)

IF k1[0] or \*no writemask\*

THEN DEST[63:0] ← Convert\_Single\_Precision\_To\_Double\_Precision\_Floating\_Point(SRC2[31:0]);

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[63:0] remains unchanged\*

ELSE ; zeroing-masking

THEN DEST[63:0] = 0

FI;

FI;

DEST[127:64] ← SRC1[127:64]

DEST[MAX\_VL-1:128] ← 0



**VCVTSS2SD (VEX.128 encoded version)**

DEST[63:0] ← Convert\_Single\_Precision\_To\_Double\_Precision\_Floating\_Point(SRC2[31:0])  
 DEST[127:64] ← SRC1[127:64]  
 DEST[MAX\_VL-1:128] ← 0

**CVTSS2SD (128-bit Legacy SSE version)**

DEST[63:0] ← Convert\_Single\_Precision\_To\_Double\_Precision\_Floating\_Point(SRC[31:0]);  
 DEST[MAX\_VL-1:64] (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**

VCVTSS2SD \_\_m128d \_\_mm\_cvt\_roundss\_sd(\_\_m128d a, \_\_m128 b, int r);  
 VCVTSS2SD \_\_m128d \_\_mm\_mask\_cvt\_roundss\_sd(\_\_m128d s, \_\_mmask8 m, \_\_m128d a, \_\_m128 b, int r);  
 VCVTSS2SD \_\_m128d \_\_mm\_maskz\_cvt\_roundss\_sd(\_\_mmask8 k, \_\_m128d a, \_\_m128 a, int r);  
 VCVTSS2SD \_\_m128d \_\_mm\_mask\_cvtss\_sd(\_\_m128d s, \_\_mmask8 m, \_\_m128d a, \_\_m128 b);  
 VCVTSS2SD \_\_m128d \_\_mm\_maskz\_cvtss\_sd(\_\_mmask8 m, \_\_m128d a, \_\_m128 b);  
 CVTSS2SD \_\_m128d \_\_mm\_cvtss\_sd(\_\_m128d a, \_\_m128 a);

**SIMD Floating-Point Exceptions**

Invalid, Denormal

**Other Exceptions**

VEX-encoded instructions, see Exceptions Type 3.

EVEX-encoded instructions, see Exceptions Type E3.

## CVTSS2SI—Convert Scalar Single-Precision Floating-Point Value to Doubleword Integer

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 2D /r CVTSS2SI r32, xmm1/m32	RM	V/V	SSE	Convert one single-precision floating-point value from xmm1/m32 to one signed doubleword integer in r32.
F3 REX.W 0F 2D /r CVTSS2SI r64, xmm1/m32	RM	V/N.E.	SSE	Convert one single-precision floating-point value from xmm1/m32 to one signed quadword integer in r64.
VEX.128.F3.0F.W0 2D /r VCVTSS2SI r32, xmm1/m32	RM	V/V	AVX	Convert one single-precision floating-point value from xmm1/m32 to one signed doubleword integer in r32.
VEX.128.F3.0F.W1 2D /r VCVTSS2SI r64, xmm1/m32	RM	V/N.E. <sup>1</sup>	AVX	Convert one single-precision floating-point value from xmm1/m32 to one signed quadword integer in r64.
EVEX.LIG.F3.0F.W0 2D /r VCVTSS2SI r32, xmm1/m32{er}	T1F	V/V	AVX512F	Convert one single-precision floating-point value from xmm1/m32 to one signed doubleword integer in r32.
EVEX.LIG.F3.0F.W1 2D /r VCVTSS2SI r64, xmm1/m32{er}	T1F	V/N.E. <sup>1</sup>	AVX512F	Convert one single-precision floating-point value from xmm1/m32 to one signed quadword integer in r64.

### NOTES:

1. VEX.W1/EVEX.W1 in non-64 bit is ignored; the instructions behaves as if the W0 version is used.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
T1F	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Converts a single-precision floating-point value in the source operand (the second operand) to a signed doubleword integer (or signed quadword integer if operand size is 64 bits) in the destination operand (the first operand). 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 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, the indefinite integer value ( $2^{w-1}$ , where  $w$  represents the number of bits in the destination format) is returned.

Legacy SSE instructions: In 64-bit mode, Use of the REX.W prefix promotes the instruction to produce 64-bit data. See the summary chart at the beginning of this section for encoding data and limits.

VEX.W1 and EVEX.W1 versions: promotes the instruction to produce 64-bit data in 64-bit mode.

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

Software should ensure VCVTSS2SI is encoded with VEX.L=0. Encoding VCVTSS2SI with VEX.L=1 may encounter unpredictable behavior across different processor generations.

## Operation

### VCVTSS2SI (EVEX encoded version)

IF (SRC \*is register\*) AND (EVEX.b = 1)

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

IF 64-bit Mode and OperandSize = 64

THEN

DEST[63:0] ← Convert\_Single\_Precision\_Floating\_Point\_To\_Integer(SRC[31:0]);

ELSE

DEST[31:0] ← Convert\_Single\_Precision\_Floating\_Point\_To\_Integer(SRC[31:0]);

FI;

### (V)CVTSS2SI (Legacy and VEX.128 encoded version)

IF 64-bit Mode and OperandSize = 64

THEN

DEST[63:0] ← Convert\_Single\_Precision\_Floating\_Point\_To\_Integer(SRC[31:0]);

ELSE

DEST[31:0] ← Convert\_Single\_Precision\_Floating\_Point\_To\_Integer(SRC[31:0]);

FI;

## Intel C/C++ Compiler Intrinsic Equivalent

VCVTSS2SI int \_\_mm\_cvtss\_i32( \_\_m128 a);

VCVTSS2SI int \_\_mm\_cvt\_roundss\_i32( \_\_m128 a, int r);

VCVTSS2SI \_\_int64 \_\_mm\_cvtss\_i64( \_\_m128 a);

VCVTSS2SI \_\_int64 \_\_mm\_cvt\_roundss\_i64( \_\_m128 a, int r);

## SIMD Floating-Point Exceptions

Invalid, Precision

## Other Exceptions

VEX-encoded instructions, see Exceptions Type 3; additionally

#UD If VEX.vvvv != 1111B.

EVEX-encoded instructions, see Exceptions Type E3NF.

## CVTTPD2DQ—Convert with Truncation Packed Double-Precision Floating-Point Values to Packed Doubleword Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F E6 /r CVTTPD2DQ xmm1, xmm2/m128	RM	V/V	SSE2	Convert two packed double-precision floating-point values in xmm2/mem to two signed doubleword integers in xmm1 using truncation.
VEX.128.66.0F.WIG E6 /r VCVTPD2DQ xmm1, xmm2/m128	RM	V/V	AVX	Convert two packed double-precision floating-point values in xmm2/mem to two signed doubleword integers in xmm1 using truncation.
VEX.256.66.0F.WIG E6 /r VCVTPD2DQ xmm1, ymm2/m256	RM	V/V	AVX	Convert four packed double-precision floating-point values in ymm2/mem to four signed doubleword integers in xmm1 using truncation.
EVEX.128.66.0F.W1 E6 /r VCVTPD2DQ xmm1 {k1}{z}, xmm2/m128/m64bcst	FV	V/V	AVX512VL AVX512F	Convert two packed double-precision floating-point values in xmm2/m128/m64bcst to two signed doubleword integers in xmm1 using truncation subject to writemask k1.
EVEX.256.66.0F.W1 E6 /r VCVTPD2DQ xmm1 {k1}{z}, ymm2/m256/m64bcst	FV	V/V	AVX512VL AVX512F	Convert four packed double-precision floating-point values in ymm2/m256/m64bcst to four signed doubleword integers in xmm1 using truncation subject to writemask k1.
EVEX.512.66.0F.W1 E6 /r VCVTPD2DQ ymm1 {k1}{z}, zmm2/m512/m64bcst{sae}	FV	V/V	AVX512F	Convert eight packed double-precision floating-point values in zmm2/m512/m64bcst to eight signed doubleword integers in ymm1 using truncation subject to writemask k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
FV	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Converts two, four or eight packed double-precision floating-point values in the source operand (second operand) to two, four or eight packed signed doubleword integers in the destination operand (first operand).

When a conversion is inexact, a truncated (round toward zero) value is returned. If a converted result is larger than the maximum signed doubleword integer, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value (80000000H) is returned.

EVEX encoded versions: 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 64-bit memory location. The destination operand is a YMM/XMM/XMM (low 64 bits) register conditionally updated with writemask k1. The upper bits (MAX\_VL-1:256) of the corresponding destination are zeroed.

VEX.256 encoded version: The source operand is a YMM register or 256-bit memory location. The destination operand is an XMM register. The upper bits (MAX\_VL-1:128) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: The source operand is an XMM register or 128-bit memory location. The destination operand is a XMM register. The upper bits (MAX\_VL-1:64) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAX\_VL-1:128) of the corresponding ZMM register destination are unmodified.

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

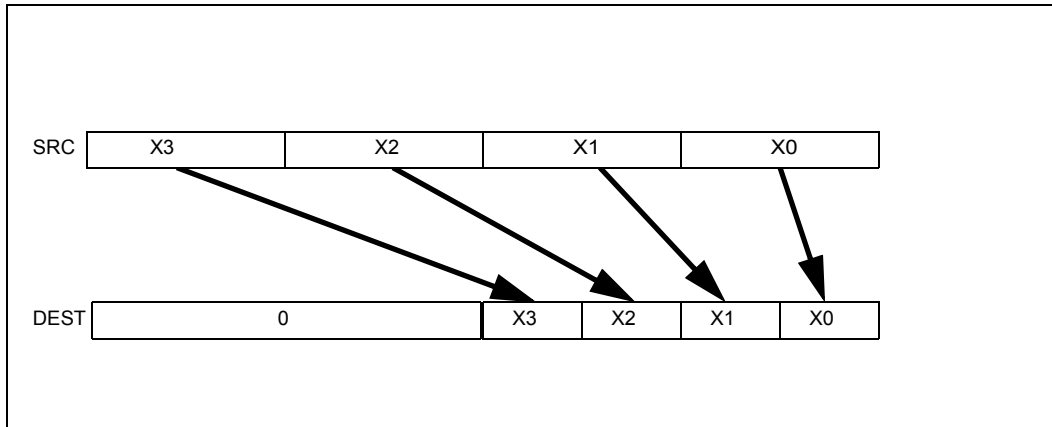


Figure 3-12. VCVTTPD2DQ (VEX.256 encoded version)

### Operation

**VCVTTPD2DQ (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 * 32
  k ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ←
      Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[k+63:k])
  ELSE
    IF *merging-masking*           ; merging-masking
      THEN *DEST[i+31:i] remains unchanged*
    ELSE                             ; zeroing-masking
      DEST[i+31:i] ← 0
    FI
  FI;
ENDFOR
DEST[MAX_VL-1:VL/2] ← 0

```

**VCVTPD2DQ (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_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[63:0])
        ELSE
          DEST[i+31:i] ←
          Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[k+63:k])
      FI;
    ELSE
      IF *merging-masking*           ; merging-masking
        THEN *DEST[i+31:i] remains unchanged*
      ELSE                             ; zeroing-masking
        DEST[i+31:i] ← 0
      FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL/2] ← 0

```

**VCVTPD2DQ (VEX.256 encoded version)**

```

DEST[31:0] ← Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[63:0])
DEST[63:32] ← Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[127:64])
DEST[95:64] ← Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[191:128])
DEST[127:96] ← Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[255:192])
DEST[MAX_VL-1:128] ← 0

```

**VCVTPD2DQ (VEX.128 encoded version)**

```

DEST[31:0] ← Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[63:0])
DEST[63:32] ← Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[127:64])
DEST[MAX_VL-1:64] ← 0

```

**CVTTPD2DQ (128-bit Legacy SSE version)**

```

DEST[31:0] ← Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[63:0])
DEST[63:32] ← Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[127:64])
DEST[127:64] ← 0
DEST[MAX_VL-1:128] (unmodified)

```

### Intel C/C++ Compiler Intrinsic Equivalent

```

VCVTPD2DQ __m256i __mm512_cvttpd_epi32( __m512d a);
VCVTPD2DQ __m256i __mm512_mask_cvttpd_epi32( __m256i s, __mmask8 k, __m512d a);
VCVTPD2DQ __m256i __mm512_maskz_cvttpd_epi32( __mmask8 k, __m512d a);
VCVTPD2DQ __m256i __mm512_cvtt_roundpd_epi32( __m512d a, int sae);
VCVTPD2DQ __m256i __mm512_mask_cvtt_roundpd_epi32( __m256i s, __mmask8 k, __m512d a, int sae);
VCVTPD2DQ __m256i __mm512_maskz_cvtt_roundpd_epi32( __mmask8 k, __m512d a, int sae);
VCVTPD2DQ __m128i __mm256_mask_cvttpd_epi32( __m128i s, __mmask8 k, __m256d a);
VCVTPD2DQ __m128i __mm256_maskz_cvttpd_epi32( __mmask8 k, __m256d a);
VCVTPD2DQ __m128i __mm_mask_cvttpd_epi32( __m128i s, __mmask8 k, __m128d a);
VCVTPD2DQ __m128i __mm_maskz_cvttpd_epi32( __mmask8 k, __m128d a);
VCVTPD2DQ __m128i __mm256_cvttpd_epi32( __m256d src);
CVTTPD2DQ __m128i __mm_cvttpd_epi32( __m128d src);

```

### SIMD Floating-Point Exceptions

Invalid, Precision

### Other Exceptions

VEX-encoded instructions, see Exceptions Type 2;

EVEX-encoded instructions, see Exceptions Type E2.

#UD If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

## CVTTTPS2DQ—Convert with Truncation Packed Single-Precision Floating-Point Values to Packed Signed Doubleword Integer Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 5B /r CVTTTPS2DQ xmm1, xmm2/m128	RM	V/V	SSE2	Convert four packed single-precision floating-point values from xmm2/mem to four packed signed doubleword values in xmm1 using truncation.
VEX.128.F3.0F.WIG 5B /r VCVTTTPS2DQ xmm1, xmm2/m128	RM	V/V	AVX	Convert four packed single-precision floating-point values from xmm2/mem to four packed signed doubleword values in xmm1 using truncation.
VEX.256.F3.0F.WIG 5B /r VCVTTTPS2DQ ymm1, ymm2/m256	RM	V/V	AVX	Convert eight packed single-precision floating-point values from ymm2/mem to eight packed signed doubleword values in ymm1 using truncation.
EVEX.128.F3.0F.W0 5B /r VCVTTTPS2DQ xmm1 {k1}{z}, xmm2/m128/m32bcst	FV	V/V	AVX512VL AVX512F	Convert four packed single precision floating-point values from xmm2/m128/m32bcst to four packed signed doubleword values in xmm1 using truncation subject to writemask k1.
EVEX.256.F3.0F.W0 5B /r VCVTTTPS2DQ ymm1 {k1}{z}, ymm2/m256/m32bcst	FV	V/V	AVX512VL AVX512F	Convert eight packed single precision floating-point values from ymm2/m256/m32bcst to eight packed signed doubleword values in ymm1 using truncation subject to writemask k1.
EVEX.512.F3.0F.W0 5B /r VCVTTTPS2DQ zmm1 {k1}{z}, zmm2/m512/m32bcst {sae}	FV	V/V	AVX512F	Convert sixteen packed single-precision floating-point values from zmm2/m512/m32bcst to sixteen packed signed doubleword values in zmm1 using truncation subject to writemask k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
FV	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Converts four, eight or sixteen packed single-precision floating-point values in the source operand to four, eight or sixteen signed doubleword integers in the destination operand.

When a conversion is inexact, a truncated (round toward zero) value is returned. If a converted result is larger than the maximum signed doubleword integer, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value (80000000H) is returned.

EVEX encoded versions: 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 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

VEX.256 encoded version: The source operand is a YMM register or 256-bit memory location. The destination operand is a YMM register. The upper bits (MAX\_VL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: The source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAX\_VL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAX\_VL-1:128) of the corresponding ZMM register destination are unmodified.

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



**Operation****VCVTTPS2DQ (EVEX encoded versions) when src operand is a register**

(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] ←
      Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[i+31:i])
    ELSE
      IF *merging-masking*           ; merging-masking
        THEN *DEST[i+31:i] remains unchanged*
      ELSE                             ; zeroing-masking
        DEST[i+31:i] ← 0
      FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VCVTTPS2DQ (EVEX encoded versions) when src operand is a memory source**

(KL, VL) = (4, 128), (8, 256), (16, 512)

```

FOR j ← 0 TO 15
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1)
        THEN
          DEST[i+31:i] ←
            Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[31:0])
        ELSE
          DEST[i+31:i] ←
            Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[i+31:i])
        FI;
      ELSE
        IF *merging-masking*           ; merging-masking
          THEN *DEST[i+31:i] remains unchanged*
        ELSE                             ; zeroing-masking
          DEST[i+31:i] ← 0
        FI
      FI;
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VCVTTPS2DQ (VEX.256 encoded version)**

```

DEST[31:0] ← Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[31:0])
DEST[63:32] ← Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[63:32])
DEST[95:64] ← Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[95:64])
DEST[127:96] ← Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[127:96])
DEST[159:128] ← Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[159:128])
DEST[191:160] ← Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[191:160])
DEST[223:192] ← Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[223:192])
DEST[255:224] ← Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[255:224])

```

**VCVTTPS2DQ (VEX.128 encoded version)**

DEST[31:0] ← Convert\_Single\_Precision\_Floating\_Point\_To\_Integer\_Truncate(SRC[31:0])  
 DEST[63:32] ← Convert\_Single\_Precision\_Floating\_Point\_To\_Integer\_Truncate(SRC[63:32])  
 DEST[95:64] ← Convert\_Single\_Precision\_Floating\_Point\_To\_Integer\_Truncate(SRC[95:64])  
 DEST[127:96] ← Convert\_Single\_Precision\_Floating\_Point\_To\_Integer\_Truncate(SRC[127:96])  
 DEST[MAX\_VL-1:128] ← 0

**CVTTPS2DQ (128-bit Legacy SSE version)**

DEST[31:0] ← Convert\_Single\_Precision\_Floating\_Point\_To\_Integer\_Truncate(SRC[31:0])  
 DEST[63:32] ← Convert\_Single\_Precision\_Floating\_Point\_To\_Integer\_Truncate(SRC[63:32])  
 DEST[95:64] ← Convert\_Single\_Precision\_Floating\_Point\_To\_Integer\_Truncate(SRC[95:64])  
 DEST[127:96] ← Convert\_Single\_Precision\_Floating\_Point\_To\_Integer\_Truncate(SRC[127:96])  
 DEST[MAX\_VL-1:128] (unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**

VCVTTPS2DQ \_\_m512i \_\_mm512\_cvttps\_epi32( \_\_m512 a);  
 VCVTTPS2DQ \_\_m512i \_\_mm512\_mask\_cvttps\_epi32( \_\_m512i s, \_\_mmask16 k, \_\_m512 a);  
 VCVTTPS2DQ \_\_m512i \_\_mm512\_maskz\_cvttps\_epi32( \_\_mmask16 k, \_\_m512 a);  
 VCVTTPS2DQ \_\_m512i \_\_mm512\_cvtt\_roundps\_epi32( \_\_m512 a, int sae);  
 VCVTTPS2DQ \_\_m512i \_\_mm512\_mask\_cvtt\_roundps\_epi32( \_\_m512i s, \_\_mmask16 k, \_\_m512 a, int sae);  
 VCVTTPS2DQ \_\_m512i \_\_mm512\_maskz\_cvtt\_roundps\_epi32( \_\_mmask16 k, \_\_m512 a, int sae);  
 VCVTTPS2DQ \_\_m256i \_\_mm256\_mask\_cvttps\_epi32( \_\_m256i s, \_\_mmask8 k, \_\_m256 a);  
 VCVTTPS2DQ \_\_m256i \_\_mm256\_maskz\_cvttps\_epi32( \_\_mmask8 k, \_\_m256 a);  
 VCVTTPS2DQ \_\_m128i \_\_mm128\_mask\_cvttps\_epi32( \_\_m128i s, \_\_mmask8 k, \_\_m128 a);  
 VCVTTPS2DQ \_\_m128i \_\_mm128\_maskz\_cvttps\_epi32( \_\_mmask8 k, \_\_m128 a);  
 VCVTTPS2DQ \_\_m256i \_\_mm256\_cvttps\_epi32( \_\_m256 a)  
 CVTTPS2DQ \_\_m128i \_\_mm128\_cvttps\_epi32( \_\_m128 a)

**SIMD Floating-Point Exceptions**

Invalid, Precision

**Other Exceptions**

VEX-encoded instructions, see Exceptions Type 2; additionally

EVEX-encoded instructions, see Exceptions Type E2.

#UD If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

## CVTTSD2SI—Convert with Truncation Scalar Double-Precision Floating-Point Value to Signed Integer

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 2C /r CVTTSD2SI r32, xmm1/m64	RM	V/V	SSE2	Convert one double-precision floating-point value from xmm1/m64 to one signed doubleword integer in r32 using truncation.
F2 REX.W 0F 2C /r CVTTSD2SI r64, xmm1/m64	RM	V/N.E.	SSE2	Convert one double-precision floating-point value from xmm1/m64 to one signed quadword integer in r64 using truncation.
VEX.128.F2.0F.W0 2C /r VCVTTSD2SI r32, xmm1/m64	RM	V/V	AVX	Convert one double-precision floating-point value from xmm1/m64 to one signed doubleword integer in r32 using truncation.
VEX.128.F2.0F.W1 2C /r VCVTTSD2SI r64, xmm1/m64	T1F	V/N.E. <sup>1</sup>	AVX	Convert one double-precision floating-point value from xmm1/m64 to one signed quadword integer in r64 using truncation.
EVEX.LIG.F2.0F.W0 2C /r VCVTTSD2SI r32, xmm1/m64{sae}	T1F	V/V	AVX512F	Convert one double-precision floating-point value from xmm1/m64 to one signed doubleword integer in r32 using truncation.
EVEX.LIG.F2.0F.W1 2C /r VCVTTSD2SI r64, xmm1/m64{sae}	T1F	V/N.E. <sup>1</sup>	AVX512F	Convert one double-precision floating-point value from xmm1/m64 to one signed quadword integer in r64 using truncation.

### NOTES:

- For this specific instruction, VEX.W/EVEX.W in non-64 bit is ignored; the instructions behaves as if the W0 version is used.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
T1F	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Converts a double-precision floating-point value in the source operand (the second operand) to a signed doubleword integer (or signed quadword integer if operand size is 64 bits) in the destination operand (the first operand). 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 value returned is rounded according to the rounding control bits in the MXCSR register.

If a converted result exceeds the range limits of signed doubleword integer (in non-64-bit modes or 64-bit mode with REX.W/VEX.W/EVEX.W=0), the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value (80000000H) is returned.

If a converted result exceeds the range limits of signed quadword integer (in 64-bit mode and REX.W/VEX.W/EVEX.W = 1), the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value (80000000\_00000000H) is returned.

Legacy SSE instructions: In 64-bit mode, Use of the REX.W prefix promotes the instruction to 64-bit operation. See the summary chart at the beginning of this section for encoding data and limits.

VEX.W1 and EVEX.W1 versions: promotes the instruction to produce 64-bit data in 64-bit mode.

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

Software should ensure VCVTTSD2SI is encoded with VEX.L=0. Encoding VCVTTSD2SI with VEX.L=1 may encounter unpredictable behavior across different processor generations.

### Operation

#### (V)VCVTTSD2SI (All versions)

IF 64-Bit Mode and OperandSize = 64

THEN

DEST[63:0] ← Convert\_Double\_Precision\_Floating\_Point\_To\_Integer\_Truncate(SRC[63:0]);

ELSE

DEST[31:0] ← Convert\_Double\_Precision\_Floating\_Point\_To\_Integer\_Truncate(SRC[63:0]);

FI;

### Intel C/C++ Compiler Intrinsic Equivalent

VCVTTSD2SI int \_\_mm\_cvttssd\_i32( \_\_m128d a);

VCVTTSD2SI int \_\_mm\_cvtt\_roundssd\_i32( \_\_m128d a, int sae);

VCVTTSD2SI \_\_int64 \_\_mm\_cvttssd\_i64( \_\_m128d a);

VCVTTSD2SI \_\_int64 \_\_mm\_cvtt\_roundssd\_i64( \_\_m128d a, int sae);

CVTTSD2SI int \_\_mm\_cvttssd\_si32( \_\_m128d a);

CVTTSD2SI \_\_int64 \_\_mm\_cvttssd\_si64( \_\_m128d a);

### SIMD Floating-Point Exceptions

Invalid, Precision

### Other Exceptions

VEX-encoded instructions, see Exceptions Type 3; additionally

#UD If VEX.vvvv != 1111B.

EVEX-encoded instructions, see Exceptions Type E3NF.

## CVTTSS2SI—Convert with Truncation Scalar Single-Precision Floating-Point Value to Integer

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 2C /r CVTTSS2SI r32, xmm1/m32	RM	V/V	SSE	Convert one single-precision floating-point value from xmm1/m32 to one signed doubleword integer in r32 using truncation.
F3 REX.W 0F 2C /r CVTTSS2SI r64, xmm1/m32	RM	V/N.E.	SSE	Convert one single-precision floating-point value from xmm1/m32 to one signed quadword integer in r64 using truncation.
VEX.128.F3.0F.W0 2C /r VCVTTSS2SI r32, xmm1/m32	RM	V/V	AVX	Convert one single-precision floating-point value from xmm1/m32 to one signed doubleword integer in r32 using truncation.
VEX.128.F3.0F.W1 2C /r VCVTTSS2SI r64, xmm1/m32	RM	V/N.E. <sup>1</sup>	AVX	Convert one single-precision floating-point value from xmm1/m32 to one signed quadword integer in r64 using truncation.
EVEX.LIG.F3.0F.W0 2C /r VCVTTSS2SI r32, xmm1/m32{sae}	T1F	V/V	AVX512F	Convert one single-precision floating-point value from xmm1/m32 to one signed doubleword integer in r32 using truncation.
EVEX.LIG.F3.0F.W1 2C /r VCVTTSS2SI r64, xmm1/m32{sae}	T1F	V/N.E. <sup>1</sup>	AVX512F	Convert one single-precision floating-point value from xmm1/m32 to one signed quadword integer in r64 using truncation.

### NOTES:

1. For this specific instruction, VEX.W/EVEX.W in non-64 bit is ignored; the instructions behaves as if the W0 version is used.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
T1F	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Converts a single-precision floating-point value in the source operand (the second operand) to a signed doubleword integer (or signed quadword integer if operand size is 64 bits) in the destination operand (the first operand). The source operand can be an XMM register or a 32-bit 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, a truncated (round toward zero) result is returned. If a converted result is larger than the maximum signed doubleword integer, the floating-point invalid exception is raised. If this exception is masked, the indefinite integer value (80000000H or 80000000\_00000000H if operand size is 64 bits) is returned.

Legacy SSE instructions: In 64-bit mode, Use of the REX.W prefix promotes the instruction to 64-bit operation. See the summary chart at the beginning of this section for encoding data and limits.

VEX.W1 and EVEX.W1 versions: promotes the instruction to produce 64-bit data in 64-bit mode.

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

Software should ensure VCVTTSS2SI is encoded with VEX.L=0. Encoding VCVTTSS2SI with VEX.L=1 may encounter unpredictable behavior across different processor generations.

**Operation****(V)CVTTSS2SI (All versions)**

IF 64-Bit Mode and OperandSize = 64

THEN

DEST[63:0] ← Convert\_Single\_Precision\_Floating\_Point\_To\_Integer\_Truncate(SRC[31:0]);

ELSE

DEST[31:0] ← Convert\_Single\_Precision\_Floating\_Point\_To\_Integer\_Truncate(SRC[31:0]);

FI;

**Intel C/C++ Compiler Intrinsic Equivalent**

VCVTTSS2SI int \_\_mm\_cvtss\_i32( \_\_m128 a);

VCVTTSS2SI int \_\_mm\_cvtt\_roundss\_i32( \_\_m128 a, int sae);

VCVTTSS2SI \_\_int64 \_\_mm\_cvtss\_i64( \_\_m128 a);

VCVTTSS2SI \_\_int64 \_\_mm\_cvtt\_roundss\_i64( \_\_m128 a, int sae);

CVTTSS2SI int \_\_mm\_cvtss\_si32( \_\_m128 a);

CVTTSS2SI \_\_int64 \_\_mm\_cvtss\_si64( \_\_m128 a);

**SIMD Floating-Point Exceptions**

Invalid, Precision

**Other Exceptions**

See Exceptions Type 3; additionally

#UD If VEX.vvvv != 1111B.

EVEX-encoded instructions, see Exceptions Type E3NF.

## DIVPD—Divide Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 5E /r DIVPD xmm1, xmm2/m128	RM	V/V	SSE2	Divide packed double-precision floating-point values in xmm1 by packed double-precision floating-point values in xmm2/mem.
VEX.NDS.128.66.0F.WIG 5E /r VDIVPD xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Divide packed double-precision floating-point values in xmm2 by packed double-precision floating-point values in xmm3/mem.
VEX.NDS.256.66.0F.WIG 5E /r VDIVPD ymm1, ymm2, ymm3/m256	RVM	V/V	AVX	Divide packed double-precision floating-point values in ymm2 by packed double-precision floating-point values in ymm3/mem.
EVEX.NDS.128.66.0F.W1 5E /r VDIVPD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	FV	V/V	AVX512VL AVX512F	Divide packed double-precision floating-point values in xmm2 by packed double-precision floating-point values in xmm3/m128/m64bcst and write results to xmm1 subject to writemask k1.
EVEX.NDS.256.66.0F.W1 5E /r VDIVPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	FV	V/V	AVX512VL AVX512F	Divide packed double-precision floating-point values in ymm2 by packed double-precision floating-point values in ymm3/m256/m64bcst and write results to ymm1 subject to writemask k1.
EVEX.NDS.512.66.0F.W1 5E /r VDIVPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	FV	V/V	AVX512F	Divide packed double-precision floating-point values in zmm2 by packed double-precision FP values in zmm3/m512/m64bcst and write results to zmm1 subject to writemask k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
FV	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

### Description

Performs a SIMD divide of the double-precision floating-point values in the first source operand by the floating-point values in the second source operand (the third operand). Results are written to the destination operand (the first operand).

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 64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

VEX.256 encoded version: The first source operand (the second operand) is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAX\_VL-1:256) of the corresponding destination are zeroed.

VEX.128 encoded version: The first source operand (the second operand) is a XMM register. The second source operand can be a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper bits (MAX\_VL-1:128) of the corresponding destination are zeroed.

128-bit Legacy SSE version: The second source operand (the second operand) can be an XMM register or a 128-bit memory location. The destination is the same as the first source operand. The upper bits (MAX\_VL-1:128) of the corresponding destination are unmodified.

**Operation****VDIVPD (EVEX encoded versions)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

IF (VL = 512) AND (EVEX.b = 1) AND SRC2 \*is a register\*

THEN

SET\_RM(EVEX.RC); ; refer to Table 2-4

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN

IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

THEN

DEST[i+63:i] ← SRC1[i+63:i] / SRC2[63:0]

ELSE

DEST[i+63:i] ← SRC1[i+63:i] / SRC2[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[MAX\_VL-1:VL] ← 0

**VDIVPD (VEX.256 encoded version)**

DEST[63:0] ← SRC1[63:0] / SRC2[63:0]

DEST[127:64] ← SRC1[127:64] / SRC2[127:64]

DEST[191:128] ← SRC1[191:128] / SRC2[191:128]

DEST[255:192] ← SRC1[255:192] / SRC2[255:192]

DEST[MAX\_VL-1:256] ← 0;

**VDIVPD (VEX.128 encoded version)**

DEST[63:0] ← SRC1[63:0] / SRC2[63:0]

DEST[127:64] ← SRC1[127:64] / SRC2[127:64]

DEST[MAX\_VL-1:128] ← 0;

**DIVPD (128-bit Legacy SSE version)**

DEST[63:0] ← SRC1[63:0] / SRC2[63:0]

DEST[127:64] ← SRC1[127:64] / SRC2[127:64]

DEST[MAX\_VL-1:128] (Unmodified)



### Intel C/C++ Compiler Intrinsic Equivalent

```

VDIVPD __m512d __mm512_div_pd( __m512d a, __m512d b);
VDIVPD __m512d __mm512_mask_div_pd(__m512d s, __mmask8 k, __m512d a, __m512d b);
VDIVPD __m512d __mm512_maskz_div_pd( __mmask8 k, __m512d a, __m512d b);
VDIVPD __m256d __mm256_mask_div_pd(__m256d s, __mmask8 k, __m256d a, __m256d b);
VDIVPD __m256d __mm256_maskz_div_pd( __mmask8 k, __m256d a, __m256d b);
VDIVPD __m128d __mm_mask_div_pd(__m128d s, __mmask8 k, __m128d a, __m128d b);
VDIVPD __m128d __mm_maskz_div_pd( __mmask8 k, __m128d a, __m128d b);
VDIVPD __m512d __mm512_div_round_pd( __m512d a, __m512d b, int);
VDIVPD __m512d __mm512_mask_div_round_pd(__m512d s, __mmask8 k, __m512d a, __m512d b, int);
VDIVPD __m512d __mm512_maskz_div_round_pd( __mmask8 k, __m512d a, __m512d b, int);
VDIVPD __m256d __mm256_div_pd( __m256d a, __m256d b);
DIVPD __m128d __mm_div_pd( __m128d a, __m128d b);

```

### SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Divide-by-Zero, Precision, Denormal

### Other Exceptions

VEX-encoded instructions, see Exceptions Type 2.

EVEX-encoded instructions, see Exceptions Type E2.

## DIVPS—Divide Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
OF 5E /r DIVPS xmm1, xmm2/m128	RM	V/V	SSE	Divide packed single-precision floating-point values in xmm1 by packed single-precision floating-point values in xmm2/mem.
VEX.NDS.128.OF.WIG 5E /r VDIVPS xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Divide packed single-precision floating-point values in xmm2 by packed single-precision floating-point values in xmm3/mem.
VEX.NDS.256.OF.WIG 5E /r VDIVPS ymm1, ymm2, ymm3/m256	RVM	V/V	AVX	Divide packed single-precision floating-point values in ymm2 by packed single-precision floating-point values in ymm3/mem.
EVEX.NDS.128.OF.W0 5E /r VDIVPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	FV	V/V	AVX512VL AVX512F	Divide packed single-precision floating-point values in xmm2 by packed single-precision floating-point values in xmm3/m128/m32bcst and write results to xmm1 subject to writemask k1.
EVEX.NDS.256.OF.W0 5E /r VDIVPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	FV	V/V	AVX512VL AVX512F	Divide packed single-precision floating-point values in ymm2 by packed single-precision floating-point values in ymm3/m256/m32bcst and write results to ymm1 subject to writemask k1.
EVEX.NDS.512.OF.W0 5E /r VDIVPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}	FV	V/V	AVX512F	Divide packed single-precision floating-point values in zmm2 by packed single-precision floating-point values in zmm3/m512/m32bcst and write results to zmm1 subject to writemask k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
FV	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

### Description

Performs a SIMD divide of the four, eight or sixteen packed single-precision floating-point values in the first source operand (the second operand) by the four, eight or sixteen packed single-precision floating-point values in the second source operand (the third operand). Results are written to the destination operand (the first operand).

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 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.

VEX.128 encoded version: The first source operand is a XMM register. The second source operand can be a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper bits (MAX\_VL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAX\_VL-1:128) of the corresponding ZMM register destination are unmodified.

## Operation

### VDIVPS (EVEX encoded versions)

(KL, VL) = (4, 128), (8, 256), (16, 512)

IF (VL = 512) AND (EVEX.b = 1) AND SRC2 \*is a register\*

```

    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN
                    DEST[i+31:i] ← SRC1[i+31:i] / SRC2[31:0]
                ELSE
                    DEST[i+31:i] ← SRC1[i+31:i] / SRC2[i+31:i]
            FI;
        ELSE
            IF *merging-masking*           ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
            ELSE                             ; zeroing-masking
                DEST[i+31:i] ← 0
            FI
        FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

### VDIVPS (VEX.256 encoded version)

DEST[31:0] ← SRC1[31:0] / SRC2[31:0]

DEST[63:32] ← SRC1[63:32] / SRC2[63:32]

DEST[95:64] ← SRC1[95:64] / SRC2[95:64]

DEST[127:96] ← SRC1[127:96] / SRC2[127:96]

DEST[159:128] ← SRC1[159:128] / SRC2[159:128]

DEST[191:160] ← SRC1[191:160] / SRC2[191:160]

DEST[223:192] ← SRC1[223:192] / SRC2[223:192]

DEST[255:224] ← SRC1[255:224] / SRC2[255:224].

DEST[MAX\_VL-1:256] ← 0;

### VDIVPS (VEX.128 encoded version)

DEST[31:0] ← SRC1[31:0] / SRC2[31:0]

DEST[63:32] ← SRC1[63:32] / SRC2[63:32]

DEST[95:64] ← SRC1[95:64] / SRC2[95:64]

DEST[127:96] ← SRC1[127:96] / SRC2[127:96]

DEST[MAX\_VL-1:128] ← 0

**DIVPS (128-bit Legacy SSE version)**

DEST[31:0] ← SRC1[31:0] / SRC2[31:0]

DEST[63:32] ← SRC1[63:32] / SRC2[63:32]

DEST[95:64] ← SRC1[95:64] / SRC2[95:64]

DEST[127:96] ← SRC1[127:96] / SRC2[127:96]

DEST[MAX\_VL-1:128] (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**

VDIVPS \_\_m512 \_\_mm512\_div\_ps(\_\_m512 a, \_\_m512 b);

VDIVPS \_\_m512 \_\_mm512\_mask\_div\_ps(\_\_m512 s, \_\_mmask16 k, \_\_m512 a, \_\_m512 b);

VDIVPS \_\_m512 \_\_mm512\_maskz\_div\_ps(\_\_mmask16 k, \_\_m512 a, \_\_m512 b);

VDIVPD \_\_m256d \_\_mm256\_mask\_div\_pd(\_\_m256d s, \_\_mmask8 k, \_\_m256d a, \_\_m256d b);

VDIVPD \_\_m256d \_\_mm256\_maskz\_div\_pd(\_\_mmask8 k, \_\_m256d a, \_\_m256d b);

VDIVPD \_\_m128d \_\_mm\_mask\_div\_pd(\_\_m128d s, \_\_mmask8 k, \_\_m128d a, \_\_m128d b);

VDIVPD \_\_m128d \_\_mm\_maskz\_div\_pd(\_\_mmask8 k, \_\_m128d a, \_\_m128d b);

VDIVPS \_\_m512 \_\_mm512\_div\_round\_ps(\_\_m512 a, \_\_m512 b, int);

VDIVPS \_\_m512 \_\_mm512\_mask\_div\_round\_ps(\_\_m512 s, \_\_mmask16 k, \_\_m512 a, \_\_m512 b, int);

VDIVPS \_\_m512 \_\_mm512\_maskz\_div\_round\_ps(\_\_mmask16 k, \_\_m512 a, \_\_m512 b, int);

VDIVPS \_\_m256 \_\_mm256\_div\_ps(\_\_m256 a, \_\_m256 b);

DIVPS \_\_m128 \_\_mm\_div\_ps(\_\_m128 a, \_\_m128 b);

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Divide-by-Zero, Precision, Denormal

**Other Exceptions**

VEX-encoded instructions, see Exceptions Type 2.

EVEX-encoded instructions, see Exceptions Type E2.

## DIVSD—Divide Scalar Double-Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 5E /r DIVSD xmm1, xmm2/m64	RM	V/V	SSE2	Divide low double-precision floating-point value in xmm1 by low double-precision floating-point value in xmm2/m64.
VEX.NDS.128.F2.0F.WIG 5E /r VDIVSD xmm1, xmm2, xmm3/m64	RVM	V/V	AVX	Divide low double-precision floating-point value in xmm2 by low double-precision floating-point value in xmm3/m64.
EVEX.NDS.LIG.F2.0F.W1 5E /r VDIVSD xmm1 {k1}{z}, xmm2, xmm3/m64{er}	T1S	V/V	AVX512F	Divide low double-precision floating-point value in xmm2 by low double-precision floating-point value in xmm3/m64.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
T1S	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

### Description

Divides the low double-precision floating-point value in the first source operand by the low double-precision floating-point value in the second source operand, and stores the double-precision floating-point result in the destination operand. The second source operand can be an XMM register or a 64-bit memory location. The first source and destination are XMM registers.

128-bit Legacy SSE version: The first source operand and the destination operand are the same. Bits (MAX\_VL-1:64) of the corresponding ZMM destination register remain unchanged.

VEX.128 encoded version: The first source operand is an xmm register encoded by VEX.vvvv. The quadword at bits 127:64 of the destination operand is copied from the corresponding quadword of the first source operand. Bits (MAX\_VL-1:128) of the destination register are zeroed.

EVEX.128 encoded version: The first source operand is an xmm register encoded by EVEX.vvvv. The quadword element of the destination operand at bits 127:64 are copied from the first source operand. Bits (MAX\_VL-1:128) of the destination register are zeroed.

EVEX version: The low quadword element of the destination is updated according to the writemask.

Software should ensure VDIVSD is encoded with VEX.L=0. Encoding VDIVSD with VEX.L=1 may encounter unpredictable behavior across different processor generations.

**Operation****VDIVSD (EVEX encoded version)**

```

IF (EVEX.b = 1) AND SRC2 *is a register*
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
    THEN DEST[63:0] ← SRC1[63:0] / SRC2[63:0]
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE ; zeroing-masking
                THEN DEST[63:0] ← 0
        FI;
FI;
DEST[127:64] ← SRC1[127:64]
DEST[MAX_VL-1:128] ← 0

```

**VDIVSD (VEX.128 encoded version)**

```

DEST[63:0] ← SRC1[63:0] / SRC2[63:0]
DEST[127:64] ← SRC1[127:64]
DEST[MAX_VL-1:128] ← 0

```

**DIVSD (128-bit Legacy SSE version)**

```

DEST[63:0] ← DEST[63:0] / SRC[63:0]
DEST[MAX_VL-1:64] (Unmodified)

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VDIVSD __m128d __mm_mask_div_sd(__m128d s, __mmask8 k, __m128d a, __m128d b);
VDIVSD __m128d __mm_maskz_div_sd(__mmask8 k, __m128d a, __m128d b);
VDIVSD __m128d __mm_div_round_sd(__m128d a, __m128d b, int);
VDIVSD __m128d __mm_mask_div_round_sd(__m128d s, __mmask8 k, __m128d a, __m128d b, int);
VDIVSD __m128d __mm_maskz_div_round_sd(__mmask8 k, __m128d a, __m128d b, int);
DIVSD __m128d __mm_div_sd(__m128d a, __m128d b);

```

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Divide-by-Zero, Precision, Denormal

**Other Exceptions**

VEX-encoded instructions, see Exceptions Type 3.

EVEX-encoded instructions, see Exceptions Type E3.

## DIVSS—Divide Scalar Single-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 5E /r DIVSS xmm1, xmm2/m32	RM	V/V	SSE	Divide low single-precision floating-point value in xmm1 by low single-precision floating-point value in xmm2/m32.
VEX.NDS.128.F3.0F.WIG 5E /r VDIVSS xmm1, xmm2, xmm3/m32	RVM	V/V	AVX	Divide low single-precision floating-point value in xmm2 by low single-precision floating-point value in xmm3/m32.
EVEX.NDS.LIG.F3.0F.W0 5E /r VDIVSS xmm1 {k1}{z}, xmm2, xmm3/m32{er}	T1S	V/V	AVX512F	Divide low single-precision floating-point value in xmm2 by low single-precision floating-point value in xmm3/m32.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
T1S	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

### Description

Divides the low single-precision floating-point value in the first source operand by the low single-precision floating-point value in the second source operand, and stores the single-precision floating-point result in the destination operand. The second source operand can be an XMM register or a 32-bit memory location.

128-bit Legacy SSE version: The first source operand and the destination operand are the same. Bits (MAX\_VL-1:32) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source operand is an xmm register encoded by VEX.vvvv. The three high-order doublewords of the destination operand are copied from the first source operand. Bits (MAX\_VL-1:128) of the destination register are zeroed.

EVEX.128 encoded version: The first source operand is an xmm register encoded by EVEX.vvvv. The doubleword elements of the destination operand at bits 127:32 are copied from the first source operand. Bits (MAX\_VL-1:128) of the destination register are zeroed.

EVEX version: The low doubleword element of the destination is updated according to the writemask.

Software should ensure VDIVSS is encoded with VEX.L=0. Encoding VDIVSS with VEX.L=1 may encounter unpredictable behavior across different processor generations.

**Operation****VDIVSS (EVEX encoded version)**

```

IF (EVEX.b = 1) AND SRC2 *is a register*
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
    THEN DEST[31:0] ← SRC1[31:0] / SRC2[31:0]
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE ; zeroing-masking
                THEN DEST[31:0] ← 0
        FI;
FI;
DEST[127:32] ← SRC1[127:32]
DEST[MAX_VL-1:128] ← 0

```

**VDIVSS (VEX.128 encoded version)**

```

DEST[31:0] ← SRC1[31:0] / SRC2[31:0]
DEST[127:32] ← SRC1[127:32]
DEST[MAX_VL-1:128] ← 0

```

**DIVSS (128-bit Legacy SSE version)**

```

DEST[31:0] ← DEST[31:0] / SRC[31:0]
DEST[MAX_VL-1:32] (Unmodified)

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VDIVSS __m128 __mm_mask_div_ss(__m128 s, __mmask8 k, __m128 a, __m128 b);
VDIVSS __m128 __mm_maskz_div_ss( __mmask8 k, __m128 a, __m128 b);
VDIVSS __m128 __mm_div_round_ss( __m128 a, __m128 b, int);
VDIVSS __m128 __mm_mask_div_round_ss(__m128 s, __mmask8 k, __m128 a, __m128 b, int);
VDIVSS __m128 __mm_maskz_div_round_ss( __mmask8 k, __m128 a, __m128 b, int);
DIVSS __m128 __mm_div_ss(__m128 a, __m128 b);

```

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Divide-by-Zero, Precision, Denormal

**Other Exceptions**

VEX-encoded instructions, see Exceptions Type 3.

EVEX-encoded instructions, see Exceptions Type E3.



## EXTRACTPS—Extract Packed Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 3A 17 /r ib EXTRACTPS reg/m32, xmm1, imm8	RMI	VV	SSE4_1	Extract one single-precision floating-point value from xmm1 at the offset specified by imm8 and store the result in reg or m32. Zero extend the results in 64-bit register if applicable.
VEX.128.66.0F3A.WIG 17 /r ib VEXTRACTPS reg/m32, xmm1, imm8	RMI	V/V	AVX	Extract one single-precision floating-point value from xmm1 at the offset specified by imm8 and store the result in reg or m32. Zero extend the results in 64-bit register if applicable.
EVEX.128.66.0F3A.WIG 17 /r ib VEXTRACTPS reg/m32, xmm1, imm8	T1S	V/V	AVX512F	Extract one single-precision floating-point value from xmm1 at the offset specified by imm8 and store the result in reg or m32. Zero extend the results in 64-bit register if applicable.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:r/m (w)	ModRM:reg (r)	Imm8	NA
T1S	ModRM:r/m (w)	ModRM:reg (r)	Imm8	NA

### Description

Extracts a single-precision floating-point value from the source operand (second operand) at the 32-bit offset specified from imm8. Immediate bits higher than the most significant offset for the vector length are ignored.

The extracted single-precision floating-point value is stored in the low 32-bits of the destination operand

In 64-bit mode, destination register operand has default operand size of 64 bits. The upper 32-bits of the register are filled with zero. REX.W is ignored.

VEX.128 and EVEX encoded version: When VEX.W1 or EVEX.W1 form is used in 64-bit mode with a general purpose register (GPR) as a destination operand, the packed single quantity is zero extended to 64 bits.

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

128-bit Legacy SSE version: When a REX.W prefix is used in 64-bit mode with a general purpose register (GPR) as a destination operand, the packed single quantity is zero extended to 64 bits.

The source register is an XMM register. Imm8[1:0] determine the starting DWORD offset from which to extract the 32-bit floating-point value.

If VEXTRACTPS is encoded with VEX.L= 1, an attempt to execute the instruction encoded with VEX.L= 1 will cause an #UD exception.

### Operation

#### VEXTRACTPS (EVEX and VEX.128 encoded version)

SRC\_OFFSET ← IMM8[1:0]

IF (64-Bit Mode and DEST is register)

DEST[31:0] ← (SRC[127:0] >> (SRC\_OFFSET\*32)) AND 0FFFFFFFh

DEST[63:32] ← 0

ELSE

DEST[31:0] ← (SRC[127:0] >> (SRC\_OFFSET\*32)) AND 0FFFFFFFh

FI

**EXTRACTPS (128-bit Legacy SSE version)**

SRC\_OFFSET ← IMM8[1:0]

IF (64-Bit Mode and DEST is register)

DEST[31:0] ← (SRC[127:0] &gt;&gt; (SRC\_OFFSET\*32)) AND 0FFFFFFFh

DEST[63:32] ← 0

ELSE

DEST[31:0] ← (SRC[127:0] &gt;&gt; (SRC\_OFFSET\*32)) AND 0FFFFFFFh

FI

**Intel C/C++ Compiler Intrinsic Equivalent**

EXTRACTPS int \_mm\_extract\_ps (\_\_m128 a, const int nidx);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

VEX-encoded instructions, see Exceptions Type 5; Additionally

EVEX-encoded instructions, see Exceptions Type E9NF.

#UD IF VEX.L = 0.

#UD If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

## FXAM—Examine Floating-Point

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
D9 E5	FXAM	Valid	Valid	Classify value or number in ST(0).

### Description

Examines the contents of the ST(0) register and sets the condition code flags C0, C2, and C3 in the FPU status word to indicate the class of value or number in the register (see the table below).

**Table 3-14. FXAM Results**

Class	C3	C2	C0
Unsupported	0	0	0
NaN	0	0	1
Normal finite number	0	1	0
Infinity	0	1	1
Zero	1	0	0
Empty	1	0	1
Denormal number	1	1	0

The C1 flag is set to the sign of the value in ST(0), regardless of whether the register is empty or full.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

### Operation

C1 ← sign bit of ST; (\* 0 for positive, 1 for negative \*)

CASE (class of value or number in ST(0)) OF

  Unsupported: C3, C2, C0 ← 000;

  NaN: C3, C2, C0 ← 001;

  Normal: C3, C2, C0 ← 010;

  Infinity: C3, C2, C0 ← 011;

  Zero: C3, C2, C0 ← 100;

  Empty: C3, C2, C0 ← 101;

  Denormal: C3, C2, C0 ← 110;

ESAC;

### FPU Flags Affected

C1 Sign of value in ST(0).

C0, C2, C3 See Table 3-42.

### Floating-Point Exceptions

None

### Protected Mode Exceptions

#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.

#MF If there is a pending x87 FPU exception.

#UD If the LOCK prefix is used.

### Real-Address Mode Exceptions

Same exceptions as in protected mode.

**Virtual-8086 Mode Exceptions**

Same exceptions as in protected mode.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

Same exceptions as in protected mode.

## IMUL—Signed Multiply

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
F6 /5	IMUL <i>r/m8</i> *	M	Valid	Valid	AX ← AL * <i>r/m</i> byte.
F7 /5	IMUL <i>r/m16</i>	M	Valid	Valid	DX:AX ← AX * <i>r/m</i> word.
F7 /5	IMUL <i>r/m32</i>	M	Valid	Valid	EDX:EAX ← EAX * <i>r/m32</i> .
REX.W + F7 /5	IMUL <i>r/m64</i>	M	Valid	N.E.	RDX:RAX ← RAX * <i>r/m64</i> .
OF AF /r	IMUL <i>r16, r/m16</i>	RM	Valid	Valid	word register ← word register * <i>r/m16</i> .
OF AF /r	IMUL <i>r32, r/m32</i>	RM	Valid	Valid	doubleword register ← doubleword register * <i>r/m32</i> .
REX.W + OF AF /r	IMUL <i>r64, r/m64</i>	RM	Valid	N.E.	Quadword register ← Quadword register * <i>r/m64</i> .
6B /r ib	IMUL <i>r16, r/m16, imm8</i>	RMI	Valid	Valid	word register ← <i>r/m16</i> * sign-extended immediate byte.
6B /r ib	IMUL <i>r32, r/m32, imm8</i>	RMI	Valid	Valid	doubleword register ← <i>r/m32</i> * sign-extended immediate byte.
REX.W + 6B /r ib	IMUL <i>r64, r/m64, imm8</i>	RMI	Valid	N.E.	Quadword register ← <i>r/m64</i> * sign-extended immediate byte.
69 /r iw	IMUL <i>r16, r/m16, imm16</i>	RMI	Valid	Valid	word register ← <i>r/m16</i> * immediate word.
69 /r id	IMUL <i>r32, r/m32, imm32</i>	RMI	Valid	Valid	doubleword register ← <i>r/m32</i> * immediate doubleword.
REX.W + 69 /r id	IMUL <i>r64, r/m64, imm32</i>	RMI	Valid	N.E.	Quadword register ← <i>r/m64</i> * immediate doubleword.

**NOTES:**

\* In 64-bit mode, *r/m8* can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m ( <i>r, w</i> )	NA	NA	NA
RM	ModRM:reg ( <i>r, w</i> )	ModRM:r/m ( <i>r</i> )	NA	NA
RMI	ModRM:reg ( <i>r, w</i> )	ModRM:r/m ( <i>r</i> )	imm8/16/32	NA

### Description

Performs a signed multiplication of two operands. This instruction has three forms, depending on the number of operands.

- **One-operand form** — This form is identical to that used by the MUL instruction. Here, the source operand (in a general-purpose register or memory location) is multiplied by the value in the AL, AX, EAX, or RAX register (depending on the operand size) and the product (twice the size of the input operand) is stored in the AX, DX:AX, EDX:EAX, or RDX:RAX registers, respectively.
- **Two-operand form** — With this form the destination operand (the first operand) is multiplied by the source operand (second operand). The destination operand is a general-purpose register and the source operand is an immediate value, a general-purpose register, or a memory location. The intermediate product (twice the size of the input operand) is truncated and stored in the destination operand location.
- **Three-operand form** — This form requires a destination operand (the first operand) and two source operands (the second and the third operands). Here, the first source operand (which can be a general-purpose register or a memory location) is multiplied by the second source operand (an immediate value). The intermediate product (twice the size of the first source operand) is truncated and stored in the destination operand (a general-purpose register).

When an immediate value is used as an operand, it is sign-extended to the length of the destination operand format.

The CF and OF flags are set when the signed integer value of the intermediate product differs from the sign extended operand-size-truncated product, otherwise the CF and OF flags are cleared.

The three forms of the IMUL instruction are similar in that the length of the product is calculated to twice the length of the operands. With the one-operand form, the product is stored exactly in the destination. With the two- and three- operand forms, however, the result is truncated to the length of the destination before it is stored in the destination register. Because of this truncation, the CF or OF flag should be tested to ensure that no significant bits are lost.

The two- and three-operand forms may also be used with unsigned operands because the lower half of the product is the same regardless if the operands are signed or unsigned. The CF and OF flags, however, cannot be used to determine if the upper half of the result is non-zero.

In 64-bit mode, the instruction's default operation size is 32 bits. Use of the REX.R prefix permits access to additional registers (R8-R15). Use of the REX.W prefix promotes operation to 64 bits. Use of REX.W modifies the three forms of the instruction as follows.

- **One-operand form** —The source operand (in a 64-bit general-purpose register or memory location) is multiplied by the value in the RAX register and the product is stored in the RDX:RAX registers.
- **Two-operand form** — The source operand is promoted to 64 bits if it is a register or a memory location. The destination operand is promoted to 64 bits.
- **Three-operand form** — The first source operand (either a register or a memory location) and destination operand are promoted to 64 bits. If the source operand is an immediate, it is sign extended to 64 bits.

## Operation

```

IF (NumberOfOperands = 1)
  THEN IF (OperandSize = 8)
    THEN
      TMP_XP ← AL * SRC (* Signed multiplication; TMP_XP is a signed integer at twice the width of the SRC *);
      AX ← TMP_XP[15:0];
      IF SignExtend(TMP_XP[7:0]) = TMP_XP
        THEN CF ← 0; OF ← 0;
        ELSE CF ← 1; OF ← 1; FI;
    ELSE IF OperandSize = 16
      THEN
        TMP_XP ← AX * SRC (* Signed multiplication; TMP_XP is a signed integer at twice the width of the SRC *)
        DX:AX ← TMP_XP[31:0];
        IF SignExtend(TMP_XP[15:0]) = TMP_XP
          THEN CF ← 0; OF ← 0;
          ELSE CF ← 1; OF ← 1; FI;
    ELSE IF OperandSize = 32
      THEN
        TMP_XP ← EAX * SRC (* Signed multiplication; TMP_XP is a signed integer at twice the width of the SRC*)
        EDX:EAX ← TMP_XP[63:0];
        IF SignExtend(TMP_XP[31:0]) = TMP_XP
          THEN CF ← 0; OF ← 0;
          ELSE CF ← 1; OF ← 1; FI;
    ELSE (* OperandSize = 64 *)
      TMP_XP ← RAX * SRC (* Signed multiplication; TMP_XP is a signed integer at twice the width of the SRC *)
      EDX:EAX ← TMP_XP[127:0];
      IF SignExtend(TMP_XP[63:0]) = TMP_XP
        THEN CF ← 0; OF ← 0;
        ELSE CF ← 1; OF ← 1; FI;
  FI;

```

```

FI;
ELSE IF (NumberOfOperands = 2)
  THEN
    TMP_XP ← DEST * SRC (* Signed multiplication; TMP_XP is a signed integer at twice the width of the SRC *)
    DEST ← TruncateToOperandSize(TMP_XP);
    IF SignExtend(DEST) ≠ TMP_XP
      THEN CF ← 1; OF ← 1;
      ELSE CF ← 0; OF ← 0; FI;
  ELSE (* NumberOfOperands = 3 *)
    TMP_XP ← SRC1 * SRC2 (* Signed multiplication; TMP_XP is a signed integer at twice the width of the SRC1 *)
    DEST ← TruncateToOperandSize(TMP_XP);
    IF SignExtend(DEST) ≠ TMP_XP
      THEN CF ← 1; OF ← 1;
      ELSE CF ← 0; OF ← 0; FI;
FI;
FI;

```

### Flags Affected

For the one operand form of the instruction, the CF and OF flags are set when significant bits are carried into the upper half of the result and cleared when the result fits exactly in the lower half of the result. For the two- and three-operand forms of the instruction, the CF and OF flags are set when the result must be truncated to fit in the destination operand size and cleared when the result fits exactly in the destination operand size. The SF, ZF, AF, and PF flags are undefined.

### Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register is used to access memory and it contains a NULL NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

### Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used.

### Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

- #SS(0) If a memory address referencing the SS segment is in a non-canonical form.
- #GP(0) If the memory address is in a non-canonical form.
- #PF(fault-code) If a page fault occurs.
- #AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
- #UD If the LOCK prefix is used.



## INSERTPS—Insert Scalar Single-Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 3A 21 /r ib INSERTPS xmm1, xmm2/m32, imm8	RMI	V/V	SSE4_1	Insert a single-precision floating-point value selected by imm8 from xmm2/m32 into xmm1 at the specified destination element specified by imm8 and zero out destination elements in xmm1 as indicated in imm8.
VEX.NDS.128.66.0F3A.WIG 21 /r ib VINSERTPS xmm1, xmm2, xmm3/m32, imm8	RVMI	V/V	AVX	Insert a single-precision floating-point value selected by imm8 from xmm3/m32 and merge with values in xmm2 at the specified destination element specified by imm8 and write out the result and zero out destination elements in xmm1 as indicated in imm8.
EVEX.NDS.128.66.0F3A.W0 21 /r ib VINSERTPS xmm1, xmm2, xmm3/m32, imm8	T1S	V/V	AVX512F	Insert a single-precision floating-point value selected by imm8 from xmm3/m32 and merge with values in xmm2 at the specified destination element specified by imm8 and write out the result and zero out destination elements in xmm1 as indicated in imm8.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (r, w)	ModRM:r/m (r)	Imm8	NA
RVMI	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	Imm8
T1S	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	Imm8

### Description

(register source form)

Select a single-precision floating-point element from second source as indicated by Count\_S bits of the immediate operand and destination operand it into the first source at the location indicated by the Count\_D bits of the immediate operand. Store in the destination and zero out destination elements based on the ZMask bits of the immediate operand.

(memory source form)

Load a floating-point element from a 32-bit memory location and destination operand it into the first source at the location indicated by the Count\_D bits of the immediate operand. Store in the destination and zero out destination elements based on the ZMask bits of the immediate operand.

128-bit Legacy SSE version: The first source register is an XMM register. The second source operand is either an XMM register or a 32-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAX\_VL-1:128) of the corresponding register destination are unmodified.

VEX.128 and EVEX encoded version: The destination and first source register is an XMM register. The second source operand is either an XMM register or a 32-bit memory location. The upper bits (MAX\_VL-1:128) of the corresponding register destination are zeroed.

If VINSERTPS is encoded with VEX.L= 1, an attempt to execute the instruction encoded with VEX.L= 1 will cause an #UD exception.

**Operation****VINSERTPS (VEX.128 and EVEX encoded version)**

```

IF (SRC = REG) THEN COUNT_S ← imm8[7:6]
  ELSE COUNT_S ← 0
COUNT_D ← imm8[5:4]
ZMASK ← imm8[3:0]
CASE (COUNT_S) OF
  0: TMP ← SRC2[31:0]
  1: TMP ← SRC2[63:32]
  2: TMP ← SRC2[95:64]
  3: TMP ← SRC2[127:96]
ESAC;
CASE (COUNT_D) OF
  0: TMP2[31:0] ← TMP
      TMP2[127:32] ← SRC1[127:32]
  1: TMP2[63:32] ← TMP
      TMP2[31:0] ← SRC1[31:0]
      TMP2[127:64] ← SRC1[127:64]
  2: TMP2[95:64] ← TMP
      TMP2[63:0] ← SRC1[63:0]
      TMP2[127:96] ← SRC1[127:96]
  3: TMP2[127:96] ← TMP
      TMP2[95:0] ← SRC1[95:0]
ESAC;

IF (ZMASK[0] = 1) THEN DEST[31:0] ← 00000000H
  ELSE DEST[31:0] ← TMP2[31:0]
IF (ZMASK[1] = 1) THEN DEST[63:32] ← 00000000H
  ELSE DEST[63:32] ← TMP2[63:32]
IF (ZMASK[2] = 1) THEN DEST[95:64] ← 00000000H
  ELSE DEST[95:64] ← TMP2[95:64]
IF (ZMASK[3] = 1) THEN DEST[127:96] ← 00000000H
  ELSE DEST[127:96] ← TMP2[127:96]
DEST[MAX_VL-1:128] ← 0

```

**INSERTPS (128-bit Legacy SSE version)**

```

IF (SRC = REG) THEN COUNT_S ← imm8[7:6]
  ELSE COUNT_S ← 0
COUNT_D ← imm8[5:4]
ZMASK ← imm8[3:0]
CASE (COUNT_S) OF
  0: TMP ← SRC[31:0]
  1: TMP ← SRC[63:32]
  2: TMP ← SRC[95:64]
  3: TMP ← SRC[127:96]
ESAC;

CASE (COUNT_D) OF
  0: TMP2[31:0] ← TMP
      TMP2[127:32] ← DEST[127:32]
  1: TMP2[63:32] ← TMP
      TMP2[31:0] ← DEST[31:0]
      TMP2[127:64] ← DEST[127:64]
  2: TMP2[95:64] ← TMP

```

```

    TMP2[63:0] ←DEST[63:0]
    TMP2[127:96] ←DEST[127:96]
3: TMP2[127:96] ←TMP
    TMP2[95:0] ←DEST[95:0]

```

ESAC;

```

IF (ZMASK[0] = 1) THEN DEST[31:0] ←00000000H
    ELSE DEST[31:0] ←TMP2[31:0]
IF (ZMASK[1] = 1) THEN DEST[63:32] ←00000000H
    ELSE DEST[63:32] ←TMP2[63:32]
IF (ZMASK[2] = 1) THEN DEST[95:64] ←00000000H
    ELSE DEST[95:64] ←TMP2[95:64]
IF (ZMASK[3] = 1) THEN DEST[127:96] ←00000000H
    ELSE DEST[127:96] ←TMP2[127:96]
DEST[MAX_VL-1:128] (Unmodified)

```

### Intel C/C++ Compiler Intrinsic Equivalent

```

VINSERTPS __m128 _mm_insert_ps(__m128 dst, __m128 src, const int nidx);
INSETRTPS __m128 _mm_insert_ps(__m128 dst, __m128 src, const int nidx);

```

### SIMD Floating-Point Exceptions

None

### Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 5; additionally

#UD If VEX.L = 0.

EVEX-encoded instruction, see Exceptions Type E9NF.

## INVLPG—Invalidate TLB Entries

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
OF 01/7	INVLPG <i>m</i>	M	Valid	Valid	Invalidate TLB entries for page containing <i>m</i> .

### NOTES:

\* See the IA-32 Architecture Compatibility section below.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r)	NA	NA	NA

### Description

Invalidates any translation lookaside buffer (TLB) entries specified with the source operand. The source operand is a memory address. The processor determines the page that contains that address and flushes all TLB entries for that page.<sup>1</sup>

The INVLPG instruction is a privileged instruction. When the processor is running in protected mode, the CPL must be 0 to execute this instruction.

The INVLPG instruction normally flushes TLB entries only for the specified page; however, in some cases, it may flush more entries, even the entire TLB. The instruction is guaranteed to invalidate only TLB entries associated with the current PCID. (If PCIDs are disabled — CR4.PCIDE = 0 — the current PCID is 000H.) The instruction also invalidates any global TLB entries for the specified page, regardless of PCID.

For more details on operations that flush the TLB, see “MOV—Move to/from Control Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2B* and Section 4.10.4.1, “Operations that Invalidate TLBs and Paging-Structure Caches,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

This instruction’s operation is the same in all non-64-bit modes. It also operates the same in 64-bit mode, except if the memory address is in non-canonical form. In this case, INVLPG is the same as a NOP.

### IA-32 Architecture Compatibility

The INVLPG instruction is implementation dependent, and its function may be implemented differently on different families of Intel 64 or IA-32 processors. This instruction is not supported on IA-32 processors earlier than the Intel486 processor.

### Operation

Invalidate(RelevantTLBEntries);  
Continue; (\* Continue execution \*)

### Flags Affected

None.

### Protected Mode Exceptions

#GP(0) If the current privilege level is not 0.  
#UD Operand is a register.  
If the LOCK prefix is used.

1. If the paging structures map the linear address using a page larger than 4 KBytes and there are multiple TLB entries for that page (see Section 4.10.2.3, “Details of TLB Use,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*), the instruction invalidates all of them.

**Real-Address Mode Exceptions**

#UD                   Operand is a register.  
                          If the LOCK prefix is used.

**Virtual-8086 Mode Exceptions**

#GP(0)                The INVLPG instruction cannot be executed at the virtual-8086 mode.

**64-Bit Mode Exceptions**

#GP(0)                If the current privilege level is not 0.  
#UD                    Operand is a register.  
                          If the LOCK prefix is used.

**KADDW/KADDB/KADDQ/KADD—ADD Two Masks**

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.L1.0F.W0 4A /r KADDW k1, k2, k3	RVR	V/V	AVX512DQ	Add 16 bits masks in k2 and k3 and place result in k1.
VEX.L1.66.0F.W0 4A /r KADDB k1, k2, k3	RVR	V/V	AVX512DQ	Add 8 bits masks in k2 and k3 and place result in k1.
VEX.L1.0F.W1 4A /r KADDQ k1, k2, k3	RVR	V/V	AVX512BW	Add 64 bits masks in k2 and k3 and place result in k1.
VEX.L1.66.0F.W1 4A /r KADD k1, k2, k3	RVR	V/V	AVX512BW	Add 32 bits masks in k2 and k3 and place result in k1.

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3
RVR	ModRM:reg (w)	VEX.1vsv (r)	ModRM:r/m (r, ModRM:[7:6] must be 11b)

**Description**

Adds the vector mask k2 and the vector mask k3, and writes the result into vector mask k1.

**Operation****KADDW**

DEST[15:0] ← SRC1[15:0] + SRC2[15:0]  
DEST[MAX\_KL-1:16] ← 0

**KADDB**

DEST[7:0] ← SRC1[7:0] + SRC2[7:0]  
DEST[MAX\_KL-1:8] ← 0

**KADDQ**

DEST[63:0] ← SRC1[63:0] + SRC2[63:0]  
DEST[MAX\_KL-1:64] ← 0

**KADD**

DEST[31:0] ← SRC1[31:0] + SRC2[31:0]  
DEST[MAX\_KL-1:32] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent****SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type K20.

## KANDW/KANDB/KANDQ/KANDD—Bitwise Logical AND Masks

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.NDS.L1.0F.W0 41 /r KANDW k1, k2, k3	RVR	V/V	AVX512F	Bitwise AND 16 bits masks k2 and k3 and place result in k1.
VEX.L1.66.0F.W0 41 /r KANDB k1, k2, k3	RVR	V/V	AVX512DQ	Bitwise AND 8 bits masks k2 and k3 and place result in k1.
VEX.L1.0F.W1 41 /r KANDQ k1, k2, k3	RVR	V/V	AVX512BW	Bitwise AND 64 bits masks k2 and k3 and place result in k1.
VEX.L1.66.0F.W1 41 /r KANDD k1, k2, k3	RVR	V/V	AVX512BW	Bitwise AND 32 bits masks k2 and k3 and place result in k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3
RVR	ModRM:reg (w)	VEX.1vvv (r)	ModRM:r/m (r, ModRM:[7:6] must be 11b)

### Description

Performs a bitwise AND between the vector mask k2 and the vector mask k3, and writes the result into vector mask k1.

### Operation

#### KANDW

DEST[15:0] ← SRC1[15:0] BITWISE AND SRC2[15:0]  
DEST[MAX\_KL-1:16] ← 0

#### KANDB

DEST[7:0] ← SRC1[7:0] BITWISE AND SRC2[7:0]  
DEST[MAX\_KL-1:8] ← 0

#### KANDQ

DEST[63:0] ← SRC1[63:0] BITWISE AND SRC2[63:0]  
DEST[MAX\_KL-1:64] ← 0

#### KANDD

DEST[31:0] ← SRC1[31:0] BITWISE AND SRC2[31:0]  
DEST[MAX\_KL-1:32] ← 0

### Intel C/C++ Compiler Intrinsic Equivalent

KANDW `__mmask16 _mm512_kand(__mmask16 a, __mmask16 b);`

### Flags Affected

None

### SIMD Floating-Point Exceptions

None

### Other Exceptions

See Exceptions Type K20.

**KANDNW/KANDNB/KANDNQ/KANDND—Bitwise Logical AND NOT Masks**

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.NDS.L1.0F.W0 42 /r KANDNW k1, k2, k3	RVR	V/V	AVX512F	Bitwise AND NOT 16 bits masks k2 and k3 and place result in k1.
VEX.L1.66.0F.W0 42 /r KANDNB k1, k2, k3	RVR	V/V	AVX512DQ	Bitwise AND NOT 8 bits masks k1 and k2 and place result in k1.
VEX.L1.0F.W1 42 /r KANDNQ k1, k2, k3	RVR	V/V	AVX512BW	Bitwise AND NOT 64 bits masks k2 and k3 and place result in k1.
VEX.L1.66.0F.W1 42 /r KANDND k1, k2, k3	RVR	V/V	AVX512BW	Bitwise AND NOT 32 bits masks k2 and k3 and place result in k1.

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3
RVR	ModRM:reg (w)	VEX.1vvv (r)	ModRM:r/m (r, ModRM:[7:6] must be 11b)

**Description**

Performs a bitwise AND NOT between the vector mask k2 and the vector mask k3, and writes the result into vector mask k1.

**Operation****KANDNW**

DEST[15:0] ← (BITWISE NOT SRC1[15:0]) BITWISE AND SRC2[15:0]  
DEST[MAX\_KL-1:16] ← 0

**KANDNB**

DEST[7:0] ← (BITWISE NOT SRC1[7:0]) BITWISE AND SRC2[7:0]  
DEST[MAX\_KL-1:8] ← 0

**KANDNQ**

DEST[63:0] ← (BITWISE NOT SRC1[63:0]) BITWISE AND SRC2[63:0]  
DEST[MAX\_KL-1:64] ← 0

**KANDND**

DEST[31:0] ← (BITWISE NOT SRC1[31:0]) BITWISE AND SRC2[31:0]  
DEST[MAX\_KL-1:32] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

KANDNW \_\_mmask16 \_\_mm512\_kandn(\_\_mmask16 a, \_\_mmask16 b);

**Flags Affected**

None

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type K20.



## KMOVW/KMOVB/KMOVQ/KMOVD—Move from and to Mask Registers

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.L0.0F.W0 90 /r KMOVW k1, k2/m16	RM	V/V	AVX512F	Move 16 bits mask from k2/m16 and store the result in k1.
VEX.L0.66.0F.W0 90 /r KMOVB k1, k2/m8	RM	V/V	AVX512DQ	Move 8 bits mask from k2/m8 and store the result in k1.
VEX.L0.0F.W1 90 /r KMOVQ k1, k2/m64	RM	V/V	AVX512BW	Move 64 bits mask from k2/m64 and store the result in k1.
VEX.L0.66.0F.W1 90 /r KMOVD k1, k2/m32	RM	V/V	AVX512BW	Move 32 bits mask from k2/m32 and store the result in k1.
VEX.L0.0F.W0 91 /r KMOVW m16, k1	MR	V/V	AVX512F	Move 16 bits mask from k1 and store the result in m16.
VEX.L0.66.0F.W0 91 /r KMOVB m8, k1	MR	V/V	AVX512DQ	Move 8 bits mask from k1 and store the result in m8.
VEX.L0.0F.W1 91 /r KMOVQ m64, k1	MR	V/V	AVX512BW	Move 64 bits mask from k1 and store the result in m64.
VEX.L0.66.0F.W1 91 /r KMOVD m32, k1	MR	V/V	AVX512BW	Move 32 bits mask from k1 and store the result in m32.
VEX.L0.0F.W0 92 /r KMOVW k1, r32	RR	V/V	AVX512F	Move 16 bits mask from r32 to k1.
VEX.L0.66.0F.W0 92 /r KMOVB k1, r32	RR	V/V	AVX512DQ	Move 8 bits mask from r32 to k1.
VEX.L0.F2.0F.W1 92 /r KMOVQ k1, r64	RR	V/I	AVX512BW	Move 64 bits mask from r64 to k1.
VEX.L0.F2.0F.W0 92 /r KMOVD k1, r32	RR	V/V	AVX512BW	Move 32 bits mask from r32 to k1.
VEX.L0.0F.W0 93 /r KMOVW r32, k1	RR	V/V	AVX512F	Move 16 bits mask from k1 to r32.
VEX.L0.66.0F.W0 93 /r KMOVB r32, k1	RR	V/V	AVX512DQ	Move 8 bits mask from k1 to r32.
VEX.L0.F2.0F.W1 93 /r KMOVQ r64, k1	RR	V/I	AVX512BW	Move 64 bits mask from k1 to r64.
VEX.L0.F2.0F.W0 93 /r KMOVD r32, k1	RR	V/V	AVX512BW	Move 32 bits mask from k1 to r32.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2
RM	ModRM:reg (w)	ModRM:r/m (r)
MR	ModRM:r/m (w, ModRM:[7:6] must not be 11b)	ModRM:reg (r)
RR	ModRM:reg (w)	ModRM:r/m (r, ModRM:[7:6] must be 11b)

### Description

Copies values from the source operand (second operand) to the destination operand (first operand). The source and destination operands can be mask registers, memory location or general purpose. The instruction cannot be used to transfer data between general purpose registers and or memory locations.

When moving to a mask register, the result is zero extended to MAX\_KL size (i.e., 64 bits currently). When moving to a general-purpose register (GPR), the result is zero-extended to the size of the destination. In 32-bit mode, the default GPR destination's size is 32 bits. In 64-bit mode, the default GPR destination's size is 64 bits. Note that REX.W cannot be used to modify the size of the general-purpose destination.

**Operation****KMOVW**

IF \*destination is a memory location\*

$$\text{DEST}[15:0] \leftarrow \text{SRC}[15:0]$$

IF \*destination is a mask register or a GPR \*

$$\text{DEST} \leftarrow \text{ZeroExtension}(\text{SRC}[15:0])$$
**KMOVB**

IF \*destination is a memory location\*

$$\text{DEST}[7:0] \leftarrow \text{SRC}[7:0]$$

IF \*destination is a mask register or a GPR \*

$$\text{DEST} \leftarrow \text{ZeroExtension}(\text{SRC}[7:0])$$
**KMOVQ**

IF \*destination is a memory location or a GPR\*

$$\text{DEST}[63:0] \leftarrow \text{SRC}[63:0]$$

IF \*destination is a mask register\*

$$\text{DEST} \leftarrow \text{ZeroExtension}(\text{SRC}[63:0])$$
**KMOVD**

IF \*destination is a memory location\*

$$\text{DEST}[31:0] \leftarrow \text{SRC}[31:0]$$

IF \*destination is a mask register or a GPR \*

$$\text{DEST} \leftarrow \text{ZeroExtension}(\text{SRC}[31:0])$$
**Intel C/C++ Compiler Intrinsic Equivalent**

KMOVW `__mmask16 _mm512_kmov(__mmask16 a);`

**Flags Affected**

None

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

Instructions with RR operand encoding See Exceptions Type K20.

Instructions with RM or MR operand encoding See Exceptions Type K21.

## KNOTW/KNOTB/KNOTQ/KNOTD—NOT Mask Register

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.L0.0F.W0 44 /r KNOTW k1, k2	RR	V/V	AVX512F	Bitwise NOT of 16 bits mask k2.
VEX.L0.66.0F.W0 44 /r KNOTB k1, k2	RR	V/V	AVX512DQ	Bitwise NOT of 8 bits mask k2.
VEX.L0.0F.W1 44 /r KNOTQ k1, k2	RR	V/V	AVX512BW	Bitwise NOT of 64 bits mask k2.
VEX.L0.66.0F.W1 44 /r KNOTD k1, k2	RR	V/V	AVX512BW	Bitwise NOT of 32 bits mask k2.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2
RR	ModRM:reg (w)	ModRM:r/m (r, ModRM:[7:6] must be 11b)

### Description

Performs a bitwise NOT of vector mask k2 and writes the result into vector mask k1.

### Operation

#### KNOTW

DEST[15:0] ← BITWISE NOT SRC[15:0]

DEST[MAX\_KL-1:16] ← 0

#### KNOTB

DEST[7:0] ← BITWISE NOT SRC[7:0]

DEST[MAX\_KL-1:8] ← 0

#### KNOTQ

DEST[63:0] ← BITWISE NOT SRC[63:0]

DEST[MAX\_KL-1:64] ← 0

#### KNOTD

DEST[31:0] ← BITWISE NOT SRC[31:0]

DEST[MAX\_KL-1:32] ← 0

### Intel C/C++ Compiler Intrinsic Equivalent

KNOTW `__mmask16 _mm512_knot(__mmask16 a);`

### Flags Affected

None

### SIMD Floating-Point Exceptions

None

### Other Exceptions

See Exceptions Type K20.

**KORW/KORB/KORQ/KORD—Bitwise Logical OR Masks**

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.NDS.L1.0F.W0 45 /r KORW k1, k2, k3	RVR	V/V	AVX512F	Bitwise OR 16 bits masks k2 and k3 and place result in k1.
VEX.L1.66.0F.W0 45 /r KORB k1, k2, k3	RVR	V/V	AVX512DQ	Bitwise OR 8 bits masks k2 and k3 and place result in k1.
VEX.L1.0F.W1 45 /r KORQ k1, k2, k3	RVR	V/V	AVX512BW	Bitwise OR 64 bits masks k2 and k3 and place result in k1.
VEX.L1.66.0F.W1 45 /r KORD k1, k2, k3	RVR	V/V	AVX512BW	Bitwise OR 32 bits masks k2 and k3 and place result in k1.

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3
RVR	ModRM:reg (w)	VEX.1vvv (r)	ModRM:r/m (r, ModRM:[7:6] must be 11b)

**Description**

Performs a bitwise OR between the vector mask k2 and the vector mask k3, and writes the result into vector mask k1 (three-operand form).

**Operation****KORW**

DEST[15:0] ← SRC1[15:0] BITWISE OR SRC2[15:0]  
DEST[MAX\_KL-1:16] ← 0

**KORB**

DEST[7:0] ← SRC1[7:0] BITWISE OR SRC2[7:0]  
DEST[MAX\_KL-1:8] ← 0

**KORQ**

DEST[63:0] ← SRC1[63:0] BITWISE OR SRC2[63:0]  
DEST[MAX\_KL-1:64] ← 0

**KORD**

DEST[31:0] ← SRC1[31:0] BITWISE OR SRC2[31:0]  
DEST[MAX\_KL-1:32] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

KORW `__mmask16 __mm512_kor(__mmask16 a, __mmask16 b);`

**Flags Affected**

None

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type K20.

## KORTESTW/KORTESTB/KORTESTQ/KORTESTD—OR Masks And Set Flags

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.L0.0F.W0 98 /r KORTESTW k1, k2	RR	V/V	AVX512F	Bitwise OR 16 bits masks k1 and k2 and update ZF and CF accordingly.
VEX.L0.66.0F.W0 98 /r KORTESTB k1, k2	RR	V/V	AVX512DQ	Bitwise OR 8 bits masks k1 and k2 and update ZF and CF accordingly.
VEX.L0.0F.W1 98 /r KORTESTQ k1, k2	RR	V/V	AVX512BW	Bitwise OR 64 bits masks k1 and k2 and update ZF and CF accordingly.
VEX.L0.66.0F.W1 98 /r KORTESTD k1, k2	RR	V/V	AVX512BW	Bitwise OR 32 bits masks k1 and k2 and update ZF and CF accordingly.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2
RR	ModRM:reg (w)	ModRM:r/m (r, ModRM:[7:6] must be 11b)

### Description

Performs a bitwise OR between the vector mask register k2, and the vector mask register k1, and sets CF and ZF based on the operation result.

ZF flag is set if both sources are 0x0. CF is set if, after the OR operation is done, the operation result is all 1's.

### Operation

#### KORTESTW

TMP[15:0] ← DEST[15:0] BITWISE OR SRC[15:0]

IF(TMP[15:0]=0)  
THEN ZF ← 1  
ELSE ZF ← 0

FI;

IF(TMP[15:0]=FFFFh)  
THEN CF ← 1  
ELSE CF ← 0

FI;

#### KORTESTB

TMP[7:0] ← DEST[7:0] BITWISE OR SRC[7:0]

IF(TMP[7:0]=0)  
THEN ZF ← 1  
ELSE ZF ← 0

FI;

IF(TMP[7:0]=FFh)  
THEN CF ← 1  
ELSE CF ← 0

FI;

**KORTESTQ**

TMP[63:0] ← DEST[63:0] BITWISE OR SRC[63:0]

IF(TMP[63:0]=0)

THEN ZF ← 1

ELSE ZF ← 0

FI;

IF(TMP[63:0]==FFFFFFFF\_FFFFFFFFh)

THEN CF ← 1

ELSE CF ← 0

FI;

**KORTESTD**

TMP[31:0] ← DEST[31:0] BITWISE OR SRC[31:0]

IF(TMP[31:0]=0)

THEN ZF ← 1

ELSE ZF ← 0

FI;

IF(TMP[31:0]=FFFFFFFFh)

THEN CF ← 1

ELSE CF ← 0

FI;

**Intel C/C++ Compiler Intrinsic Equivalent**

KORTESTW \_\_mmask16 \_mm512\_kortest[cz](\_\_mmask16 a, \_\_mmask16 b);

**Flags Affected**

The ZF flag is set if the result of OR-ing both sources is all 0s.

The CF flag is set if the result of OR-ing both sources is all 1s.

The OF, SF, AF, and PF flags are set to 0.

**Other Exceptions**

See Exceptions Type K20.

## KSHIFTLW/KSHIFTLB/KSHIFTLQ/KSHIFTLD—Shift Left Mask Registers

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.L0.66.0F3A.W1 32 /r KSHIFTLW k1, k2, imm8	RRI	V/V	AVX512F	Shift left 16 bits in k2 by immediate and write result in k1.
VEX.L0.66.0F3A.W0 32 /r KSHIFTLB k1, k2, imm8	RRI	V/V	AVX512DQ	Shift left 8 bits in k2 by immediate and write result in k1.
VEX.L0.66.0F3A.W1 33 /r KSHIFTLQ k1, k2, imm8	RRI	V/V	AVX512BW	Shift left 64 bits in k2 by immediate and write result in k1.
VEX.L0.66.0F3A.W0 33 /r KSHIFTLD k1, k2, imm8	RRI	V/V	AVX512BW	Shift left 32 bits in k2 by immediate and write result in k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3
RRI	ModRM:reg (w)	ModRM:r/m (r, ModRM:[7:6] must be 11b)	Imm8

### Description

Shifts 8/16/32/64 bits in the second operand (source operand) left by the count specified in immediate byte and place the least significant 8/16/32/64 bits of the result in the destination operand. The higher bits of the destination are zero-extended. The destination is set to zero if the count value is greater than 7 (for byte shift), 15 (for word shift), 31 (for doubleword shift) or 63 (for quadword shift).

### Operation

#### KSHIFTLW

```
COUNT ← imm8[7:0]
DEST[MAX_KL-1:0] ← 0
IF COUNT ≤ 15
    THEN DEST[15:0] ← SRC1[15:0] << COUNT;
FI;
```

#### KSHIFTLB

```
COUNT ← imm8[7:0]
DEST[MAX_KL-1:0] ← 0
IF COUNT ≤ 7
    THEN DEST[7:0] ← SRC1[7:0] << COUNT;
FI;
```

#### KSHIFTLQ

```
COUNT ← imm8[7:0]
DEST[MAX_KL-1:0] ← 0
IF COUNT ≤ 63
    THEN DEST[63:0] ← SRC1[63:0] << COUNT;
FI;
```

## **KSHIFTLD**

```
COUNT ← imm8[7:0]
DEST[MAX_KL-1:0] ← 0
IF COUNT ≤ 31
    THEN DEST[31:0] ← SRC1[31:0] << COUNT;
FI;
```

### **Intel C/C++ Compiler Intrinsic Equivalent**

Compiler auto generates KSHIFTLW when needed.

### **Flags Affected**

None

### **SIMD Floating-Point Exceptions**

None

### **Other Exceptions**

See Exceptions Type K20.



## KSHIFTRW/KSHIFTRB/KSHIFTRQ/KSHIFTRD—Shift Right Mask Registers

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.L0.66.0F3A.W1 30 /r KSHIFTRW k1, k2, imm8	RRI	V/V	AVX512F	Shift right 16 bits in k2 by immediate and write result in k1.
VEX.L0.66.0F3A.W0 30 /r KSHIFTRB k1, k2, imm8	RRI	V/V	AVX512DQ	Shift right 8 bits in k2 by immediate and write result in k1.
VEX.L0.66.0F3A.W1 31 /r KSHIFTRQ k1, k2, imm8	RRI	V/V	AVX512BW	Shift right 64 bits in k2 by immediate and write result in k1.
VEX.L0.66.0F3A.W0 31 /r KSHIFTRD k1, k2, imm8	RRI	V/V	AVX512BW	Shift right 32 bits in k2 by immediate and write result in k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3
RRI	ModRM:reg (w)	ModRM:r/m (r, ModRM:[7:6] must be 11b)	Imm8

### Description

Shifts 8/16/32/64 bits in the second operand (source operand) right by the count specified in immediate and place the least significant 8/16/32/64 bits of the result in the destination operand. The higher bits of the destination are zero-extended. The destination is set to zero if the count value is greater than 7 (for byte shift), 15 (for word shift), 31 (for doubleword shift) or 63 (for quadword shift).

### Operation

#### KSHIFTRW

```
COUNT ← imm8[7:0]
DEST[MAX_KL-1:0] ← 0
IF COUNT ≤ 15
    THEN DEST[15:0] ← SRC1[15:0] >> COUNT;
FI;
```

#### KSHIFTRB

```
COUNT ← imm8[7:0]
DEST[MAX_KL-1:0] ← 0
IF COUNT ≤ 7
    THEN DEST[7:0] ← SRC1[7:0] >> COUNT;
FI;
```

#### KSHIFTRQ

```
COUNT ← imm8[7:0]
DEST[MAX_KL-1:0] ← 0
IF COUNT ≤ 63
    THEN DEST[63:0] ← SRC1[63:0] >> COUNT;
FI;
```

## **KSHIFTRD**

```
COUNT ← imm8[7:0]
DEST[MAX_KL-1:0] ← 0
IF COUNT ≤ 31
    THEN DEST[31:0] ← SRC1[31:0] >> COUNT;
FI;
```

### **Intel C/C++ Compiler Intrinsic Equivalent**

Compiler auto generates KSHIFTRW when needed.

### **Flags Affected**

None

### **SIMD Floating-Point Exceptions**

None

### **Other Exceptions**

See Exceptions Type K20.

## KTESTW/KTESTB/KTESTQ/KTESTD—Packed Bit Test Masks and Set Flags

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.L0.0F.W0 99 /r KTESTW k1, k2	RR	V/V	AVX512DQ	Set ZF and CF depending on sign bit AND and ANDN of 16 bits mask register sources.
VEX.L0.66.0F.W0 99 /r KTESTB k1, k2	RR	V/V	AVX512DQ	Set ZF and CF depending on sign bit AND and ANDN of 8 bits mask register sources.
VEX.L0.0F.W1 99 /r KTESTQ k1, k2	RR	V/V	AVX512BW	Set ZF and CF depending on sign bit AND and ANDN of 64 bits mask register sources.
VEX.L0.66.0F.W1 99 /r KTESTD k1, k2	RR	V/V	AVX512BW	Set ZF and CF depending on sign bit AND and ANDN of 32 bits mask register sources.

### Instruction Operand Encoding

Op/En	Operand 1	Operand2
RR	ModRM:reg (r)	ModRM:r/m (r, ModRM:[7:6] must be 11b)

### Description

Performs a bitwise comparison of the bits of the first source operand and corresponding bits in the second source operand. If the AND operation produces all zeros, the ZF is set else the ZF is clear. If the bitwise AND operation of the inverted first source operand with the second source operand produces all zeros the CF is set else the CF is clear. Only the EFLAGS register is updated.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

### Operation

#### KTESTW

TEMP[15:0] ← SRC2[15:0] AND SRC1[15:0]

IF (TEMP[15:0] == 0)

THEN ZF ← 1;

ELSE ZF ← 0;

FI;

TEMP[15:0] ← SRC2[15:0] AND NOT SRC1[15:0]

IF (TEMP[15:0] == 0)

THEN CF ← 1;

ELSE CF ← 0;

FI;

AF ← OF ← PF ← SF ← 0;

#### KTESTB

TEMP[7:0] ← SRC2[7:0] AND SRC1[7:0]

IF (TEMP[7:0] == 0)

THEN ZF ← 1;

ELSE ZF ← 0;

FI;

TEMP[7:0] ← SRC2[7:0] AND NOT SRC1[7:0]

IF (TEMP[7:0] == 0)

THEN CF ← 1;

ELSE CF ← 0;

FI;

AF ← OF ← PF ← SF ← 0;

**KTESTQ** $TEMP[63:0] \leftarrow SRC2[63:0] \text{ AND } SRC1[63:0]$ IF ( $TEMP[63:0] = 0$ )    THEN ZF  $\leftarrow$  1;    ELSE ZF  $\leftarrow$  0;

FI;

 $TEMP[63:0] \leftarrow SRC2[63:0] \text{ AND NOT } SRC1[63:0]$ IF ( $TEMP[63:0] = 0$ )    THEN CF  $\leftarrow$  1;    ELSE CF  $\leftarrow$  0;

FI;

AF  $\leftarrow$  OF  $\leftarrow$  PF  $\leftarrow$  SF  $\leftarrow$  0;**KTESTD** $TEMP[31:0] \leftarrow SRC2[31:0] \text{ AND } SRC1[31:0]$ IF ( $TEMP[31:0] = 0$ )    THEN ZF  $\leftarrow$  1;    ELSE ZF  $\leftarrow$  0;

FI;

 $TEMP[31:0] \leftarrow SRC2[31:0] \text{ AND NOT } SRC1[31:0]$ IF ( $TEMP[31:0] = 0$ )    THEN CF  $\leftarrow$  1;    ELSE CF  $\leftarrow$  0;

FI;

AF  $\leftarrow$  OF  $\leftarrow$  PF  $\leftarrow$  SF  $\leftarrow$  0;**Intel C/C++ Compiler Intrinsic Equivalent****SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type K20.

## KUNPCKBW/KUNPCKWD/KUNPCKDQ—Unpack for Mask Registers

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.NDS.L1.66.0F.W0 4B /r KUNPCKBW k1, k2, k3	RVR	V/V	AVX512F	Unpack and interleave 8 bits masks in k2 and k3 and write word result in k1.
VEX.NDS.L1.0F.W0 4B /r KUNPCKWD k1, k2, k3	RVR	V/V	AVX512BW	Unpack and interleave 16 bits in k2 and k3 and write double-word result in k1.
VEX.NDS.L1.0F.W1 4B /r KUNPCKDQ k1, k2, k3	RVR	V/V	AVX512BW	Unpack and interleave 32 bits masks in k2 and k3 and write quadword result in k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3
RVR	ModRM:reg (w)	VEX.1vvv (r)	ModRM:r/m (r, ModRM:[7:6] must be 11b)

### Description

Unpacks the lower 8/16/32 bits of the second and third operands (source operands) into the low part of the first operand (destination operand), starting from the low bytes. The result is zero-extended in the destination.

### Operation

#### KUNPCKBW

```
DEST[7:0] ← SRC2[7:0]
DEST[15:8] ← SRC1[7:0]
DEST[MAX_KL-1:16] ← 0
```

#### KUNPCKWD

```
DEST[15:0] ← SRC2[15:0]
DEST[31:16] ← SRC1[15:0]
DEST[MAX_KL-1:32] ← 0
```

#### KUNPCKDQ

```
DEST[31:0] ← SRC2[31:0]
DEST[63:32] ← SRC1[31:0]
DEST[MAX_KL-1:64] ← 0
```

### Intel C/C++ Compiler Intrinsic Equivalent

```
KUNPCKBW __mmask16 __mm512_kunpackb(__mmask16 a, __mmask16 b);
KUNPCKDQ __mmask64 __mm512_kunpackd(__mmask64 a, __mmask64 b);
KUNPCKWD __mmask32 __mm512_kunpackw(__mmask32 a, __mmask32 b);
```

### Flags Affected

None

### SIMD Floating-Point Exceptions

None

### Other Exceptions

See Exceptions Type K20.

**KXNORW/KXNORB/KXNORQ/KXNORD—Bitwise Logical XNOR Masks**

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.NDS.L1.0F.W0 46 /r KXNORW k1, k2, k3	RVR	V/V	AVX512F	Bitwise XNOR 16 bits masks k2 and k3 and place result in k1.
VEX.L1.66.0F.W0 46 /r KXNORB k1, k2, k3	RVR	V/V	AVX512DQ	Bitwise XNOR 8 bits masks k2 and k3 and place result in k1.
VEX.L1.0F.W1 46 /r KXNORQ k1, k2, k3	RVR	V/V	AVX512BW	Bitwise XNOR 64 bits masks k2 and k3 and place result in k1.
VEX.L1.66.0F.W1 46 /r KXNORD k1, k2, k3	RVR	V/V	AVX512BW	Bitwise XNOR 32 bits masks k2 and k3 and place result in k1.

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3
RVR	ModRM:reg (w)	VEX.1vvv (r)	ModRM:r/m (r, ModRM:[7:6] must be 11b)

**Description**

Performs a bitwise XNOR between the vector mask k2 and the vector mask k3, and writes the result into vector mask k1 (three-operand form).

**Operation****KXNORW**

DEST[15:0] ← NOT (SRC1[15:0] BITWISE XOR SRC2[15:0])  
DEST[MAX\_KL-1:16] ← 0

**KXNORB**

DEST[7:0] ← NOT (SRC1[7:0] BITWISE XOR SRC2[7:0])  
DEST[MAX\_KL-1:8] ← 0

**KXNORQ**

DEST[63:0] ← NOT (SRC1[63:0] BITWISE XOR SRC2[63:0])  
DEST[MAX\_KL-1:64] ← 0

**KXNORD**

DEST[31:0] ← NOT (SRC1[31:0] BITWISE XOR SRC2[31:0])  
DEST[MAX\_KL-1:32] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

KXNORW \_\_mmask16 \_\_mm512\_kxnor(\_\_mmask16 a, \_\_mmask16 b);

**Flags Affected**

None

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type K20.

## KXORW/KXORB/KXORQ/KXORD—Bitwise Logical XOR Masks

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.NDS.L1.0F.W0 47 /r KXORW k1, k2, k3	RVR	V/V	AVX512F	Bitwise XOR 16 bits masks k2 and k3 and place result in k1.
VEX.L1.66.0F.W0 47 /r KXORB k1, k2, k3	RVR	V/V	AVX512DQ	Bitwise XOR 8 bits masks k2 and k3 and place result in k1.
VEX.L1.0F.W1 47 /r KXORQ k1, k2, k3	RVR	V/V	AVX512BW	Bitwise XOR 64 bits masks k2 and k3 and place result in k1.
VEX.L1.66.0F.W1 47 /r KXORD k1, k2, k3	RVR	V/V	AVX512BW	Bitwise XOR 32 bits masks k2 and k3 and place result in k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3
RVR	ModRM:reg (w)	VEX.1 vvv (r)	ModRM:r/m (r, ModRM:[7:6] must be 11b)

### Description

Performs a bitwise XOR between the vector mask k2 and the vector mask k3, and writes the result into vector mask k1 (three-operand form).

### Operation

#### KXORW

DEST[15:0] ← SRC1[15:0] BITWISE XOR SRC2[15:0]  
DEST[MAX\_KL-1:16] ← 0

#### KXORB

DEST[7:0] ← SRC1[7:0] BITWISE XOR SRC2[7:0]  
DEST[MAX\_KL-1:8] ← 0

#### KXORQ

DEST[63:0] ← SRC1[63:0] BITWISE XOR SRC2[63:0]  
DEST[MAX\_KL-1:64] ← 0

#### KXORD

DEST[31:0] ← SRC1[31:0] BITWISE XOR SRC2[31:0]  
DEST[MAX\_KL-1:32] ← 0

### Intel C/C++ Compiler Intrinsic Equivalent

KXORW \_\_mmask16 \_mm512\_kxor(\_\_mmask16 a, \_\_mmask16 b);

### Flags Affected

None

### SIMD Floating-Point Exceptions

None

### Other Exceptions

See Exceptions Type K20.





## 9. Updates to Chapter 4, Volume 2B

Change bars show changes to Chapter 4 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2B: Instruction Set Reference, M-U*.

---

Changes include minor updates to reflect volume addition, arrangement of information and document order numbers. Addition of Intel® AVX-512 instructions. The following instructions have been added and/or updated: MAXPD, MAXPS, MAXSD, MAXSS, MINPD, MINPS, MINS, MINSS, MOVAPD, MOVAPS, MOVD/MOVQ, MOVDDUP, MOVDQA, VMOVDQA32/64, MOVDQU/MOVDQU8/16/32/64, MOVHLPS, MOVHPD, MOVHPS, MOVLHPS, MOVLPD, MOVLPS, MOVNTDQA, MOVNTDQ, MOVNTPD, MOVNTPS, MOVQ, MOVSD, MOVSHDUP, MOVSLDUP, MOVSS, MOVUPD, MOVUPS, MULPD, MULPS, MULSD, MULSS, ORPD, ORPS, PABSB/PABSW/PABSD/PABSQ, PACKUSDW, PACKUSWB, PADDB/PADDW/PADDD/PADDQ, PACKSSWB/PACKSSDW, PACKUSWB, PADDB/PADDW/PADDD/PADDQ, PADDSB/PADDSW, PADDUSB/PADDUSW, PALIGNR, PAND, PANDN, PAVGB/PAVGW, PBLENDVB, PBLENDW, PCLMULQDQ, PCMPEQB/PCMPEQW/PCMPEQD, PCMPEQQ, PCMPGTB/PCMPGTW/PCMPGTD, PCMPGTQ, PEXTRB/PEXTRD/PEXTRQ, PINSRB/PINSRD/PINSRQ, PINSRW, PMADDUBSW, PMADDWD, PMAXSB/PMAXSW/PMAXSD/PMAXSQ, PMAXUB/PMAXUW, PMINSB/PMINSW, PMINS, PMINSD/PMINSQ, PMINUB/PMINUW, PMINUD/PMINUQ, PMOVVSX, PMOVZX, PMULDQ, PMULHR, PMULHSW, PMULHUW, PMULHW, PMULLD/PMULLQ, PMULLW, PMULUDQ, POR, PSADB, PSADWB, PSADHW, PSHUF, PSHUFB, PSHUFD, PSHUFHW, PSHUFLW, PSLLDQ, PSLLW/PSLLD/PSLLQ, PSRAW/PSRAD/PSRAQ, PSRLDQ, PSRLW/PSRLD/PSRLQ, PSUBB/PSUBW/PSUBD, PSUBQ, PSUBSB/PSUBSW, PSUBUSB/PSUBUSW, PTWRITE, PUNPCKHBW/PUNPCKHWD/PUNPCKHDQ/PUNPCKHQDQ, PUNPCKLBW/PUNPCKLWD/PUNPCKLDQ/PUNPCKLQDQ, PXOR, RCL/RCLL/RCLLQ/ROR, SHUFPD, SHUFPS, SQRTPD, SQRTPS, SQRTSD, SQRTSS, SUBPD, SUBPS, SUBSD, SUBSS, UCOMISD, UCOMISS, UNPCKHPD, UNPCKHPS, UNPCKLPD, UNPCKLPS.

## 4.1 IMM8 CONTROL BYTE OPERATION FOR PCMPESTRI / PCMPESTRM / PCMPISTRI / PCMPISTRM

The notations introduced in this section are referenced in the reference pages of PCMPESTRI, PCMPESTRM, PCMPISTRI, PCMPISTRM. The operation of the immediate control byte is common to these four string text processing instructions of SSE4.2. This section describes the common operations.

### 4.1.1 General Description

The operation of PCMPESTRI, PCMPESTRM, PCMPISTRI, PCMPISTRM is defined by the combination of the respective opcode and the interpretation of an immediate control byte that is part of the instruction encoding.

The opcode controls the relationship of input bytes/words to each other (determines whether the inputs terminated strings or whether lengths are expressed explicitly) as well as the desired output (index or mask).

The Imm8 Control Byte for PCMPESTRM/PCMPESTRI/PCMPISTRM/PCMPISTRI encodes a significant amount of programmable control over the functionality of those instructions. Some functionality is unique to each instruction while some is common across some or all of the four instructions. This section describes functionality which is common across the four instructions.

The arithmetic flags (ZF, CF, SF, OF, AF, PF) are set as a result of these instructions. However, the meanings of the flags have been overloaded from their typical meanings in order to provide additional information regarding the relationships of the two inputs.

PCMPxSTRx instructions perform arithmetic comparisons between all possible pairs of bytes or words, one from each packed input source operand. The boolean results of those comparisons are then aggregated in order to produce meaningful results. The Imm8 Control Byte is used to affect the interpretation of individual input elements as well as control the arithmetic comparisons used and the specific aggregation scheme.

Specifically, the Imm8 Control Byte consists of bit fields that control the following attributes:

- **Source data format** — Byte/word data element granularity, signed or unsigned elements
- **Aggregation operation** — Encodes the mode of per-element comparison operation and the aggregation of per-element comparisons into an intermediate result
- **Polarity** — Specifies intermediate processing to be performed on the intermediate result
- **Output selection** — Specifies final operation to produce the output (depending on index or mask) from the intermediate result

## 4.1.2 Source Data Format

Table 4-1. Source Data Format

Imm8[1:0]	Meaning	Description
00b	Unsigned bytes	Both 128-bit sources are treated as packed, unsigned bytes.
01b	Unsigned words	Both 128-bit sources are treated as packed, unsigned words.
10b	Signed bytes	Both 128-bit sources are treated as packed, signed bytes.
11b	Signed words	Both 128-bit sources are treated as packed, signed words.

If the Imm8 Control Byte has bit[0] cleared, each source contains 16 packed bytes. If the bit is set each source contains 8 packed words. If the Imm8 Control Byte has bit[1] cleared, each input contains unsigned data. If the bit is set each source contains signed data.

## 4.1.3 Aggregation Operation

Table 4-2. Aggregation Operation

Imm8[3:2]	Mode	Comparison
00b	Equal any	The arithmetic comparison is "equal."
01b	Ranges	Arithmetic comparison is "greater than or equal" between even indexed bytes/words of reg and each byte/word of reg/mem. Arithmetic comparison is "less than or equal" between odd indexed bytes/words of reg and each byte/word of reg/mem. (reg/mem[m] >= reg[n] for n = even, reg/mem[m] <= reg[n] for n = odd)
10b	Equal each	The arithmetic comparison is "equal."
11b	Equal ordered	The arithmetic comparison is "equal."

All 256 (64) possible comparisons are always performed. The individual Boolean results of those comparisons are referred by "BoolRes[Reg/Mem element index, Reg element index]." Comparisons evaluating to "True" are represented with a 1, False with a 0 (positive logic). The initial results are then aggregated into a 16-bit (8-bit) intermediate result (IntRes1) using one of the modes described in the table below, as determined by Imm8 Control Byte bit[3:2].

See Section 4.1.6 for a description of the `overrideIfDataInvalid()` function used in Table 4-3.

**Table 4-3. Aggregation Operation**

Mode	Pseudocode
Equal any (find characters from a set)	<pre>UpperBound = imm8[0] ? 7 : 15; IntRes1 = 0; For j = 0 to UpperBound, j++ For i = 0 to UpperBound, i++ IntRes1[j] OR= overrideIfDataInvalid(BoolRes[j,i])</pre>
Ranges (find characters from ranges)	<pre>UpperBound = imm8[0] ? 7 : 15; IntRes1 = 0; For j = 0 to UpperBound, j++ For i = 0 to UpperBound, i+=2 IntRes1[j] OR= (overrideIfDataInvalid(BoolRes[j,i]) AND overrideIfDataInvalid(BoolRes[j,i+1]))</pre>
Equal each (string compare)	<pre>UpperBound = imm8[0] ? 7 : 15; IntRes1 = 0; For i = 0 to UpperBound, i++ IntRes1[i] = overrideIfDataInvalid(BoolRes[i,i])</pre>
Equal ordered (substring search)	<pre>UpperBound = imm8[0] ? 7 : 15; IntRes1 = imm8[0] ? FFH : FFFFH For j = 0 to UpperBound, j++ For i = 0 to UpperBound-j, k=j to UpperBound, k++, i++ IntRes1[j] AND= overrideIfDataInvalid(BoolRes[k,i])</pre>

#### 4.1.4 Polarity

`IntRes1` may then be further modified by performing a 1's complement, according to the value of the `Imm8` Control Byte bit[4]. Optionally, a mask may be used such that only those `IntRes1` bits which correspond to "valid" reg/mem input elements are complemented (note that the definition of a valid input element is dependant on the specific opcode and is defined in each opcode's description). The result of the possible negation is referred to as `IntRes2`.

**Table 4-4. Polarity**

Imm8[5:4]	Operation	Description
00b	Positive Polarity (+)	<code>IntRes2 = IntRes1</code>
01b	Negative Polarity (-)	<code>IntRes2 = -1 XOR IntRes1</code>
10b	Masked (+)	<code>IntRes2 = IntRes1</code>
11b	Masked (-)	<code>IntRes2[i] = IntRes1[i] if reg/mem[i] invalid, else = ~IntRes1[i]</code>

## 4.1.5 Output Selection

**Table 4-5. Output Selection**

Imm8[6]	Operation	Description
0b	Least significant index	The index returned to ECX is of the least significant set bit in IntRes2.
1b	Most significant index	The index returned to ECX is of the most significant set bit in IntRes2.

For PCMPSTRI/PCMPISTRI, the Imm8 Control Byte bit[6] is used to determine if the index is of the least significant or most significant bit of IntRes2.

**Table 4-6. Output Selection**

Imm8[6]	Operation	Description
0b	Bit mask	IntRes2 is returned as the mask to the least significant bits of XMM0 with zero extension to 128 bits.
1b	Byte/word mask	IntRes2 is expanded into a byte/word mask (based on imm8[1]) and placed in XMM0. The expansion is performed by replicating each bit into all of the bits of the byte/word of the same index.

Specifically for PCMPSTRM/PCMPISTRM, the Imm8 Control Byte bit[6] is used to determine if the mask is a 16 (8) bit mask or a 128 bit byte/word mask.

## 4.1.6 Valid/Invalid Override of Comparisons

PCMPxSTRx instructions allow for the possibility that an end-of-string (EOS) situation may occur within the 128-bit packed data value (see the instruction descriptions below for details). Any data elements on either source that are determined to be past the EOS are considered to be invalid, and the treatment of invalid data within a comparison pair varies depending on the aggregation function being performed.

In general, the individual comparison result for each element pair BoolRes[i.j] can be forced true or false if one or more elements in the pair are invalid. See Table 4-7.

**Table 4-7. Comparison Result for Each Element Pair BoolRes[i.j]**

xmm1 byte/ word	xmm2/ m128 byte/word	Imm8[3:2] = 00b (equal any)	Imm8[3:2] = 01b (ranges)	Imm8[3:2] = 10b (equal each)	Imm8[3:2] = 11b (equal ordered)
Invalid	Invalid	Force false	Force false	Force true	Force true
Invalid	Valid	Force false	Force false	Force false	Force true
Valid	Invalid	Force false	Force false	Force false	Force false
Valid	Valid	Do not force	Do not force	Do not force	Do not force

## 4.1.7 Summary of Im8 Control byte

Table 4-8. Summary of Imm8 Control Byte

Imm8	Description
-----0b	128-bit sources treated as 16 packed bytes.
-----1b	128-bit sources treated as 8 packed words.
-----0-b	Packed bytes/words are unsigned.
-----1-b	Packed bytes/words are signed.
---00--b	Mode is equal any.
---01--b	Mode is ranges.
---10--b	Mode is equal each.
---11--b	Mode is equal ordered.
--0---b	IntRes1 is unmodified.
--1---b	IntRes1 is negated (1's complement).
-0----b	Negation of IntRes1 is for all 16 (8) bits.
-1----b	Negation of IntRes1 is masked by reg/mem validity.
-0-----b	Index of the least significant, set, bit is used (regardless of corresponding input element validity). IntRes2 is returned in least significant bits of XMM0.
-1-----b	Index of the most significant, set, bit is used (regardless of corresponding input element validity). Each bit of IntRes2 is expanded to byte/word.
0-----b	This bit currently has no defined effect, should be 0.
1-----b	This bit currently has no defined effect, should be 0.

## 4.1.8 Diagram Comparison and Aggregation Process

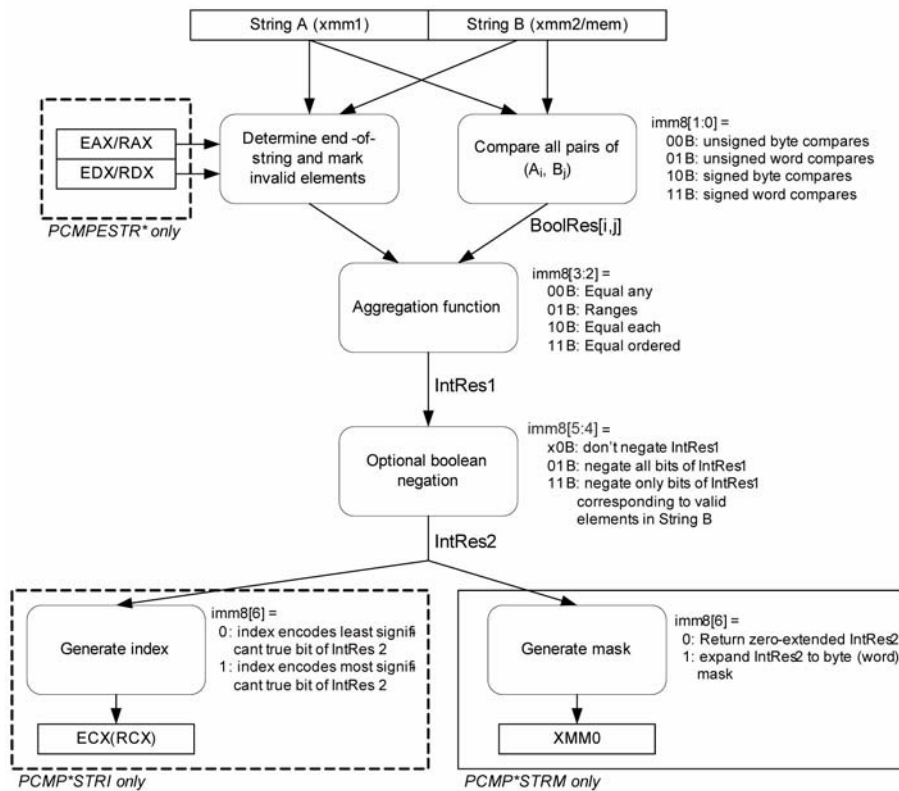


Figure 4-1. Operation of PCMPSTRx and PCMPSTRx

## 4.2 COMMON TRANSFORMATION AND PRIMITIVE FUNCTIONS FOR SHA1XXX AND SHA256XXX

The following primitive functions and transformations are used in the algorithmic descriptions of SHA1 and SHA256 instruction extensions SHA1NEXTE, SHA1RNDS4, SHA1MSG1, SHA1MSG2, SHA256RNDS4, SHA256MSG1 and SHA256MSG2. The operands of these primitives and transformation are generally 32-bit DWORD integers.

- $f_0()$ : A bit oriented logical operation that derives a new dword from three SHA1 state variables (dword). This function is used in SHA1 round 1 to 20 processing.

$$f_0(B,C,D) \leftarrow (B \text{ AND } C) \text{ XOR } ((\text{NOT}(B) \text{ AND } D))$$
- $f_1()$ : A bit oriented logical operation that derives a new dword from three SHA1 state variables (dword). This function is used in SHA1 round 21 to 40 processing.

$$f_1(B,C,D) \leftarrow B \text{ XOR } C \text{ XOR } D$$
- $f_2()$ : A bit oriented logical operation that derives a new dword from three SHA1 state variables (dword). This function is used in SHA1 round 41 to 60 processing.

$$f_2(B,C,D) \leftarrow (B \text{ AND } C) \text{ XOR } (B \text{ AND } D) \text{ XOR } (C \text{ AND } D)$$

- $f3()$ : A bit oriented logical operation that derives a new dword from three SHA1 state variables (dword). This function is used in SHA1 round 61 to 80 processing. It is the same as  $f1()$ .  
 $f3(B,C,D) \leftarrow B \text{ XOR } C \text{ XOR } D$

- $Ch()$ : A bit oriented logical operation that derives a new dword from three SHA256 state variables (dword).  
 $Ch(E,F,G) \leftarrow (E \text{ AND } F) \text{ XOR } ((\text{NOT } E) \text{ AND } G)$

- $Maj()$ : A bit oriented logical operation that derives a new dword from three SHA256 state variables (dword).  
 $Maj(A,B,C) \leftarrow (A \text{ AND } B) \text{ XOR } (A \text{ AND } C) \text{ XOR } (B \text{ AND } C)$

ROR is rotate right operation

$$(A \text{ ROR } N) \leftarrow A[N-1:0] \parallel A[\text{Width}-1:N]$$

ROL is rotate left operation

$$(A \text{ ROL } N) \leftarrow A \text{ ROR } (\text{Width}-N)$$

SHR is the right shift operation

$$(A \text{ SHR } N) \leftarrow \text{ZEROES}[N-1:0] \parallel A[\text{Width}-1:N]$$

- $\Sigma_0()$ : A bit oriented logical and rotational transformation performed on a dword SHA256 state variable.  
 $\Sigma_0(A) \leftarrow (A \text{ ROR } 2) \text{ XOR } (A \text{ ROR } 13) \text{ XOR } (A \text{ ROR } 22)$
- $\Sigma_1()$ : A bit oriented logical and rotational transformation performed on a dword SHA256 state variable.  
 $\Sigma_1(E) \leftarrow (E \text{ ROR } 6) \text{ XOR } (E \text{ ROR } 11) \text{ XOR } (E \text{ ROR } 25)$
- $\sigma_0()$ : A bit oriented logical and rotational transformation performed on a SHA256 message dword used in the message scheduling.  
 $\sigma_0(W) \leftarrow (W \text{ ROR } 7) \text{ XOR } (W \text{ ROR } 18) \text{ XOR } (W \text{ SHR } 3)$
- $\sigma_1()$ : A bit oriented logical and rotational transformation performed on a SHA256 message dword used in the message scheduling.  
 $\sigma_1(W) \leftarrow (W \text{ ROR } 17) \text{ XOR } (W \text{ ROR } 19) \text{ XOR } (W \text{ SHR } 10)$
- $K_i$ : SHA1 Constants dependent on immediate  $i$ .  
 $K0 = 0x5A827999$   
 $K1 = 0x6ED9EBA1$   
 $K2 = 0X8F1BBCDC$   
 $K3 = 0xCA62C1D6$

## 4.3 INSTRUCTIONS (M-U)

Chapter 4 continues an alphabetical discussion of Intel<sup>®</sup> 64 and IA-32 instructions (M-U). See also: Chapter 3, "Instruction Set Reference, A-L," in the *Intel<sup>®</sup> 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*, and Chapter 5, "Instruction Set Reference, V-Z," in the *Intel<sup>®</sup> 64 and IA-32 Architectures Software Developer's Manual, Volume 2C*.



## MAXPD—Maximum of Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 5F /r MAXPD xmm1, xmm2/m128	RM	V/V	SSE2	Return the maximum double-precision floating-point values between xmm1 and xmm2/m128.
VEX.NDS.128.66.0F.WIG 5F /r VMAXPD xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Return the maximum double-precision floating-point values between xmm2 and xmm3/m128.
VEX.NDS.256.66.0F.WIG 5F /r VMAXPD ymm1, ymm2, ymm3/m256	RVM	V/V	AVX	Return the maximum packed double-precision floating-point values between ymm2 and ymm3/m256.
EVEX.NDS.128.66.0F.W1 5F /r VMAXPD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	FV	V/V	AVX512VL AVX512F	Return the maximum packed double-precision floating-point values between xmm2 and xmm3/m128/m64bcst and store result in xmm1 subject to writemask k1.
EVEX.NDS.256.66.0F.W1 5F /r VMAXPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	FV	V/V	AVX512VL AVX512F	Return the maximum packed double-precision floating-point values between ymm2 and ymm3/m256/m64bcst and store result in ymm1 subject to writemask k1.
EVEX.NDS.512.66.0F.W1 5F /r VMAXPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{sae}	FV	V/V	AVX512F	Return the maximum packed double-precision floating-point values between zmm2 and zmm3/m512/m64bcst and store result in zmm1 subject to writemask k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
FV	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

### Description

Performs a SIMD compare of the packed double-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 MAXPD can be emulated using a sequence of instructions, such as a comparison followed by AND, ANDN and OR.

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 64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAX\_VL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: The first source operand is a XMM register. The second source operand can be a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper bits (MAX\_VL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAX\_VL-1:128) of the corresponding ZMM register destination are unmodified.

### Operation

```
MAX(SRC1, SRC2)
{
  IF ((SRC1 = 0.0) and (SRC2 = 0.0)) THEN DEST ← SRC2;
  ELSE IF (SRC1 = SNaN) THEN DEST ← SRC2; FI;
  ELSE IF (SRC2 = SNaN) THEN DEST ← SRC2; FI;
  ELSE IF (SRC1 > SRC2) THEN DEST ← SRC1;
  ELSE DEST ← SRC2;
  FI;
}
```

### VMAXPD (EVEX encoded versions)

(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) AND (SRC2 *is memory*)
        THEN
          DEST[i+63:i] ← MAX(SRC1[j+63:i], SRC2[63:0])
        ELSE
          DEST[i+63:i] ← MAX(SRC1[j+63:i], SRC2[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[MAX_VL-1:VL] ← 0
```

### VMAXPD (VEX.256 encoded version)

```
DEST[63:0] ← MAX(SRC1[63:0], SRC2[63:0])
DEST[127:64] ← MAX(SRC1[127:64], SRC2[127:64])
DEST[191:128] ← MAX(SRC1[191:128], SRC2[191:128])
DEST[255:192] ← MAX(SRC1[255:192], SRC2[255:192])
DEST[MAX_VL-1:256] ← 0
```

### VMAXPD (VEX.128 encoded version)

```
DEST[63:0] ← MAX(SRC1[63:0], SRC2[63:0])
DEST[127:64] ← MAX(SRC1[127:64], SRC2[127:64])
DEST[MAX_VL-1:128] ← 0
```

**MAXPD (128-bit Legacy SSE version)**

DEST[63:0] ← MAX(DEST[63:0], SRC[63:0])

DEST[127:64] ← MAX(DEST[127:64], SRC[127:64])

DEST[MAX\_VL-1:128] (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**

VMAXPD \_\_m512d \_mm512\_max\_pd( \_\_m512d a, \_\_m512d b);

VMAXPD \_\_m512d \_mm512\_mask\_max\_pd(\_\_m512d s, \_\_mmask8 k, \_\_m512d a, \_\_m512d b,);

VMAXPD \_\_m512d \_mm512\_maskz\_max\_pd( \_\_mmask8 k, \_\_m512d a, \_\_m512d b);

VMAXPD \_\_m512d \_mm512\_max\_round\_pd( \_\_m512d a, \_\_m512d b, int);

VMAXPD \_\_m512d \_mm512\_mask\_max\_round\_pd(\_\_m512d s, \_\_mmask8 k, \_\_m512d a, \_\_m512d b, int);

VMAXPD \_\_m512d \_mm512\_maskz\_max\_round\_pd( \_\_mmask8 k, \_\_m512d a, \_\_m512d b, int);

VMAXPD \_\_m256d \_mm256\_mask\_max\_pd(\_\_m256d s, \_\_mmask8 k, \_\_m256d a, \_\_m256d b);

VMAXPD \_\_m256d \_mm256\_maskz\_max\_pd( \_\_mmask8 k, \_\_m256d a, \_\_m256d b);

VMAXPD \_\_m128d \_mm\_mask\_max\_pd(\_\_m128d s, \_\_mmask8 k, \_\_m128d a, \_\_m128d b);

VMAXPD \_\_m128d \_mm\_maskz\_max\_pd( \_\_mmask8 k, \_\_m128d a, \_\_m128d b);

VMAXPD \_\_m256d \_mm256\_max\_pd( \_\_m256d a, \_\_m256d b);

(V)VMAXPD \_\_m128d \_mm\_max\_pd( \_\_m128d a, \_\_m128d b);

**SIMD Floating-Point Exceptions**

Invalid (including QNaN Source Operand), Denormal

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 2.

EVEX-encoded instruction, see Exceptions Type E2.

## MAXPS—Maximum of Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
OF 5F /r MAXPS xmm1, xmm2/m128	RM	V/V	SSE	Return the maximum single-precision floating-point values between xmm1 and xmm2/mem.
VEX.NDS.128.OF.WIG 5F /r VMAXPS xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Return the maximum single-precision floating-point values between xmm2 and xmm3/mem.
VEX.NDS.256.OF.WIG 5F /r VMAXPS ymm1, ymm2, ymm3/m256	RVM	V/V	AVX	Return the maximum single-precision floating-point values between ymm2 and ymm3/mem.
EVEX.NDS.128.OF.W0 5F /r VMAXPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	FV	V/V	AVX512VL AVX512F	Return the maximum packed single-precision floating-point values between xmm2 and xmm3/m128/m32bcst and store result in xmm1 subject to writemask k1.
EVEX.NDS.256.OF.W0 5F /r VMAXPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	FV	V/V	AVX512VL AVX512F	Return the maximum packed single-precision floating-point values between ymm2 and ymm3/m256/m32bcst and store result in ymm1 subject to writemask k1.
EVEX.NDS.512.OF.W0 5F /r VMAXPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{sae}	FV	V/V	AVX512F	Return the maximum packed single-precision floating-point values between zmm2 and zmm3/m512/m32bcst and store result in zmm1 subject to writemask k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
FV	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

### Description

Performs a SIMD compare of the packed single-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 MAXPS can be emulated using a sequence of instructions, such as, a comparison followed by AND, ANDN and OR.

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 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAX\_VL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: The first source operand is a XMM register. The second source operand can be a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper bits (MAX\_VL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAX\_VL-1:128) of the corresponding ZMM register destination are unmodified.

**Operation**

```

MAX(SRC1, SRC2)
{
  IF ((SRC1 = 0.0) and (SRC2 = 0.0)) THEN DEST ← SRC2;
  ELSE IF (SRC1 = SNaN) THEN DEST ← SRC2; FI;
  ELSE IF (SRC2 = SNaN) THEN DEST ← SRC2; FI;
  ELSE IF (SRC1 > SRC2) THEN DEST ← SRC1;
  ELSE DEST ← SRC2;
  FI;
}

```

**VMAXPS (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

  i ← j \* 32

  IF k1[j] OR \*no writemask\*

    THEN

      IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

        THEN

          DEST[i+31:i] ← MAX(SRC1[i+31:i], SRC2[31:0])

        ELSE

          DEST[i+31:i] ← MAX(SRC1[i+31:i], SRC2[i+31:i])

      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[MAX\_VL-1:VL] ← 0

**VMAXPS (VEX.256 encoded version)**

DEST[31:0] ← MAX(SRC1[31:0], SRC2[31:0])

DEST[63:32] ← MAX(SRC1[63:32], SRC2[63:32])

DEST[95:64] ← MAX(SRC1[95:64], SRC2[95:64])

DEST[127:96] ← MAX(SRC1[127:96], SRC2[127:96])

DEST[159:128] ← MAX(SRC1[159:128], SRC2[159:128])

DEST[191:160] ← MAX(SRC1[191:160], SRC2[191:160])

DEST[223:192] ← MAX(SRC1[223:192], SRC2[223:192])

DEST[255:224] ← MAX(SRC1[255:224], SRC2[255:224])

DEST[MAX\_VL-1:256] ← 0

**VMAXPS (VEX.128 encoded version)**

DEST[31:0] ← MAX(SRC1[31:0], SRC2[31:0])

DEST[63:32] ← MAX(SRC1[63:32], SRC2[63:32])

DEST[95:64] ← MAX(SRC1[95:64], SRC2[95:64])

DEST[127:96] ← MAX(SRC1[127:96], SRC2[127:96])

DEST[MAX\_VL-1:128] ← 0

**MAXPS (128-bit Legacy SSE version)**

$\text{DEST}[31:0] \leftarrow \text{MAX}(\text{DEST}[31:0], \text{SRC}[31:0])$   
 $\text{DEST}[63:32] \leftarrow \text{MAX}(\text{DEST}[63:32], \text{SRC}[63:32])$   
 $\text{DEST}[95:64] \leftarrow \text{MAX}(\text{DEST}[95:64], \text{SRC}[95:64])$   
 $\text{DEST}[127:96] \leftarrow \text{MAX}(\text{DEST}[127:96], \text{SRC}[127:96])$   
 $\text{DEST}[\text{MAX\_VL}-1:128]$  (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**

`VMAXPS __m512 __mm512_max_ps( __m512 a, __m512 b);`  
`VMAXPS __m512 __mm512_mask_max_ps(__m512 s, __mmask16 k, __m512 a, __m512 b);`  
`VMAXPS __m512 __mm512_maskz_max_ps( __mmask16 k, __m512 a, __m512 b);`  
`VMAXPS __m512 __mm512_max_round_ps( __m512 a, __m512 b, int);`  
`VMAXPS __m512 __mm512_mask_max_round_ps(__m512 s, __mmask16 k, __m512 a, __m512 b, int);`  
`VMAXPS __m512 __mm512_maskz_max_round_ps( __mmask16 k, __m512 a, __m512 b, int);`  
`VMAXPS __m256 __mm256_mask_max_ps(__m256 s, __mmask8 k, __m256 a, __m256 b);`  
`VMAXPS __m256 __mm256_maskz_max_ps( __mmask8 k, __m256 a, __m256 b);`  
`VMAXPS __m128 __mm_mask_max_ps(__m128 s, __mmask8 k, __m128 a, __m128 b);`  
`VMAXPS __m128 __mm_maskz_max_ps( __mmask8 k, __m128 a, __m128 b);`  
`VMAXPS __m256 __mm256_max_ps ( __m256 a, __m256 b);`  
`MAXPS __m128 __mm_max_ps ( __m128 a, __m128 b);`

**SIMD Floating-Point Exceptions**

Invalid (including QNaN Source Operand), Denormal

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 2.

EVEX-encoded instruction, see Exceptions Type E2.

## MAXSD—Return Maximum Scalar Double-Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 5F /r MAXSD xmm1, xmm2/m64	RM	V/V	SSE2	Return the maximum scalar double-precision floating-point value between xmm2/m64 and xmm1.
VEX.NDS.128.F2.0F.WIG 5F /r VMAXSD xmm1, xmm2, xmm3/m64	RVM	V/V	AVX	Return the maximum scalar double-precision floating-point value between xmm3/m64 and xmm2.
EVEX.NDS.LIG.F2.0F.W1 5F /r VMAXSD xmm1 {k1}{z}, xmm2, xmm3/m64{sae}	T1S	V/V	AVX512F	Return the maximum scalar double-precision floating-point value between xmm3/m64 and xmm2.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
T1S	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

### Description

Compares the low double-precision floating-point values in the first source operand and second the source operand, and returns the maximum value to the low quadword of the destination operand. The second source operand can be an XMM register or a 64-bit memory location. The first source and destination operands are XMM registers. When the second source operand is a memory operand, only 64 bits are accessed.

If the values being compared are both 0.0s (of either sign), the value in the second source operand is returned. If a value in the second source operand is an SNaN, that SNaN is returned 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 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 of either source operand be returned, the action of MAXSD can be emulated using a sequence of instructions, such as, a comparison followed by AND, ANDN and OR.

128-bit Legacy SSE version: The destination and first source operand are the same. Bits (MAX\_VL-1:64) of the corresponding destination register remain unchanged.

VEX.128 and EVEX encoded version: Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (MAX\_VL-1:128) of the destination register are zeroed.

EVEX encoded version: The low quadword element of the destination operand is updated according to the writemask.

Software should ensure VMAXSD is encoded with VEX.L=0. Encoding VMAXSD with VEX.L=1 may encounter unpredictable behavior across different processor generations.

**Operation**

```

MAX(SRC1, SRC2)
{
  IF ((SRC1 = 0.0) and (SRC2 = 0.0)) THEN DEST ← SRC2;
  ELSE IF (SRC1 = SNaN) THEN DEST ← SRC2; FI;
  ELSE IF (SRC2 = SNaN) THEN DEST ← SRC2; FI;
  ELSE IF (SRC1 > SRC2) THEN DEST ← SRC1;
  ELSE DEST ← SRC2;
  FI;
}

```

**VMAXSD (EVEX encoded version)**

```

IF k1[0] or *no writemask*
  THEN  DEST[63:0] ← MAX(SRC1[63:0], SRC2[63:0])
  ELSE
    IF *merging-masking*           ; merging-masking
      THEN *DEST[63:0] remains unchanged*
    ELSE                             ; zeroing-masking
      DEST[63:0] ← 0
    FI;
  FI;
DEST[127:64] ← SRC1[127:64]
DEST[MAX_VL-1:128] ← 0

```

**VMAXSD (VEX.128 encoded version)**

```

DEST[63:0] ← MAX(SRC1[63:0], SRC2[63:0])
DEST[127:64] ← SRC1[127:64]
DEST[MAX_VL-1:128] ← 0

```

**MAXSD (128-bit Legacy SSE version)**

```

DEST[63:0] ← MAX(DEST[63:0], SRC[63:0])
DEST[MAX_VL-1:64] (Unmodified)

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VMAXSD __m128d __mm_max_round_sd( __m128d a, __m128d b, int);
VMAXSD __m128d __mm_mask_max_round_sd(__m128d s, __mmask8 k, __m128d a, __m128d b, int);
VMAXSD __m128d __mm_maskz_max_round_sd( __mmask8 k, __m128d a, __m128d b, int);
MAXSD __m128d __mm_max_sd(__m128d a, __m128d b)

```

**SIMD Floating-Point Exceptions**

Invalid (Including QNaN Source Operand), Denormal

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 3.  
 EVEX-encoded instruction, see Exceptions Type E3.



## MAXSS—Return Maximum Scalar Single-Precision Floating-Point Value

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 5F /r MAXSS xmm1, xmm2/m32	RM	V/V	SSE	Return the maximum scalar single-precision floating-point value between xmm2/m32 and xmm1.
VEX.NDS.128.F3.0F.WIG 5F /r VMAXSS xmm1, xmm2, xmm3/m32	RVM	V/V	AVX	Return the maximum scalar single-precision floating-point value between xmm3/m32 and xmm2.
EVEX.NDS.LIG.F3.0F.W0 5F /r VMAXSS xmm1 {k1}{z}, xmm2, xmm3/m32{sae}	T1S	V/V	AVX512F	Return the maximum scalar single-precision floating-point value between xmm3/m32 and xmm2.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
T1S	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

### Description

Compares the low single-precision floating-point values in the first source operand and the second source operand, and returns the maximum value to the low doubleword of the destination operand.

If the values being compared are both 0.0s (of either sign), the value in the second source operand is returned. If a value in the second source operand is an SNaN, that SNaN is returned 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 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 from either source operand be returned, the action of MAXSS can be emulated using a sequence of instructions, such as, a comparison followed by AND, ANDN and OR.

The second source operand can be an XMM register or a 32-bit memory location. The first source and destination operands are XMM registers.

128-bit Legacy SSE version: The destination and first source operand are the same. Bits (MAX\_VL:32) of the corresponding destination register remain unchanged.

VEX.128 and EVEX encoded version: The first source operand is an xmm register encoded by VEX.vvvv. Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (MAX\_VL:128) of the destination register are zeroed.

EVEX encoded version: The low doubleword element of the destination operand is updated according to the writemask.

Software should ensure VMAXSS is encoded with VEX.L=0. Encoding VMAXSS with VEX.L=1 may encounter unpredictable behavior across different processor generations.

**Operation**

```

MAX(SRC1, SRC2)
{
  IF ((SRC1 = 0.0) and (SRC2 = 0.0)) THEN DEST ← SRC2;
  ELSE IF (SRC1 = SNaN) THEN DEST ← SRC2; FI;
  ELSE IF (SRC2 = SNaN) THEN DEST ← SRC2; FI;
  ELSE IF (SRC1 > SRC2) THEN DEST ← SRC1;
  ELSE DEST ← SRC2;
  FI;
}

```

**VMAXSS (EVEX encoded version)**

```

IF k1[0] or *no writemask*
  THEN  DEST[31:0] ← MAX(SRC1[31:0], SRC2[31:0])
  ELSE
    IF *merging-masking*           ; merging-masking
      THEN *DEST[31:0] remains unchanged*
    ELSE                             ; zeroing-masking
      THEN DEST[31:0] ← 0
    FI;
  FI;
DEST[127:32] ← SRC1[127:32]
DEST[MAX_VL-1:128] ← 0

```

**VMAXSS (VEX.128 encoded version)**

```

DEST[31:0] ← MAX(SRC1[31:0], SRC2[31:0])
DEST[127:32] ← SRC1[127:32]
DEST[MAX_VL-1:128] ← 0

```

**MAXSS (128-bit Legacy SSE version)**

```

DEST[31:0] ← MAX(DEST[31:0], SRC[31:0])
DEST[MAX_VL-1:32] (Unmodified)

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VMAXSS __m128 _mm_max_round_ss(__m128 a, __m128 b, int);
VMAXSS __m128 _mm_mask_max_round_ss(__m128 s, __mmask8 k, __m128 a, __m128 b, int);
VMAXSS __m128 _mm_maskz_max_round_ss(__mmask8 k, __m128 a, __m128 b, int);
MAXSS __m128 _mm_max_ss(__m128 a, __m128 b)

```

**SIMD Floating-Point Exceptions**

Invalid (Including QNaN Source Operand), Denormal

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 3.  
 EVEX-encoded instruction, see Exceptions Type E3.

## MINPD—Minimum of Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 5D /r MINPD xmm1, xmm2/m128	RM	V/V	SSE2	Return the minimum double-precision floating-point values between xmm1 and xmm2/mem
VEX.NDS.128.66.0F.WIG 5D /r VMINPD xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Return the minimum double-precision floating-point values between xmm2 and xmm3/mem.
VEX.NDS.256.66.0F.WIG 5D /r VMINPD ymm1, ymm2, ymm3/m256	RVM	V/V	AVX	Return the minimum packed double-precision floating-point values between ymm2 and ymm3/mem.
EVEX.NDS.128.66.0F.W1 5D /r VMINPD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	FV	V/V	AVX512VL AVX512F	Return the minimum packed double-precision floating-point values between xmm2 and xmm3/m128/m64bcst and store result in xmm1 subject to writemask k1.
EVEX.NDS.256.66.0F.W1 5D /r VMINPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	FV	V/V	AVX512VL AVX512F	Return the minimum packed double-precision floating-point values between ymm2 and ymm3/m256/m64bcst and store result in ymm1 subject to writemask k1.
EVEX.NDS.512.66.0F.W1 5D /r VMINPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{sae}	FV	V/V	AVX512F	Return the minimum packed double-precision floating-point values between zmm2 and zmm3/m512/m64bcst and store result in zmm1 subject to writemask k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
FV	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

### Description

Performs a SIMD compare of the packed double-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 MINPD can be emulated using a sequence of instructions, such as, a comparison followed by AND, ANDN and OR.

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 64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAX\_VL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: The first source operand is a XMM register. The second source operand can be a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper bits (MAX\_VL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAX\_VL-1:128) of the corresponding ZMM register destination are unmodified.

**Operation**

```

MIN(SRC1, SRC2)
{
  IF ((SRC1 = 0.0) and (SRC2 = 0.0)) THEN DEST ← SRC2;
  ELSE IF (SRC1 = SNaN) THEN DEST ← SRC2; FI;
  ELSE IF (SRC2 = SNaN) THEN DEST ← SRC2; FI;
  ELSE IF (SRC1 < SRC2) THEN DEST ← SRC1;
  ELSE DEST ← SRC2;
  FI;
}

```

**VMINPD (EVEX encoded version)**

(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) AND (SRC2 *is memory*)
        THEN
          DEST[i+63:i] ← MIN(SRC1[i+63:i], SRC2[63:0])
        ELSE
          DEST[i+63:i] ← MIN(SRC1[i+63:i], SRC2[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[MAX_VL-1:VL] ← 0

```

**VMINPD (VEX.256 encoded version)**

```

DEST[63:0] ← MIN(SRC1[63:0], SRC2[63:0])
DEST[127:64] ← MIN(SRC1[127:64], SRC2[127:64])
DEST[191:128] ← MIN(SRC1[191:128], SRC2[191:128])
DEST[255:192] ← MIN(SRC1[255:192], SRC2[255:192])

```

**VMINPD (VEX.128 encoded version)**

```

DEST[63:0] ← MIN(SRC1[63:0], SRC2[63:0])
DEST[127:64] ← MIN(SRC1[127:64], SRC2[127:64])
DEST[MAX_VL-1:128] ← 0

```

**MINPD (128-bit Legacy SSE version)**

```

DEST[63:0] ← MIN(SRC1[63:0], SRC2[63:0])
DEST[127:64] ← MIN(SRC1[127:64], SRC2[127:64])
DEST[MAX_VL-1:128] (Unmodified)

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VMINPD __m512d __mm512_min_pd( __m512d a, __m512d b);
VMINPD __m512d __mm512_mask_min_pd( __m512d s, __mmask8 k, __m512d a, __m512d b);
VMINPD __m512d __mm512_maskz_min_pd( __mmask8 k, __m512d a, __m512d b);
VMINPD __m512d __mm512_min_round_pd( __m512d a, __m512d b, int);
VMINPD __m512d __mm512_mask_min_round_pd( __m512d s, __mmask8 k, __m512d a, __m512d b, int);
VMINPD __m512d __mm512_maskz_min_round_pd( __mmask8 k, __m512d a, __m512d b, int);
VMINPD __m256d __mm256_mask_min_pd( __m256d s, __mmask8 k, __m256d a, __m256d b);
VMINPD __m256d __mm256_maskz_min_pd( __mmask8 k, __m256d a, __m256d b);
VMINPD __m128d __mm_mask_min_pd( __m128d s, __mmask8 k, __m128d a, __m128d b);
VMINPD __m128d __mm_maskz_min_pd( __mmask8 k, __m128d a, __m128d b);
VMINPD __m256d __mm256_min_pd( __m256d a, __m256d b);
MINPD __m128d __mm_min_pd( __m128d a, __m128d b);

```

**SIMD Floating-Point Exceptions**

Invalid (including QNaN Source Operand), Denormal

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 2.

EVEX-encoded instruction, see Exceptions Type E2.

## MINPS—Minimum of Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
OF 5D /r MINPS xmm1, xmm2/m128	RM	V/V	SSE	Return the minimum single-precision floating-point values between xmm1 and xmm2/mem.
VEX.NDS.128.OF.WIG 5D /r VMINPS xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Return the minimum single-precision floating-point values between xmm2 and xmm3/mem.
VEX.NDS.256.OF.WIG 5D /r VMINPS ymm1, ymm2, ymm3/m256	RVM	V/V	AVX	Return the minimum single double-precision floating-point values between ymm2 and ymm3/mem.
EVEX.NDS.128.OF.W0 5D /r VMINPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	FV	V/V	AVX512VL AVX512F	Return the minimum packed single-precision floating-point values between xmm2 and xmm3/m128/m32bcst and store result in xmm1 subject to writemask k1.
EVEX.NDS.256.OF.W0 5D /r VMINPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	FV	V/V	AVX512VL AVX512F	Return the minimum packed single-precision floating-point values between ymm2 and ymm3/m256/m32bcst and store result in ymm1 subject to writemask k1.
EVEX.NDS.512.OF.W0 5D /r VMINPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{sae}	FV	V/V	AVX512F	Return the minimum packed single-precision floating-point values between zmm2 and zmm3/m512/m32bcst and store result in zmm1 subject to writemask k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
FV	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

### Description

Performs a SIMD compare of the packed single-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 MINPS can be emulated using a sequence of instructions, such as, a comparison followed by AND, ANDN and OR.

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 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAX\_VL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: The first source operand is a XMM register. The second source operand can be a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper bits (MAX\_VL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAX\_VL-1:128) of the corresponding ZMM register destination are unmodified.

**Operation**

MIN(SRC1, SRC2)

```
{
  IF ((SRC1 = 0.0) and (SRC2 = 0.0)) THEN DEST ← SRC2;
  ELSE IF (SRC1 = SNaN) THEN DEST ← SRC2; FI;
  ELSE IF (SRC2 = SNaN) THEN DEST ← SRC2; FI;
  ELSE IF (SRC1 < SRC2) THEN DEST ← SRC1;
  ELSE DEST ← SRC2;
  FI;
}
```

**VMINPS (EVEX encoded version)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\*

THEN

IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

THEN

DEST[i+31:i] ← MIN(SRC1[i+31:i], SRC2[31:0])

ELSE

DEST[i+31:i] ← MIN(SRC1[i+31:i], SRC2[i+31:i])

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[MAX\_VL-1:VL] ← 0

**VMINPS (VEX.256 encoded version)**

DEST[31:0] ← MIN(SRC1[31:0], SRC2[31:0])

DEST[63:32] ← MIN(SRC1[63:32], SRC2[63:32])

DEST[95:64] ← MIN(SRC1[95:64], SRC2[95:64])

DEST[127:96] ← MIN(SRC1[127:96], SRC2[127:96])

DEST[159:128] ← MIN(SRC1[159:128], SRC2[159:128])

DEST[191:160] ← MIN(SRC1[191:160], SRC2[191:160])

DEST[223:192] ← MIN(SRC1[223:192], SRC2[223:192])

DEST[255:224] ← MIN(SRC1[255:224], SRC2[255:224])

**VMINPS (VEX.128 encoded version)**

DEST[31:0] ← MIN(SRC1[31:0], SRC2[31:0])

DEST[63:32] ← MIN(SRC1[63:32], SRC2[63:32])

DEST[95:64] ← MIN(SRC1[95:64], SRC2[95:64])

DEST[127:96] ← MIN(SRC1[127:96], SRC2[127:96])

DEST[MAX\_VL-1:128] ← 0

**MINPS (128-bit Legacy SSE version)**

DEST[31:0] ← MIN(SRC1[31:0], SRC2[31:0])  
 DEST[63:32] ← MIN(SRC1[63:32], SRC2[63:32])  
 DEST[95:64] ← MIN(SRC1[95:64], SRC2[95:64])  
 DEST[127:96] ← MIN(SRC1[127:96], SRC2[127:96])  
 DEST[MAX\_VL-1:128] (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**

VMINPS \_\_m512 \_\_mm512\_min\_ps( \_\_m512 a, \_\_m512 b);  
 VMINPS \_\_m512 \_\_mm512\_mask\_min\_ps(\_\_m512 s, \_\_mmask16 k, \_\_m512 a, \_\_m512 b);  
 VMINPS \_\_m512 \_\_mm512\_maskz\_min\_ps(\_\_mmask16 k, \_\_m512 a, \_\_m512 b);  
 VMINPS \_\_m512 \_\_mm512\_min\_round\_ps( \_\_m512 a, \_\_m512 b, int);  
 VMINPS \_\_m512 \_\_mm512\_mask\_min\_round\_ps(\_\_m512 s, \_\_mmask16 k, \_\_m512 a, \_\_m512 b, int);  
 VMINPS \_\_m512 \_\_mm512\_maskz\_min\_round\_ps( \_\_mmask16 k, \_\_m512 a, \_\_m512 b, int);  
 VMINPS \_\_m256 \_\_mm256\_mask\_min\_ps(\_\_m256 s, \_\_mmask8 k, \_\_m256 a, \_\_m256 b);  
 VMINPS \_\_m256 \_\_mm256\_maskz\_min\_ps(\_\_mmask8 k, \_\_m256 a, \_\_m256 b);  
 VMINPS \_\_m128 \_\_mm\_mask\_min\_ps(\_\_m128 s, \_\_mmask8 k, \_\_m128 a, \_\_m128 b);  
 VMINPS \_\_m128 \_\_mm\_maskz\_min\_ps( \_\_mmask8 k, \_\_m128 a, \_\_m128 b);  
 VMINPS \_\_m256 \_\_mm256\_min\_ps( \_\_m256 a, \_\_m256 b);  
 MINPS \_\_m128 \_\_mm\_min\_ps( \_\_m128 a, \_\_m128 b);

**SIMD Floating-Point Exceptions**

Invalid (including QNaN Source Operand), Denormal

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 2.

EVEX-encoded instruction, see Exceptions Type E2.



## MINSND—Return Minimum Scalar Double-Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 5D /r MINSND xmm1, xmm2/m64	RM	V/V	SSE2	Return the minimum scalar double-precision floating-point value between xmm2/m64 and xmm1.
VEX.NDS.128.F2.0F.WIG 5D /r VMINSND xmm1, xmm2, xmm3/m64	RVM	V/V	AVX	Return the minimum scalar double-precision floating-point value between xmm3/m64 and xmm2.
EVEX.NDS.LIG.F2.0F.W1 5D /r VMINSND xmm1 {k1}{z}, xmm2, xmm3/m64{sae}	T1S	V/V	AVX512F	Return the minimum scalar double-precision floating-point value between xmm3/m64 and xmm2.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
T1S	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

### Description

Compares the low double-precision floating-point values in the first source operand and the second source operand, and returns the minimum value to the low quadword of the destination operand. When the source operand is a memory operand, only the 64 bits are accessed.

If the values being compared are both 0.0s (of either sign), the value in the second source operand is returned. If a value in the second source operand is an SNaN, then SNaN is returned 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 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 source) be returned, the action of MINSND can be emulated using a sequence of instructions, such as, a comparison followed by AND, ANDN and OR.

The second source operand can be an XMM register or a 64-bit memory location. The first source and destination operands are XMM registers.

128-bit Legacy SSE version: The destination and first source operand are the same. Bits (MAX\_VL-1:64) of the corresponding destination register remain unchanged.

VEX.128 and EVEX encoded version: Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (MAX\_VL-1:128) of the destination register are zeroed.

EVEX encoded version: The low quadword element of the destination operand is updated according to the writemask.

Software should ensure VMINSND is encoded with VEX.L=0. Encoding VMINSND with VEX.L=1 may encounter unpredictable behavior across different processor generations.

**Operation**

```

MIN(SRC1, SRC2)
{
  IF ((SRC1 = 0.0) and (SRC2 = 0.0)) THEN DEST ← SRC2;
  ELSE IF (SRC1 = SNaN) THEN DEST ← SRC2; FI;
  ELSE IF (SRC2 = SNaN) THEN DEST ← SRC2; FI;
  ELSE IF (SRC1 < SRC2) THEN DEST ← SRC1;
  ELSE DEST ← SRC2;
  FI;
}

```

**MINSD (EVEX encoded version)**

```

IF k1[0] or *no writemask*
  THEN  DEST[63:0] ← MIN(SRC1[63:0], SRC2[63:0])
  ELSE
    IF *merging-masking*           ; merging-masking
      THEN *DEST[63:0] remains unchanged*
    ELSE                             ; zeroing-masking
      THEN DEST[63:0] ← 0
    FI;
  FI;
DEST[127:64] ← SRC1[127:64]
DEST[MAX_VL-1:128] ← 0

```

**MINSD (VEX.128 encoded version)**

```

DEST[63:0] ← MIN(SRC1[63:0], SRC2[63:0])
DEST[127:64] ← SRC1[127:64]
DEST[MAX_VL-1:128] ← 0

```

**MINSD (128-bit Legacy SSE version)**

```

DEST[63:0] ← MIN(SRC1[63:0], SRC2[63:0])
DEST[MAX_VL-1:64] (Unmodified)

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VMINSD __m128d __mm_min_round_sd(__m128d a, __m128d b, int);
VMINSD __m128d __mm_mask_min_round_sd(__m128d s, __mmask8 k, __m128d a, __m128d b, int);
VMINSD __m128d __mm_maskz_min_round_sd(__mmask8 k, __m128d a, __m128d b, int);
MINSD __m128d __mm_min_sd(__m128d a, __m128d b)

```

**SIMD Floating-Point Exceptions**

Invalid (including QNaN Source Operand), Denormal

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 3.  
 EVEX-encoded instruction, see Exceptions Type E3.

## MINSS—Return Minimum Scalar Single-Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 5D /r MINSS xmm1,xmm2/m32	RM	V/V	SSE	Return the minimum scalar single-precision floating-point value between xmm2/m32 and xmm1.
VEX.NDS.128.F3.0F.WIG 5D /r VMINSS xmm1,xmm2, xmm3/m32	RVM	V/V	AVX	Return the minimum scalar single-precision floating-point value between xmm3/m32 and xmm2.
EVEX.NDS.LIG.F3.0F.W0 5D /r VMINSS xmm1 {k1}{z}, xmm2, xmm3/m32{sae}	T1S	V/V	AVX512F	Return the minimum scalar single-precision floating-point value between xmm3/m32 and xmm2.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
T1S	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

### Description

Compares the low single-precision floating-point values in the first source operand and the second source operand and returns the minimum value to the low doubleword of the destination operand.

If the values being compared are both 0.0s (of either sign), the value in the second source operand is returned. If a value in the second operand is an SNaN, that SNaN is returned 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 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 in either source operand be returned, the action of MINSR can be emulated using a sequence of instructions, such as, a comparison followed by AND, ANDN and OR.

The second source operand can be an XMM register or a 32-bit memory location. The first source and destination operands are XMM registers.

128-bit Legacy SSE version: The destination and first source operand are the same. Bits (MAX\_VL:32) of the corresponding destination register remain unchanged.

VEX.128 and EVEX encoded version: The first source operand is an xmm register encoded by (E)VEX.vvvv. Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (MAX\_VL-1:128) of the destination register are zeroed.

EVEX encoded version: The low doubleword element of the destination operand is updated according to the writemask.

Software should ensure VMINSS is encoded with VEX.L=0. Encoding VMINSS with VEX.L=1 may encounter unpredictable behavior across different processor generations.

**Operation**

```

MIN(SRC1, SRC2)
{
  IF ((SRC1 = 0.0) and (SRC2 = 0.0)) THEN DEST ← SRC2;
  ELSE IF (SRC1 = SNaN) THEN DEST ← SRC2; FI;
  ELSE IF (SRC2 = SNaN) THEN DEST ← SRC2; FI;
  ELSE IF (SRC1 < SRC2) THEN DEST ← SRC1;
  ELSE DEST ← SRC2;
  FI;
}

```

**MINSS (EVEX encoded version)**

```

IF k1[0] or *no writemask*
  THEN  DEST[31:0] ← MIN(SRC1[31:0], SRC2[31:0])
  ELSE
    IF *merging-masking*           ; merging-masking
      THEN *DEST[31:0] remains unchanged*
    ELSE                             ; zeroing-masking
      THEN DEST[31:0] ← 0
    FI;
  FI;
DEST[127:32] ← SRC1[127:32]
DEST[MAX_VL-1:128] ← 0

```

**VMINSS (VEX.128 encoded version)**

```

DEST[31:0] ← MIN(SRC1[31:0], SRC2[31:0])
DEST[127:32] ← SRC1[127:32]
DEST[MAX_VL-1:128] ← 0

```

**MINSS (128-bit Legacy SSE version)**

```

DEST[31:0] ← MIN(SRC1[31:0], SRC2[31:0])
DEST[MAX_VL-1:128] (Unmodified)

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VMINSS __m128 _mm_min_round_ss(__m128 a, __m128 b, int);
VMINSS __m128 _mm_mask_min_round_ss(__m128 s, __mmask8 k, __m128 a, __m128 b, int);
VMINSS __m128 _mm_maskz_min_round_ss(__mmask8 k, __m128 a, __m128 b, int);
MINSS __m128 _mm_min_ss(__m128 a, __m128 b)

```

**SIMD Floating-Point Exceptions**

Invalid (Including QNaN Source Operand), Denormal

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 2.

EVEX-encoded instruction, see Exceptions Type E2.

## MOVAPD—Move Aligned Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 28 /r MOVAPD xmm1, xmm2/m128	RM	V/V	SSE2	Move aligned packed double-precision floating-point values from xmm2/mem to xmm1.
66 0F 29 /r MOVAPD xmm2/m128, xmm1	MR	V/V	SSE2	Move aligned packed double-precision floating-point values from xmm1 to xmm2/mem.
VEX.128.66.0F.WIG 28 /r VMOVAPD xmm1, xmm2/m128	RM	V/V	AVX	Move aligned packed double-precision floating-point values from xmm2/mem to xmm1.
VEX.128.66.0F.WIG 29 /r VMOVAPD xmm2/m128, xmm1	MR	V/V	AVX	Move aligned packed double-precision floating-point values from xmm1 to xmm2/mem.
VEX.256.66.0F.WIG 28 /r VMOVAPD ymm1, ymm2/m256	RM	V/V	AVX	Move aligned packed double-precision floating-point values from ymm2/mem to ymm1.
VEX.256.66.0F.WIG 29 /r VMOVAPD ymm2/m256, ymm1	MR	V/V	AVX	Move aligned packed double-precision floating-point values from ymm1 to ymm2/mem.
EVEX.128.66.0F.W1 28 /r VMOVAPD xmm1 {k1}{z}, xmm2/m128	FVM-RM	V/V	AVX512VL AVX512F	Move aligned packed double-precision floating-point values from xmm2/m128 to xmm1 using writemask k1.
EVEX.256.66.0F.W1 28 /r VMOVAPD ymm1 {k1}{z}, ymm2/m256	FVM-RM	V/V	AVX512VL AVX512F	Move aligned packed double-precision floating-point values from ymm2/m256 to ymm1 using writemask k1.
EVEX.512.66.0F.W1 28 /r VMOVAPD zmm1 {k1}{z}, zmm2/m512	FVM-RM	V/V	AVX512F	Move aligned packed double-precision floating-point values from zmm2/m512 to zmm1 using writemask k1.
EVEX.128.66.0F.W1 29 /r VMOVAPD xmm2/m128 {k1}{z}, xmm1	FVM-MR	V/V	AVX512VL AVX512F	Move aligned packed double-precision floating-point values from xmm1 to xmm2/m128 using writemask k1.
EVEX.256.66.0F.W1 29 /r VMOVAPD ymm2/m256 {k1}{z}, ymm1	FVM-MR	V/V	AVX512VL AVX512F	Move aligned packed double-precision floating-point values from ymm1 to ymm2/m256 using writemask k1.
EVEX.512.66.0F.W1 29 /r VMOVAPD zmm2/m512 {k1}{z}, zmm1	FVM-MR	V/V	AVX512F	Move aligned packed double-precision floating-point values from zmm1 to zmm2/m512 using writemask k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
FVM-RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
FVM-MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

## Description

Moves 2, 4 or 8 double-precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load an XMM, YMM or ZMM register from an 128-bit, 256-bit or 512-bit memory location, to store the contents of an XMM, YMM or ZMM register into a 128-bit, 256-bit or 512-bit memory location, or to move data between two XMM, two YMM or two ZMM registers.

When the source or destination operand is a memory operand, the operand must be aligned on a 16-byte (128-bit versions), 32-byte (256-bit version) or 64-byte (EVEX.512 encoded version) boundary or a general-protection exception (#GP) will be generated. For EVEX encoded versions, the operand must be aligned to the size of the memory operand. To move double-precision floating-point values to and from unaligned memory locations, use the VMOVUPD instruction.

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

EVEX.512 encoded version:

Moves 512 bits of packed double-precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load a ZMM register from a 512-bit float64 memory location, to store the contents of a ZMM register into a 512-bit float64 memory location, or to move data between two ZMM registers. When the source or destination operand is a memory operand, the operand must be aligned on a 64-byte boundary or a general-protection exception (#GP) will be generated. To move single-precision floating-point values to and from unaligned memory locations, use the VMOVUPD instruction.

VEX.256 and EVEX.256 encoded versions:

Moves 256 bits of packed double-precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load a YMM register from a 256-bit memory location, to store the contents of a YMM register into a 256-bit memory location, or to move data between two YMM registers. When the source or destination operand is a memory operand, the operand must be aligned on a 32-byte boundary or a general-protection exception (#GP) will be generated. To move double-precision floating-point values to and from unaligned memory locations, use the VMOVUPD instruction.

128-bit versions:

Moves 128 bits of packed double-precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load an XMM register from a 128-bit memory location, to store the contents of an XMM register into a 128-bit memory location, or to move data between two XMM registers. When the source or destination operand is a memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated. To move single-precision floating-point values to and from unaligned memory locations, use the VMOVUPD instruction.

128-bit Legacy SSE version: Bits (MAX\_VL-1:128) of the corresponding ZMM destination register remain unchanged.

(E)VEX.128 encoded version: Bits (MAX\_VL-1:128) of the destination ZMM register destination are zeroed.

## Operation

### VMOVAPD (EVEX encoded versions, register-copy form)

(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[MAX\_VL-1:VL] ← 0

**VMOVAPD (EVEX encoded versions, store-form)**

(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

ELSE \*DEST[i+63:i] remains unchanged\* ; merging-masking

FI;

ENDFOR;

**VMOVAPD (EVEX encoded versions, load-form)**

(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[MAX\_VL-1:VL] ← 0

**VMOVAPD (VEX.256 encoded version, load - and register copy)**

DEST[255:0] ← SRC[255:0]

DEST[MAX\_VL-1:256] ← 0

**VMOVAPD (VEX.256 encoded version, store-form)**

DEST[255:0] ← SRC[255:0]

**VMOVAPD (VEX.128 encoded version, load - and register copy)**

DEST[127:0] ← SRC[127:0]

DEST[MAX\_VL-1:128] ← 0

**MOVAPD (128-bit load- and register-copy- form Legacy SSE version)**

DEST[127:0] ← SRC[127:0]

DEST[MAX\_VL-1:128] (Unmodified)

**(V)MOVAPD (128-bit store-form version)**

DEST[127:0] ← SRC[127:0]

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VMOVAPD __m512d __mm512_load_pd( void * m);
VMOVAPD __m512d __mm512_mask_load_pd(__m512d s, __mmask8 k, void * m);
VMOVAPD __m512d __mm512_maskz_load_pd( __mmask8 k, void * m);
VMOVAPD void __mm512_store_pd( void * d, __m512d a);
VMOVAPD void __mm512_mask_store_pd( void * d, __mmask8 k, __m512d a);
VMOVAPD __m256d __mm256_mask_load_pd(__m256d s, __mmask8 k, void * m);
VMOVAPD __m256d __mm256_maskz_load_pd( __mmask8 k, void * m);
VMOVAPD void __mm256_mask_store_pd( void * d, __mmask8 k, __m256d a);
VMOVAPD __m128d __mm_mask_load_pd(__m128d s, __mmask8 k, void * m);
VMOVAPD __m128d __mm_maskz_load_pd( __mmask8 k, void * m);
VMOVAPD void __mm_mask_store_pd( void * d, __mmask8 k, __m128d a);
MOVAPD __m256d __mm256_load_pd( double * p);
MOVAPD void __mm256_store_pd(double * p, __m256d a);
MOVAPD __m128d __mm_load_pd( double * p);
MOVAPD void __mm_store_pd(double * p, __m128d a);

```

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type1.SSE2;

EVEX-encoded instruction, see Exceptions Type E1.

#UD                    If EVEX.vvvv != 1111B or VEX.vvvv != 1111B.



## MOVAPS—Move Aligned Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
OF 28 /r MOVAPS xmm1, xmm2/m128	RM	V/V	SSE	Move aligned packed single-precision floating-point values from xmm2/mem to xmm1.
OF 29 /r MOVAPS xmm2/m128, xmm1	MR	V/V	SSE	Move aligned packed single-precision floating-point values from xmm1 to xmm2/mem.
VEX.128.OF.WIG 28 /r VMOVAPS xmm1, xmm2/m128	RM	V/V	AVX	Move aligned packed single-precision floating-point values from xmm2/mem to xmm1.
VEX.128.OF.WIG 29 /r VMOVAPS xmm2/m128, xmm1	MR	V/V	AVX	Move aligned packed single-precision floating-point values from xmm1 to xmm2/mem.
VEX.256.OF.WIG 28 /r VMOVAPS ymm1, ymm2/m256	RM	V/V	AVX	Move aligned packed single-precision floating-point values from ymm2/mem to ymm1.
VEX.256.OF.WIG 29 /r VMOVAPS ymm2/m256, ymm1	MR	V/V	AVX	Move aligned packed single-precision floating-point values from ymm1 to ymm2/mem.
EVEX.128.OF.W0 28 /r VMOVAPS xmm1 {k1}{z}, xmm2/m128	FVM-RM	V/V	AVX512VL AVX512F	Move aligned packed single-precision floating-point values from xmm2/m128 to xmm1 using writemask k1.
EVEX.256.OF.W0 28 /r VMOVAPS ymm1 {k1}{z}, ymm2/m256	FVM-RM	V/V	AVX512VL AVX512F	Move aligned packed single-precision floating-point values from ymm2/m256 to ymm1 using writemask k1.
EVEX.512.OF.W0 28 /r VMOVAPS zmm1 {k1}{z}, zmm2/m512	FVM-RM	V/V	AVX512F	Move aligned packed single-precision floating-point values from zmm2/m512 to zmm1 using writemask k1.
EVEX.128.OF.W0 29 /r VMOVAPS xmm2/m128 {k1}{z}, xmm1	FVM-MR	V/V	AVX512VL AVX512F	Move aligned packed single-precision floating-point values from xmm1 to xmm2/m128 using writemask k1.
EVEX.256.OF.W0 29 /r VMOVAPS ymm2/m256 {k1}{z}, ymm1	FVM-MR	V/V	AVX512VL AVX512F	Move aligned packed single-precision floating-point values from ymm1 to ymm2/m256 using writemask k1.
EVEX.512.OF.W0 29 /r VMOVAPS zmm2/m512 {k1}{z}, zmm1	FVM-MR	V/V	AVX512F	Move aligned packed single-precision floating-point values from zmm1 to zmm2/m512 using writemask k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
FVM-RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
FVM-MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

### Description

Moves 4, 8 or 16 single-precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load an XMM, YMM or ZMM register from an 128-bit, 256-bit or 512-bit memory location, to store the contents of an XMM, YMM or ZMM register into a 128-bit, 256-bit or 512-bit memory location, or to move data between two XMM, two YMM or two ZMM registers.

When the source or destination operand is a memory operand, the operand must be aligned on a 16-byte (128-bit version), 32-byte (VEX.256 encoded version) or 64-byte (EVEX.512 encoded version) boundary or a general-protection exception (#GP) will be generated. For EVEX.512 encoded versions, the operand must be aligned to the size of the memory operand. To move single-precision floating-point values to and from unaligned memory locations, use the VMOVUPS instruction.

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

EVEX.512 encoded version:

Moves 512 bits of packed single-precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load a ZMM register from a 512-bit float32 memory location, to store the contents of a ZMM register into a float32 memory location, or to move data between two ZMM registers. When the source or destination operand is a memory operand, the operand must be aligned on a 64-byte boundary or a general-protection exception (#GP) will be generated. To move single-precision floating-point values to and from unaligned memory locations, use the VMOVUPS instruction.

VEX.256 and EVEX.256 encoded version:

Moves 256 bits of packed single-precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load a YMM register from a 256-bit memory location, to store the contents of a YMM register into a 256-bit memory location, or to move data between two YMM registers. When the source or destination operand is a memory operand, the operand must be aligned on a 32-byte boundary or a general-protection exception (#GP) will be generated.

128-bit versions:

Moves 128 bits of packed single-precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load an XMM register from a 128-bit memory location, to store the contents of an XMM register into a 128-bit memory location, or to move data between two XMM registers. When the source or destination operand is a memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated. To move single-precision floating-point values to and from unaligned memory locations, use the VMOVUPS instruction.

128-bit Legacy SSE version: Bits (MAX\_VL-1:128) of the corresponding ZMM destination register remain unchanged.

(E)VEX.128 encoded version: Bits (MAX\_VL-1:128) of the destination ZMM register are zeroed.

## Operation

### VMOVAPS (EVEX encoded versions, register-copy form)

(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[MAX\_VL-1:VL] ← 0

### VMOVAPS (EVEX encoded versions, store-form)

(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 \*DEST[i+31:i] remains unchanged\* ; merging-masking

  FI;

ENDFOR;

**VMOVAPS (EVEX encoded versions, load-form)**

(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[MAX\_VL-1:VL] ← 0

**VMOVAPS (VEX.256 encoded version, load - and register copy)**

DEST[255:0] ← SRC[255:0]

DEST[MAX\_VL-1:256] ← 0

**VMOVAPS (VEX.256 encoded version, store-form)**

DEST[255:0] ← SRC[255:0]

**VMOVAPS (VEX.128 encoded version, load - and register copy)**

DEST[127:0] ← SRC[127:0]

DEST[MAX\_VL-1:128] ← 0

**MOVAPS (128-bit load- and register-copy- form Legacy SSE version)**

DEST[127:0] ← SRC[127:0]

DEST[MAX\_VL-1:128] (Unmodified)

**(V)MOVAPS (128-bit store-form version)**

DEST[127:0] ← SRC[127:0]

**Intel C/C++ Compiler Intrinsic Equivalent**

VMOVAPS \_\_m512 \_\_mm512\_load\_ps( void \* m);

VMOVAPS \_\_m512 \_\_mm512\_mask\_load\_ps(\_\_m512 s, \_\_mmask16 k, void \* m);

VMOVAPS \_\_m512 \_\_mm512\_maskz\_load\_ps( \_\_mmask16 k, void \* m);

VMOVAPS void \_\_mm512\_store\_ps( void \* d, \_\_m512 a);

VMOVAPS void \_\_mm512\_mask\_store\_ps( void \* d, \_\_mmask16 k, \_\_m512 a);

VMOVAPS \_\_m256 \_\_mm256\_mask\_load\_ps(\_\_m256 a, \_\_mmask8 k, void \* s);

VMOVAPS \_\_m256 \_\_mm256\_maskz\_load\_ps( \_\_mmask8 k, void \* s);

VMOVAPS void \_\_mm256\_mask\_store\_ps( void \* d, \_\_mmask8 k, \_\_m256 a);

VMOVAPS \_\_m128 \_\_mm\_mask\_load\_ps(\_\_m128 a, \_\_mmask8 k, void \* s);

VMOVAPS \_\_m128 \_\_mm\_maskz\_load\_ps( \_\_mmask8 k, void \* s);

VMOVAPS void \_\_mm\_mask\_store\_ps( void \* d, \_\_mmask8 k, \_\_m128 a);

MOVAPS \_\_m256 \_\_mm256\_load\_ps( float \* p);

MOVAPS void \_\_mm256\_store\_ps(float \* p, \_\_m256 a);

MOVAPS \_\_m128 \_\_mm\_load\_ps( float \* p);

MOVAPS void \_\_mm\_store\_ps(float \* p, \_\_m128 a);

**SIMD Floating-Point Exceptions**

None



### Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 1.SSE; additionally  
#UD If VEX.vvvv != 1111B.  
EVEX-encoded instruction, see Exceptions Type E1.

## MOVD/MOVQ—Move Doubleword/Move Quadword

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
0F 6E /r MOVD <i>mm, r/m32</i>	RM	V/V	MMX	Move doubleword from <i>r/m32</i> to <i>mm</i> .
REX.W + 0F 6E /r MOVQ <i>mm, r/m64</i>	RM	V/N.E.	MMX	Move quadword from <i>r/m64</i> to <i>mm</i> .
0F 7E /r MOVD <i>r/m32, mm</i>	MR	V/V	MMX	Move doubleword from <i>mm</i> to <i>r/m32</i> .
REX.W + 0F 7E /r MOVQ <i>r/m64, mm</i>	MR	V/N.E.	MMX	Move quadword from <i>mm</i> to <i>r/m64</i> .
66 0F 6E /r MOVD <i>xmm, r/m32</i>	RM	V/V	SSE2	Move doubleword from <i>r/m32</i> to <i>xmm</i> .
66 REX.W 0F 6E /r MOVQ <i>xmm, r/m64</i>	RM	V/N.E.	SSE2	Move quadword from <i>r/m64</i> to <i>xmm</i> .
66 0F 7E /r MOVD <i>r/m32, xmm</i>	MR	V/V	SSE2	Move doubleword from <i>xmm</i> register to <i>r/m32</i> .
66 REX.W 0F 7E /r MOVQ <i>r/m64, xmm</i>	MR	V/N.E.	SSE2	Move quadword from <i>xmm</i> register to <i>r/m64</i> .
VEX.128.66.0F.W0 6E / VMOVD <i>xmm1, r32/m32</i>	RM	V/V	AVX	Move doubleword from <i>r/m32</i> to <i>xmm1</i> .
VEX.128.66.0F.W1 6E /r VMOVQ <i>xmm1, r64/m64</i>	RM	V/N.E. <sup>1</sup>	AVX	Move quadword from <i>r/m64</i> to <i>xmm1</i> .
VEX.128.66.0F.W0 7E /r VMOVD <i>r32/m32, xmm1</i>	MR	V/V	AVX	Move doubleword from <i>xmm1</i> register to <i>r/m32</i> .
VEX.128.66.0F.W1 7E /r VMOVQ <i>r64/m64, xmm1</i>	MR	V/N.E. <sup>1</sup>	AVX	Move quadword from <i>xmm1</i> register to <i>r/m64</i> .
EVEX.128.66.0F.W0 6E /r VMOVD <i>xmm1, r32/m32</i>	T1S-RM	V/V	AVX512F	Move doubleword from <i>r/m32</i> to <i>xmm1</i> .
EVEX.128.66.0F.W1 6E /r VMOVQ <i>xmm1, r64/m64</i>	T1S-RM	V/N.E. <sup>1</sup>	AVX512F	Move quadword from <i>r/m64</i> to <i>xmm1</i> .
EVEX.128.66.0F.W0 7E /r VMOVD <i>r32/m32, xmm1</i>	T1S-MR	V/V	AVX512F	Move doubleword from <i>xmm1</i> register to <i>r/m32</i> .
EVEX.128.66.0F.W1 7E /r VMOVQ <i>r64/m64, xmm1</i>	T1S-MR	V/N.E. <sup>1</sup>	AVX512F	Move quadword from <i>xmm1</i> register to <i>r/m64</i> .

### NOTES:

1. For this specific instruction, VEX.W/EVEX.W in non-64 bit is ignored; the instructions behaves as if the W0 version is used.

## Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
T1S-RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
T1S-MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

## Description

Copies a doubleword from the source operand (second operand) to the destination operand (first operand). The source and destination operands can be general-purpose registers, MMX technology registers, XMM registers, or 32-bit memory locations. This instruction can be used to move a doubleword to and from the low doubleword of an MMX technology register and a general-purpose register or a 32-bit memory location, or to and from the low doubleword of an XMM register and a general-purpose register or a 32-bit memory location. The instruction cannot be used to transfer data between MMX technology registers, between XMM registers, between general-purpose registers, or between memory locations.

When the destination operand is an MMX technology register, the source operand is written to the low doubleword of the register, and the register is zero-extended to 64 bits. When the destination operand is an XMM register, the source operand is written to the low doubleword of the register, and the register is zero-extended to 128 bits.

In 64-bit mode, the instruction's default operation size is 32 bits. Use of the REX.R prefix permits access to additional registers (R8-R15). Use of the REX.W prefix promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

MOVD/Q with XMM destination:

Moves a dword/qword integer from the source operand and stores it in the low 32/64-bits of the destination XMM register. The upper bits of the destination are zeroed. The source operand can be a 32/64-bit register or 32/64-bit memory location.

128-bit Legacy SSE version: Bits (MAX\_VL-1:128) of the corresponding YMM destination register remain unchanged. Qword operation requires the use of REX.W=1.

VEX.128 encoded version: Bits (MAX\_VL-1:128) of the destination register are zeroed. Qword operation requires the use of VEX.W=1.

EVEX.128 encoded version: Bits (MAX\_VL-1:128) of the destination register are zeroed. Qword operation requires the use of EVEX.W=1.

MOVD/Q with 32/64 reg/mem destination:

Stores the low dword/qword of the source XMM register to 32/64-bit memory location or general-purpose register. Qword operation requires the use of REX.W=1, VEX.W=1, or EVEX.W=1.

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

If VMOVD or VMOVQ is encoded with VEX.L= 1, an attempt to execute the instruction encoded with VEX.L= 1 will cause an #UD exception.

## Operation

**MOVD (when destination operand is MMX technology register)**

```
DEST[31:0] ← SRC;
DEST[63:32] ← 00000000H;
```

**MOVD (when destination operand is XMM register)**

```
DEST[31:0] ← SRC;
DEST[127:32] ← 000000000000000000000000H;
DEST[VLMAX-1:128] (Unmodified)
```

**MOVD (when source operand is MMX technology or XMM register)**

DEST ← SRC[31:0];

**VMOVD (VEX-encoded version when destination is an XMM register)**

DEST[31:0] ← SRC[31:0]

DEST[VLMAX-1:32] ← 0

**MOVQ (when destination operand is XMM register)**

DEST[63:0] ← SRC[63:0];

DEST[127:64] ← 0000000000000000H;

DEST[VLMAX-1:128] (Unmodified)

**MOVQ (when destination operand is r/m64)**

DEST[63:0] ← SRC[63:0];

**MOVQ (when source operand is XMM register or r/m64)**

DEST ← SRC[63:0];

**VMOVQ (VEX-encoded version when destination is an XMM register)**

DEST[63:0] ← SRC[63:0]

DEST[VLMAX-1:64] ← 0

**VMOVD (EVEX-encoded version when destination is an XMM register)**

DEST[31:0] ← SRC[31:0]

DEST[511:32] ← 0H

**VMOVQ (EVEX-encoded version when destination is an XMM register)**

DEST[63:0] ← SRC[63:0]

DEST[511:64] ← 0H

**Intel C/C++ Compiler Intrinsic Equivalent**MOVD: `__m64 _mm_cvtsi32_si64 (int i)`MOVD: `int _mm_cvtsi64_si32 (__m64 m)`MOVD: `__m128i _mm_cvtsi32_si128 (int a)`MOVD: `int _mm_cvtsi128_si32 (__m128i a)`MOVQ: `__int64 _mm_cvtsi128_si64(__m128i);`MOVQ: `__m128i _mm_cvtsi64_si128(__int64);`VMOVD `__m128i _mm_cvtsi32_si128( int);`VMOVD `int _mm_cvtsi128_si32( __m128i);`VMOVQ `__m128i _mm_cvtsi64_si128 (__int64);`VMOVQ `__int64 _mm_cvtsi128_si64(__m128i);`VMOVQ `__m128i _mm_load_epi64( __m128i * s);`VMOVQ `void _mm_store_epi64( __m128i * d, __m128i s);`**Flags Affected**

None

**SIMD Floating-Point Exceptions**

None



## Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 5.

EVEX-encoded instruction, see Exceptions Type E9NF.

#UD                    If VEX.L = 1.  
                          If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.



## MOVDDUP—Replicate Double FP Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 12 /r MOVDDUP xmm1, xmm2/m64	RM	V/V	SSE3	Move double-precision floating-point value from xmm2/m64 and duplicate into xmm1.
VEX.128.F2.0F.WIG 12 /r VMOVDDUP xmm1, xmm2/m64	RM	V/V	AVX	Move double-precision floating-point value from xmm2/m64 and duplicate into xmm1.
VEX.256.F2.0F.WIG 12 /r VMOVDDUP ymm1, ymm2/m256	RM	V/V	AVX	Move even index double-precision floating-point values from ymm2/mem and duplicate each element into ymm1.
EVEX.128.F2.0F.W1 12 /r VMOVDDUP xmm1 {k1}{z}, xmm2/m64	DUP-RM	V/V	AVX512VL AVX512F	Move double-precision floating-point value from xmm2/m64 and duplicate each element into xmm1 subject to writemask k1.
EVEX.256.F2.0F.W1 12 /r VMOVDDUP ymm1 {k1}{z}, ymm2/m256	DUP-RM	V/V	AVX512VL AVX512F	Move even index double-precision floating-point values from ymm2/m256 and duplicate each element into ymm1 subject to writemask k1.
EVEX.512.F2.0F.W1 12 /r VMOVDDUP zmm1 {k1}{z}, zmm2/m512	DUP-RM	V/V	AVX512F	Move even index double-precision floating-point values from zmm2/m512 and duplicate each element into zmm1 subject to writemask k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
DUP-RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

For 256-bit or higher versions: Duplicates even-indexed double-precision floating-point values from the source operand (the second operand) and into adjacent pair and store to the destination operand (the first operand).

For 128-bit versions: Duplicates the low double-precision floating-point value from the source operand (the second operand) and store to the destination operand (the first operand).

128-bit Legacy SSE version: Bits (MAX\_VL-1:128) of the corresponding destination register are unchanged. The source operand is XMM register or a 64-bit memory location.

VEX.128 and EVEX.128 encoded version: Bits (MAX\_VL-1:128) of the destination register are zeroed. The source operand is XMM register or a 64-bit memory location. The destination is updated conditionally under the writemask for EVEX version.

VEX.256 and EVEX.256 encoded version: Bits (MAX\_VL-1:256) of the destination register are zeroed. The source operand is YMM register or a 256-bit memory location. The destination is updated conditionally under the writemask for EVEX version.

EVEX.512 encoded version: The destination is updated according to the writemask. The source operand is ZMM register or a 512-bit memory location.

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

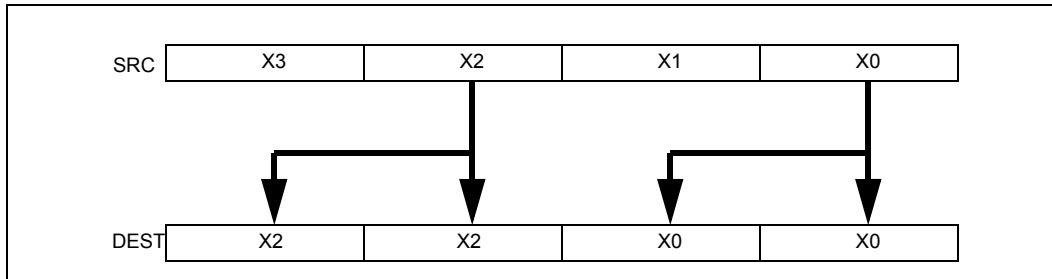


Figure 4-2. VMOVDDUP Operation

**Operation****VMOVDDUP (EVEX encoded versions)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

TMP\_SRC[63:0] ← SRC[63:0]

TMP\_SRC[127:64] ← SRC[63:0]

IF VL ≥ 256

    TMP\_SRC[191:128] ← SRC[191:128]

    TMP\_SRC[255:192] ← SRC[191:128]

FI;

IF VL ≥ 512

    TMP\_SRC[319:256] ← SRC[319:256]

    TMP\_SRC[383:320] ← SRC[319:256]

    TMP\_SRC[477:384] ← SRC[477:384]

    TMP\_SRC[511:484] ← SRC[477:384]

FI;

FOR j ← 0 TO KL-1

    i ← j \* 64

    IF k1[j] OR \*no writemask\*

        THEN DEST[i+63:i] ← TMP\_SRC[i+63:i]

    ELSE

        IF \*merging-masking\* ; merging-masking

            THEN \*DEST[i+63:i] remains unchanged\*

        ELSE ; zeroing-masking

            DEST[i+63:i] ← 0 ; zeroing-masking

    FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VMOVDDUP (VEX.256 encoded version)**

DEST[63:0] ← SRC[63:0]

DEST[127:64] ← SRC[63:0]

DEST[191:128] ← SRC[191:128]

DEST[255:192] ← SRC[191:128]

DEST[MAX\_VL-1:256] ← 0

**VMOVDDUP (VEX.128 encoded version)**

DEST[63:0] ← SRC[63:0]

DEST[127:64] ← SRC[63:0]

DEST[MAX\_VL-1:128] ← 0

**MOVDDUP (128-bit Legacy SSE version)**

DEST[63:0] ← SRC[63:0]

DEST[127:64] ← SRC[63:0]

DEST[MAX\_VL-1:128] (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**

VMOVDDUP \_\_m512d \_\_mm512\_movedup\_pd(\_\_m512d a);

VMOVDDUP \_\_m512d \_\_mm512\_mask\_movedup\_pd(\_\_m512d s, \_\_mmask8 k, \_\_m512d a);

VMOVDDUP \_\_m512d \_\_mm512\_maskz\_movedup\_pd(\_\_mmask8 k, \_\_m512d a);

VMOVDDUP \_\_m256d \_\_mm256\_mask\_movedup\_pd(\_\_m256d s, \_\_mmask8 k, \_\_m256d a);

VMOVDDUP \_\_m256d \_\_mm256\_maskz\_movedup\_pd(\_\_mmask8 k, \_\_m256d a);

VMOVDDUP \_\_m128d \_\_mm\_mask\_movedup\_pd(\_\_m128d s, \_\_mmask8 k, \_\_m128d a);

VMOVDDUP \_\_m128d \_\_mm\_maskz\_movedup\_pd(\_\_mmask8 k, \_\_m128d a);

MOVDDUP \_\_m256d \_\_mm256\_movedup\_pd(\_\_m256d a);

MOVDDUP \_\_m128d \_\_mm\_movedup\_pd(\_\_m128d a);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 5;

EVEX-encoded instruction, see Exceptions Type E5NF.

#UD If EVEX.vvvv != 1111B or VEX.vvvv != 1111B.

## MOVDQA, VMOVDQA32/64—Move Aligned Packed Integer Values

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 6F /r MOVDQA xmm1, xmm2/m128	RM	V/V	SSE2	Move aligned packed integer values from xmm2/mem to xmm1.
66 0F 7F /r MOVDQA xmm2/m128, xmm1	MR	V/V	SSE2	Move aligned packed integer values from xmm1 to xmm2/mem.
VEX.128.66.0F.WIG 6F /r VMOVDQA xmm1, xmm2/m128	RM	V/V	AVX	Move aligned packed integer values from xmm2/mem to xmm1.
VEX.128.66.0F.WIG 7F /r VMOVDQA xmm2/m128, xmm1	MR	V/V	AVX	Move aligned packed integer values from xmm1 to xmm2/mem.
VEX.256.66.0F.WIG 6F /r VMOVDQA ymm1, ymm2/m256	RM	V/V	AVX	Move aligned packed integer values from ymm2/mem to ymm1.
VEX.256.66.0F.WIG 7F /r VMOVDQA ymm2/m256, ymm1	MR	V/V	AVX	Move aligned packed integer values from ymm1 to ymm2/mem.
EVEX.128.66.0F.W0 6F /r VMOVDQA32 xmm1 {k1}{z}, xmm2/m128	FVM-RM	V/V	AVX512VL AVX512F	Move aligned packed doubleword integer values from xmm2/m128 to xmm1 using writemask k1.
EVEX.256.66.0F.W0 6F /r VMOVDQA32 ymm1 {k1}{z}, ymm2/m256	FVM-RM	V/V	AVX512VL AVX512F	Move aligned packed doubleword integer values from ymm2/m256 to ymm1 using writemask k1.
EVEX.512.66.0F.W0 6F /r VMOVDQA32 zmm1 {k1}{z}, zmm2/m512	FVM-RM	V/V	AVX512F	Move aligned packed doubleword integer values from zmm2/m512 to zmm1 using writemask k1.
EVEX.128.66.0F.W0 7F /r VMOVDQA32 xmm2/m128 {k1}{z}, xmm1	FVM-MR	V/V	AVX512VL AVX512F	Move aligned packed doubleword integer values from xmm1 to xmm2/m128 using writemask k1.
EVEX.256.66.0F.W0 7F /r VMOVDQA32 ymm2/m256 {k1}{z}, ymm1	FVM-MR	V/V	AVX512VL AVX512F	Move aligned packed doubleword integer values from ymm1 to ymm2/m256 using writemask k1.
EVEX.512.66.0F.W0 7F /r VMOVDQA32 zmm2/m512 {k1}{z}, zmm1	FVM-MR	V/V	AVX512F	Move aligned packed doubleword integer values from zmm1 to zmm2/m512 using writemask k1.
EVEX.128.66.0F.W1 6F /r VMOVDQA64 xmm1 {k1}{z}, xmm2/m128	FVM-RM	V/V	AVX512VL AVX512F	Move aligned quadword integer values from xmm2/m128 to xmm1 using writemask k1.
EVEX.256.66.0F.W1 6F /r VMOVDQA64 ymm1 {k1}{z}, ymm2/m256	FVM-RM	V/V	AVX512VL AVX512F	Move aligned quadword integer values from ymm2/m256 to ymm1 using writemask k1.
EVEX.512.66.0F.W1 6F /r VMOVDQA64 zmm1 {k1}{z}, zmm2/m512	FVM-RM	V/V	AVX512F	Move aligned packed quadword integer values from zmm2/m512 to zmm1 using writemask k1.
EVEX.128.66.0F.W1 7F /r VMOVDQA64 xmm2/m128 {k1}{z}, xmm1	FVM-MR	V/V	AVX512VL AVX512F	Move aligned packed quadword integer values from xmm1 to xmm2/m128 using writemask k1.
EVEX.256.66.0F.W1 7F /r VMOVDQA64 ymm2/m256 {k1}{z}, ymm1	FVM-MR	V/V	AVX512VL AVX512F	Move aligned packed quadword integer values from ymm1 to ymm2/m256 using writemask k1.
EVEX.512.66.0F.W1 7F /r VMOVDQA64 zmm2/m512 {k1}{z}, zmm1	FVM-MR	V/V	AVX512F	Move aligned packed quadword integer values from zmm1 to zmm2/m512 using writemask k1.

## Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
FVM-RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
FVM-MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

## Description

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

EVEX encoded versions:

Moves 128, 256 or 512 bits of packed doubleword/quadword integer values from the source operand (the second operand) to the destination operand (the first operand). This instruction can be used to load a vector register from an int32/int64 memory location, to store the contents of a vector register into an int32/int64 memory location, or to move data between two ZMM registers. When the source or destination operand is a memory operand, the operand must be aligned on a 16 (EVEX.128)/32(EVEX.256)/64(EVEX.512)-byte boundary or a general-protection exception (#GP) will be generated. To move integer data to and from unaligned memory locations, use the VMOVDQU instruction.

The destination operand is updated at 32-bit (VMOVDQA32) or 64-bit (VMOVDQA64) granularity according to the writemask.

VEX.256 encoded version:

Moves 256 bits of packed integer values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load a YMM register from a 256-bit memory location, to store the contents of a YMM register into a 256-bit memory location, or to move data between two YMM registers.

When the source or destination operand is a memory operand, the operand must be aligned on a 32-byte boundary or a general-protection exception (#GP) will be generated. To move integer data to and from unaligned memory locations, use the VMOVDQU instruction. Bits (MAX\_VL-1:256) of the destination register are zeroed.

128-bit versions:

Moves 128 bits of packed integer values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load an XMM register from a 128-bit memory location, to store the contents of an XMM register into a 128-bit memory location, or to move data between two XMM registers.

When the source or destination operand is a memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated. To move integer data to and from unaligned memory locations, use the VMOVDQU instruction.

128-bit Legacy SSE version: Bits (MAX\_VL-1:128) of the corresponding ZMM destination register remain unchanged.

VEX.128 encoded version: Bits (MAX\_VL-1:128) of the destination register are zeroed.

**Operation****VMOVDQA32 (EVEX encoded versions, register-copy form)**

(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[MAX\_VL-1:VL] ← 0

**VMOVDQA32 (EVEX encoded versions, store-form)**

(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 \*DEST[i+31:i] remains unchanged\* ; merging-masking

FI;

ENDFOR;

**VMOVDQA32 (EVEX encoded versions, load-form)**

(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[MAX\_VL-1:VL] ← 0

**VMOVDQA64 (EVEX encoded versions, register-copy form)**

(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[MAX\_VL-1:VL] ← 0

**VMOVDQA64 (EVEX encoded versions, store-form)**

(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 \*DEST[i+63:i] remains unchanged\* ; merging-masking

FI;

ENDFOR;

**VMOVDQA64 (EVEX encoded versions, load-form)**

(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[MAX\_VL-1:VL] ← 0

**VMOVDQA (VEX.256 encoded version, load - and register copy)**

DEST[255:0] ← SRC[255:0]

DEST[MAX\_VL-1:256] ← 0

**VMOVDQA (VEX.256 encoded version, store-form)**

DEST[255:0] ← SRC[255:0]

**VMOVDQA (VEX.128 encoded version)**

DEST[127:0] ← SRC[127:0]

DEST[MAX\_VL-1:128] ← 0

**VMOVDQA (128-bit load- and register-copy- form Legacy SSE version)**

DEST[127:0] ← SRC[127:0]

DEST[MAX\_VL-1:128] (Unmodified)

**(V)MOVDQA (128-bit store-form version)**

DEST[127:0] ← SRC[127:0]

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VMOVDQA32 __m512i __mm512_load_epi32( void * sa);
VMOVDQA32 __m512i __mm512_mask_load_epi32(__m512i s, __mmask16 k, void * sa);
VMOVDQA32 __m512i __mm512_maskz_load_epi32( __mmask16 k, void * sa);
VMOVDQA32 void __mm512_store_epi32(void * d, __m512i a);
VMOVDQA32 void __mm512_mask_store_epi32(void * d, __mmask16 k, __m512i a);
VMOVDQA32 __m256i __mm256_mask_load_epi32(__m256i s, __mmask8 k, void * sa);
VMOVDQA32 __m256i __mm256_maskz_load_epi32( __mmask8 k, void * sa);
VMOVDQA32 void __mm256_store_epi32(void * d, __m256i a);
VMOVDQA32 void __mm256_mask_store_epi32(void * d, __mmask8 k, __m256i a);
VMOVDQA32 __m128i __mm_mask_load_epi32(__m128i s, __mmask8 k, void * sa);
VMOVDQA32 __m128i __mm_maskz_load_epi32( __mmask8 k, void * sa);
VMOVDQA32 void __mm_store_epi32(void * d, __m128i a);
VMOVDQA32 void __mm_mask_store_epi32(void * d, __mmask8 k, __m128i a);
VMOVDQA64 __m512i __mm512_load_epi64( void * sa);
VMOVDQA64 __m512i __mm512_mask_load_epi64(__m512i s, __mmask8 k, void * sa);
VMOVDQA64 __m512i __mm512_maskz_load_epi64( __mmask8 k, void * sa);
VMOVDQA64 void __mm512_store_epi64(void * d, __m512i a);
VMOVDQA64 void __mm512_mask_store_epi64(void * d, __mmask8 k, __m512i a);
VMOVDQA64 __m256i __mm256_mask_load_epi64(__m256i s, __mmask8 k, void * sa);
VMOVDQA64 __m256i __mm256_maskz_load_epi64( __mmask8 k, void * sa);
VMOVDQA64 void __mm256_store_epi64(void * d, __m256i a);
VMOVDQA64 void __mm256_mask_store_epi64(void * d, __mmask8 k, __m256i a);
VMOVDQA64 __m128i __mm_mask_load_epi64(__m128i s, __mmask8 k, void * sa);
VMOVDQA64 __m128i __mm_maskz_load_epi64( __mmask8 k, void * sa);
VMOVDQA64 void __mm_store_epi64(void * d, __m128i a);
VMOVDQA64 void __mm_mask_store_epi64(void * d, __mmask8 k, __m128i a);
MOVDQA void __m256i __mm256_load_si256 (__m256i * p);
MOVDQA __m256i __mm256_store_si256(__m256i *p, __m256i a);
MOVDQA __m128i __mm_load_si128 (__m128i * p);
MOVDQA void __mm_store_si128(__m128i *p, __m128i a);

```

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type1.SSE2;

EVEX-encoded instruction, see Exceptions Type E1.

#UD If EVEX.vvvv != 1111B or VEX.vvvv != 1111B.



## MOVDQU, VMOVDQU8/16/32/64—Move Unaligned Packed Integer Values

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 6F /r MOVDQU xmm1, xmm2/m128	RM	V/V	SSE2	Move unaligned packed integer values from xmm2/m128 to xmm1.
F3 0F 7F /r MOVDQU xmm2/m128, xmm1	MR	V/V	SSE2	Move unaligned packed integer values from xmm1 to xmm2/m128.
VEX.128.F3.0F.WIG 6F /r VMOVDQU xmm1, xmm2/m128	RM	V/V	AVX	Move unaligned packed integer values from xmm2/m128 to xmm1.
VEX.128.F3.0F.WIG 7F /r VMOVDQU xmm2/m128, xmm1	MR	V/V	AVX	Move unaligned packed integer values from xmm1 to xmm2/m128.
VEX.256.F3.0F.WIG 6F /r VMOVDQU ymm1, ymm2/m256	RM	V/V	AVX	Move unaligned packed integer values from ymm2/m256 to ymm1.
VEX.256.F3.0F.WIG 7F /r VMOVDQU ymm2/m256, ymm1	MR	V/V	AVX	Move unaligned packed integer values from ymm1 to ymm2/m256.
EVEX.128.F2.0F.W0 6F /r VMOVDQU8 xmm1 {k1}{z}, xmm2/m128	FVM-RM	V/V	AVX512VL AVX512BW	Move unaligned packed byte integer values from xmm2/m128 to xmm1 using writemask k1.
EVEX.256.F2.0F.W0 6F /r VMOVDQU8 ymm1 {k1}{z}, ymm2/m256	FVM-RM	V/V	AVX512VL AVX512BW	Move unaligned packed byte integer values from ymm2/m256 to ymm1 using writemask k1.
EVEX.512.F2.0F.W0 6F /r VMOVDQU8 zmm1 {k1}{z}, zmm2/m512	FVM-RM	V/V	AVX512BW	Move unaligned packed byte integer values from zmm2/m512 to zmm1 using writemask k1.
EVEX.128.F2.0F.W0 7F /r VMOVDQU8 xmm2/m128 {k1}{z}, xmm1	FVM-MR	V/V	AVX512VL AVX512BW	Move unaligned packed byte integer values from xmm1 to xmm2/m128 using writemask k1.
EVEX.256.F2.0F.W0 7F /r VMOVDQU8 ymm2/m256 {k1}{z}, ymm1	FVM-MR	V/V	AVX512VL AVX512BW	Move unaligned packed byte integer values from ymm1 to ymm2/m256 using writemask k1.
EVEX.512.F2.0F.W0 7F /r VMOVDQU8 zmm2/m512 {k1}{z}, zmm1	FVM-MR	V/V	AVX512BW	Move unaligned packed byte integer values from zmm1 to zmm2/m512 using writemask k1.
EVEX.128.F2.0F.W1 6F /r VMOVDQU16 xmm1 {k1}{z}, xmm2/m128	FVM-RM	V/V	AVX512VL AVX512BW	Move unaligned packed word integer values from xmm2/m128 to xmm1 using writemask k1.
EVEX.256.F2.0F.W1 6F /r VMOVDQU16 ymm1 {k1}{z}, ymm2/m256	FVM-RM	V/V	AVX512VL AVX512BW	Move unaligned packed word integer values from ymm2/m256 to ymm1 using writemask k1.
EVEX.512.F2.0F.W1 6F /r VMOVDQU16 zmm1 {k1}{z}, zmm2/m512	FVM-RM	V/V	AVX512BW	Move unaligned packed word integer values from zmm2/m512 to zmm1 using writemask k1.
EVEX.128.F2.0F.W1 7F /r VMOVDQU16 xmm2/m128 {k1}{z}, xmm1	FVM-MR	V/V	AVX512VL AVX512BW	Move unaligned packed word integer values from xmm1 to xmm2/m128 using writemask k1.
EVEX.256.F2.0F.W1 7F /r VMOVDQU16 ymm2/m256 {k1}{z}, ymm1	FVM-MR	V/V	AVX512VL AVX512BW	Move unaligned packed word integer values from ymm1 to ymm2/m256 using writemask k1.
EVEX.512.F2.0F.W1 7F /r VMOVDQU16 zmm2/m512 {k1}{z}, zmm1	FVM-MR	V/V	AVX512BW	Move unaligned packed word integer values from zmm1 to zmm2/m512 using writemask k1.
EVEX.128.F3.0F.W0 6F /r VMOVDQU32 xmm1 {k1}{z}, xmm2/m128	FVM-RM	V/V	AVX512VL AVX512F	Move unaligned packed doubleword integer values from xmm2/m128 to xmm1 using writemask k1.

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.256.F3.0F.W0 6F /r VMOVDQU32 ymm1 {k1}{z}, ymm2/m256	FVM-RM	V/V	AVX512VL AVX512F	Move unaligned packed doubleword integer values from ymm2/m256 to ymm1 using writemask k1.
EVEX.512.F3.0F.W0 6F /r VMOVDQU32 zmm1 {k1}{z}, zmm2/m512	FVM-RM	V/V	AVX512F	Move unaligned packed doubleword integer values from zmm2/m512 to zmm1 using writemask k1.
EVEX.128.F3.0F.W0 7F /r VMOVDQU32 xmm2/m128 {k1}{z}, xmm1	FVM-MR	V/V	AVX512VL AVX512F	Move unaligned packed doubleword integer values from xmm2/m128 to xmm1 using writemask k1.
EVEX.256.F3.0F.W0 7F /r VMOVDQU32 ymm2/m256 {k1}{z}, ymm1	FVM-MR	V/V	AVX512VL AVX512F	Move unaligned packed doubleword integer values from ymm1 to ymm2/m256 using writemask k1.
EVEX.512.F3.0F.W0 7F /r VMOVDQU32 zmm2/m512 {k1}{z}, zmm1	FVM-MR	V/V	AVX512F	Move unaligned packed doubleword integer values from zmm1 to zmm2/m512 using writemask k1.
EVEX.128.F3.0F.W1 6F /r VMOVDQU64 xmm1 {k1}{z}, xmm2/m128	FVM-RM	V/V	AVX512VL AVX512F	Move unaligned packed quadword integer values from xmm2/m128 to xmm1 using writemask k1.
EVEX.256.F3.0F.W1 6F /r VMOVDQU64 ymm1 {k1}{z}, ymm2/m256	FVM-RM	V/V	AVX512VL AVX512F	Move unaligned packed quadword integer values from ymm2/m256 to ymm1 using writemask k1.
EVEX.512.F3.0F.W1 6F /r VMOVDQU64 zmm1 {k1}{z}, zmm2/m512	FVM-RM	V/V	AVX512F	Move unaligned packed quadword integer values from zmm2/m512 to zmm1 using writemask k1.
EVEX.128.F3.0F.W1 7F /r VMOVDQU64 xmm2/m128 {k1}{z}, xmm1	FVM-MR	V/V	AVX512VL AVX512F	Move unaligned packed quadword integer values from xmm2/m128 to xmm1 using writemask k1.
EVEX.256.F3.0F.W1 7F /r VMOVDQU64 ymm2/m256 {k1}{z}, ymm1	FVM-MR	V/V	AVX512VL AVX512F	Move unaligned packed quadword integer values from ymm1 to ymm2/m256 using writemask k1.
EVEX.512.F3.0F.W1 7F /r VMOVDQU64 zmm2/m512 {k1}{z}, zmm1	FVM-MR	V/V	AVX512F	Move unaligned packed quadword integer values from zmm1 to zmm2/m512 using writemask k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
FVM-RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
FVM-MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

#### Description

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

#### EVEX encoded versions:

Moves 128, 256 or 512 bits of packed byte/word/doubleword/quadword integer values from the source operand (the second operand) to the destination operand (first operand). This instruction can be used to load a vector register from a memory location, to store the contents of a vector register into a memory location, or to move data between two vector registers.

The destination operand is updated at 8-bit (VMOVDQU8), 16-bit (VMOVDQU16), 32-bit (VMOVDQU32), or 64-bit (VMOVDQU64) granularity according to the writemask.

**VEX.256 encoded version:**

Moves 256 bits of packed integer values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load a YMM register from a 256-bit memory location, to store the contents of a YMM register into a 256-bit memory location, or to move data between two YMM registers.

Bits (MAX\_VL-1:256) of the destination register are zeroed.

128-bit versions:

Moves 128 bits of packed integer values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load an XMM register from a 128-bit memory location, to store the contents of an XMM register into a 128-bit memory location, or to move data between two XMM registers.

**128-bit Legacy SSE version:** Bits (MAX\_VL-1:128) of the corresponding destination register remain unchanged.

When the source or destination operand is a memory operand, the operand may be unaligned to any alignment without causing a general-protection exception (#GP) to be generated

**VEX.128 encoded version:** Bits (MAX\_VL-1:128) of the destination register are zeroed.

**Operation**

**VMOVDQU8 (EVEX encoded versions, register-copy form)**

(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[MAX\_VL-1:VL] ← 0

**VMOVDQU8 (EVEX encoded versions, store-form)**

(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 \*DEST[i+7:i] remains unchanged\* ; merging-masking

  FI;

ENDFOR;

**VMOVDQU8 (EVEX encoded versions, load-form)**

(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[MAX\_VL-1:VL] ← 0

**VMOVDQU16 (EVEX encoded versions, register-copy form)**

(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[MAX\_VL-1:VL] ← 0

**VMOVDQU16 (EVEX encoded versions, store-form)**

(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 \*DEST[i+15:i] remains unchanged\* ; merging-masking

FI;

ENDFOR;

**VMOVDQU16 (EVEX encoded versions, load-form)**

(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[MAX\_VL-1:VL] ← 0

**VMOVDQU32 (EVEX encoded versions, register-copy form)**

(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[MAX\_VL-1:VL] ← 0

**VMOVDQU32 (EVEX encoded versions, store-form)**

(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 \*DEST[i+31:i] remains unchanged\* ; merging-masking

FI;

ENDFOR;

**VMOVDQU32 (EVEX encoded versions, load-form)**

(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[MAX\_VL-1:VL] ← 0

**VMOVDQU64 (EVEX encoded versions, register-copy form)**

(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[MAX\_VL-1:VL] ← 0

**VMOVDQU64 (EVEX encoded versions, store-form)**

(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 \*DEST[i+63:i] remains unchanged\* ; merging-masking

FI;

ENDFOR;

**VMOVDQU64 (EVEX encoded versions, load-form)**

(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[MAX\_VL-1:VL] ← 0

**VMOVDQU (VEX.256 encoded version, load - and register copy)**

DEST[255:0] ← SRC[255:0]

DEST[MAX\_VL-1:256] ← 0

**VMOVDQU (VEX.256 encoded version, store-form)**

DEST[255:0] ← SRC[255:0]

VMOVDQU (VEX.128 encoded version)

DEST[127:0] ← SRC[127:0]

DEST[MAX\_VL-1:128] ← 0

**VMOVDQU (128-bit load- and register-copy- form Legacy SSE version)**

DEST[127:0] ← SRC[127:0]

DEST[MAX\_VL-1:128] (Unmodified)

**(V)MOVDQU (128-bit store-form version)**

DEST[127:0] ← SRC[127:0]

**Intel C/C++ Compiler Intrinsic Equivalent**

VMOVDQU16 \_\_m512i \_mm512\_mask\_loadu\_epi16(\_\_m512i s, \_\_mmask32 k, void \* sa);

VMOVDQU16 \_\_m512i \_mm512\_maskz\_loadu\_epi16(\_\_mmask32 k, void \* sa);

VMOVDQU16 void \_mm512\_mask\_storeu\_epi16(void \* d, \_\_mmask32 k, \_\_m512i a);

VMOVDQU16 \_\_m256i \_mm256\_mask\_loadu\_epi16(\_\_m256i s, \_\_mmask16 k, void \* sa);

VMOVDQU16 \_\_m256i \_mm256\_maskz\_loadu\_epi16(\_\_mmask16 k, void \* sa);

VMOVDQU16 void \_mm256\_mask\_storeu\_epi16(void \* d, \_\_mmask16 k, \_\_m256i a);

VMOVDQU16 void \_mm256\_maskz\_storeu\_epi16(void \* d, \_\_mmask16 k);

VMOVDQU16 \_\_m128i \_mm\_mask\_loadu\_epi16(\_\_m128i s, \_\_mmask8 k, void \* sa);

VMOVDQU16 \_\_m128i \_mm\_maskz\_loadu\_epi16(\_\_mmask8 k, void \* sa);

VMOVDQU16 void \_mm\_mask\_storeu\_epi16(void \* d, \_\_mmask8 k, \_\_m128i a);

VMOVDQU32 \_\_m512i \_mm512\_loadu\_epi32(void \* sa);

VMOVDQU32 \_\_m512i \_mm512\_mask\_loadu\_epi32(\_\_m512i s, \_\_mmask16 k, void \* sa);

VMOVDQU32 \_\_m512i \_mm512\_maskz\_loadu\_epi32(\_\_mmask16 k, void \* sa);

VMOVDQU32 void \_mm512\_storeu\_epi32(void \* d, \_\_m512i a);

VMOVDQU32 void \_mm512\_mask\_storeu\_epi32(void \* d, \_\_mmask16 k, \_\_m512i a);

VMOVDQU32 \_\_m256i \_mm256\_mask\_loadu\_epi32(\_\_m256i s, \_\_mmask8 k, void \* sa);

VMOVDQU32 \_\_m256i \_mm256\_maskz\_loadu\_epi32(\_\_mmask8 k, void \* sa);

VMOVDQU32 void \_mm256\_storeu\_epi32(void \* d, \_\_m256i a);

VMOVDQU32 void \_mm256\_mask\_storeu\_epi32(void \* d, \_\_mmask8 k, \_\_m256i a);

VMOVDQU32 \_\_m128i \_mm\_mask\_loadu\_epi32(\_\_m128i s, \_\_mmask8 k, void \* sa);

```

VMOVDQU32 __m128i _mm_mask_loadu_epi32( __mmask8 k, void * sa);
VMOVDQU32 void _mm_storeu_epi32(void * d, __m128i a);
VMOVDQU32 void _mm_mask_storeu_epi32(void * d, __mmask8 k, __m128i a);
VMOVDQU64 __m512i _mm512_loadu_epi64( void * sa);
VMOVDQU64 __m512i _mm512_mask_loadu_epi64(__m512i s, __mmask8 k, void * sa);
VMOVDQU64 __m512i _mm512_maskz_loadu_epi64( __mmask8 k, void * sa);
VMOVDQU64 void _mm512_storeu_epi64(void * d, __m512i a);
VMOVDQU64 void _mm512_mask_storeu_epi64(void * d, __mmask8 k, __m512i a);
VMOVDQU64 __m256i _mm256_mask_loadu_epi64(__m256i s, __mmask8 k, void * sa);
VMOVDQU64 __m256i _mm256_maskz_loadu_epi64( __mmask8 k, void * sa);
VMOVDQU64 void _mm256_storeu_epi64(void * d, __m256i a);
VMOVDQU64 void _mm256_mask_storeu_epi64(void * d, __mmask8 k, __m256i a);
VMOVDQU64 __m128i _mm_mask_loadu_epi64(__m128i s, __mmask8 k, void * sa);
VMOVDQU64 __m128i _mm_maskz_loadu_epi64( __mmask8 k, void * sa);
VMOVDQU64 void _mm_storeu_epi64(void * d, __m128i a);
VMOVDQU64 void _mm_mask_storeu_epi64(void * d, __mmask8 k, __m128i a);
VMOVDQU8 __m512i _mm512_mask_loadu_epi8(__m512i s, __mmask64 k, void * sa);
VMOVDQU8 __m512i _mm512_maskz_loadu_epi8( __mmask64 k, void * sa);
VMOVDQU8 void _mm512_mask_storeu_epi8(void * d, __mmask64 k, __m512i a);
VMOVDQU8 __m256i _mm256_mask_loadu_epi8(__m256i s, __mmask32 k, void * sa);
VMOVDQU8 __m256i _mm256_maskz_loadu_epi8( __mmask32 k, void * sa);
VMOVDQU8 void _mm256_mask_storeu_epi8(void * d, __mmask32 k, __m256i a);
VMOVDQU8 void _mm256_maskz_storeu_epi8(void * d, __mmask32 k);
VMOVDQU8 __m128i _mm_mask_loadu_epi8(__m128i s, __mmask16 k, void * sa);
VMOVDQU8 __m128i _mm_maskz_loadu_epi8( __mmask16 k, void * sa);
VMOVDQU8 void _mm_mask_storeu_epi8(void * d, __mmask16 k, __m128i a);
MOVDQU __m256i _mm256_loadu_si256( __m256i * p);
MOVDQU __m256i _mm256_storeu_si256(__m256i *p, __m256i a);
MOVDQU __m128i _mm_loadu_si128( __m128i * p);
MOVDQU __m128i _mm_storeu_si128(__m128i *p, __m128i a);

```

### SIMD Floating-Point Exceptions

None

### Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4;

EVEX-encoded instruction, see Exceptions Type E4.nb.

#UD                      If EVEX.vvvv != 1111B or VEX.vvvv != 1111B.



## MOVHPLS—Move Packed Single-Precision Floating-Point Values High to Low

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
OF 12 /r MOVHPLS xmm1, xmm2	RM	V/V	SSE	Move two packed single-precision floating-point values from high quadword of xmm2 to low quadword of xmm1.
VEX.NDS.128.OF.WIG 12 /r VMOVHPLS xmm1, xmm2, xmm3	RVM	V/V	AVX	Merge two packed single-precision floating-point values from high quadword of xmm3 and low quadword of xmm2.
EVEX.NDS.128.OF.W0 12 /r VMOVHPLS xmm1, xmm2, xmm3	RVM	V/V	AVX512F	Merge two packed single-precision floating-point values from high quadword of xmm3 and low quadword of xmm2.

### Instruction Operand Encoding<sup>1</sup>

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	vvvv (r)	ModRM:r/m (r)	NA

### Description

This instruction cannot be used for memory to register moves.

#### 128-bit two-argument form:

Moves two packed single-precision floating-point values from the high quadword of the second XMM argument (second operand) to the low quadword of the first XMM register (first argument). The quadword at bits 127:64 of the destination operand is left unchanged. Bits (MAX\_VL-1:128) of the corresponding destination register remain unchanged.

#### 128-bit and EVEX three-argument form

Moves two packed single-precision floating-point values from the high quadword of the third XMM argument (third operand) to the low quadword of the destination (first operand). Copies the high quadword from the second XMM argument (second operand) to the high quadword of the destination (first operand). Bits (MAX\_VL-1:128) of the corresponding destination register are zeroed.

If VMOVHPLS is encoded with VEX.L or EVEX.L'L= 1, an attempt to execute the instruction encoded with VEX.L or EVEX.L'L= 1 will cause an #UD exception.

### Operation

#### MOVHPLS (128-bit two-argument form)

DEST[63:0] ← SRC[127:64]  
DEST[MAX\_VL-1:64] (Unmodified)

#### VMOVHPLS (128-bit three-argument form - VEX & EVEX)

DEST[63:0] ← SRC2[127:64]  
DEST[127:64] ← SRC1[127:64]  
DEST[MAX\_VL-1:128] ← 0

### Intel C/C++ Compiler Intrinsic Equivalent

MOVHPLS \_\_m128 \_\_mm\_movehl\_ps(\_\_m128 a, \_\_m128 b)

### SIMD Floating-Point Exceptions

None

1. ModRM.MOD = 011B required



### Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 7; additionally  
#UD If VEX.L = 1.

EVEX-encoded instruction, see Exceptions Type E7NM.128.

## MOVHPD—Move High Packed Double-Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 16 /r MOVHPD xmm1, m64	RM	V/V	SSE2	Move double-precision floating-point value from m64 to high quadword of xmm1.
VEX.NDS.128.66.0F.WIG 16 /r VMOVHPD xmm2, xmm1, m64	RVM	V/V	AVX	Merge double-precision floating-point value from m64 and the low quadword of xmm1.
EVEX.NDS.128.66.0F.W1 16 /r VMOVHPD xmm2, xmm1, m64	T1S	V/V	AVX512F	Merge double-precision floating-point value from m64 and the low quadword of xmm1.
66 0F 17 /r MOVHPD m64, xmm1	MR	V/V	SSE2	Move double-precision floating-point value from high quadword of xmm1 to m64.
VEX.128.66.0F.WIG 17 /r VMOVHPD m64, xmm1	MR	V/V	AVX	Move double-precision floating-point value from high quadword of xmm1 to m64.
EVEX.128.66.0F.W1 17 /r VMOVHPD m64, xmm1	T1S-MR	V/V	AVX512F	Move double-precision floating-point value from high quadword of xmm1 to m64.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
T1S	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA
T1S-MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

### Description

This instruction cannot be used for register to register or memory to memory moves.

#### 128-bit Legacy SSE load:

Moves a double-precision floating-point value from the source 64-bit memory operand and stores it in the high 64-bits of the destination XMM register. The lower 64bits of the XMM register are preserved. Bits (MAX\_VL-1:128) of the corresponding destination register are preserved.

#### VEX.128 & EVEX encoded load:

Loads a double-precision floating-point value from the source 64-bit memory operand (the third operand) and stores it in the upper 64-bits of the destination XMM register (first operand). The low 64-bits from the first source operand (second operand) are copied to the low 64-bits of the destination. Bits (MAX\_VL-1:128) of the corresponding destination register are zeroed.

#### 128-bit store:

Stores a double-precision floating-point value from the high 64-bits of the XMM register source (second operand) to the 64-bit memory location (first operand).

Note: VMOVHPD (store) (VEX.128.66.0F 17 /r) is legal and has the same behavior as the existing 66 0F 17 store. For VMOVHPD (store) VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instruction will #UD.

If VMOVHPD is encoded with VEX.L or EVEX.L'L= 1, an attempt to execute the instruction encoded with VEX.L or EVEX.L'L= 1 will cause an #UD exception.

**Operation****MOVHPD (128-bit Legacy SSE load)**

DEST[63:0] (Unmodified)  
 DEST[127:64] ← SRC[63:0]  
 DEST[MAX\_VL-1:128] (Unmodified)

**VMOVHPD (VEX.128 & EVEX encoded load)**

DEST[63:0] ← SRC1[63:0]  
 DEST[127:64] ← SRC2[63:0]  
 DEST[MAX\_VL-1:128] ← 0

**VMOVHPD (store)**

DEST[63:0] ← SRC[127:64]

**Intel C/C++ Compiler Intrinsic Equivalent**

MOVHPD \_\_m128d \_mm\_loadh\_pd ( \_\_m128d a, double \*p)  
 MOVHPD void \_mm\_storeh\_pd (double \*p, \_\_m128d a)

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 5; additionally

#UD                      If VEX.L = 1.

EVEX-encoded instruction, see Exceptions Type E9NF.

## MOVHPS—Move High Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F 16 /r MOVHPS xmm1, m64	RM	V/V	SSE	Move two packed single-precision floating-point values from m64 to high quadword of xmm1.
VEX.NDS.128.0F.WIG 16 /r VMOVHPS xmm2, xmm1, m64	RVM	V/V	AVX	Merge two packed single-precision floating-point values from m64 and the low quadword of xmm1.
EVEX.NDS.128.0F.W0 16 /r VMOVHPS xmm2, xmm1, m64	T2	V/V	AVX512F	Merge two packed single-precision floating-point values from m64 and the low quadword of xmm1.
0F 17 /r MOVHPS m64, xmm1	MR	V/V	SSE	Move two packed single-precision floating-point values from high quadword of xmm1 to m64.
VEX.128.0F.WIG 17 /r VMOVHPS m64, xmm1	MR	V/V	AVX	Move two packed single-precision floating-point values from high quadword of xmm1 to m64.
EVEX.128.0F.W0 17 /r VMOVHPS m64, xmm1	T2-MR	V/V	AVX512F	Move two packed single-precision floating-point values from high quadword of xmm1 to m64.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
T2	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA
T2-MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

### Description

This instruction cannot be used for register to register or memory to memory moves.

#### 128-bit Legacy SSE load:

Moves two packed single-precision floating-point values from the source 64-bit memory operand and stores them in the high 64-bits of the destination XMM register. The lower 64bits of the XMM register are preserved. Bits (MAX\_VL-1:128) of the corresponding destination register are preserved.

#### VEX.128 & EVEX encoded load:

Loads two single-precision floating-point values from the source 64-bit memory operand (the third operand) and stores it in the upper 64-bits of the destination XMM register (first operand). The low 64-bits from the first source operand (the second operand) are copied to the lower 64-bits of the destination. Bits (MAX\_VL-1:128) of the corresponding destination register are zeroed.

#### 128-bit store:

Stores two packed single-precision floating-point values from the high 64-bits of the XMM register source (second operand) to the 64-bit memory location (first operand).

Note: VMOVHPS (store) (VEX.NDS.128.0F 17 /r) is legal and has the same behavior as the existing 0F 17 store. For VMOVHPS (store) VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instruction will #UD.

If VMOVHPS is encoded with VEX.L or EVEX.L'L= 1, an attempt to execute the instruction encoded with VEX.L or EVEX.L'L= 1 will cause an #UD exception.

**Operation****MOVHPS (128-bit Legacy SSE load)**

DEST[63:0] (Unmodified)

DEST[127:64] ← SRC[63:0]

DEST[MAX\_VL-1:128] (Unmodified)

**VMOVHPS (VEX.128 and EVEX encoded load)**

DEST[63:0] ← SRC1[63:0]

DEST[127:64] ← SRC2[63:0]

DEST[MAX\_VL-1:128] ← 0

**VMOVHPS (store)**

DEST[63:0] ← SRC[127:64]

**Intel C/C++ Compiler Intrinsic Equivalent**

MOVHPS \_\_m128 \_mm\_loadh\_pi ( \_\_m128 a, \_\_m64 \*p)

MOVHPS void \_mm\_storeh\_pi ( \_\_m64 \*p, \_\_m128 a)

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 5; additionally

#UD If VEX.L = 1.

EVEX-encoded instruction, see Exceptions Type E9NF.

## MOVLHPS—Move Packed Single-Precision Floating-Point Values Low to High

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
OF 16 /r MOVLHPS xmm1, xmm2	RM	V/V	SSE	Move two packed single-precision floating-point values from low quadword of xmm2 to high quadword of xmm1.
VEX.NDS.128.OF.WIG 16 /r VMOVLHPS xmm1, xmm2, xmm3	RVM	V/V	AVX	Merge two packed single-precision floating-point values from low quadword of xmm3 and low quadword of xmm2.
EVEX.NDS.128.OF.WO 16 /r VMOVLHPS xmm1, xmm2, xmm3	RVM	V/V	AVX512F	Merge two packed single-precision floating-point values from low quadword of xmm3 and low quadword of xmm2.

### Instruction Operand Encoding<sup>1</sup>

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	vvvv (r)	ModRM:r/m (r)	NA

### Description

This instruction cannot be used for memory to register moves.

#### 128-bit two-argument form:

Moves two packed single-precision floating-point values from the low quadword of the second XMM argument (second operand) to the high quadword of the first XMM register (first argument). The low quadword of the destination operand is left unchanged. Bits (MAX\_VL-1:128) of the corresponding destination register are unmodified.

#### 128-bit three-argument forms:

Moves two packed single-precision floating-point values from the low quadword of the third XMM argument (third operand) to the high quadword of the destination (first operand). Copies the low quadword from the second XMM argument (second operand) to the low quadword of the destination (first operand). Bits (MAX\_VL-1:128) of the corresponding destination register are zeroed.

If VMOVLHPS is encoded with VEX.L or EVEX.L'L= 1, an attempt to execute the instruction encoded with VEX.L or EVEX.L'L= 1 will cause an #UD exception.

### Operation

#### MOVLHPS (128-bit two-argument form)

DEST[63:0] (Unmodified)  
 DEST[127:64] ← SRC[63:0]  
 DEST[MAX\_VL-1:128] (Unmodified)

#### VMOVLHPS (128-bit three-argument form - VEX & EVEX)

DEST[63:0] ← SRC1[63:0]  
 DEST[127:64] ← SRC2[63:0]  
 DEST[MAX\_VL-1:128] ← 0

### Intel C/C++ Compiler Intrinsic Equivalent

MOVLHPS \_\_m128 \_\_mm\_movelh\_ps(\_\_m128 a, \_\_m128 b)

### SIMD Floating-Point Exceptions

None

1. ModRM.MOD = 011B required



### Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 7; additionally

#UD                      If VEX.L = 1.

EVEX-encoded instruction, see Exceptions Type E7NM.128.



## MOVLPD—Move Low Packed Double-Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 12 /r MOVLPD xmm1, m64	RM	V/V	SSE2	Move double-precision floating-point value from m64 to low quadword of xmm1.
VEX.NDS.128.66.0F.WIG 12 /r VMOVLPD xmm2, xmm1, m64	RVM	V/V	AVX	Merge double-precision floating-point value from m64 and the high quadword of xmm1.
EVEX.NDS.128.66.0F.W1 12 /r VMOVLPD xmm2, xmm1, m64	T1S	V/V	AVX512F	Merge double-precision floating-point value from m64 and the high quadword of xmm1.
66 0F 13/r MOVLPD m64, xmm1	MR	V/V	SSE2	Move double-precision floating-point value from low quadword of xmm1 to m64.
VEX.128.66.0F.WIG 13/r VMOVLPD m64, xmm1	MR	V/V	AVX	Move double-precision floating-point value from low quadword of xmm1 to m64.
EVEX.128.66.0F.W1 13/r VMOVLPD m64, xmm1	T1S-MR	V/V	AVX512F	Move double-precision floating-point value from low quadword of xmm1 to m64.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:r/m (r)	VEX.vvvv	ModRM:r/m (r)	NA
MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
T1S	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA
T1S-MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

### Description

This instruction cannot be used for register to register or memory to memory moves.

#### 128-bit Legacy SSE load:

Moves a double-precision floating-point value from the source 64-bit memory operand and stores it in the low 64-bits of the destination XMM register. The upper 64bits of the XMM register are preserved. Bits (MAX\_VL-1:128) of the corresponding destination register are preserved.

#### VEX.128 & EVEX encoded load:

Loads a double-precision floating-point value from the source 64-bit memory operand (third operand), merges it with the upper 64-bits of the first source XMM register (second operand), and stores it in the low 128-bits of the destination XMM register (first operand). Bits (MAX\_VL-1:128) of the corresponding destination register are zeroed.

#### 128-bit store:

Stores a double-precision floating-point value from the low 64-bits of the XMM register source (second operand) to the 64-bit memory location (first operand).

Note: VMOVLPD (store) (VEX.128.66.0F 13 /r) is legal and has the same behavior as the existing 66 0F 13 store. For VMOVLPD (store) VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instruction will #UD.

If VMOVLPD is encoded with VEX.L or EVEX.L'L= 1, an attempt to execute the instruction encoded with VEX.L or EVEX.L'L= 1 will cause an #UD exception.

### Operation

#### MOVLPD (128-bit Legacy SSE load)

DEST[63:0] ← SRC[63:0]

DEST[MAX\_VL-1:64] (Unmodified)

**VMOVLPD (VEX.128 & EVEX encoded load)**

DEST[63:0] ← SRC2[63:0]

DEST[127:64] ← SRC1[127:64]

DEST[MAX\_VL-1:128] ← 0

**VMOVLPD (store)**

DEST[63:0] ← SRC[63:0]

**Intel C/C++ Compiler Intrinsic Equivalent**

MOVLPD \_\_m128d \_mm\_load\_pd ( \_\_m128d a, double \*p)

MOVLPD void \_mm\_store\_pd (double \*p, \_\_m128d a)

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 5; additionally

#UD If VEX.L = 1.

EVEX-encoded instruction, see Exceptions Type E9NF.

## MOVLPS—Move Low Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F 12 /r MOVLPS xmm1, m64	RM	V/V	SSE	Move two packed single-precision floating-point values from m64 to low quadword of xmm1.
VEX.NDS.128.0F.WIG 12 /r VMOVLPS xmm2, xmm1, m64	RVM	V/V	AVX	Merge two packed single-precision floating-point values from m64 and the high quadword of xmm1.
EVEX.NDS.128.0F.W0 12 /r VMOVLPS xmm2, xmm1, m64	T2	V/V	AVX512F	Merge two packed single-precision floating-point values from m64 and the high quadword of xmm1.
0F 13/r MOVLPS m64, xmm1	MR	V/V	SSE	Move two packed single-precision floating-point values from low quadword of xmm1 to m64.
VEX.128.0F.WIG 13/r VMOVLPS m64, xmm1	MR	V/V	AVX	Move two packed single-precision floating-point values from low quadword of xmm1 to m64.
EVEX.128.0F.W0 13/r VMOVLPS m64, xmm1	T2-MR	V/V	AVX512F	Move two packed single-precision floating-point values from low quadword of xmm1 to m64.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
T2	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA
T2-MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

### Description

This instruction cannot be used for register to register or memory to memory moves.

#### 128-bit Legacy SSE load:

Moves two packed single-precision floating-point values from the source 64-bit memory operand and stores them in the low 64-bits of the destination XMM register. The upper 64bits of the XMM register are preserved. Bits (MAX\_VL-1:128) of the corresponding destination register are preserved.

#### VEX.128 & EVEX encoded load:

Loads two packed single-precision floating-point values from the source 64-bit memory operand (the third operand), merges them with the upper 64-bits of the first source operand (the second operand), and stores them in the low 128-bits of the destination register (the first operand). Bits (MAX\_VL-1:128) of the corresponding destination register are zeroed.

#### 128-bit store:

Loads two packed single-precision floating-point values from the low 64-bits of the XMM register source (second operand) to the 64-bit memory location (first operand).

Note: VMOVLPS (store) (VEX.128.0F 13 /r) is legal and has the same behavior as the existing 0F 13 store. For VMOVLPS (store) VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instruction will #UD.

If VMOVLPS is encoded with VEX.L or EVEX.L'L= 1, an attempt to execute the instruction encoded with VEX.L or EVEX.L'L= 1 will cause an #UD exception.

**Operation****MOVLPS (128-bit Legacy SSE load)**

DEST[63:0] ← SRC[63:0]

DEST[MAX\_VL-1:64] (Unmodified)

**VMOVLPS (VEX.128 & EVEX encoded load)**

DEST[63:0] ← SRC2[63:0]

DEST[127:64] ← SRC1[127:64]

DEST[MAX\_VL-1:128] ← 0

**VMOVLPS (store)**

DEST[63:0] ← SRC[63:0]

**Intel C/C++ Compiler Intrinsic Equivalent**

MOVLPS \_\_m128 \_mm\_load\_pi (\_\_m128 a, \_\_m64 \*p)

MOVLPS void \_mm\_store\_pi (\_\_m64 \*p, \_\_m128 a)

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 5; additionally

#UD If VEX.L = 1.

EVEX-encoded instruction, see Exceptions Type E9NF.

## MOVNTDQA—Load Double Quadword Non-Temporal Aligned Hint

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 2A /r MOVNTDQA xmm1, m128	RM	V/V	SSE4_1	Move double quadword from m128 to xmm1 using non-temporal hint if WC memory type.
VEX.128.66.0F38.WIG 2A /r VMOVNTDQA xmm1, m128	RM	V/V	AVX	Move double quadword from m128 to xmm using non-temporal hint if WC memory type.
VEX.256.66.0F38.WIG 2A /r VMOVNTDQA ymm1, m256	RM	V/V	AVX2	Move 256-bit data from m256 to ymm using non-temporal hint if WC memory type.
EVEX.128.66.0F38.W0 2A /r VMOVNTDQA xmm1, m128	FVM	V/V	AVX512VL AVX512F	Move 128-bit data from m128 to xmm using non-temporal hint if WC memory type.
EVEX.256.66.0F38.W0 2A /r VMOVNTDQA ymm1, m256	FVM	V/V	AVX512VL AVX512F	Move 256-bit data from m256 to ymm using non-temporal hint if WC memory type.
EVEX.512.66.0F38.W0 2A /r VMOVNTDQA zmm1, m512	FVM	V/V	AVX512F	Move 512-bit data from m512 to zmm using non-temporal hint if WC memory type.

### Instruction Operand Encoding<sup>1</sup>

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
FVM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

MOVNTDQA loads a double quadword from the source operand (second operand) to the destination operand (first operand) using a non-temporal hint if the memory source is WC (write combining) memory type. For WC memory type, the nontemporal hint may be implemented by loading a temporary internal buffer with the equivalent of an aligned cache line without filling this data to the cache. Any memory-type aliased lines in the cache will be snooped and flushed. Subsequent MOVNTDQA reads to unread portions of the WC cache line will receive data from the temporary internal buffer if data is available. The temporary internal buffer may be flushed by the processor at any time for any reason, for example:

- A load operation other than a MOVNTDQA which references memory already resident in a temporary internal buffer.
- A non-WC reference to memory already resident in a temporary internal buffer.
- Interleaving of reads and writes to a single temporary internal buffer.
- Repeated (V)MOVNTDQA loads of a particular 16-byte item in a streaming line.
- Certain micro-architectural conditions including resource shortages, detection of a mis-speculation condition, and various fault conditions

The non-temporal hint is implemented by using a write combining (WC) memory type protocol when reading the data from memory. Using this protocol, the processor

does not read the data into the cache hierarchy, nor does it fetch the corresponding cache line from memory into the cache hierarchy. The memory type of the region being read can override the non-temporal hint, if the memory address specified for the non-temporal read is not a WC memory region. Information on non-temporal reads and writes can be found in “Caching of Temporal vs. Non-Temporal Data” in Chapter 10 in the Intel® 64 and IA-32 Architecture Software Developer’s Manual, Volume 3A.

Because the WC protocol uses a weakly-ordered memory consistency model, a fencing operation implemented with a MFENCE instruction should be used in conjunction with MOVNTDQA instructions if multiple processors might use different memory types for the referenced memory locations or to synchronize reads of a processor with writes by other agents in the system. A processor’s implementation of the streaming load hint does not override the effective

1. ModRM.MOD = 011B required

memory type, but the implementation of the hint is processor dependent. For example, a processor implementation may choose to ignore the hint and process the instruction as a normal `MOVNDQA` for any memory type. Alternatively, another implementation may optimize cache reads generated by `MOVNTDQA` on WB memory type to reduce cache evictions.

The 128-bit (V)`MOVNTDQA` addresses must be 16-byte aligned or the instruction will cause a `#GP`.

The 256-bit `VMOVNTDQA` addresses must be 32-byte aligned or the instruction will cause a `#GP`.

The 512-bit `VMOVNTDQA` addresses must be 64-byte aligned or the instruction will cause a `#GP`.

### Operation

#### **MOVNTDQA (128bit- Legacy SSE form)**

`DEST ← SRC`

`DEST[MAX_VL-1:128]` (Unmodified)

#### **VMOVNTDQA (VEX.128 and EVEX.128 encoded form)**

`DEST ← SRC`

`DEST[MAX_VL-1:128] ← 0`

#### **VMOVNTDQA (VEX.256 and EVEX.256 encoded forms)**

`DEST[255:0] ← SRC[255:0]`

`DEST[MAX_VL-1:256] ← 0`

#### **VMOVNTDQA (EVEX.512 encoded form)**

`DEST[511:0] ← SRC[511:0]`

### Intel C/C++ Compiler Intrinsic Equivalent

`VMOVNTDQA __m512i _mm512_stream_load_si512(void * p);`

`MOVNTDQA __m128i _mm_stream_load_si128 (__m128i *p);`

`VMOVNTDQA __m256i _mm_stream_load_si256 (__m256i *p);`

### SIMD Floating-Point Exceptions

None

### Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type1;

EVEX-encoded instruction, see Exceptions Type E1NF.

`#UD` If `VEX.vvvv != 1111B` or `EVEX.vvvv != 1111B`.

## MOVNTDQ—Store Packed Integers Using Non-Temporal Hint

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F E7 /r MOVNTDQ m128, xmm1	MR	V/V	SSE2	Move packed integer values in xmm1 to m128 using non-temporal hint.
VEX.128.66.0F.WIG E7 /r VMOVNTDQ m128, xmm1	MR	V/V	AVX	Move packed integer values in xmm1 to m128 using non-temporal hint.
VEX.256.66.0F.WIG E7 /r VMOVNTDQ m256, ymm1	MR	V/V	AVX	Move packed integer values in ymm1 to m256 using non-temporal hint.
EVEX.128.66.0F.W0 E7 /r VMOVNTDQ m128, xmm1	FVM	V/V	AVX512VL AVX512F	Move packed integer values in xmm1 to m128 using non-temporal hint.
EVEX.256.66.0F.W0 E7 /r VMOVNTDQ m256, ymm1	FVM	V/V	AVX512VL AVX512F	Move packed integer values in zmm1 to m256 using non-temporal hint.
EVEX.512.66.0F.W0 E7 /r VMOVNTDQ m512, zmm1	FVM	V/V	AVX512F	Move packed integer values in zmm1 to m512 using non-temporal hint.

### Instruction Operand Encoding<sup>1</sup>

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
FVM	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

### Description

Moves the packed integers in the source operand (second operand) to the destination operand (first operand) using a non-temporal hint to prevent caching of the data during the write to memory. The source operand is an XMM register, YMM register or ZMM register, which is assumed to contain integer data (packed bytes, words, double-words, or quadwords). The destination operand is a 128-bit, 256-bit or 512-bit memory location. The memory operand must be aligned on a 16-byte (128-bit version), 32-byte (VEX.256 encoded version) or 64-byte (512-bit version) boundary otherwise a general-protection exception (#GP) will be generated.

The non-temporal hint is implemented by using a write combining (WC) memory type protocol when writing the data to memory. Using this protocol, the processor does not write the data into the cache hierarchy, nor does it fetch the corresponding cache line from memory into the cache hierarchy. The memory type of the region being written to can override the non-temporal hint, if the memory address specified for the non-temporal store is in an uncacheable (UC) or write protected (WP) memory region. For more information on non-temporal stores, see “Caching of Temporal vs. Non-Temporal Data” in Chapter 10 in the IA-32 Intel Architecture Software Developer’s Manual, Volume 1.

Because the WC protocol uses a weakly-ordered memory consistency model, a fencing operation implemented with the SFENCE or MFENCE instruction should be used in conjunction with VMOVNTDQ instructions if multiple processors might use different memory types to read/write the destination memory locations.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b, VEX.L must be 0; otherwise instructions will #UD.

### Operation

#### VMOVNTDQ(EVEX encoded versions)

VL = 128, 256, 512

DEST[VL-1:0] ← SRC[VL-1:0]

DEST[MAX\_VL-1:VL] ← 0

1. ModRM.MOD = 011B required

**MOVNTDQ (Legacy and VEX versions)**

DEST ← SRC

**Intel C/C++ Compiler Intrinsic Equivalent**

VMOVNTDQ void \_mm512\_stream\_si512(void \* p, \_\_m512i a);

VMOVNTDQ void \_mm256\_stream\_si256 (\_\_m256i \* p, \_\_m256i a);

MOVNTDQ void \_mm\_stream\_si128 (\_\_m128i \* p, \_\_m128i a);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type1.SSE2;

EVEX-encoded instruction, see Exceptions Type E1NF.

#UD If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.



## MOVNTPD—Store Packed Double-Precision Floating-Point Values Using Non-Temporal Hint

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 2B /r MOVNTPD m128, xmm1	MR	V/V	SSE2	Move packed double-precision values in xmm1 to m128 using non-temporal hint.
VEX.128.66.0F.WIG 2B /r VMOVNTPD m128, xmm1	MR	V/V	AVX	Move packed double-precision values in xmm1 to m128 using non-temporal hint.
VEX.256.66.0F.WIG 2B /r VMOVNTPD m256, ymm1	MR	V/V	AVX	Move packed double-precision values in ymm1 to m256 using non-temporal hint.
EVEX.128.66.0F.W1 2B /r VMOVNTPD m128, xmm1	FVM	V/V	AVX512VL AVX512F	Move packed double-precision values in xmm1 to m128 using non-temporal hint.
EVEX.256.66.0F.W1 2B /r VMOVNTPD m256, ymm1	FVM	V/V	AVX512VL AVX512F	Move packed double-precision values in ymm1 to m256 using non-temporal hint.
EVEX.512.66.0F.W1 2B /r VMOVNTPD m512, zmm1	FVM	V/V	AVX512F	Move packed double-precision values in zmm1 to m512 using non-temporal hint.

### Instruction Operand Encoding<sup>1</sup>

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
FVM	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

### Description

Moves the packed double-precision floating-point values in the source operand (second operand) to the destination operand (first operand) using a non-temporal hint to prevent caching of the data during the write to memory. The source operand is an XMM register, YMM register or ZMM register, which is assumed to contain packed double-precision, floating-pointing data. The destination operand is a 128-bit, 256-bit or 512-bit memory location. The memory operand must be aligned on a 16-byte (128-bit version), 32-byte (VEX.256 encoded version) or 64-byte (EVEX.512 encoded version) boundary otherwise a general-protection exception (#GP) will be generated.

The non-temporal hint is implemented by using a write combining (WC) memory type protocol when writing the data to memory. Using this protocol, the processor does not write the data into the cache hierarchy, nor does it fetch the corresponding cache line from memory into the cache hierarchy. The memory type of the region being written to can override the non-temporal hint, if the memory address specified for the non-temporal store is in an uncacheable (UC) or write protected (WP) memory region. For more information on non-temporal stores, see “Caching of Temporal vs. Non-Temporal Data” in Chapter 10 in the IA-32 Intel Architecture Software Developer’s Manual, Volume 1.

Because the WC protocol uses a weakly-ordered memory consistency model, a fencing operation implemented with the SFENCE or MFENCE instruction should be used in conjunction with MOVNTPD instructions if multiple processors might use different memory types to read/write the destination memory locations.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b, VEX.L must be 0; otherwise instructions will #UD.

### Operation

#### VMOVNTPD (EVEX encoded versions)

VL = 128, 256, 512

DEST[VL-1:0] ← SRC[VL-1:0]

DEST[MAX\_VL-1:VL] ← 0

1. ModRM.MOD = 011B required

**MOVNTPD (Legacy and VEX versions)**

DEST ← SRC

**Intel C/C++ Compiler Intrinsic Equivalent**

VMOVNTPD void \_mm512\_stream\_pd(double \* p, \_\_m512d a);

VMOVNTPD void \_mm256\_stream\_pd (double \* p, \_\_m256d a);

MOVNTPD void \_mm\_stream\_pd (double \* p, \_\_m128d a);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type1.SSE2;

EVEX-encoded instruction, see Exceptions Type E1NF.

#UD                    If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

## MOVNTPS—Store Packed Single-Precision Floating-Point Values Using Non-Temporal Hint

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
OF 2B /r MOVNTPS m128, xmm1	MR	V/V	SSE	Move packed single-precision values xmm1 to mem using non-temporal hint.
VEX.128.OF.WIG 2B /r VMOVNTPS m128, xmm1	MR	V/V	AVX	Move packed single-precision values xmm1 to mem using non-temporal hint.
VEX.256.OF.WIG 2B /r VMOVNTPS m256, ymm1	MR	V/V	AVX	Move packed single-precision values ymm1 to mem using non-temporal hint.
EVEX.128.OF.W0 2B /r VMOVNTPS m128, xmm1	FVM	V/V	AVX512VL AVX512F	Move packed single-precision values in xmm1 to m128 using non-temporal hint.
EVEX.256.OF.W0 2B /r VMOVNTPS m256, ymm1	FVM	V/V	AVX512VL AVX512F	Move packed single-precision values in ymm1 to m256 using non-temporal hint.
EVEX.512.OF.W0 2B /r VMOVNTPS m512, zmm1	FVM	V/V	AVX512F	Move packed single-precision values in zmm1 to m512 using non-temporal hint.

### Instruction Operand Encoding<sup>1</sup>

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
FVM	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

### Description

Moves the packed single-precision floating-point values in the source operand (second operand) to the destination operand (first operand) using a non-temporal hint to prevent caching of the data during the write to memory. The source operand is an XMM register, YMM register or ZMM register, which is assumed to contain packed single-precision, floating-pointing. The destination operand is a 128-bit, 256-bit or 512-bit memory location. The memory operand must be aligned on a 16-byte (128-bit version), 32-byte (VEX.256 encoded version) or 64-byte (EVEX.512 encoded version) boundary otherwise a general-protection exception (#GP) will be generated.

The non-temporal hint is implemented by using a write combining (WC) memory type protocol when writing the data to memory. Using this protocol, the processor does not write the data into the cache hierarchy, nor does it fetch the corresponding cache line from memory into the cache hierarchy. The memory type of the region being written to can override the non-temporal hint, if the memory address specified for the non-temporal store is in an uncacheable (UC) or write protected (WP) memory region. For more information on non-temporal stores, see “Caching of Temporal vs. Non-Temporal Data” in Chapter 10 in the IA-32 Intel Architecture Software Developer’s Manual, Volume 1.

Because the WC protocol uses a weakly-ordered memory consistency model, a fencing operation implemented with the SFENCE or MFENCE instruction should be used in conjunction with MOVNTPS instructions if multiple processors might use different memory types to read/write the destination memory locations.

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

### Operation

#### VMOVNTPS (EVEX encoded versions)

VL = 128, 256, 512

DEST[VL-1:0] ← SRC[VL-1:0]

DEST[MAX\_VL-1:VL] ← 0

1. ModRM.MOD = 011B required

**MOVNTPS**

DEST ← SRC

**Intel C/C++ Compiler Intrinsic Equivalent**

VMOVNTPS void \_mm512\_stream\_ps(float \* p, \_\_m512d a);

MOVNTPS void \_mm\_stream\_ps (float \* p, \_\_m128d a);

VMOVNTPS void \_mm256\_stream\_ps (float \* p, \_\_m256 a);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type1.SSE; additionally

EVEX-encoded instruction, see Exceptions Type E1NF.

#UD                    If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

## MOVQ—Move Quadword

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
0F 6F /r MOVQ <i>mm</i> , <i>mm/m64</i>	RM	V/V	MMX	Move quadword from <i>mm/m64</i> to <i>mm</i> .
0F 7F /r MOVQ <i>mm/m64</i> , <i>mm</i>	MR	V/V	MMX	Move quadword from <i>mm</i> to <i>mm/m64</i> .
F3 0F 7E /r MOVQ <i>xmm1</i> , <i>xmm2/m64</i>	RM	V/V	SSE2	Move quadword from <i>xmm2/mem64</i> to <i>xmm1</i> .
VEX.128.F3.0F.WIG 7E /r VMOVQ <i>xmm1</i> , <i>xmm2/m64</i>	RM	V/V	AVX	Move quadword from <i>xmm2</i> to <i>xmm1</i> .
EVEX.128.F3.0F.W1 7E /r VMOVQ <i>xmm1</i> , <i>xmm2/m64</i>	T1S-RM	V/V	AVX512F	Move quadword from <i>xmm2/m64</i> to <i>xmm1</i> .
66 0F D6 /r MOVQ <i>xmm2/m64</i> , <i>xmm1</i>	MR	V/V	SSE2	Move quadword from <i>xmm1</i> to <i>xmm2/mem64</i> .
VEX.128.66.0F.WIG D6 /r VMOVQ <i>xmm1/m64</i> , <i>xmm2</i>	MR	V/V	AVX	Move quadword from <i>xmm2</i> register to <i>xmm1/m64</i> .
EVEX.128.66.0F.W1 D6 /r VMOVQ <i>xmm1/m64</i> , <i>xmm2</i>	T1S-MR	V/V	AVX512F	Move quadword from <i>xmm2</i> register to <i>xmm1/m64</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
T1S-RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
T1S-MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

### Description

Copies a quadword from the source operand (second operand) to the destination operand (first operand). The source and destination operands can be MMX technology registers, XMM registers, or 64-bit memory locations. This instruction can be used to move a quadword between two MMX technology registers or between an MMX technology register and a 64-bit memory location, or to move data between two XMM registers or between an XMM register and a 64-bit memory location. The instruction cannot be used to transfer data between memory locations.

When the source operand is an XMM register, the low quadword is moved; when the destination operand is an XMM register, the quadword is stored to the low quadword of the register, and the high quadword is cleared to all 0s.

In 64-bit mode and if not encoded using VEX/EVEX, use of the REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

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

If VMOVQ is encoded with VEX.L= 1, an attempt to execute the instruction encoded with VEX.L= 1 will cause an #UD exception.

## Operation

MOVQ instruction when operating on MMX technology registers and memory locations:  
 $DEST \leftarrow SRC;$

MOVQ instruction when source and destination operands are XMM registers:  
 $DEST[63:0] \leftarrow SRC[63:0];$   
 $DEST[127:64] \leftarrow 0000000000000000H;$

MOVQ instruction when source operand is XMM register and destination operand is memory location:  
 $DEST \leftarrow SRC[63:0];$

MOVQ instruction when source operand is memory location and destination operand is XMM register:  
 $DEST[63:0] \leftarrow SRC;$   
 $DEST[127:64] \leftarrow 0000000000000000H;$

VMOVQ (VEX.NDS.128.F3.0F 7E) with XMM register source and destination:  
 $DEST[63:0] \leftarrow SRC[63:0]$   
 $DEST[VLMAX-1:64] \leftarrow 0$

VMOVQ (VEX.128.66.0F D6) with XMM register source and destination:  
 $DEST[63:0] \leftarrow SRC[63:0]$   
 $DEST[VLMAX-1:64] \leftarrow 0$

**VMOVQ (7E - EVEX encoded version) with XMM register source and destination:**

$DEST[63:0] \leftarrow SRC[63:0]$   
 $DEST[MAX\_VL-1:64] \leftarrow 0$

**VMOVQ (D6 - EVEX encoded version) with XMM register source and destination:**

$DEST[63:0] \leftarrow SRC[63:0]$   
 $DEST[MAX\_VL-1:64] \leftarrow 0$

VMOVQ (7E) with memory source:

$DEST[63:0] \leftarrow SRC[63:0]$   
 $DEST[VLMAX-1:64] \leftarrow 0$

**VMOVQ (7E - EVEX encoded version) with memory source:**

$DEST[63:0] \leftarrow SRC[63:0]$   
 $DEST[:MAX\_VL-1:64] \leftarrow 0$

VMOVQ (D6) with memory dest:

$DEST[63:0] \leftarrow SRC2[63:0]$

## Flags Affected

None.

## Intel C/C++ Compiler Intrinsic Equivalent

VMOVQ `__m128i _mm_loadu_si64( void * s);`  
VMOVQ `void _mm_storeu_si64( void * d, __m128i s);`  
MOVQ `m128i _mm_mov_epi64(__m128i a)`

## SIMD Floating-Point Exceptions

None

## Other Exceptions

See Table 22-8, “Exception Conditions for Legacy SIMD/MMX Instructions without FP Exception,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B*.

## MOVSD—Move or Merge Scalar Double-Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 10 /r MOVSD xmm1, xmm2	RM	V/V	SSE2	Move scalar double-precision floating-point value from xmm2 to xmm1 register.
F2 0F 10 /r MOVSD xmm1, m64	RM	V/V	SSE2	Load scalar double-precision floating-point value from m64 to xmm1 register.
F2 0F 11 /r MOVSD xmm1/m64, xmm2	MR	V/V	SSE2	Move scalar double-precision floating-point value from xmm2 register to xmm1/m64.
VEX.NDS.LIG.F2.OF.WIG 10 /r VMOVSD xmm1, xmm2, xmm3	RVM	V/V	AVX	Merge scalar double-precision floating-point value from xmm2 and xmm3 to xmm1 register.
VEX.LIG.F2.OF.WIG 10 /r VMOVSD xmm1, m64	XM	V/V	AVX	Load scalar double-precision floating-point value from m64 to xmm1 register.
VEX.NDS.LIG.F2.OF.WIG 11 /r VMOVSD xmm1, xmm2, xmm3	MVR	V/V	AVX	Merge scalar double-precision floating-point value from xmm2 and xmm3 registers to xmm1.
VEX.LIG.F2.OF.WIG 11 /r VMOVSD m64, xmm1	MR	V/V	AVX	Store scalar double-precision floating-point value from xmm1 register to m64.
EVEX.NDS.LIG.F2.OF.W1 10 /r VMOVSD xmm1 {k1}{z}, xmm2, xmm3	RVM	V/V	AVX512F	Merge scalar double-precision floating-point value from xmm2 and xmm3 registers to xmm1 under writemask k1.
EVEX.LIG.F2.OF.W1 10 /r VMOVSD xmm1 {k1}{z}, m64	T1S-RM	V/V	AVX512F	Load scalar double-precision floating-point value from m64 to xmm1 register under writemask k1.
EVEX.NDS.LIG.F2.OF.W1 11 /r VMOVSD xmm1 {k1}{z}, xmm2, xmm3	MVR	V/V	AVX512F	Merge scalar double-precision floating-point value from xmm2 and xmm3 registers to xmm1 under writemask k1.
EVEX.LIG.F2.OF.W1 11 /r VMOVSD m64 {k1}, xmm1	T1S-MR	V/V	AVX512F	Store scalar double-precision floating-point value from xmm1 register to m64 under writemask k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
XM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
MVR	ModRM:r/m (w)	vvvv (r)	ModRM:reg (r)	NA
T1S-RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
T1S-MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA



**Description**

Moves a scalar double-precision floating-point value from the source operand (second operand) to the destination operand (first operand). The source and destination operands can be XMM registers or 64-bit memory locations. This instruction can be used to move a double-precision floating-point value to and from the low quadword of an XMM register and a 64-bit memory location, or to move a double-precision floating-point value between the low quadwords of two XMM registers. The instruction cannot be used to transfer data between memory locations.

Legacy version: When the source and destination operands are XMM registers, bits MAX\_VL:64 of the destination operand remains unchanged. When the source operand is a memory location and destination operand is an XMM registers, the quadword at bits 127:64 of the destination operand is cleared to all 0s, bits MAX\_VL:128 of the destination operand remains unchanged.

VEX and EVEX encoded register-register syntax: Moves a scalar double-precision floating-point value from the second source operand (the third operand) to the low quadword element of the destination operand (the first operand). Bits 127:64 of the destination operand are copied from the first source operand (the second operand). Bits (MAX\_VL-1:128) of the corresponding destination register are zeroed.

VEX and EVEX encoded memory store syntax: When the source operand is a memory location and destination operand is an XMM registers, bits MAX\_VL:64 of the destination operand is cleared to all 0s.

EVEX encoded versions: The low quadword of the destination is updated according to the writemask.

Note: For VMOVSD (memory store and load forms), VEX.vvvv and EVEX.vvvv are reserved and must be 1111b, otherwise instruction will #UD.

**Operation****VMOVSD (EVEX.NDS.LIG.F2.OF 10 /r: VMOVSD xmm1, m64 with support for 32 registers)**

```
IF k1[0] or *no writemask*
    THEN  DEST[63:0] ← SRC[63:0]
    ELSE
        IF *merging-masking*           ; merging-masking
            THEN *DEST[63:0] remains unchanged*
        ELSE                             ; zeroing-masking
            THEN DEST[63:0] ← 0
    FI;
FI;
DEST[511:64] ← 0
```

**VMOVSD (EVEX.NDS.LIG.F2.OF 11 /r: VMOVSD m64, xmm1 with support for 32 registers)**

```
IF k1[0] or *no writemask*
    THEN  DEST[63:0] ← SRC[63:0]
    ELSE  *DEST[63:0] remains unchanged*   ; merging-masking
FI;
```

**VMOVSD (EVEX.NDS.LIG.F2.OF 11 /r: VMOVSD xmm1, xmm2, xmm3)**

```
IF k1[0] or *no writemask*
    THEN  DEST[63:0] ← SRC2[63:0]
    ELSE
        IF *merging-masking*           ; merging-masking
            THEN *DEST[63:0] remains unchanged*
        ELSE                             ; zeroing-masking
            THEN DEST[63:0] ← 0
    FI;
FI;
DEST[127:64] ← SRC1[127:64]
DEST[MAX_VL-1:128] ← 0
```

**MOVSD (128-bit Legacy SSE version: MOVSD XMM1, XMM2)**

DEST[63:0] ←SRC[63:0]  
 DEST[MAX\_VL-1:64] (Unmodified)

**VMOVSD (VEX.NDS.128.F2.0F 11 /r: VMOVSD xmm1, xmm2, xmm3)**

DEST[63:0] ←SRC2[63:0]  
 DEST[127:64] ←SRC1[127:64]  
 DEST[MAX\_VL-1:128] ←0

**VMOVSD (VEX.NDS.128.F2.0F 10 /r: VMOVSD xmm1, xmm2, xmm3)**

DEST[63:0] ←SRC2[63:0]  
 DEST[127:64] ←SRC1[127:64]  
 DEST[MAX\_VL-1:128] ←0

**VMOVSD (VEX.NDS.128.F2.0F 10 /r: VMOVSD xmm1, m64)**

DEST[63:0] ←SRC[63:0]  
 DEST[MAX\_VL-1:64] ←0

**MOVSD/VMOVSD (128-bit versions: MOVSD m64, xmm1 or VMOVSD m64, xmm1)**

DEST[63:0] ←SRC[63:0]

**MOVSD (128-bit Legacy SSE version: MOVSD XMM1, m64)**

DEST[63:0] ←SRC[63:0]  
 DEST[127:64] ←0  
 DEST[MAX\_VL-1:128] (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**

VMOVSD \_\_m128d \_\_mm\_mask\_load\_sd(\_\_m128d s, \_\_mmask8 k, double \* p);  
 VMOVSD \_\_m128d \_\_mm\_maskz\_load\_sd(\_\_mmask8 k, double \* p);  
 VMOVSD \_\_m128d \_\_mm\_mask\_move\_sd(\_\_m128d sh, \_\_mmask8 k, \_\_m128d sl, \_\_m128d a);  
 VMOVSD \_\_m128d \_\_mm\_maskz\_move\_sd(\_\_mmask8 k, \_\_m128d s, \_\_m128d a);  
 VMOVSD void \_\_mm\_mask\_store\_sd(double \* p, \_\_mmask8 k, \_\_m128d s);  
 MOVSD \_\_m128d \_\_mm\_load\_sd (double \*p)  
 MOVSD void \_\_mm\_store\_sd (double \*p, \_\_m128d a)  
 MOVSD \_\_m128d \_\_mm\_move\_sd ( \_\_m128d a, \_\_m128d b)

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 5; additionally

#UD If VEX.vvvv != 1111B.

EVEX-encoded instruction, see Exceptions Type E10.

## MOVSHDUP—Replicate Single FP Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 16 /r MOVSHDUP xmm1, xmm2/m128	RM	V/V	SSE3	Move odd index single-precision floating-point values from xmm2/mem and duplicate each element into xmm1.
VEX.128.F3.0F.WIG 16 /r VMOVSHDUP xmm1, xmm2/m128	RM	V/V	AVX	Move odd index single-precision floating-point values from xmm2/mem and duplicate each element into xmm1.
VEX.256.F3.0F.WIG 16 /r VMOVSHDUP ymm1, ymm2/m256	RM	V/V	AVX	Move odd index single-precision floating-point values from ymm2/mem and duplicate each element into ymm1.
EVEX.128.F3.0F.W0 16 /r VMOVSHDUP xmm1 {k1}{z}, xmm2/m128	FVM	V/V	AVX512VL AVX512F	Move odd index single-precision floating-point values from xmm2/m128 and duplicate each element into xmm1 under writemask.
EVEX.256.F3.0F.W0 16 /r VMOVSHDUP ymm1 {k1}{z}, ymm2/m256	FVM	V/V	AVX512VL AVX512F	Move odd index single-precision floating-point values from ymm2/m256 and duplicate each element into ymm1 under writemask.
EVEX.512.F3.0F.W0 16 /r VMOVSHDUP zmm1 {k1}{z}, zmm2/m512	FVM	V/V	AVX512F	Move odd index single-precision floating-point values from zmm2/m512 and duplicate each element into zmm1 under writemask.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
FVM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Duplicates odd-indexed single-precision floating-point values from the source operand (the second operand) to adjacent element pair in the destination operand (the first operand). See Figure 4-3. The source operand is an XMM, YMM or ZMM register or 128, 256 or 512-bit memory location and the destination operand is an XMM, YMM or ZMM register.

128-bit Legacy SSE version: Bits (MAX\_VL-1:128) of the corresponding destination register remain unchanged.

VEX.128 encoded version: Bits (MAX\_VL-1:128) of the destination register are zeroed.

VEX.256 encoded version: Bits (MAX\_VL-1:256) of the destination register are zeroed.

EVEX encoded version: The destination operand is updated at 32-bit granularity according to the writemask.

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

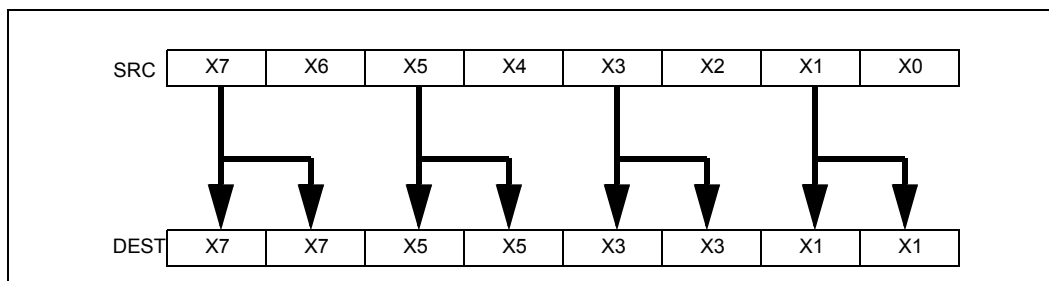


Figure 4-3. MOVSHDUP Operation

**Operation****VMOVSHDUP (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

TMP\_SRC[31:0] ← SRC[63:32]

TMP\_SRC[63:32] ← SRC[63:32]

TMP\_SRC[95:64] ← SRC[127:96]

TMP\_SRC[127:96] ← SRC[127:96]

IF VL ≥ 256

TMP\_SRC[159:128] ← SRC[191:160]

TMP\_SRC[191:160] ← SRC[191:160]

TMP\_SRC[223:192] ← SRC[255:224]

TMP\_SRC[255:224] ← SRC[255:224]

FI;

IF VL ≥ 512

TMP\_SRC[287:256] ← SRC[319:288]

TMP\_SRC[319:288] ← SRC[319:288]

TMP\_SRC[351:320] ← SRC[383:352]

TMP\_SRC[383:352] ← SRC[383:352]

TMP\_SRC[415:384] ← SRC[447:416]

TMP\_SRC[447:416] ← SRC[447:416]

TMP\_SRC[479:448] ← SRC[511:480]

TMP\_SRC[511:480] ← SRC[511:480]

FI;

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\*

THEN DEST[i+31:i] ← TMP\_SRC[i+31:i]

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+31:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VMOVSHDUP (VEX.256 encoded version)**

DEST[31:0] ← SRC[63:32]

DEST[63:32] ← SRC[63:32]

DEST[95:64] ← SRC[127:96]

DEST[127:96] ← SRC[127:96]

DEST[159:128] ← SRC[191:160]

DEST[191:160] ← SRC[191:160]

DEST[223:192] ← SRC[255:224]

DEST[255:224] ← SRC[255:224]

DEST[MAX\_VL-1:256] ← 0

**VMOVSHDUP (VEX.128 encoded version)**

DEST[31:0] ← SRC[63:32]

DEST[63:32] ← SRC[63:32]

DEST[95:64] ← SRC[127:96]

DEST[127:96] ← SRC[127:96]

DEST[MAX\_VL-1:128] ← 0

**MOVSHDUP (128-bit Legacy SSE version)**

DEST[31:0] ← SRC[63:32]  
 DEST[63:32] ← SRC[63:32]  
 DEST[95:64] ← SRC[127:96]  
 DEST[127:96] ← SRC[127:96]  
 DEST[MAX\_VL-1:128] (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**

VMOVSHDUP \_\_m512 \_\_mm512\_movehdup\_ps(\_\_m512 a);  
 VMOVSHDUP \_\_m512 \_\_mm512\_mask\_movehdup\_ps(\_\_m512 s, \_\_mmask16 k, \_\_m512 a);  
 VMOVSHDUP \_\_m512 \_\_mm512\_maskz\_movehdup\_ps(\_\_mmask16 k, \_\_m512 a);  
 VMOVSHDUP \_\_m256 \_\_mm256\_mask\_movehdup\_ps(\_\_m256 s, \_\_mmask8 k, \_\_m256 a);  
 VMOVSHDUP \_\_m256 \_\_mm256\_maskz\_movehdup\_ps(\_\_mmask8 k, \_\_m256 a);  
 VMOVSHDUP \_\_m128 \_\_mm\_mask\_movehdup\_ps(\_\_m128 s, \_\_mmask8 k, \_\_m128 a);  
 VMOVSHDUP \_\_m128 \_\_mm\_maskz\_movehdup\_ps(\_\_mmask8 k, \_\_m128 a);  
 VMOVSHDUP \_\_m256 \_\_mm256\_movehdup\_ps(\_\_m256 a);  
 VMOVSHDUP \_\_m128 \_\_mm\_movehdup\_ps(\_\_m128 a);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 4;

EVEX-encoded instruction, see Exceptions Type E4NF.nb.

#UD                    If EVEX.vvvv != 1111B or VEX.vvvv != 1111B.

## MOVSLDUP—Replicate Single FP Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 12 /r MOVSLDUP xmm1, xmm2/m128	A	V/V	SSE3	Move even index single-precision floating-point values from xmm2/mem and duplicate each element into xmm1.
VEX.128.F3.0F.WIG 12 /r VMOVSLDUP xmm1, xmm2/m128	RM	V/V	AVX	Move even index single-precision floating-point values from xmm2/mem and duplicate each element into xmm1.
VEX.256.F3.0F.WIG 12 /r VMOVSLDUP ymm1, ymm2/m256	RM	V/V	AVX	Move even index single-precision floating-point values from ymm2/mem and duplicate each element into ymm1.
EVEX.128.F3.0F.W0 12 /r VMOVSLDUP xmm1 {k1}{z}, xmm2/m128	FVM	V/V	AVX512VL AVX512F	Move even index single-precision floating-point values from xmm2/m128 and duplicate each element into xmm1 under writemask.
EVEX.256.F3.0F.W0 12 /r VMOVSLDUP ymm1 {k1}{z}, ymm2/m256	FVM	V/V	AVX512VL AVX512F	Move even index single-precision floating-point values from ymm2/m256 and duplicate each element into ymm1 under writemask.
EVEX.512.F3.0F.W0 12 /r VMOVSLDUP zmm1 {k1}{z}, zmm2/m512	FVM	V/V	AVX512F	Move even index single-precision floating-point values from zmm2/m512 and duplicate each element into zmm1 under writemask.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
FVM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Duplicates even-indexed single-precision floating-point values from the source operand (the second operand). See Figure 4-4. The source operand is an XMM, YMM or ZMM register or 128, 256 or 512-bit memory location and the destination operand is an XMM, YMM or ZMM register.

128-bit Legacy SSE version: Bits (MAX\_VL-1:128) of the corresponding destination register remain unchanged.

VEX.128 encoded version: Bits (MAX\_VL-1:128) of the destination register are zeroed.

VEX.256 encoded version: Bits (MAX\_VL-1:256) of the destination register are zeroed.

EVEX encoded version: The destination operand is updated at 32-bit granularity according to the writemask.

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

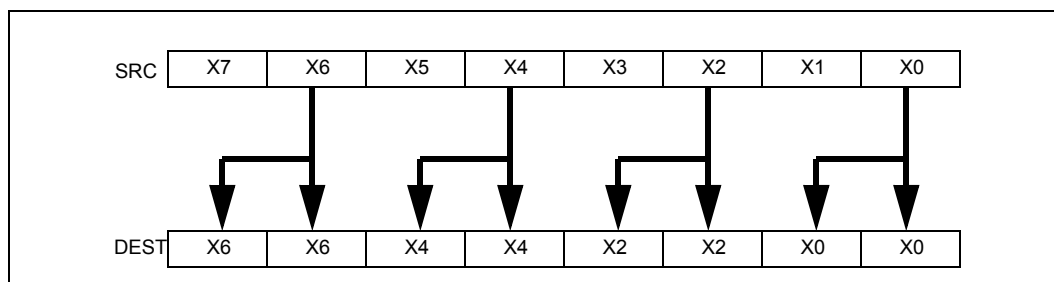


Figure 4-4. MOVSLDUP Operation

**Operation****VMOVSLDUP (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

TMP\_SRC[31:0] ← SRC[31:0]

TMP\_SRC[63:32] ← SRC[31:0]

TMP\_SRC[95:64] ← SRC[95:64]

TMP\_SRC[127:96] ← SRC[95:64]

IF VL ≥ 256

TMP\_SRC[159:128] ← SRC[159:128]

TMP\_SRC[191:160] ← SRC[159:128]

TMP\_SRC[223:192] ← SRC[223:192]

TMP\_SRC[255:224] ← SRC[223:192]

FI;

IF VL ≥ 512

TMP\_SRC[287:256] ← SRC[287:256]

TMP\_SRC[319:288] ← SRC[287:256]

TMP\_SRC[351:320] ← SRC[351:320]

TMP\_SRC[383:352] ← SRC[351:320]

TMP\_SRC[415:384] ← SRC[415:384]

TMP\_SRC[447:416] ← SRC[415:384]

TMP\_SRC[479:448] ← SRC[479:448]

TMP\_SRC[511:480] ← SRC[479:448]

FI;

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\*

THEN DEST[i+31:i] ← TMP\_SRC[i+31:i]

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+31:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VMOVSLDUP (VEX.256 encoded version)**

DEST[31:0] ← SRC[31:0]

DEST[63:32] ← SRC[31:0]

DEST[95:64] ← SRC[95:64]

DEST[127:96] ← SRC[95:64]

DEST[159:128] ← SRC[159:128]

DEST[191:160] ← SRC[159:128]

DEST[223:192] ← SRC[223:192]

DEST[255:224] ← SRC[223:192]

DEST[MAX\_VL-1:256] ← 0

**VMOVSLDUP (VEX.128 encoded version)**

DEST[31:0] ← SRC[31:0]

DEST[63:32] ← SRC[31:0]

DEST[95:64] ← SRC[95:64]

DEST[127:96] ← SRC[95:64]

DEST[MAX\_VL-1:128] ← 0

**MOVSLDUP (128-bit Legacy SSE version)**

DEST[31:0] ← SRC[31:0]  
 DEST[63:32] ← SRC[31:0]  
 DEST[95:64] ← SRC[95:64]  
 DEST[127:96] ← SRC[95:64]  
 DEST[MAX\_VL-1:128] (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**

VMOVSLDUP \_\_m512 \_\_mm512\_moveldup\_ps( \_\_m512 a);  
 VMOVSLDUP \_\_m512 \_\_mm512\_mask\_moveldup\_ps(\_\_m512 s, \_\_mmask16 k, \_\_m512 a);  
 VMOVSLDUP \_\_m512 \_\_mm512\_maskz\_moveldup\_ps( \_\_mmask16 k, \_\_m512 a);  
 VMOVSLDUP \_\_m256 \_\_mm256\_mask\_moveldup\_ps(\_\_m256 s, \_\_mmask8 k, \_\_m256 a);  
 VMOVSLDUP \_\_m256 \_\_mm256\_maskz\_moveldup\_ps( \_\_mmask8 k, \_\_m256 a);  
 VMOVSLDUP \_\_m128 \_\_mm\_mask\_moveldup\_ps(\_\_m128 s, \_\_mmask8 k, \_\_m128 a);  
 VMOVSLDUP \_\_m128 \_\_mm\_maskz\_moveldup\_ps( \_\_mmask8 k, \_\_m128 a);  
 VMOVSLDUP \_\_m256 \_\_mm256\_moveldup\_ps ( \_\_m256 a);  
 VMOVSLDUP \_\_m128 \_\_mm\_moveldup\_ps ( \_\_m128 a);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 4;

EVEX-encoded instruction, see Exceptions Type E4NF.nb.

#UD                    If EVEX.vvvv != 1111B or VEX.vvvv != 1111B.



## MOVSS—Move or Merge Scalar Single-Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 10 /r MOVSS xmm1, xmm2	RM	V/V	SSE	Merge scalar single-precision floating-point value from xmm2 to xmm1 register.
F3 0F 10 /r MOVSS xmm1, m32	RM	V/V	SSE	Load scalar single-precision floating-point value from m32 to xmm1 register.
VEX.NDS.LIG.F3.0F.WIG 10 /r VMOVSS xmm1, xmm2, xmm3	RVM	V/V	AVX	Merge scalar single-precision floating-point value from xmm2 and xmm3 to xmm1 register
VEX.LIG.F3.0F.WIG 10 /r VMOVSS xmm1, m32	XM	V/V	AVX	Load scalar single-precision floating-point value from m32 to xmm1 register.
F3 0F 11 /r MOVSS xmm2/m32, xmm1	MR	V/V	SSE	Move scalar single-precision floating-point value from xmm1 register to xmm2/m32.
VEX.NDS.LIG.F3.0F.WIG 11 /r VMOVSS xmm1, xmm2, xmm3	MVR	V/V	AVX	Move scalar single-precision floating-point value from xmm2 and xmm3 to xmm1 register.
VEX.LIG.F3.0F.WIG 11 /r VMOVSS m32, xmm1	MR	V/V	AVX	Move scalar single-precision floating-point value from xmm1 register to m32.
EVEX.NDS.LIG.F3.0F.W0 10 /r VMOVSS xmm1 {k1}{z}, xmm2, xmm3	RVM	V/V	AVX512F	Move scalar single-precision floating-point value from xmm2 and xmm3 to xmm1 register under writemask k1.
EVEX.LIG.F3.0F.W0 10 /r VMOVSS xmm1 {k1}{z}, m32	T1S-RM	V/V	AVX512F	Move scalar single-precision floating-point values from m32 to xmm1 under writemask k1.
EVEX.NDS.LIG.F3.0F.W0 11 /r VMOVSS xmm1 {k1}{z}, xmm2, xmm3	MVR	V/V	AVX512F	Move scalar single-precision floating-point value from xmm2 and xmm3 to xmm1 register under writemask k1.
EVEX.LIG.F3.0F.W0 11 /r VMOVSS m32 {k1}, xmm1	T1S-MR	V/V	AVX512F	Move scalar single-precision floating-point values from xmm1 to m32 under writemask k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
XM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
MVR	ModRM:r/m (w)	vvvv (r)	ModRM:reg (r)	NA
T1S-RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
T1S-MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

## Description

Moves a scalar single-precision floating-point value 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 single-precision floating-point value to and from the low doubleword of an XMM register and a 32-bit memory location, or to move a single-precision floating-point value between the low doublewords of two XMM registers. The instruction cannot be used to transfer data between memory locations.

Legacy version: When the source and destination operands are XMM registers, bits (MAX\_VL-1:32) of the corresponding destination register are unmodified. When the source operand is a memory location and destination operand is an XMM registers, Bits (127:32) of the destination operand is cleared to all 0s, bits MAX\_VL:128 of the destination operand remains unchanged.

VEX and EVEX encoded register-register syntax: Moves a scalar single-precision floating-point value from the second source operand (the third operand) to the low doubleword element of the destination operand (the first operand). Bits 127:32 of the destination operand are copied from the first source operand (the second operand). Bits (MAX\_VL-1:128) of the corresponding destination register are zeroed.

VEX and EVEX encoded memory load syntax: When the source operand is a memory location and destination operand is an XMM registers, bits MAX\_VL:32 of the destination operand is cleared to all 0s.

EVEX encoded versions: The low doubleword of the destination is updated according to the writemask.

Note: For memory store form instruction "VMOVSS m32, xmm1", VEX.vvvv is reserved and must be 1111b otherwise instruction will #UD. For memory store form instruction "VMOVSS mv {k1}, xmm1", EVEX.vvvv is reserved and must be 1111b otherwise instruction will #UD.

Software should ensure VMOVSS is encoded with VEX.L=0. Encoding VMOVSS with VEX.L=1 may encounter unpredictable behavior across different processor generations.

## Operation

**VMOVSS (EVEX.NDS.LIG.F3.OF.W0 11 /r when the source operand is memory and the destination is an XMM register)**

IF k1[0] or \*no writemask\*

THEN DEST[31:0] ← SRC[31:0]

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[31:0] remains unchanged\*

ELSE ; zeroing-masking

THEN DEST[31:0] ← 0

FI;

FI;

DEST[511:32] ← 0

**VMOVSS (EVEX.NDS.LIG.F3.OF.W0 10 /r when the source operand is an XMM register and the destination is memory)**

IF k1[0] or \*no writemask\*

THEN DEST[31:0] ← SRC[31:0]

ELSE \*DEST[31:0] remains unchanged\* ; merging-masking

FI;

**VMOVSS (EVEX.NDS.LIG.F3.0F.W0 10/11 /r where the source and destination are XMM registers)**

```

IF k1[0] or *no writemask*
  THEN  DEST[31:0] ← SRC2[31:0]
  ELSE
    IF *merging-masking*           ; merging-masking
      THEN *DEST[31:0] remains unchanged*
    ELSE                             ; zeroing-masking
      THEN DEST[31:0] ← 0
  FI;
FI;
DEST[127:32] ← SRC1[127:32]
DEST[MAX_VL-1:128] ← 0

```

**MOVSS (Legacy SSE version when the source and destination operands are both XMM registers)**

```

DEST[31:0] ← SRC[31:0]
DEST[MAX_VL-1:32] (Unmodified)

```

**VMOVSS (VEX.NDS.128.F3.0F 11 /r where the destination is an XMM register)**

```

DEST[31:0] ← SRC2[31:0]
DEST[127:32] ← SRC1[127:32]
DEST[MAX_VL-1:128] ← 0

```

**VMOVSS (VEX.NDS.128.F3.0F 10 /r where the source and destination are XMM registers)**

```

DEST[31:0] ← SRC2[31:0]
DEST[127:32] ← SRC1[127:32]
DEST[MAX_VL-1:128] ← 0

```

**VMOVSS (VEX.NDS.128.F3.0F 10 /r when the source operand is memory and the destination is an XMM register)**

```

DEST[31:0] ← SRC[31:0]
DEST[MAX_VL-1:32] ← 0

```

**MOVSS/VMOVSS (when the source operand is an XMM register and the destination is memory)**

```

DEST[31:0] ← SRC[31:0]

```

**MOVSS (Legacy SSE version when the source operand is memory and the destination is an XMM register)**

```

DEST[31:0] ← SRC[31:0]
DEST[127:32] ← 0
DEST[MAX_VL-1:128] (Unmodified)

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VMOVSS __m128 __mm_mask_load_ss(__m128 s, __mmask8 k, float * p);
VMOVSS __m128 __mm_maskz_load_ss(__mmask8 k, float * p);
VMOVSS __m128 __mm_mask_move_ss(__m128 sh, __mmask8 k, __m128 sl, __m128 a);
VMOVSS __m128 __mm_maskz_move_ss(__mmask8 k, __m128 s, __m128 a);
VMOVSS void __mm_mask_store_ss(float * p, __mmask8 k, __m128 a);
MOVSS __m128 __mm_load_ss(float * p)
MOVSS void __mm_store_ss(float * p, __m128 a)
MOVSS __m128 __mm_move_ss(__m128 a, __m128 b)

```

**SIMD Floating-Point Exceptions**

None



### Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 5; additionally

#UD If VEX.vvvv != 1111B.

EVEX-encoded instruction, see Exceptions Type E10.

## MOVUPD—Move Unaligned Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 10 /r MOVUPD xmm1, xmm2/m128	RM	V/V	SSE2	Move unaligned packed double-precision floating-point from xmm2/mem to xmm1.
66 0F 11 /r MOVUPD xmm2/m128, xmm1	MR	V/V	SSE2	Move unaligned packed double-precision floating-point from xmm1 to xmm2/mem.
VEX.128.66.0F.WIG 10 /r VMOVUPD xmm1, xmm2/m128	RM	V/V	AVX	Move unaligned packed double-precision floating-point from xmm2/mem to xmm1.
VEX.128.66.0F.WIG 11 /r VMOVUPD xmm2/m128, xmm1	MR	V/V	AVX	Move unaligned packed double-precision floating-point from xmm1 to xmm2/mem.
VEX.256.66.0F.WIG 10 /r VMOVUPD ymm1, ymm2/m256	RM	V/V	AVX	Move unaligned packed double-precision floating-point from ymm2/mem to ymm1.
VEX.256.66.0F.WIG 11 /r VMOVUPD ymm2/m256, ymm1	MR	V/V	AVX	Move unaligned packed double-precision floating-point from ymm1 to ymm2/mem.
EVEX.128.66.0F.W1 10 /r VMOVUPD xmm1 {k1}{z}, xmm2/m128	FVM-RM	V/V	AVX512VL AVX512F	Move unaligned packed double-precision floating-point from xmm2/m128 to xmm1 using writemask k1.
EVEX.128.66.0F.W1 11 /r VMOVUPD xmm2/m128 {k1}{z}, xmm1	FVM-MR	V/V	AVX512VL AVX512F	Move unaligned packed double-precision floating-point from xmm1 to xmm2/m128 using writemask k1.
EVEX.256.66.0F.W1 10 /r VMOVUPD ymm1 {k1}{z}, ymm2/m256	FVM-RM	V/V	AVX512VL AVX512F	Move unaligned packed double-precision floating-point from ymm2/m256 to ymm1 using writemask k1.
EVEX.256.66.0F.W1 11 /r VMOVUPD ymm2/m256 {k1}{z}, ymm1	FVM-MR	V/V	AVX512VL AVX512F	Move unaligned packed double-precision floating-point from ymm1 to ymm2/m256 using writemask k1.
EVEX.512.66.0F.W1 10 /r VMOVUPD zmm1 {k1}{z}, zmm2/m512	FVM-RM	V/V	AVX512F	Move unaligned packed double-precision floating-point values from zmm2/m512 to zmm1 using writemask k1.
EVEX.512.66.0F.W1 11 /r VMOVUPD zmm2/m512 {k1}{z}, zmm1	FVM-MR	V/V	AVX512F	Move unaligned packed double-precision floating-point values from zmm1 to zmm2/m512 using writemask k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
FVM-RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
FVM-MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

### Description

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

#### EVEX.512 encoded version:

Moves 512 bits of packed double-precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load a ZMM register from a float64 memory location, to store the contents of a ZMM register into a memory. The destination operand is updated according to the writemask.

**VEX.256 encoded version:**

Moves 256 bits of packed double-precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load a YMM register from a 256-bit memory location, to store the contents of a YMM register into a 256-bit memory location, or to move data between two YMM registers. Bits (MAX\_VL-1:256) of the destination register are zeroed.

128-bit versions:

Moves 128 bits of packed double-precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load an XMM register from a 128-bit memory location, to store the contents of an XMM register into a 128-bit memory location, or to move data between two XMM registers.

**128-bit Legacy SSE version:** Bits (MAX\_VL-1:128) of the corresponding destination register remain unchanged.

When the source or destination operand is a memory operand, the operand may be unaligned on a 16-byte boundary without causing a general-protection exception (#GP) to be generated

**VEX.128 and EVEX.128 encoded versions:** Bits (MAX\_VL-1:128) of the destination register are zeroed.

**Operation****VMOVUPD (EVEX encoded versions, register-copy form)**

(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[MAX\_VL-1:VL] ← 0

**VMOVUPD (EVEX encoded versions, store-form)**

(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 \*DEST[i+63:i] remains unchanged\* ; merging-masking

  FI;

ENDFOR;

**VMOVUPD (EVEX encoded versions, load-form)**

(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[MAX\_VL-1:VL] ← 0

VMOVUPD (VEX.256 encoded version, load - and register copy)

DEST[255:0] ← SRC[255:0]

DEST[MAX\_VL-1:256] ← 0

**VMOVUPD (VEX.256 encoded version, store-form)**

DEST[255:0] ← SRC[255:0]

**VMOVUPD (VEX.128 encoded version)**

DEST[127:0] ← SRC[127:0]

DEST[MAX\_VL-1:128] ← 0

**MOVUPD (128-bit load- and register-copy- form Legacy SSE version)**

DEST[127:0] ← SRC[127:0]

DEST[MAX\_VL-1:128] (Unmodified)

**(V)MOVUPD (128-bit store-form version)**

DEST[127:0] ← SRC[127:0]

**Intel C/C++ Compiler Intrinsic Equivalent**

VMOVUPD \_\_m512d \_\_mm512\_loadu\_pd( void \* s);

VMOVUPD \_\_m512d \_\_mm512\_mask\_loadu\_pd(\_\_m512d a, \_\_mmask8 k, void \* s);

VMOVUPD \_\_m512d \_\_mm512\_maskz\_loadu\_pd( \_\_mmask8 k, void \* s);

VMOVUPD void \_\_mm512\_storeu\_pd( void \* d, \_\_m512d a);

VMOVUPD void \_\_mm512\_mask\_storeu\_pd( void \* d, \_\_mmask8 k, \_\_m512d a);

VMOVUPD \_\_m256d \_\_mm256\_mask\_loadu\_pd(\_\_m256d s, \_\_mmask8 k, void \* m);

VMOVUPD \_\_m256d \_\_mm256\_maskz\_loadu\_pd( \_\_mmask8 k, void \* m);

VMOVUPD void \_\_mm256\_mask\_storeu\_pd( void \* d, \_\_mmask8 k, \_\_m256d a);

VMOVUPD \_\_m128d \_\_mm\_mask\_loadu\_pd(\_\_m128d s, \_\_mmask8 k, void \* m);

VMOVUPD \_\_m128d \_\_mm\_maskz\_loadu\_pd( \_\_mmask8 k, void \* m);

VMOVUPD void \_\_mm\_mask\_storeu\_pd( void \* d, \_\_mmask8 k, \_\_m128d a);

MOVUPD \_\_m256d \_\_mm256\_loadu\_pd( double \* p);

MOVUPD void \_\_mm256\_storeu\_pd( double \*p, \_\_m256d a);

MOVUPD \_\_m128d \_\_mm\_loadu\_pd( double \* p);

MOVUPD void \_\_mm\_storeu\_pd( double \*p, \_\_m128d a);

**SIMD Floating-Point Exceptions**

None



### Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

Note treatment of #AC varies; additionally

#UD                      If VEX.vvvv != 1111B.

EVEX-encoded instruction, see Exceptions Type E4.nb.



## MOVUPS—Move Unaligned Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
OF 10 /r MOVUPS xmm1, xmm2/m128	RM	V/V	SSE	Move unaligned packed single-precision floating-point from xmm2/mem to xmm1.
OF 11 /r MOVUPS xmm2/m128, xmm1	MR	V/V	SSE	Move unaligned packed single-precision floating-point from xmm1 to xmm2/mem.
VEX.128.OF.WIG 10 /r VMOVUPS xmm1, xmm2/m128	RM	V/V	AVX	Move unaligned packed single-precision floating-point from xmm2/mem to xmm1.
VEX.128.OF 11.WIG /r VMOVUPS xmm2/m128, xmm1	MR	V/V	AVX	Move unaligned packed single-precision floating-point from xmm1 to xmm2/mem.
VEX.256.OF 10.WIG /r VMOVUPS ymm1, ymm2/m256	RM	V/V	AVX	Move unaligned packed single-precision floating-point from ymm2/mem to ymm1.
VEX.256.OF 11.WIG /r VMOVUPS ymm2/m256, ymm1	MR	V/V	AVX	Move unaligned packed single-precision floating-point from ymm1 to ymm2/mem.
EVEX.128.OF.W0 10 /r VMOVUPS xmm1 {k1}{z}, xmm2/m128	FVM-RM	V/V	AVX512VL AVX512F	Move unaligned packed single-precision floating-point values from xmm2/m128 to xmm1 using writemask k1.
EVEX.256.OF.W0 10 /r VMOVUPS ymm1 {k1}{z}, ymm2/m256	FVM-RM	V/V	AVX512VL AVX512F	Move unaligned packed single-precision floating-point values from ymm2/m256 to ymm1 using writemask k1.
EVEX.512.OF.W0 10 /r VMOVUPS zmm1 {k1}{z}, zmm2/m512	FVM-RM	V/V	AVX512F	Move unaligned packed single-precision floating-point values from zmm2/m512 to zmm1 using writemask k1.
EVEX.128.OF.W0 11 /r VMOVUPS xmm2/m128 {k1}{z}, xmm1	FVM-MR	V/V	AVX512VL AVX512F	Move unaligned packed single-precision floating-point values from xmm1 to xmm2/m128 using writemask k1.
EVEX.256.OF.W0 11 /r VMOVUPS ymm2/m256 {k1}{z}, ymm1	FVM-MR	V/V	AVX512VL AVX512F	Move unaligned packed single-precision floating-point values from ymm1 to ymm2/m256 using writemask k1.
EVEX.512.OF.W0 11 /r VMOVUPS zmm2/m512 {k1}{z}, zmm1	FVM-MR	V/V	AVX512F	Move unaligned packed single-precision floating-point values from zmm1 to zmm2/m512 using writemask k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
FVM-RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
RVM-MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

### Description

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

#### EVEX.512 encoded version:

Moves 512 bits of packed single-precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load a ZMM register from a 512-bit float32 memory location, to store the contents of a ZMM register into memory. The destination operand is updated according to the writemask.

**VEX.256 and EVEX.256 encoded versions:**

Moves 256 bits of packed single-precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load a YMM register from a 256-bit memory location, to store the contents of a YMM register into a 256-bit memory location, or to move data between two YMM registers. Bits (MAX\_VL-1:256) of the destination register are zeroed.

128-bit versions:

Moves 128 bits of packed single-precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load an XMM register from a 128-bit memory location, to store the contents of an XMM register into a 128-bit memory location, or to move data between two XMM registers.

**128-bit Legacy SSE version:** Bits (MAX\_VL-1:128) of the corresponding destination register remain unchanged.

When the source or destination operand is a memory operand, the operand may be unaligned without causing a general-protection exception (#GP) to be generated.

**VEX.128 and EVEX.128 encoded versions:** Bits (MAX\_VL-1:128) of the destination register are zeroed.

**Operation****VMOVUPS (EVEX encoded versions, register-copy form)**

(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[MAX\_VL-1:VL] ← 0

**VMOVUPS (EVEX encoded versions, store-form)**

(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 \*DEST[i+31:i] remains unchanged\* ; merging-masking

  FI;

ENDFOR;

**VMOVUPS (EVEX encoded versions, load-form)**

(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[MAX\_VL-1:VL] ← 0

**VMOVUPS (VEX.256 encoded version, load - and register copy)**

DEST[255:0] ← SRC[255:0]

DEST[MAX\_VL-1:256] ← 0

**VMOVUPS (VEX.256 encoded version, store-form)**

DEST[255:0] ← SRC[255:0]

**VMOVUPS (VEX.128 encoded version)**

DEST[127:0] ← SRC[127:0]

DEST[MAX\_VL-1:128] ← 0

**MOVUPS (128-bit load- and register-copy- form Legacy SSE version)**

DEST[127:0] ← SRC[127:0]

DEST[MAX\_VL-1:128] (Unmodified)

**(V)MOVUPS (128-bit store-form version)**

DEST[127:0] ← SRC[127:0]

**Intel C/C++ Compiler Intrinsic Equivalent**

VMOVUPS \_\_m512 \_\_mm512\_loadu\_ps( void \* s);

VMOVUPS \_\_m512 \_\_mm512\_mask\_loadu\_ps(\_\_m512 a, \_\_mmask16 k, void \* s);

VMOVUPS \_\_m512 \_\_mm512\_maskz\_loadu\_ps( \_\_mmask16 k, void \* s);

VMOVUPS void \_\_mm512\_storeu\_ps( void \* d, \_\_m512 a);

VMOVUPS void \_\_mm512\_mask\_storeu\_ps( void \* d, \_\_mmask8 k, \_\_m512 a);

VMOVUPS \_\_m256 \_\_mm256\_mask\_loadu\_ps(\_\_m256 a, \_\_mmask8 k, void \* s);

VMOVUPS \_\_m256 \_\_mm256\_maskz\_loadu\_ps( \_\_mmask8 k, void \* s);

VMOVUPS void \_\_mm256\_mask\_storeu\_ps( void \* d, \_\_mmask8 k, \_\_m256 a);

VMOVUPS \_\_m128 \_\_mm\_mask\_loadu\_ps(\_\_m128 a, \_\_mmask8 k, void \* s);

VMOVUPS \_\_m128 \_\_mm\_maskz\_loadu\_ps( \_\_mmask8 k, void \* s);

VMOVUPS void \_\_mm\_mask\_storeu\_ps( void \* d, \_\_mmask8 k, \_\_m128 a);

MOVUPS \_\_m256 \_\_mm256\_loadu\_ps ( float \* p);

MOVUPS void \_\_mm256\_storeu\_ps( float \*p, \_\_m256 a);

MOVUPS \_\_m128 \_\_mm\_loadu\_ps ( float \* p);

MOVUPS void \_\_mm\_storeu\_ps( float \*p, \_\_m128 a);

**SIMD Floating-Point Exceptions**

None



### Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

Note treatment of #AC varies;

EVEX-encoded instruction, see Exceptions Type E4.nb.

#UD                    If EVEX.vvvv != 1111B or VEX.vvvv != 1111B.

## MULPD—Multiply Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 59 /r MULPD xmm1, xmm2/m128	RM	V/V	SSE2	Multiply packed double-precision floating-point values in xmm2/m128 with xmm1 and store result in xmm1.
VEX.NDS.128.66.0F.WIG 59 /r VMULPD xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Multiply packed double-precision floating-point values in xmm3/m128 with xmm2 and store result in xmm1.
VEX.NDS.256.66.0F.WIG 59 /r VMULPD ymm1, ymm2, ymm3/m256	RVM	V/V	AVX	Multiply packed double-precision floating-point values in ymm3/m256 with ymm2 and store result in ymm1.
EVEX.NDS.128.66.0F.W1 59 /r VMULPD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	FV	V/V	AVX512VL AVX512F	Multiply packed double-precision floating-point values from xmm3/m128/m64bcst to xmm2 and store result in xmm1.
EVEX.NDS.256.66.0F.W1 59 /r VMULPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	FV	V/V	AVX512VL AVX512F	Multiply packed double-precision floating-point values from ymm3/m256/m64bcst to ymm2 and store result in ymm1.
EVEX.NDS.512.66.0F.W1 59 /r VMULPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	FV	V/V	AVX512F	Multiply packed double-precision floating-point values in zmm3/m512/m64bcst with zmm2 and store result in zmm1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FV	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Multiply packed double-precision floating-point values from the first source operand with corresponding values in the second source operand, and stores the packed double-precision floating-point results in the destination operand.

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 64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register. Bits (MAX\_VL-1:256) of the corresponding destination ZMM register are zeroed.

VEX.128 encoded version: The first source operand is a XMM register. The second source operand can be a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper bits (MAX\_VL-1:128) of the destination YMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAX\_VL-1:128) of the corresponding ZMM register destination are unmodified.

**Operation****VMULPD (EVEX encoded versions)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

IF (VL = 512) AND (EVEX.b = 1) AND SRC2 \*is a register\*

```

THEN
    SET_RM(EVEX.RC);
ELSE
    SET_RM(MXCSR.RM);
FI;
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN
                    DEST[i+63:i] ← SRC1[i+63:i] * SRC2[63:0]
                ELSE
                    DEST[i+63:i] ← SRC1[i+63:i] * SRC2[i+63:i]
            FI;
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+63:i] ← 0
            FI
        FI;
    ENDFOR
    DEST[MAX_VL-1:VL] ← 0

```

**VMULPD (VEX.256 encoded version)**

DEST[63:0] ← SRC1[63:0] \* SRC2[63:0]

DEST[127:64] ← SRC1[127:64] \* SRC2[127:64]

DEST[191:128] ← SRC1[191:128] \* SRC2[191:128]

DEST[255:192] ← SRC1[255:192] \* SRC2[255:192]

DEST[MAX\_VL-1:256] ← 0;

**VMULPD (VEX.128 encoded version)**

DEST[63:0] ← SRC1[63:0] \* SRC2[63:0]

DEST[127:64] ← SRC1[127:64] \* SRC2[127:64]

DEST[MAX\_VL-1:128] ← 0

**MULPD (128-bit Legacy SSE version)**

DEST[63:0] ← DEST[63:0] \* SRC[63:0]

DEST[127:64] ← DEST[127:64] \* SRC[127:64]

DEST[MAX\_VL-1:128] (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VMULPD __m512d __mm512_mul_pd( __m512d a, __m512d b);
VMULPD __m512d __mm512_mask_mul_pd(__m512d s, __mmask8 k, __m512d a, __m512d b);
VMULPD __m512d __mm512_maskz_mul_pd( __mmask8 k, __m512d a, __m512d b);
VMULPD __m512d __mm512_mul_round_pd( __m512d a, __m512d b, int);
VMULPD __m512d __mm512_mask_mul_round_pd(__m512d s, __mmask8 k, __m512d a, __m512d b, int);
VMULPD __m512d __mm512_maskz_mul_round_pd( __mmask8 k, __m512d a, __m512d b, int);
VMULPD __m256d __mm256_mul_pd( __m256d a, __m256d b);
MULPD __m128d __mm_mul_pd( __m128d a, __m128d b);

```

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 2.

EVEX-encoded instruction, see Exceptions Type E2.

## MULPS—Multiply Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
OF 59 /r MULPS xmm1, xmm2/m128	RM	V/V	SSE	Multiply packed single-precision floating-point values in xmm2/m128 with xmm1 and store result in xmm1.
VEX.NDS.128.OF.WIG 59 /r VMULPS xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Multiply packed single-precision floating-point values in xmm3/m128 with xmm2 and store result in xmm1.
VEX.NDS.256.OF.WIG 59 /r VMULPS ymm1, ymm2, ymm3/m256	RVM	V/V	AVX	Multiply packed single-precision floating-point values in ymm3/m256 with ymm2 and store result in ymm1.
EVEX.NDS.128.OF.W0 59 /r VMULPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	FV	V/V	AVX512VL AVX512F	Multiply packed single-precision floating-point values from xmm3/m128/m32bcst to xmm2 and store result in xmm1.
EVEX.NDS.256.OF.W0 59 /r VMULPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	FV	V/V	AVX512VL AVX512F	Multiply packed single-precision floating-point values from ymm3/m256/m32bcst to ymm2 and store result in ymm1.
EVEX.NDS.512.OF.W0 59 /r VMULPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst {er}	FV	V/V	AVX512F	Multiply packed single-precision floating-point values in zmm3/m512/m32bcst with zmm2 and store result in zmm1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FV	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Multiply the packed single-precision floating-point values from the first source operand with the corresponding values in the second source operand, and stores the packed double-precision floating-point results in the destination operand.

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 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register. Bits (MAX\_VL-1:256) of the corresponding destination ZMM register are zeroed.

VEX.128 encoded version: The first source operand is a XMM register. The second source operand can be a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper bits (MAX\_VL-1:128) of the destination YMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAX\_VL-1:128) of the corresponding ZMM register destination are unmodified.



**Operation****VMULPS (EVEX encoded version)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

IF (VL = 512) AND (EVEX.b = 1) AND SRC2 \*is a register\*

```

THEN
    SET_RM(EVEX.RC);
ELSE
    SET_RM(MXCSR.RM);
FI;
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN
                    DEST[i+31:i] ← SRC1[i+31:i] * SRC2[31:0]
                ELSE
                    DEST[i+31:i] ← SRC1[i+31:i] * SRC2[i+31:i]
            FI;
        ELSE
            IF *merging-masking*           ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
            ELSE                             ; zeroing-masking
                DEST[i+31:i] ← 0
            FI
        FI;
    ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VMULPS (VEX.256 encoded version)**

```

DEST[31:0] ← SRC1[31:0] * SRC2[31:0]
DEST[63:32] ← SRC1[63:32] * SRC2[63:32]
DEST[95:64] ← SRC1[95:64] * SRC2[95:64]
DEST[127:96] ← SRC1[127:96] * SRC2[127:96]
DEST[159:128] ← SRC1[159:128] * SRC2[159:128]
DEST[191:160] ← SRC1[191:160] * SRC2[191:160]
DEST[223:192] ← SRC1[223:192] * SRC2[223:192]
DEST[255:224] ← SRC1[255:224] * SRC2[255:224].
DEST[MAX_VL-1:256] ← 0;

```

**VMULPS (VEX.128 encoded version)**

```

DEST[31:0] ← SRC1[31:0] * SRC2[31:0]
DEST[63:32] ← SRC1[63:32] * SRC2[63:32]
DEST[95:64] ← SRC1[95:64] * SRC2[95:64]
DEST[127:96] ← SRC1[127:96] * SRC2[127:96]
DEST[MAX_VL-1:128] ← 0

```

**MULPS (128-bit Legacy SSE version)**

```

DEST[31:0] ← SRC1[31:0] * SRC2[31:0]
DEST[63:32] ← SRC1[63:32] * SRC2[63:32]
DEST[95:64] ← SRC1[95:64] * SRC2[95:64]
DEST[127:96] ← SRC1[127:96] * SRC2[127:96]
DEST[MAX_VL-1:128] (Unmodified)

```

**Intel C/C++ Compiler Intrinsic Equivalent**

VMULPS \_\_m512 \_\_mm512\_mul\_ps( \_\_m512 a, \_\_m512 b);  
 VMULPS \_\_m512 \_\_mm512\_mask\_mul\_ps(\_\_m512 s, \_\_mmask16 k, \_\_m512 a, \_\_m512 b);  
 VMULPS \_\_m512 \_\_mm512\_maskz\_mul\_ps(\_\_mmask16 k, \_\_m512 a, \_\_m512 b);  
 VMULPS \_\_m512 \_\_mm512\_mul\_round\_ps( \_\_m512 a, \_\_m512 b, int);  
 VMULPS \_\_m512 \_\_mm512\_mask\_mul\_round\_ps(\_\_m512 s, \_\_mmask16 k, \_\_m512 a, \_\_m512 b, int);  
 VMULPS \_\_m512 \_\_mm512\_maskz\_mul\_round\_ps(\_\_mmask16 k, \_\_m512 a, \_\_m512 b, int);  
 VMULPS \_\_m256 \_\_mm256\_mask\_mul\_ps(\_\_m256 s, \_\_mmask8 k, \_\_m256 a, \_\_m256 b);  
 VMULPS \_\_m256 \_\_mm256\_maskz\_mul\_ps(\_\_mmask8 k, \_\_m256 a, \_\_m256 b);  
 VMULPS \_\_m128 \_\_mm\_mask\_mul\_ps(\_\_m128 s, \_\_mmask8 k, \_\_m128 a, \_\_m128 b);  
 VMULPS \_\_m128 \_\_mm\_maskz\_mul\_ps(\_\_mmask8 k, \_\_m128 a, \_\_m128 b);  
 VMULPS \_\_m256 \_\_mm256\_mul\_ps( \_\_m256 a, \_\_m256 b);  
 MULPS \_\_m128 \_\_mm\_mul\_ps( \_\_m128 a, \_\_m128 b);

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 2.

EVEX-encoded instruction, see Exceptions Type E2.

## MULSD—Multiply Scalar Double-Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 59 /r MULSD xmm1,xmm2/m64	RM	V/V	SSE2	Multiply the low double-precision floating-point value in xmm2/m64 by low double-precision floating-point value in xmm1.
VEX.NDS.128.F2.0F.WIG 59 /r VMULSD xmm1,xmm2, xmm3/m64	RVM	V/V	AVX	Multiply the low double-precision floating-point value in xmm3/m64 by low double-precision floating-point value in xmm2.
EVEX.NDS.LIG.F2.0F.W1 59 /r VMULSD xmm1 {k1}{z}, xmm2, xmm3/m64 {er}	T1S	V/V	AVX512F	Multiply the low double-precision floating-point value in xmm3/m64 by low double-precision floating-point value in xmm2.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
T1S	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Multiplies the low double-precision floating-point value in the second source operand by the low double-precision floating-point value in the first source operand, and stores the double-precision floating-point result in the destination operand. The second source operand can be an XMM register or a 64-bit memory location. The first source operand and the destination operands are XMM registers.

128-bit Legacy SSE version: The first source operand and the destination operand are the same. Bits (MAX\_VL-1:64) of the corresponding destination register remain unchanged.

VEX.128 and EVEX encoded version: The quadword at bits 127:64 of the destination operand is copied from the same bits of the first source operand. Bits (MAX\_VL-1:128) of the destination register are zeroed.

EVEX encoded version: The low quadword element of the destination operand is updated according to the writemask.

Software should ensure VMULSD is encoded with VEX.L=0. Encoding VMULSD with VEX.L=1 may encounter unpredictable behavior across different processor generations.

**Operation****VMULSD (EVEX encoded version)**

```

IF (EVEX.b = 1) AND SRC2 *is a register*
  THEN
    SET_RM(EVEX.RC);
  ELSE
    SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
  THEN  DEST[63:0] ← SRC1[63:0] * SRC2[63:0]
  ELSE
    IF *merging-masking*           ; merging-masking
      THEN *DEST[63:0] remains unchanged*
    ELSE                             ; zeroing-masking
      THEN DEST[63:0] ← 0
    FI
  FI;
ENDIFOR
DEST[127:64] ← SRC1[127:64]
DEST[MAX_VL-1:128] ← 0

```

**VMULSD (VEX.128 encoded version)**

```

DEST[63:0] ← SRC1[63:0] * SRC2[63:0]
DEST[127:64] ← SRC1[127:64]
DEST[MAX_VL-1:128] ← 0

```

**MULSD (128-bit Legacy SSE version)**

```

DEST[63:0] ← DEST[63:0] * SRC[63:0]
DEST[MAX_VL-1:64] (Unmodified)

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VMULSD __m128d __mm_mask_mul_sd(__m128d s, __mmask8 k, __m128d a, __m128d b);
VMULSD __m128d __mm_maskz_mul_sd( __mmask8 k, __m128d a, __m128d b);
VMULSD __m128d __mm_mul_round_sd( __m128d a, __m128d b, int);
VMULSD __m128d __mm_mask_mul_round_sd(__m128d s, __mmask8 k, __m128d a, __m128d b, int);
VMULSD __m128d __mm_maskz_mul_round_sd( __mmask8 k, __m128d a, __m128d b, int);
MULSD __m128d __mm_mul_sd( __m128d a, __m128d b)

```

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 3.

EVEX-encoded instruction, see Exceptions Type E3.

## MULSS—Multiply Scalar Single-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 59 /r MULSS xmm1,xmm2/m32	RM	V/V	SSE	Multiply the low single-precision floating-point value in xmm2/m32 by the low single-precision floating-point value in xmm1.
VEX.NDS.128.F3.0F.WIG 59 /r VMULSS xmm1,xmm2, xmm3/m32	RVM	V/V	AVX	Multiply the low single-precision floating-point value in xmm3/m32 by the low single-precision floating-point value in xmm2.
EVEX.NDS.LIG.F3.0F.WO 59 /r VMULSS xmm1 {k1}{z}, xmm2, xmm3/m32 {er}	T1S	V/V	AVX512F	Multiply the low single-precision floating-point value in xmm3/m32 by the low single-precision floating-point value in xmm2.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
T1S	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Multiplies the low single-precision floating-point value from the second source operand by the low single-precision floating-point value in the first source operand, and stores the single-precision floating-point result in the destination operand. The second source operand can be an XMM register or a 32-bit memory location. The first source operand and the destination operands are XMM registers.

128-bit Legacy SSE version: The first source operand and the destination operand are the same. Bits (MAX\_VL-1:32) of the corresponding YMM destination register remain unchanged.

VEX.128 and EVEX encoded version: The first source operand is an xmm register encoded by VEX.vvvv. The three high-order doublewords of the destination operand are copied from the first source operand. Bits (MAX\_VL-1:128) of the destination register are zeroed.

EVEX encoded version: The low doubleword element of the destination operand is updated according to the writemask.

Software should ensure VMULSS is encoded with VEX.L=0. Encoding VMULSS with VEX.L=1 may encounter unpredictable behavior across different processor generations.

**Operation****VMULSS (EVEX encoded version)**

```

IF (EVEX.b = 1) AND SRC2 *is a register*
  THEN
    SET_RM(EVEX.RC);
  ELSE
    SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
  THEN  DEST[31:0] ← SRC1[31:0] * SRC2[31:0]
  ELSE
    IF *merging-masking*           ; merging-masking
      THEN *DEST[31:0] remains unchanged*
    ELSE                             ; zeroing-masking
      THEN DEST[31:0] ← 0
    FI
  FI;
ENDIFOR
DEST[127:32] ← SRC1[127:32]
DEST[MAX_VL-1:128] ← 0

```

**VMULSS (VEX.128 encoded version)**

```

DEST[31:0] ← SRC1[31:0] * SRC2[31:0]
DEST[127:32] ← SRC1[127:32]
DEST[MAX_VL-1:128] ← 0

```

**MULSS (128-bit Legacy SSE version)**

```

DEST[31:0] ← DEST[31:0] * SRC[31:0]
DEST[MAX_VL-1:32] (Unmodified)

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VMULSS __m128 _mm_mask_mul_ss(__m128 s, __mmask8 k, __m128 a, __m128 b);
VMULSS __m128 _mm_maskz_mul_ss( __mmask8 k, __m128 a, __m128 b);
VMULSS __m128 _mm_mul_round_ss( __m128 a, __m128 b, int);
VMULSS __m128 _mm_mask_mul_round_ss(__m128 s, __mmask8 k, __m128 a, __m128 b, int);
VMULSS __m128 _mm_maskz_mul_round_ss( __mmask8 k, __m128 a, __m128 b, int);
MULSS __m128 _mm_mul_ss(__m128 a, __m128 b)

```

**SIMD Floating-Point Exceptions**

Underflow, Overflow, Invalid, Precision, Denormal

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 3.

EVEX-encoded instruction, see Exceptions Type E3.

## ORPD—Bitwise Logical OR of Packed Double Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 56/r ORPD xmm1, xmm2/m128	RM	V/V	SSE2	Return the bitwise logical OR of packed double-precision floating-point values in xmm1 and xmm2/mem
VEX.NDS.128.66.0F 56 /r VORPD xmm1,xmm2, xmm3/m128	RVM	V/V	AVX	Return the bitwise logical OR of packed double-precision floating-point values in xmm2 and xmm3/mem
VEX.NDS.256.66.0F 56 /r VORPD ymm1, ymm2, ymm3/m256	RVM	V/V	AVX	Return the bitwise logical OR of packed double-precision floating-point values in ymm2 and ymm3/mem
EVEX.NDS.128.66.0F.W1 56 /r VORPD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	FV	V/V	AVX512VL AVX512DQ	Return the bitwise logical OR of packed double-precision floating-point values in xmm2 and xmm3/m128/m64bcst subject to writemask k1.
EVEX.NDS.256.66.0F.W1 56 /r VORPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	FV	V/V	AVX512VL AVX512DQ	Return the bitwise logical OR of packed double-precision floating-point values in ymm2 and ymm3/m256/m64bcst subject to writemask k1.
EVEX.NDS.512.66.0F.W1 56 /r VORPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	FV	V/V	AVX512DQ	Return the bitwise logical OR of packed double-precision floating-point values in zmm2 and zmm3/m512/m64bcst subject to writemask k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FV	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Performs a bitwise logical OR of the two, four or eight packed double-precision floating-point values from the first source operand and the second source operand, and stores the result in the destination operand.

EVEX encoded versions: The first source 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 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAX\_VL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAX\_VL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAX\_VL-1:128) of the corresponding register destination are unmodified.

**Operation****VORPD (EVEX encoded versions)**

(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) AND (SRC2 *is memory*)
        THEN
          DEST[i+63:i] ← SRC1[i+63:i] BITWISE OR SRC2[63:0]
        ELSE
          DEST[i+63:i] ← SRC1[i+63:i] BITWISE OR SRC2[i+63:i]
        FI;
      ELSE
        IF *merging-masking*           ; merging-masking
          THEN *DEST[i+63:i] remains unchanged*
        ELSE *zeroing-masking*         ; zeroing-masking
          DEST[i+63:i] ← 0
        FI
      FI;
    ENDFOR
  DEST[MAX_VL-1:VL] ← 0

```

**VORPD (VEX.256 encoded version)**

```

DEST[63:0] ← SRC1[63:0] BITWISE OR SRC2[63:0]
DEST[127:64] ← SRC1[127:64] BITWISE OR SRC2[127:64]
DEST[191:128] ← SRC1[191:128] BITWISE OR SRC2[191:128]
DEST[255:192] ← SRC1[255:192] BITWISE OR SRC2[255:192]
DEST[MAX_VL-1:256] ← 0

```

**VORPD (VEX.128 encoded version)**

```

DEST[63:0] ← SRC1[63:0] BITWISE OR SRC2[63:0]
DEST[127:64] ← SRC1[127:64] BITWISE OR SRC2[127:64]
DEST[MAX_VL-1:128] ← 0

```

**ORPD (128-bit Legacy SSE version)**

```

DEST[63:0] ← DEST[63:0] BITWISE OR SRC[63:0]
DEST[127:64] ← DEST[127:64] BITWISE OR SRC[127:64]
DEST[MAX_VL-1:128] (Unmodified)

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VORPD __m512d __mm512_or_pd ( __m512d a, __m512d b);
VORPD __m512d __mm512_mask_or_pd ( __m512d s, __mmask8 k, __m512d a, __m512d b);
VORPD __m512d __mm512_maskz_or_pd ( __mmask8 k, __m512d a, __m512d b);
VORPD __m256d __mm256_mask_or_pd ( __m256d s, __mmask8 k, __m256d a, __m256d b);
VORPD __m256d __mm256_maskz_or_pd ( __mmask8 k, __m256d a, __m256d b);
VORPD __m128d __mm_mask_or_pd ( __m128d s, __mmask8 k, __m128d a, __m128d b);
VORPD __m128d __mm_maskz_or_pd ( __mmask8 k, __m128d a, __m128d b);
VORPD __m256d __mm256_or_pd ( __m256d a, __m256d b);
ORPD __m128d __mm_or_pd ( __m128d a, __m128d b);

```





**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.

## ORPS—Bitwise Logical OR of Packed Single Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
OF 56 /r ORPS xmm1, xmm2/m128	RM	V/V	SSE	Return the bitwise logical OR of packed single-precision floating-point values in xmm1 and xmm2/mem
VEX.NDS.128.OF 56 /r VORPS xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Return the bitwise logical OR of packed single-precision floating-point values in xmm2 and xmm3/mem
VEX.NDS.256.OF 56 /r VORPS ymm1, ymm2, ymm3/m256	RVM	V/V	AVX	Return the bitwise logical OR of packed single-precision floating-point values in ymm2 and ymm3/mem
EVEX.NDS.128.OF.W0 56 /r VORPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	FV	V/V	AVX512VL AVX512DQ	Return the bitwise logical OR of packed single-precision floating-point values in xmm2 and xmm3/m128/m32bcst subject to writemask k1.
EVEX.NDS.256.OF.W0 56 /r VORPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	FV	V/V	AVX512VL AVX512DQ	Return the bitwise logical OR of packed single-precision floating-point values in ymm2 and ymm3/m256/m32bcst subject to writemask k1.
EVEX.NDS.512.OF.W0 56 /r VORPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	FV	V/V	AVX512DQ	Return the bitwise logical OR of packed single-precision floating-point values in zmm2 and zmm3/m512/m32bcst subject to writemask k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FV	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Performs a bitwise logical OR of the four, eight or sixteen packed single-precision floating-point values from the first source operand and the second source operand, and stores the result in the destination operand

EVEX encoded versions: The first source 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 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAX\_VL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAX\_VL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAX\_VL-1:128) of the corresponding register destination are unmodified.

**Operation****VORPS (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\*

THEN

IF (EVEX.b == 1) AND (SRC2 \*is memory\*)

THEN

DEST[i+31:i] ← SRC1[i+31:i] BITWISE OR SRC2[31:0]

ELSE

DEST[i+31:i] ← SRC1[i+31:i] BITWISE OR SRC2[i+31:i]

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+31:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VORPS (VEX.256 encoded version)**

DEST[31:0] ← SRC1[31:0] BITWISE OR SRC2[31:0]

DEST[63:32] ← SRC1[63:32] BITWISE OR SRC2[63:32]

DEST[95:64] ← SRC1[95:64] BITWISE OR SRC2[95:64]

DEST[127:96] ← SRC1[127:96] BITWISE OR SRC2[127:96]

DEST[159:128] ← SRC1[159:128] BITWISE OR SRC2[159:128]

DEST[191:160] ← SRC1[191:160] BITWISE OR SRC2[191:160]

DEST[223:192] ← SRC1[223:192] BITWISE OR SRC2[223:192]

DEST[255:224] ← SRC1[255:224] BITWISE OR SRC2[255:224].

DEST[MAX\_VL-1:256] ← 0

**VORPS (VEX.128 encoded version)**

DEST[31:0] ← SRC1[31:0] BITWISE OR SRC2[31:0]

DEST[63:32] ← SRC1[63:32] BITWISE OR SRC2[63:32]

DEST[95:64] ← SRC1[95:64] BITWISE OR SRC2[95:64]

DEST[127:96] ← SRC1[127:96] BITWISE OR SRC2[127:96]

DEST[MAX\_VL-1:128] ← 0

**ORPS (128-bit Legacy SSE version)**

DEST[31:0] ← SRC1[31:0] BITWISE OR SRC2[31:0]

DEST[63:32] ← SRC1[63:32] BITWISE OR SRC2[63:32]

DEST[95:64] ← SRC1[95:64] BITWISE OR SRC2[95:64]

DEST[127:96] ← SRC1[127:96] BITWISE OR SRC2[127:96]

DEST[MAX\_VL-1:128] (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**

VORPS \_\_m512\_mm512\_or\_ps (\_\_m512 a, \_\_m512 b);  
 VORPS \_\_m512\_mm512\_mask\_or\_ps (\_\_m512 s, \_\_mmask16 k, \_\_m512 a, \_\_m512 b);  
 VORPS \_\_m512\_mm512\_maskz\_or\_ps (\_\_mmask16 k, \_\_m512 a, \_\_m512 b);  
 VORPS \_\_m256\_mm256\_mask\_or\_ps (\_\_m256 s, \_\_mmask8 k, \_\_m256 a, \_\_m256 b);  
 VORPS \_\_m256\_mm256\_maskz\_or\_ps (\_\_mmask8 k, \_\_m256 a, \_\_m256 b);  
 VORPS \_\_m128\_mm\_mask\_or\_ps (\_\_m128 s, \_\_mmask8 k, \_\_m128 a, \_\_m128 b);  
 VORPS \_\_m128\_mm\_maskz\_or\_ps (\_\_mmask8 k, \_\_m128 a, \_\_m128 b);  
 VORPS \_\_m256\_mm256\_or\_ps (\_\_m256 a, \_\_m256 b);  
 ORPS \_\_m128\_mm\_or\_ps (\_\_m128 a, \_\_m128 b);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.

## PABSB/PABSW/PABSD/PABSQ — Packed Absolute Value

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F 38 1C /r <sup>1</sup> PABSB <i>mm1, mm2/m64</i>	RM	V/V	SSSE3	Compute the absolute value of bytes in <i>mm2/m64</i> and store UNSIGNED result in <i>mm1</i> .
66 0F 38 1C /r PABSB <i>xmm1, xmm2/m128</i>	RM	V/V	SSSE3	Compute the absolute value of bytes in <i>xmm2/m128</i> and store UNSIGNED result in <i>xmm1</i> .
0F 38 1D /r <sup>1</sup> PABSW <i>mm1, mm2/m64</i>	RM	V/V	SSSE3	Compute the absolute value of 16-bit integers in <i>mm2/m64</i> and store UNSIGNED result in <i>mm1</i> .
66 0F 38 1D /r PABSW <i>xmm1, xmm2/m128</i>	RM	V/V	SSSE3	Compute the absolute value of 16-bit integers in <i>xmm2/m128</i> and store UNSIGNED result in <i>xmm1</i> .
0F 38 1E /r <sup>1</sup> PABSD <i>mm1, mm2/m64</i>	RM	V/V	SSSE3	Compute the absolute value of 32-bit integers in <i>mm2/m64</i> and store UNSIGNED result in <i>mm1</i> .
66 0F 38 1E /r PABSD <i>xmm1, xmm2/m128</i>	RM	V/V	SSSE3	Compute the absolute value of 32-bit integers in <i>xmm2/m128</i> and store UNSIGNED result in <i>xmm1</i> .
VEX.128.66.0F38.WIG 1C /r VPABSB <i>xmm1, xmm2/m128</i>	RM	V/V	AVX	Compute the absolute value of bytes in <i>xmm2/m128</i> and store UNSIGNED result in <i>xmm1</i> .
VEX.128.66.0F38.WIG 1D /r VPABSW <i>xmm1, xmm2/m128</i>	RM	V/V	AVX	Compute the absolute value of 16-bit integers in <i>xmm2/m128</i> and store UNSIGNED result in <i>xmm1</i> .
VEX.128.66.0F38.WIG 1E /r VPABSD <i>xmm1, xmm2/m128</i>	RM	V/V	AVX	Compute the absolute value of 32-bit integers in <i>xmm2/m128</i> and store UNSIGNED result in <i>xmm1</i> .
VEX.256.66.0F38.WIG 1C /r VPABSB <i>ymm1, ymm2/m256</i>	RM	V/V	AVX2	Compute the absolute value of bytes in <i>ymm2/m256</i> and store UNSIGNED result in <i>ymm1</i> .
VEX.256.66.0F38.WIG 1D /r VPABSW <i>ymm1, ymm2/m256</i>	RM	V/V	AVX2	Compute the absolute value of 16-bit integers in <i>ymm2/m256</i> and store UNSIGNED result in <i>ymm1</i> .
VEX.256.66.0F38.WIG 1E /r VPABSD <i>ymm1, ymm2/m256</i>	RM	V/V	AVX2	Compute the absolute value of 32-bit integers in <i>ymm2/m256</i> and store UNSIGNED result in <i>ymm1</i> .
EVEX.128.66.0F38.WIG 1C /r VPABSB <i>xmm1 {k1}{z}, xmm2/m128</i>	FVM	V/V	AVX512VL AVX512BW	Compute the absolute value of bytes in <i>xmm2/m128</i> and store UNSIGNED result in <i>xmm1</i> using writemask <i>k1</i> .
EVEX.256.66.0F38.WIG 1C /r VPABSB <i>ymm1 {k1}{z}, ymm2/m256</i>	FVM	V/V	AVX512VL AVX512BW	Compute the absolute value of bytes in <i>ymm2/m256</i> and store UNSIGNED result in <i>ymm1</i> using writemask <i>k1</i> .
EVEX.512.66.0F38.WIG 1C /r VPABSB <i>zmm1 {k1}{z}, zmm2/m512</i>	FVM	V/V	AVX512BW	Compute the absolute value of bytes in <i>zmm2/m512</i> and store UNSIGNED result in <i>zmm1</i> using writemask <i>k1</i> .
EVEX.128.66.0F38.WIG 1D /r VPABSW <i>xmm1 {k1}{z}, xmm2/m128</i>	FVM	V/V	AVX512VL AVX512BW	Compute the absolute value of 16-bit integers in <i>xmm2/m128</i> and store UNSIGNED result in <i>xmm1</i> using writemask <i>k1</i> .

EVEX.256.66.0F38.WIG 1D /r VPABSW ymm1 {k1}{z}, ymm2/m256	FVM V/V	AVX512VL AVX512BW	Compute the absolute value of 16-bit integers in ymm2/m256 and store UNSIGNED result in ymm1 using writemask k1.
EVEX.512.66.0F38.WIG 1D /r VPABSW zmm1 {k1}{z}, zmm2/m512	FVM V/V	AVX512BW	Compute the absolute value of 16-bit integers in zmm2/m512 and store UNSIGNED result in zmm1 using writemask k1.
EVEX.128.66.0F38.W0 1E /r VPABSD xmm1 {k1}{z}, xmm2/m128/m32bcst	FV V/V	AVX512VL AVX512F	Compute the absolute value of 32-bit integers in xmm2/m128/m32bcst and store UNSIGNED result in xmm1 using writemask k1.
EVEX.256.66.0F38.W0 1E /r VPABSD ymm1 {k1}{z}, ymm2/m256/m32bcst	FV V/V	AVX512VL AVX512F	Compute the absolute value of 32-bit integers in ymm2/m256/m32bcst and store UNSIGNED result in ymm1 using writemask k1.
EVEX.512.66.0F38.W0 1E /r VPABSD zmm1 {k1}{z}, zmm2/m512/m32bcst	FV V/V	AVX512F	Compute the absolute value of 32-bit integers in zmm2/m512/m32bcst and store UNSIGNED result in zmm1 using writemask k1.
EVEX.128.66.0F38.W1 1F /r VPABSQ xmm1 {k1}{z}, xmm2/m128/m64bcst	FV V/V	AVX512VL AVX512F	Compute the absolute value of 64-bit integers in xmm2/m128/m64bcst and store UNSIGNED result in xmm1 using writemask k1.
EVEX.256.66.0F38.W1 1F /r VPABSQ ymm1 {k1}{z}, ymm2/m256/m64bcst	FV V/V	AVX512VL AVX512F	Compute the absolute value of 64-bit integers in ymm2/m256/m64bcst and store UNSIGNED result in ymm1 using writemask k1.
EVEX.512.66.0F38.W1 1F /r VPABSQ zmm1 {k1}{z}, zmm2/m512/m64bcst	FV V/V	AVX512F	Compute the absolute value of 64-bit integers in zmm2/m512/m64bcst and store UNSIGNED result in zmm1 using writemask k1.

**NOTES:**

1. See note in Section 2.4, “AVX and SSE Instruction Exception Specification” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
FVM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
FV	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

**Description**

PABSB/W/D computes the absolute value of each data element of the source operand (the second operand) and stores the UNSIGNED results in the destination operand (the first operand). PABSB operates on signed bytes, PABSW operates on signed 16-bit words, and PABSD operates on signed 32-bit integers.

EVEX encoded VPABSD/Q: 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 32/64-bit memory location. The destination operand is a ZMM/YMM/XMM register updated according to the writemask.

EVEX encoded VPABSB/W: 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 updated according to the writemask.

VEX.256 encoded versions: The source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAX\_VL-1:256) of the corresponding register destination are zeroed.

VEX.128 encoded versions: The source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAX\_VL-1:128) of the corresponding register destination are zeroed.

128-bit Legacy SSE version: The source operand can be an XMM register or an 128-bit memory location. The destination is an XMM register. The upper bits (VL\_MAX-1:128) of the corresponding register destination are unmodified.

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

### Operation

#### PABSB with 128 bit operands:

Unsigned DEST[7:0] ← ABS(SRC[7: 0])  
 Repeat operation for 2nd through 15th bytes  
 Unsigned DEST[127:120] ← ABS(SRC[127:120])

#### VPABSB with 128 bit operands:

Unsigned DEST[7:0] ← ABS(SRC[7: 0])  
 Repeat operation for 2nd through 15th bytes  
 Unsigned DEST[127:120] ← ABS(SRC[127:120])

#### VPABSB with 256 bit operands:

Unsigned DEST[7:0] ← ABS(SRC[7: 0])  
 Repeat operation for 2nd through 31st bytes  
 Unsigned DEST[255:248] ← ABS(SRC[255:248])

#### VPABSB (EVEX encoded versions)

(KL, VL) = (16, 128), (32, 256), (64, 512)

FOR j ← 0 TO KL-1

i ← j \* 8

IF k1[j] OR \*no writemask\*

THEN

Unsigned DEST[i+7:i] ← ABS(SRC[i+7:i])

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+7:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[i+7:i] ← 0

FI

FI;

ENDFOR;

DEST[MAX\_VL-1:VL] ← 0

#### PABSW with 128 bit operands:

Unsigned DEST[15:0] ← ABS(SRC[15:0])  
 Repeat operation for 2nd through 7th 16-bit words  
 Unsigned DEST[127:112] ← ABS(SRC[127:112])

#### VPABSW with 128 bit operands:

Unsigned DEST[15:0] ← ABS(SRC[15:0])  
 Repeat operation for 2nd through 7th 16-bit words  
 Unsigned DEST[127:112] ← ABS(SRC[127:112])

#### VPABSW with 256 bit operands:

Unsigned DEST[15:0] ← ABS(SRC[15:0])  
 Repeat operation for 2nd through 15th 16-bit words  
 Unsigned DEST[255:240] ← ABS(SRC[255:240])

**VPABSW (EVEX encoded versions)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 16
  IF k1[j] OR *no writemask*
    THEN
      Unsigned DEST[j+15:i] ← ABS(SRC[j+15:i])
    ELSE
      IF *merging-masking*           ; merging-masking
        THEN *DEST[j+15:i] remains unchanged*
      ELSE *zeroing-masking*       ; zeroing-masking
        DEST[j+15:i] ← 0
      FI
    FI
  FI;
ENDFOR;
DEST[MAX_VL-1:VL] ← 0

```

**PABSD with 128 bit operands:**

```

Unsigned DEST[31:0] ← ABS(SRC[31:0])
Repeat operation for 2nd through 3rd 32-bit double words
Unsigned DEST[127:96] ← ABS(SRC[127:96])

```

**VPABSD with 128 bit operands:**

```

Unsigned DEST[31:0] ← ABS(SRC[31:0])
Repeat operation for 2nd through 3rd 32-bit double words
Unsigned DEST[127:96] ← ABS(SRC[127:96])

```

**VPABSD with 256 bit operands:**

```

Unsigned DEST[31:0] ← ABS(SRC[31:0])
Repeat operation for 2nd through 7th 32-bit double words
Unsigned DEST[255:224] ← ABS(SRC[255:224])

```

**VPABSD (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1) AND (SRC *is memory*)
        THEN
          Unsigned DEST[j+31:i] ← ABS(SRC[31:0])
        ELSE
          Unsigned DEST[j+31:i] ← ABS(SRC[j+31:i])
        FI;
      ELSE
      IF *merging-masking*           ; merging-masking
        THEN *DEST[j+31:i] remains unchanged*
      ELSE *zeroing-masking*       ; zeroing-masking
        DEST[j+31:i] ← 0
      FI
    FI;
  FI;
ENDFOR;
DEST[MAX_VL-1:VL] ← 0

```



**VPABSQ (EVEX encoded versions)**

(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) AND (SRC \*is memory\*)

THEN

Unsigned DEST[i+63:i] ← ABS(SRC[63:0])

ELSE

Unsigned DEST[i+63:i] ← ABS(SRC[i+63:i])

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR;

DEST[MAX\_VL-1:VL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalents**

VPABSB\_\_m512i\_\_mm512\_abs\_epi8 (\_\_m512i a)

VPABSW\_\_m512i\_\_mm512\_abs\_epi16 (\_\_m512i a)

VPABSB\_\_m512i\_\_mm512\_mask\_abs\_epi8 (\_\_m512i s, \_\_mmask64 m, \_\_m512i a)

VPABSW\_\_m512i\_\_mm512\_mask\_abs\_epi16 (\_\_m512i s, \_\_mmask32 m, \_\_m512i a)

VPABSB\_\_m512i\_\_mm512\_maskz\_abs\_epi8 (\_\_mmask64 m, \_\_m512i a)

VPABSW\_\_m512i\_\_mm512\_maskz\_abs\_epi16 (\_\_mmask32 m, \_\_m512i a)

VPABSB\_\_m256i\_\_mm256\_mask\_abs\_epi8 (\_\_m256i s, \_\_mmask32 m, \_\_m256i a)

VPABSW\_\_m256i\_\_mm256\_mask\_abs\_epi16 (\_\_m256i s, \_\_mmask16 m, \_\_m256i a)

VPABSB\_\_m256i\_\_mm256\_maskz\_abs\_epi8 (\_\_mmask32 m, \_\_m256i a)

VPABSW\_\_m256i\_\_mm256\_maskz\_abs\_epi16 (\_\_mmask16 m, \_\_m256i a)

VPABSB\_\_m128i\_\_mm\_mask\_abs\_epi8 (\_\_m128i s, \_\_mmask16 m, \_\_m128i a)

VPABSW\_\_m128i\_\_mm\_mask\_abs\_epi16 (\_\_m128i s, \_\_mmask8 m, \_\_m128i a)

VPABSB\_\_m128i\_\_mm\_maskz\_abs\_epi8 (\_\_mmask16 m, \_\_m128i a)

VPABSW\_\_m128i\_\_mm\_maskz\_abs\_epi16 (\_\_mmask8 m, \_\_m128i a)

VPABSD \_\_m256i\_\_mm256\_mask\_abs\_epi32(\_\_m256i s, \_\_mmask8 k, \_\_m256i a);

VPABSD \_\_m256i\_\_mm256\_maskz\_abs\_epi32(\_\_mmask8 k, \_\_m256i a);

VPABSD \_\_m128i\_\_mm\_mask\_abs\_epi32(\_\_m128i s, \_\_mmask8 k, \_\_m128i a);

VPABSD \_\_m128i\_\_mm\_maskz\_abs\_epi32(\_\_mmask8 k, \_\_m128i a);

VPABSD \_\_m512i\_\_mm512\_abs\_epi32(\_\_m512i a);

VPABSD \_\_m512i\_\_mm512\_mask\_abs\_epi32(\_\_m512i s, \_\_mmask16 k, \_\_m512i a);

VPABSD \_\_m512i\_\_mm512\_maskz\_abs\_epi32(\_\_mmask16 k, \_\_m512i a);

VPABSQ \_\_m512i\_\_mm512\_abs\_epi64(\_\_m512i a);

VPABSQ \_\_m512i\_\_mm512\_mask\_abs\_epi64(\_\_m512i s, \_\_mmask8 k, \_\_m512i a);

VPABSQ \_\_m512i\_\_mm512\_maskz\_abs\_epi64(\_\_mmask8 k, \_\_m512i a);

VPABSQ \_\_m256i\_\_mm256\_mask\_abs\_epi64(\_\_m256i s, \_\_mmask8 k, \_\_m256i a);

VPABSQ \_\_m256i\_\_mm256\_maskz\_abs\_epi64(\_\_mmask8 k, \_\_m256i a);

VPABSQ \_\_m128i\_\_mm\_mask\_abs\_epi64(\_\_m128i s, \_\_mmask8 k, \_\_m128i a);

VPABSQ \_\_m128i\_\_mm\_maskz\_abs\_epi64(\_\_mmask8 k, \_\_m128i a);

PABSB \_\_m128i\_\_mm\_abs\_epi8 (\_\_m128i a)

VPABSB \_\_m128i\_\_mm\_abs\_epi8 (\_\_m128i a)

VPABSB \_\_m256i \_\_mm256\_abs\_epi8 (\_\_m256i a)  
PABSW \_\_m128i \_\_mm\_abs\_epi16 (\_\_m128i a)  
VPABSW \_\_m128i \_\_mm\_abs\_epi16 (\_\_m128i a)  
VPABSW \_\_m256i \_\_mm256\_abs\_epi16 (\_\_m256i a)  
PABSD \_\_m128i \_\_mm\_abs\_epi32 (\_\_m128i a)  
VPABSD \_\_m128i \_\_mm\_abs\_epi32 (\_\_m128i a)  
VPABSD \_\_m256i \_\_mm256\_abs\_epi32 (\_\_m256i a)

### SIMD Floating-Point Exceptions

None

### Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded VPABSD/Q, see Exceptions Type E4.

EVEX-encoded VPABSB/W, see Exceptions Type E4.nb.

## PACKSSWB/PACKSSDW—Pack with Signed Saturation

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F 63 /r <sup>1</sup> PACKSSWB <i>mm1, mm2/m64</i>	RM	V/V	MMX	Converts 4 packed signed word integers from <i>mm1</i> and from <i>mm2/m64</i> into 8 packed signed byte integers in <i>mm1</i> using signed saturation.
66 0F 63 /r PACKSSWB <i>xmm1, xmm2/m128</i>	RM	V/V	SSE2	Converts 8 packed signed word integers from <i>xmm1</i> and from <i>xmm2/m128</i> into 16 packed signed byte integers in <i>xmm1</i> using signed saturation.
0F 6B /r <sup>1</sup> PACKSSDW <i>mm1, mm2/m64</i>	RM	V/V	MMX	Converts 2 packed signed doubleword integers from <i>mm1</i> and from <i>mm2/m64</i> into 4 packed signed word integers in <i>mm1</i> using signed saturation.
66 0F 6B /r PACKSSDW <i>xmm1, xmm2/m128</i>	RM	V/V	SSE2	Converts 4 packed signed doubleword integers from <i>xmm1</i> and from <i>xmm2/m128</i> into 8 packed signed word integers in <i>xmm1</i> using signed saturation.
VEX.NDS.128.66.0F.WIG 63 /r VPACKSSWB <i>xmm1, xmm2, xmm3/m128</i>	RVM	V/V	AVX	Converts 8 packed signed word integers from <i>xmm2</i> and from <i>xmm3/m128</i> into 16 packed signed byte integers in <i>xmm1</i> using signed saturation.
VEX.NDS.128.66.0F.WIG 6B /r VPACKSSDW <i>xmm1, xmm2, xmm3/m128</i>	RVM	V/V	AVX	Converts 4 packed signed doubleword integers from <i>xmm2</i> and from <i>xmm3/m128</i> into 8 packed signed word integers in <i>xmm1</i> using signed saturation.
VEX.NDS.256.66.0F.WIG 63 /r VPACKSSWB <i>ymm1, ymm2, ymm3/m256</i>	RVM	V/V	AVX2	Converts 16 packed signed word integers from <i>ymm2</i> and from <i>ymm3/m256</i> into 32 packed signed byte integers in <i>ymm1</i> using signed saturation.
VEX.NDS.256.66.0F.WIG 6B /r VPACKSSDW <i>ymm1, ymm2, ymm3/m256</i>	RVM	V/V	AVX2	Converts 8 packed signed doubleword integers from <i>ymm2</i> and from <i>ymm3/m256</i> into 16 packed signed word integers in <i>ymm1</i> using signed saturation.
EVEX.NDS.128.66.0F.WIG 63 /r VPACKSSWB <i>xmm1 {k1}{z}, xmm2, xmm3/m128</i>	FVM	V/V	AVX512VL AVX512BW	Converts packed signed word integers from <i>xmm2</i> and from <i>xmm3/m128</i> into packed signed byte integers in <i>xmm1</i> using signed saturation under writemask <i>k1</i> .
EVEX.NDS.256.66.0F.WIG 63 /r VPACKSSWB <i>ymm1 {k1}{z}, ymm2, ymm3/m256</i>	FVM	V/V	AVX512VL AVX512BW	Converts packed signed word integers from <i>ymm2</i> and from <i>ymm3/m256</i> into packed signed byte integers in <i>ymm1</i> using signed saturation under writemask <i>k1</i> .
EVEX.NDS.512.66.0F.WIG 63 /r VPACKSSWB <i>zmm1 {k1}{z}, zmm2, zmm3/m512</i>	FVM	V/V	AVX512BW	Converts packed signed word integers from <i>zmm2</i> and from <i>zmm3/m512</i> into packed signed byte integers in <i>zmm1</i> using signed saturation under writemask <i>k1</i> .
EVEX.NDS.128.66.0F.W0 6B /r VPACKSSDW <i>xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst</i>	FV	V/V	AVX512VL AVX512BW	Converts packed signed doubleword integers from <i>xmm2</i> and from <i>xmm3/m128/m32bcst</i> into packed signed word integers in <i>xmm1</i> using signed saturation under writemask <i>k1</i> .

EVEX.NDS.256.66.OF.WO 6B /r VPACKSSDW ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	FV	V/V	AVX512VL AVX512BW	Converts packed signed doubleword integers from <i>ymm2</i> and from <i>ymm3/m256/m32bcst</i> into packed signed word integers in <i>ymm1</i> using signed saturation under writemask <i>k1</i> .
EVEX.NDS.512.66.OF.WO 6B /r VPACKSSDW zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	FV	V/V	AVX512BW	Converts packed signed doubleword integers from <i>zmm2</i> and from <i>zmm3/m512/m32bcst</i> into packed signed word integers in <i>zmm1</i> using signed saturation under writemask <i>k1</i> .

**NOTES:**

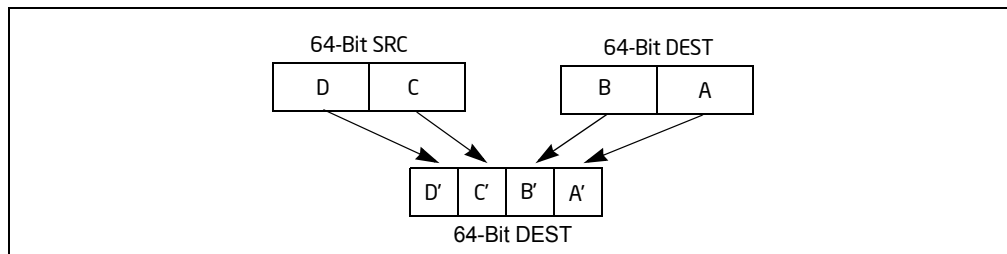
1. See note in Section 2.4, “AVX and SSE Instruction Exception Specification” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FVM	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA
FV	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

**Description**

Converts packed signed word integers into packed signed byte integers (PACKSSWB) or converts packed signed doubleword integers into packed signed word integers (PACKSSDW), using saturation to handle overflow conditions. See Figure 4-5 for an example of the packing operation.



**Figure 4-5. Operation of the PACKSSDW Instruction Using 64-bit Operands**

PACKSSWB converts packed signed word integers in the first and second source operands into packed signed byte integers using signed saturation to handle overflow conditions beyond the range of signed byte integers. If the signed doubleword value is beyond the range of an unsigned word (i.e. greater than 7FH or less than 80H), the saturated signed byte integer value of 7FH or 80H, respectively, is stored in the destination. PACKSSDW converts packed signed doubleword integers in the first and second source operands into packed signed word integers using signed saturation to handle overflow conditions beyond 7FFFH and 8000H.

EVEX encoded PACKSSWB: The first source operand is a ZMM/YMM/XMM register. The second 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, updated conditional under the writemask *k1*.

EVEX encoded PACKSSDW: The first source operand is a ZMM/YMM/XMM register. The second 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 32-bit memory location. The destination operand is a ZMM/YMM/XMM register, updated conditional under the writemask *k1*.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAX\_VL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAX\_VL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The first source operand is an XMM register. The second operand can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAX\_VL-1:128) of the corresponding ZMM destination register destination are unmodified.

## Operation

### PACKSSWB instruction (128-bit Legacy SSE version)

```
DEST[7:0] ← SaturateSignedWordToSignedByte (DEST[15:0]);
DEST[15:8] ← SaturateSignedWordToSignedByte (DEST[31:16]);
DEST[23:16] ← SaturateSignedWordToSignedByte (DEST[47:32]);
DEST[31:24] ← SaturateSignedWordToSignedByte (DEST[63:48]);
DEST[39:32] ← SaturateSignedWordToSignedByte (DEST[79:64]);
DEST[47:40] ← SaturateSignedWordToSignedByte (DEST[95:80]);
DEST[55:48] ← SaturateSignedWordToSignedByte (DEST[111:96]);
DEST[63:56] ← SaturateSignedWordToSignedByte (DEST[127:112]);
DEST[71:64] ← SaturateSignedWordToSignedByte (SRC[15:0]);
DEST[79:72] ← SaturateSignedWordToSignedByte (SRC[31:16]);
DEST[87:80] ← SaturateSignedWordToSignedByte (SRC[47:32]);
DEST[95:88] ← SaturateSignedWordToSignedByte (SRC[63:48]);
DEST[103:96] ← SaturateSignedWordToSignedByte (SRC[79:64]);
DEST[111:104] ← SaturateSignedWordToSignedByte (SRC[95:80]);
DEST[119:112] ← SaturateSignedWordToSignedByte (SRC[111:96]);
DEST[127:120] ← SaturateSignedWordToSignedByte (SRC[127:112]);
DEST[MAX_VL-1:128] (Unmodified)
```

### PACKSSDW instruction (128-bit Legacy SSE version)

```
DEST[15:0] ← SaturateSignedDwordToSignedWord (DEST[31:0]);
DEST[31:16] ← SaturateSignedDwordToSignedWord (DEST[63:32]);
DEST[47:32] ← SaturateSignedDwordToSignedWord (DEST[95:64]);
DEST[63:48] ← SaturateSignedDwordToSignedWord (DEST[127:96]);
DEST[79:64] ← SaturateSignedDwordToSignedWord (SRC[31:0]);
DEST[95:80] ← SaturateSignedDwordToSignedWord (SRC[63:32]);
DEST[111:96] ← SaturateSignedDwordToSignedWord (SRC[95:64]);
DEST[127:112] ← SaturateSignedDwordToSignedWord (SRC[127:96]);
DEST[MAX_VL-1:128] (Unmodified)
```

**VPACKSSWB instruction (VEX.128 encoded version)**

DEST[7:0] ← SaturateSignedWordToSignedByte (SRC1[15:0]);  
 DEST[15:8] ← SaturateSignedWordToSignedByte (SRC1[31:16]);  
 DEST[23:16] ← SaturateSignedWordToSignedByte (SRC1[47:32]);  
 DEST[31:24] ← SaturateSignedWordToSignedByte (SRC1[63:48]);  
 DEST[39:32] ← SaturateSignedWordToSignedByte (SRC1[79:64]);  
 DEST[47:40] ← SaturateSignedWordToSignedByte (SRC1[95:80]);  
 DEST[55:48] ← SaturateSignedWordToSignedByte (SRC1[111:96]);  
 DEST[63:56] ← SaturateSignedWordToSignedByte (SRC1[127:112]);  
 DEST[71:64] ← SaturateSignedWordToSignedByte (SRC2[15:0]);  
 DEST[79:72] ← SaturateSignedWordToSignedByte (SRC2[31:16]);  
 DEST[87:80] ← SaturateSignedWordToSignedByte (SRC2[47:32]);  
 DEST[95:88] ← SaturateSignedWordToSignedByte (SRC2[63:48]);  
 DEST[103:96] ← SaturateSignedWordToSignedByte (SRC2[79:64]);  
 DEST[111:104] ← SaturateSignedWordToSignedByte (SRC2[95:80]);  
 DEST[119:112] ← SaturateSignedWordToSignedByte (SRC2[111:96]);  
 DEST[127:120] ← SaturateSignedWordToSignedByte (SRC2[127:112]);  
 DEST[MAX\_VL-1:128] ← 0;

**VPACKSSDW instruction (VEX.128 encoded version)**

DEST[15:0] ← SaturateSignedDwordToSignedWord (SRC1[31:0]);  
 DEST[31:16] ← SaturateSignedDwordToSignedWord (SRC1[63:32]);  
 DEST[47:32] ← SaturateSignedDwordToSignedWord (SRC1[95:64]);  
 DEST[63:48] ← SaturateSignedDwordToSignedWord (SRC1[127:96]);  
 DEST[79:64] ← SaturateSignedDwordToSignedWord (SRC2[31:0]);  
 DEST[95:80] ← SaturateSignedDwordToSignedWord (SRC2[63:32]);  
 DEST[111:96] ← SaturateSignedDwordToSignedWord (SRC2[95:64]);  
 DEST[127:112] ← SaturateSignedDwordToSignedWord (SRC2[127:96]);  
 DEST[MAX\_VL-1:128] ← 0;

**VPACKSSWB instruction (VEX.256 encoded version)**

DEST[7:0] ← SaturateSignedWordToSignedByte (SRC1[15:0]);  
 DEST[15:8] ← SaturateSignedWordToSignedByte (SRC1[31:16]);  
 DEST[23:16] ← SaturateSignedWordToSignedByte (SRC1[47:32]);  
 DEST[31:24] ← SaturateSignedWordToSignedByte (SRC1[63:48]);  
 DEST[39:32] ← SaturateSignedWordToSignedByte (SRC1[79:64]);  
 DEST[47:40] ← SaturateSignedWordToSignedByte (SRC1[95:80]);  
 DEST[55:48] ← SaturateSignedWordToSignedByte (SRC1[111:96]);  
 DEST[63:56] ← SaturateSignedWordToSignedByte (SRC1[127:112]);  
 DEST[71:64] ← SaturateSignedWordToSignedByte (SRC2[15:0]);  
 DEST[79:72] ← SaturateSignedWordToSignedByte (SRC2[31:16]);  
 DEST[87:80] ← SaturateSignedWordToSignedByte (SRC2[47:32]);  
 DEST[95:88] ← SaturateSignedWordToSignedByte (SRC2[63:48]);  
 DEST[103:96] ← SaturateSignedWordToSignedByte (SRC2[79:64]);  
 DEST[111:104] ← SaturateSignedWordToSignedByte (SRC2[95:80]);  
 DEST[119:112] ← SaturateSignedWordToSignedByte (SRC2[111:96]);  
 DEST[127:120] ← SaturateSignedWordToSignedByte (SRC2[127:112]);  
 DEST[135:128] ← SaturateSignedWordToSignedByte (SRC1[143:128]);  
 DEST[143:136] ← SaturateSignedWordToSignedByte (SRC1[159:144]);  
 DEST[151:144] ← SaturateSignedWordToSignedByte (SRC1[175:160]);  
 DEST[159:152] ← SaturateSignedWordToSignedByte (SRC1[191:176]);  
 DEST[167:160] ← SaturateSignedWordToSignedByte (SRC1[207:192]);  
 DEST[175:168] ← SaturateSignedWordToSignedByte (SRC1[223:208]);  
 DEST[183:176] ← SaturateSignedWordToSignedByte (SRC1[239:224]);

DEST[191:184] ← SaturateSignedWordToSignedByte (SRC1[255:240]);  
 DEST[199:192] ← SaturateSignedWordToSignedByte (SRC2[143:128]);  
 DEST[207:200] ← SaturateSignedWordToSignedByte (SRC2[159:144]);  
 DEST[215:208] ← SaturateSignedWordToSignedByte (SRC2[175:160]);  
 DEST[223:216] ← SaturateSignedWordToSignedByte (SRC2[191:176]);  
 DEST[231:224] ← SaturateSignedWordToSignedByte (SRC2[207:192]);  
 DEST[239:232] ← SaturateSignedWordToSignedByte (SRC2[223:208]);  
 DEST[247:240] ← SaturateSignedWordToSignedByte (SRC2[239:224]);  
 DEST[255:248] ← SaturateSignedWordToSignedByte (SRC2[255:240]);  
 DEST[MAX\_VL-1:256] ← 0;

**VPACKSSDW instruction (VEX.256 encoded version)**

DEST[15:0] ← SaturateSignedDwordToSignedWord (SRC1[31:0]);  
 DEST[31:16] ← SaturateSignedDwordToSignedWord (SRC1[63:32]);  
 DEST[47:32] ← SaturateSignedDwordToSignedWord (SRC1[95:64]);  
 DEST[63:48] ← SaturateSignedDwordToSignedWord (SRC1[127:96]);  
 DEST[79:64] ← SaturateSignedDwordToSignedWord (SRC2[31:0]);  
 DEST[95:80] ← SaturateSignedDwordToSignedWord (SRC2[63:32]);  
 DEST[111:96] ← SaturateSignedDwordToSignedWord (SRC2[95:64]);  
 DEST[127:112] ← SaturateSignedDwordToSignedWord (SRC2[127:96]);  
 DEST[143:128] ← SaturateSignedDwordToSignedWord (SRC1[159:128]);  
 DEST[159:144] ← SaturateSignedDwordToSignedWord (SRC1[191:160]);  
 DEST[175:160] ← SaturateSignedDwordToSignedWord (SRC1[223:192]);  
 DEST[191:176] ← SaturateSignedDwordToSignedWord (SRC1[255:224]);  
 DEST[207:192] ← SaturateSignedDwordToSignedWord (SRC2[159:128]);  
 DEST[223:208] ← SaturateSignedDwordToSignedWord (SRC2[191:160]);  
 DEST[239:224] ← SaturateSignedDwordToSignedWord (SRC2[223:192]);  
 DEST[255:240] ← SaturateSignedDwordToSignedWord (SRC2[255:224]);  
 DEST[MAX\_VL-1:256] ← 0;

**VPACKSSWB (EVEX encoded versions)**

(KL, VL) = (16, 128), (32, 256), (64, 512)

TMP\_DEST[7:0] ← SaturateSignedWordToSignedByte (SRC1[15:0]);  
 TMP\_DEST[15:8] ← SaturateSignedWordToSignedByte (SRC1[31:16]);  
 TMP\_DEST[23:16] ← SaturateSignedWordToSignedByte (SRC1[47:32]);  
 TMP\_DEST[31:24] ← SaturateSignedWordToSignedByte (SRC1[63:48]);  
 TMP\_DEST[39:32] ← SaturateSignedWordToSignedByte (SRC1[79:64]);  
 TMP\_DEST[47:40] ← SaturateSignedWordToSignedByte (SRC1[95:80]);  
 TMP\_DEST[55:48] ← SaturateSignedWordToSignedByte (SRC1[111:96]);  
 TMP\_DEST[63:56] ← SaturateSignedWordToSignedByte (SRC1[127:112]);  
 TMP\_DEST[71:64] ← SaturateSignedWordToSignedByte (SRC2[15:0]);  
 TMP\_DEST[79:72] ← SaturateSignedWordToSignedByte (SRC2[31:16]);  
 TMP\_DEST[87:80] ← SaturateSignedWordToSignedByte (SRC2[47:32]);  
 TMP\_DEST[95:88] ← SaturateSignedWordToSignedByte (SRC2[63:48]);  
 TMP\_DEST[103:96] ← SaturateSignedWordToSignedByte (SRC2[79:64]);  
 TMP\_DEST[111:104] ← SaturateSignedWordToSignedByte (SRC2[95:80]);  
 TMP\_DEST[119:112] ← SaturateSignedWordToSignedByte (SRC2[111:96]);  
 TMP\_DEST[127:120] ← SaturateSignedWordToSignedByte (SRC2[127:112]);  
 IF VL >= 256  
   TMP\_DEST[135:128] ← SaturateSignedWordToSignedByte (SRC1[143:128]);  
   TMP\_DEST[143:136] ← SaturateSignedWordToSignedByte (SRC1[159:144]);  
   TMP\_DEST[151:144] ← SaturateSignedWordToSignedByte (SRC1[175:160]);  
   TMP\_DEST[159:152] ← SaturateSignedWordToSignedByte (SRC1[191:176]);  
   TMP\_DEST[167:160] ← SaturateSignedWordToSignedByte (SRC1[207:192]);



```

TMP_DEST[175:168] ← SaturateSignedWordToSignedByte (SRC1[223:208]);
TMP_DEST[183:176] ← SaturateSignedWordToSignedByte (SRC1[239:224]);
TMP_DEST[191:184] ← SaturateSignedWordToSignedByte (SRC1[255:240]);
TMP_DEST[199:192] ← SaturateSignedWordToSignedByte (SRC2[143:128]);
TMP_DEST[207:200] ← SaturateSignedWordToSignedByte (SRC2[159:144]);
TMP_DEST[215:208] ← SaturateSignedWordToSignedByte (SRC2[175:160]);
TMP_DEST[223:216] ← SaturateSignedWordToSignedByte (SRC2[191:176]);
TMP_DEST[231:224] ← SaturateSignedWordToSignedByte (SRC2[207:192]);
TMP_DEST[239:232] ← SaturateSignedWordToSignedByte (SRC2[223:208]);
TMP_DEST[247:240] ← SaturateSignedWordToSignedByte (SRC2[239:224]);
TMP_DEST[255:248] ← SaturateSignedWordToSignedByte (SRC2[255:240]);

```

FI;

IF VL >= 512

```

TMP_DEST[263:256] ← SaturateSignedWordToSignedByte (SRC1[271:256]);
TMP_DEST[271:264] ← SaturateSignedWordToSignedByte (SRC1[287:272]);
TMP_DEST[279:272] ← SaturateSignedWordToSignedByte (SRC1[303:288]);
TMP_DEST[287:280] ← SaturateSignedWordToSignedByte (SRC1[319:304]);
TMP_DEST[295:288] ← SaturateSignedWordToSignedByte (SRC1[335:320]);
TMP_DEST[303:296] ← SaturateSignedWordToSignedByte (SRC1[351:336]);
TMP_DEST[311:304] ← SaturateSignedWordToSignedByte (SRC1[367:352]);
TMP_DEST[319:312] ← SaturateSignedWordToSignedByte (SRC1[383:368]);

```

```

TMP_DEST[327:320] ← SaturateSignedWordToSignedByte (SRC2[271:256]);
TMP_DEST[335:328] ← SaturateSignedWordToSignedByte (SRC2[287:272]);
TMP_DEST[343:336] ← SaturateSignedWordToSignedByte (SRC2[303:288]);
TMP_DEST[351:344] ← SaturateSignedWordToSignedByte (SRC2[319:304]);
TMP_DEST[359:352] ← SaturateSignedWordToSignedByte (SRC2[335:320]);
TMP_DEST[367:360] ← SaturateSignedWordToSignedByte (SRC2[351:336]);
TMP_DEST[375:368] ← SaturateSignedWordToSignedByte (SRC2[367:352]);
TMP_DEST[383:376] ← SaturateSignedWordToSignedByte (SRC2[383:368]);

```

```

TMP_DEST[391:384] ← SaturateSignedWordToSignedByte (SRC1[399:384]);
TMP_DEST[399:392] ← SaturateSignedWordToSignedByte (SRC1[415:400]);
TMP_DEST[407:400] ← SaturateSignedWordToSignedByte (SRC1[431:416]);
TMP_DEST[415:408] ← SaturateSignedWordToSignedByte (SRC1[447:432]);
TMP_DEST[423:416] ← SaturateSignedWordToSignedByte (SRC1[463:448]);
TMP_DEST[431:424] ← SaturateSignedWordToSignedByte (SRC1[479:464]);
TMP_DEST[439:432] ← SaturateSignedWordToSignedByte (SRC1[495:480]);
TMP_DEST[447:440] ← SaturateSignedWordToSignedByte (SRC1[511:496]);

```

```

TMP_DEST[455:448] ← SaturateSignedWordToSignedByte (SRC2[399:384]);
TMP_DEST[463:456] ← SaturateSignedWordToSignedByte (SRC2[415:400]);
TMP_DEST[471:464] ← SaturateSignedWordToSignedByte (SRC2[431:416]);
TMP_DEST[479:472] ← SaturateSignedWordToSignedByte (SRC2[447:432]);
TMP_DEST[487:480] ← SaturateSignedWordToSignedByte (SRC2[463:448]);
TMP_DEST[495:488] ← SaturateSignedWordToSignedByte (SRC2[479:464]);
TMP_DEST[503:496] ← SaturateSignedWordToSignedByte (SRC2[495:480]);
TMP_DEST[511:504] ← SaturateSignedWordToSignedByte (SRC2[511:496]);

```

FI;

FOR j ← 0 TO KL-1

  i ← j \* 8

  IF k1[j] OR \*no writemask\*

    THEN

      DEST[i+7:i] ← TMP\_DEST[i+7:i]



```

ELSE
    IF *merging-masking*           ; merging-masking
        THEN *DEST[j+7:i] remains unchanged*
    ELSE *zeroing-masking*         ; zeroing-masking
        DEST[j+7:i] ← 0
    FI
FI;
ENDFOR;
DEST[MAX_VL-1:VL] ← 0

```

**VPACKSSDw (EVEX encoded versions)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j ← 0 TO ((KL/2) - 1)

i ← j \* 32

```

IF (EVEX.b == 1) AND (SRC2 *is memory*)
    THEN
        TMP_SRC2[j+31:i] ← SRC2[31:0]
    ELSE
        TMP_SRC2[j+31:i] ← SRC2[j+31:i]
    FI;
ENDFOR;

```

```

TMP_DEST[15:0] ← SaturateSignedDwordToSignedWord (SRC1[31:0]);
TMP_DEST[31:16] ← SaturateSignedDwordToSignedWord (SRC1[63:32]);
TMP_DEST[47:32] ← SaturateSignedDwordToSignedWord (SRC1[95:64]);
TMP_DEST[63:48] ← SaturateSignedDwordToSignedWord (SRC1[127:96]);
TMP_DEST[79:64] ← SaturateSignedDwordToSignedWord (TMP_SRC2[31:0]);
TMP_DEST[95:80] ← SaturateSignedDwordToSignedWord (TMP_SRC2[63:32]);
TMP_DEST[111:96] ← SaturateSignedDwordToSignedWord (TMP_SRC2[95:64]);
TMP_DEST[127:112] ← SaturateSignedDwordToSignedWord (TMP_SRC2[127:96]);
IF VL >= 256
    TMP_DEST[143:128] ← SaturateSignedDwordToSignedWord (SRC1[159:128]);
    TMP_DEST[159:144] ← SaturateSignedDwordToSignedWord (SRC1[191:160]);
    TMP_DEST[175:160] ← SaturateSignedDwordToSignedWord (SRC1[223:192]);
    TMP_DEST[191:176] ← SaturateSignedDwordToSignedWord (SRC1[255:224]);
    TMP_DEST[207:192] ← SaturateSignedDwordToSignedWord (TMP_SRC2[159:128]);
    TMP_DEST[223:208] ← SaturateSignedDwordToSignedWord (TMP_SRC2[191:160]);
    TMP_DEST[239:224] ← SaturateSignedDwordToSignedWord (TMP_SRC2[223:192]);
    TMP_DEST[255:240] ← SaturateSignedDwordToSignedWord (TMP_SRC2[255:224]);
FI;
IF VL >= 512
    TMP_DEST[271:256] ← SaturateSignedDwordToSignedWord (SRC1[287:256]);
    TMP_DEST[287:272] ← SaturateSignedDwordToSignedWord (SRC1[319:288]);
    TMP_DEST[303:288] ← SaturateSignedDwordToSignedWord (SRC1[351:320]);
    TMP_DEST[319:304] ← SaturateSignedDwordToSignedWord (SRC1[383:352]);
    TMP_DEST[335:320] ← SaturateSignedDwordToSignedWord (TMP_SRC2[287:256]);
    TMP_DEST[351:336] ← SaturateSignedDwordToSignedWord (TMP_SRC2[319:288]);
    TMP_DEST[367:352] ← SaturateSignedDwordToSignedWord (TMP_SRC2[351:320]);
    TMP_DEST[383:368] ← SaturateSignedDwordToSignedWord (TMP_SRC2[383:352]);

    TMP_DEST[399:384] ← SaturateSignedDwordToSignedWord (SRC1[415:384]);
    TMP_DEST[415:400] ← SaturateSignedDwordToSignedWord (SRC1[447:416]);
    TMP_DEST[431:416] ← SaturateSignedDwordToSignedWord (SRC1[479:448]);

```

```

TMP_DEST[447:432] ← SaturateSignedDwordToSignedWord (SRC1[511:480]);
TMP_DEST[463:448] ← SaturateSignedDwordToSignedWord (TMP_SRC2[415:384]);
TMP_DEST[479:464] ← SaturateSignedDwordToSignedWord (TMP_SRC2[447:416]);
TMP_DEST[495:480] ← SaturateSignedDwordToSignedWord (TMP_SRC2[479:448]);
TMP_DEST[511:496] ← SaturateSignedDwordToSignedWord (TMP_SRC2[511:480]);
FI;
FOR j ← 0 TO KL-1
  i ← j * 16
  IF k1[j] OR *no writemask*
    THEN DEST[i+15:i] ← TMP_DEST[i+15:i]
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+15:i] remains unchanged*
    ELSE *zeroing-masking* ; zeroing-masking
      DEST[i+15:i] ← 0
    FI
  FI;
ENDFOR;
DEST[MAX_VL-1:VL] ← 0

```

### Intel C/C++ Compiler Intrinsic Equivalents

```

VPACKSSDW __m512i __mm512_packs_epi32(__m512i m1, __m512i m2);
VPACKSSDW __m512i __mm512_mask_packs_epi32(__m512i s, __mmask32 k, __m512i m1, __m512i m2);
VPACKSSDW __m512i __mm512_maskz_packs_epi32(__mmask32 k, __m512i m1, __m512i m2);
VPACKSSDW __m256i __mm256_mask_packs_epi32(__m256i s, __mmask16 k, __m256i m1, __m256i m2);
VPACKSSDW __m256i __mm256_maskz_packs_epi32(__mmask16 k, __m256i m1, __m256i m2);
VPACKSSDW __m128i __mm_mask_packs_epi32(__m128i s, __mmask8 k, __m128i m1, __m128i m2);
VPACKSSDW __m128i __mm_maskz_packs_epi32(__mmask8 k, __m128i m1, __m128i m2);
VPACKSSWB __m512i __mm512_packs_epi16(__m512i m1, __m512i m2);
VPACKSSWB __m512i __mm512_mask_packs_epi16(__m512i s, __mmask32 k, __m512i m1, __m512i m2);
VPACKSSWB __m512i __mm512_maskz_packs_epi16(__mmask32 k, __m512i m1, __m512i m2);
VPACKSSWB __m256i __mm256_mask_packs_epi16(__m256i s, __mmask16 k, __m256i m1, __m256i m2);
VPACKSSWB __m256i __mm256_maskz_packs_epi16(__mmask16 k, __m256i m1, __m256i m2);
VPACKSSWB __m128i __mm_mask_packs_epi16(__m128i s, __mmask8 k, __m128i m1, __m128i m2);
VPACKSSWB __m128i __mm_maskz_packs_epi16(__mmask8 k, __m128i m1, __m128i m2);
PACKSSWB __m128i __mm_packs_epi16(__m128i m1, __m128i m2)
PACKSSDW __m128i __mm_packs_epi32(__m128i m1, __m128i m2)
VPACKSSWB __m256i __mm256_packs_epi16(__m256i m1, __m256i m2)
VPACKSSDW __m256i __mm256_packs_epi32(__m256i m1, __m256i m2)

```

### SIMD Floating-Point Exceptions

None

### Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded VPACKSSDW, see Exceptions Type E4NF.

EVEX-encoded VPACKSSWB, see Exceptions Type E4NF.nb.

## PACKUSDW—Pack with Unsigned Saturation

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 2B /r PACKUSDW <i>xmm1, xmm2/m128</i>	RM	V/V	SSE4_1	Convert 4 packed signed doubleword integers from <i>xmm1</i> and 4 packed signed doubleword integers from <i>xmm2/m128</i> into 8 packed unsigned word integers in <i>xmm1</i> using unsigned saturation.
VEX.NDS.128.66.0F38 2B /r VPACKUSDW <i>xmm1, xmm2, xmm3/m128</i>	RVM	V/V	AVX	Convert 4 packed signed doubleword integers from <i>xmm2</i> and 4 packed signed doubleword integers from <i>xmm3/m128</i> into 8 packed unsigned word integers in <i>xmm1</i> using unsigned saturation.
VEX.NDS.256.66.0F38 2B /r VPACKUSDW <i>ymm1, ymm2, ymm3/m256</i>	RVM	V/V	AVX2	Convert 8 packed signed doubleword integers from <i>ymm2</i> and 8 packed signed doubleword integers from <i>ymm3/m256</i> into 16 packed unsigned word integers in <i>ymm1</i> using unsigned saturation.
EVEX.NDS.128.66.0F38.W0 2B /r VPACKUSDW <i>xmm1{k1}{z}, xmm2, xmm3/m128/m32bcst</i>	FV	V/V	AVX512VL AVX512BW	Convert packed signed doubleword integers from <i>xmm2</i> and packed signed doubleword integers from <i>xmm3/m128/m32bcst</i> into packed unsigned word integers in <i>xmm1</i> using unsigned saturation under writemask <i>k1</i> .
EVEX.NDS.256.66.0F38.W0 2B /r VPACKUSDW <i>ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst</i>	FV	V/V	AVX512VL AVX512BW	Convert packed signed doubleword integers from <i>ymm2</i> and packed signed doubleword integers from <i>ymm3/m256/m32bcst</i> into packed unsigned word integers in <i>ymm1</i> using unsigned saturation under writemask <i>k1</i> .
EVEX.NDS.512.66.0F38.W0 2B /r VPACKUSDW <i>zmm1{k1}{z}, zmm2, zmm3/m512/m32bcst</i>	FV	V/V	AVX512BW	Convert packed signed doubleword integers from <i>zmm2</i> and packed signed doubleword integers from <i>zmm3/m512/m32bcst</i> into packed unsigned word integers in <i>zmm1</i> using unsigned saturation under writemask <i>k1</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FV	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Converts packed signed doubleword integers in the first and second source operands into packed unsigned word integers using unsigned saturation to handle overflow conditions. If the signed doubleword value is beyond the range of an unsigned word (that is, greater than FFFFH or less than 0000H), the saturated unsigned word integer value of FFFFH or 0000H, respectively, is stored in the destination.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second 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 32-bit memory location. The destination operand is a ZMM register, updated conditionally under the writemask *k1*.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAX\_VL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAX\_VL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The first source operand is an XMM register. The second operand can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAX\_VL-1:128) of the corresponding destination register destination are unmodified.

## Operation

### PACKUSDW (Legacy SSE instruction)

```

TMP[15:0] ← (DEST[31:0] < 0) ? 0 : DEST[15:0];
DEST[15:0] ← (DEST[31:0] > FFFFH) ? FFFFH : TMP[15:0];
TMP[31:16] ← (DEST[63:32] < 0) ? 0 : DEST[47:32];
DEST[31:16] ← (DEST[63:32] > FFFFH) ? FFFFH : TMP[31:16];
TMP[47:32] ← (DEST[95:64] < 0) ? 0 : DEST[79:64];
DEST[47:32] ← (DEST[95:64] > FFFFH) ? FFFFH : TMP[47:32];
TMP[63:48] ← (DEST[127:96] < 0) ? 0 : DEST[111:96];
DEST[63:48] ← (DEST[127:96] > FFFFH) ? FFFFH : TMP[63:48];
TMP[79:64] ← (SRC[31:0] < 0) ? 0 : SRC[15:0];
DEST[79:64] ← (SRC[31:0] > FFFFH) ? FFFFH : TMP[79:64];
TMP[95:80] ← (SRC[63:32] < 0) ? 0 : SRC[47:32];
DEST[95:80] ← (SRC[63:32] > FFFFH) ? FFFFH : TMP[95:80];
TMP[111:96] ← (SRC[95:64] < 0) ? 0 : SRC[79:64];
DEST[111:96] ← (SRC[95:64] > FFFFH) ? FFFFH : TMP[111:96];
TMP[127:112] ← (SRC[127:96] < 0) ? 0 : SRC[111:96];
DEST[127:112] ← (SRC[127:96] > FFFFH) ? FFFFH : TMP[127:112];
DEST[MAX_VL-1:128] (Unmodified)

```

### PACKUSDW (VEX.128 encoded version)

```

TMP[15:0] ← (SRC1[31:0] < 0) ? 0 : SRC1[15:0];
DEST[15:0] ← (SRC1[31:0] > FFFFH) ? FFFFH : TMP[15:0];
TMP[31:16] ← (SRC1[63:32] < 0) ? 0 : SRC1[47:32];
DEST[31:16] ← (SRC1[63:32] > FFFFH) ? FFFFH : TMP[31:16];
TMP[47:32] ← (SRC1[95:64] < 0) ? 0 : SRC1[79:64];
DEST[47:32] ← (SRC1[95:64] > FFFFH) ? FFFFH : TMP[47:32];
TMP[63:48] ← (SRC1[127:96] < 0) ? 0 : SRC1[111:96];
DEST[63:48] ← (SRC1[127:96] > FFFFH) ? FFFFH : TMP[63:48];
TMP[79:64] ← (SRC2[31:0] < 0) ? 0 : SRC2[15:0];
DEST[79:64] ← (SRC2[31:0] > FFFFH) ? FFFFH : TMP[79:64];
TMP[95:80] ← (SRC2[63:32] < 0) ? 0 : SRC2[47:32];
DEST[95:80] ← (SRC2[63:32] > FFFFH) ? FFFFH : TMP[95:80];
TMP[111:96] ← (SRC2[95:64] < 0) ? 0 : SRC2[79:64];
DEST[111:96] ← (SRC2[95:64] > FFFFH) ? FFFFH : TMP[111:96];
TMP[127:112] ← (SRC2[127:96] < 0) ? 0 : SRC2[111:96];
DEST[127:112] ← (SRC2[127:96] > FFFFH) ? FFFFH : TMP[127:112];
DEST[MAX_VL-1:128] ← 0;

```

### VPACKUSDW (VEX.256 encoded version)

```

TMP[15:0] ← (SRC1[31:0] < 0) ? 0 : SRC1[15:0];
DEST[15:0] ← (SRC1[31:0] > FFFFH) ? FFFFH : TMP[15:0];
TMP[31:16] ← (SRC1[63:32] < 0) ? 0 : SRC1[47:32];
DEST[31:16] ← (SRC1[63:32] > FFFFH) ? FFFFH : TMP[31:16];
TMP[47:32] ← (SRC1[95:64] < 0) ? 0 : SRC1[79:64];
DEST[47:32] ← (SRC1[95:64] > FFFFH) ? FFFFH : TMP[47:32];
TMP[63:48] ← (SRC1[127:96] < 0) ? 0 : SRC1[111:96];
DEST[63:48] ← (SRC1[127:96] > FFFFH) ? FFFFH : TMP[63:48];
TMP[79:64] ← (SRC2[31:0] < 0) ? 0 : SRC2[15:0];
DEST[79:64] ← (SRC2[31:0] > FFFFH) ? FFFFH : TMP[79:64];
TMP[95:80] ← (SRC2[63:32] < 0) ? 0 : SRC2[47:32];
DEST[95:80] ← (SRC2[63:32] > FFFFH) ? FFFFH : TMP[95:80];
TMP[111:96] ← (SRC2[95:64] < 0) ? 0 : SRC2[79:64];
DEST[111:96] ← (SRC2[95:64] > FFFFH) ? FFFFH : TMP[111:96];

```

```

TMP[127:112] ← (SRC2[127:96] < 0) ? 0 : SRC2[111:96];
DEST[127:112] ← (SRC2[127:96] > FFFFH) ? FFFFH : TMP[127:112];
TMP[143:128] ← (SRC1[159:128] < 0) ? 0 : SRC1[143:128];
DEST[143:128] ← (SRC1[159:128] > FFFFH) ? FFFFH : TMP[143:128];
TMP[159:144] ← (SRC1[191:160] < 0) ? 0 : SRC1[175:160];
DEST[159:144] ← (SRC1[191:160] > FFFFH) ? FFFFH : TMP[159:144];
TMP[175:160] ← (SRC1[223:192] < 0) ? 0 : SRC1[207:192];
DEST[175:160] ← (SRC1[223:192] > FFFFH) ? FFFFH : TMP[175:160];
TMP[191:176] ← (SRC1[255:224] < 0) ? 0 : SRC1[239:224];
DEST[191:176] ← (SRC1[255:224] > FFFFH) ? FFFFH : TMP[191:176];
TMP[207:192] ← (SRC2[159:128] < 0) ? 0 : SRC2[143:128];
DEST[207:192] ← (SRC2[159:128] > FFFFH) ? FFFFH : TMP[207:192];
TMP[223:208] ← (SRC2[191:160] < 0) ? 0 : SRC2[175:160];
DEST[223:208] ← (SRC2[191:160] > FFFFH) ? FFFFH : TMP[223:208];
TMP[239:224] ← (SRC2[223:192] < 0) ? 0 : SRC2[207:192];
DEST[239:224] ← (SRC2[223:192] > FFFFH) ? FFFFH : TMP[239:224];
TMP[255:240] ← (SRC2[255:224] < 0) ? 0 : SRC2[239:224];
DEST[255:240] ← (SRC2[255:224] > FFFFH) ? FFFFH : TMP[255:240];
DEST[MAX_VL-1:256] ← 0;

```

**VPACKUSDW (EVEX encoded versions)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j ← 0 TO ((KL/2) - 1)

  i ← j \* 32

  IF (EVEX.b == 1) AND (SRC2 \*is memory\*)

    THEN

      TMP\_SRC2[j+31:i] ← SRC2[31:0]

    ELSE

      TMP\_SRC2[j+31:i] ← SRC2[j+31:i]

  FI;

ENDFOR;

```

TMP[15:0] ← (SRC1[31:0] < 0) ? 0 : SRC1[15:0];
DEST[15:0] ← (SRC1[31:0] > FFFFH) ? FFFFH : TMP[15:0];
TMP[31:16] ← (SRC1[63:32] < 0) ? 0 : SRC1[47:32];
DEST[31:16] ← (SRC1[63:32] > FFFFH) ? FFFFH : TMP[31:16];
TMP[47:32] ← (SRC1[95:64] < 0) ? 0 : SRC1[79:64];
DEST[47:32] ← (SRC1[95:64] > FFFFH) ? FFFFH : TMP[47:32];
TMP[63:48] ← (SRC1[127:96] < 0) ? 0 : SRC1[111:96];
DEST[63:48] ← (SRC1[127:96] > FFFFH) ? FFFFH : TMP[63:48];
TMP[79:64] ← (TMP_SRC2[31:0] < 0) ? 0 : TMP_SRC2[15:0];
DEST[79:64] ← (TMP_SRC2[31:0] > FFFFH) ? FFFFH : TMP[79:64];
TMP[95:80] ← (TMP_SRC2[63:32] < 0) ? 0 : TMP_SRC2[47:32];
DEST[95:80] ← (TMP_SRC2[63:32] > FFFFH) ? FFFFH : TMP[95:80];
TMP[111:96] ← (TMP_SRC2[95:64] < 0) ? 0 : TMP_SRC2[79:64];
DEST[111:96] ← (TMP_SRC2[95:64] > FFFFH) ? FFFFH : TMP[111:96];
TMP[127:112] ← (TMP_SRC2[127:96] < 0) ? 0 : TMP_SRC2[111:96];
DEST[127:112] ← (TMP_SRC2[127:96] > FFFFH) ? FFFFH : TMP[127:112];
IF VL >= 256
  TMP[143:128] ← (SRC1[159:128] < 0) ? 0 : SRC1[143:128];
  DEST[143:128] ← (SRC1[159:128] > FFFFH) ? FFFFH : TMP[143:128];
  TMP[159:144] ← (SRC1[191:160] < 0) ? 0 : SRC1[175:160];
  DEST[159:144] ← (SRC1[191:160] > FFFFH) ? FFFFH : TMP[159:144];

```

```

TMP[175:160] ← (SRC1[223:192] < 0) ? 0 : SRC1[207:192];
DEST[175:160] ← (SRC1[223:192] > FFFFH) ? FFFFH : TMP[175:160];
TMP[191:176] ← (SRC1[255:224] < 0) ? 0 : SRC1[239:224];
DEST[191:176] ← (SRC1[255:224] > FFFFH) ? FFFFH : TMP[191:176];
TMP[207:192] ← (TMP_SRC2[159:128] < 0) ? 0 : TMP_SRC2[143:128];
DEST[207:192] ← (TMP_SRC2[159:128] > FFFFH) ? FFFFH : TMP[207:192];
TMP[223:208] ← (TMP_SRC2[191:160] < 0) ? 0 : TMP_SRC2[175:160];
DEST[223:208] ← (TMP_SRC2[191:160] > FFFFH) ? FFFFH : TMP[223:208];
TMP[239:224] ← (TMP_SRC2[223:192] < 0) ? 0 : TMP_SRC2[207:192];
DEST[239:224] ← (TMP_SRC2[223:192] > FFFFH) ? FFFFH : TMP[239:224];
TMP[255:240] ← (TMP_SRC2[255:224] < 0) ? 0 : TMP_SRC2[239:224];
DEST[255:240] ← (TMP_SRC2[255:224] > FFFFH) ? FFFFH : TMP[255:240];

```

FI;

IF VL >= 512

```

TMP[271:256] ← (SRC1[287:256] < 0) ? 0 : SRC1[271:256];
DEST[271:256] ← (SRC1[287:256] > FFFFH) ? FFFFH : TMP[271:256];
TMP[287:272] ← (SRC1[319:288] < 0) ? 0 : SRC1[303:288];
DEST[287:272] ← (SRC1[319:288] > FFFFH) ? FFFFH : TMP[287:272];
TMP[303:288] ← (SRC1[351:320] < 0) ? 0 : SRC1[335:320];
DEST[303:288] ← (SRC1[351:320] > FFFFH) ? FFFFH : TMP[303:288];
TMP[319:304] ← (SRC1[383:352] < 0) ? 0 : SRC1[367:352];
DEST[319:304] ← (SRC1[383:352] > FFFFH) ? FFFFH : TMP[319:304];
TMP[335:320] ← (TMP_SRC2[287:256] < 0) ? 0 : TMP_SRC2[271:256];
DEST[335:304] ← (TMP_SRC2[287:256] > FFFFH) ? FFFFH : TMP[79:64];
TMP[351:336] ← (TMP_SRC2[319:288] < 0) ? 0 : TMP_SRC2[303:288];
DEST[351:336] ← (TMP_SRC2[319:288] > FFFFH) ? FFFFH : TMP[351:336];
TMP[367:352] ← (TMP_SRC2[351:320] < 0) ? 0 : TMP_SRC2[315:320];
DEST[367:352] ← (TMP_SRC2[351:320] > FFFFH) ? FFFFH : TMP[367:352];
TMP[383:368] ← (TMP_SRC2[383:352] < 0) ? 0 : TMP_SRC2[367:352];
DEST[383:368] ← (TMP_SRC2[383:352] > FFFFH) ? FFFFH : TMP[383:368];
TMP[399:384] ← (SRC1[415:384] < 0) ? 0 : SRC1[399:384];
DEST[399:384] ← (SRC1[415:384] > FFFFH) ? FFFFH : TMP[399:384];
TMP[415:400] ← (SRC1[447:416] < 0) ? 0 : SRC1[431:416];
DEST[415:400] ← (SRC1[447:416] > FFFFH) ? FFFFH : TMP[415:400];
TMP[431:416] ← (SRC1[479:448] < 0) ? 0 : SRC1[463:448];
DEST[431:416] ← (SRC1[479:448] > FFFFH) ? FFFFH : TMP[431:416];
TMP[447:432] ← (SRC1[511:480] < 0) ? 0 : SRC1[495:480];
DEST[447:432] ← (SRC1[511:480] > FFFFH) ? FFFFH : TMP[447:432];
TMP[463:448] ← (TMP_SRC2[415:384] < 0) ? 0 : TMP_SRC2[399:384];
DEST[463:448] ← (TMP_SRC2[415:384] > FFFFH) ? FFFFH : TMP[463:448];
TMP[475:464] ← (TMP_SRC2[447:416] < 0) ? 0 : TMP_SRC2[431:416];
DEST[475:464] ← (TMP_SRC2[447:416] > FFFFH) ? FFFFH : TMP[475:464];
TMP[491:476] ← (TMP_SRC2[479:448] < 0) ? 0 : TMP_SRC2[463:448];
DEST[491:476] ← (TMP_SRC2[479:448] > FFFFH) ? FFFFH : TMP[491:476];
TMP[511:492] ← (TMP_SRC2[511:480] < 0) ? 0 : TMP_SRC2[495:480];
DEST[511:492] ← (TMP_SRC2[511:480] > FFFFH) ? FFFFH : TMP[511:492];

```

FI;

FOR j ← 0 TO KL-1

  i ← j \* 16

  IF k1[j] OR \*no writemask\*

    THEN

      DEST[i+15:i] ← TMP\_DEST[i+15:i]

    ELSE

      IF \*merging-masking\*                                   ; merging-masking

```

        THEN *DEST[j+15:i] remains unchanged*
        ELSE *zeroing-masking*           ; zeroing-masking
          DEST[j+15:i] ← 0
      FI
  FI;
ENDFOR;
DEST[MAX_VL-1:VL] ← 0

```

### Intel C/C++ Compiler Intrinsic Equivalents

```

VPACKUSDW__m512i__mm512_packus_epi32(__m512i m1, __m512i m2);
VPACKUSDW__m512i__mm512_mask_packus_epi32(__m512i s, __mmask32 k, __m512i m1, __m512i m2);
VPACKUSDW__m512i__mm512_maskz_packus_epi32(__mmask32 k, __m512i m1, __m512i m2);
VPACKUSDW__m256i__mm256_mask_packus_epi32(__m256i s, __mmask16 k, __m256i m1, __m256i m2);
VPACKUSDW__m256i__mm256_maskz_packus_epi32(__mmask16 k, __m256i m1, __m256i m2);
VPACKUSDW__m128i__mm_mask_packus_epi32(__m128i s, __mmask8 k, __m128i m1, __m128i m2);
VPACKUSDW__m128i__mm_maskz_packus_epi32(__mmask8 k, __m128i m1, __m128i m2);
PACKUSDW__m128i__mm_packus_epi32(__m128i m1, __m128i m2);
VPACKUSDW__m256i__mm256_packus_epi32(__m256i m1, __m256i m2);

```

### SIMD Floating-Point Exceptions

None

### Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4NF.



## PACKUSWB—Pack with Unsigned Saturation

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F 67 /r <sup>1</sup> PACKUSWB <i>mm</i> , <i>mm/m64</i>	RM	V/V	MMX	Converts 4 signed word integers from <i>mm</i> and 4 signed word integers from <i>mm/m64</i> into 8 unsigned byte integers in <i>mm</i> using unsigned saturation.
66 0F 67 /r PACKUSWB <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE2	Converts 8 signed word integers from <i>xmm1</i> and 8 signed word integers from <i>xmm2/m128</i> into 16 unsigned byte integers in <i>xmm1</i> using unsigned saturation.
VEX.NDS.128.66.0F.WIG 67 /r VPACKUSWB <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Converts 8 signed word integers from <i>xmm2</i> and 8 signed word integers from <i>xmm3/m128</i> into 16 unsigned byte integers in <i>xmm1</i> using unsigned saturation.
VEX.NDS.256.66.0F.WIG 67 /r VPACKUSWB <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	RVM	V/V	AVX2	Converts 16 signed word integers from <i>ymm2</i> and 16 signed word integers from <i>ymm3/m256</i> into 32 unsigned byte integers in <i>ymm1</i> using unsigned saturation.
EVEX.NDS.128.66.0F.WIG 67 /r VPACKUSWB <i>xmm1</i> { <i>k1</i> }{ <i>z</i> }, <i>xmm2</i> , <i>xmm3/m128</i>	FVM	V/V	AVX512VL AVX512BW	Converts signed word integers from <i>xmm2</i> and signed word integers from <i>xmm3/m128</i> into unsigned byte integers in <i>xmm1</i> using unsigned saturation under writemask <i>k1</i> .
EVEX.NDS.256.66.0F.WIG 67 /r VPACKUSWB <i>ymm1</i> { <i>k1</i> }{ <i>z</i> }, <i>ymm2</i> , <i>ymm3/m256</i>	FVM	V/V	AVX512VL AVX512BW	Converts signed word integers from <i>ymm2</i> and signed word integers from <i>ymm3/m256</i> into unsigned byte integers in <i>ymm1</i> using unsigned saturation under writemask <i>k1</i> .
EVEX.NDS.512.66.0F.WIG 67 /r VPACKUSWB <i>zmm1</i> { <i>k1</i> }{ <i>z</i> }, <i>zmm2</i> , <i>zmm3/m512</i>	FVM	V/V	AVX512BW	Converts signed word integers from <i>zmm2</i> and signed word integers from <i>zmm3/m512</i> into unsigned byte integers in <i>zmm1</i> using unsigned saturation under writemask <i>k1</i> .

### NOTES:

1. See note in Section 2.4, “AVX and SSE Instruction Exception Specification” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FVM	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Converts 4, 8, 16 or 32 signed word integers from the destination operand (first operand) and 4, 8, 16 or 32 signed word integers from the source operand (second operand) into 8, 16, 32 or 64 unsigned byte integers and stores the result in the destination operand. (See Figure 4-5 for an example of the packing operation.) If a signed word integer value is beyond the range of an unsigned byte integer (that is, greater than FFH or less than 00H), the saturated unsigned byte integer value of FFH or 00H, respectively, is stored in the destination.

EVEX.512 encoded version: The first source operand is a ZMM register. The second source operand is a ZMM register or a 512-bit memory location. The destination operand is a ZMM register.



VEX.256 and EVEX.256 encoded versions: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAX\_VL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 and EVEX.128 encoded versions: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAX\_VL-1:128) of the corresponding register destination are zeroed.

128-bit Legacy SSE version: The first source operand is an XMM register. The second operand can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAX\_VL-1:128) of the corresponding register destination are unmodified.

## Operation

### PACKUSWB (with 64-bit operands)

```
DEST[7:0] ← SaturateSignedWordToUnsignedByte DEST[15:0];
DEST[15:8] ← SaturateSignedWordToUnsignedByte DEST[31:16];
DEST[23:16] ← SaturateSignedWordToUnsignedByte DEST[47:32];
DEST[31:24] ← SaturateSignedWordToUnsignedByte DEST[63:48];
DEST[39:32] ← SaturateSignedWordToUnsignedByte SRC[15:0];
DEST[47:40] ← SaturateSignedWordToUnsignedByte SRC[31:16];
DEST[55:48] ← SaturateSignedWordToUnsignedByte SRC[47:32];
DEST[63:56] ← SaturateSignedWordToUnsignedByte SRC[63:48];
```

### PACKUSWB (Legacy SSE instruction)

```
DEST[7:0] ← SaturateSignedWordToUnsignedByte (DEST[15:0]);
DEST[15:8] ← SaturateSignedWordToUnsignedByte (DEST[31:16]);
DEST[23:16] ← SaturateSignedWordToUnsignedByte (DEST[47:32]);
DEST[31:24] ← SaturateSignedWordToUnsignedByte (DEST[63:48]);
DEST[39:32] ← SaturateSignedWordToUnsignedByte (DEST[79:64]);
DEST[47:40] ← SaturateSignedWordToUnsignedByte (DEST[95:80]);
DEST[55:48] ← SaturateSignedWordToUnsignedByte (DEST[111:96]);
DEST[63:56] ← SaturateSignedWordToUnsignedByte (DEST[127:112]);
DEST[71:64] ← SaturateSignedWordToUnsignedByte (SRC[15:0]);
DEST[79:72] ← SaturateSignedWordToUnsignedByte (SRC[31:16]);
DEST[87:80] ← SaturateSignedWordToUnsignedByte (SRC[47:32]);
DEST[95:88] ← SaturateSignedWordToUnsignedByte (SRC[63:48]);
DEST[103:96] ← SaturateSignedWordToUnsignedByte (SRC[79:64]);
DEST[111:104] ← SaturateSignedWordToUnsignedByte (SRC[95:80]);
DEST[119:112] ← SaturateSignedWordToUnsignedByte (SRC[111:96]);
DEST[127:120] ← SaturateSignedWordToUnsignedByte (SRC[127:112]);
```

### PACKUSWB (VEX.128 encoded version)

```
DEST[7:0] ← SaturateSignedWordToUnsignedByte (SRC1[15:0]);
DEST[15:8] ← SaturateSignedWordToUnsignedByte (SRC1[31:16]);
DEST[23:16] ← SaturateSignedWordToUnsignedByte (SRC1[47:32]);
DEST[31:24] ← SaturateSignedWordToUnsignedByte (SRC1[63:48]);
DEST[39:32] ← SaturateSignedWordToUnsignedByte (SRC1[79:64]);
DEST[47:40] ← SaturateSignedWordToUnsignedByte (SRC1[95:80]);
DEST[55:48] ← SaturateSignedWordToUnsignedByte (SRC1[111:96]);
DEST[63:56] ← SaturateSignedWordToUnsignedByte (SRC1[127:112]);
DEST[71:64] ← SaturateSignedWordToUnsignedByte (SRC2[15:0]);
DEST[79:72] ← SaturateSignedWordToUnsignedByte (SRC2[31:16]);
DEST[87:80] ← SaturateSignedWordToUnsignedByte (SRC2[47:32]);
DEST[95:88] ← SaturateSignedWordToUnsignedByte (SRC2[63:48]);
DEST[103:96] ← SaturateSignedWordToUnsignedByte (SRC2[79:64]);
DEST[111:104] ← SaturateSignedWordToUnsignedByte (SRC2[95:80]);
```

DEST[119:112] ← SaturateSignedWordToUnsignedByte (SRC2[111:96]);  
 DEST[127:120] ← SaturateSignedWordToUnsignedByte (SRC2[127:112]);  
 DEST[VLMAX-1:128] ← 0;

#### VPACKUSWB (VEX.256 encoded version)

DEST[7:0] ← SaturateSignedWordToUnsignedByte (SRC1[15:0]);  
 DEST[15:8] ← SaturateSignedWordToUnsignedByte (SRC1[31:16]);  
 DEST[23:16] ← SaturateSignedWordToUnsignedByte (SRC1[47:32]);  
 DEST[31:24] ← SaturateSignedWordToUnsignedByte (SRC1[63:48]);  
 DEST[39:32] ← SaturateSignedWordToUnsignedByte (SRC1[79:64]);  
 DEST[47:40] ← SaturateSignedWordToUnsignedByte (SRC1[95:80]);  
 DEST[55:48] ← SaturateSignedWordToUnsignedByte (SRC1[111:96]);  
 DEST[63:56] ← SaturateSignedWordToUnsignedByte (SRC1[127:112]);  
 DEST[71:64] ← SaturateSignedWordToUnsignedByte (SRC2[15:0]);  
 DEST[79:72] ← SaturateSignedWordToUnsignedByte (SRC2[31:16]);  
 DEST[87:80] ← SaturateSignedWordToUnsignedByte (SRC2[47:32]);  
 DEST[95:88] ← SaturateSignedWordToUnsignedByte (SRC2[63:48]);  
 DEST[103:96] ← SaturateSignedWordToUnsignedByte (SRC2[79:64]);  
 DEST[111:104] ← SaturateSignedWordToUnsignedByte (SRC2[95:80]);  
 DEST[119:112] ← SaturateSignedWordToUnsignedByte (SRC2[111:96]);  
 DEST[127:120] ← SaturateSignedWordToUnsignedByte (SRC2[127:112]);  
 DEST[135:128] ← SaturateSignedWordToUnsignedByte (SRC1[143:128]);  
 DEST[143:136] ← SaturateSignedWordToUnsignedByte (SRC1[159:144]);  
 DEST[151:144] ← SaturateSignedWordToUnsignedByte (SRC1[175:160]);  
 DEST[159:152] ← SaturateSignedWordToUnsignedByte (SRC1[191:176]);  
 DEST[167:160] ← SaturateSignedWordToUnsignedByte (SRC1[207:192]);  
 DEST[175:168] ← SaturateSignedWordToUnsignedByte (SRC1[223:208]);  
 DEST[183:176] ← SaturateSignedWordToUnsignedByte (SRC1[239:224]);  
 DEST[191:184] ← SaturateSignedWordToUnsignedByte (SRC1[255:240]);  
 DEST[199:192] ← SaturateSignedWordToUnsignedByte (SRC2[143:128]);  
 DEST[207:200] ← SaturateSignedWordToUnsignedByte (SRC2[159:144]);  
 DEST[215:208] ← SaturateSignedWordToUnsignedByte (SRC2[175:160]);  
 DEST[223:216] ← SaturateSignedWordToUnsignedByte (SRC2[191:176]);  
 DEST[231:224] ← SaturateSignedWordToUnsignedByte (SRC2[207:192]);  
 DEST[239:232] ← SaturateSignedWordToUnsignedByte (SRC2[223:208]);  
 DEST[247:240] ← SaturateSignedWordToUnsignedByte (SRC2[239:224]);  
 DEST[255:248] ← SaturateSignedWordToUnsignedByte (SRC2[255:240]);

#### VPACKUSWB (EVEX encoded versions)

(KL, VL) = (16, 128), (32, 256), (64, 512)

TMP\_DEST[7:0] ← SaturateSignedWordToUnsignedByte (SRC1[15:0]);  
 TMP\_DEST[15:8] ← SaturateSignedWordToUnsignedByte (SRC1[31:16]);  
 TMP\_DEST[23:16] ← SaturateSignedWordToUnsignedByte (SRC1[47:32]);  
 TMP\_DEST[31:24] ← SaturateSignedWordToUnsignedByte (SRC1[63:48]);  
 TMP\_DEST[39:32] ← SaturateSignedWordToUnsignedByte (SRC1[79:64]);  
 TMP\_DEST[47:40] ← SaturateSignedWordToUnsignedByte (SRC1[95:80]);  
 TMP\_DEST[55:48] ← SaturateSignedWordToUnsignedByte (SRC1[111:96]);  
 TMP\_DEST[63:56] ← SaturateSignedWordToUnsignedByte (SRC1[127:112]);  
 TMP\_DEST[71:64] ← SaturateSignedWordToUnsignedByte (SRC2[15:0]);  
 TMP\_DEST[79:72] ← SaturateSignedWordToUnsignedByte (SRC2[31:16]);  
 TMP\_DEST[87:80] ← SaturateSignedWordToUnsignedByte (SRC2[47:32]);  
 TMP\_DEST[95:88] ← SaturateSignedWordToUnsignedByte (SRC2[63:48]);  
 TMP\_DEST[103:96] ← SaturateSignedWordToUnsignedByte (SRC2[79:64]);  
 TMP\_DEST[111:104] ← SaturateSignedWordToUnsignedByte (SRC2[95:80]);

```

TMP_DEST[119:112] ← SaturateSignedWordToUnsignedByte (SRC2[111:96]);
TMP_DEST[127:120] ← SaturateSignedWordToUnsignedByte (SRC2[127:112]);
IF VL >= 256
    TMP_DEST[135:128] ← SaturateSignedWordToUnsignedByte (SRC1[143:128]);
    TMP_DEST[143:136] ← SaturateSignedWordToUnsignedByte (SRC1[159:144]);
    TMP_DEST[151:144] ← SaturateSignedWordToUnsignedByte (SRC1[175:160]);
    TMP_DEST[159:152] ← SaturateSignedWordToUnsignedByte (SRC1[191:176]);
    TMP_DEST[167:160] ← SaturateSignedWordToUnsignedByte (SRC1[207:192]);
    TMP_DEST[175:168] ← SaturateSignedWordToUnsignedByte (SRC1[223:208]);
    TMP_DEST[183:176] ← SaturateSignedWordToUnsignedByte (SRC1[239:224]);
    TMP_DEST[191:184] ← SaturateSignedWordToUnsignedByte (SRC1[255:240]);
    TMP_DEST[199:192] ← SaturateSignedWordToUnsignedByte (SRC2[143:128]);
    TMP_DEST[207:200] ← SaturateSignedWordToUnsignedByte (SRC2[159:144]);
    TMP_DEST[215:208] ← SaturateSignedWordToUnsignedByte (SRC2[175:160]);
    TMP_DEST[223:216] ← SaturateSignedWordToUnsignedByte (SRC2[191:176]);
    TMP_DEST[231:224] ← SaturateSignedWordToUnsignedByte (SRC2[207:192]);
    TMP_DEST[239:232] ← SaturateSignedWordToUnsignedByte (SRC2[223:208]);
    TMP_DEST[247:240] ← SaturateSignedWordToUnsignedByte (SRC2[239:224]);
    TMP_DEST[255:248] ← SaturateSignedWordToUnsignedByte (SRC2[255:240]);
FI;
IF VL >= 512
    TMP_DEST[263:256] ← SaturateSignedWordToUnsignedByte (SRC1[271:256]);
    TMP_DEST[271:264] ← SaturateSignedWordToUnsignedByte (SRC1[287:272]);
    TMP_DEST[279:272] ← SaturateSignedWordToUnsignedByte (SRC1[303:288]);
    TMP_DEST[287:280] ← SaturateSignedWordToUnsignedByte (SRC1[319:304]);
    TMP_DEST[295:288] ← SaturateSignedWordToUnsignedByte (SRC1[335:320]);
    TMP_DEST[303:296] ← SaturateSignedWordToUnsignedByte (SRC1[351:336]);
    TMP_DEST[311:304] ← SaturateSignedWordToUnsignedByte (SRC1[367:352]);
    TMP_DEST[319:312] ← SaturateSignedWordToUnsignedByte (SRC1[383:368]);

    TMP_DEST[327:320] ← SaturateSignedWordToUnsignedByte (SRC2[271:256]);
    TMP_DEST[335:328] ← SaturateSignedWordToUnsignedByte (SRC2[287:272]);
    TMP_DEST[343:336] ← SaturateSignedWordToUnsignedByte (SRC2[303:288]);
    TMP_DEST[351:344] ← SaturateSignedWordToUnsignedByte (SRC2[319:304]);
    TMP_DEST[359:352] ← SaturateSignedWordToUnsignedByte (SRC2[335:320]);
    TMP_DEST[367:360] ← SaturateSignedWordToUnsignedByte (SRC2[351:336]);
    TMP_DEST[375:368] ← SaturateSignedWordToUnsignedByte (SRC2[367:352]);
    TMP_DEST[383:376] ← SaturateSignedWordToUnsignedByte (SRC2[383:368]);

    TMP_DEST[391:384] ← SaturateSignedWordToUnsignedByte (SRC1[399:384]);
    TMP_DEST[399:392] ← SaturateSignedWordToUnsignedByte (SRC1[415:400]);
    TMP_DEST[407:400] ← SaturateSignedWordToUnsignedByte (SRC1[431:416]);
    TMP_DEST[415:408] ← SaturateSignedWordToUnsignedByte (SRC1[447:432]);
    TMP_DEST[423:416] ← SaturateSignedWordToUnsignedByte (SRC1[463:448]);
    TMP_DEST[431:424] ← SaturateSignedWordToUnsignedByte (SRC1[479:464]);
    TMP_DEST[439:432] ← SaturateSignedWordToUnsignedByte (SRC1[495:480]);
    TMP_DEST[447:440] ← SaturateSignedWordToUnsignedByte (SRC1[511:496]);

    TMP_DEST[455:448] ← SaturateSignedWordToUnsignedByte (SRC2[399:384]);
    TMP_DEST[463:456] ← SaturateSignedWordToUnsignedByte (SRC2[415:400]);
    TMP_DEST[471:464] ← SaturateSignedWordToUnsignedByte (SRC2[431:416]);
    TMP_DEST[479:472] ← SaturateSignedWordToUnsignedByte (SRC2[447:432]);
    TMP_DEST[487:480] ← SaturateSignedWordToUnsignedByte (SRC2[463:448]);
    TMP_DEST[495:488] ← SaturateSignedWordToUnsignedByte (SRC2[479:464]);

```

```

TMP_DEST[503:496] ← SaturateSignedWordToUnsignedByte (SRC2[495:480]);
TMP_DEST[511:504] ← SaturateSignedWordToUnsignedByte (SRC2[511:496]);
FI;
FOR j ← 0 TO KL-1
  i ← j * 8
  IF k1[j] OR *no writemask*
    THEN
      DEST[i+7:i] ← TMP_DEST[i+7:i]
    ELSE
      IF *merging-masking*           ; merging-masking
        THEN *DEST[i+7:i] remains unchanged*
      ELSE *zeroing-masking*         ; zeroing-masking
        DEST[i+7:i] ← 0
    FI
  FI;
ENDFOR;
DEST[MAX_VL-1:VL] ← 0

```

### Intel C/C++ Compiler Intrinsic Equivalents

```

VPACKUSWB__m512i__mm512_packus_epi16(__m512i m1, __m512i m2);
VPACKUSWB__m512i__mm512_mask_packus_epi16(__m512i s, __mmask64 k, __m512i m1, __m512i m2);
VPACKUSWB__m512i__mm512_maskz_packus_epi16(__mmask64 k, __m512i m1, __m512i m2);
VPACKUSWB__m256i__mm256_mask_packus_epi16(__m256i s, __mmask32 k, __m256i m1, __m256i m2);
VPACKUSWB__m256i__mm256_maskz_packus_epi16(__mmask32 k, __m256i m1, __m256i m2);
VPACKUSWB__m128i__mm_mask_packus_epi16(__m128i s, __mmask16 k, __m128i m1, __m128i m2);
VPACKUSWB__m128i__mm_maskz_packus_epi16(__mmask16 k, __m128i m1, __m128i m2);

PACKUSWB:   __m64 __mm_packs_pu16(__m64 m1, __m64 m2)
(V)PACKUSWB: __m128i __mm_packus_epi16(__m128i m1, __m128i m2)
VPACKUSWB:  __m256i __mm256_packus_epi16(__m256i m1, __m256i m2);

```

### Flags Affected

None

### SIMD Floating-Point Exceptions

None

### Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4NF.nb.

## PADDB/PADDW/PADDD/PADDQ—Add Packed Integers

Opcode/ Instruction	Op / En	64/32 bitMode Support	CPUID Feature Flag	Description
OF FC /r <sup>1</sup> PADDB mm, mm/m64	RM	V/V	MMX	Add packed byte integers from mm/m64 and mm.
OF FD /r <sup>1</sup> PADDW mm, mm/m64	RM	V/V	MMX	Add packed word integers from mm/m64 and mm.
66 OF FC /r PADDB xmm1, xmm2/m128	RM	V/V	SSE2	Add packed byte integers from xmm2/m128 and xmm1.
66 OF FD /r PADDW xmm1, xmm2/m128	RM	V/V	SSE2	Add packed word integers from xmm2/m128 and xmm1.
66 OF FE /r PADDD xmm1, xmm2/m128	RM	V/V	SSE2	Add packed doubleword integers from xmm2/m128 and xmm1.
66 OF D4 /r PADDQ xmm1, xmm2/m128	RM	V/V	SSE2	Add packed quadword integers from xmm2/m128 and xmm1.
VEX.NDS.128.66.OF.WIG FC /r VPADDB xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Add packed byte integers from xmm2, and xmm3/m128 and store in xmm1.
VEX.NDS.128.66.OF.WIG FD /r VPADDW xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Add packed word integers from xmm2, xmm3/m128 and store in xmm1.
VEX.NDS.128.66.OF.WIG FE /r VPADDD xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Add packed doubleword integers from xmm2, xmm3/m128 and store in xmm1.
VEX.NDS.128.66.OF.WIG D4 /r VPADDQ xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Add packed quadword integers from xmm2, xmm3/m128 and store in xmm1.
VEX.NDS.256.66.OF.WIG FC /r VPADDB ymm1, ymm2, ymm3/m256	RVM	V/V	AVX2	Add packed byte integers from ymm2, and ymm3/m256 and store in ymm1.
VEX.NDS.256.66.OF.WIG FD /r VPADDW ymm1, ymm2, ymm3/m256	RVM	V/V	AVX2	Add packed word integers from ymm2, ymm3/m256 and store in ymm1.
VEX.NDS.256.66.OF.WIG FE /r VPADDD ymm1, ymm2, ymm3/m256	RVM	V/V	AVX2	Add packed doubleword integers from ymm2, ymm3/m256 and store in ymm1.
VEX.NDS.256.66.OF.WIG D4 /r VPADDQ ymm1, ymm2, ymm3/m256	RVM	V/V	AVX2	Add packed quadword integers from ymm2, ymm3/m256 and store in ymm1.
EVEX.NDS.128.66.OF.WIG FC /r VPADDB xmm1 {k1}{z}, xmm2, xmm3/m128	FVM	V/V	AVX512VL AVX512BW	Add packed byte integers from xmm2, and xmm3/m128 and store in xmm1 using writemask k1.
EVEX.NDS.128.66.OF.WIG FD /r VPADDW xmm1 {k1}{z}, xmm2, xmm3/m128	FVM	V/V	AVX512VL AVX512BW	Add packed word integers from xmm2, and xmm3/m128 and store in xmm1 using writemask k1.
EVEX.NDS.128.66.OF.W0 FE /r VPADDD xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	FV	V/V	AVX512VL AVX512F	Add packed doubleword integers from xmm2, and xmm3/m128/m32bcst and store in xmm1 using writemask k1.
EVEX.NDS.128.66.OF.W1 D4 /r VPADDQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	FV	V/V	AVX512VL AVX512F	Add packed quadword integers from xmm2, and xmm3/m128/m64bcst and store in xmm1 using writemask k1.
EVEX.NDS.256.66.OF.WIG FC /r VPADDB ymm1 {k1}{z}, ymm2, ymm3/m256	FVM	V/V	AVX512VL AVX512BW	Add packed byte integers from ymm2, and ymm3/m256 and store in ymm1 using writemask k1.
EVEX.NDS.256.66.OF.WIG FD /r VPADDW ymm1 {k1}{z}, ymm2, ymm3/m256	FVM	V/V	AVX512VL AVX512BW	Add packed word integers from ymm2, and ymm3/m256 and store in ymm1 using writemask k1.

Opcode/ Instruction	Op / En	64/32 bitMode Support	CPUID Feature Flag	Description
EVEX.NDS.256.66.0F.W0 FE /r VPADDD <i>ymm1</i> { <i>k1</i> }{ <i>z</i> }, <i>ymm2</i> , <i>ymm3/m256/m32bcst</i>	FV	V/V	AVX512VL AVX512F	Add packed doubleword integers from <i>ymm2</i> , <i>ymm3/m256/m32bcst</i> and store in <i>ymm1</i> using writemask <i>k1</i> .
EVEX.NDS.256.66.0F.W1 D4 /r VPADDQ <i>ymm1</i> { <i>k1</i> }{ <i>z</i> }, <i>ymm2</i> , <i>ymm3/m256/m64bcst</i>	FV	V/V	AVX512VL AVX512F	Add packed quadword integers from <i>ymm2</i> , <i>ymm3/m256/m64bcst</i> and store in <i>ymm1</i> using writemask <i>k1</i> .
EVEX.NDS.512.66.0F.WIG FC /r VPADDB <i>zmm1</i> { <i>k1</i> }{ <i>z</i> }, <i>zmm2</i> , <i>zmm3/m512</i>	FVM	V/V	AVX512BW	Add packed byte integers from <i>zmm2</i> , and <i>zmm3/m512</i> and store in <i>zmm1</i> using writemask <i>k1</i> .
EVEX.NDS.512.66.0F.WIG FD /r VPADDW <i>zmm1</i> { <i>k1</i> }{ <i>z</i> }, <i>zmm2</i> , <i>zmm3/m512</i>	FVM	V/V	AVX512BW	Add packed word integers from <i>zmm2</i> , and <i>zmm3/m512</i> and store in <i>zmm1</i> using writemask <i>k1</i> .
EVEX.NDS.512.66.0F.W0 FE /r VPADDD <i>zmm1</i> { <i>k1</i> }{ <i>z</i> }, <i>zmm2</i> , <i>zmm3/m512/m32bcst</i>	FV	V/V	AVX512F	Add packed doubleword integers from <i>zmm2</i> , <i>zmm3/m512/m32bcst</i> and store in <i>zmm1</i> using writemask <i>k1</i> .
EVEX.NDS.512.66.0F.W1 D4 /r VPADDQ <i>zmm1</i> { <i>k1</i> }{ <i>z</i> }, <i>zmm2</i> , <i>zmm3/m512/m64bcst</i>	FV	V/V	AVX512F	Add packed quadword integers from <i>zmm2</i> , <i>zmm3/m512/m64bcst</i> and store in <i>zmm1</i> using writemask <i>k1</i> .

**NOTES:**

1. See note in Section 2.4, "AVX and SSE Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FVM	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA
FV	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

**Description**

Performs a SIMD add of the packed integers from the source operand (second operand) and the destination operand (first operand), and stores the packed integer results in the destination operand. See Figure 9-4 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for an illustration of a SIMD operation. Overflow is handled with wraparound, as described in the following paragraphs.

The PADDB and VPADDB instructions add packed byte integers from the first source operand and second source operand and store the packed integer results in the destination operand. When an individual result is too large to be represented in 8 bits (overflow), the result is wrapped around and the low 8 bits are written to the destination operand (that is, the carry is ignored).

The PADDW and VPADDW instructions add packed word integers from the first source operand and second source operand and store the packed integer results in the destination operand. When an individual result is too large to be represented in 16 bits (overflow), the result is wrapped around and the low 16 bits are written to the destination operand (that is, the carry is ignored).

The PADDD and VPADDD instructions add packed doubleword integers from the first source operand and second source operand and store the packed integer results in the destination operand. When an individual result is too large to be represented in 32 bits (overflow), the result is wrapped around and the low 32 bits are written to the destination operand (that is, the carry is ignored).



The PADDQ and VPADDQ instructions add packed quadword integers from the first source operand and second source operand and store the packed integer results in the destination operand. When a quadword result is too large to be represented in 64 bits (overflow), the result is wrapped around and the low 64 bits are written to the destination operand (that is, the carry is ignored).

Note that the (V)PADDB, (V)PADDW, (V)PADDD and (V)PADDQ instructions can operate on either unsigned or signed (two's complement notation) packed integers; however, it does not set bits in the EFLAGS register to indicate overflow and/or a carry. To prevent undetected overflow conditions, software must control the ranges of values operated on.

**EVEX encoded VPADDD/Q:** The first source operand is a ZMM/YMM/XMM register. The second 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 32/64-bit memory location. The destination operand is a ZMM/YMM/XMM register updated according to the writemask.

**EVEX encoded VPADDB/W:** The first source operand is a ZMM/YMM/XMM register. The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination operand is a ZMM/YMM/XMM register updated according to the writemask.

**VEX.256 encoded version:** The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAX\_VL-1:256) of the destination are cleared.

**VEX.128 encoded version:** The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAX\_VL-1:128) of the corresponding ZMM register destination are zeroed.

**128-bit Legacy SSE version:** The first source operand is an XMM register. The second operand can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAX\_VL-1:128) of the corresponding ZMM register destination are unmodified.

## Operation

### PADDB (with 64-bit operands)

```
DEST[7:0] ← DEST[7:0] + SRC[7:0];
(* Repeat add operation for 2nd through 7th byte *)
DEST[63:56] ← DEST[63:56] + SRC[63:56];
```

### PADDW (with 64-bit operands)

```
DEST[15:0] ← DEST[15:0] + SRC[15:0];
(* Repeat add operation for 2nd and 3th word *)
DEST[63:48] ← DEST[63:48] + SRC[63:48];
```

### PADDD (with 64-bit operands)

```
DEST[31:0] ← DEST[31:0] + SRC[31:0];
DEST[63:32] ← DEST[63:32] + SRC[63:32];
```

### PADDQ (with 64-Bit operands)

```
DEST[63:0] ← DEST[63:0] + SRC[63:0];
```

### PADDB (Legacy SSE instruction)

```
DEST[7:0] ← DEST[7:0] + SRC[7:0];
(* Repeat add operation for 2nd through 15th byte *)
DEST[127:120] ← DEST[127:120] + SRC[127:120];
DEST[MAX_VL-1:128] (Unmodified)
```

### PADDW (Legacy SSE instruction)

```
DEST[15:0] ← DEST[15:0] + SRC[15:0];
(* Repeat add operation for 2nd through 7th word *)
DEST[127:112] ← DEST[127:112] + SRC[127:112];
DEST[MAX_VL-1:128] (Unmodified)
```

**PADD (Legacy SSE instruction)**

DEST[31:0] ← DEST[31:0] + SRC[31:0];  
 (\* Repeat add operation for 2nd and 3th doubleword \*)  
 DEST[127:96] ← DEST[127:96] + SRC[127:96];  
 DEST[MAX\_VL-1:128] (Unmodified)

**PADDQ (Legacy SSE instruction)**

DEST[63:0] ← DEST[63:0] + SRC[63:0];  
 DEST[127:64] ← DEST[127:64] + SRC[127:64];  
 DEST[MAX\_VL-1:128] (Unmodified)

**VPADDB (VEX.128 encoded instruction)**

DEST[7:0] ← SRC1[7:0] + SRC2[7:0];  
 (\* Repeat add operation for 2nd through 15th byte \*)  
 DEST[127:120] ← SRC1[127:120] + SRC2[127:120];  
 DEST[MAX\_VL-1:128] ← 0;

**VPADDW (VEX.128 encoded instruction)**

DEST[15:0] ← SRC1[15:0] + SRC2[15:0];  
 (\* Repeat add operation for 2nd through 7th word \*)  
 DEST[127:112] ← SRC1[127:112] + SRC2[127:112];  
 DEST[MAX\_VL-1:128] ← 0;

**VPADDD (VEX.128 encoded instruction)**

DEST[31:0] ← SRC1[31:0] + SRC2[31:0];  
 (\* Repeat add operation for 2nd and 3th doubleword \*)  
 DEST[127:96] ← SRC1[127:96] + SRC2[127:96];  
 DEST[MAX\_VL-1:128] ← 0;

**VPADDQ (VEX.128 encoded instruction)**

DEST[63:0] ← SRC1[63:0] + SRC2[63:0];  
 DEST[127:64] ← SRC1[127:64] + SRC2[127:64];  
 DEST[MAX\_VL-1:128] ← 0;

**VPADDB (VEX.256 encoded instruction)**

DEST[7:0] ← SRC1[7:0] + SRC2[7:0];  
 (\* Repeat add operation for 2nd through 31th byte \*)  
 DEST[255:248] ← SRC1[255:248] + SRC2[255:248];

**VPADDW (VEX.256 encoded instruction)**

DEST[15:0] ← SRC1[15:0] + SRC2[15:0];  
 (\* Repeat add operation for 2nd through 15th word \*)  
 DEST[255:240] ← SRC1[255:240] + SRC2[255:240];

**VPADDD (VEX.256 encoded instruction)**

DEST[31:0] ← SRC1[31:0] + SRC2[31:0];  
 (\* Repeat add operation for 2nd and 7th doubleword \*)  
 DEST[255:224] ← SRC1[255:224] + SRC2[255:224];

**VPADDQ (VEX.256 encoded instruction)**

DEST[63:0] ← SRC1[63:0] + SRC2[63:0];  
 DEST[127:64] ← SRC1[127:64] + SRC2[127:64];  
 DEST[191:128] ← SRC1[191:128] + SRC2[191:128];  
 DEST[255:192] ← SRC1[255:192] + SRC2[255:192];



**VPADDB (EVEX encoded versions)**

(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] ← SRC1[i+7:i] + SRC2[i+7:i]
  ELSE
    IF *merging-masking*           ; merging-masking
      THEN *DEST[i+7:i] remains unchanged*
    ELSE *zeroing-masking*         ; zeroing-masking
      DEST[i+7:i] = 0
    FI
  FI;
ENDFOR;
DEST[MAX_VL-1:VL] ← 0

```

**VPADDW (EVEX encoded versions)**

(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] ← SRC1[i+15:i] + SRC2[i+15:i]
  ELSE
    IF *merging-masking*           ; merging-masking
      THEN *DEST[i+15:i] remains unchanged*
    ELSE *zeroing-masking*         ; zeroing-masking
      DEST[i+15:i] = 0
    FI
  FI;
ENDFOR;
DEST[MAX_VL-1:VL] ← 0

```

**VPADDD (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1) AND (SRC2 *is memory*)
        THEN DEST[i+31:i] ← SRC1[i+31:i] + SRC2[31:0]
        ELSE DEST[i+31:i] ← SRC1[i+31:i] + SRC2[i+31:i]
      FI;
    ELSE
      IF *merging-masking*           ; merging-masking
        THEN *DEST[i+31:i] remains unchanged*
      ELSE *zeroing-masking*         ; zeroing-masking
        DEST[i+31:i] ← 0
      FI
    FI;
  FI;
ENDFOR;
DEST[MAX_VL-1:VL] ← 0

```

**VPADDQ (EVEX encoded versions)**

(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) AND (SRC2 \*is memory\*)

THEN DEST[i+63:i] ← SRC1[i+63:i] + SRC2[63:0]

ELSE DEST[i+63:i] ← SRC1[i+63:i] + SRC2[i+63:i]

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR;

DEST[MAX\_VL-1:VL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalents**

VPADDB \_\_m512i\_mm512\_add\_epi8 (\_\_m512i a, \_\_m512i b)

VPADDW \_\_m512i\_mm512\_add\_epi16 (\_\_m512i a, \_\_m512i b)

VPADDB \_\_m512i\_mm512\_mask\_add\_epi8 (\_\_m512i s, \_\_mmask64 m, \_\_m512i a, \_\_m512i b)

VPADDW \_\_m512i\_mm512\_mask\_add\_epi16 (\_\_m512i s, \_\_mmask32 m, \_\_m512i a, \_\_m512i b)

VPADDB \_\_m512i\_mm512\_maskz\_add\_epi8 (\_\_mmask64 m, \_\_m512i a, \_\_m512i b)

VPADDW \_\_m512i\_mm512\_maskz\_add\_epi16 (\_\_mmask32 m, \_\_m512i a, \_\_m512i b)

VPADDB \_\_m256i\_mm256\_mask\_add\_epi8 (\_\_m256i s, \_\_mmask32 m, \_\_m256i a, \_\_m256i b)

VPADDW \_\_m256i\_mm256\_mask\_add\_epi16 (\_\_m256i s, \_\_mmask16 m, \_\_m256i a, \_\_m256i b)

VPADDB \_\_m256i\_mm256\_maskz\_add\_epi8 (\_\_mmask32 m, \_\_m256i a, \_\_m256i b)

VPADDW \_\_m256i\_mm256\_maskz\_add\_epi16 (\_\_mmask16 m, \_\_m256i a, \_\_m256i b)

VPADDB \_\_m128i\_mm\_mask\_add\_epi8 (\_\_m128i s, \_\_mmask16 m, \_\_m128i a, \_\_m128i b)

VPADDW \_\_m128i\_mm\_mask\_add\_epi16 (\_\_m128i s, \_\_mmask8 m, \_\_m128i a, \_\_m128i b)

VPADDB \_\_m128i\_mm\_maskz\_add\_epi8 (\_\_mmask16 m, \_\_m128i a, \_\_m128i b)

VPADDW \_\_m128i\_mm\_maskz\_add\_epi16 (\_\_mmask8 m, \_\_m128i a, \_\_m128i b)

VPADDD \_\_m512i\_mm512\_add\_epi32(\_\_m512i a, \_\_m512i b);

VPADDD \_\_m512i\_mm512\_mask\_add\_epi32(\_\_m512i s, \_\_mmask16 k, \_\_m512i a, \_\_m512i b);

VPADDD \_\_m512i\_mm512\_maskz\_add\_epi32(\_\_mmask16 k, \_\_m512i a, \_\_m512i b);

VPADDD \_\_m256i\_mm256\_mask\_add\_epi32(\_\_m256i s, \_\_mmask8 k, \_\_m256i a, \_\_m256i b);

VPADDD \_\_m256i\_mm256\_maskz\_add\_epi32(\_\_mmask8 k, \_\_m256i a, \_\_m256i b);

VPADDD \_\_m128i\_mm\_mask\_add\_epi32(\_\_m128i s, \_\_mmask8 k, \_\_m128i a, \_\_m128i b);

VPADDD \_\_m128i\_mm\_maskz\_add\_epi32(\_\_mmask8 k, \_\_m128i a, \_\_m128i b);

VPADDQ \_\_m512i\_mm512\_add\_epi64(\_\_m512i a, \_\_m512i b);

VPADDQ \_\_m512i\_mm512\_mask\_add\_epi64(\_\_m512i s, \_\_mmask8 k, \_\_m512i a, \_\_m512i b);

VPADDQ \_\_m512i\_mm512\_maskz\_add\_epi64(\_\_mmask8 k, \_\_m512i a, \_\_m512i b);

VPADDQ \_\_m256i\_mm256\_mask\_add\_epi64(\_\_m256i s, \_\_mmask8 k, \_\_m256i a, \_\_m256i b);

VPADDQ \_\_m256i\_mm256\_maskz\_add\_epi64(\_\_mmask8 k, \_\_m256i a, \_\_m256i b);

VPADDQ \_\_m128i\_mm\_mask\_add\_epi64(\_\_m128i s, \_\_mmask8 k, \_\_m128i a, \_\_m128i b);

VPADDQ \_\_m128i\_mm\_maskz\_add\_epi64(\_\_mmask8 k, \_\_m128i a, \_\_m128i b);

PADDB \_\_m128i\_mm\_add\_epi8 (\_\_m128i a, \_\_m128i b);

PADDW \_\_m128i\_mm\_add\_epi16 (\_\_m128i a, \_\_m128i b);

PADDD \_\_m128i\_mm\_add\_epi32 (\_\_m128i a, \_\_m128i b);

PADDDQ \_\_m128i\_mm\_add\_epi64 (\_\_m128i a, \_\_m128i b);

VPADDB \_\_m256i \_\_mm256\_add\_epi8 (\_\_m256ia, \_\_m256i b);  
VPADDW \_\_m256i \_\_mm256\_add\_epi16 (\_\_m256i a, \_\_m256i b);  
VPADDQ \_\_m256i \_\_mm256\_add\_epi32 (\_\_m256i a, \_\_m256i b);  
VPADDQ \_\_m256i \_\_mm256\_add\_epi64 (\_\_m256i a, \_\_m256i b);  
PADDB \_\_m64 \_\_mm\_add\_pi8(\_\_m64 m1, \_\_m64 m2)  
PADDW \_\_m64 \_\_mm\_add\_pi16(\_\_m64 m1, \_\_m64 m2)  
PADDD \_\_m64 \_\_mm\_add\_pi32(\_\_m64 m1, \_\_m64 m2)  
PADDQ \_\_m64 \_\_mm\_add\_pi64(\_\_m64 m1, \_\_m64 m2)

### SIMD Floating-Point Exceptions

None

### Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded VPADDQ, see Exceptions Type E4.

EVEX-encoded VPADDB/W, see Exceptions Type E4.nb.

## PADDSB/PADDSW—Add Packed Signed Integers with Signed Saturation

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F EC /r <sup>1</sup> PADDSB <i>mm, mm/m64</i>	RM	V/V	MMX	Add packed signed byte integers from <i>mm/m64</i> and <i>mm</i> and saturate the results.
66 0F EC /r PADDSB <i>xmm1, xmm2/m128</i>	RM	V/V	SSE2	Add packed signed byte integers from <i>xmm2/m128</i> and <i>xmm1</i> saturate the results.
0F ED /r <sup>1</sup> PADDSW <i>mm, mm/m64</i>	RM	V/V	MMX	Add packed signed word integers from <i>mm/m64</i> and <i>mm</i> and saturate the results.
66 0F ED /r PADDSW <i>xmm1, xmm2/m128</i>	RM	V/V	SSE2	Add packed signed word integers from <i>xmm2/m128</i> and <i>xmm1</i> and saturate the results.
VEX.NDS.128.66.0F.WIG EC /r VPADDSB <i>xmm1, xmm2, xmm3/m128</i>	RVM	V/V	AVX	Add packed signed byte integers from <i>xmm3/m128</i> and <i>xmm2</i> saturate the results.
VEX.NDS.128.66.0F.WIG ED /r VPADDSW <i>xmm1, xmm2, xmm3/m128</i>	RVM	V/V	AVX	Add packed signed word integers from <i>xmm3/m128</i> and <i>xmm2</i> and saturate the results.
VEX.NDS.256.66.0F.WIG EC /r VPADDSB <i>ymm1, ymm2, ymm3/m256</i>	RVM	V/V	AVX2	Add packed signed byte integers from <i>ymm2</i> , and <i>ymm3/m256</i> and store the saturated results in <i>ymm1</i> .
VEX.NDS.256.66.0F.WIG ED /r VPADDSW <i>ymm1, ymm2, ymm3/m256</i>	RVM	V/V	AVX2	Add packed signed word integers from <i>ymm2</i> , and <i>ymm3/m256</i> and store the saturated results in <i>ymm1</i> .
EVEX.NDS.128.66.0F.WIG EC /r VPADDSB <i>xmm1 {k1}{z}, xmm2, xmm3/m128</i>	FVM	V/V	AVX512VL AVX512BW	Add packed signed byte integers from <i>xmm2</i> , and <i>xmm3/m128</i> and store the saturated results in <i>xmm1</i> under writemask <i>k1</i> .
EVEX.NDS.256.66.0F.WIG EC /r VPADDSB <i>ymm1 {k1}{z}, ymm2, ymm3/m256</i>	FVM	V/V	AVX512VL AVX512BW	Add packed signed byte integers from <i>ymm2</i> , and <i>ymm3/m256</i> and store the saturated results in <i>ymm1</i> under writemask <i>k1</i> .
EVEX.NDS.512.66.0F.WIG EC /r VPADDSB <i>zmm1 {k1}{z}, zmm2, zmm3/m512</i>	FVM	V/V	AVX512BW	Add packed signed byte integers from <i>zmm2</i> , and <i>zmm3/m512</i> and store the saturated results in <i>zmm1</i> under writemask <i>k1</i> .
EVEX.NDS.128.66.0F.WIG ED /r VPADDSW <i>xmm1 {k1}{z}, xmm2, xmm3/m128</i>	FVM	V/V	AVX512VL AVX512BW	Add packed signed word integers from <i>xmm2</i> , and <i>xmm3/m128</i> and store the saturated results in <i>xmm1</i> under writemask <i>k1</i> .
EVEX.NDS.256.66.0F.WIG ED /r VPADDSW <i>ymm1 {k1}{z}, ymm2, ymm3/m256</i>	FVM	V/V	AVX512VL AVX512BW	Add packed signed word integers from <i>ymm2</i> , and <i>ymm3/m256</i> and store the saturated results in <i>ymm1</i> under writemask <i>k1</i> .
EVEX.NDS.512.66.0F.WIG ED /r VPADDSW <i>zmm1 {k1}{z}, zmm2, zmm3/m512</i>	FVM	V/V	AVX512BW	Add packed signed word integers from <i>zmm2</i> , and <i>zmm3/m512</i> and store the saturated results in <i>zmm1</i> under writemask <i>k1</i> .

### NOTES:

1. See note in Section 2.4, "AVX and SSE Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

## Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FVM	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

## Description

Performs a SIMD add of the packed signed integers from the source operand (second operand) and the destination operand (first operand), and stores the packed integer results in the destination operand. See Figure 9-4 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for an illustration of a SIMD operation. Overflow is handled with signed saturation, as described in the following paragraphs.

(V)PADD SB performs a SIMD add of the packed signed integers with saturation from the first source operand and second source operand and stores the packed integer results in the destination operand. When an individual byte result is beyond the range of a signed byte integer (that is, greater than 7FH or less than 80H), the saturated value of 7FH or 80H, respectively, is written to the destination operand.

(V)PADD SW performs a SIMD add of the packed signed word integers with saturation from the first source operand and second source operand and stores the packed integer results in the destination operand. When an individual word result is beyond the range of a signed word integer (that is, greater than 7FFFH or less than 8000H), the saturated value of 7FFFH or 8000H, respectively, is written to the destination operand.

EVEX encoded versions: The first source operand is an ZMM/YMM/XMM register. The second source operand is an ZMM/YMM/XMM register or a memory location. The destination operand is an ZMM/YMM/XMM register.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register.

VEX.128 encoded version: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAX\_VL-1:128) of the corresponding register destination are zeroed.

128-bit Legacy SSE version: The first source operand is an XMM register. The second operand can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAX\_VL-1:128) of the corresponding register destination are unmodified.

## Operation

**PADD SB (with 64-bit operands)**

```
DEST[7:0] ← SaturateToSignedByte(DEST[7:0] + SRC[7:0]);
(* Repeat add operation for 2nd through 7th bytes *)
DEST[63:56] ← SaturateToSignedByte(DEST[63:56] + SRC[63:56]);
```

**PADD SB (with 128-bit operands)**

```
DEST[7:0] ← SaturateToSignedByte (DEST[7:0] + SRC[7:0]);
(* Repeat add operation for 2nd through 14th bytes *)
DEST[127:120] ← SaturateToSignedByte (DEST[111:120] + SRC[127:120]);
```

**VPADD SB (VEX.128 encoded version)**

```
DEST[7:0] ← SaturateToSignedByte (SRC1[7:0] + SRC2[7:0]);
(* Repeat subtract operation for 2nd through 14th bytes *)
DEST[127:120] ← SaturateToSignedByte (SRC1[111:120] + SRC2[127:120]);
DEST[VLMAX-1:128] ← 0
```

**VPADD SB (VEX.256 encoded version)**

```
DEST[7:0] ← SaturateToSignedByte (SRC1[7:0] + SRC2[7:0]);
(* Repeat add operation for 2nd through 31st bytes *)
DEST[255:248] ← SaturateToSignedByte (SRC1[255:248] + SRC2[255:248]);
```

**VPADDSB (EVEX encoded versions)**

(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] ← SaturateToSignedByte (SRC1[i+7:i] + SRC2[i+7:i])
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+7:i] remains unchanged*
    ELSE *zeroing-masking* ; zeroing-masking
      DEST[i+7:i] = 0
    FI
  FI;
ENDFOR;
DEST[MAX_VL-1:VL] ← 0

```

**PADDSW (with 64-bit operands)**

```

DEST[15:0] ← SaturateToSignedWord(DEST[15:0] + SRC[15:0]);
(* Repeat add operation for 2nd and 7th words *)
DEST[63:48] ← SaturateToSignedWord(DEST[63:48] + SRC[63:48]);

```

**PADDSW (with 128-bit operands)**

```

DEST[15:0] ← SaturateToSignedWord (DEST[15:0] + SRC[15:0]);
(* Repeat add operation for 2nd through 7th words *)
DEST[127:112] ← SaturateToSignedWord (DEST[127:112] + SRC[127:112]);

```

**VPADDSW (VEX.128 encoded version)**

```

DEST[15:0] ← SaturateToSignedWord (SRC1[15:0] + SRC2[15:0]);
(* Repeat subtract operation for 2nd through 7th words *)
DEST[127:112] ← SaturateToSignedWord (SRC1[127:112] + SRC2[127:112]);
DEST[VLMAX-1:128] ← 0

```

**VPADDSW (VEX.256 encoded version)**

```

DEST[15:0] ← SaturateToSignedWord (SRC1[15:0] + SRC2[15:0]);
(* Repeat add operation for 2nd through 15th words *)
DEST[255:240] ← SaturateToSignedWord (SRC1[255:240] + SRC2[255:240])

```

**VPADDSW (EVEX encoded versions)**

(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] ← SaturateToSignedWord (SRC1[i+15:i] + SRC2[i+15:i])
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+15:i] remains unchanged*
    ELSE *zeroing-masking* ; zeroing-masking
      DEST[i+15:i] = 0
    FI
  FI;
ENDFOR;
DEST[MAX_VL-1:VL] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalents**

PADDSSB: `__m64 _mm_adds_pi8(__m64 m1, __m64 m2)`  
 (V)PADDSSB: `__m128i _mm_adds_epi8 (__m128i a, __m128i b)`  
 VPADDSSB: `__m256i _mm256_adds_epi8 (__m256i a, __m256i b)`  
 PADDSSW: `__m64 _mm_adds_pi16(__m64 m1, __m64 m2)`  
 (V)PADDSSW: `__m128i _mm_adds_epi16 (__m128i a, __m128i b)`  
 VPADDSSW: `__m256i _mm256_adds_epi16 (__m256i a, __m256i b)`  
 VPADDSSB\_\_m512i\_\_mm512\_adds\_epi8 (`__m512i a, __m512i b`)  
 VPADDSSW\_\_m512i\_\_mm512\_adds\_epi16 (`__m512i a, __m512i b`)  
 VPADDSSB\_\_m512i\_\_mm512\_mask\_adds\_epi8 (`__m512i s, __mmask64 m, __m512i a, __m512i b`)  
 VPADDSSW\_\_m512i\_\_mm512\_mask\_adds\_epi16 (`__m512i s, __mmask32 m, __m512i a, __m512i b`)  
 VPADDSSB\_\_m512i\_\_mm512\_maskz\_adds\_epi8 (`__mmask64 m, __m512i a, __m512i b`)  
 VPADDSSW\_\_m512i\_\_mm512\_maskz\_adds\_epi16 (`__mmask32 m, __m512i a, __m512i b`)  
 VPADDSSB\_\_m256i\_\_mm256\_mask\_adds\_epi8 (`__m256i s, __mmask32 m, __m256i a, __m256i b`)  
 VPADDSSW\_\_m256i\_\_mm256\_mask\_adds\_epi16 (`__m256i s, __mmask16 m, __m256i a, __m256i b`)  
 VPADDSSB\_\_m256i\_\_mm256\_maskz\_adds\_epi8 (`__mmask32 m, __m256i a, __m256i b`)  
 VPADDSSW\_\_m256i\_\_mm256\_maskz\_adds\_epi16 (`__mmask16 m, __m256i a, __m256i b`)  
 VPADDSSB\_\_m128i\_\_mm\_mask\_adds\_epi8 (`__m128i s, __mmask16 m, __m128i a, __m128i b`)  
 VPADDSSW\_\_m128i\_\_mm\_mask\_adds\_epi16 (`__m128i s, __mmask8 m, __m128i a, __m128i b`)  
 VPADDSSB\_\_m128i\_\_mm\_maskz\_adds\_epi8 (`__mmask16 m, __m128i a, __m128i b`)  
 VPADDSSW\_\_m128i\_\_mm\_maskz\_adds\_epi16 (`__mmask8 m, __m128i a, __m128i b`)

**Flags Affected**

None.

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.nb.

## PADDUSB/PADDUSW—Add Packed Unsigned Integers with Unsigned Saturation

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F DC /r <sup>1</sup> PADDUSB <i>mm</i> , <i>mm/m64</i>	RM	V/V	MMX	Add packed unsigned byte integers from <i>mm/m64</i> and <i>mm</i> and saturate the results.
66 0F DC /r PADDUSB <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE2	Add packed unsigned byte integers from <i>xmm2/m128</i> and <i>xmm1</i> saturate the results.
0F DD /r <sup>1</sup> PADDUSW <i>mm</i> , <i>mm/m64</i>	RM	V/V	MMX	Add packed unsigned word integers from <i>mm/m64</i> and <i>mm</i> and saturate the results.
66 0F DD /r PADDUSW <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE2	Add packed unsigned word integers from <i>xmm2/m128</i> to <i>xmm1</i> and saturate the results.
VEX.NDS.128.66.0F.WIG DC /r VPADDUSB <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Add packed unsigned byte integers from <i>xmm3/m128</i> to <i>xmm2</i> and saturate the results.
VEX.NDS.128.66.0F.WIG DD /r VPADDUSW <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Add packed unsigned word integers from <i>xmm3/m128</i> to <i>xmm2</i> and saturate the results.
VEX.NDS.256.66.0F.WIG DC /r VPADDUSB <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	RVM	V/V	AVX2	Add packed unsigned byte integers from <i>ymm2</i> , and <i>ymm3/m256</i> and store the saturated results in <i>ymm1</i> .
VEX.NDS.256.66.0F.WIG DD /r VPADDUSW <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	RVM	V/V	AVX2	Add packed unsigned word integers from <i>ymm2</i> , and <i>ymm3/m256</i> and store the saturated results in <i>ymm1</i> .
EVEX.NDS.128.66.0F.WIG DC /r VPADDUSB <i>xmm1</i> { <i>k1</i> }{ <i>z</i> }, <i>xmm2</i> , <i>xmm3/m128</i>	FVM	V/V	AVX512VL AVX512BW	Add packed unsigned byte integers from <i>xmm2</i> , and <i>xmm3/m128</i> and store the saturated results in <i>xmm1</i> under writemask <i>k1</i> .
EVEX.NDS.256.66.0F.WIG DC /r VPADDUSB <i>ymm1</i> { <i>k1</i> }{ <i>z</i> }, <i>ymm2</i> , <i>ymm3/m256</i>	FVM	V/V	AVX512VL AVX512BW	Add packed unsigned byte integers from <i>ymm2</i> , and <i>ymm3/m256</i> and store the saturated results in <i>ymm1</i> under writemask <i>k1</i> .
EVEX.NDS.512.66.0F.WIG DC /r VPADDUSB <i>zmm1</i> { <i>k1</i> }{ <i>z</i> }, <i>zmm2</i> , <i>zmm3/m512</i>	FVM	V/V	AVX512BW	Add packed unsigned byte integers from <i>zmm2</i> , and <i>zmm3/m512</i> and store the saturated results in <i>zmm1</i> under writemask <i>k1</i> .
EVEX.NDS.128.66.0F.WIG DD /r VPADDUSW <i>xmm1</i> { <i>k1</i> }{ <i>z</i> }, <i>xmm2</i> , <i>xmm3/m128</i>	FVM	V/V	AVX512VL AVX512BW	Add packed unsigned word integers from <i>xmm2</i> , and <i>xmm3/m128</i> and store the saturated results in <i>xmm1</i> under writemask <i>k1</i> .
EVEX.NDS.256.66.0F.WIG DD /r VPADDUSW <i>ymm1</i> { <i>k1</i> }{ <i>z</i> }, <i>ymm2</i> , <i>ymm3/m256</i>	FVM	V/V	AVX512VL AVX512BW	Add packed unsigned word integers from <i>ymm2</i> , and <i>ymm3/m256</i> and store the saturated results in <i>ymm1</i> under writemask <i>k1</i> .



EVEX.NDS.512.66.0F.WIG DD /r VPADDUSW <i>zmm1</i> { <i>k1</i> }{ <i>z</i> }, <i>zmm2</i> , <i>zmm3/m512</i>	FVM V/V	AVX512BW	Add packed unsigned word integers from <i>zmm2</i> , and <i>zmm3/m512</i> and store the saturated results in <i>zmm1</i> under writemask <i>k1</i> .
----------------------------------------------------------------------------------------------------------------	---------	----------	------------------------------------------------------------------------------------------------------------------------------------------------------

**NOTES:**

1. See note in Section 2.4, “AVX and SSE Instruction Exception Specification” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FVM	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

**Description**

Performs a SIMD add of the packed unsigned integers from the source operand (second operand) and the destination operand (first operand), and stores the packed integer results in the destination operand. See Figure 9-4 in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for an illustration of a SIMD operation. Overflow is handled with unsigned saturation, as described in the following paragraphs.

(V)PADDUSB performs a SIMD add of the packed unsigned integers with saturation from the first source operand and second source operand and stores the packed integer results in the destination operand. When an individual byte result is beyond the range of an unsigned byte integer (that is, greater than FFH), the saturated value of FFH is written to the destination operand.

(V)PADDUSW performs a SIMD add of the packed unsigned word integers with saturation from the first source operand and second source operand and stores the packed integer results in the destination operand. When an individual word result is beyond the range of an unsigned word integer (that is, greater than FFFFH), the saturated value of FFFFH is written to the destination operand.

EVEX encoded versions: The first source operand is an ZMM/YMM/XMM register. The second source operand is an ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination is an ZMM/YMM/XMM register.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register.

VEX.128 encoded version: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAX\_VL-1:128) of the corresponding destination register destination are zeroed.

128-bit Legacy SSE version: The first source operand is an XMM register. The second operand can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAX\_VL-1:128) of the corresponding register destination are unmodified.

**Operation****PADDUSB (with 64-bit operands)**

```
DEST[7:0] ← SaturateToUnsignedByte(DEST[7:0] + SRC[7:0]);
(* Repeat add operation for 2nd through 7th bytes *)
DEST[63:56] ← SaturateToUnsignedByte(DEST[63:56] + SRC[63:56])
```

**PADDUSW (with 128-bit operands)**

```
DEST[7:0] ← SaturateToUnsignedByte (DEST[7:0] + SRC[7:0]);
(* Repeat add operation for 2nd through 14th bytes *)
DEST[127:120] ← SaturateToUnSignedByte (DEST[127:120] + SRC[127:120]);
```

**VPADDUSB (VEX.128 encoded version)**

DEST[7:0] ← SaturateToUnsignedByte (SRC1[7:0] + SRC2[7:0]);  
 (\* Repeat subtract operation for 2nd through 14th bytes \*)  
 DEST[127:120] ← SaturateToUnsignedByte (SRC1[111:120] + SRC2[127:120]);  
 DEST[VLMAX-1:128] ← 0

**VPADDUSB (VEX.256 encoded version)**

DEST[7:0] ← SaturateToUnsignedByte (SRC1[7:0] + SRC2[7:0]);  
 (\* Repeat add operation for 2nd through 31st bytes \*)  
 DEST[255:248] ← SaturateToUnsignedByte (SRC1[255:248] + SRC2[255:248]);

**PADDUSW (with 64-bit operands)**

DEST[15:0] ← SaturateToUnsignedWord (DEST[15:0] + SRC[15:0] );  
 (\* Repeat add operation for 2nd and 3rd words \*)  
 DEST[63:48] ← SaturateToUnsignedWord (DEST[63:48] + SRC[63:48] );

**PADDUSW (with 128-bit operands)**

DEST[15:0] ← SaturateToUnsignedWord (DEST[15:0] + SRC[15:0]);  
 (\* Repeat add operation for 2nd through 7th words \*)  
 DEST[127:112] ← SaturateToUnsignedWord (DEST[127:112] + SRC[127:112]);

**VPADDUSW (VEX.128 encoded version)**

DEST[15:0] ← SaturateToUnsignedWord (SRC1[15:0] + SRC2[15:0]);  
 (\* Repeat subtract operation for 2nd through 7th words \*)  
 DEST[127:112] ← SaturateToUnsignedWord (SRC1[127:112] + SRC2[127:112]);  
 DEST[VLMAX-1:128] ← 0

**VPADDUSW (VEX.256 encoded version)**

DEST[15:0] ← SaturateToUnsignedWord (SRC1[15:0] + SRC2[15:0]);  
 (\* Repeat add operation for 2nd through 15th words \*)  
 DEST[255:240] ← SaturateToUnsignedWord (SRC1[255:240] + SRC2[255:240])

**VPADDUSB (EVEX encoded versions)**

(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] ← SaturateToUnsignedByte (SRC1[i+7:i] + SRC2[i+7:i])

    ELSE

      IF \*merging-masking\* ; merging-masking

        THEN \*DEST[i+7:i] remains unchanged\*

        ELSE \*zeroing-masking\* ; zeroing-masking

          DEST[i+7:i] = 0

    FI

  FI;

ENDFOR;

DEST[MAX\_VL-1:VL] ← 0

**VPADDUSW (EVEX encoded versions)**

(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[j+15:i] ← SaturateToUnsignedWord (SRC1[j+15:i] + SRC2[j+15:i])
ELSE
  IF *merging-masking*           ; merging-masking
    THEN *DEST[j+15:i] remains unchanged*
    ELSE *zeroing-masking*       ; zeroing-masking
      DEST[j+15:i] = 0
  FI
FI;
ENDFOR;
DEST[MAX_VL-1:VL] ← 0

```

### Intel C/C++ Compiler Intrinsic Equivalents

```

PADDUSB:   __m64 _mm_adds_pu8(__m64 m1, __m64 m2)
PADDUSW:   __m64 _mm_adds_pu16(__m64 m1, __m64 m2)
(V)PADDUSB: __m128i _mm_adds_epu8 (__m128i a, __m128i b)
(V)PADDUSW: __m128i _mm_adds_epu16 (__m128i a, __m128i b)
VPADDUSB:  __m256i _mm256_adds_epu8 (__m256i a, __m256i b)
VPADDUSW:  __m256i _mm256_adds_epu16 (__m256i a, __m256i b)
VPADDUSB__m512i _mm512_adds_epu8 (__m512i a, __m512i b)
VPADDUSW__m512i _mm512_adds_epu16 (__m512i a, __m512i b)
VPADDUSB__m512i _mm512_mask_adds_epu8 (__m512i s, __mmask64 m, __m512i a, __m512i b)
VPADDUSW__m512i _mm512_mask_adds_epu16 (__m512i s, __mmask32 m, __m512i a, __m512i b)
VPADDUSB__m512i _mm512_maskz_adds_epu8 (__mmask64 m, __m512i a, __m512i b)
VPADDUSW__m512i _mm512_maskz_adds_epu16 (__mmask32 m, __m512i a, __m512i b)
VPADDUSB__m256i _mm256_mask_adds_epu8 (__m256i s, __mmask32 m, __m256i a, __m256i b)
VPADDUSW__m256i _mm256_mask_adds_epu16 (__m256i s, __mmask16 m, __m256i a, __m256i b)
VPADDUSB__m256i _mm256_maskz_adds_epu8 (__mmask32 m, __m256i a, __m256i b)
VPADDUSW__m256i _mm256_maskz_adds_epu16 (__mmask16 m, __m256i a, __m256i b)
VPADDUSB__m128i _mm_mask_adds_epu8 (__m128i s, __mmask16 m, __m128i a, __m128i b)
VPADDUSW__m128i _mm_mask_adds_epu16 (__m128i s, __mmask8 m, __m128i a, __m128i b)
VPADDUSB__m128i _mm_maskz_adds_epu8 (__mmask16 m, __m128i a, __m128i b)
VPADDUSW__m128i _mm_maskz_adds_epu16 (__mmask8 m, __m128i a, __m128i b)

```

### Flags Affected

None.

### Numeric Exceptions

None.

### Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.nb.

## PALIGNR — Packed Align Right

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
OF 3A OF /r ib <sup>1</sup> PALIGNR <i>mm1</i> , <i>mm2/m64</i> , <i>imm8</i>	RMI	V/V	SSSE3	Concatenate destination and source operands, extract byte-aligned result shifted to the right by constant value in <i>imm8</i> into <i>mm1</i> .
66 OF 3A OF /r ib PALIGNR <i>xmm1</i> , <i>xmm2/m128</i> , <i>imm8</i>	RMI	V/V	SSSE3	Concatenate destination and source operands, extract byte-aligned result shifted to the right by constant value in <i>imm8</i> into <i>xmm1</i> .
VEX.NDS.128.66.OF3A.WIG OF /r ib VPALIGNR <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i> , <i>imm8</i>	RVMI	V/V	AVX	Concatenate <i>xmm2</i> and <i>xmm3/m128</i> , extract byte aligned result shifted to the right by constant value in <i>imm8</i> and result is stored in <i>xmm1</i> .
VEX.NDS.256.66.OF3A.WIG OF /r ib VPALIGNR <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i> , <i>imm8</i>	RVMI	V/V	AVX2	Concatenate pairs of 16 bytes in <i>ymm2</i> and <i>ymm3/m256</i> into 32-byte intermediate result, extract byte-aligned, 16-byte result shifted to the right by constant values in <i>imm8</i> from each intermediate result, and two 16-byte results are stored in <i>ymm1</i> .
EVEX.NDS.128.66.OF3A.WIG OF /r ib VPALIGNR <i>xmm1</i> { <i>k1</i> }{ <i>z</i> }, <i>xmm2</i> , <i>xmm3/m128</i> , <i>imm8</i>	FVM	V/V	AVX512VL AVX512BW	Concatenate <i>xmm2</i> and <i>xmm3/m128</i> into a 32-byte intermediate result, extract byte aligned result shifted to the right by constant value in <i>imm8</i> and result is stored in <i>xmm1</i> .
EVEX.NDS.256.66.OF3A.WIG OF /r ib VPALIGNR <i>ymm1</i> { <i>k1</i> }{ <i>z</i> }, <i>ymm2</i> , <i>ymm3/m256</i> , <i>imm8</i>	FVM	V/V	AVX512VL AVX512BW	Concatenate pairs of 16 bytes in <i>ymm2</i> and <i>ymm3/m256</i> into 32-byte intermediate result, extract byte-aligned, 16-byte result shifted to the right by constant values in <i>imm8</i> from each intermediate result, and two 16-byte results are stored in <i>ymm1</i> .
EVEX.NDS.512.66.OF3A.WIG OF /r ib VPALIGNR <i>zmm1</i> { <i>k1</i> }{ <i>z</i> }, <i>zmm2</i> , <i>zmm3/m512</i> , <i>imm8</i>	FVM	V/V	AVX512BW	Concatenate pairs of 16 bytes in <i>zmm2</i> and <i>zmm3/m512</i> into 32-byte intermediate result, extract byte-aligned, 16-byte result shifted to the right by constant values in <i>imm8</i> from each intermediate result, and four 16-byte results are stored in <i>zmm1</i> .

### NOTES:

1. See note in Section 2.4, “AVX and SSE Instruction Exception Specification” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (r, w)	ModRM:r/m (r)	imm8	NA
RVMI	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8
FVM	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

(V)PALIGNR concatenates the destination operand (the first operand) and the source operand (the second operand) into an intermediate composite, shifts the composite at byte granularity to the right by a constant immediate, and extracts the right-aligned result into the destination. The first and the second operands can be an MMX,

XMM or a YMM register. The immediate value is considered unsigned. Immediate shift counts larger than the 2L (i.e. 32 for 128-bit operands, or 16 for 64-bit operands) produce a zero result. Both operands can be MMX registers, XMM registers or YMM registers. When the source operand is a 128-bit memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

In 64-bit mode and not encoded by VEX/EVEX prefix, use the REX prefix to access additional registers.

128-bit Legacy SSE version: Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

EVEX.512 encoded version: The first source operand is a ZMM register and contains four 16-byte blocks. The second source operand is a ZMM register or a 512-bit memory location containing four 16-byte block. The destination operand is a ZMM register and contain four 16-byte results. The `imm8[7:0]` is the common shift count used for each of the four successive 16-byte block sources. The low 16-byte block of the two source operands produce the low 16-byte result of the destination operand, the high 16-byte block of the two source operands produce the high 16-byte result of the destination operand and so on for the blocks in the middle.

VEX.256 and EVEX.256 encoded versions: The first source operand is a YMM register and contains two 16-byte blocks. The second source operand is a YMM register or a 256-bit memory location containing two 16-byte block. The destination operand is a YMM register and contain two 16-byte results. The `imm8[7:0]` is the common shift count used for the two lower 16-byte block sources and the two upper 16-byte block sources. The low 16-byte block of the two source operands produce the low 16-byte result of the destination operand, the high 16-byte block of the two source operands produce the high 16-byte result of the destination operand. The upper bits (MAX\_VL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 and EVEX.128 encoded versions: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAX\_VL-1:128) of the corresponding ZMM register destination are zeroed.

Concatenation is done with 128-bit data in the first and second source operand for both 128-bit and 256-bit instructions. The high 128-bits of the intermediate composite 256-bit result came from the 128-bit data from the first source operand; the low 128-bits of the intermediate result came from the 128-bit data of the second source operand.

Note: VEX.L must be 0, otherwise the instruction will #UD.

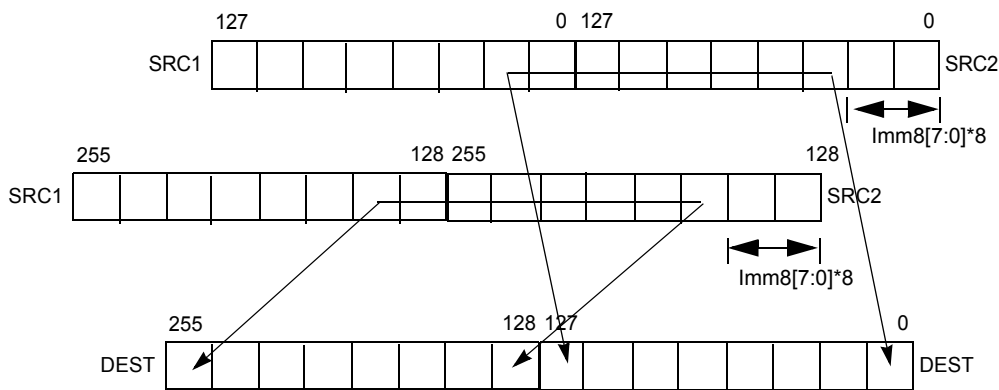


Figure 4-6. 256-bit VPALIGN Instruction Operation

## Operation

### PALIGNR (with 64-bit operands)

`temp1[127:0] = CONCATENATE(DEST, SRC) >> (imm8*8)`

`DEST[63:0] = temp1[63:0]`

**PALIGNR (with 128-bit operands)**

```
temp1[255:0] ← ((DEST[127:0] << 128) OR SRC[127:0])>>(imm8*8);
DEST[127:0] ← temp1[127:0]
DEST[VLMAX-1:128] (Unmodified)
```

**VPALIGNR (VEX.128 encoded version)**

```
temp1[255:0] ← ((SRC1[127:0] << 128) OR SRC2[127:0])>>(imm8*8);
DEST[127:0] ← temp1[127:0]
DEST[VLMAX-1:128] ← 0
```

**VPALIGNR (VEX.256 encoded version)**

```
temp1[255:0] ← ((SRC1[127:0] << 128) OR SRC2[127:0])>>(imm8[7:0]*8);
DEST[127:0] ← temp1[127:0]
temp1[255:0] ← ((SRC1[255:128] << 128) OR SRC2[255:128])>>(imm8[7:0]*8);
DEST[MAX_VL-1:128] ← temp1[127:0]
```

**VPALIGNR (EVEX encoded versions)**

(KL, VL) = (16, 128), (32, 256), (64, 512)

FOR I ← 0 TO VL-1 with increments of 128

```
temp1[255:0] ← ((SRC1[I+127:I] << 128) OR SRC2[I+127:I])>>(imm8[7:0]*8);
TMP_DEST[I+127:I] ← temp1[127:0]
```

ENDFOR;

FOR j ← 0 TO KL-1

i ← j \* 8

IF k1[j] OR \*no writemask\*

THEN DEST[i+7:i] ← TMP\_DEST[i+7:i]

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+7:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[i+7:i] = 0

FI

FI;

ENDFOR;

DEST[MAX\_VL-1:VL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalents**

PALIGNR: `__m64 _mm_alignr_pi8 (__m64 a, __m64 b, int n)`

(V)PALIGNR: `__m128i _mm_alignr_epi8 (__m128i a, __m128i b, int n)`

VPALIGNR: `__m256i _mm256_alignr_epi8 (__m256i a, __m256i b, const int n)`

VPALIGNR `__m512i _mm512_alignr_epi8 (__m512i a, __m512i b, const int n)`

VPALIGNR `__m512i _mm512_mask_alignr_epi8 (__m512i s, __mmask64 m, __m512i a, __m512i b, const int n)`

VPALIGNR `__m512i _mm512_maskz_alignr_epi8 (__mmask64 m, __m512i a, __m512i b, const int n)`

VPALIGNR `__m256i _mm256_mask_alignr_epi8 (__m256i s, __mmask32 m, __m256i a, __m256i b, const int n)`

VPALIGNR `__m256i _mm256_maskz_alignr_epi8 (__mmask32 m, __m256i a, __m256i b, const int n)`

VPALIGNR `__m128i _mm_mask_alignr_epi8 (__m128i s, __mmask16 m, __m128i a, __m128i b, const int n)`

VPALIGNR `__m128i _mm_maskz_alignr_epi8 (__mmask16 m, __m128i a, __m128i b, const int n)`

**SIMD Floating-Point Exceptions**

None.



### Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4NF.nb.

## PAND—Logical AND

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F DB /r <sup>1</sup> PAND mm, mm/m64	RM	V/V	MMX	Bitwise AND mm/m64 and mm.
66 0F DB /r PAND xmm1, xmm2/m128	RM	V/V	SSE2	Bitwise AND of xmm2/m128 and xmm1.
VEX.NDS.128.66.0F.WIG DB /r VPAND xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Bitwise AND of xmm3/m128 and xmm1.
VEX.NDS.256.66.0F.WIG DB /r VPAND ymm1, ymm2, ymm3/m256	RVM	V/V	AVX2	Bitwise AND of ymm2, and ymm3/m256 and store result in ymm1.
EVEX.NDS.128.66.0F.W0 DB /r VPANDD xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	FV	V/V	AVX512VL AVX512F	Bitwise AND of packed doubleword integers in xmm2 and xmm3/m128/m32bcst and store result in xmm1 using writemask k1.
EVEX.NDS.256.66.0F.W0 DB /r VPANDD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	FV	V/V	AVX512VL AVX512F	Bitwise AND of packed doubleword integers in ymm2 and ymm3/m256/m32bcst and store result in ymm1 using writemask k1.
EVEX.NDS.512.66.0F.W0 DB /r VPANDD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	FV	V/V	AVX512F	Bitwise AND of packed doubleword integers in zmm2 and zmm3/m512/m32bcst and store result in zmm1 using writemask k1.
EVEX.NDS.128.66.0F.W1 DB /r VPANDQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	FV	V/V	AVX512VL AVX512F	Bitwise AND of packed quadword integers in xmm2 and xmm3/m128/m64bcst and store result in xmm1 using writemask k1.
EVEX.NDS.256.66.0F.W1 DB /r VPANDQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	FV	V/V	AVX512VL AVX512F	Bitwise AND of packed quadword integers in ymm2 and ymm3/m256/m64bcst and store result in ymm1 using writemask k1.
EVEX.NDS.512.66.0F.W1 DB /r VPANDQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	FV	V/V	AVX512F	Bitwise AND of packed quadword integers in zmm2 and zmm3/m512/m64bcst and store result in zmm1 using writemask k1.

### NOTES:

1. See note in Section 2.4, “AVX and SSE Instruction Exception Specification” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FV	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description



Performs a bitwise logical AND operation on the first source operand and second source operand and stores the result in the destination operand. Each bit of the result is set to 1 if the corresponding bits of the first and second operands are 1, otherwise it is set to 0.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE instructions: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand can be an MMX technology register.

128-bit Legacy SSE version: The first source operand is an XMM register. The second operand can be an XMM register or a 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAX\_VL-1:128) of the corresponding ZMM register destination are unmodified.

EVEX encoded versions: The first source 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 32/64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1 at 32/64-bit granularity.

VEX.256 encoded versions: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAX\_VL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded versions: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAX\_VL-1:128) of the corresponding ZMM register destination are zeroed.

## Operation

### PAND (64-bit operand)

DEST ← DEST AND SRC

### PAND (128-bit Legacy SSE version)

DEST ← DEST AND SRC

DEST[VLMAX-1:128] (Unmodified)

### VPAND (VEX.128 encoded version)

DEST ← SRC1 AND SRC2

DEST[VLMAX-1:128] ← 0

### VPAND (VEX.256 encoded instruction)

DEST[255:0] ← (SRC1[255:0] AND SRC2[255:0])

DEST[VLMAX-1:256] ← 0

### VPANDD (EVEX encoded versions)

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

  i ← j \* 32

  IF k1[j] OR \*no writemask\*

    THEN

      IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

        THEN DEST[i+31:i] ← SRC1[i+31:i] BITWISE AND SRC2[31:0]

        ELSE DEST[i+31:i] ← SRC1[i+31:i] BITWISE AND SRC2[i+31:i]

      FI;

    ELSE

      IF \*merging-masking\* ; merging-masking

        THEN \*DEST[i+31:i] remains unchanged\*

        ELSE ; zeroing-masking

          DEST[i+31:i] ← 0

    FI

```

FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

### VPANDQ (EVEX encoded versions)

(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) AND (SRC2 *is memory*)
        THEN DEST[i+63:i] ← SRC1[i+63:i] BITWISE AND SRC2[63:0]
        ELSE DEST[i+63:i] ← SRC1[i+63:i] BITWISE AND SRC2[i+63:i]
      FI;
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+63:i] remains unchanged*
      ELSE ; zeroing-masking
        DEST[i+63:i] ← 0
      FI
    FI;
  ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

### Intel C/C++ Compiler Intrinsic Equivalents

```

VPANDD __m512i __mm512_and_epi32(__m512i a, __m512i b);
VPANDD __m512i __mm512_mask_and_epi32(__m512i s, __mmask16 k, __m512i a, __m512i b);
VPANDD __m512i __mm512_maskz_and_epi32(__mmask16 k, __m512i a, __m512i b);
VPANDQ __m512i __mm512_and_epi64(__m512i a, __m512i b);
VPANDQ __m512i __mm512_mask_and_epi64(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPANDQ __m512i __mm512_maskz_and_epi64(__mmask8 k, __m512i a, __m512i b);
VPANDND __m256i __mm256_mask_and_epi32(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPANDND __m256i __mm256_maskz_and_epi32(__mmask8 k, __m256i a, __m256i b);
VPANDND __m128i __mm_mask_and_epi32(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPANDND __m128i __mm_maskz_and_epi32(__mmask8 k, __m128i a, __m128i b);
VPANDNQ __m256i __mm256_mask_and_epi64(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPANDNQ __m256i __mm256_maskz_and_epi64(__mmask8 k, __m256i a, __m256i b);
VPANDNQ __m128i __mm_mask_and_epi64(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPANDNQ __m128i __mm_maskz_and_epi64(__mmask8 k, __m128i a, __m128i b);
PAND: __m64 __mm_and_si64(__m64 m1, __m64 m2)
(V)PAND: __m128i __mm_and_si128(__m128i a, __m128i b)
VPAND: __m256i __mm256_and_si256(__m256i a, __m256i b)

```

### Flags Affected

None.

### Numeric Exceptions

None.

### Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.

## PANDN—Logical AND NOT

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F DF /r <sup>1</sup> PANDN mm, mm/m64	RM	V/V	MMX	Bitwise AND NOT of mm/m64 and mm.
66 0F DF /r PANDN xmm1, xmm2/m128	RM	V/V	SSE2	Bitwise AND NOT of xmm2/m128 and xmm1.
VEX.NDS.128.66.0F.WIG DF /r VPANDN xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Bitwise AND NOT of xmm3/m128 and xmm2.
VEX.NDS.256.66.0F.WIG DF /r VPANDN ymm1, ymm2, ymm3/m256	RVM	V/V	AVX2	Bitwise AND NOT of ymm2, and ymm3/m256 and store result in ymm1.
EVEX.NDS.128.66.0F.WO DF /r VPANDND xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	FV	V/V	AVX512VL AVX512F	Bitwise AND NOT of packed doubleword integers in xmm2 and xmm3/m128/m32bcst and store result in xmm1 using writemask k1.
EVEX.NDS.256.66.0F.WO DF /r VPANDND ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	FV	V/V	AVX512VL AVX512F	Bitwise AND NOT of packed doubleword integers in ymm2 and ymm3/m256/m32bcst and store result in ymm1 using writemask k1.
EVEX.NDS.512.66.0F.WO DF /r VPANDND zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	FV	V/V	AVX512F	Bitwise AND NOT of packed doubleword integers in zmm2 and zmm3/m512/m32bcst and store result in zmm1 using writemask k1.
EVEX.NDS.128.66.0F.W1 DF /r VPANDNQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	FV	V/V	AVX512VL AVX512F	Bitwise AND NOT of packed quadword integers in xmm2 and xmm3/m128/m64bcst and store result in xmm1 using writemask k1.
EVEX.NDS.256.66.0F.W1 DF /r VPANDNQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	FV	V/V	AVX512VL AVX512F	Bitwise AND NOT of packed quadword integers in ymm2 and ymm3/m256/m64bcst and store result in ymm1 using writemask k1.
EVEX.NDS.512.66.0F.W1 DF /r VPANDNQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	FV	V/V	AVX512F	Bitwise AND NOT of packed quadword integers in zmm2 and zmm3/m512/m64bcst and store result in zmm1 using writemask k1.

### NOTES:

1. See note in Section 2.4, “AVX and SSE Instruction Exception Specification” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FV	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Performs a bitwise logical NOT operation on the first source operand, then performs bitwise AND with second source operand and stores the result in the destination operand. Each bit of the result is set to 1 if the corresponding bit in the first operand is 0 and the corresponding bit in the second operand is 1, otherwise it is set to 0.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE instructions: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand can be an MMX technology register.

128-bit Legacy SSE version: The first source operand is an XMM register. The second operand can be an XMM register or a 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAX\_VL-1:128) of the corresponding ZMM register destination are unmodified.

EVEX encoded versions: The first source 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 32/64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1 at 32/64-bit granularity.

VEX.256 encoded versions: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAX\_VL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded versions: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAX\_VL-1:128) of the corresponding ZMM register destination are zeroed.

## Operation

### PANDN (64-bit operand)

DEST  $\leftarrow$  NOT(DEST) AND SRC

### PANDN (128-bit Legacy SSE version)

DEST  $\leftarrow$  NOT(DEST) AND SRC

DEST[VLMAX-1:128] (Unmodified)

### VPANDN (VEX.128 encoded version)

DEST  $\leftarrow$  NOT(SRC1) AND SRC2

DEST[VLMAX-1:128]  $\leftarrow$  0

### VPANDN (VEX.256 encoded instruction)

DEST[255:0]  $\leftarrow$  ((NOT SRC1[255:0]) AND SRC2[255:0])

DEST[VLMAX-1:256]  $\leftarrow$  0

### VPANDND (EVEX encoded versions)

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j  $\leftarrow$  0 TO KL-1

  i  $\leftarrow$  j \* 32

  IF k1[j] OR \*no writemask\*

    THEN

      IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

        THEN DEST[i+31:i]  $\leftarrow$  ((NOT SRC1[i+31:i]) AND SRC2[31:0])

        ELSE DEST[i+31:i]  $\leftarrow$  ((NOT SRC1[i+31:i]) AND SRC2[i+31:i])

      FI;

    ELSE

      IF \*merging-masking\* ; merging-masking

        THEN \*DEST[i+31:i] remains unchanged\*

        ELSE ; zeroing-masking

          DEST[i+31:i]  $\leftarrow$  0

      FI

  FI;

ENDFOR

DEST[MAX\_VL-1:VL]  $\leftarrow$  0

**VPANDNQ (EVEX encoded versions)**

(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) AND (SRC2 \*is memory\*)

THEN DEST[i+63:i] ← ((NOT SRC1[i+63:i]) AND SRC2[63:0])

ELSE DEST[i+63:i] ← ((NOT SRC1[j+63:i]) AND SRC2[j+63:i])

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalents**

VPANDND \_\_m512i \_\_mm512\_andnot\_epi32(\_\_m512i a, \_\_m512i b);

VPANDND \_\_m512i \_\_mm512\_mask\_andnot\_epi32(\_\_m512i s, \_\_mmask16 k, \_\_m512i a, \_\_m512i b);

VPANDND \_\_m512i \_\_mm512\_maskz\_andnot\_epi32(\_\_mmask16 k, \_\_m512i a, \_\_m512i b);

VPANDND \_\_m256i \_\_mm256\_mask\_andnot\_epi32(\_\_m256i s, \_\_mmask8 k, \_\_m256i a, \_\_m256i b);

VPANDND \_\_m256i \_\_mm256\_maskz\_andnot\_epi32(\_\_mmask8 k, \_\_m256i a, \_\_m256i b);

VPANDND \_\_m128i \_\_mm\_mask\_andnot\_epi32(\_\_m128i s, \_\_mmask8 k, \_\_m128i a, \_\_m128i b);

VPANDND \_\_m128i \_\_mm\_maskz\_andnot\_epi32(\_\_mmask8 k, \_\_m128i a, \_\_m128i b);

VPANDNQ \_\_m512i \_\_mm512\_andnot\_epi64(\_\_m512i a, \_\_m512i b);

VPANDNQ \_\_m512i \_\_mm512\_mask\_andnot\_epi64(\_\_m512i s, \_\_mmask8 k, \_\_m512i a, \_\_m512i b);

VPANDNQ \_\_m512i \_\_mm512\_maskz\_andnot\_epi64(\_\_mmask8 k, \_\_m512i a, \_\_m512i b);

VPANDNQ \_\_m256i \_\_mm256\_mask\_andnot\_epi64(\_\_m256i s, \_\_mmask8 k, \_\_m256i a, \_\_m256i b);

VPANDNQ \_\_m256i \_\_mm256\_maskz\_andnot\_epi64(\_\_mmask8 k, \_\_m256i a, \_\_m256i b);

VPANDNQ \_\_m128i \_\_mm\_mask\_andnot\_epi64(\_\_m128i s, \_\_mmask8 k, \_\_m128i a, \_\_m128i b);

VPANDNQ \_\_m128i \_\_mm\_maskz\_andnot\_epi64(\_\_mmask8 k, \_\_m128i a, \_\_m128i b);

PANDN: \_\_m64 \_\_mm\_andnot\_si64(\_\_m64 m1, \_\_m64 m2)

(V)PANDN: \_\_m128i \_\_mm\_andnot\_si128(\_\_m128i a, \_\_m128i b)

VPANDN: \_\_m256i \_\_mm256\_andnot\_si256(\_\_m256i a, \_\_m256i b)

**Flags Affected**

None.

**Numeric Exceptions**

None.

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.

## PAVGB/PAVGW—Average Packed Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F E0 /r <sup>1</sup> PAVGB <i>mm1, mm2/m64</i>	RM	V/V	SSE	Average packed unsigned byte integers from <i>mm2/m64</i> and <i>mm1</i> with rounding.
66 0F E0, /r PAVGB <i>xmm1, xmm2/m128</i>	RM	V/V	SSE2	Average packed unsigned byte integers from <i>xmm2/m128</i> and <i>xmm1</i> with rounding.
0F E3 /r <sup>1</sup> PAVGW <i>mm1, mm2/m64</i>	RM	V/V	SSE	Average packed unsigned word integers from <i>mm2/m64</i> and <i>mm1</i> with rounding.
66 0F E3 /r PAVGW <i>xmm1, xmm2/m128</i>	RM	V/V	SSE2	Average packed unsigned word integers from <i>xmm2/m128</i> and <i>xmm1</i> with rounding.
VEEX.NDS.128.66.0F.WIG E0 /r VPAVGB <i>xmm1, xmm2, xmm3/m128</i>	RVM	V/V	AVX	Average packed unsigned byte integers from <i>xmm3/m128</i> and <i>xmm2</i> with rounding.
VEEX.NDS.128.66.0F.WIG E3 /r VPAVGW <i>xmm1, xmm2, xmm3/m128</i>	RVM	V/V	AVX	Average packed unsigned word integers from <i>xmm3/m128</i> and <i>xmm2</i> with rounding.
VEEX.NDS.256.66.0F.WIG E0 /r VPAVGB <i>ymm1, ymm2, ymm3/m256</i>	RVM	V/V	AVX2	Average packed unsigned byte integers from <i>ymm2</i> , and <i>ymm3/m256</i> with rounding and store to <i>ymm1</i> .
VEEX.NDS.256.66.0F.WIG E3 /r VPAVGW <i>ymm1, ymm2, ymm3/m256</i>	RVM	V/V	AVX2	Average packed unsigned word integers from <i>ymm2</i> , <i>ymm3/m256</i> with rounding to <i>ymm1</i> .
EVEX.NDS.128.66.0F.WIG E0 /r VPAVGB <i>xmm1 {k1}{z}, xmm2, xmm3/m128</i>	FVM	V/V	AVX512VL AVX512BW	Average packed unsigned byte integers from <i>xmm2</i> , and <i>xmm3/m128</i> with rounding and store to <i>xmm1</i> under writemask <i>k1</i> .
EVEX.NDS.256.66.0F.WIG E0 /r VPAVGB <i>ymm1 {k1}{z}, ymm2, ymm3/m256</i>	FVM	V/V	AVX512VL AVX512BW	Average packed unsigned byte integers from <i>ymm2</i> , and <i>ymm3/m256</i> with rounding and store to <i>ymm1</i> under writemask <i>k1</i> .
EVEX.NDS.512.66.0F.WIG E0 /r VPAVGB <i>zmm1 {k1}{z}, zmm2, zmm3/m512</i>	FVM	V/V	AVX512BW	Average packed unsigned byte integers from <i>zmm2</i> , and <i>zmm3/m512</i> with rounding and store to <i>zmm1</i> under writemask <i>k1</i> .
EVEX.NDS.128.66.0F.WIG E3 /r VPAVGW <i>xmm1 {k1}{z}, xmm2, xmm3/m128</i>	FVM	V/V	AVX512VL AVX512BW	Average packed unsigned word integers from <i>xmm2</i> , <i>xmm3/m128</i> with rounding to <i>xmm1</i> under writemask <i>k1</i> .
EVEX.NDS.256.66.0F.WIG E3 /r VPAVGW <i>ymm1 {k1}{z}, ymm2, ymm3/m256</i>	FVM	V/V	AVX512VL AVX512BW	Average packed unsigned word integers from <i>ymm2</i> , <i>ymm3/m256</i> with rounding to <i>ymm1</i> under writemask <i>k1</i> .
EVEX.NDS.512.66.0F.WIG E3 /r VPAVGW <i>zmm1 {k1}{z}, zmm2, zmm3/m512</i>	FVM	V/V	AVX512BW	Average packed unsigned word integers from <i>zmm2</i> , <i>zmm3/m512</i> with rounding to <i>zmm1</i> under writemask <i>k1</i> .

## NOTES:

1. See note in Section 2.4, “AVX and SSE Instruction Exception Specification” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

## Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FVM	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

## Description

Performs a SIMD average of the packed unsigned integers from the source operand (second operand) and the destination operand (first operand), and stores the results in the destination operand. For each corresponding pair of data elements in the first and second operands, the elements are added together, a 1 is added to the temporary sum, and that result is shifted right one bit position.

The (V)PAVGB instruction operates on packed unsigned bytes and the (V)PAVGW instruction operates on packed unsigned words.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE instructions: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand can be an MMX technology register.

128-bit Legacy SSE version: The first source operand is an XMM register. The second operand can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAX\_VL-1:128) of the corresponding register destination are unmodified.

EVEX.512 encoded version: The first source operand is a ZMM register. The second source operand is a ZMM register or a 512-bit memory location. The destination operand is a ZMM register.

VEX.256 and EVEX.256 encoded versions: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register.

VEX.128 and EVEX.128 encoded versions: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAX\_VL-1:128) of the corresponding register destination are zeroed.

## Operation

**PAVGB (with 64-bit operands)**

$$\text{DEST}[7:0] \leftarrow (\text{SRC}[7:0] + \text{DEST}[7:0] + 1) \gg 1; \text{ (* Temp sum before shifting is 9 bits *)}$$

(\* Repeat operation performed for bytes 2 through 6 \*)

$$\text{DEST}[63:56] \leftarrow (\text{SRC}[63:56] + \text{DEST}[63:56] + 1) \gg 1;$$
**PAVGW (with 64-bit operands)**

$$\text{DEST}[15:0] \leftarrow (\text{SRC}[15:0] + \text{DEST}[15:0] + 1) \gg 1; \text{ (* Temp sum before shifting is 17 bits *)}$$

(\* Repeat operation performed for words 2 and 3 \*)

$$\text{DEST}[63:48] \leftarrow (\text{SRC}[63:48] + \text{DEST}[63:48] + 1) \gg 1;$$
**PAVGB (with 128-bit operands)**

$$\text{DEST}[7:0] \leftarrow (\text{SRC}[7:0] + \text{DEST}[7:0] + 1) \gg 1; \text{ (* Temp sum before shifting is 9 bits *)}$$

(\* Repeat operation performed for bytes 2 through 14 \*)

$$\text{DEST}[127:120] \leftarrow (\text{SRC}[127:120] + \text{DEST}[127:120] + 1) \gg 1;$$
**PAVGW (with 128-bit operands)**

$$\text{DEST}[15:0] \leftarrow (\text{SRC}[15:0] + \text{DEST}[15:0] + 1) \gg 1; \text{ (* Temp sum before shifting is 17 bits *)}$$

(\* Repeat operation performed for words 2 through 6 \*)

$$\text{DEST}[127:112] \leftarrow (\text{SRC}[127:112] + \text{DEST}[127:112] + 1) \gg 1;$$

**VPAVGB (VEX.128 encoded version)**

```

DEST[7:0] ← (SRC1[7:0] + SRC2[7:0] + 1) >> 1;
(* Repeat operation performed for bytes 2 through 15 *)
DEST[127:120] ← (SRC1[127:120] + SRC2[127:120] + 1) >> 1
DEST[VLMAX-1:128] ← 0

```

**VPAVGW (VEX.128 encoded version)**

```

DEST[15:0] ← (SRC1[15:0] + SRC2[15:0] + 1) >> 1;
(* Repeat operation performed for 16-bit words 2 through 7 *)
DEST[127:112] ← (SRC1[127:112] + SRC2[127:112] + 1) >> 1
DEST[VLMAX-1:128] ← 0

```

**VPAVGB (VEX.256 encoded instruction)**

```

DEST[7:0] ← (SRC1[7:0] + SRC2[7:0] + 1) >> 1; (* Temp sum before shifting is 9 bits *)
(* Repeat operation performed for bytes 2 through 31 *)
DEST[255:248] ← (SRC1[255:248] + SRC2[255:248] + 1) >> 1;

```

**VPAVGW (VEX.256 encoded instruction)**

```

DEST[15:0] ← (SRC1[15:0] + SRC2[15:0] + 1) >> 1; (* Temp sum before shifting is 17 bits *)
(* Repeat operation performed for words 2 through 15 *)
DEST[255:14] ← (SRC1[255:240] + SRC2[255:240] + 1) >> 1;

```

**VPAVGB (EVEX encoded versions)**

(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] ← (SRC1[i+7:i] + SRC2[i+7:i] + 1) >> 1; (\* Temp sum before shifting is 9 bits \*)

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+7:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[i+7:i] = 0

FI

FI;

ENDFOR;

DEST[MAX\_VL-1:VL] ← 0

**VPAVGW (EVEX encoded versions)**

(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] ← (SRC1[i+15:i] + SRC2[i+15:i] + 1) >> 1  
; (\* Temp sum before shifting is 17 bits \*)

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+15:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[i+15:i] = 0

FI

FI;

ENDFOR;

DEST[MAX\_VL-1:VL] ← 0



**Intel C/C++ Compiler Intrinsic Equivalents**

```

VPAVGB __m512i _mm512_avg_epu8( __m512i a, __m512i b);
VPAVGW __m512i _mm512_avg_epu16( __m512i a, __m512i b);
VPAVGB __m512i _mm512_mask_avg_epu8(__m512i s, __mmask64 m, __m512i a, __m512i b);
VPAVGW __m512i _mm512_mask_avg_epu16(__m512i s, __mmask32 m, __m512i a, __m512i b);
VPAVGB __m512i _mm512_maskz_avg_epu8( __mmask64 m, __m512i a, __m512i b);
VPAVGW __m512i _mm512_maskz_avg_epu16( __mmask32 m, __m512i a, __m512i b);
VPAVGB __m256i _mm256_mask_avg_epu8(__m256i s, __mmask32 m, __m256i a, __m256i b);
VPAVGW __m256i _mm256_mask_avg_epu16(__m256i s, __mmask16 m, __m256i a, __m256i b);
VPAVGB __m256i _mm256_maskz_avg_epu8( __mmask32 m, __m256i a, __m256i b);
VPAVGW __m256i _mm256_maskz_avg_epu16( __mmask16 m, __m256i a, __m256i b);
VPAVGB __m128i _mm_mask_avg_epu8(__m128i s, __mmask16 m, __m128i a, __m128i b);
VPAVGW __m128i _mm_mask_avg_epu16(__m128i s, __mmask8 m, __m128i a, __m128i b);
VPAVGB __m128i _mm_maskz_avg_epu8( __mmask16 m, __m128i a, __m128i b);
VPAVGW __m128i _mm_maskz_avg_epu16( __mmask8 m, __m128i a, __m128i b);
PAVGB: __m64 _mm_avg_pu8 ( __m64 a, __m64 b)
PAVGW: __m64 _mm_avg_pu16 ( __m64 a, __m64 b)
(V)PAVGB: __m128i _mm_avg_epu8 ( __m128i a, __m128i b)
(V)PAVGW: __m128i _mm_avg_epu16 ( __m128i a, __m128i b)
VPAVGB:      __m256i _mm256_avg_epu8 ( __m256i a, __m256i b)
VPAVGW:      __m256i _mm256_avg_epu16 ( __m256i a, __m256i b)

```

**Flags Affected**

None.

**Numeric Exceptions**

None.

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.nb.

## PCLMULQDQ - Carry-Less Multiplication Quadword

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 3A 44 /r ib PCLMULQDQ <i>xmm1, xmm2/m128, imm8</i>	RMI	V/V	PCLMUL- QDQ	Carry-less multiplication of one quadword of <i>xmm1</i> by one quadword of <i>xmm2/m128</i> , stores the 128-bit result in <i>xmm1</i> . The immediate is used to determine which quadwords of <i>xmm1</i> and <i>xmm2/m128</i> should be used.
VEX.NDS.128.66.0F3A.WIG 44 /r ib VPCLMULQDQ <i>xmm1, xmm2, xmm3/m128, imm8</i>	RVMI	V/V	Both PCL- MULQDQ and AVX flags	Carry-less multiplication of one quadword of <i>xmm2</i> by one quadword of <i>xmm3/m128</i> , stores the 128-bit result in <i>xmm1</i> . The immediate is used to determine which quadwords of <i>xmm2</i> and <i>xmm3/m128</i> should be used.

### Instruction Operand Encoding

Op/En	Operand 1	Operand2	Operand3	Operand4
RMI	ModRM:reg (r, w)	ModRM:r/m (r)	imm8	NA
RVMI	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8

### Description

Performs a carry-less multiplication of two quadwords, selected from the first source and second source operand according to the value of the immediate byte. Bits 4 and 0 are used to select which 64-bit half of each operand to use according to Table 4-9, other bits of the immediate byte are ignored.

**Table 4-9. PCLMULQDQ Quadword Selection of Immediate Byte**

Imm[4]	Imm[0]	PCLMULQDQ Operation
0	0	CL_MUL( SRC2 <sup>1</sup> [63:0], SRC1[63:0] )
0	1	CL_MUL( SRC2[63:0], SRC1[127:64] )
1	0	CL_MUL( SRC2[127:64], SRC1[63:0] )
1	1	CL_MUL( SRC2[127:64], SRC1[127:64] )

### NOTES:

1. SRC2 denotes the second source operand, which can be a register or memory; SRC1 denotes the first source and destination operand.

The first source operand and the destination operand are the same and must be an XMM register. The second source operand can be an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

Compilers and assemblers may implement the following pseudo-op syntax to simplify programming and emit the required encoding for Imm8.

**Table 4-10. Pseudo-Op and PCLMULQDQ Implementation**

Pseudo-Op	Imm8 Encoding
PCLMULLQLQDQ <i>xmm1, xmm2</i>	0000_0000B
PCLMULHQLQDQ <i>xmm1, xmm2</i>	0000_0001B
PCLMULLQHQQDQ <i>xmm1, xmm2</i>	0001_0000B
PCLMULHQHQQDQ <i>xmm1, xmm2</i>	0001_0001B

**Operation****PCLMULQDQ**

```

IF (Imm8[0] = 0 )
    THEN
        TEMP1 ← SRC1 [63:0];
    ELSE
        TEMP1 ← SRC1 [127:64];
FI
IF (Imm8[4] = 0 )
    THEN
        TEMP2 ← SRC2 [63:0];
    ELSE
        TEMP2 ← SRC2 [127:64];
FI
For i = 0 to 63 {
    TmpB [ i ] ← (TEMP1[ 0 ] and TEMP2[ i ]);
    For j = 1 to i {
        TmpB [ i ] ← TmpB [ i ] xor (TEMP1[ j ] and TEMP2[ i - j ])
    }
    DEST[ i ] ← TmpB[ i ];
}
For i = 64 to 126 {
    TmpB [ i ] ← 0;
    For j = i - 63 to 63 {
        TmpB [ i ] ← TmpB [ i ] xor (TEMP1[ j ] and TEMP2[ i - j ])
    }
    DEST[ i ] ← TmpB[ i ];
}
DEST[127] ← 0;
DEST[VLMAX-1:128] (Unmodified)

```

**VPCLMULQDQ**

```

IF (Imm8[0] = 0 )
    THEN
        TEMP1 ← SRC1 [63:0];
    ELSE
        TEMP1 ← SRC1 [127:64];
FI
IF (Imm8[4] = 0 )
    THEN
        TEMP2 ← SRC2 [63:0];
    ELSE
        TEMP2 ← SRC2 [127:64];
FI
For i = 0 to 63 {
    TmpB [ i ] ← (TEMP1[ 0 ] and TEMP2[ i ]);
    For j = 1 to i {
        TmpB [i] ← TmpB [i] xor (TEMP1[ j ] and TEMP2[ i - j ])
    }
    DEST[i] ← TmpB[i];
}
For i = 64 to 126 {
    TmpB [ i ] ← 0;
    For j = i - 63 to 63 {

```

```

    TmpB [i] ← TmpB [i] xor (TEMP1[j] and TEMP2[ i - j ])
  }
  DEST[i] ← TmpB[i];
}
DEST[VLMAX-1:127] ← 0;

```

### Intel C/C++ Compiler Intrinsic Equivalent

(V)PCLMULQDQ: `__m128i _mm_clmulepi64_si128 (__m128i, __m128i, const int)`

### SIMD Floating-Point Exceptions

None.

### Other Exceptions

See Exceptions Type 4, additionally

#UD If VEX.L = 1.

## PCMPEQB/PCMPEQW/PCMPEQD— Compare Packed Data for Equal

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
OF 74 /r <sup>1</sup> PCMPEQB <i>mm</i> , <i>mm/m64</i>	RM	V/V	MMX	Compare packed bytes in <i>mm/m64</i> and <i>mm</i> for equality.
66 OF 74 /r PCMPEQB <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE2	Compare packed bytes in <i>xmm2/m128</i> and <i>xmm1</i> for equality.
OF 75 /r <sup>1</sup> PCMPEQW <i>mm</i> , <i>mm/m64</i>	RM	V/V	MMX	Compare packed words in <i>mm/m64</i> and <i>mm</i> for equality.
66 OF 75 /r PCMPEQW <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE2	Compare packed words in <i>xmm2/m128</i> and <i>xmm1</i> for equality.
OF 76 /r <sup>1</sup> PCMPEQD <i>mm</i> , <i>mm/m64</i>	RM	V/V	MMX	Compare packed doublewords in <i>mm/m64</i> and <i>mm</i> for equality.
66 OF 76 /r PCMPEQD <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE2	Compare packed doublewords in <i>xmm2/m128</i> and <i>xmm1</i> for equality.
VEX.NDS.128.66.0F.WIG 74 /r VPCMPEQB <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Compare packed bytes in <i>xmm3/m128</i> and <i>xmm2</i> for equality.
VEX.NDS.128.66.0F.WIG 75 /r VPCMPEQW <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Compare packed words in <i>xmm3/m128</i> and <i>xmm2</i> for equality.
VEX.NDS.128.66.0F.WIG 76 /r VPCMPEQD <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Compare packed doublewords in <i>xmm3/m128</i> and <i>xmm2</i> for equality.
VEX.NDS.256.66.0F.WIG 74 /r VPCMPEQB <i>ymm1</i> , <i>ymm2</i> , <i>ymm3 /m256</i>	RVM	V/V	AVX2	Compare packed bytes in <i>ymm3/m256</i> and <i>ymm2</i> for equality.
VEX.NDS.256.66.0F.WIG 75 /r VPCMPEQW <i>ymm1</i> , <i>ymm2</i> , <i>ymm3 /m256</i>	RVM	V/V	AVX2	Compare packed words in <i>ymm3/m256</i> and <i>ymm2</i> for equality.
VEX.NDS.256.66.0F.WIG 76 /r VPCMPEQD <i>ymm1</i> , <i>ymm2</i> , <i>ymm3 /m256</i>	RVM	V/V	AVX2	Compare packed doublewords in <i>ymm3/m256</i> and <i>ymm2</i> for equality.
EVEX.NDS.128.66.0F.W0 76 /r VPCMPEQD <i>k1 {k2}</i> , <i>xmm2</i> , <i>xmm3/m128/m32bcst</i>	FV	V/V	AVX512V L AVX512F	Compare Equal between int32 vector <i>xmm2</i> and int32 vector <i>xmm3/m128/m32bcst</i> , and set vector mask <i>k1</i> to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.NDS.256.66.0F.W0 76 /r VPCMPEQD <i>k1 {k2}</i> , <i>ymm2</i> , <i>ymm3/m256/m32bcst</i>	FV	V/V	AVX512V L AVX512F	Compare Equal between int32 vector <i>ymm2</i> and int32 vector <i>ymm3/m256/m32bcst</i> , and set vector mask <i>k1</i> to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.NDS.512.66.0F.W0 76 /r VPCMPEQD <i>k1 {k2}</i> , <i>zmm2</i> , <i>zmm3/m512/m32bcst</i>	FV	V/V	AVX512F	Compare Equal between int32 vectors in <i>zmm2</i> and <i>zmm3/m512/m32bcst</i> , and set destination <i>k1</i> according to the comparison results under writemask <i>k2</i> ,
EVEX.NDS.128.66.0F.WIG 74 /r VPCMPEQB <i>k1 {k2}</i> , <i>xmm2</i> , <i>xmm3 /m128</i>	FVM	V/V	AVX512V L AVX512B W	Compare packed bytes in <i>xmm3/m128</i> and <i>xmm2</i> for equality and set vector mask <i>k1</i> to reflect the zero/nonzero status of each element of the result, under writemask.

EVEX.NDS.256.66.0F.WIG 74 /r VPCMPEQB k1 {k2}, ymm2, ymm3 /m256	FVM V/V	AVX512V L AVX512B W	Compare packed bytes in ymm3/m256 and ymm2 for equality and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.NDS.512.66.0F.WIG 74 /r VPCMPEQB k1 {k2}, zmm2, zmm3 /m512	FVM V/V	AVX512B W	Compare packed bytes in zmm3/m512 and zmm2 for equality and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.NDS.128.66.0F.WIG 75 /r VPCMPEQW k1 {k2}, xmm2, xmm3 /m128	FVM V/V	AVX512V L AVX512B W	Compare packed words in xmm3/m128 and xmm2 for equality and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.NDS.256.66.0F.WIG 75 /r VPCMPEQW k1 {k2}, ymm2, ymm3 /m256	FVM V/V	AVX512V L AVX512B W	Compare packed words in ymm3/m256 and ymm2 for equality and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.NDS.512.66.0F.WIG 75 /r VPCMPEQW k1 {k2}, zmm2, zmm3 /m512	FVM V/V	AVX512B W	Compare packed words in zmm3/m512 and zmm2 for equality and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.

**NOTES:**

1. See note in Section 2.4, “AVX and SSE Instruction Exception Specification” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FV	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA
FVM	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

**Description**

Performs a SIMD compare for equality of the packed bytes, words, or doublewords in the destination operand (first operand) and the source operand (second operand). If a pair of data elements is equal, the corresponding data element in the destination operand is set to all 1s; otherwise, it is set to all 0s.

The (V)PCMPEQB instruction compares the corresponding bytes in the destination and source operands; the (V)PCMPEQW instruction compares the corresponding words in the destination and source operands; and the (V)PCMPEQD instruction compares the corresponding doublewords in the destination and source operands.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE instructions: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand can be an MMX technology register.

128-bit Legacy SSE version: The second source operand can be an XMM register or a 128-bit memory location. The first source and destination operands are XMM registers. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The second source operand can be an XMM register or a 128-bit memory location. The first source and destination operands are XMM registers. Bits (VLMAX-1:128) of the corresponding YMM register are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register.

EVEX encoded VPCMPEQD: The first source operand (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 32-bit memory location. The destination operand (first operand) is a mask register updated according to the writemask k2.

EVEX encoded VPCMPEQB/W: The first source operand (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. The destination operand (first operand) is a mask register updated according to the writemask k2.

## Operation

### PCMPEQB (with 64-bit operands)

```
IF DEST[7:0] = SRC[7:0]
  THEN DEST[7:0] ← FFH;
  ELSE DEST[7:0] ← 0; FI;
```

(\* Continue comparison of 2nd through 7th bytes in DEST and SRC \*)

```
IF DEST[63:56] = SRC[63:56]
  THEN DEST[63:56] ← FFH;
  ELSE DEST[63:56] ← 0; FI;
```

### COMPARE\_BYTES\_EQUAL (SRC1, SRC2)

```
IF SRC1[7:0] = SRC2[7:0]
  THEN DEST[7:0] ← FFH;
  ELSE DEST[7:0] ← 0; FI;
```

(\* Continue comparison of 2nd through 15th bytes in SRC1 and SRC2 \*)

```
IF SRC1[127:120] = SRC2[127:120]
  THEN DEST[127:120] ← FFH;
  ELSE DEST[127:120] ← 0; FI;
```

### COMPARE\_WORDS\_EQUAL (SRC1, SRC2)

```
IF SRC1[15:0] = SRC2[15:0]
  THEN DEST[15:0] ← FFFFH;
  ELSE DEST[15:0] ← 0; FI;
```

(\* Continue comparison of 2nd through 7th 16-bit words in SRC1 and SRC2 \*)

```
IF SRC1[127:112] = SRC2[127:112]
  THEN DEST[127:112] ← FFFFH;
  ELSE DEST[127:112] ← 0; FI;
```

### COMPARE\_DWORDS\_EQUAL (SRC1, SRC2)

```
IF SRC1[31:0] = SRC2[31:0]
  THEN DEST[31:0] ← FFFFFFFFH;
  ELSE DEST[31:0] ← 0; FI;
```

(\* Continue comparison of 2nd through 3rd 32-bit dwords in SRC1 and SRC2 \*)

```
IF SRC1[127:96] = SRC2[127:96]
  THEN DEST[127:96] ← FFFFFFFFH;
  ELSE DEST[127:96] ← 0; FI;
```

### PCMPEQB (with 128-bit operands)

```
DEST[127:0] ← COMPARE_BYTES_EQUAL(DEST[127:0], SRC[127:0])
DEST[MAX_VL-1:128] (Unmodified)
```

### VPCMPEQB (VEX.128 encoded version)

```
DEST[127:0] ← COMPARE_BYTES_EQUAL(SRC1[127:0], SRC2[127:0])
```

DEST[VLMAX-1:128] ← 0

#### VPCMPEQB (VEX.256 encoded version)

DEST[127:0] ← COMPARE\_BYTES\_EQUAL(SRC1[127:0],SRC2[127:0])  
 DEST[255:128] ← COMPARE\_BYTES\_EQUAL(SRC1[255:128],SRC2[255:128])  
 DEST[VLMAX-1:256] ← 0

#### VPCMPEQB (EVEX encoded versions)

(KL, VL) = (16, 128), (32, 256), (64, 512)

```
FOR j ← 0 TO KL-1
  i ← j * 8
  IF k2[j] OR *no writemask*
    THEN
      /* signed comparison */
      CMP ← SRC1[i+7:i] == SRC2[i+7:i];
      IF CMP = TRUE
        THEN DEST[j] ← 1;
        ELSE DEST[j] ← 0; FI;
      ELSE DEST[j] ← 0 ; zeroing-masking onlyFI;
    FI;
ENDFOR
DEST[MAX_KL-1:KL] ← 0
```

#### PCMPEQW (with 64-bit operands)

```
IF DEST[15:0] = SRC[15:0]
  THEN DEST[15:0] ← FFFFH;
  ELSE DEST[15:0] ← 0; FI;
(* Continue comparison of 2nd and 3rd words in DEST and SRC *)
IF DEST[63:48] = SRC[63:48]
  THEN DEST[63:48] ← FFFFH;
  ELSE DEST[63:48] ← 0; FI;
```

#### PCMPEQW (with 128-bit operands)

DEST[127:0] ← COMPARE\_WORDS\_EQUAL(DEST[127:0],SRC[127:0])  
 DEST[MAX\_VL-1:128] (Unmodified)

#### VPCMPEQW (VEX.128 encoded version)

DEST[127:0] ← COMPARE\_WORDS\_EQUAL(SRC1[127:0],SRC2[127:0])  
 DEST[VLMAX-1:128] ← 0

#### VPCMPEQW (VEX.256 encoded version)

DEST[127:0] ← COMPARE\_WORDS\_EQUAL(SRC1[127:0],SRC2[127:0])  
 DEST[255:128] ← COMPARE\_WORDS\_EQUAL(SRC1[255:128],SRC2[255:128])  
 DEST[VLMAX-1:256] ← 0

#### VPCMPEQW (EVEX encoded versions)

(KL, VL) = (8, 128), (16, 256), (32, 512)

```
FOR j ← 0 TO KL-1
  i ← j * 16
  IF k2[j] OR *no writemask*
    THEN
      /* signed comparison */
      CMP ← SRC1[i+15:i] == SRC2[i+15:i];
      IF CMP = TRUE
```



```

        THEN DEST[j] ← 1;
        ELSE DEST[j] ← 0; FI;
ELSE    DEST[j] ← 0          ; zeroing-masking onlyFI;
FI;
ENDFOR
DEST[MAX_KL-1:KL] ← 0

```

**PCMPEQD (with 64-bit operands)**

```

IF DEST[31:0] = SRC[31:0]
    THEN DEST[31:0] ← FFFFFFFFH;
    ELSE DEST[31:0] ← 0; FI;
IF DEST[63:32] = SRC[63:32]
    THEN DEST[63:32] ← FFFFFFFFH;
    ELSE DEST[63:32] ← 0; FI;

```

**PCMPEQD (with 128-bit operands)**

```

DEST[127:0] ← COMPARE_DWORDS_EQUAL(DEST[127:0],SRC[127:0])
DEST[MAX_VL-1:128] (Unmodified)

```

**VPCMPEQD (VEX.128 encoded version)**

```

DEST[127:0] ← COMPARE_DWORDS_EQUAL(SRC1[127:0],SRC2[127:0])
DEST[VLMAX-1:128] ← 0

```

**VPCMPEQD (VEX.256 encoded version)**

```

DEST[127:0] ← COMPARE_DWORDS_EQUAL(SRC1[127:0],SRC2[127:0])
DEST[255:128] ← COMPARE_DWORDS_EQUAL(SRC1[255:128],SRC2[255:128])
DEST[VLMAX-1:256] ← 0

```

**VPCMPEQD (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

```

FOR j ← 0 TO KL-1
    i ← j * 32
    IF k2[j] OR *no writemask*
        THEN
            /* signed comparison */
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN CMP ← SRC1[i+31:i] = SRC2[31:0];
                ELSE CMP ← SRC1[i+31:i] = SRC2[i+31:i];
            FI;
            IF CMP = TRUE
                THEN DEST[j] ← 1;
                ELSE DEST[j] ← 0; FI;
        ELSE    DEST[j] ← 0          ; zeroing-masking only
    FI;
ENDFOR
DEST[MAX_KL-1:KL] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalents**

```

VPCMPEQB __mmask64 __mm512_cmpeq_epi8_mask(__m512i a, __m512i b);
VPCMPEQB __mmask64 __mm512_mask_cmpeq_epi8_mask(__mmask64 k, __m512i a, __m512i b);
VPCMPEQB __mmask32 __m256_cmpeq_epi8_mask(__m256i a, __m256i b);
VPCMPEQB __mmask32 __m256_mask_cmpeq_epi8_mask(__mmask32 k, __m256i a, __m256i b);
VPCMPEQB __mmask16 __m128_cmpeq_epi8_mask(__m128i a, __m128i b);
VPCMPEQB __mmask16 __m128_mask_cmpeq_epi8_mask(__mmask16 k, __m128i a, __m128i b);

```

VPCMPEQW \_\_mmask32 \_\_mm512\_cmpeq\_epi16\_mask(\_\_m512i a, \_\_m512i b);  
 VPCMPEQW \_\_mmask32 \_\_mm512\_mask\_cmpeq\_epi16\_mask(\_\_mmask32 k, \_\_m512i a, \_\_m512i b);  
 VPCMPEQW \_\_mmask16 \_\_mm256\_cmpeq\_epi16\_mask(\_\_m256i a, \_\_m256i b);  
 VPCMPEQW \_\_mmask16 \_\_mm256\_mask\_cmpeq\_epi16\_mask(\_\_mmask16 k, \_\_m256i a, \_\_m256i b);  
 VPCMPEQW \_\_mmask8 \_\_mm\_cmpeq\_epi16\_mask(\_\_m128i a, \_\_m128i b);  
 VPCMPEQW \_\_mmask8 \_\_mm\_mask\_cmpeq\_epi16\_mask(\_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
 VPCMPEQD \_\_mmask16 \_\_mm512\_cmpeq\_epi32\_mask(\_\_m512i a, \_\_m512i b);  
 VPCMPEQD \_\_mmask16 \_\_mm512\_mask\_cmpeq\_epi32\_mask(\_\_mmask16 k, \_\_m512i a, \_\_m512i b);  
 VPCMPEQD \_\_mmask8 \_\_mm256\_cmpeq\_epi32\_mask(\_\_m256i a, \_\_m256i b);  
 VPCMPEQD \_\_mmask8 \_\_mm256\_mask\_cmpeq\_epi32\_mask(\_\_mmask8 k, \_\_m256i a, \_\_m256i b);  
 VPCMPEQD \_\_mmask8 \_\_mm\_cmpeq\_epi32\_mask(\_\_m128i a, \_\_m128i b);  
 VPCMPEQD \_\_mmask8 \_\_mm\_mask\_cmpeq\_epi32\_mask(\_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
 PCMPEQB: \_\_m64 \_\_mm\_cmpeq\_pi8 (\_\_m64 m1, \_\_m64 m2)  
 PCMPEQW: \_\_m64 \_\_mm\_cmpeq\_pi16 (\_\_m64 m1, \_\_m64 m2)  
 PCMPEQD: \_\_m64 \_\_mm\_cmpeq\_pi32 (\_\_m64 m1, \_\_m64 m2)  
 (V)PCMPEQB: \_\_m128i \_\_mm\_cmpeq\_epi8 (\_\_m128i a, \_\_m128i b)  
 (V)PCMPEQW: \_\_m128i \_\_mm\_cmpeq\_epi16 (\_\_m128i a, \_\_m128i b)  
 (V)PCMPEQD: \_\_m128i \_\_mm\_cmpeq\_epi32 (\_\_m128i a, \_\_m128i b)  
 VPCMPEQB: \_\_m256i \_\_mm256\_cmpeq\_epi8 (\_\_m256i a, \_\_m256i b)  
 VPCMPEQW: \_\_m256i \_\_mm256\_cmpeq\_epi16 (\_\_m256i a, \_\_m256i b)  
 VPCMPEQD: \_\_m256i \_\_mm256\_cmpeq\_epi32 (\_\_m256i a, \_\_m256i b)

### Flags Affected

None.

### SIMD Floating-Point Exceptions

None.

### Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded VPCMPEQD, see Exceptions Type E4.

EVEX-encoded VPCMPEQB/W, see Exceptions Type E4.nb.

## PCMPEQQ — Compare Packed Qword Data for Equal

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 29 /r PCMPEQQ <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE4_1	Compare packed qwords in <i>xmm2/m128</i> and <i>xmm1</i> for equality.
VEX.NDS.128.66.0F38.WIG 29 /r VPCMPEQQ <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Compare packed quadwords in <i>xmm3/m128</i> and <i>xmm2</i> for equality.
VEX.NDS.256.66.0F38.WIG 29 /r VPCMPEQQ <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	RVM	V/V	AVX2	Compare packed quadwords in <i>ymm3/m256</i> and <i>ymm2</i> for equality.
EVEX.NDS.128.66.0F38.W1 29 /r VPCMPEQQ <i>k1</i> { <i>k2</i> }, <i>xmm2</i> , <i>xmm3/m128/m64bcst</i>	FV	V/V	AVX512VL AVX512F	Compare Equal between int64 vector <i>xmm2</i> and int64 vector <i>xmm3/m128/m64bcst</i> , and set vector mask <i>k1</i> to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.NDS.256.66.0F38.W1 29 /r VPCMPEQQ <i>k1</i> { <i>k2</i> }, <i>ymm2</i> , <i>ymm3/m256/m64bcst</i>	FV	V/V	AVX512VL AVX512F	Compare Equal between int64 vector <i>ymm2</i> and int64 vector <i>ymm3/m256/m64bcst</i> , and set vector mask <i>k1</i> to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.NDS.512.66.0F38.W1 29 /r VPCMPEQQ <i>k1</i> { <i>k2</i> }, <i>zmm2</i> , <i>zmm3/m512/m64bcst</i>	FV	V/V	AVX512F	Compare Equal between int64 vector <i>zmm2</i> and int64 vector <i>zmm3/m512/m64bcst</i> , and set vector mask <i>k1</i> to reflect the zero/nonzero status of each element of the result, under writemask.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg ( <i>r</i> , <i>w</i> )	ModRM:r/m ( <i>r</i> )	NA	NA
RVM	ModRM:reg ( <i>w</i> )	VEX.vvvv ( <i>r</i> )	ModRM:r/m ( <i>r</i> )	NA
FV	ModRM:reg ( <i>w</i> )	EVEX.vvvv ( <i>r</i> )	ModRM:r/m ( <i>r</i> )	NA

### Description

Performs an SIMD compare for equality of the packed quadwords in the destination operand (first operand) and the source operand (second operand). If a pair of data elements is equal, the corresponding data element in the destination is set to all 1s; otherwise, it is set to 0s.

128-bit Legacy SSE version: The second source operand can be an XMM register or a 128-bit memory location. The first source and destination operands are XMM registers. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The second source operand can be an XMM register or a 128-bit memory location. The first source and destination operands are XMM registers. Bits (VLMAX-1:128) of the corresponding YMM register are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register.

EVEX encoded VPCMPEQQ: The first source operand (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 64-bit memory location. The destination operand (first operand) is a mask register updated according to the writemask *k2*.

## Operation

### PCMPEQQ (with 128-bit operands)

```

IF (DEST[63:0] = SRC[63:0])
    THEN DEST[63:0] ← FFFFFFFFFFFFFFFFH;
    ELSE DEST[63:0] ← 0; FI;
IF (DEST[127:64] = SRC[127:64])
    THEN DEST[127:64] ← FFFFFFFFFFFFFFFFH;
    ELSE DEST[127:64] ← 0; FI;
DEST[MAX_VL-1:128] (Unmodified)

```

### COMPARE\_QWORDS\_EQUAL (SRC1, SRC2)

```

IF SRC1[63:0] = SRC2[63:0]
    THEN DEST[63:0] ← FFFFFFFFFFFFFFFFH;
    ELSE DEST[63:0] ← 0; FI;
IF SRC1[127:64] = SRC2[127:64]
    THEN DEST[127:64] ← FFFFFFFFFFFFFFFFH;
    ELSE DEST[127:64] ← 0; FI;

```

### VPCMPEQQ (VEX.128 encoded version)

```

DEST[127:0] ← COMPARE_QWORDS_EQUAL(SRC1, SRC2)
DEST[VLMAX-1:128] ← 0

```

### VPCMPEQQ (VEX.256 encoded version)

```

DEST[127:0] ← COMPARE_QWORDS_EQUAL(SRC1[127:0], SRC2[127:0])
DEST[255:128] ← COMPARE_QWORDS_EQUAL(SRC1[255:128], SRC2[255:128])
DEST[VLMAX-1:256] ← 0

```

### VPCMPEQQ (EVEX encoded versions)

(KL, VL) = (2, 128), (4, 256), (8, 512)

```

FOR j ← 0 TO KL-1
    i ← j * 64
    IF k2[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN CMP ← SRC1[i+63:i] = SRC2[63:0];
                ELSE CMP ← SRC1[i+63:i] = SRC2[i+63:i];
            FI;
            IF CMP = TRUE
                THEN DEST[j] ← 1;
                ELSE DEST[j] ← 0; FI;
        ELSE DEST[j] ← 0 ; zeroing-masking only
    FI;
ENDFOR
DEST[MAX_KL-1:KL] ← 0

```

### Intel C/C++ Compiler Intrinsic Equivalent

```

VPCMPEQQ __mmask8 __mm512_cmpeq_epi64_mask( __m512i a, __m512i b);
VPCMPEQQ __mmask8 __mm512_mask_cmpeq_epi64_mask( __mmask8 k, __m512i a, __m512i b);
VPCMPEQQ __mmask8 __mm256_cmpeq_epi64_mask( __m256i a, __m256i b);
VPCMPEQQ __mmask8 __mm256_mask_cmpeq_epi64_mask( __mmask8 k, __m256i a, __m256i b);
VPCMPEQQ __mmask8 __mm_cmpeq_epi64_mask( __m128i a, __m128i b);
VPCMPEQQ __mmask8 __mm_mask_cmpeq_epi64_mask( __mmask8 k, __m128i a, __m128i b);
(V)PCMPEQQ: __m128i __mm_cmpeq_epi64( __m128i a, __m128i b);

```

VPCMPEQQ: `__m256i _mm256_cmpeq_epi64( __m256i a, __m256i b);`

### Flags Affected

None.

### SIMD Floating-Point Exceptions

None.

### Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded VPCMPEQQ, see Exceptions Type E4.

## PCMPGTB/PCMPGTW/PCMPGTD—Compare Packed Signed Integers for Greater Than

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F 64 /r <sup>1</sup> PCMPGTB <i>mm</i> , <i>mm/m64</i>	RM	V/V	MMX	Compare packed signed byte integers in <i>mm</i> and <i>mm/m64</i> for greater than.
66 0F 64 /r PCMPGTB <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE2	Compare packed signed byte integers in <i>xmm1</i> and <i>xmm2/m128</i> for greater than.
0F 65 /r <sup>1</sup> PCMPGTW <i>mm</i> , <i>mm/m64</i>	RM	V/V	MMX	Compare packed signed word integers in <i>mm</i> and <i>mm/m64</i> for greater than.
66 0F 65 /r PCMPGTW <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE2	Compare packed signed word integers in <i>xmm1</i> and <i>xmm2/m128</i> for greater than.
0F 66 /r <sup>1</sup> PCMPGTD <i>mm</i> , <i>mm/m64</i>	RM	V/V	MMX	Compare packed signed doubleword integers in <i>mm</i> and <i>mm/m64</i> for greater than.
66 0F 66 /r PCMPGTD <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE2	Compare packed signed doubleword integers in <i>xmm1</i> and <i>xmm2/m128</i> for greater than.
VEX.NDS.128.66.0F.WIG 64 /r VPCMPGTB <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Compare packed signed byte integers in <i>xmm2</i> and <i>xmm3/m128</i> for greater than.
VEX.NDS.128.66.0F.WIG 65 /r VPCMPGTW <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Compare packed signed word integers in <i>xmm2</i> and <i>xmm3/m128</i> for greater than.
VEX.NDS.128.66.0F.WIG 66 /r VPCMPGTD <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Compare packed signed doubleword integers in <i>xmm2</i> and <i>xmm3/m128</i> for greater than.
VEX.NDS.256.66.0F.WIG 64 /r VPCMPGTB <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	RVM	V/V	AVX2	Compare packed signed byte integers in <i>ymm2</i> and <i>ymm3/m256</i> for greater than.
VEX.NDS.256.66.0F.WIG 65 /r VPCMPGTW <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	RVM	V/V	AVX2	Compare packed signed word integers in <i>ymm2</i> and <i>ymm3/m256</i> for greater than.
VEX.NDS.256.66.0F.WIG 66 /r VPCMPGTD <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	RVM	V/V	AVX2	Compare packed signed doubleword integers in <i>ymm2</i> and <i>ymm3/m256</i> for greater than.
EVEX.NDS.128.66.0F.WO 66 /r VPCMPGTD <i>k1</i> { <i>k2</i> }, <i>xmm2</i> , <i>xmm3/m128/m32bcst</i>	FV	V/V	AVX512VL AVX512F	Compare Greater between int32 vector <i>xmm2</i> and int32 vector <i>xmm3/m128/m32bcst</i> , and set vector mask <i>k1</i> to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.NDS.256.66.0F.WO 66 /r VPCMPGTD <i>k1</i> { <i>k2</i> }, <i>ymm2</i> , <i>ymm3/m256/m32bcst</i>	FV	V/V	AVX512VL AVX512F	Compare Greater between int32 vector <i>ymm2</i> and int32 vector <i>ymm3/m256/m32bcst</i> , and set vector mask <i>k1</i> to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.NDS.512.66.0F.WO 66 /r VPCMPGTD <i>k1</i> { <i>k2</i> }, <i>zmm2</i> , <i>zmm3/m512/m32bcst</i>	FV	V/V	AVX512F	Compare Greater between int32 elements in <i>zmm2</i> and <i>zmm3/m512/m32bcst</i> , and set destination <i>k1</i> according to the comparison results under writemask. <i>k2</i> .
EVEX.NDS.128.66.0F.WIG 64 /r VPCMPGTB <i>k1</i> { <i>k2</i> }, <i>xmm2</i> , <i>xmm3/m128</i>	FVM	V/V	AVX512VL AVX512BW	Compare packed signed byte integers in <i>xmm2</i> and <i>xmm3/m128</i> for greater than, and set vector mask <i>k1</i> to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.NDS.256.66.0F.WIG 64 /r VPCMPGTB <i>k1</i> { <i>k2</i> }, <i>ymm2</i> , <i>ymm3/m256</i>	FVM	V/V	AVX512VL AVX512BW	Compare packed signed byte integers in <i>ymm2</i> and <i>ymm3/m256</i> for greater than, and set vector mask <i>k1</i> to reflect the zero/nonzero status of each element of the result, under writemask.

EVEX.NDS.512.66.0F.WIG 64 /r VPCMPGTB k1 {k2}, zmm2, zmm3/m512	FVM V/V	AVX512BW	Compare packed signed byte integers in zmm2 and zmm3/m512 for greater than, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.NDS.128.66.0F.WIG 65 /r VPCMPGTW k1 {k2}, xmm2, xmm3/m128	FVM V/V	AVX512VL AVX512BW	Compare packed signed word integers in xmm2 and xmm3/m128 for greater than, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.NDS.256.66.0F.WIG 65 /r VPCMPGTW k1 {k2}, ymm2, ymm3/m256	FVM V/V	AVX512VL AVX512BW	Compare packed signed word integers in ymm2 and ymm3/m256 for greater than, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.NDS.512.66.0F.WIG 65 /r VPCMPGTW k1 {k2}, zmm2, zmm3/m512	FVM V/V	AVX512BW	Compare packed signed word integers in zmm2 and zmm3/m512 for greater than, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.

**NOTES:**

1. See note in Section 2.4, “AVX and SSE Instruction Exception Specification” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FV	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA
FVM	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

**Description**

Performs an SIMD signed compare for the greater value of the packed byte, word, or doubleword integers in the destination operand (first operand) and the source operand (second operand). If a data element in the destination operand is greater than the corresponding data element in the source operand, the corresponding data element in the destination operand is set to all 1s; otherwise, it is set to all 0s.

The PCMPGTB instruction compares the corresponding signed byte integers in the destination and source operands; the PCMPGTW instruction compares the corresponding signed word integers in the destination and source operands; and the PCMPGTD instruction compares the corresponding signed doubleword integers in the destination and source operands.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE instructions: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand can be an MMX technology register.

128-bit Legacy SSE version: The second source operand can be an XMM register or a 128-bit memory location. The first source operand and destination operand are XMM registers. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The second source operand can be an XMM register or a 128-bit memory location. The first source operand and destination operand are XMM registers. Bits (VLMAX-1:128) of the corresponding YMM register are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register.

EVEX encoded VPCMPGTD: The first source operand (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 32-bit memory location. The destination operand (first operand) is a mask register updated according to the writemask k2.

EVEX encoded VPCMPGTB/W: The first source operand (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. The destination operand (first operand) is a mask register updated according to the writemask k2.

## Operation

### PCMPGTB (with 64-bit operands)

```
IF DEST[7:0] > SRC[7:0]
  THEN DEST[7:0] ← FFH;
  ELSE DEST[7:0] ← 0; FI;
(* Continue comparison of 2nd through 7th bytes in DEST and SRC *)
IF DEST[63:56] > SRC[63:56]
  THEN DEST[63:56] ← FFH;
  ELSE DEST[63:56] ← 0; FI;
```

### COMPARE\_BYTES\_GREATER (SRC1, SRC2)

```
IF SRC1[7:0] > SRC2[7:0]
  THEN DEST[7:0] ← FFH;
  ELSE DEST[7:0] ← 0; FI;
(* Continue comparison of 2nd through 15th bytes in SRC1 and SRC2 *)
IF SRC1[127:120] > SRC2[127:120]
  THEN DEST[127:120] ← FFH;
  ELSE DEST[127:120] ← 0; FI;
```

### COMPARE\_WORDS\_GREATER (SRC1, SRC2)

```
IF SRC1[15:0] > SRC2[15:0]
  THEN DEST[15:0] ← FFFFH;
  ELSE DEST[15:0] ← 0; FI;
(* Continue comparison of 2nd through 7th 16-bit words in SRC1 and SRC2 *)
IF SRC1[127:112] > SRC2[127:112]
  THEN DEST[127:112] ← FFFFH;
  ELSE DEST[127:112] ← 0; FI;
```

### COMPARE\_DWORDS\_GREATER (SRC1, SRC2)

```
IF SRC1[31:0] > SRC2[31:0]
  THEN DEST[31:0] ← FFFFFFFFH;
  ELSE DEST[31:0] ← 0; FI;
(* Continue comparison of 2nd through 3rd 32-bit dwords in SRC1 and SRC2 *)
IF SRC1[127:96] > SRC2[127:96]
  THEN DEST[127:96] ← FFFFFFFFH;
  ELSE DEST[127:96] ← 0; FI;
```

### PCMPGTB (with 128-bit operands)

```
DEST[127:0] ← COMPARE_BYTES_GREATER(DEST[127:0], SRC[127:0])
DEST[MAX_VL-1:128] (Unmodified)
```

### VPCMPGTB (VEX.128 encoded version)

```
DEST[127:0] ← COMPARE_BYTES_GREATER(SRC1, SRC2)
DEST[VLMAX-1:128] ← 0
```



**VPCMPGTB (VEX.256 encoded version)**

```
DEST[127:0] ← COMPARE_BYTES_GREATER(SRC1[127:0], SRC2[127:0])
DEST[255:128] ← COMPARE_BYTES_GREATER(SRC1[255:128], SRC2[255:128])
DEST[VLMAX-1:256] ← 0
```

**VPCMPGTB (EVEX encoded versions)**

(KL, VL) = (16, 128), (32, 256), (64, 512)

```
FOR j ← 0 TO KL-1
```

```
  i ← j * 8
```

```
  IF k2[j] OR *no writemask*
```

```
    THEN
```

```
      /* signed comparison */
```

```
      CMP ← SRC1[i+7:i] > SRC2[i+7:i];
```

```
      IF CMP = TRUE
```

```
        THEN DEST[j] ← 1;
```

```
        ELSE DEST[j] ← 0; FI;
```

```
      ELSE DEST[j] ← 0 ; zeroing-masking only FI;
```

```
    FI;
```

```
ENDFOR
```

```
DEST[MAX_KL-1:KL] ← 0
```

**PCMPGTW (with 64-bit operands)**

```
IF DEST[15:0] > SRC[15:0]
```

```
  THEN DEST[15:0] ← FFFFH;
```

```
  ELSE DEST[15:0] ← 0; FI;
```

(\* Continue comparison of 2nd and 3rd words in DEST and SRC \*)

```
IF DEST[63:48] > SRC[63:48]
```

```
  THEN DEST[63:48] ← FFFFH;
```

```
  ELSE DEST[63:48] ← 0; FI;
```

**PCMPGTW (with 128-bit operands)**

```
DEST[127:0] ← COMPARE_WORDS_GREATER(DEST[127:0], SRC[127:0])
```

```
DEST[MAX_VL-1:128] (Unmodified)
```

**VPCMPGTW (VEX.128 encoded version)**

```
DEST[127:0] ← COMPARE_WORDS_GREATER(SRC1, SRC2)
```

```
DEST[VLMAX-1:128] ← 0
```

**VPCMPGTW (VEX.256 encoded version)**

```
DEST[127:0] ← COMPARE_WORDS_GREATER(SRC1[127:0], SRC2[127:0])
```

```
DEST[255:128] ← COMPARE_WORDS_GREATER(SRC1[255:128], SRC2[255:128])
```

```
DEST[VLMAX-1:256] ← 0
```

**VPCMPGTW (EVEX encoded versions)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

```
FOR j ← 0 TO KL-1
```

```
  i ← j * 16
```

```
  IF k2[j] OR *no writemask*
```

```
    THEN
```

```
      /* signed comparison */
```

```
      CMP ← SRC1[i+15:i] > SRC2[i+15:i];
```

```
      IF CMP = TRUE
```

```
        THEN DEST[j] ← 1;
```

```
        ELSE DEST[j] ← 0; FI;
```

```

        ELSE    DEST[j] ← 0                ; zeroing-masking onlyFi;
    Fi;
ENDFOR
DEST[MAX_KL-1:KL] ← 0

```

**PCMPGTD (with 64-bit operands)**

```

    IF DEST[31:0] > SRC[31:0]
        THEN DEST[31:0] ← FFFFFFFFH;
        ELSE DEST[31:0] ← 0; Fi;
    IF DEST[63:32] > SRC[63:32]
        THEN DEST[63:32] ← FFFFFFFFH;
        ELSE DEST[63:32] ← 0; Fi;

```

**PCMPGTD (with 128-bit operands)**

```

DEST[127:0] ← COMPARE_DWORDS_GREATER(DEST[127:0],SRC[127:0])
DEST[MAX_VL-1:128] (Unmodified)

```

**VPCMPGTD (VEX.128 encoded version)**

```

DEST[127:0] ← COMPARE_DWORDS_GREATER(SRC1,SRC2)
DEST[VLMAX-1:128] ← 0

```

**VPCMPGTD (VEX.256 encoded version)**

```

DEST[127:0] ← COMPARE_DWORDS_GREATER(SRC1[127:0],SRC2[127:0])
DEST[255:128] ← COMPARE_DWORDS_GREATER(SRC1[255:128],SRC2[255:128])
DEST[VLMAX-1:256] ← 0

```

**VPCMPGTD (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (8, 512)

```

FOR j ← 0 TO KL-1
    i ← j * 32
    IF k2[j] OR *no writemask*
        THEN
            /* signed comparison */
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN CMP ← SRC1[i+31:i] > SRC2[31:0];
                ELSE CMP ← SRC1[i+31:i] > SRC2[i+31:i];
            Fi;
            IF CMP = TRUE
                THEN DEST[j] ← 1;
                ELSE DEST[j] ← 0; Fi;
        ELSE    DEST[j] ← 0                ; zeroing-masking only
    Fi;
ENDFOR
DEST[MAX_KL-1:KL] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalents**

```

VPCMPGTB __mmask64 __mm512_cmpgt_epi8_mask(__m512i a, __m512i b);
VPCMPGTB __mmask64 __mm512_mask_cmpgt_epi8_mask(__mmask64 k, __m512i a, __m512i b);
VPCMPGTB __mmask32 __m256_cmpgt_epi8_mask(__m256i a, __m256i b);
VPCMPGTB __mmask32 __m256_mask_cmpgt_epi8_mask(__mmask32 k, __m256i a, __m256i b);
VPCMPGTB __mmask16 __m128_cmpgt_epi8_mask(__m128i a, __m128i b);
VPCMPGTB __mmask16 __m128_mask_cmpgt_epi8_mask(__mmask16 k, __m128i a, __m128i b);
VPCMPGTD __mmask16 __mm512_cmpgt_epi32_mask(__m512i a, __m512i b);

```

VPCMPGTD \_\_mmask16 \_\_mm512\_mask\_cmpgt\_epi32\_mask(\_\_mmask16 k, \_\_m512i a, \_\_m512i b);  
 VPCMPGTD \_\_mmask8 \_\_mm256\_cmpgt\_epi32\_mask(\_\_m256i a, \_\_m256i b);  
 VPCMPGTD \_\_mmask8 \_\_mm256\_mask\_cmpgt\_epi32\_mask(\_\_mmask8 k, \_\_m256i a, \_\_m256i b);  
 VPCMPGTD \_\_mmask8 \_\_mm\_cmpgt\_epi32\_mask(\_\_m128i a, \_\_m128i b);  
 VPCMPGTD \_\_mmask8 \_\_mm\_mask\_cmpgt\_epi32\_mask(\_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
 VPCMPGTW \_\_mmask32 \_\_mm512\_cmpgt\_epi16\_mask(\_\_m512i a, \_\_m512i b);  
 VPCMPGTW \_\_mmask32 \_\_mm512\_mask\_cmpgt\_epi16\_mask(\_\_mmask32 k, \_\_m512i a, \_\_m512i b);  
 VPCMPGTW \_\_mmask16 \_\_mm256\_cmpgt\_epi16\_mask(\_\_m256i a, \_\_m256i b);  
 VPCMPGTW \_\_mmask16 \_\_mm256\_mask\_cmpgt\_epi16\_mask(\_\_mmask16 k, \_\_m256i a, \_\_m256i b);  
 VPCMPGTW \_\_mmask8 \_\_mm\_cmpgt\_epi16\_mask(\_\_m128i a, \_\_m128i b);  
 VPCMPGTW \_\_mmask8 \_\_mm\_mask\_cmpgt\_epi16\_mask(\_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
 PCMPGTB: \_\_m64 \_\_mm\_cmpgt\_pi8 (\_\_m64 m1, \_\_m64 m2)  
 PCMPGTW: \_\_m64 \_\_mm\_pcmpgt\_pi16 (\_\_m64 m1, \_\_m64 m2)  
 PCMPGTD: \_\_m64 \_\_mm\_pcmpgt\_pi32 (\_\_m64 m1, \_\_m64 m2)  
 (V)PCMPGTB: \_\_m128i \_\_mm\_cmpgt\_epi8 (\_\_m128i a, \_\_m128i b)  
 (V)PCMPGTW: \_\_m128i \_\_mm\_cmpgt\_epi16 (\_\_m128i a, \_\_m128i b)  
 (V)DCMPGTD: \_\_m128i \_\_mm\_cmpgt\_epi32 (\_\_m128i a, \_\_m128i b)  
 VPCMPGTB: \_\_m256i \_\_mm256\_cmpgt\_epi8 (\_\_m256i a, \_\_m256i b)  
 VPCMPGTW: \_\_m256i \_\_mm256\_cmpgt\_epi16 (\_\_m256i a, \_\_m256i b)  
 VPCMPGTD: \_\_m256i \_\_mm256\_cmpgt\_epi32 (\_\_m256i a, \_\_m256i b)

### Flags Affected

None.

### Numeric Exceptions

None.

### Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.  
 EVEX-encoded VPCMPGTD, see Exceptions Type E4.  
 EVEX-encoded VPCMPGTB/W, see Exceptions Type E4.nb.

## PCMPGTQ – Compare Packed Data for Greater Than

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 37 /r PCMPGTQ <i>xmm1,xmm2/m128</i>	RM	V/V	SSE4_2	Compare packed signed qwords in <i>xmm2/m128</i> and <i>xmm1</i> for greater than.
VEX.NDS.128.66.0F38.WIG 37 /r VPCMPGTQ <i>xmm1, xmm2, xmm3/m128</i>	RVM	V/V	AVX	Compare packed signed qwords in <i>xmm2</i> and <i>xmm3/m128</i> for greater than.
VEX.NDS.256.66.0F38.WIG 37 /r VPCMPGTQ <i>ymm1, ymm2, ymm3/m256</i>	RVM	V/V	AVX2	Compare packed signed qwords in <i>ymm2</i> and <i>ymm3/m256</i> for greater than.
EVEX.NDS.128.66.0F38.W1 37 /r VPCMPGTQ <i>k1 {k2}, xmm2, xmm3/m128/m64bcst</i>	FV	V/V	AVX512VL AVX512F	Compare Greater between int64 vector <i>xmm2</i> and int64 vector <i>xmm3/m128/m64bcst</i> , and set vector mask <i>k1</i> to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.NDS.256.66.0F38.W1 37 /r VPCMPGTQ <i>k1 {k2}, ymm2, ymm3/m256/m64bcst</i>	FV	V/V	AVX512VL AVX512F	Compare Greater between int64 vector <i>ymm2</i> and int64 vector <i>ymm3/m256/m64bcst</i> , and set vector mask <i>k1</i> to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.NDS.512.66.0F38.W1 37 /r VPCMPGTQ <i>k1 {k2}, zmm2, zmm3/m512/m64bcst</i>	FV	V/V	AVX512F	Compare Greater between int64 vector <i>zmm2</i> and int64 vector <i>zmm3/m512/m64bcst</i> , and set vector mask <i>k1</i> to reflect the zero/nonzero status of each element of the result, under writemask.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FV	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Performs an SIMD signed compare for the packed quadwords in the destination operand (first operand) and the source operand (second operand). If the data element in the first (destination) operand is greater than the corresponding element in the second (source) operand, the corresponding data element in the destination is set to all 1s; otherwise, it is set to 0s.

128-bit Legacy SSE version: The second source operand can be an XMM register or a 128-bit memory location. The first source operand and destination operand are XMM registers. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The second source operand can be an XMM register or a 128-bit memory location. The first source operand and destination operand are XMM registers. Bits (VLMAX-1:128) of the corresponding YMM register are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register.

EVEX encoded VPCMPGTQ: The first source operand (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 64-bit memory location. The destination operand (first operand) is a mask register updated according to the writemask *k2*.

**Operation****COMPARE\_QWORDS\_GREATER (SRC1, SRC2)**

```

IF SRC1[63:0] > SRC2[63:0]
THEN DEST[63:0] ← FFFFFFFFFFFFFFFFH;
ELSE DEST[63:0] ← 0; FI;
IF SRC1[127:64] > SRC2[127:64]
THEN DEST[127:64] ← FFFFFFFFFFFFFFFFH;
ELSE DEST[127:64] ← 0; FI;

```

**VPCMPGTQ (VEX.128 encoded version)**

```

DEST[127:0] ← COMPARE_QWORDS_GREATER(SRC1, SRC2)
DEST[VLMAX-1:128] ← 0

```

**VPCMPGTQ (VEX.256 encoded version)**

```

DEST[127:0] ← COMPARE_QWORDS_GREATER(SRC1[127:0], SRC2[127:0])
DEST[255:128] ← COMPARE_QWORDS_GREATER(SRC1[255:128], SRC2[255:128])
DEST[VLMAX-1:256] ← 0

```

**VPCMPGTQ (EVEX encoded versions)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

  i ← j \* 64

  IF k2[j] OR \*no writemask\*

    THEN

      /\* signed comparison \*/

      IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

        THEN CMP ← SRC1[i+63:i] > SRC2[63:0];

        ELSE CMP ← SRC1[i+63:i] > SRC2[i+63:i];

      FI;

      IF CMP = TRUE

        THEN DEST[j] ← 1;

        ELSE DEST[j] ← 0; FI;

    ELSE DEST[j] ← 0 ; zeroing-masking only

  FI;

ENDFOR

DEST[MAX\_KL-1:KL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VPCMPGTQ __mmask8 _mm512_cmpgt_epi64_mask( __m512i a, __m512i b);
VPCMPGTQ __mmask8 _mm512_mask_cmpgt_epi64_mask(__mmask8 k, __m512i a, __m512i b);
VPCMPGTQ __mmask8 _mm256_cmpgt_epi64_mask( __m256i a, __m256i b);
VPCMPGTQ __mmask8 _mm256_mask_cmpgt_epi64_mask(__mmask8 k, __m256i a, __m256i b);
VPCMPGTQ __mmask8 _mm_cmpgt_epi64_mask( __m128i a, __m128i b);
VPCMPGTQ __mmask8 _mm_mask_cmpgt_epi64_mask(__mmask8 k, __m128i a, __m128i b);
(V)VPCMPGTQ: __m128i _mm_cmpgt_epi64(__m128i a, __m128i b)
VPCMPGTQ: __m256i _mm256_cmpgt_epi64( __m256i a, __m256i b);

```

**Flags Affected**

None.

**SIMD Floating-Point Exceptions**

None.



## Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded VPCMPGTQ, see Exceptions Type E4.

**PEXTRB/PEXTRD/PEXTRQ – Extract Byte/Dword/Qword**

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 3A 14 /r ib PEXTRB <i>reg/m8, xmm2, imm8</i>	MRI	V/V	SSE4_1	Extract a byte integer value from <i>xmm2</i> at the source byte offset specified by <i>imm8</i> into <i>reg</i> or <i>m8</i> . The upper bits of <i>r32</i> or <i>r64</i> are zeroed.
66 0F 3A 16 /r ib PEXTRD <i>r/m32, xmm2, imm8</i>	MRI	V/V	SSE4_1	Extract a dword integer value from <i>xmm2</i> at the source dword offset specified by <i>imm8</i> into <i>r/m32</i> .
66 REX.W 0F 3A 16 /r ib PEXTRQ <i>r/m64, xmm2, imm8</i>	MRI	V/N.E.	SSE4_1	Extract a qword integer value from <i>xmm2</i> at the source qword offset specified by <i>imm8</i> into <i>r/m64</i> .
VEX.128.66.0F3A.W0 14 /r ib VPEXTRB <i>reg/m8, xmm2, imm8</i>	MRI	V <sup>1</sup> /V	AVX	Extract a byte integer value from <i>xmm2</i> at the source byte offset specified by <i>imm8</i> into <i>reg</i> or <i>m8</i> . The upper bits of <i>r64/r32</i> is filled with zeros.
VEX.128.66.0F3A.W0 16 /r ib VPEXTRD <i>r32/m32, xmm2, imm8</i>	MRI	V/V	AVX	Extract a dword integer value from <i>xmm2</i> at the source dword offset specified by <i>imm8</i> into <i>r32/m32</i> .
VEX.128.66.0F3A.W1 16 /r ib VPEXTRQ <i>r64/m64, xmm2, imm8</i>	MRI	V/i	AVX	Extract a qword integer value from <i>xmm2</i> at the source dword offset specified by <i>imm8</i> into <i>r64/m64</i> .
EVEX.128.66.0F3A.WIG 14 /r ib VPEXTRB <i>reg/m8, xmm2, imm8</i>	T1S-MRI	V/V	AVX512BW	Extract a byte integer value from <i>xmm2</i> at the source byte offset specified by <i>imm8</i> into <i>reg</i> or <i>m8</i> . The upper bits of <i>r64/r32</i> is filled with zeros.
EVEX.128.66.0F3A.W0 16 /r ib VPEXTRD <i>r32/m32, xmm2, imm8</i>	T1S-MRI	V/V	AVX512DQ	Extract a dword integer value from <i>xmm2</i> at the source dword offset specified by <i>imm8</i> into <i>r32/m32</i> .
EVEX.128.66.0F3A.W1 16 /r ib VPEXTRQ <i>r64/m64, xmm2, imm8</i>	T1S-MRI	V/N.E. <sup>1</sup>	AVX512DQ	Extract a qword integer value from <i>xmm2</i> at the source dword offset specified by <i>imm8</i> into <i>r64/m64</i> .

**NOTES:**

1. In 64-bit mode, VEX.W1 is ignored for VPEXTRB (similar to legacy REX.W=1 prefix in PEXTRB).

**NOTES:**

1. VEX.W/EVEX.W in non-64 bit is ignored; the instructions behaves as if the W0 version is used.

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MRI	ModRM:r/m (w)	ModRM:reg (r)	imm8	NA

**Description**

Extract a byte/dword/qword integer value from the source XMM register at a byte/dword/qword offset determined from *imm8*[3:0]. The destination can be a register or byte/dword/qword memory location. If the destination is a register, the upper bits of the register are zero extended.

In legacy non-VEX encoded version and if the destination operand is a register, the default operand size in 64-bit mode for PEXTRB/PEXTRD is 64 bits, the bits above the least significant byte/dword data are filled with zeros. PEXTRQ is not encodable in non-64-bit modes and requires REX.W in 64-bit mode.

Note: In VEX.128 encoded versions, VEX.vvvv is reserved and must be 1111b, VEX.L must be 0, otherwise the instruction will #UD. In EVEX.128 encoded versions, EVEX.vvvv is reserved and must be 1111b, EVEX.L'L must be 0, otherwise the instruction will #UD. If the destination operand is a register, the default operand size in 64-bit mode for VPEXTRB/VPEXTRD is 64 bits, the bits above the least significant byte/word/dword data are filled with zeros. Attempt to execute VPEXTRQ in non-64-bit mode will cause #UD.

## Operation

CASE of

```

PEXTRB: SEL ← COUNT[3:0];
        TEMP ← (Src >> SEL*8) AND FFH;
        IF (DEST = Mem8)
            THEN
                Mem8 ← TEMP[7:0];
        ELSE IF (64-Bit Mode and 64-bit register selected)
            THEN
                R64[7:0] ← TEMP[7:0];
                r64[63:8] ← ZERO_FILL;
        ELSE
                R32[7:0] ← TEMP[7:0];
                r32[31:8] ← ZERO_FILL;
        FI;
PEXTRD:SEL ← COUNT[1:0];
        TEMP ← (Src >> SEL*32) AND FFFF_FFFFH;
        DEST ← TEMP;
PEXTRQ: SEL ← COUNT[0];
        TEMP ← (Src >> SEL*64);
        DEST ← TEMP;

```

EASC:

### VPEXTRTD/VPEXTRQ

```

IF (64-Bit Mode and 64-bit dest operand)
    THEN
        Src_Offset ← Imm8[0]
        r64/m64 ← (Src >> Src_Offset * 64)
    ELSE
        Src_Offset ← Imm8[1:0]
        r32/m32 ← ((Src >> Src_Offset * 32) AND 0FFFFFFFh);
    FI

```

### VPEXTRB ( dest=m8)

```

SRC_Offset ← Imm8[3:0]
Mem8 ← (Src >> Src_Offset*8)

```

### VPEXTRB ( dest=reg)

```

IF (64-Bit Mode )
    THEN
        SRC_Offset ← Imm8[3:0]
        DEST[7:0] ← ((Src >> Src_Offset*8) AND 0FFh)
        DEST[63:8] ← ZERO_FILL;
    ELSE
        SRC_Offset ← Imm8[3:0];
        DEST[7:0] ← ((Src >> Src_Offset*8) AND 0FFh);
        DEST[31:8] ← ZERO_FILL;
    FI

```



### Intel C/C++ Compiler Intrinsic Equivalent

PEXTRB:     int \_mm\_extract\_epi8 (\_\_m128i src, const int ndx);  
PEXTRD:     int \_mm\_extract\_epi32 (\_\_m128i src, const int ndx);  
PEXTRQ:     \_\_int64 \_mm\_extract\_epi64 (\_\_m128i src, const int ndx);

### Flags Affected

None.

### SIMD Floating-Point Exceptions

None.

### Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 5;

EVEX-encoded instruction, see Exceptions Type E9NF.

#UD                 If VEX.L = 1 or EVEX.L'L > 0.  
                      If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.  
                      If VPEXTRQ in non-64-bit mode, VEX.W=1.

## PEXTRW—Extract Word

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F C5 /r ib <sup>1</sup> PEXTRW <i>reg, mm, imm8</i>	RMI	V/V	SSE	Extract the word specified by <i>imm8</i> from <i>mm</i> and move it to <i>reg</i> , bits 15-0. The upper bits of r32 or r64 is zeroed.
66 0F C5 /r ib PEXTRW <i>reg, xmm, imm8</i>	RMI	V/V	SSE2	Extract the word specified by <i>imm8</i> from <i>xmm</i> and move it to <i>reg</i> , bits 15-0. The upper bits of r32 or r64 is zeroed.
66 0F 3A 15 /r ib PEXTRW <i>reg/m16, xmm, imm8</i>	MRI	V/V	SSE4_1	Extract the word specified by <i>imm8</i> from <i>xmm</i> and copy it to lowest 16 bits of <i>reg</i> or <i>m16</i> . Zero-extend the result in the destination, r32 or r64.
VEX.128.66.0F.W0 C5 /r ib VPEXTRW <i>reg, xmm1, imm8</i>	RMI	V <sup>2</sup> /V	AVX	Extract the word specified by <i>imm8</i> from <i>xmm1</i> and move it to <i>reg</i> , bits 15:0. Zero-extend the result. The upper bits of r64/r32 is filled with zeros.
VEX.128.66.0F3A.W0 15 /r ib VPEXTRW <i>reg/m16, xmm2, imm8</i>	MRI	V/V	AVX	Extract a word integer value from <i>xmm2</i> at the source word offset specified by <i>imm8</i> into <i>reg</i> or <i>m16</i> . The upper bits of r64/r32 is filled with zeros.
EVEX.128.66.0F.WIG C5 /r ib VPEXTRW <i>reg, xmm1, imm8</i>	RMI	V/V	AVX512B W	Extract the word specified by <i>imm8</i> from <i>xmm1</i> and move it to <i>reg</i> , bits 15:0. Zero-extend the result. The upper bits of r64/r32 is filled with zeros.
EVEX.128.66.0F3A.WIG 15 /r ib VPEXTRW <i>reg/m16, xmm2, imm8</i>	T1S- MRI	V/V	AVX512B W	Extract a word integer value from <i>xmm2</i> at the source word offset specified by <i>imm8</i> into <i>reg</i> or <i>m16</i> . The upper bits of r64/r32 is filled with zeros.

### NOTES:

- See note in Section 2.4, “AVX and SSE Instruction Exception Specification” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.
- In 64-bit mode, VEX.W1 is ignored for VPEXTRW (similar to legacy REX.W=1 prefix in PEXTRW).

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (w)	ModRM:r/m (r)	imm8	NA
MRI	ModRM:r/m (w)	ModRM:reg (r)	imm8	NA

### Description

Copies the word in the source operand (second operand) specified by the count operand (third operand) to the destination operand (first operand). The source operand can be an MMX technology register or an XMM register. The destination operand can be the low word of a general-purpose register or a 16-bit memory address. The count operand is an 8-bit immediate. When specifying a word location in an MMX technology register, the 2 least-significant bits of the count operand specify the location; for an XMM register, the 3 least-significant bits specify the location. The content of the destination register above bit 16 is cleared (set to all 0s).

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15, R8-15). If the destination operand is a general-purpose register, the default operand size is 64-bits in 64-bit mode.

Note: In VEX.128 encoded versions, VEX.vvvv is reserved and must be 1111b, VEX.L must be 0, otherwise the instruction will #UD. In EVEX.128 encoded versions, EVEX.vvvv is reserved and must be 1111b, EVEX.L must be 0, otherwise the instruction will #UD. If the destination operand is a register, the default operand size in 64-bit mode for VPEXTRW is 64 bits, the bits above the least significant byte/word/dword data are filled with zeros.

## Operation

```

IF (DEST = Mem16)
THEN
    SEL ← COUNT[2:0];
    TEMP ← (Src >> SEL*16) AND FFFFH;
    Mem16 ← TEMP[15:0];
ELSE IF (64-Bit Mode and destination is a general-purpose register)
THEN
    FOR (PEXTRW instruction with 64-bit source operand)
    { SEL ← COUNT[1:0];
      TEMP ← (SRC >> (SEL * 16)) AND FFFFH;
      r64[15:0] ← TEMP[15:0];
      r64[63:16] ← ZERO_FILL; };
    FOR (PEXTRW instruction with 128-bit source operand)
    { SEL ← COUNT[2:0];
      TEMP ← (SRC >> (SEL * 16)) AND FFFFH;
      r64[15:0] ← TEMP[15:0];
      r64[63:16] ← ZERO_FILL; }
ELSE
    FOR (PEXTRW instruction with 64-bit source operand)
    { SEL ← COUNT[1:0];
      TEMP ← (SRC >> (SEL * 16)) AND FFFFH;
      r32[15:0] ← TEMP[15:0];
      r32[31:16] ← ZERO_FILL; };
    FOR (PEXTRW instruction with 128-bit source operand)
    { SEL ← COUNT[2:0];
      TEMP ← (SRC >> (SEL * 16)) AND FFFFH;
      r32[15:0] ← TEMP[15:0];
      r32[31:16] ← ZERO_FILL; };
FI;
FI;

```

### VPEXTRW ( dest=m16)

```

SRC_Offset ← Imm8[2:0]
Mem16 ← (Src >> Src_Offset*16)

```

### VPEXTRW ( dest=reg)

```

IF (64-Bit Mode )
THEN
    SRC_Offset ← Imm8[2:0]
    DEST[15:0] ← ((Src >> Src_Offset*16) AND 0FFFFh)
    DEST[63:16] ← ZERO_FILL;
ELSE
    SRC_Offset ← Imm8[2:0]
    DEST[15:0] ← ((Src >> Src_Offset*16) AND 0FFFFh)
    DEST[31:16] ← ZERO_FILL;

```

FI

**Intel C/C++ Compiler Intrinsic Equivalent**PEXTRW: `int _mm_extract_pi16 (__m64 a, int n)`PEXTRW: `int _mm_extract_epi16 (__m128i a, int imm)`**Flags Affected**

None.

**Numeric Exceptions**

None.

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 5;

EVEX-encoded instruction, see Exceptions Type E9NF.

#UD	If VEX.L = 1 or EVEX.L'L > 0.
	If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

## PINSRB/PINSRD/PINSRQ – Insert Byte/Dword/Qword

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 3A 20 /r ib PINSRB <i>xmm1</i> , <i>r32/m8</i> , <i>imm8</i>	RMI	V/V	SSE4_1	Insert a byte integer value from <i>r32/m8</i> into <i>xmm1</i> at the destination element in <i>xmm1</i> specified by <i>imm8</i> .
66 0F 3A 22 /r ib PINSRD <i>xmm1</i> , <i>r/m32</i> , <i>imm8</i>	RMI	V/V	SSE4_1	Insert a dword integer value from <i>r/m32</i> into the <i>xmm1</i> at the destination element specified by <i>imm8</i> .
66 REX.W 0F 3A 22 /r ib PINSRQ <i>xmm1</i> , <i>r/m64</i> , <i>imm8</i>	RMI	V/N. E.	SSE4_1	Insert a qword integer value from <i>r/m64</i> into the <i>xmm1</i> at the destination element specified by <i>imm8</i> .
VEX.NDS.128.66.0F3A.W0 20 /r ib VPINSRB <i>xmm1</i> , <i>xmm2</i> , <i>r32/m8</i> , <i>imm8</i>	RVMI	V <sup>1</sup> /V	AVX	Merge a byte integer value from <i>r32/m8</i> and rest from <i>xmm2</i> into <i>xmm1</i> at the byte offset in <i>imm8</i> .
VEX.NDS.128.66.0F3A.W0 22 /r ib VPINSRD <i>xmm1</i> , <i>xmm2</i> , <i>r/m32</i> , <i>imm8</i>	RVMI	V/V	AVX	Insert a dword integer value from <i>r32/m32</i> and rest from <i>xmm2</i> into <i>xmm1</i> at the dword offset in <i>imm8</i> .
VEX.NDS.128.66.0F3A.W1 22 /r ib VPINSRQ <i>xmm1</i> , <i>xmm2</i> , <i>r/m64</i> , <i>imm8</i>	RVMI	V/I	AVX	Insert a qword integer value from <i>r64/m64</i> and rest from <i>xmm2</i> into <i>xmm1</i> at the qword offset in <i>imm8</i> .
EVEX.NDS.128.66.0F3A.WIG 20 /r ib VPINSRB <i>xmm1</i> , <i>xmm2</i> , <i>r32/m8</i> , <i>imm8</i>	T1S- RVMI	V/V	AVX512BW	Merge a byte integer value from <i>r32/m8</i> and rest from <i>xmm2</i> into <i>xmm1</i> at the byte offset in <i>imm8</i> .
EVEX.NDS.128.66.0F3A.W0 22 /r ib VPINSRD <i>xmm1</i> , <i>xmm2</i> , <i>r32/m32</i> , <i>imm8</i>	T1S- RVMI	V/V	AVX512DQ	Insert a dword integer value from <i>r32/m32</i> and rest from <i>xmm2</i> into <i>xmm1</i> at the dword offset in <i>imm8</i> .
EVEX.NDS.128.66.0F3A.W1 22 /r ib VPINSRQ <i>xmm1</i> , <i>xmm2</i> , <i>r64/m64</i> , <i>imm8</i>	T1S- RVMI	V/N.E. <sup>1</sup>	AVX512DQ	Insert a qword integer value from <i>r64/m64</i> and rest from <i>xmm2</i> into <i>xmm1</i> at the qword offset in <i>imm8</i> .

### NOTES:

1. In 64-bit mode, VEX.W1 is ignored for VPINSRB (similar to legacy REX.W=1 prefix with PINSRB).

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (w)	ModRM:r/m (r)	imm8	NA
RVMI	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8
T1S-RVMI	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	Imm8

### Description

Copies a byte/dword/qword from the source operand (second operand) and inserts it in the destination operand (first operand) at the location specified with the count operand (third operand). (The other elements in the destination register are left untouched.) The source operand can be a general-purpose register or a memory location. (When the source operand is a general-purpose register, PINSRB copies the low byte of the register.) The destination operand is an XMM register. The count operand is an 8-bit immediate. When specifying a qword[dword, byte] location in an XMM register, the [2, 4] least-significant bit(s) of the count operand specify the location.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15, R8-15). Use of REX.W permits the use of 64 bit general purpose registers.

128-bit Legacy SSE version: Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination register are zeroed. VEX.L must be 0, otherwise the instruction will #UD. Attempt to execute VPINSRQ in non-64-bit mode will cause #UD.

EVEX.128 encoded version: Bits (VLMAX-1:128) of the destination register are zeroed. EVEX.L'L must be 0, otherwise the instruction will #UD.

## Operation

CASE OF

```
PINSRB: SEL ← COUNT[3:0];
        MASK ← (OFFH << (SEL * 8));
        TEMP ← (((SRC[7:0] << (SEL * 8)) AND MASK);
PINSRD: SEL ← COUNT[1:0];
        MASK ← (OFFFFFFFFFH << (SEL * 32));
        TEMP ← (((SRC << (SEL * 32)) AND MASK) ;
PINSRQ: SEL ← COUNT[0]
        MASK ← (OFFFFFFFFFHH << (SEL * 64));
        TEMP ← (((SRC << (SEL * 32)) AND MASK) ;
```

ESAC;

```
DEST ← ((DEST AND NOT MASK) OR TEMP);
```

### VPINSRB (VEX/EVEX encoded version)

```
SEL ← imm8[3:0]
DEST[127:0] ← write_b_element(SEL, SRC2, SRC1)
DEST[VLMAX-1:128] ← 0
```

### VPINSRD (VEX/EVEX encoded version)

```
SEL ← imm8[1:0]
DEST[127:0] ← write_d_element(SEL, SRC2, SRC1)
DEST[VLMAX-1:128] ← 0
```

### VPINSRQ (VEX/EVEX encoded version)

```
SEL ← imm8[0]
DEST[127:0] ← write_q_element(SEL, SRC2, SRC1)
DEST[VLMAX-1:128] ← 0
```

## Intel C/C++ Compiler Intrinsic Equivalent

```
PINSRB:    __m128i _mm_insert_epi8 (__m128i s1, int s2, const int ndx);
PINSRD:    __m128i _mm_insert_epi32 (__m128i s2, int s, const int ndx);
PINSRQ:    __m128i _mm_insert_epi64(__m128i s2, __int64 s, const int ndx);
```

## Flags Affected

None.

## SIMD Floating-Point Exceptions

None.

### Other Exceptions

EVEX-encoded instruction, see Exceptions Type 5;

EVEX-encoded instruction, see Exceptions Type E9NF.

#UD                    If VEX.L = 1 or EVEX.L'L > 0.  
                          If VPINSRQ in non-64-bit mode with VEX.W=1.

## PINSRW—Insert Word

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F C4 /r ib <sup>1</sup> PINSRW <i>mm</i> , <i>r32/m16</i> , <i>imm8</i>	RMI	V/V	SSE	Insert the low word from <i>r32</i> or from <i>m16</i> into <i>mm</i> at the word position specified by <i>imm8</i> .
66 0F C4 /r ib PINSRW <i>xmm</i> , <i>r32/m16</i> , <i>imm8</i>	RMI	V/V	SSE2	Move the low word of <i>r32</i> or from <i>m16</i> into <i>xmm</i> at the word position specified by <i>imm8</i> .
VEX.NDS.128.66.0F.W0 C4 /r ib VPINSRW <i>xmm1</i> , <i>xmm2</i> , <i>r32/m16</i> , <i>imm8</i>	RVMI	V <sup>2</sup> /V	AVX	Insert a word integer value from <i>r32/m16</i> and rest from <i>xmm2</i> into <i>xmm1</i> at the word offset in <i>imm8</i> .
EVEX.NDS.128.66.0F.WIG C4 /r ib VPINSRW <i>xmm1</i> , <i>xmm2</i> , <i>r32/m16</i> , <i>imm8</i>	T1S- RVMI	V/V	AVX512BW	Insert a word integer value from <i>r32/m16</i> and rest from <i>xmm2</i> into <i>xmm1</i> at the word offset in <i>imm8</i> .

### NOTES:

- See note in Section 2.4, “AVX and SSE Instruction Exception Specification” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.
- In 64-bit mode, VEX.W1 is ignored for VPINSRW (similar to legacy REX.W=1 prefix in PINSRW).

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (w)	ModRM:r/m (r)	imm8	NA
RVMI	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8
T1S-RVMI	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	Imm8

### Description

Copies a word from the source operand (second operand) and inserts it in the destination operand (first operand) at the location specified with the count operand (third operand). (The other words in the destination register are left untouched.) The source operand can be a general-purpose register or a 16-bit memory location. (When the source operand is a general-purpose register, the low word of the register is copied.) The destination operand can be an MMX technology register or an XMM register. The count operand is an 8-bit immediate. When specifying a word location in an MMX technology register, the 2 least-significant bits of the count operand specify the location; for an XMM register, the 3 least-significant bits specify the location.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15, R8-15).

128-bit Legacy SSE version: Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

EVEX.128 encoded version: Bits (VLMAX-1:128) of the destination register are zeroed. EVEX.L'L must be 0, otherwise the instruction will #UD.

### Operation

#### PINSRW (with 64-bit source operand)

```
SEL ← COUNT AND 3H;
CASE (Determine word position) OF
```



SEL ← 0: MASK ← 000000000000FFFFH;  
 SEL ← 1: MASK ← 00000000FFFF0000H;  
 SEL ← 2: MASK ← 0000FFFF00000000H;  
 SEL ← 3: MASK ← FFFF000000000000H;

DEST ← (DEST AND NOT MASK) OR (((SRC << (SEL \* 16)) AND MASK);

#### PINSRW (with 128-bit source operand)

SEL ← COUNT AND 7H;

CASE (Determine word position) OF

SEL ← 0: MASK ← 000000000000000000000000FFFFH;  
 SEL ← 1: MASK ← 000000000000000000000000FFFF0000H;  
 SEL ← 2: MASK ← 000000000000000000000000FFFF00000000H;  
 SEL ← 3: MASK ← 000000000000000000000000FFFF000000000000H;  
 SEL ← 4: MASK ← 000000000000000000000000FFFF00000000000000H;  
 SEL ← 5: MASK ← 00000000FFFF00000000000000000000000000H;  
 SEL ← 6: MASK ← 0000FFFF000000000000000000000000000000H;  
 SEL ← 7: MASK ← FFFF0000000000000000000000000000000000H;

DEST ← (DEST AND NOT MASK) OR (((SRC << (SEL \* 16)) AND MASK);

#### VPINSRW (VEX/EVEX encoded version)

SEL ← imm8[2:0]

DEST[127:0] ← write\_w\_element(SEL, SRC2, SRC1)

DEST[VLMAX-1:128] ← 0

#### Intel C/C++ Compiler Intrinsic Equivalent

PINSRW: `__m64 _mm_insert_pi16 (__m64 a, int d, int n)`

PINSRW: `__m128i _mm_insert_epi16 (__m128i a, int b, int imm)`

#### Flags Affected

None.

#### Numeric Exceptions

None.

#### Other Exceptions

EVEX-encoded instruction, see Exceptions Type 5;

EVEX-encoded instruction, see Exceptions Type E9NF.

#UD If VEX.L = 1 or EVEX.L'L > 0.

## PMADDUBSW – Multiply and Add Packed Signed and Unsigned Bytes

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F 38 04 /r <sup>1</sup> PMADDUBSW <i>mm1, mm2/m64</i>	RM	V/V	SSSE3	Multiply signed and unsigned bytes, add horizontal pair of signed words, pack saturated signed-words to <i>mm1</i> .
66 0F 38 04 /r PMADDUBSW <i>xmm1, xmm2/m128</i>	RM	V/V	SSSE3	Multiply signed and unsigned bytes, add horizontal pair of signed words, pack saturated signed-words to <i>xmm1</i> .
VEX.NDS.128.66.0F38.WIG 04 /r VPMADDUBSW <i>xmm1, xmm2, xmm3/m128</i>	RVM	V/V	AVX	Multiply signed and unsigned bytes, add horizontal pair of signed words, pack saturated signed-words to <i>xmm1</i> .
VEX.NDS.256.66.0F38.WIG 04 /r VPMADDUBSW <i>ymm1, ymm2, ymm3/m256</i>	RVM	V/V	AVX2	Multiply signed and unsigned bytes, add horizontal pair of signed words, pack saturated signed-words to <i>ymm1</i> .
EVEX.NDS.128.66.0F38.WIG 04 /r VPMADDUBSW <i>xmm1 {k1}{z}, xmm2, xmm3/m128</i>	FVM	V/V	AVX512VL AVX512BW	Multiply signed and unsigned bytes, add horizontal pair of signed words, pack saturated signed-words to <i>xmm1</i> under writemask <i>k1</i> .
EVEX.NDS.256.66.0F38.WIG 04 /r VPMADDUBSW <i>ymm1 {k1}{z}, ymm2, ymm3/m256</i>	FVM	V/V	AVX512VL AVX512BW	Multiply signed and unsigned bytes, add horizontal pair of signed words, pack saturated signed-words to <i>ymm1</i> under writemask <i>k1</i> .
EVEX.NDS.512.66.0F38.WIG 04 /r VPMADDUBSW <i>zmm1 {k1}{z}, zmm2, zmm3/m512</i>	FVM	V/V	AVX512BW	Multiply signed and unsigned bytes, add horizontal pair of signed words, pack saturated signed-words to <i>zmm1</i> under writemask <i>k1</i> .

### NOTES:

1. See note in Section 2.4, “AVX and SSE Instruction Exception Specification” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg ( <i>r, w</i> )	ModRM:r/m ( <i>r</i> )	NA	NA
RVM	ModRM:reg ( <i>w</i> )	VEX.vvvv ( <i>r</i> )	ModRM:r/m ( <i>r</i> )	NA
FVM	ModRM:reg ( <i>w</i> )	EVEX.vvvv ( <i>r</i> )	ModRM:r/m ( <i>r</i> )	NA

### Description

(V)PMADDUBSW multiplies vertically each unsigned byte of the destination operand (first operand) with the corresponding signed byte of the source operand (second operand), producing intermediate signed 16-bit integers. Each adjacent pair of signed words is added and the saturated result is packed to the destination operand. For example, the lowest-order bytes (bits 7-0) in the source and destination operands are multiplied and the intermediate signed word result is added with the corresponding intermediate result from the 2nd lowest-order bytes (bits 15-8) of the operands; the sign-saturated result is stored in the lowest word of the destination register (15-0). The same operation is performed on the other pairs of adjacent bytes. Both operands can be MMX register or XMM registers. When the source operand is a 128-bit memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

In 64-bit mode and not encoded with VEX/EVEX, use the REX prefix to access XMM8-XMM15.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAX\_VL-1:128) of the corresponding destination register remain unchanged.

VEX.128 and EVEX.128 encoded versions: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAX\_VL-1:128) of the corresponding destination register are zeroed.

VEX.256 and EVEX.256 encoded versions: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers. Bits (MAX\_VL-1:256) of the corresponding ZMM register are zeroed.

EVEX.512 encoded version: The second source operand can be an ZMM register or a 512-bit memory location. The first source and destination operands are ZMM registers.

## Operation

### PMADDUBSW (with 64 bit operands)

```
DEST[15:0] = SaturateToSignedWord(SRC[15:8]*DEST[15:8]+SRC[7:0]*DEST[7:0]);
DEST[31:16] = SaturateToSignedWord(SRC[31:24]*DEST[31:24]+SRC[23:16]*DEST[23:16]);
DEST[47:32] = SaturateToSignedWord(SRC[47:40]*DEST[47:40]+SRC[39:32]*DEST[39:32]);
DEST[63:48] = SaturateToSignedWord(SRC[63:56]*DEST[63:56]+SRC[55:48]*DEST[55:48]);
```

### PMADDUBSW (with 128 bit operands)

```
DEST[15:0] = SaturateToSignedWord(SRC[15:8]* DEST[15:8]+SRC[7:0]*DEST[7:0]);
// Repeat operation for 2nd through 7th word
SRC1/DEST[127:112] = SaturateToSignedWord(SRC[127:120]*DEST[127:120]+ SRC[119:112]* DEST[119:112]);
```

### VPMADDUBSW (VEX.128 encoded version)

```
DEST[15:0] ← SaturateToSignedWord(SRC2[15:8]* SRC1[15:8]+SRC2[7:0]*SRC1[7:0])
// Repeat operation for 2nd through 7th word
DEST[127:112] ← SaturateToSignedWord(SRC2[127:120]*SRC1[127:120]+ SRC2[119:112]* SRC1[119:112])
DEST[VLMAX-1:128] ← 0
```

### VPMADDUBSW (VEX.256 encoded version)

```
DEST[15:0] ← SaturateToSignedWord(SRC2[15:8]* SRC1[15:8]+SRC2[7:0]*SRC1[7:0])
// Repeat operation for 2nd through 15th word
DEST[255:240] ← SaturateToSignedWord(SRC2[255:248]*SRC1[255:248]+ SRC2[247:240]* SRC1[247:240])
DEST[VLMAX-1:256] ← 0
```

### VPMADDUBSW (EVEX encoded versions)

(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] ← SaturateToSignedWord(SRC2[i+15:i+8]\* SRC1[i+15:i+8] + SRC2[i+7:i]\*SRC1[i+7:i])

  ELSE

    IF \*merging-masking\* ; merging-masking

      THEN \*DEST[i+15:i] remains unchanged\*

    ELSE \*zeroing-masking\* ; zeroing-masking

      DEST[i+15:i] = 0

  FI

FI;

ENDFOR;

DEST[MAX\_VL-1:VL] ← 0

### Intel C/C++ Compiler Intrinsic Equivalents

VPMADDUBSW \_\_m512i \_\_mm512\_mddubs\_epi16(\_\_m512i a, \_\_m512i b);  
 VPMADDUBSW \_\_m512i \_\_mm512\_mask\_mddubs\_epi16(\_\_m512i s, \_\_mmask32 k, \_\_m512i a, \_\_m512i b);  
 VPMADDUBSW \_\_m512i \_\_mm512\_maskz\_mddubs\_epi16(\_\_mmask32 k, \_\_m512i a, \_\_m512i b);  
 VPMADDUBSW \_\_m256i \_\_mm256\_mask\_mddubs\_epi16(\_\_m256i s, \_\_mmask16 k, \_\_m256i a, \_\_m256i b);  
 VPMADDUBSW \_\_m256i \_\_mm256\_maskz\_mddubs\_epi16(\_\_mmask16 k, \_\_m256i a, \_\_m256i b);  
 VPMADDUBSW \_\_m128i \_\_mm\_mask\_mddubs\_epi16(\_\_m128i s, \_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
 VPMADDUBSW \_\_m128i \_\_mm\_maskz\_mddubs\_epi16(\_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
 PMADDUBSW: \_\_m64 \_\_mm\_maddubs\_pi16(\_\_m64 a, \_\_m64 b)  
 (V)PMADDUBSW: \_\_m128i \_\_mm\_maddubs\_epi16(\_\_m128i a, \_\_m128i b)  
 VPMADDUBSW: \_\_m256i \_\_mm256\_maddubs\_epi16(\_\_m256i a, \_\_m256i b)

### SIMD Floating-Point Exceptions

None.

### Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.  
 EVEX-encoded instruction, see Exceptions Type E4NF.nb.

## PMADDWD—Multiply and Add Packed Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F F5 /r <sup>1</sup> PMADDWD <i>mm</i> , <i>mm/m64</i>	RM	V/V	MMX	Multiply the packed words in <i>mm</i> by the packed words in <i>mm/m64</i> , add adjacent doubleword results, and store in <i>mm</i> .
66 0F F5 /r PMADDWD <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE2	Multiply the packed word integers in <i>xmm1</i> by the packed word integers in <i>xmm2/m128</i> , add adjacent doubleword results, and store in <i>xmm1</i> .
VEX.NDS.128.66.0F.WIG F5 /r VPMADDWD <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Multiply the packed word integers in <i>xmm2</i> by the packed word integers in <i>xmm3/m128</i> , add adjacent doubleword results, and store in <i>xmm1</i> .
VEX.NDS.256.66.0F.WIG F5 /r VPMADDWD <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	RVM	V/V	AVX2	Multiply the packed word integers in <i>ymm2</i> by the packed word integers in <i>ymm3/m256</i> , add adjacent doubleword results, and store in <i>ymm1</i> .
EVEX.NDS.128.66.0F.WIG F5 /r VPMADDWD <i>xmm1</i> {k1}{z}, <i>xmm2</i> , <i>xmm3/m128</i>	FVM	V/V	AVX512VL AVX512BW	Multiply the packed word integers in <i>xmm2</i> by the packed word integers in <i>xmm3/m128</i> , add adjacent doubleword results, and store in <i>xmm1</i> under writemask k1.
EVEX.NDS.256.66.0F.WIG F5 /r VPMADDWD <i>ymm1</i> {k1}{z}, <i>ymm2</i> , <i>ymm3/m256</i>	FVM	V/V	AVX512VL AVX512BW	Multiply the packed word integers in <i>ymm2</i> by the packed word integers in <i>ymm3/m256</i> , add adjacent doubleword results, and store in <i>ymm1</i> under writemask k1.
EVEX.NDS.512.66.0F.WIG F5 /r VPMADDWD <i>zmm1</i> {k1}{z}, <i>zmm2</i> , <i>zmm3/m512</i>	FVM	V/V	AVX512BW	Multiply the packed word integers in <i>zmm2</i> by the packed word integers in <i>zmm3/m512</i> , add adjacent doubleword results, and store in <i>zmm1</i> under writemask k1.

### NOTES:

1. See note in Section 2.4, “AVX and SSE Instruction Exception Specification” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FVM	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Multiplies the individual signed words of the destination operand (first operand) by the corresponding signed words of the source operand (second operand), producing temporary signed, doubleword results. The adjacent doubleword results are then summed and stored in the destination operand. For example, the corresponding low-order words (15-0) and (31-16) in the source and destination operands are multiplied by one another and the doubleword results are added together and stored in the low doubleword of the destination register (31-0). The same operation is performed on the other pairs of adjacent words. (Figure 4-7 shows this operation when using 64-bit operands).

The (V)PMADDWD instruction wraps around only in one situation: when the 2 pairs of words being operated on in a group are all 8000H. In this case, the result wraps around to 80000000H.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE version: The first source and destination operands are MMX registers. The second source operand is an MMX register or a 64-bit memory location.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers.

EVEX.512 encoded version: The second source operand can be an ZMM register or a 512-bit memory location. The first source and destination operands are ZMM registers.

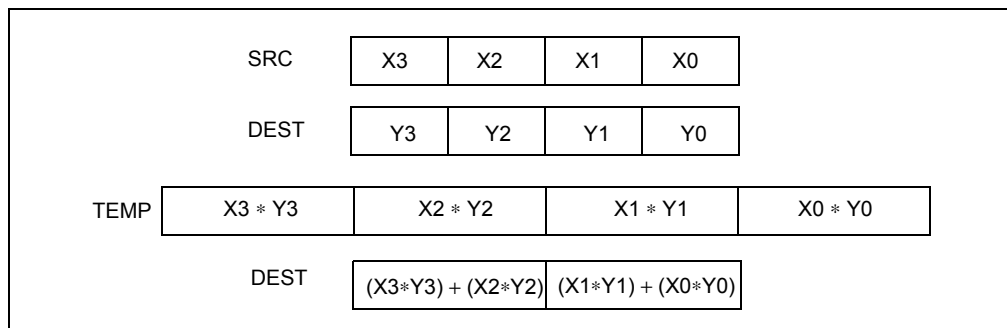


Figure 4-7. PMADDWD Execution Model Using 64-bit Operands

## Operation

### PMADDWD (with 64-bit operands)

$$\text{DEST}[31:0] \leftarrow (\text{DEST}[15:0] * \text{SRC}[15:0]) + (\text{DEST}[31:16] * \text{SRC}[31:16]);$$

$$\text{DEST}[63:32] \leftarrow (\text{DEST}[47:32] * \text{SRC}[47:32]) + (\text{DEST}[63:48] * \text{SRC}[63:48]);$$

### PMADDWD (with 128-bit operands)

$$\text{DEST}[31:0] \leftarrow (\text{DEST}[15:0] * \text{SRC}[15:0]) + (\text{DEST}[31:16] * \text{SRC}[31:16]);$$

$$\text{DEST}[63:32] \leftarrow (\text{DEST}[47:32] * \text{SRC}[47:32]) + (\text{DEST}[63:48] * \text{SRC}[63:48]);$$

$$\text{DEST}[95:64] \leftarrow (\text{DEST}[79:64] * \text{SRC}[79:64]) + (\text{DEST}[95:80] * \text{SRC}[95:80]);$$

$$\text{DEST}[127:96] \leftarrow (\text{DEST}[111:96] * \text{SRC}[111:96]) + (\text{DEST}[127:112] * \text{SRC}[127:112]);$$

### VPMADDWD (VEX.128 encoded version)

$$\text{DEST}[31:0] \leftarrow (\text{SRC1}[15:0] * \text{SRC2}[15:0]) + (\text{SRC1}[31:16] * \text{SRC2}[31:16])$$

$$\text{DEST}[63:32] \leftarrow (\text{SRC1}[47:32] * \text{SRC2}[47:32]) + (\text{SRC1}[63:48] * \text{SRC2}[63:48])$$

$$\text{DEST}[95:64] \leftarrow (\text{SRC1}[79:64] * \text{SRC2}[79:64]) + (\text{SRC1}[95:80] * \text{SRC2}[95:80])$$

$$\text{DEST}[127:96] \leftarrow (\text{SRC1}[111:96] * \text{SRC2}[111:96]) + (\text{SRC1}[127:112] * \text{SRC2}[127:112])$$

$$\text{DEST}[\text{VLMAX}-1:128] \leftarrow 0$$

### VPMADDWD (VEX.256 encoded version)

$$\text{DEST}[31:0] \leftarrow (\text{SRC1}[15:0] * \text{SRC2}[15:0]) + (\text{SRC1}[31:16] * \text{SRC2}[31:16])$$

$$\text{DEST}[63:32] \leftarrow (\text{SRC1}[47:32] * \text{SRC2}[47:32]) + (\text{SRC1}[63:48] * \text{SRC2}[63:48])$$

$$\text{DEST}[95:64] \leftarrow (\text{SRC1}[79:64] * \text{SRC2}[79:64]) + (\text{SRC1}[95:80] * \text{SRC2}[95:80])$$

$$\text{DEST}[127:96] \leftarrow (\text{SRC1}[111:96] * \text{SRC2}[111:96]) + (\text{SRC1}[127:112] * \text{SRC2}[127:112])$$

```

DEST[159:128] ← (SRC1[143:128] * SRC2[143:128]) + (SRC1[159:144] * SRC2[159:144])
DEST[191:160] ← (SRC1[175:160] * SRC2[175:160]) + (SRC1[191:176] * SRC2[191:176])
DEST[223:192] ← (SRC1[207:192] * SRC2[207:192]) + (SRC1[223:208] * SRC2[223:208])
DEST[255:224] ← (SRC1[239:224] * SRC2[239:224]) + (SRC1[255:240] * SRC2[255:240])
DEST[VLMAX-1:256] ← 0

```

**VPMADDWD (EVEX encoded versions)**

(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[j+31:i] ← (SRC2[j+31:i+16]* SRC1[j+31:i+16]) + (SRC2[i+15:i]*SRC1[i+15:i])
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[j+31:i] remains unchanged*
    ELSE *zeroing-masking* ; zeroing-masking
      DEST[j+31:i] = 0
    FI
  FI;
ENDFOR;
DEST[MAX_VL-1:VL] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VPMADDWD __m512i __mm512_mdd_epi16(__m512i a, __m512i b);
VPMADDWD __m512i __mm512_mask_mdd_epi16(__m512i s, __mmask16 k, __m512i a, __m512i b);
VPMADDWD __m512i __mm512_maskz_mdd_epi16(__mmask16 k, __m512i a, __m512i b);
VPMADDWD __m256i __mm256_mask_mdd_epi16(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPMADDWD __m256i __mm256_maskz_mdd_epi16(__mmask8 k, __m256i a, __m256i b);
VPMADDWD __m128i __mm_mask_mdd_epi16(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMADDWD __m128i __mm_maskz_mdd_epi16(__mmask8 k, __m128i a, __m128i b);
PMADDWD: __m64 __mm_madd_pi16(__m64 m1, __m64 m2)
(V)PMADDWD: __m128i __mm_madd_epi16 (__m128i a, __m128i b)
VPMADDWD: __m256i __mm256_madd_epi16 (__m256i a, __m256i b)

```

**Flags Affected**

None.

**Numeric Exceptions**

None.

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4NF.nb.

## PMAXSB/PMAXSW/PMAXSD/PMAXSQ—Maximum of Packed Signed Integers

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F EE /r <sup>1</sup> PMAXSW mm1, mm2/m64	RM	V/V	SSE	Compare signed word integers in mm2/m64 and mm1 and return maximum values.
66 0F 38 3C /r PMAXSB xmm1, xmm2/m128	RM	V/V	SSE4_1	Compare packed signed byte integers in xmm1 and xmm2/m128 and store packed maximum values in xmm1.
66 0F EE /r PMAXSW xmm1, xmm2/m128	RM	V/V	SSE2	Compare packed signed word integers in xmm2/m128 and xmm1 and stores maximum packed values in xmm1.
66 0F 38 3D /r PMAXSD xmm1, xmm2/m128	RM	V/V	SSE4_1	Compare packed signed dword integers in xmm1 and xmm2/m128 and store packed maximum values in xmm1.
VEX.NDS.128.66.0F38.WIG 3C /r VPMAXSB xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Compare packed signed byte integers in xmm2 and xmm3/m128 and store packed maximum values in xmm1.
VEX.NDS.128.66.0F.WIG EE /r VPMAXSW xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Compare packed signed word integers in xmm3/m128 and xmm2 and store packed maximum values in xmm1.
VEX.NDS.128.66.0F38.WIG 3D /r VPMAXSD xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Compare packed signed dword integers in xmm2 and xmm3/m128 and store packed maximum values in xmm1.
VEX.NDS.256.66.0F38.WIG 3C /r VPMAXSB ymm1, ymm2, ymm3/m256	RVM	V/V	AVX2	Compare packed signed byte integers in ymm2 and ymm3/m256 and store packed maximum values in ymm1.
VEX.NDS.256.66.0F.WIG EE /r VPMAXSW ymm1, ymm2, ymm3/m256	RVM	V/V	AVX2	Compare packed signed word integers in ymm3/m256 and ymm2 and store packed maximum values in ymm1.
VEX.NDS.256.66.0F38.WIG 3D /r VPMAXSD ymm1, ymm2, ymm3/m256	RVM	V/V	AVX2	Compare packed signed dword integers in ymm2 and ymm3/m256 and store packed maximum values in ymm1.
EVEX.NDS.128.66.0F38.WIG 3C /r VPMAXSB xmm1{k1}{z}, xmm2, xmm3/m128	FVM	V/V	AVX512VL AVX512BW	Compare packed signed byte integers in xmm2 and xmm3/m128 and store packed maximum values in xmm1 under writemask k1.
EVEX.NDS.256.66.0F38.WIG 3C /r VPMAXSB ymm1{k1}{z}, ymm2, ymm3/m256	FVM	V/V	AVX512VL AVX512BW	Compare packed signed byte integers in ymm2 and ymm3/m256 and store packed maximum values in ymm1 under writemask k1.
EVEX.NDS.512.66.0F38.WIG 3C /r VPMAXSB zmm1{k1}{z}, zmm2, zmm3/m512	FVM	V/V	AVX512BW	Compare packed signed byte integers in zmm2 and zmm3/m512 and store packed maximum values in zmm1 under writemask k1.
EVEX.NDS.128.66.0F.WIG EE /r VPMAXSW xmm1{k1}{z}, xmm2, xmm3/m128	FVM	V/V	AVX512VL AVX512BW	Compare packed signed word integers in xmm2 and xmm3/m128 and store packed maximum values in xmm1 under writemask k1.
EVEX.NDS.256.66.0F.WIG EE /r VPMAXSW ymm1{k1}{z}, ymm2, ymm3/m256	FVM	V/V	AVX512VL AVX512BW	Compare packed signed word integers in ymm2 and ymm3/m256 and store packed maximum values in ymm1 under writemask k1.



Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.512.66.0F.WIG EE /r VPMAXSW zmm1{k1}{z}, zmm2, zmm3/m512	FVM	V/V	AVX512BW	Compare packed signed word integers in zmm2 and zmm3/m512 and store packed maximum values in zmm1 under writemask k1.
EVEX.NDS.128.66.0F38.W0 3D /r VPMAXSD xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	FV	V/V	AVX512VL AVX512F	Compare packed signed dword integers in xmm2 and xmm3/m128/m32bcst and store packed maximum values in xmm1 using writemask k1.
EVEX.NDS.256.66.0F38.W0 3D /r VPMAXSD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	FV	V/V	AVX512VL AVX512F	Compare packed signed dword integers in ymm2 and ymm3/m256/m32bcst and store packed maximum values in ymm1 using writemask k1.
EVEX.NDS.512.66.0F38.W0 3D /r VPMAXSD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	FV	V/V	AVX512F	Compare packed signed dword integers in zmm2 and zmm3/m512/m32bcst and store packed maximum values in zmm1 using writemask k1.
EVEX.NDS.128.66.0F38.W1 3D /r VPMAXSQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	FV	V/V	AVX512VL AVX512F	Compare packed signed qword integers in xmm2 and xmm3/m128/m64bcst and store packed maximum values in xmm1 using writemask k1.
EVEX.NDS.256.66.0F38.W1 3D /r VPMAXSQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	FV	V/V	AVX512VL AVX512F	Compare packed signed qword integers in ymm2 and ymm3/m256/m64bcst and store packed maximum values in ymm1 using writemask k1.
EVEX.NDS.512.66.0F38.W1 3D /r VPMAXSQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	FV	V/V	AVX512F	Compare packed signed qword integers in zmm2 and zmm3/m512/m64bcst and store packed maximum values in zmm1 using writemask k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FVM	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA
FV	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

#### Description

Performs a SIMD compare of the packed signed byte, word, dword or qword integers in the second source operand and the first source operand and returns the maximum value for each pair of integers to the destination operand.

Legacy SSE version PMAWSW: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand can be an MMX technology register.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAX\_VL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAX\_VL-1:128) of the corresponding destination register are zeroed.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers. Bits (MAX\_VL-1:256) of the corresponding destination register are zeroed.

EVEX encoded VPMAXSD/Q: The first source operand is a ZMM/YMM/XMM register; The second 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 32/64-bit memory location. The destination operand is conditionally updated based on writemask k1.

EVEX encoded VPMAXSB/W: The first source operand is a ZMM/YMM/XMM register; The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination operand is conditionally updated based on writemask k1.

## Operation

### PMAXSw (64-bit operands)

IF DEST[15:0] > SRC[15:0] THEN

DEST[15:0] ← DEST[15:0];

ELSE

DEST[15:0] ← SRC[15:0]; FI;

(\* Repeat operation for 2nd and 3rd words in source and destination operands \*)

IF DEST[63:48] > SRC[63:48] THEN

DEST[63:48] ← DEST[63:48];

ELSE

DEST[63:48] ← SRC[63:48]; FI;

### PMAXSB (128-bit Legacy SSE version)

IF DEST[7:0] > SRC[7:0] THEN

DEST[7:0] ← DEST[7:0];

ELSE

DEST[7:0] ← SRC[7:0]; FI;

(\* Repeat operation for 2nd through 15th bytes in source and destination operands \*)

IF DEST[127:120] > SRC[127:120] THEN

DEST[127:120] ← DEST[127:120];

ELSE

DEST[127:120] ← SRC[127:120]; FI;

DEST[MAX\_VL-1:128] (Unmodified)

### VPMAXSB (VEX.128 encoded version)

IF SRC1[7:0] > SRC2[7:0] THEN

DEST[7:0] ← SRC1[7:0];

ELSE

DEST[7:0] ← SRC2[7:0]; FI;

(\* Repeat operation for 2nd through 15th bytes in source and destination operands \*)

IF SRC1[127:120] > SRC2[127:120] THEN

DEST[127:120] ← SRC1[127:120];

ELSE

DEST[127:120] ← SRC2[127:120]; FI;

DEST[MAX\_VL-1:128] ← 0

### VPMAXSB (VEX.256 encoded version)

IF SRC1[7:0] > SRC2[7:0] THEN

DEST[7:0] ← SRC1[7:0];

ELSE

DEST[7:0] ← SRC2[7:0]; FI;

(\* Repeat operation for 2nd through 31st bytes in source and destination operands \*)

IF SRC1[255:248] > SRC2[255:248] THEN

DEST[255:248] ← SRC1[255:248];

```

ELSE
  DEST[255:248] ← SRC2[255:248]; FI;
DEST[MAX_VL-1:256] ← 0

```

**VPMASB (EVEX encoded versions)**

(KL, VL) = (16, 128), (32, 256), (64, 512)

FOR j ← 0 TO KL-1

i ← j \* 8

IF k1[j] OR \*no writemask\* THEN

IF SRC1[i+7:i] &gt; SRC2[i+7:i]

THEN DEST[i+7:i] ← SRC1[i+7:i];

ELSE DEST[i+7:i] ← SRC2[i+7:i];

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+7:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+7:i] ← 0

FI

FI;

ENDFOR;

DEST[MAX\_VL-1:VL] ← 0

**PMAWS (128-bit Legacy SSE version)**

IF DEST[15:0] &gt; SRC[15:0] THEN

DEST[15:0] ← DEST[15:0];

ELSE

DEST[15:0] ← SRC[15:0]; FI;

(\* Repeat operation for 2nd through 7th words in source and destination operands \*)

IF DEST[127:112] &gt; SRC[127:112] THEN

DEST[127:112] ← DEST[127:112];

ELSE

DEST[127:112] ← SRC[127:112]; FI;

DEST[MAX\_VL-1:128] (Unmodified)

**VPMASW (VEX.128 encoded version)**

IF SRC1[15:0] &gt; SRC2[15:0] THEN

DEST[15:0] ← SRC1[15:0];

ELSE

DEST[15:0] ← SRC2[15:0]; FI;

(\* Repeat operation for 2nd through 7th words in source and destination operands \*)

IF SRC1[127:112] &gt; SRC2[127:112] THEN

DEST[127:112] ← SRC1[127:112];

ELSE

DEST[127:112] ← SRC2[127:112]; FI;

DEST[MAX\_VL-1:128] ← 0

**VPMASW (VEX.256 encoded version)**

IF SRC1[15:0] &gt; SRC2[15:0] THEN

DEST[15:0] ← SRC1[15:0];

ELSE

DEST[15:0] ← SRC2[15:0]; FI;

(\* Repeat operation for 2nd through 15th words in source and destination operands \*)

IF SRC1[255:240] &gt; SRC2[255:240] THEN

```

    DEST[255:240] ← SRC1[255:240];
ELSE
    DEST[255:240] ← SRC2[255:240]; FI;
DEST[MAX_VL-1:256] ← 0
VPMAXSW (EVEX encoded versions)
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask* THEN
        IF SRC1[i+15:i] > SRC2[i+15:i]
            THEN DEST[i+15:i] ← SRC1[i+15:i];
            ELSE DEST[i+15:i] ← SRC2[i+15:i];
        FI;
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+15:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+15:i] ← 0
        FI
    FI;
ENDFOR;
DEST[MAX_VL-1:VL] ← 0

PMAxsd (128-bit Legacy SSE version)
IF DEST[31:0] > SRC[31:0] THEN
    DEST[31:0] ← DEST[31:0];
ELSE
    DEST[31:0] ← SRC[31:0]; FI;
(* Repeat operation for 2nd through 7th words in source and destination operands *)
IF DEST[127:96] > SRC[127:96] THEN
    DEST[127:96] ← DEST[127:96];
ELSE
    DEST[127:96] ← SRC[127:96]; FI;
DEST[MAX_VL-1:128] (Unmodified)

VPMAXSD (VEX.128 encoded version)
IF SRC1[31:0] > SRC2[31:0] THEN
    DEST[31:0] ← SRC1[31:0];
ELSE
    DEST[31:0] ← SRC2[31:0]; FI;
(* Repeat operation for 2nd through 3rd dwords in source and destination operands *)
IF SRC1[127:96] > SRC2[127:96] THEN
    DEST[127:96] ← SRC1[127:96];
ELSE
    DEST[127:96] ← SRC2[127:96]; FI;
DEST[MAX_VL-1:128] ← 0

VPMAXSD (VEX.256 encoded version)
IF SRC1[31:0] > SRC2[31:0] THEN
    DEST[31:0] ← SRC1[31:0];
ELSE
    DEST[31:0] ← SRC2[31:0]; FI;
(* Repeat operation for 2nd through 7th dwords in source and destination operands *)
IF SRC1[255:224] > SRC2[255:224] THEN

```

```

    DEST[255:224] ← SRC1[255:224];
ELSE
    DEST[255:224] ← SRC2[255:224]; FI;
DEST[MAX_VL-1:256] ← 0
VPMAXSD (EVEX encoded versions)
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask* THEN
        IF (EVEX.b = 1) AND (SRC2 *is memory*)
            THEN
                IF SRC1[i+31:i] > SRC2[31:0]
                    THEN DEST[i+31:i] ← SRC1[i+31:i];
                    ELSE DEST[i+31:i] ← SRC2[31:0];
                FI;
            ELSE
                IF SRC1[i+31:i] > SRC2[i+31:i]
                    THEN DEST[i+31:i] ← SRC1[i+31:i];
                    ELSE DEST[i+31:i] ← SRC2[i+31:i];
                FI;
            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[MAX_VL-1:VL] ← 0

```

```

VPMAXSQ (EVEX encoded versions)
(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) AND (SRC2 *is memory*)
            THEN
                IF SRC1[i+63:i] > SRC2[63:0]
                    THEN DEST[i+63:i] ← SRC1[i+63:i];
                    ELSE DEST[i+63:i] ← SRC2[63:0];
                FI;
            ELSE
                IF SRC1[i+63:i] > SRC2[i+63:i]
                    THEN DEST[i+63:i] ← SRC1[i+63:i];
                    ELSE DEST[i+63:i] ← SRC2[i+63:i];
                FI;
            FI;
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
            ELSE ; zeroing-masking
                THEN DEST[i+63:i] ← 0
            FI
        FI;
    ENDFOR
    FI;

```

ENDFOR;  
 DEST[MAX\_VL-1:VL] ← 0

### Intel C/C++ Compiler Intrinsic Equivalent

VPMSB \_\_m512i \_\_mm512\_max\_epi8( \_\_m512i a, \_\_m512i b);  
 VPMSB \_\_m512i \_\_mm512\_mask\_max\_epi8( \_\_m512i s, \_\_mmask64 k, \_\_m512i a, \_\_m512i b);  
 VPMSB \_\_m512i \_\_mm512\_maskz\_max\_epi8( \_\_mmask64 k, \_\_m512i a, \_\_m512i b);  
 VPMASW \_\_m512i \_\_mm512\_max\_epi16( \_\_m512i a, \_\_m512i b);  
 VPMASW \_\_m512i \_\_mm512\_mask\_max\_epi16( \_\_m512i s, \_\_mmask32 k, \_\_m512i a, \_\_m512i b);  
 VPMASW \_\_m512i \_\_mm512\_maskz\_max\_epi16( \_\_mmask32 k, \_\_m512i a, \_\_m512i b);  
 VPMASB \_\_m256i \_\_mm256\_mask\_max\_epi8( \_\_m256i s, \_\_mmask32 k, \_\_m256i a, \_\_m256i b);  
 VPMASB \_\_m256i \_\_mm256\_maskz\_max\_epi8( \_\_mmask32 k, \_\_m256i a, \_\_m256i b);  
 VPMASW \_\_m256i \_\_mm256\_mask\_max\_epi16( \_\_m256i s, \_\_mmask16 k, \_\_m256i a, \_\_m256i b);  
 VPMASW \_\_m256i \_\_mm256\_maskz\_max\_epi16( \_\_mmask16 k, \_\_m256i a, \_\_m256i b);  
 VPMASB \_\_m128i \_\_mm\_mask\_max\_epi8( \_\_m128i s, \_\_mmask16 k, \_\_m128i a, \_\_m128i b);  
 VPMASB \_\_m128i \_\_mm\_maskz\_max\_epi8( \_\_mmask16 k, \_\_m128i a, \_\_m128i b);  
 VPMASW \_\_m128i \_\_mm\_mask\_max\_epi16( \_\_m128i s, \_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
 VPMASW \_\_m128i \_\_mm\_maskz\_max\_epi16( \_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
 VPMASD \_\_m256i \_\_mm256\_mask\_max\_epi32( \_\_m256i s, \_\_mmask16 k, \_\_m256i a, \_\_m256i b);  
 VPMASD \_\_m256i \_\_mm256\_maskz\_max\_epi32( \_\_mmask16 k, \_\_m256i a, \_\_m256i b);  
 VPMASQ \_\_m256i \_\_mm256\_mask\_max\_epi64( \_\_m256i s, \_\_mmask8 k, \_\_m256i a, \_\_m256i b);  
 VPMASQ \_\_m256i \_\_mm256\_maskz\_max\_epi64( \_\_mmask8 k, \_\_m256i a, \_\_m256i b);  
 VPMASD \_\_m128i \_\_mm\_mask\_max\_epi32( \_\_m128i s, \_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
 VPMASD \_\_m128i \_\_mm\_maskz\_max\_epi32( \_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
 VPMASQ \_\_m128i \_\_mm\_mask\_max\_epi64( \_\_m128i s, \_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
 VPMASQ \_\_m128i \_\_mm\_maskz\_max\_epu64( \_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
 VPMASD \_\_m512i \_\_mm512\_max\_epi32( \_\_m512i a, \_\_m512i b);  
 VPMASD \_\_m512i \_\_mm512\_mask\_max\_epi32( \_\_m512i s, \_\_mmask16 k, \_\_m512i a, \_\_m512i b);  
 VPMASD \_\_m512i \_\_mm512\_maskz\_max\_epi32( \_\_mmask16 k, \_\_m512i a, \_\_m512i b);  
 VPMASQ \_\_m512i \_\_mm512\_max\_epi64( \_\_m512i a, \_\_m512i b);  
 VPMASQ \_\_m512i \_\_mm512\_mask\_max\_epi64( \_\_m512i s, \_\_mmask8 k, \_\_m512i a, \_\_m512i b);  
 VPMASQ \_\_m512i \_\_mm512\_maskz\_max\_epi64( \_\_mmask8 k, \_\_m512i a, \_\_m512i b);  
 (V)PMSB \_\_m128i \_\_mm\_max\_epi8 ( \_\_m128i a, \_\_m128i b);  
 (V)PMASW \_\_m128i \_\_mm\_max\_epi16 ( \_\_m128i a, \_\_m128i b);  
 (V)PMASD \_\_m128i \_\_mm\_max\_epi32 ( \_\_m128i a, \_\_m128i b);  
 VPMASB \_\_m256i \_\_mm256\_max\_epi8 ( \_\_m256i a, \_\_m256i b);  
 VPMASW \_\_m256i \_\_mm256\_max\_epi16 ( \_\_m256i a, \_\_m256i b);  
 VPMASD \_\_m256i \_\_mm256\_max\_epi32 ( \_\_m256i a, \_\_m256i b);  
 PMASW: \_\_m64 \_\_mm\_max\_pi16( \_\_m64 a, \_\_m64 b)

### SIMD Floating-Point Exceptions

None

### Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.  
 EVEX-encoded VPMASD/Q, see Exceptions Type E4.  
 EVEX-encoded VPMASB/W, see Exceptions Type E4.nb.

## PMAXUB/PMAXUW—Maximum of Packed Unsigned Integers

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F DE /r <sup>1</sup> PMAXUB mm1, mm2/m64	RM	V/V	SSE	Compare unsigned byte integers in mm2/m64 and mm1 and returns maximum values.
66 0F DE /r PMAXUB xmm1, xmm2/m128	RM	V/V	SSE2	Compare packed unsigned byte integers in xmm1 and xmm2/m128 and store packed maximum values in xmm1.
66 0F 38 3E/r PMAXUW xmm1, xmm2/m128	RM	V/V	SSE4_1	Compare packed unsigned word integers in xmm2/m128 and xmm1 and stores maximum packed values in xmm1.
VEX.NDS.128.66.0F DE /r VPMAXUB xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Compare packed unsigned byte integers in xmm2 and xmm3/m128 and store packed maximum values in xmm1.
VEX.NDS.128.66.0F38 3E/r VPMAXUW xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Compare packed unsigned word integers in xmm3/m128 and xmm2 and store maximum packed values in xmm1.
VEX.NDS.256.66.0F DE /r VPMAXUB ymm1, ymm2, ymm3/m256	RVM	V/V	AVX2	Compare packed unsigned byte integers in ymm2 and ymm3/m256 and store packed maximum values in ymm1.
VEX.NDS.256.66.0F38 3E/r VPMAXUW ymm1, ymm2, ymm3/m256	RVM	V/V	AVX2	Compare packed unsigned word integers in ymm3/m256 and ymm2 and store maximum packed values in ymm1.
EVEX.NDS.128.66.0F.WIG DE /r VPMAXUB xmm1{k1}{z}, xmm2, xmm3/m128	FVM	V/V	AVX512VL AVX512BW	Compare packed unsigned byte integers in xmm2 and xmm3/m128 and store packed maximum values in xmm1 under writemask k1.
EVEX.NDS.256.66.0F.WIG DE /r VPMAXUB ymm1{k1}{z}, ymm2, ymm3/m256	FVM	V/V	AVX512VL AVX512BW	Compare packed unsigned byte integers in ymm2 and ymm3/m256 and store packed maximum values in ymm1 under writemask k1.
EVEX.NDS.512.66.0F.WIG DE /r VPMAXUB zmm1{k1}{z}, zmm2, zmm3/m512	FVM	V/V	AVX512BW	Compare packed unsigned byte integers in zmm2 and zmm3/m512 and store packed maximum values in zmm1 under writemask k1.
EVEX.NDS.128.66.0F38.WIG 3E /r VPMAXUW xmm1{k1}{z}, xmm2, xmm3/m128	FVM	V/V	AVX512VL AVX512BW	Compare packed unsigned word integers in xmm2 and xmm3/m128 and store packed maximum values in xmm1 under writemask k1.
EVEX.NDS.256.66.0F38.WIG 3E /r VPMAXUW ymm1{k1}{z}, ymm2, ymm3/m256	FVM	V/V	AVX512VL AVX512BW	Compare packed unsigned word integers in ymm2 and ymm3/m256 and store packed maximum values in ymm1 under writemask k1.
EVEX.NDS.512.66.0F38.WIG 3E /r VPMAXUW zmm1{k1}{z}, zmm2, zmm3/m512	FVM	V/V	AVX512BW	Compare packed unsigned word integers in zmm2 and zmm3/m512 and store packed maximum values in zmm1 under writemask k1.

### NOTES:

1. See note in Section 2.4, “AVX and SSE Instruction Exception Specification” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

## Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FVM	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

## Description

Performs a SIMD compare of the packed unsigned byte, word integers in the second source operand and the first source operand and returns the maximum value for each pair of integers to the destination operand.

Legacy SSE version PMAXUB: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand can be an MMX technology register.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAX\_VL-1:128) of the corresponding destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAX\_VL-1:128) of the corresponding destination register are zeroed.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register; The second source operand is a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operand is conditionally updated based on writemask k1.

## Operation

**PMAXUB (64-bit operands)**

```
IF DEST[7:0] > SRC[7:0] THEN
    DEST[7:0] ← DEST[7:0];
ELSE
    DEST[7:0] ← SRC[7:0]; FI;
(* Repeat operation for 2nd through 7th bytes in source and destination operands *)
IF DEST[63:56] > SRC[63:56] THEN
    DEST[63:56] ← DEST[63:56];
ELSE
    DEST[63:56] ← SRC[63:56]; FI;
```

**PMAXUB (128-bit Legacy SSE version)**

```
IF DEST[7:0] > SRC[7:0] THEN
    DEST[7:0] ← DEST[7:0];
ELSE
    DEST[15:0] ← SRC[7:0]; FI;
(* Repeat operation for 2nd through 15th bytes in source and destination operands *)
IF DEST[127:120] > SRC[127:120] THEN
    DEST[127:120] ← DEST[127:120];
ELSE
    DEST[127:120] ← SRC[127:120]; FI;
DEST[MAX_VL-1:128] (Unmodified)
```

**VPMAXUB (VEX.128 encoded version)**

```
IF SRC1[7:0] > SRC2[7:0] THEN
    DEST[7:0] ← SRC1[7:0];
ELSE
    DEST[7:0] ← SRC2[7:0]; FI;
```



```
(* Repeat operation for 2nd through 15th bytes in source and destination operands *)
IF SRC1[127:120] > SRC2[127:120] THEN
    DEST[127:120] ← SRC1[127:120];
ELSE
    DEST[127:120] ← SRC2[127:120]; FI;
DEST[MAX_VL-1:128] ← 0
```

**VPMAXUB (VEX.256 encoded version)**

```
IF SRC1[7:0] > SRC2[7:0] THEN
    DEST[7:0] ← SRC1[7:0];
ELSE
    DEST[15:0] ← SRC2[7:0]; FI;
(* Repeat operation for 2nd through 31st bytes in source and destination operands *)
IF SRC1[255:248] > SRC2[255:248] THEN
    DEST[255:248] ← SRC1[255:248];
ELSE
    DEST[255:248] ← SRC2[255:248]; FI;
DEST[MAX_VL-1:128] ← 0
```

**VPMAXUB (EVEX encoded versions)**

(KL, VL) = (16, 128), (32, 256), (64, 512)

```
FOR j ← 0 TO KL-1
    i ← j * 8
    IF k1[j] OR *no writemask* THEN
        IF SRC1[i+7:i] > SRC2[i+7:i]
            THEN DEST[i+7:i] ← SRC1[i+7:i];
            ELSE DEST[i+7:i] ← SRC2[i+7:i];
        FI;
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+7:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+7:i] ← 0
        FI
    FI;
ENDFOR;
DEST[MAX_VL-1:VL] ← 0
```

**PMAXUW (128-bit Legacy SSE version)**

```
IF DEST[15:0] > SRC[15:0] THEN
    DEST[15:0] ← DEST[15:0];
ELSE
    DEST[15:0] ← SRC[15:0]; FI;
(* Repeat operation for 2nd through 7th words in source and destination operands *)
IF DEST[127:112] > SRC[127:112] THEN
    DEST[127:112] ← DEST[127:112];
ELSE
    DEST[127:112] ← SRC[127:112]; FI;
DEST[MAX_VL-1:128] (Unmodified)
```

**VPMAXUW (VEX.128 encoded version)**

```
IF SRC1[15:0] > SRC2[15:0] THEN
    DEST[15:0] ← SRC1[15:0];
ELSE
```

```

    DEST[15:0] ← SRC2[15:0]; FI;
(* Repeat operation for 2nd through 7th words in source and destination operands *)
IF SRC1[127:112] > SRC2[127:112] THEN
    DEST[127:112] ← SRC1[127:112];
ELSE
    DEST[127:112] ← SRC2[127:112]; FI;
DEST[MAX_VL-1:128] ← 0

```

**VPMAXUW (VEX.256 encoded version)**

```

IF SRC1[15:0] > SRC2[15:0] THEN
    DEST[15:0] ← SRC1[15:0];
ELSE
    DEST[15:0] ← SRC2[15:0]; FI;
(* Repeat operation for 2nd through 15th words in source and destination operands *)
IF SRC1[255:240] > SRC2[255:240] THEN
    DEST[255:240] ← SRC1[255:240];
ELSE
    DEST[255:240] ← SRC2[255:240]; FI;
DEST[MAX_VL-1:128] ← 0

```

**VPMAXUW (EVEX encoded versions)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

```

FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask* THEN
        IF SRC1[i+15:i] > SRC2[i+15:i]
            THEN DEST[i+15:i] ← SRC1[i+15:i];
            ELSE DEST[i+15:i] ← SRC2[i+15:i];
        FI;
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+15:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+15:i] ← 0
            FI
    FI;
ENDFOR;
DEST[MAX_VL-1:VL] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VPMAXUB __m512i __mm512_max_epu8( __m512i a, __m512i b);
VPMAXUB __m512i __mm512_mask_max_epu8(__m512i s, __mmask64 k, __m512i a, __m512i b);
VPMAXUB __m512i __mm512_maskz_max_epu8(__mmask64 k, __m512i a, __m512i b);
VPMAXUW __m512i __mm512_max_epu16( __m512i a, __m512i b);
VPMAXUW __m512i __mm512_mask_max_epu16(__m512i s, __mmask32 k, __m512i a, __m512i b);
VPMAXUW __m512i __mm512_maskz_max_epu16(__mmask32 k, __m512i a, __m512i b);
VPMAXUB __m256i __mm256_mask_max_epu8(__m256i s, __mmask32 k, __m256i a, __m256i b);
VPMAXUB __m256i __mm256_maskz_max_epu8(__mmask32 k, __m256i a, __m256i b);
VPMAXUW __m256i __mm256_mask_max_epu16(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPMAXUW __m256i __mm256_maskz_max_epu16(__mmask16 k, __m256i a, __m256i b);
VPMAXUB __m128i __mm_mask_max_epu8(__m128i s, __mmask16 k, __m128i a, __m128i b);
VPMAXUB __m128i __mm_maskz_max_epu8(__mmask16 k, __m128i a, __m128i b);
VPMAXUW __m128i __mm_mask_max_epu16(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMAXUW __m128i __mm_maskz_max_epu16(__mmask8 k, __m128i a, __m128i b);

```

(V)PMAXB \_\_m128i \_\_mm\_max\_epu8 (\_\_m128i a, \_\_m128i b);  
(V)PMAXUW \_\_m128i \_\_mm\_max\_epu16 (\_\_m128i a, \_\_m128i b);  
VPAXB \_\_m256i \_\_mm256\_max\_epu8 (\_\_m256i a, \_\_m256i b);  
VPAXUW \_\_m256i \_\_mm256\_max\_epu16 (\_\_m256i a, \_\_m256i b);  
PAXB: \_\_m64 \_\_mm\_max\_pu8(\_\_m64 a, \_\_m64 b);

### SIMD Floating-Point Exceptions

None

### Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.nb.

## PMAXUD/PMAXUQ—Maximum of Packed Unsigned Integers

Opcode/ Instruction	Op/ En	64/32 bitMode Support	CPUID Feature Flag	Description
66 0F 38 3F /r PMAXUD xmm1, xmm2/m128	RM	V/V	SSE4_1	Compare packed unsigned dword integers in xmm1 and xmm2/m128 and store packed maximum values in xmm1.
VEX.NDS.128.66.0F38.WIG 3F /r VPMAXUD xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Compare packed unsigned dword integers in xmm2 and xmm3/m128 and store packed maximum values in xmm1.
VEX.NDS.256.66.0F38.WIG 3F /r VPMAXUD ymm1, ymm2, ymm3/m256	RVM	V/V	AVX2	Compare packed unsigned dword integers in ymm2 and ymm3/m256 and store packed maximum values in ymm1.
EVEX.NDS.128.66.0F38.W0 3F /r VPMAXUD xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	FV	V/V	AVX512VL AVX512F	Compare packed unsigned dword integers in xmm2 and xmm3/m128/m32bcst and store packed maximum values in xmm1 under writemask k1.
EVEX.NDS.256.66.0F38.W0 3F /r VPMAXUD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	FV	V/V	AVX512VL AVX512F	Compare packed unsigned dword integers in ymm2 and ymm3/m256/m32bcst and store packed maximum values in ymm1 under writemask k1.
EVEX.NDS.512.66.0F38.W0 3F /r VPMAXUD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	FV	V/V	AVX512F	Compare packed unsigned dword integers in zmm2 and zmm3/m512/m32bcst and store packed maximum values in zmm1 under writemask k1.
EVEX.NDS.128.66.0F38.W1 3F /r VPMAXUQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	FV	V/V	AVX512VL AVX512F	Compare packed unsigned qword integers in xmm2 and xmm3/m128/m64bcst and store packed maximum values in xmm1 under writemask k1.
EVEX.NDS.256.66.0F38.W1 3F /r VPMAXUQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	FV	V/V	AVX512VL AVX512F	Compare packed unsigned qword integers in ymm2 and ymm3/m256/m64bcst and store packed maximum values in ymm1 under writemask k1.
EVEX.NDS.512.66.0F38.W1 3F /r VPMAXUQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	FV	V/V	AVX512F	Compare packed unsigned qword integers in zmm2 and zmm3/m512/m64bcst and store packed maximum values in zmm1 under writemask k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
FV	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

### Description

Performs a SIMD compare of the packed unsigned dword or qword integers in the second source operand and the first source operand and returns the maximum value for each pair of integers to the destination operand.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAX\_VL-1:128) of the corresponding destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAX\_VL-1:128) of the corresponding destination register are zeroed.

VEX.256 encoded version: The first source operand is a YMM register; The second source operand is a YMM register or 256-bit memory location. Bits (MAX\_VL-1:256) of the corresponding destination register are zeroed.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register; The second 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 32/64-bit memory location. The destination operand is conditionally updated based on writemask k1.

## Operation

### PMAXUD (128-bit Legacy SSE version)

```
IF DEST[31:0] > SRC[31:0] THEN
    DEST[31:0] ← DEST[31:0];
ELSE
    DEST[31:0] ← SRC[31:0]; FI;
(* Repeat operation for 2nd through 7th words in source and destination operands *)
IF DEST[127:96] > SRC[127:96] THEN
    DEST[127:96] ← DEST[127:96];
ELSE
    DEST[127:96] ← SRC[127:96]; FI;
DEST[MAX_VL-1:128] (Unmodified)
```

### VPMAXUD (VEX.128 encoded version)

```
IF SRC1[31:0] > SRC2[31:0] THEN
    DEST[31:0] ← SRC1[31:0];
ELSE
    DEST[31:0] ← SRC2[31:0]; FI;
(* Repeat operation for 2nd through 3rd dwords in source and destination operands *)
IF SRC1[127:96] > SRC2[127:96] THEN
    DEST[127:96] ← SRC1[127:96];
ELSE
    DEST[127:96] ← SRC2[127:96]; FI;
DEST[MAX_VL-1:128] ← 0
```

### VPMAXUD (VEX.256 encoded version)

```
IF SRC1[31:0] > SRC2[31:0] THEN
    DEST[31:0] ← SRC1[31:0];
ELSE
    DEST[31:0] ← SRC2[31:0]; FI;
(* Repeat operation for 2nd through 7th dwords in source and destination operands *)
IF SRC1[255:224] > SRC2[255:224] THEN
    DEST[255:224] ← SRC1[255:224];
ELSE
    DEST[255:224] ← SRC2[255:224]; FI;
DEST[MAX_VL-1:256] ← 0
```

**VPMAXUD (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask* THEN
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
      THEN
        IF SRC1[j+31:i] > SRC2[31:0]
          THEN DEST[j+31:i] ← SRC1[j+31:i];
          ELSE DEST[j+31:i] ← SRC2[31:0];
        FI;
      ELSE
        IF SRC1[j+31:i] > SRC2[i+31:i]
          THEN DEST[j+31:i] ← SRC1[j+31:i];
          ELSE DEST[j+31:i] ← SRC2[i+31:i];
        FI;
      FI;
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[j+31:i] remains unchanged*
      ELSE ; zeroing-masking
        THEN DEST[j+31:i] ← 0
      FI
    FI;
  ENDFOR;
  DEST[MAX_VL-1:VL] ← 0

```

**VPMAXUQ (EVEX encoded versions)**

(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) AND (SRC2 *is memory*)
      THEN
        IF SRC1[j+63:i] > SRC2[63:0]
          THEN DEST[j+63:i] ← SRC1[j+63:i];
          ELSE DEST[j+63:i] ← SRC2[63:0];
        FI;
      ELSE
        IF SRC1[j+31:i] > SRC2[i+31:i]
          THEN DEST[j+63:i] ← SRC1[j+63:i];
          ELSE DEST[j+63:i] ← SRC2[i+63:i];
        FI;
      FI;
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[j+63:i] remains unchanged*
      ELSE ; zeroing-masking
        THEN DEST[j+63:i] ← 0
      FI
    FI;
  ENDFOR;
  DEST[MAX_VL-1:VL] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

VPMAXUD \_\_m512i\_mm512\_max\_epu32(\_\_m512i a, \_\_m512i b);  
 VPMAXUD \_\_m512i\_mm512\_mask\_max\_epu32(\_\_m512i s, \_\_mmask16 k, \_\_m512i a, \_\_m512i b);  
 VPMAXUD \_\_m512i\_mm512\_maskz\_max\_epu32(\_\_mmask16 k, \_\_m512i a, \_\_m512i b);  
 VPMAXUQ \_\_m512i\_mm512\_max\_epu64(\_\_m512i a, \_\_m512i b);  
 VPMAXUQ \_\_m512i\_mm512\_mask\_max\_epu64(\_\_m512i s, \_\_mmask8 k, \_\_m512i a, \_\_m512i b);  
 VPMAXUQ \_\_m512i\_mm512\_maskz\_max\_epu64(\_\_mmask8 k, \_\_m512i a, \_\_m512i b);  
 VPMAXUD \_\_m256i\_mm256\_mask\_max\_epu32(\_\_m256i s, \_\_mmask16 k, \_\_m256i a, \_\_m256i b);  
 VPMAXUD \_\_m256i\_mm256\_maskz\_max\_epu32(\_\_mmask16 k, \_\_m256i a, \_\_m256i b);  
 VPMAXUQ \_\_m256i\_mm256\_mask\_max\_epu64(\_\_m256i s, \_\_mmask8 k, \_\_m256i a, \_\_m256i b);  
 VPMAXUQ \_\_m256i\_mm256\_maskz\_max\_epu64(\_\_mmask8 k, \_\_m256i a, \_\_m256i b);  
 VPMAXUD \_\_m128i\_mm\_mask\_max\_epu32(\_\_m128i s, \_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
 VPMAXUD \_\_m128i\_mm\_maskz\_max\_epu32(\_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
 VPMAXUQ \_\_m128i\_mm\_mask\_max\_epu64(\_\_m128i s, \_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
 VPMAXUQ \_\_m128i\_mm\_maskz\_max\_epu64(\_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
 (V)PMAXUD \_\_m128i\_mm\_max\_epu32 (\_\_m128i a, \_\_m128i b);  
 VPMAXUD \_\_m256i\_mm256\_max\_epu32 (\_\_m256i a, \_\_m256i b);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.

## PMINSB/PMINSW—Minimum of Packed Signed Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F EA /r <sup>1</sup> PMINSW mm1, mm2/m64	RM	V/V	SSE	Compare signed word integers in mm2/m64 and mm1 and return minimum values.
66 0F 38 38 /r PMINSB xmm1, xmm2/m128	RM	V/V	SSE4_1	Compare packed signed byte integers in xmm1 and xmm2/m128 and store packed minimum values in xmm1.
66 0F EA /r PMINSW xmm1, xmm2/m128	RM	V/V	SSE2	Compare packed signed word integers in xmm2/m128 and xmm1 and store packed minimum values in xmm1.
VEX.NDS.128.66.0F38 38 /r VPMINSB xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Compare packed signed byte integers in xmm2 and xmm3/m128 and store packed minimum values in xmm1.
VEX.NDS.128.66.0F EA /r VPMINSW xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Compare packed signed word integers in xmm3/m128 and xmm2 and return packed minimum values in xmm1.
VEX.NDS.256.66.0F38 38 /r VPMINSB ymm1, ymm2, ymm3/m256	RVM	V/V	AVX2	Compare packed signed byte integers in ymm2 and ymm3/m256 and store packed minimum values in ymm1.
VEX.NDS.256.66.0F EA /r VPMINSW ymm1, ymm2, ymm3/m256	RVM	V/V	AVX2	Compare packed signed word integers in ymm3/m256 and ymm2 and return packed minimum values in ymm1.
EVEX.NDS.128.66.0F38.WIG 38 /r VPMINSB xmm1{k1}{z}, xmm2, xmm3/m128	FVM	V/V	AVX512VL AVX512BW	Compare packed signed byte integers in xmm2 and xmm3/m128 and store packed minimum values in xmm1 under writemask k1.
EVEX.NDS.256.66.0F38.WIG 38 /r VPMINSB ymm1{k1}{z}, ymm2, ymm3/m256	FVM	V/V	AVX512VL AVX512BW	Compare packed signed byte integers in ymm2 and ymm3/m256 and store packed minimum values in ymm1 under writemask k1.
EVEX.NDS.512.66.0F38.WIG 38 /r VPMINSB zmm1{k1}{z}, zmm2, zmm3/m512	FVM	V/V	AVX512BW	Compare packed signed byte integers in zmm2 and zmm3/m512 and store packed minimum values in zmm1 under writemask k1.
EVEX.NDS.128.66.0F.WIG EA /r VPMINSW xmm1{k1}{z}, xmm2, xmm3/m128	FVM	V/V	AVX512VL AVX512BW	Compare packed signed word integers in xmm2 and xmm3/m128 and store packed minimum values in xmm1 under writemask k1.
EVEX.NDS.256.66.0F.WIG EA /r VPMINSW ymm1{k1}{z}, ymm2, ymm3/m256	FVM	V/V	AVX512VL AVX512BW	Compare packed signed word integers in ymm2 and ymm3/m256 and store packed minimum values in ymm1 under writemask k1.
EVEX.NDS.512.66.0F.WIG EA /r VPMINSW zmm1{k1}{z}, zmm2, zmm3/m512	FVM	V/V	AVX512BW	Compare packed signed word integers in zmm2 and zmm3/m512 and store packed minimum values in zmm1 under writemask k1.

### NOTES:

1. See note in Section 2.4, “AVX and SSE Instruction Exception Specification” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.



## Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FVM	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

## Description

Performs a SIMD compare of the packed signed byte, word, or dword integers in the second source operand and the first source operand and returns the minimum value for each pair of integers to the destination operand.

Legacy SSE version PMINSW: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand can be an MMX technology register.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAX\_VL-1:128) of the corresponding destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAX\_VL-1:128) of the corresponding destination register are zeroed.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register; The second source operand is a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operand is conditionally updated based on writemask k1.

## Operation

**PMINSW (64-bit operands)**

```
IF DEST[15:0] < SRC[15:0] THEN
    DEST[15:0] ← DEST[15:0];
ELSE
    DEST[15:0] ← SRC[15:0]; FI;
(* Repeat operation for 2nd and 3rd words in source and destination operands *)
IF DEST[63:48] < SRC[63:48] THEN
    DEST[63:48] ← DEST[63:48];
ELSE
    DEST[63:48] ← SRC[63:48]; FI;
```

**PMINSB (128-bit Legacy SSE version)**

```
IF DEST[7:0] < SRC[7:0] THEN
    DEST[7:0] ← DEST[7:0];
ELSE
    DEST[15:0] ← SRC[7:0]; FI;
(* Repeat operation for 2nd through 15th bytes in source and destination operands *)
IF DEST[127:120] < SRC[127:120] THEN
    DEST[127:120] ← DEST[127:120];
ELSE
    DEST[127:120] ← SRC[127:120]; FI;
DEST[MAX_VL-1:128] (Unmodified)
```

**VPMINSB (VEX.128 encoded version)**

```
IF SRC1[7:0] < SRC2[7:0] THEN
    DEST[7:0] ← SRC1[7:0];
ELSE
    DEST[7:0] ← SRC2[7:0]; FI;
```

```
(* Repeat operation for 2nd through 15th bytes in source and destination operands *)
IF SRC1[127:120] < SRC2[127:120] THEN
    DEST[127:120] ← SRC1[127:120];
ELSE
    DEST[127:120] ← SRC2[127:120]; FI;
DEST[MAX_VL-1:128] ← 0
```

**VPMINSB (VEX.256 encoded version)**

```
IF SRC1[7:0] < SRC2[7:0] THEN
    DEST[7:0] ← SRC1[7:0];
ELSE
    DEST[15:0] ← SRC2[7:0]; FI;
(* Repeat operation for 2nd through 31st bytes in source and destination operands *)
IF SRC1[255:248] < SRC2[255:248] THEN
    DEST[255:248] ← SRC1[255:248];
ELSE
    DEST[255:248] ← SRC2[255:248]; FI;
DEST[MAX_VL-1:256] ← 0
```

**VPMINSB (EVEX encoded versions)**

(KL, VL) = (16, 128), (32, 256), (64, 512)

```
FOR j ← 0 TO KL-1
    i ← j * 8
    IF k1[j] OR *no writemask* THEN
        IF SRC1[j+7:i] < SRC2[j+7:i]
            THEN DEST[j+7:i] ← SRC1[j+7:i];
            ELSE DEST[j+7:i] ← SRC2[j+7:i];
        FI;
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[j+7:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[j+7:i] ← 0
        FI
    FI;
ENDFOR;
DEST[MAX_VL-1:VL] ← 0
```

**PMINSW (128-bit Legacy SSE version)**

```
IF DEST[15:0] < SRC[15:0] THEN
    DEST[15:0] ← DEST[15:0];
ELSE
    DEST[15:0] ← SRC[15:0]; FI;
(* Repeat operation for 2nd through 7th words in source and destination operands *)
IF DEST[127:112] < SRC[127:112] THEN
    DEST[127:112] ← DEST[127:112];
ELSE
    DEST[127:112] ← SRC[127:112]; FI;
DEST[MAX_VL-1:128] (Unmodified)
```

**VPMINSW (VEX.128 encoded version)**

```
IF SRC1[15:0] < SRC2[15:0] THEN
    DEST[15:0] ← SRC1[15:0];
ELSE
```

```

    DEST[15:0] ← SRC2[15:0]; FI;
(* Repeat operation for 2nd through 7th words in source and destination operands *)
IF SRC1[127:112] < SRC2[127:112] THEN
    DEST[127:112] ← SRC1[127:112];
ELSE
    DEST[127:112] ← SRC2[127:112]; FI;
DEST[MAX_VL-1:128] ← 0

```

**VPMINSW (VEX.256 encoded version)**

```

IF SRC1[15:0] < SRC2[15:0] THEN
    DEST[15:0] ← SRC1[15:0];
ELSE
    DEST[15:0] ← SRC2[15:0]; FI;
(* Repeat operation for 2nd through 15th words in source and destination operands *)
IF SRC1[255:240] < SRC2[255:240] THEN
    DEST[255:240] ← SRC1[255:240];
ELSE
    DEST[255:240] ← SRC2[255:240]; FI;
DEST[MAX_VL-1:256] ← 0

```

**VPMINSW (EVEX encoded versions)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

```

FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask* THEN
        IF SRC1[i+15:i] < SRC2[i+15:i]
            THEN DEST[i+15:i] ← SRC1[i+15:i];
            ELSE DEST[i+15:i] ← SRC2[i+15:i];
        FI;
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+15:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+15:i] ← 0
            FI
    FI;
ENDFOR;
DEST[MAX_VL-1:VL] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VPMINSB __m512i __mm512_min_epi8( __m512i a, __m512i b);
VPMINSB __m512i __mm512_mask_min_epi8(__m512i s, __mmask64 k, __m512i a, __m512i b);
VPMINSB __m512i __mm512_maskz_min_epi8( __mmask64 k, __m512i a, __m512i b);
VPMINSW __m512i __mm512_min_epi16( __m512i a, __m512i b);
VPMINSW __m512i __mm512_mask_min_epi16(__m512i s, __mmask32 k, __m512i a, __m512i b);
VPMINSW __m512i __mm512_maskz_min_epi16( __mmask32 k, __m512i a, __m512i b);
VPMINSB __m256i __mm256_mask_min_epi8(__m256i s, __mmask32 k, __m256i a, __m256i b);
VPMINSB __m256i __mm256_maskz_min_epi8( __mmask32 k, __m256i a, __m256i b);
VPMINSW __m256i __mm256_mask_min_epi16(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPMINSW __m256i __mm256_maskz_min_epi16( __mmask16 k, __m256i a, __m256i b);
VPMINSB __m128i __mm_mask_min_epi8(__m128i s, __mmask16 k, __m128i a, __m128i b);
VPMINSB __m128i __mm_maskz_min_epi8( __mmask16 k, __m128i a, __m128i b);
VPMINSW __m128i __mm_mask_min_epi16(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMINSW __m128i __mm_maskz_min_epi16( __mmask8 k, __m128i a, __m128i b);
(V)VPMINSB __m128i __mm_min_epi8 ( __m128i a, __m128i b);

```

(V)PMINSW \_\_m128i \_mm\_min\_epi16 ( \_\_m128i a, \_\_m128i b)  
 VPMINSB \_\_m256i \_mm256\_min\_epi8 ( \_\_m256i a, \_\_m256i b);  
 VPMINSW \_\_m256i \_mm256\_min\_epi16 ( \_\_m256i a, \_\_m256i b)  
 PMINSW: \_\_m64 \_mm\_min\_pi16 ( \_\_m64 a, \_\_m64 b)

### SIMD Floating-Point Exceptions

None

### Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.nb.

#MF (64-bit operations only) If there is a pending x87 FPU exception.

## PMINSD/PMINSQ—Minimum of Packed Signed Integers

Opcode/ Instruction	Op/ En	64/32 bitMode Support	CPUID Feature Flag	Description
66 0F 38 39 /r PMINSD xmm1, xmm2/m128	RM	V/V	SSE4_1	Compare packed signed dword integers in xmm1 and xmm2/m128 and store packed minimum values in xmm1.
VEX.NDS.128.66.0F38.WIG 39 /r VPMINSD xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Compare packed signed dword integers in xmm2 and xmm3/m128 and store packed minimum values in xmm1.
VEX.NDS.256.66.0F38.WIG 39 /r VPMINSD ymm1, ymm2, ymm3/m256	RVM	V/V	AVX2	Compare packed signed dword integers in ymm2 and ymm3/m128 and store packed minimum values in ymm1.
EVEX.NDS.128.66.0F38.W0 39 /r VPMINSD xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	FV	V/V	AVX512VL AVX512F	Compare packed signed dword integers in xmm2 and xmm3/m128 and store packed minimum values in xmm1 under writemask k1.
EVEX.NDS.256.66.0F38.W0 39 /r VPMINSD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	FV	V/V	AVX512VL AVX512F	Compare packed signed dword integers in ymm2 and ymm3/m256 and store packed minimum values in ymm1 under writemask k1.
EVEX.NDS.512.66.0F38.W0 39 /r VPMINSD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	FV	V/V	AVX512F	Compare packed signed dword integers in zmm2 and zmm3/m512/m32bcst and store packed minimum values in zmm1 under writemask k1.
EVEX.NDS.128.66.0F38.W1 39 /r VPMINSQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	FV	V/V	AVX512VL AVX512F	Compare packed signed qword integers in xmm2 and xmm3/m128 and store packed minimum values in xmm1 under writemask k1.
EVEX.NDS.256.66.0F38.W1 39 /r VPMINSQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	FV	V/V	AVX512VL AVX512F	Compare packed signed qword integers in ymm2 and ymm3/m256 and store packed minimum values in ymm1 under writemask k1.
EVEX.NDS.512.66.0F38.W1 39 /r VPMINSQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	FV	V/V	AVX512F	Compare packed signed qword integers in zmm2 and zmm3/m512/m64bcst and store packed minimum values in zmm1 under writemask k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FV	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Performs a SIMD compare of the packed signed dword or qword integers in the second source operand and the first source operand and returns the minimum value for each pair of integers to the destination operand.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAX\_VL-1:128) of the corresponding destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAX\_VL-1:128) of the corresponding destination register are zeroed.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers. Bits (MAX\_VL-1:256) of the corresponding destination register are zeroed.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register; The second 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 32/64-bit memory location. The destination operand is conditionally updated based on writemask k1.

## Operation

### PMINSD (128-bit Legacy SSE version)

```
IF DEST[31:0] < SRC[31:0] THEN
    DEST[31:0] ← DEST[31:0];
ELSE
    DEST[31:0] ← SRC[31:0]; FI;
(* Repeat operation for 2nd through 7th words in source and destination operands *)
IF DEST[127:96] < SRC[127:96] THEN
    DEST[127:96] ← DEST[127:96];
ELSE
    DEST[127:96] ← SRC[127:96]; FI;
DEST[MAX_VL-1:128] (Unmodified)
```

### VPMINSD (VEX.128 encoded version)

```
IF SRC1[31:0] < SRC2[31:0] THEN
    DEST[31:0] ← SRC1[31:0];
ELSE
    DEST[31:0] ← SRC2[31:0]; FI;
(* Repeat operation for 2nd through 3rd dwords in source and destination operands *)
IF SRC1[127:96] < SRC2[127:96] THEN
    DEST[127:96] ← SRC1[127:96];
ELSE
    DEST[127:96] ← SRC2[127:96]; FI;
DEST[MAX_VL-1:128] ← 0
```

### VPMINSD (VEX.256 encoded version)

```
IF SRC1[31:0] < SRC2[31:0] THEN
    DEST[31:0] ← SRC1[31:0];
ELSE
    DEST[31:0] ← SRC2[31:0]; FI;
(* Repeat operation for 2nd through 7th dwords in source and destination operands *)
IF SRC1[255:224] < SRC2[255:224] THEN
    DEST[255:224] ← SRC1[255:224];
ELSE
    DEST[255:224] ← SRC2[255:224]; FI;
DEST[MAX_VL-1:256] ← 0
```

**VPMINSD (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\* THEN

IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

THEN

IF SRC1[i+31:i] &lt; SRC2[31:0]

THEN DEST[i+31:i] ← SRC1[i+31:i];

ELSE DEST[i+31:i] ← SRC2[31:0];

FI;

ELSE

IF SRC1[i+31:i] &lt; SRC2[i+31:i]

THEN DEST[i+31:i] ← SRC1[i+31:i];

ELSE DEST[i+31:i] ← SRC2[i+31:i];

FI;

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+31:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

ENDFOR;

DEST[MAX\_VL-1:VL] ← 0

**VPMINSQ (EVEX encoded versions)**

(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) AND (SRC2 \*is memory\*)

THEN

IF SRC1[i+63:i] &lt; SRC2[63:0]

THEN DEST[i+63:i] ← SRC1[i+63:i];

ELSE DEST[i+63:i] ← SRC2[63:0];

FI;

ELSE

IF SRC1[i+63:i] &lt; SRC2[i+63:i]

THEN DEST[i+63:i] ← SRC1[i+63:i];

ELSE DEST[i+63:i] ← SRC2[i+63:i];

FI;

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR;

DEST[MAX\_VL-1:VL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

VPMINSD \_\_m512i \_mm512\_min\_epi32( \_\_m512i a, \_\_m512i b);  
 VPMINSD \_\_m512i \_mm512\_mask\_min\_epi32( \_\_m512i s, \_\_mmask16 k, \_\_m512i a, \_\_m512i b);  
 VPMINSD \_\_m512i \_mm512\_maskz\_min\_epi32( \_\_mmask16 k, \_\_m512i a, \_\_m512i b);  
 VPMINSQ \_\_m512i \_mm512\_min\_epi64( \_\_m512i a, \_\_m512i b);  
 VPMINSQ \_\_m512i \_mm512\_mask\_min\_epi64( \_\_m512i s, \_\_mmask8 k, \_\_m512i a, \_\_m512i b);  
 VPMINSQ \_\_m512i \_mm512\_maskz\_min\_epi64( \_\_mmask8 k, \_\_m512i a, \_\_m512i b);  
 VPMINSD \_\_m256i \_mm256\_mask\_min\_epi32( \_\_m256i s, \_\_mmask16 k, \_\_m256i a, \_\_m256i b);  
 VPMINSD \_\_m256i \_mm256\_maskz\_min\_epi32( \_\_mmask16 k, \_\_m256i a, \_\_m256i b);  
 VPMINSQ \_\_m256i \_mm256\_mask\_min\_epi64( \_\_m256i s, \_\_mmask8 k, \_\_m256i a, \_\_m256i b);  
 VPMINSQ \_\_m256i \_mm256\_maskz\_min\_epi64( \_\_mmask8 k, \_\_m256i a, \_\_m256i b);  
 VPMINSD \_\_m128i \_mm\_mask\_min\_epi32( \_\_m128i s, \_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
 VPMINSD \_\_m128i \_mm\_maskz\_min\_epi32( \_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
 VPMINSQ \_\_m128i \_mm\_mask\_min\_epi64( \_\_m128i s, \_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
 VPMINSQ \_\_m128i \_mm\_maskz\_min\_epi64( \_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
 (V)PMINSD \_\_m128i \_mm\_min\_epi32( \_\_m128i a, \_\_m128i b);  
 VPMINSD \_\_m256i \_mm256\_min\_epi32( \_\_m256i a, \_\_m256i b);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.



## PMINUB/PMINUW—Minimum of Packed Unsigned Integers

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F DA /r <sup>1</sup> PMINUB mm1, mm2/m64	RM	V/V	SSE	Compare unsigned byte integers in mm2/m64 and mm1 and returns minimum values.
66 0F DA /r PMINUB xmm1, xmm2/m128	RM	V/V	SSE2	Compare packed unsigned byte integers in xmm1 and xmm2/m128 and store packed minimum values in xmm1.
66 0F 38 3A/r PMINUW xmm1, xmm2/m128	RM	V/V	SSE4_1	Compare packed unsigned word integers in xmm2/m128 and xmm1 and store packed minimum values in xmm1.
VEX.NDS.128.66.0F DA /r VPMINUB xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Compare packed unsigned byte integers in xmm2 and xmm3/m128 and store packed minimum values in xmm1.
VEX.NDS.128.66.0F38 3A/r VPMINUW xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Compare packed unsigned word integers in xmm3/m128 and xmm2 and return packed minimum values in xmm1.
VEX.NDS.256.66.0F DA /r VPMINUB ymm1, ymm2, ymm3/m256	RVM	V/V	AVX2	Compare packed unsigned byte integers in ymm2 and ymm3/m256 and store packed minimum values in ymm1.
VEX.NDS.256.66.0F38 3A/r VPMINUW ymm1, ymm2, ymm3/m256	RVM	V/V	AVX2	Compare packed unsigned word integers in ymm3/m256 and ymm2 and return packed minimum values in ymm1.
EVEX.NDS.128.66.0F DA /r VPMINUB xmm1 {k1}{z}, xmm2, xmm3/m128	FVM	V/V	AVX512VL AVX512BW	Compare packed unsigned byte integers in xmm2 and xmm3/m128 and store packed minimum values in xmm1 under writemask k1.
EVEX.NDS.256.66.0F DA /r VPMINUB ymm1 {k1}{z}, ymm2, ymm3/m256	FVM	V/V	AVX512VL AVX512BW	Compare packed unsigned byte integers in ymm2 and ymm3/m256 and store packed minimum values in ymm1 under writemask k1.
EVEX.NDS.512.66.0F DA /r VPMINUB zmm1 {k1}{z}, zmm2, zmm3/m512	FVM	V/V	AVX512BW	Compare packed unsigned byte integers in zmm2 and zmm3/m512 and store packed minimum values in zmm1 under writemask k1.
EVEX.NDS.128.66.0F38 3A/r VPMINUW xmm1{k1}{z}, xmm2, xmm3/m128	FVM	V/V	AVX512VL AVX512BW	Compare packed unsigned word integers in xmm3/m128 and xmm2 and return packed minimum values in xmm1 under writemask k1.
EVEX.NDS.256.66.0F38 3A/r VPMINUW ymm1{k1}{z}, ymm2, ymm3/m256	FVM	V/V	AVX512VL AVX512BW	Compare packed unsigned word integers in ymm3/m256 and ymm2 and return packed minimum values in ymm1 under writemask k1.
EVEX.NDS.512.66.0F38 3A/r VPMINUW zmm1{k1}{z}, zmm2, zmm3/m512	FVM	V/V	AVX512BW	Compare packed unsigned word integers in zmm3/m512 and zmm2 and return packed minimum values in zmm1 under writemask k1.

### NOTES:

1. See note in Section 2.4, “AVX and SSE Instruction Exception Specification” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

## Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FVM	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

## Description

Performs a SIMD compare of the packed unsigned byte or word integers in the second source operand and the first source operand and returns the minimum value for each pair of integers to the destination operand.

Legacy SSE version PMINUB: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand can be an MMX technology register.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAX\_VL-1:128) of the corresponding destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAX\_VL-1:128) of the corresponding destination register are zeroed.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register; The second source operand is a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operand is conditionally updated based on writemask k1.

## Operation

**PMINUB (for 64-bit operands)**

```
IF DEST[7:0] < SRC[7:0] THEN
    DEST[7:0] ← DEST[7:0];
ELSE
    DEST[7:0] ← SRC[7:0]; FI;
(* Repeat operation for 2nd through 7th bytes in source and destination operands *)
IF DEST[63:56] < SRC[63:56] THEN
    DEST[63:56] ← DEST[63:56];
ELSE
    DEST[63:56] ← SRC[63:56]; FI;
```

**PMINUB instruction for 128-bit operands:**

```
IF DEST[7:0] < SRC[7:0] THEN
    DEST[7:0] ← DEST[7:0];
ELSE
    DEST[15:0] ← SRC[7:0]; FI;
(* Repeat operation for 2nd through 15th bytes in source and destination operands *)
IF DEST[127:120] < SRC[127:120] THEN
    DEST[127:120] ← DEST[127:120];
ELSE
    DEST[127:120] ← SRC[127:120]; FI;
DEST[MAX_VL-1:128] (Unmodified)
```

**VPMINUB (VEX.128 encoded version)**

```
IF SRC1[7:0] < SRC2[7:0] THEN
    DEST[7:0] ← SRC1[7:0];
ELSE
    DEST[7:0] ← SRC2[7:0]; FI;
```

```
(* Repeat operation for 2nd through 15th bytes in source and destination operands *)
IF SRC1[127:120] < SRC2[127:120] THEN
    DEST[127:120] ← SRC1[127:120];
ELSE
    DEST[127:120] ← SRC2[127:120]; FI;
DEST[MAX_VL-1:128] ← 0
```

**VPMINUB (VEX.256 encoded version)**

```
IF SRC1[7:0] < SRC2[7:0] THEN
    DEST[7:0] ← SRC1[7:0];
ELSE
    DEST[15:0] ← SRC2[7:0]; FI;
(* Repeat operation for 2nd through 31st bytes in source and destination operands *)
IF SRC1[255:248] < SRC2[255:248] THEN
    DEST[255:248] ← SRC1[255:248];
ELSE
    DEST[255:248] ← SRC2[255:248]; FI;
DEST[MAX_VL-1:256] ← 0
```

**VPMINUB (EVEX encoded versions)**

(KL, VL) = (16, 128), (32, 256), (64, 512)

```
FOR j ← 0 TO KL-1
    i ← j * 8
    IF k1[j] OR *no writemask* THEN
        IF SRC1[i+7:i] < SRC2[i+7:i]
            THEN DEST[i+7:i] ← SRC1[i+7:i];
            ELSE DEST[i+7:i] ← SRC2[i+7:i];
        FI;
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+7:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+7:i] ← 0
        FI
    FI;
ENDFOR;
DEST[MAX_VL-1:VL] ← 0
```

**PMINUW instruction for 128-bit operands:**

```
IF DEST[15:0] < SRC[15:0] THEN
    DEST[15:0] ← DEST[15:0];
ELSE
    DEST[15:0] ← SRC[15:0]; FI;
(* Repeat operation for 2nd through 7th words in source and destination operands *)
IF DEST[127:112] < SRC[127:112] THEN
    DEST[127:112] ← DEST[127:112];
ELSE
    DEST[127:112] ← SRC[127:112]; FI;
DEST[MAX_VL-1:128] (Unmodified)
```

**VPMINUW (VEX.128 encoded version)**

```
IF SRC1[15:0] < SRC2[15:0] THEN
    DEST[15:0] ← SRC1[15:0];
ELSE
```

```

    DEST[15:0] ← SRC2[15:0]; FI;
(* Repeat operation for 2nd through 7th words in source and destination operands *)
IF SRC1[127:112] < SRC2[127:112] THEN
    DEST[127:112] ← SRC1[127:112];
ELSE
    DEST[127:112] ← SRC2[127:112]; FI;
DEST[MAX_VL-1:128] ← 0

```

**VPMINUW (VEX.256 encoded version)**

```

IF SRC1[15:0] < SRC2[15:0] THEN
    DEST[15:0] ← SRC1[15:0];
ELSE
    DEST[15:0] ← SRC2[15:0]; FI;
(* Repeat operation for 2nd through 15th words in source and destination operands *)
IF SRC1[255:240] < SRC2[255:240] THEN
    DEST[255:240] ← SRC1[255:240];
ELSE
    DEST[255:240] ← SRC2[255:240]; FI;
DEST[MAX_VL-1:256] ← 0

```

**VPMINUW (EVEX encoded versions)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j ← 0 TO KL-1

i ← j \* 16

IF k1[j] OR \*no writemask\* THEN

IF SRC1[i+15:i] &lt; SRC2[i+15:i]

THEN DEST[i+15:i] ← SRC1[i+15:i];

ELSE DEST[i+15:i] ← SRC2[i+15:i];

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+15:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+15:i] ← 0

FI

FI;

ENDFOR;

DEST[MAX\_VL-1:VL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

VPMINUB \_\_m512i\_mm512\_min\_epu8(\_\_m512i a, \_\_m512i b);

VPMINUB \_\_m512i\_mm512\_mask\_min\_epu8(\_\_m512i s, \_\_mmask64 k, \_\_m512i a, \_\_m512i b);

VPMINUB \_\_m512i\_mm512\_maskz\_min\_epu8(\_\_mmask64 k, \_\_m512i a, \_\_m512i b);

VPMINUW \_\_m512i\_mm512\_min\_epu16(\_\_m512i a, \_\_m512i b);

VPMINUW \_\_m512i\_mm512\_mask\_min\_epu16(\_\_m512i s, \_\_mmask32 k, \_\_m512i a, \_\_m512i b);

VPMINUW \_\_m512i\_mm512\_maskz\_min\_epu16(\_\_mmask32 k, \_\_m512i a, \_\_m512i b);

VPMINUB \_\_m256i\_mm256\_mask\_min\_epu8(\_\_m256i s, \_\_mmask32 k, \_\_m256i a, \_\_m256i b);

VPMINUB \_\_m256i\_mm256\_maskz\_min\_epu8(\_\_mmask32 k, \_\_m256i a, \_\_m256i b);

VPMINUW \_\_m256i\_mm256\_mask\_min\_epu16(\_\_m256i s, \_\_mmask16 k, \_\_m256i a, \_\_m256i b);

VPMINUW \_\_m256i\_mm256\_maskz\_min\_epu16(\_\_mmask16 k, \_\_m256i a, \_\_m256i b);

VPMINUB \_\_m128i\_mm\_mask\_min\_epu8(\_\_m128i s, \_\_mmask16 k, \_\_m128i a, \_\_m128i b);

VPMINUB \_\_m128i\_mm\_maskz\_min\_epu8(\_\_mmask16 k, \_\_m128i a, \_\_m128i b);

VPMINUW \_\_m128i\_mm\_mask\_min\_epu16(\_\_m128i s, \_\_mmask8 k, \_\_m128i a, \_\_m128i b);

VPMINUW \_\_m128i\_mm\_maskz\_min\_epu16(\_\_mmask8 k, \_\_m128i a, \_\_m128i b);

(V)PMINUB \_\_m128i \_mm\_min\_epu8 (\_\_m128i a, \_\_m128i b)  
(V)PMINUW \_\_m128i \_mm\_min\_epu16 (\_\_m128i a, \_\_m128i b);  
VPMINUB \_\_m256i \_mm256\_min\_epu8 (\_\_m256i a, \_\_m256i b)  
VPMINUW \_\_m256i \_mm256\_min\_epu16 (\_\_m256i a, \_\_m256i b);  
PMINUB: \_\_m64 \_m\_min\_pu8 (\_\_m64 a, \_\_m64 b)

#### SIMD Floating-Point Exceptions

None

#### Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.nb.

## PMINUD/PMINUQ—Minimum of Packed Unsigned Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 3B /r PMINUD xmm1, xmm2/m128	RM	V/V	SSE4_1	Compare packed unsigned dword integers in xmm1 and xmm2/m128 and store packed minimum values in xmm1.
VEX.NDS.128.66.0F38.WIG 3B /r VPMINUD xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Compare packed unsigned dword integers in xmm2 and xmm3/m128 and store packed minimum values in xmm1.
VEX.NDS.256.66.0F38.WIG 3B /r VPMINUD ymm1, ymm2, ymm3/m256	RVM	V/V	AVX2	Compare packed unsigned dword integers in ymm2 and ymm3/m256 and store packed minimum values in ymm1.
EVEX.NDS.128.66.0F38.W0 3B /r VPMINUD xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	FV	V/V	AVX512VL AVX512F	Compare packed unsigned dword integers in xmm2 and xmm3/m128/m32bcst and store packed minimum values in xmm1 under writemask k1.
EVEX.NDS.256.66.0F38.W0 3B /r VPMINUD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	FV	V/V	AVX512VL AVX512F	Compare packed unsigned dword integers in ymm2 and ymm3/m256/m32bcst and store packed minimum values in ymm1 under writemask k1.
EVEX.NDS.512.66.0F38.W0 3B /r VPMINUD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	FV	V/V	AVX512F	Compare packed unsigned dword integers in zmm2 and zmm3/m512/m32bcst and store packed minimum values in zmm1 under writemask k1.
EVEX.NDS.128.66.0F38.W1 3B /r VPMINUQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	FV	V/V	AVX512VL AVX512F	Compare packed unsigned qword integers in xmm2 and xmm3/m128/m64bcst and store packed minimum values in xmm1 under writemask k1.
EVEX.NDS.256.66.0F38.W1 3B /r VPMINUQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	FV	V/V	AVX512VL AVX512F	Compare packed unsigned qword integers in ymm2 and ymm3/m256/m64bcst and store packed minimum values in ymm1 under writemask k1.
EVEX.NDS.512.66.0F38.W1 3B /r VPMINUQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	FV	V/V	AVX512F	Compare packed unsigned qword integers in zmm2 and zmm3/m512/m64bcst and store packed minimum values in zmm1 under writemask k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FV	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Performs a SIMD compare of the packed unsigned dword/qword integers in the second source operand and the first source operand and returns the minimum value for each pair of integers to the destination operand.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAX\_VL-1:128) of the corresponding destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAX\_VL-1:128) of the corresponding destination register are zeroed.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers. Bits (MAX\_VL-1:256) of the corresponding destination register are zeroed.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register; The second 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 32/64-bit memory location. The destination operand is conditionally updated based on writemask k1.

## Operation

### PMINUD (128-bit Legacy SSE version)

PMINUD instruction for 128-bit operands:

IF DEST[31:0] < SRC[31:0] THEN  
DEST[31:0] ← DEST[31:0];

ELSE

DEST[31:0] ← SRC[31:0]; FI;

(\* Repeat operation for 2nd through 7th words in source and destination operands \*)

IF DEST[127:96] < SRC[127:96] THEN  
DEST[127:96] ← DEST[127:96];

ELSE

DEST[127:96] ← SRC[127:96]; FI;

DEST[MAX\_VL-1:128] (Unmodified)

### VPMINUD (VEX.128 encoded version)

VPMINUD instruction for 128-bit operands:

IF SRC1[31:0] < SRC2[31:0] THEN  
DEST[31:0] ← SRC1[31:0];

ELSE

DEST[31:0] ← SRC2[31:0]; FI;

(\* Repeat operation for 2nd through 3rd dwords in source and destination operands \*)

IF SRC1[127:96] < SRC2[127:96] THEN  
DEST[127:96] ← SRC1[127:96];

ELSE

DEST[127:96] ← SRC2[127:96]; FI;

DEST[MAX\_VL-1:128] ← 0

### VPMINUD (VEX.256 encoded version)

VPMINUD instruction for 128-bit operands:

IF SRC1[31:0] < SRC2[31:0] THEN  
DEST[31:0] ← SRC1[31:0];

ELSE

DEST[31:0] ← SRC2[31:0]; FI;

(\* Repeat operation for 2nd through 7th dwords in source and destination operands \*)

IF SRC1[255:224] < SRC2[255:224] THEN  
DEST[255:224] ← SRC1[255:224];

ELSE

DEST[255:224] ← SRC2[255:224]; FI;

DEST[MAX\_VL-1:256] ← 0

**VPMINUD (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\* THEN

IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

THEN

IF SRC1[j+31:i] &lt; SRC2[31:0]

THEN DEST[j+31:i] ← SRC1[j+31:i];

ELSE DEST[j+31:i] ← SRC2[31:0];

FI;

ELSE

IF SRC1[j+31:i] &lt; SRC2[i+31:i]

THEN DEST[j+31:i] ← SRC1[j+31:i];

ELSE DEST[j+31:i] ← SRC2[i+31:i];

FI;

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[j+31:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[j+31:i] ← 0

FI

FI;

ENDFOR;

DEST[MAX\_VL-1:VL] ← 0

**VPMINUQ (EVEX encoded versions)**

(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) AND (SRC2 \*is memory\*)

THEN

IF SRC1[j+63:i] &lt; SRC2[63:0]

THEN DEST[j+63:i] ← SRC1[j+63:i];

ELSE DEST[j+63:i] ← SRC2[63:0];

FI;

ELSE

IF SRC1[j+63:i] &lt; SRC2[i+63:i]

THEN DEST[j+63:i] ← SRC1[j+63:i];

ELSE DEST[j+63:i] ← SRC2[i+63:i];

FI;

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[j+63:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[j+63:i] ← 0

FI

FI;

ENDFOR;

DEST[MAX\_VL-1:VL] ← 0



**Intel C/C++ Compiler Intrinsic Equivalent**

VPMINUD \_\_m512i \_mm512\_min\_epu32( \_\_m512i a, \_\_m512i b);  
 VPMINUD \_\_m512i \_mm512\_mask\_min\_epu32(\_\_m512i s, \_\_mmask16 k, \_\_m512i a, \_\_m512i b);  
 VPMINUD \_\_m512i \_mm512\_maskz\_min\_epu32( \_\_mmask16 k, \_\_m512i a, \_\_m512i b);  
 VPMINUQ \_\_m512i \_mm512\_min\_epu64( \_\_m512i a, \_\_m512i b);  
 VPMINUQ \_\_m512i \_mm512\_mask\_min\_epu64(\_\_m512i s, \_\_mmask8 k, \_\_m512i a, \_\_m512i b);  
 VPMINUQ \_\_m512i \_mm512\_maskz\_min\_epu64( \_\_mmask8 k, \_\_m512i a, \_\_m512i b);  
 VPMINUD \_\_m256i \_mm256\_mask\_min\_epu32(\_\_m256i s, \_\_mmask16 k, \_\_m256i a, \_\_m256i b);  
 VPMINUD \_\_m256i \_mm256\_maskz\_min\_epu32( \_\_mmask16 k, \_\_m256i a, \_\_m256i b);  
 VPMINUQ \_\_m256i \_mm256\_mask\_min\_epu64(\_\_m256i s, \_\_mmask8 k, \_\_m256i a, \_\_m256i b);  
 VPMINUQ \_\_m256i \_mm256\_maskz\_min\_epu64( \_\_mmask8 k, \_\_m256i a, \_\_m256i b);  
 VPMINUD \_\_m128i \_mm\_mask\_min\_epu32(\_\_m128i s, \_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
 VPMINUD \_\_m128i \_mm\_maskz\_min\_epu32( \_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
 VPMINUQ \_\_m128i \_mm\_mask\_min\_epu64(\_\_m128i s, \_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
 VPMINUQ \_\_m128i \_mm\_maskz\_min\_epu64( \_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
 (V)PMINUD \_\_m128i \_mm\_min\_epu32 ( \_\_m128i a, \_\_m128i b);  
 VPMINUD \_\_m256i \_mm256\_min\_epu32 ( \_\_m256i a, \_\_m256i b);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.

## PMOVSX—Packed Move with Sign Extend

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0f 38 20 /r PMOVSXBW xmm1, xmm2/m64	RM	V/V	SSE4_1	Sign extend 8 packed 8-bit integers in the low 8 bytes of xmm2/m64 to 8 packed 16-bit integers in xmm1.
66 0f 38 21 /r PMOVSXBD xmm1, xmm2/m32	RM	V/V	SSE4_1	Sign extend 4 packed 8-bit integers in the low 4 bytes of xmm2/m32 to 4 packed 32-bit integers in xmm1.
66 0f 38 22 /r PMOVSXBQ xmm1, xmm2/m16	RM	V/V	SSE4_1	Sign extend 2 packed 8-bit integers in the low 2 bytes of xmm2/m16 to 2 packed 64-bit integers in xmm1.
66 0f 38 23/r PMOVSXWD xmm1, xmm2/m64	RM	V/V	SSE4_1	Sign extend 4 packed 16-bit integers in the low 8 bytes of xmm2/m64 to 4 packed 32-bit integers in xmm1.
66 0f 38 24 /r PMOVSXWQ xmm1, xmm2/m32	RM	V/V	SSE4_1	Sign extend 2 packed 16-bit integers in the low 4 bytes of xmm2/m32 to 2 packed 64-bit integers in xmm1.
66 0f 38 25 /r PMOVSXDQ xmm1, xmm2/m64	RM	V/V	SSE4_1	Sign extend 2 packed 32-bit integers in the low 8 bytes of xmm2/m64 to 2 packed 64-bit integers in xmm1.
VEX.128.66.0F38.WIG 20 /r VPMOVSXBW xmm1, xmm2/m64	RM	V/V	AVX	Sign extend 8 packed 8-bit integers in the low 8 bytes of xmm2/m64 to 8 packed 16-bit integers in xmm1.
VEX.128.66.0F38.WIG 21 /r VPMOVSXBD xmm1, xmm2/m32	RM	V/V	AVX	Sign extend 4 packed 8-bit integers in the low 4 bytes of xmm2/m32 to 4 packed 32-bit integers in xmm1.
VEX.128.66.0F38.WIG 22 /r VPMOVSXBQ xmm1, xmm2/m16	RM	V/V	AVX	Sign extend 2 packed 8-bit integers in the low 2 bytes of xmm2/m16 to 2 packed 64-bit integers in xmm1.
VEX.128.66.0F38.WIG 23 /r VPMOVSXWD xmm1, xmm2/m64	RM	V/V	AVX	Sign extend 4 packed 16-bit integers in the low 8 bytes of xmm2/m64 to 4 packed 32-bit integers in xmm1.
VEX.128.66.0F38.WIG 24 /r VPMOVSXWQ xmm1, xmm2/m32	RM	V/V	AVX	Sign extend 2 packed 16-bit integers in the low 4 bytes of xmm2/m32 to 2 packed 64-bit integers in xmm1.
VEX.128.66.0F38.WIG 25 /r VPMOVSXDQ xmm1, xmm2/m64	RM	V/V	AVX	Sign extend 2 packed 32-bit integers in the low 8 bytes of xmm2/m64 to 2 packed 64-bit integers in xmm1.
VEX.256.66.0F38.WIG 20 /r VPMOVSXBW ymm1, xmm2/m128	RM	V/V	AVX2	Sign extend 16 packed 8-bit integers in xmm2/m128 to 16 packed 16-bit integers in ymm1.
VEX.256.66.0F38.WIG 21 /r VPMOVSXBD ymm1, xmm2/m64	RM	V/V	AVX2	Sign extend 8 packed 8-bit integers in the low 8 bytes of xmm2/m64 to 8 packed 32-bit integers in ymm1.
VEX.256.66.0F38.WIG 22 /r VPMOVSXBQ ymm1, xmm2/m32	RM	V/V	AVX2	Sign extend 4 packed 8-bit integers in the low 4 bytes of xmm2/m32 to 4 packed 64-bit integers in ymm1.
VEX.256.66.0F38.WIG 23 /r VPMOVSXWD ymm1, xmm2/m128	RM	V/V	AVX2	Sign extend 8 packed 16-bit integers in the low 16 bytes of xmm2/m128 to 8 packed 32-bit integers in ymm1.

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.256.66.0F38.WIG 24 /r VPMOVSXWQ ymm1, xmm2/m64	RM	V/V	AVX2	Sign extend 4 packed 16-bit integers in the low 8 bytes of xmm2/m64 to 4 packed 64-bit integers in ymm1.
VEX.256.66.0F38.WIG 25 /r VPMOVSXDQ ymm1, xmm2/m128	RM	V/V	AVX2	Sign extend 4 packed 32-bit integers in the low 16 bytes of xmm2/m128 to 4 packed 64-bit integers in ymm1.
EVEX.128.66.0F38.WIG 20 /r VPMOVSXBW xmm1 {k1}{z}, xmm2/m64	HVM	V/V	AVX512VL AVX512BW	Sign extend 8 packed 8-bit integers in xmm2/m64 to 8 packed 16-bit integers in zmm1.
EVEX.256.66.0F38.WIG 20 /r VPMOVSXBW ymm1 {k1}{z}, xmm2/m128	HVM	V/V	AVX512VL AVX512BW	Sign extend 16 packed 8-bit integers in xmm2/m128 to 16 packed 16-bit integers in ymm1.
EVEX.512.66.0F38.WIG 20 /r VPMOVSXBW zmm1 {k1}{z}, ymm2/m256	HVM	V/V	AVX512BW	Sign extend 32 packed 8-bit integers in ymm2/m256 to 32 packed 16-bit integers in zmm1.
EVEX.128.66.0F38.WIG 21 /r VPMOVSXBD xmm1 {k1}{z}, xmm2/m32	QVM	V/V	AVX512VL AVX512F	Sign extend 4 packed 8-bit integers in the low 4 bytes of xmm2/m32 to 4 packed 32-bit integers in xmm1 subject to writemask k1.
EVEX.256.66.0F38.WIG 21 /r VPMOVSXBD ymm1 {k1}{z}, xmm2/m64	QVM	V/V	AVX512VL AVX512F	Sign extend 8 packed 8-bit integers in the low 8 bytes of xmm2/m64 to 8 packed 32-bit integers in ymm1 subject to writemask k1.
EVEX.512.66.0F38.WIG 21 /r VPMOVSXBD zmm1 {k1}{z}, xmm2/m128	QVM	V/V	AVX512F	Sign extend 16 packed 8-bit integers in the low 16 bytes of xmm2/m128 to 16 packed 32-bit integers in zmm1 subject to writemask k1.
EVEX.128.66.0F38.WIG 22 /r VPMOVSXBQ xmm1 {k1}{z}, xmm2/m16	OVM	V/V	AVX512VL AVX512F	Sign extend 2 packed 8-bit integers in the low 2 bytes of xmm2/m16 to 2 packed 64-bit integers in xmm1 subject to writemask k1.
EVEX.256.66.0F38.WIG 22 /r VPMOVSXBQ ymm1 {k1}{z}, xmm2/m32	OVM	V/V	AVX512VL AVX512F	Sign extend 4 packed 8-bit integers in the low 4 bytes of xmm2/m32 to 4 packed 64-bit integers in ymm1 subject to writemask k1.
EVEX.512.66.0F38.WIG 22 /r VPMOVSXBQ zmm1 {k1}{z}, xmm2/m64	OVM	V/V	AVX512F	Sign extend 8 packed 8-bit integers in the low 8 bytes of xmm2/m64 to 8 packed 64-bit integers in zmm1 subject to writemask k1.
EVEX.128.66.0F38.WIG 23 /r VPMOVSXWD xmm1 {k1}{z}, xmm2/m64	HVM	V/V	AVX512VL AVX512F	Sign extend 4 packed 16-bit integers in the low 8 bytes of ymm2/mem to 4 packed 32-bit integers in xmm1 subject to writemask k1.
EVEX.256.66.0F38.WIG 23 /r VPMOVSXWD ymm1 {k1}{z}, xmm2/m128	HVM	V/V	AVX512VL AVX512F	Sign extend 8 packed 16-bit integers in the low 16 bytes of ymm2/m128 to 8 packed 32-bit integers in ymm1 subject to writemask k1.

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.512.66.0F38.WIG 23 /r VPMOVSXWD zmm1 {k1}{z}, ymm2/m256	HVM	V/V	AVX512F	Sign extend 16 packed 16-bit integers in the low 32 bytes of ymm2/m256 to 16 packed 32-bit integers in zmm1 subject to writemask k1.
EVEX.128.66.0F38.WIG 24 /r VPMOVSXWQ xmm1 {k1}{z}, xmm2/m32	QVM	V/V	AVX512VL AVX512F	Sign extend 2 packed 16-bit integers in the low 4 bytes of xmm2/m32 to 2 packed 64-bit integers in xmm1 subject to writemask k1.
EVEX.256.66.0F38.WIG 24 /r VPMOVSXWQ ymm1 {k1}{z}, xmm2/m64	QVM	V/V	AVX512VL AVX512F	Sign extend 4 packed 16-bit integers in the low 8 bytes of xmm2/m64 to 4 packed 64-bit integers in ymm1 subject to writemask k1.
EVEX.512.66.0F38.WIG 24 /r VPMOVSXWQ zmm1 {k1}{z}, xmm2/m128	QVM	V/V	AVX512F	Sign extend 8 packed 16-bit integers in the low 16 bytes of xmm2/m128 to 8 packed 64-bit integers in zmm1 subject to writemask k1.
EVEX.128.66.0F38.W0 25 /r VPMOVSXDQ xmm1 {k1}{z}, xmm2/m64	HVM	V/V	AVX512VL AVX512F	Sign extend 2 packed 32-bit integers in the low 8 bytes of xmm2/m64 to 2 packed 64-bit integers in zmm1 using writemask k1.
EVEX.256.66.0F38.W0 25 /r VPMOVSXDQ ymm1 {k1}{z}, xmm2/m128	HVM	V/V	AVX512VL AVX512F	Sign extend 4 packed 32-bit integers in the low 16 bytes of xmm2/m128 to 4 packed 64-bit integers in zmm1 using writemask k1.
EVEX.512.66.0F38.W0 25 /r VPMOVSXDQ zmm1 {k1}{z}, ymm2/m256	HVM	V/V	AVX512F	Sign extend 8 packed 32-bit integers in the low 32 bytes of ymm2/m256 to 8 packed 64-bit integers in zmm1 using writemask k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
HVM, QVM, OVM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

#### Description

Legacy and VEX encoded versions: Packed byte, word, or dword integers in the low bytes of the source operand (second operand) are sign extended to word, dword, or quadword integers and stored in packed signed bytes the destination operand.

128-bit Legacy SSE version: Bits (MAX\_VL-1:128) of the corresponding destination register remain unchanged.

VEX.128 and EVEX.128 encoded versions: Bits (MAX\_VL-1:128) of the corresponding destination register are zeroed.

VEX.256 and EVEX.256 encoded versions: Bits (MAX\_VL-1:256) of the corresponding destination register are zeroed.

EVEX encoded versions: Packed byte, word or dword integers starting from the low bytes of the source operand (second operand) are sign extended to word, dword or quadword integers and stored to the destination operand under the writemask. The destination register is XMM, YMM or ZMM Register.

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

**Operation****Packed\_Sign\_Extend\_BYTE\_to\_WORD(DEST, SRC)**

```

DEST[15:0] ← SignExtend(SRC[7:0]);
DEST[31:16] ← SignExtend(SRC[15:8]);
DEST[47:32] ← SignExtend(SRC[23:16]);
DEST[63:48] ← SignExtend(SRC[31:24]);
DEST[79:64] ← SignExtend(SRC[39:32]);
DEST[95:80] ← SignExtend(SRC[47:40]);
DEST[111:96] ← SignExtend(SRC[55:48]);
DEST[127:112] ← SignExtend(SRC[63:56]);

```

**Packed\_Sign\_Extend\_BYTE\_to\_DWORD(DEST, SRC)**

```

DEST[31:0] ← SignExtend(SRC[7:0]);
DEST[63:32] ← SignExtend(SRC[15:8]);
DEST[95:64] ← SignExtend(SRC[23:16]);
DEST[127:96] ← SignExtend(SRC[31:24]);

```

**Packed\_Sign\_Extend\_BYTE\_to\_QWORD(DEST, SRC)**

```

DEST[63:0] ← SignExtend(SRC[7:0]);
DEST[127:64] ← SignExtend(SRC[15:8]);

```

**Packed\_Sign\_Extend\_WORD\_to\_DWORD(DEST, SRC)**

```

DEST[31:0] ← SignExtend(SRC[15:0]);
DEST[63:32] ← SignExtend(SRC[31:16]);
DEST[95:64] ← SignExtend(SRC[47:32]);
DEST[127:96] ← SignExtend(SRC[63:48]);

```

**Packed\_Sign\_Extend\_WORD\_to\_QWORD(DEST, SRC)**

```

DEST[63:0] ← SignExtend(SRC[15:0]);
DEST[127:64] ← SignExtend(SRC[31:16]);

```

**Packed\_Sign\_Extend\_DWORD\_to\_QWORD(DEST, SRC)**

```

DEST[63:0] ← SignExtend(SRC[31:0]);
DEST[127:64] ← SignExtend(SRC[63:32]);

```

**VPMOVSXBW (EVEX encoded versions)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

Packed\_Sign\_Extend\_BYTE\_to\_WORD(TMP\_DEST[127:0], SRC[63:0])

IF VL >= 256

Packed\_Sign\_Extend\_BYTE\_to\_WORD(TMP\_DEST[255:128], SRC[127:64])

FI;

IF VL >= 512

Packed\_Sign\_Extend\_BYTE\_to\_WORD(TMP\_DEST[383:256], SRC[191:128])

Packed\_Sign\_Extend\_BYTE\_to\_WORD(TMP\_DEST[511:384], SRC[255:192])

FI;

FOR j ← 0 TO KL-1

i ← j \* 16

IF k1[j] OR \*no writemask\*

THEN DEST[i+15:i] ← TEMP\_DEST[i+15:i]

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+15:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[i+15:i] ← 0

```

        FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

VPMOVSXBD (EVEX encoded versions)
(KL, VL) = (4, 128), (8, 256), (16, 512)
Packed_Sign_Extend_BYTE_to_DWORD(TMP_DEST[127:0], SRC[31:0])
IF VL >= 256
    Packed_Sign_Extend_BYTE_to_DWORD(TMP_DEST[255:128], SRC[63:32])
FI;
IF VL >= 512
    Packed_Sign_Extend_BYTE_to_DWORD(TMP_DEST[383:256], SRC[95:64])
    Packed_Sign_Extend_BYTE_to_DWORD(TMP_DEST[511:384], SRC[127:96])
FI;
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← TEMP_DEST[i+31:i]
    ELSE
        IF *merging-masking*           ; merging-masking
            THEN *DEST[i+31:i] remains unchanged*
        ELSE *zeroing-masking*         ; zeroing-masking
            DEST[i+31:i] ← 0
        FI
    FI
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

```

VPMOVSXBQ (EVEX encoded versions)
(KL, VL) = (2, 128), (4, 256), (8, 512)
Packed_Sign_Extend_BYTE_to_QWORD(TMP_DEST[127:0], SRC[15:0])
IF VL >= 256
    Packed_Sign_Extend_BYTE_to_QWORD(TMP_DEST[255:128], SRC[31:16])
FI;
IF VL >= 512
    Packed_Sign_Extend_BYTE_to_QWORD(TMP_DEST[383:256], SRC[47:32])
    Packed_Sign_Extend_BYTE_to_QWORD(TMP_DEST[511:384], SRC[63:48])
FI;
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← TEMP_DEST[i+63:i]
    ELSE
        IF *merging-masking*           ; merging-masking
            THEN *DEST[i+63:i] remains unchanged*
        ELSE *zeroing-masking*         ; zeroing-masking
            DEST[i+63:i] ← 0
        FI
    FI
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VPMOVSXWD (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

Packed\_Sign\_Extend\_WORD\_to\_DWORD(TMP\_DEST[127:0], SRC[63:0])

IF VL &gt;= 256

Packed\_Sign\_Extend\_WORD\_to\_DWORD(TMP\_DEST[255:128], SRC[127:64])

FI;

IF VL &gt;= 512

Packed\_Sign\_Extend\_WORD\_to\_DWORD(TMP\_DEST[383:256], SRC[191:128])

Packed\_Sign\_Extend\_WORD\_to\_DWORD(TMP\_DEST[511:384], SRC[256:192])

FI;

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\*

THEN DEST[i+31:i] ← TEMP\_DEST[i+31:i]

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+31:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VPMOVSXWQ (EVEX encoded versions)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

Packed\_Sign\_Extend\_WORD\_to\_QWORD(TMP\_DEST[127:0], SRC[31:0])

IF VL &gt;= 256

Packed\_Sign\_Extend\_WORD\_to\_QWORD(TMP\_DEST[255:128], SRC[63:32])

FI;

IF VL &gt;= 512

Packed\_Sign\_Extend\_WORD\_to\_QWORD(TMP\_DEST[383:256], SRC[95:64])

Packed\_Sign\_Extend\_WORD\_to\_QWORD(TMP\_DEST[511:384], SRC[127:96])

FI;

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN DEST[i+63:i] ← TEMP\_DEST[i+63:i]

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VPMOVSXDQ (EVEX encoded versions)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

Packed\_Sign\_Extend\_DWORD\_to\_QWORD(TEMP\_DEST[127:0], SRC[63:0])

IF VL &gt;= 256

Packed\_Sign\_Extend\_DWORD\_to\_QWORD(TEMP\_DEST[255:128], SRC[127:64])

FI;

IF VL &gt;= 512

Packed\_Sign\_Extend\_DWORD\_to\_QWORD(TEMP\_DEST[383:256], SRC[191:128])

Packed\_Sign\_Extend\_DWORD\_to\_QWORD(TEMP\_DEST[511:384], SRC[255:192])

FI;

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN DEST[i+63:i] ← TEMP\_DEST[i+63:i]

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VPMOVSXBW (VEX.256 encoded version)**

Packed\_Sign\_Extend\_BYTE\_to\_WORD(DEST[127:0], SRC[63:0])

Packed\_Sign\_Extend\_BYTE\_to\_WORD(DEST[255:128], SRC[127:64])

DEST[MAX\_VL-1:256] ← 0

**VPMOVSXBD (VEX.256 encoded version)**

Packed\_Sign\_Extend\_BYTE\_to\_DWORD(DEST[127:0], SRC[31:0])

Packed\_Sign\_Extend\_BYTE\_to\_DWORD(DEST[255:128], SRC[63:32])

DEST[MAX\_VL-1:256] ← 0

**VPMOVSXBQ (VEX.256 encoded version)**

Packed\_Sign\_Extend\_BYTE\_to\_QWORD(DEST[127:0], SRC[15:0])

Packed\_Sign\_Extend\_BYTE\_to\_QWORD(DEST[255:128], SRC[31:16])

DEST[MAX\_VL-1:256] ← 0

**VPMOVSXWD (VEX.256 encoded version)**

Packed\_Sign\_Extend\_WORD\_to\_DWORD(DEST[127:0], SRC[63:0])

Packed\_Sign\_Extend\_WORD\_to\_DWORD(DEST[255:128], SRC[127:64])

DEST[MAX\_VL-1:256] ← 0

**VPMOVSXWQ (VEX.256 encoded version)**

Packed\_Sign\_Extend\_WORD\_to\_QWORD(DEST[127:0], SRC[31:0])

Packed\_Sign\_Extend\_WORD\_to\_QWORD(DEST[255:128], SRC[63:32])

DEST[MAX\_VL-1:256] ← 0

**VPMOVSXDQ (VEX.256 encoded version)**

Packed\_Sign\_Extend\_DWORD\_to\_QWORD(DEST[127:0], SRC[63:0])

Packed\_Sign\_Extend\_DWORD\_to\_QWORD(DEST[255:128], SRC[127:64])

DEST[MAX\_VL-1:256] ← 0



**VPMOVSXBW (VEX.128 encoded version)**

Packed\_Sign\_Extend\_BYTE\_to\_WORDDEST[127:0], SRC[127:0])  
 DEST[MAX\_VL-1:128] ←0

**VPMOVSXBD (VEX.128 encoded version)**

Packed\_Sign\_Extend\_BYTE\_to\_DWORD(DEST[127:0], SRC[127:0])  
 DEST[MAX\_VL-1:128] ←0

**VPMOVSXBQ (VEX.128 encoded version)**

Packed\_Sign\_Extend\_BYTE\_to\_QWORD(DEST[127:0], SRC[127:0])  
 DEST[MAX\_VL-1:128] ←0

**VPMOVSXWD (VEX.128 encoded version)**

Packed\_Sign\_Extend\_WORD\_to\_DWORD(DEST[127:0], SRC[127:0])  
 DEST[MAX\_VL-1:128] ←0

**VPMOVSXWQ (VEX.128 encoded version)**

Packed\_Sign\_Extend\_WORD\_to\_QWORD(DEST[127:0], SRC[127:0])  
 DEST[MAX\_VL-1:128] ←0

**VPMOVSXDQ (VEX.128 encoded version)**

Packed\_Sign\_Extend\_DWORD\_to\_QWORD(DEST[127:0], SRC[127:0])  
 DEST[MAX\_VL-1:128] ←0

**PMOVSXBW**

Packed\_Sign\_Extend\_BYTE\_to\_WORD(DEST[127:0], SRC[127:0])  
 DEST[MAX\_VL-1:128] (Unmodified)

**PMOVSXBD**

Packed\_Sign\_Extend\_BYTE\_to\_DWORD(DEST[127:0], SRC[127:0])  
 DEST[MAX\_VL-1:128] (Unmodified)

**PMOVSXBQ**

Packed\_Sign\_Extend\_BYTE\_to\_QWORD(DEST[127:0], SRC[127:0])  
 DEST[MAX\_VL-1:128] (Unmodified)

**PMOVSXWD**

Packed\_Sign\_Extend\_WORD\_to\_DWORD(DEST[127:0], SRC[127:0])  
 DEST[MAX\_VL-1:128] (Unmodified)

**PMOVSXWQ**

Packed\_Sign\_Extend\_WORD\_to\_QWORD(DEST[127:0], SRC[127:0])  
 DEST[MAX\_VL-1:128] (Unmodified)

**PMOVSXDQ**

Packed\_Sign\_Extend\_DWORD\_to\_QWORD(DEST[127:0], SRC[127:0])  
 DEST[MAX\_VL-1:128] (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**

VPMOVSXBW \_\_m512i \_\_mm512\_cvtepi8\_epi16(\_\_m512i a);  
 VPMOVSXBW \_\_m512i \_\_mm512\_mask\_cvtepi8\_epi16(\_\_m512i a, \_\_mmask32 k, \_\_m512i b);  
 VPMOVSXBW \_\_m512i \_\_mm512\_maskz\_cvtepi8\_epi16(\_\_mmask32 k, \_\_m512i b);  
 VPMOVSXBD \_\_m512i \_\_mm512\_cvtepi8\_epi32(\_\_m512i a);  
 VPMOVSXBD \_\_m512i \_\_mm512\_mask\_cvtepi8\_epi32(\_\_m512i a, \_\_mmask16 k, \_\_m512i b);

VPMOVSXBD \_\_m512i \_mm512\_maskz\_cvtepi8\_epi32( \_\_mmask16 k, \_\_m512i b);  
 VPMOVSXBQ \_\_m512i \_mm512\_cvtepi8\_epi64(\_\_m512i a);  
 VPMOVSXBQ \_\_m512i \_mm512\_mask\_cvtepi8\_epi64(\_\_m512i a, \_\_mmask8 k, \_\_m512i b);  
 VPMOVSXBQ \_\_m512i \_mm512\_maskz\_cvtepi8\_epi64( \_\_mmask8 k, \_\_m512i a);  
 VPMOVSXDQ \_\_m512i \_mm512\_cvtepi32\_epi64(\_\_m512i a);  
 VPMOVSXDQ \_\_m512i \_mm512\_mask\_cvtepi32\_epi64(\_\_m512i a, \_\_mmask8 k, \_\_m512i b);  
 VPMOVSXDQ \_\_m512i \_mm512\_maskz\_cvtepi32\_epi64( \_\_mmask8 k, \_\_m512i a);  
 VPMOVSXWD \_\_m512i \_mm512\_cvtepi16\_epi32(\_\_m512i a);  
 VPMOVSXWD \_\_m512i \_mm512\_mask\_cvtepi16\_epi32(\_\_m512i a, \_\_mmask16 k, \_\_m512i b);  
 VPMOVSXWD \_\_m512i \_mm512\_maskz\_cvtepi16\_epi32(\_\_mmask16 k, \_\_m512i a);  
 VPMOVSXWQ \_\_m512i \_mm512\_cvtepi16\_epi64(\_\_m512i a);  
 VPMOVSXWQ \_\_m512i \_mm512\_mask\_cvtepi16\_epi64(\_\_m512i a, \_\_mmask8 k, \_\_m512i b);  
 VPMOVSXWQ \_\_m512i \_mm512\_maskz\_cvtepi16\_epi64( \_\_mmask8 k, \_\_m512i a);  
 VPMOVSXBW \_\_m256i \_mm256\_cvtepi8\_epi16(\_\_m256i a);  
 VPMOVSXBW \_\_m256i \_mm256\_mask\_cvtepi8\_epi16(\_\_m256i a, \_\_mmask16 k, \_\_m256i b);  
 VPMOVSXBW \_\_m256i \_mm256\_maskz\_cvtepi8\_epi16( \_\_mmask16 k, \_\_m256i b);  
 VPMOVSXBD \_\_m256i \_mm256\_cvtepi8\_epi32(\_\_m256i a);  
 VPMOVSXBD \_\_m256i \_mm256\_mask\_cvtepi8\_epi32(\_\_m256i a, \_\_mmask8 k, \_\_m256i b);  
 VPMOVSXBD \_\_m256i \_mm256\_maskz\_cvtepi8\_epi32( \_\_mmask8 k, \_\_m256i b);  
 VPMOVSXBQ \_\_m256i \_mm256\_cvtepi8\_epi64(\_\_m256i a);  
 VPMOVSXBQ \_\_m256i \_mm256\_mask\_cvtepi8\_epi64(\_\_m256i a, \_\_mmask8 k, \_\_m256i b);  
 VPMOVSXBQ \_\_m256i \_mm256\_maskz\_cvtepi8\_epi64( \_\_mmask8 k, \_\_m256i a);  
 VPMOVSXDQ \_\_m256i \_mm256\_cvtepi32\_epi64(\_\_m256i a);  
 VPMOVSXDQ \_\_m256i \_mm256\_mask\_cvtepi32\_epi64(\_\_m256i a, \_\_mmask8 k, \_\_m256i b);  
 VPMOVSXDQ \_\_m256i \_mm256\_maskz\_cvtepi32\_epi64( \_\_mmask8 k, \_\_m256i a);  
 VPMOVSXWD \_\_m256i \_mm256\_cvtepi16\_epi32(\_\_m256i a);  
 VPMOVSXWD \_\_m256i \_mm256\_mask\_cvtepi16\_epi32(\_\_m256i a, \_\_mmask16 k, \_\_m256i b);  
 VPMOVSXWD \_\_m256i \_mm256\_maskz\_cvtepi16\_epi32(\_\_mmask16 k, \_\_m256i a);  
 VPMOVSXWQ \_\_m256i \_mm256\_cvtepi16\_epi64(\_\_m256i a);  
 VPMOVSXWQ \_\_m256i \_mm256\_mask\_cvtepi16\_epi64(\_\_m256i a, \_\_mmask8 k, \_\_m256i b);  
 VPMOVSXWQ \_\_m256i \_mm256\_maskz\_cvtepi16\_epi64( \_\_mmask8 k, \_\_m256i a);  
 VPMOVSXBW \_\_m128i \_mm\_mask\_cvtepi8\_epi16(\_\_m128i a, \_\_mmask8 k, \_\_m128i b);  
 VPMOVSXBW \_\_m128i \_mm\_maskz\_cvtepi8\_epi16( \_\_mmask8 k, \_\_m128i b);  
 VPMOVSXBD \_\_m128i \_mm\_mask\_cvtepi8\_epi32(\_\_m128i a, \_\_mmask8 k, \_\_m128i b);  
 VPMOVSXBD \_\_m128i \_mm\_maskz\_cvtepi8\_epi32( \_\_mmask8 k, \_\_m128i b);  
 VPMOVSXBQ \_\_m128i \_mm\_mask\_cvtepi8\_epi64(\_\_m128i a, \_\_mmask8 k, \_\_m128i b);  
 VPMOVSXBQ \_\_m128i \_mm\_maskz\_cvtepi8\_epi64( \_\_mmask8 k, \_\_m128i a);  
 VPMOVSXDQ \_\_m128i \_mm\_mask\_cvtepi32\_epi64(\_\_m128i a, \_\_mmask8 k, \_\_m128i b);  
 VPMOVSXDQ \_\_m128i \_mm\_maskz\_cvtepi32\_epi64( \_\_mmask8 k, \_\_m128i a);  
 VPMOVSXWD \_\_m128i \_mm\_mask\_cvtepi16\_epi32(\_\_m128i a, \_\_mmask16 k, \_\_m128i b);  
 VPMOVSXWD \_\_m128i \_mm\_maskz\_cvtepi16\_epi32(\_\_mmask16 k, \_\_m128i a);  
 VPMOVSXWQ \_\_m128i \_mm\_mask\_cvtepi16\_epi64(\_\_m128i a, \_\_mmask8 k, \_\_m128i b);  
 VPMOVSXWQ \_\_m128i \_mm\_maskz\_cvtepi16\_epi64( \_\_mmask8 k, \_\_m128i a);  
 PMOVSXBW \_\_m128i \_mm\_cvtepi8\_epi16 ( \_\_m128i a);  
 PMOVSXBD \_\_m128i \_mm\_cvtepi8\_epi32 ( \_\_m128i a);  
 PMOVSXBQ \_\_m128i \_mm\_cvtepi8\_epi64 ( \_\_m128i a);  
 PMOVSXWD \_\_m128i \_mm\_cvtepi16\_epi32 ( \_\_m128i a);  
 PMOVSXWQ \_\_m128i \_mm\_cvtepi16\_epi64 ( \_\_m128i a);  
 PMOVSXDQ \_\_m128i \_mm\_cvtepi32\_epi64 ( \_\_m128i a);

### SIMD Floating-Point Exceptions

None

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 5.

EVEX-encoded instruction, see Exceptions Type E5.

#UD                      If VEX.vvvv != 1111B, or EVEX.vvvv != 1111B.

## PMOVZX—Packed Move with Zero Extend

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0f 38 30 /r PMOVZXBW xmm1, xmm2/m64	RM	V/V	SSE4_1	Zero extend 8 packed 8-bit integers in the low 8 bytes of xmm2/m64 to 8 packed 16-bit integers in xmm1.
66 0f 38 31 /r PMOVZXBW xmm1, xmm2/m32	RM	V/V	SSE4_1	Zero extend 4 packed 8-bit integers in the low 4 bytes of xmm2/m32 to 4 packed 32-bit integers in xmm1.
66 0f 38 32 /r PMOVZXBQ xmm1, xmm2/m16	RM	V/V	SSE4_1	Zero extend 2 packed 8-bit integers in the low 2 bytes of xmm2/m16 to 2 packed 64-bit integers in xmm1.
66 0f 38 33 /r PMOVZXWD xmm1, xmm2/m64	RM	V/V	SSE4_1	Zero extend 4 packed 16-bit integers in the low 8 bytes of xmm2/m64 to 4 packed 32-bit integers in xmm1.
66 0f 38 34 /r PMOVZXWQ xmm1, xmm2/m32	RM	V/V	SSE4_1	Zero extend 2 packed 16-bit integers in the low 4 bytes of xmm2/m32 to 2 packed 64-bit integers in xmm1.
66 0f 38 35 /r PMOVZXDQ xmm1, xmm2/m64	RM	V/V	SSE4_1	Zero extend 2 packed 32-bit integers in the low 8 bytes of xmm2/m64 to 2 packed 64-bit integers in xmm1.
VEX.128.66.0F38.WIG 30 /r VPMOVZXBW xmm1, xmm2/m64	RM	V/V	AVX	Zero extend 8 packed 8-bit integers in the low 8 bytes of xmm2/m64 to 8 packed 16-bit integers in xmm1.
VEX.128.66.0F38.WIG 31 /r VPMOVZXBW xmm1, xmm2/m32	RM	V/V	AVX	Zero extend 4 packed 8-bit integers in the low 4 bytes of xmm2/m32 to 4 packed 32-bit integers in xmm1.
VEX.128.66.0F38.WIG 32 /r VPMOVZXBQ xmm1, xmm2/m16	RM	V/V	AVX	Zero extend 2 packed 8-bit integers in the low 2 bytes of xmm2/m16 to 2 packed 64-bit integers in xmm1.
VEX.128.66.0F38.WIG 33 /r VPMOVZXWD xmm1, xmm2/m64	RM	V/V	AVX	Zero extend 4 packed 16-bit integers in the low 8 bytes of xmm2/m64 to 4 packed 32-bit integers in xmm1.
VEX.128.66.0F38.WIG 34 /r VPMOVZXWQ xmm1, xmm2/m32	RM	V/V	AVX	Zero extend 2 packed 16-bit integers in the low 4 bytes of xmm2/m32 to 2 packed 64-bit integers in xmm1.
VEX.128.66.0F 38.WIG 35 /r VPMOVZXDQ xmm1, xmm2/m64	RM	V/V	AVX	Zero extend 2 packed 32-bit integers in the low 8 bytes of xmm2/m64 to 2 packed 64-bit integers in xmm1.
VEX.256.66.0F38.WIG 30 /r VPMOVZXBW ymm1, xmm2/m128	RM	V/V	AVX2	Zero extend 16 packed 8-bit integers in xmm2/m128 to 16 packed 16-bit integers in ymm1.
VEX.256.66.0F38.WIG 31 /r VPMOVZXBW ymm1, xmm2/m64	RM	V/V	AVX2	Zero extend 8 packed 8-bit integers in the low 8 bytes of xmm2/m64 to 8 packed 32-bit integers in ymm1.
VEX.256.66.0F38.WIG 32 /r VPMOVZXBQ ymm1, xmm2/m32	RM	V/V	AVX2	Zero extend 4 packed 8-bit integers in the low 4 bytes of xmm2/m32 to 4 packed 64-bit integers in ymm1.
VEX.256.66.0F38.WIG 33 /r VPMOVZXWD ymm1, xmm2/m128	RM	V/V	AVX2	Zero extend 8 packed 16-bit integers xmm2/m128 to 8 packed 32-bit integers in ymm1.

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.256.66.0F38.WIG 34 /r VPMOVZXWQ ymm1, xmm2/m64	RM	V/V	AVX2	Zero extend 4 packed 16-bit integers in the low 8 bytes of xmm2/m64 to 4 packed 64-bit integers in xmm1.
VEX.256.66.0F38.WIG 35 /r VPMOVZXDQ ymm1, xmm2/m128	RM	V/V	AVX2	Zero extend 4 packed 32-bit integers in xmm2/m128 to 4 packed 64-bit integers in ymm1.
EVEX.128.66.0F38.30.WIG /r VPMOVZXBW xmm1 {k1}{z}, xmm2/m64	HVM	V/V	AVX512VL AVX512BW	Zero extend 8 packed 8-bit integers in the low 8 bytes of xmm2/m64 to 8 packed 16-bit integers in xmm1.
EVEX.256.66.0F38.WIG 30 /r VPMOVZXBW ymm1 {k1}{z}, xmm2/m128	HVM	V/V	AVX512VL AVX512BW	Zero extend 16 packed 8-bit integers in xmm2/m128 to 16 packed 16-bit integers in ymm1.
EVEX.512.66.0F38.WIG 30 /r VPMOVZXBW zmm1 {k1}{z}, ymm2/m256	HVM	V/V	AVX512BW	Zero extend 32 packed 8-bit integers in ymm2/m256 to 32 packed 16-bit integers in zmm1.
EVEX.128.66.0F38.WIG 31 /r VPMOVZXBD xmm1 {k1}{z}, xmm2/m32	QVM	V/V	AVX512VL AVX512F	Zero extend 4 packed 8-bit integers in the low 4 bytes of xmm2/m32 to 4 packed 32-bit integers in xmm1 subject to writemask k1.
EVEX.256.66.0F38.WIG 31 /r VPMOVZXBD ymm1 {k1}{z}, xmm2/m64	QVM	V/V	AVX512VL AVX512F	Zero extend 8 packed 8-bit integers in the low 8 bytes of xmm2/m64 to 8 packed 32-bit integers in ymm1 subject to writemask k1.
EVEX.512.66.0F38.WIG 31 /r VPMOVZXBD zmm1 {k1}{z}, xmm2/m128	QVM	V/V	AVX512F	Zero extend 16 packed 8-bit integers in xmm2/m128 to 16 packed 32-bit integers in zmm1 subject to writemask k1.
EVEX.128.66.0F38.WIG 32 /r VPMOVZXBQ xmm1 {k1}{z}, xmm2/m16	OVM	V/V	AVX512VL AVX512F	Zero extend 2 packed 8-bit integers in the low 2 bytes of xmm2/m16 to 2 packed 64-bit integers in xmm1 subject to writemask k1.
EVEX.256.66.0F38.WIG 32 /r VPMOVZXBQ ymm1 {k1}{z}, xmm2/m32	OVM	V/V	AVX512VL AVX512F	Zero extend 4 packed 8-bit integers in the low 4 bytes of xmm2/m32 to 4 packed 64-bit integers in ymm1 subject to writemask k1.
EVEX.512.66.0F38.WIG 32 /r VPMOVZXBQ zmm1 {k1}{z}, xmm2/m64	OVM	V/V	AVX512F	Zero extend 8 packed 8-bit integers in the low 8 bytes of xmm2/m64 to 8 packed 64-bit integers in zmm1 subject to writemask k1.
EVEX.128.66.0F38.WIG 33 /r VPMOVZXWD xmm1 {k1}{z}, xmm2/m64	HVM	V/V	AVX512VL AVX512F	Zero extend 4 packed 16-bit integers in the low 8 bytes of xmm2/m64 to 4 packed 32-bit integers in xmm1 subject to writemask k1.
EVEX.256.66.0F38.WIG 33 /r VPMOVZXWD ymm1 {k1}{z}, xmm2/m128	HVM	V/V	AVX512VL AVX512F	Zero extend 8 packed 16-bit integers in xmm2/m128 to 8 packed 32-bit integers in zmm1 subject to writemask k1.
EVEX.512.66.0F38.WIG 33 /r VPMOVZXWD zmm1 {k1}{z}, ymm2/m256	HVM	V/V	AVX512F	Zero extend 16 packed 16-bit integers in ymm2/m256 to 16 packed 32-bit integers in zmm1 subject to writemask k1.
EVEX.128.66.0F38.WIG 34 /r VPMOVZXWQ xmm1 {k1}{z}, xmm2/m32	QVM	V/V	AVX512VL AVX512F	Zero extend 2 packed 16-bit integers in the low 4 bytes of xmm2/m32 to 2 packed 64-bit integers in xmm1 subject to writemask k1.

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.256.66.0F38.WIG 34 /r VPMOVZXWQ ymm1 {k1}{z}, xmm2/m64	QVM	V/V	AVX512VL AVX512F	Zero extend 4 packed 16-bit integers in the low 8 bytes of xmm2/m64 to 4 packed 64-bit integers in ymm1 subject to writemask k1.
EVEX.512.66.0F38.WIG 34 /r VPMOVZXWQ zmm1 {k1}{z}, xmm2/m128	QVM	V/V	AVX512F	Zero extend 8 packed 16-bit integers in xmm2/m128 to 8 packed 64-bit integers in zmm1 subject to writemask k1.
EVEX.128.66.0F38.W0 35 /r VPMOVZXDQ xmm1 {k1}{z}, xmm2/m64	HVM	V/V	AVX512VL AVX512F	Zero extend 2 packed 32-bit integers in the low 8 bytes of xmm2/m64 to 2 packed 64-bit integers in zmm1 using writemask k1.
EVEX.256.66.0F38.W0 35 /r VPMOVZXDQ ymm1 {k1}{z}, xmm2/m128	HVM	V/V	AVX512VL AVX512F	Zero extend 4 packed 32-bit integers in xmm2/m128 to 4 packed 64-bit integers in zmm1 using writemask k1.
EVEX.512.66.0F38.W0 35 /r VPMOVZXDQ zmm1 {k1}{z}, ymm2/m256	HVM	V/V	AVX512F	Zero extend 8 packed 32-bit integers in ymm2/m256 to 8 packed 64-bit integers in zmm1 using writemask k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
HVM, QVM, OVM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Legacy, VEX and EVEX encoded versions: Packed byte, word, or dword integers starting from the low bytes of the source operand (second operand) are zero extended to word, dword, or quadword integers and stored in packed signed bytes the destination operand.

128-bit Legacy SSE version: Bits (MAX\_VL-1:128) of the corresponding destination register remain unchanged.

VEX.128 encoded version: Bits (MAX\_VL-1:128) of the corresponding destination register are zeroed.

VEX.256 encoded version: Bits (MAX\_VL-1:256) of the corresponding destination register are zeroed.

EVEX encoded versions: Packed dword integers starting from the low bytes of the source operand (second operand) are zero extended to quadword integers and stored to the destination operand under the writemask. The destination register is XMM, YMM or ZMM Register.

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

### Operation

#### Packed\_Zero\_Extend\_BYTE\_to\_WORD(DEST, SRC)

```
DEST[15:0] ← ZeroExtend(SRC[7:0]);
DEST[31:16] ← ZeroExtend(SRC[15:8]);
DEST[47:32] ← ZeroExtend(SRC[23:16]);
DEST[63:48] ← ZeroExtend(SRC[31:24]);
DEST[79:64] ← ZeroExtend(SRC[39:32]);
DEST[95:80] ← ZeroExtend(SRC[47:40]);
DEST[111:96] ← ZeroExtend(SRC[55:48]);
DEST[127:112] ← ZeroExtend(SRC[63:56]);
```

#### Packed\_Zero\_Extend\_BYTE\_to\_DWORD(DEST, SRC)

```
DEST[31:0] ← ZeroExtend(SRC[7:0]);
```

DEST[63:32] ← ZeroExtend(SRC[15:8]);  
 DEST[95:64] ← ZeroExtend(SRC[23:16]);  
 DEST[127:96] ← ZeroExtend(SRC[31:24]);

**Packed\_Zero\_Extend\_BYTE\_to\_QWORD(DEST, SRC)**

DEST[63:0] ← ZeroExtend(SRC[7:0]);  
 DEST[127:64] ← ZeroExtend(SRC[15:8]);

**Packed\_Zero\_Extend\_WORD\_to\_DWORD(DEST, SRC)**

DEST[31:0] ← ZeroExtend(SRC[15:0]);  
 DEST[63:32] ← ZeroExtend(SRC[31:16]);  
 DEST[95:64] ← ZeroExtend(SRC[47:32]);  
 DEST[127:96] ← ZeroExtend(SRC[63:48]);

**Packed\_Zero\_Extend\_WORD\_to\_QWORD(DEST, SRC)**

DEST[63:0] ← ZeroExtend(SRC[15:0]);  
 DEST[127:64] ← ZeroExtend(SRC[31:16]);

**Packed\_Zero\_Extend\_DWORD\_to\_QWORD(DEST, SRC)**

DEST[63:0] ← ZeroExtend(SRC[31:0]);  
 DEST[127:64] ← ZeroExtend(SRC[63:32]);

**VPMOVZXBW (EVEX encoded versions)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

Packed\_Zero\_Extend\_BYTE\_to\_WORD(TMP\_DEST[127:0], SRC[63:0])

IF VL >= 256

    Packed\_Zero\_Extend\_BYTE\_to\_WORD(TMP\_DEST[255:128], SRC[127:64])

FI;

IF VL >= 512

    Packed\_Zero\_Extend\_BYTE\_to\_WORD(TMP\_DEST[383:256], SRC[191:128])

    Packed\_Zero\_Extend\_BYTE\_to\_WORD(TMP\_DEST[511:384], SRC[255:192])

FI;

FOR j ← 0 TO KL-1

    i ← j \* 16

    IF k1[j] OR \*no writemask\*

        THEN DEST[i+15:i] ← TEMP\_DEST[i+15:i]

    ELSE

        IF \*merging-masking\* ; merging-masking

            THEN \*DEST[i+15:i] remains unchanged\*

        ELSE \*zeroing-masking\* ; zeroing-masking

            DEST[i+15:i] ← 0

    FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VPMOVZXBW (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

Packed\_Zero\_Extend\_BYTE\_to\_DWORD(TMP\_DEST[127:0], SRC[31:0])

IF VL >= 256

    Packed\_Zero\_Extend\_BYTE\_to\_DWORD(TMP\_DEST[255:128], SRC[63:32])

FI;

IF VL >= 512

    Packed\_Zero\_Extend\_BYTE\_to\_DWORD(TMP\_DEST[383:256], SRC[95:64])

```

Packed_Zero_Extend_BYTE_to_DWORD(TMP_DEST[511:384], SRC[127:96])
FI;
FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ← TEMP_DEST[i+31:i]
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+31:i] remains unchanged*
    ELSE *zeroing-masking* ; zeroing-masking
      DEST[i+31:i] ← 0
    FI
  FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VPMOVZXBQ (EVEX encoded versions)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

```

Packed_Zero_Extend_BYTE_to_QWORD(TMP_DEST[127:0], SRC[15:0])
IF VL >= 256
  Packed_Zero_Extend_BYTE_to_QWORD(TMP_DEST[255:128], SRC[31:16])
FI;
IF VL >= 512
  Packed_Zero_Extend_BYTE_to_QWORD(TMP_DEST[383:256], SRC[47:32])
  Packed_Zero_Extend_BYTE_to_QWORD(TMP_DEST[511:384], SRC[63:48])
FI;
FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] ← TEMP_DEST[i+63:i]
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+63:i] remains unchanged*
    ELSE *zeroing-masking* ; zeroing-masking
      DEST[i+63:i] ← 0
    FI
  FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VPMOVZXWD (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

```

Packed_Zero_Extend_WORD_to_DWORD(TMP_DEST[127:0], SRC[63:0])
IF VL >= 256
  Packed_Zero_Extend_WORD_to_DWORD(TMP_DEST[255:128], SRC[127:64])
FI;
IF VL >= 512
  Packed_Zero_Extend_WORD_to_DWORD(TMP_DEST[383:256], SRC[191:128])
  Packed_Zero_Extend_WORD_to_DWORD(TMP_DEST[511:384], SRC[256:192])
FI;
FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ← TEMP_DEST[i+31:i]

```



```

ELSE
    IF *merging-masking*           ; merging-masking
        THEN *DEST[j+31:i] remains unchanged*
    ELSE *zeroing-masking*       ; zeroing-masking
        DEST[j+31:i] ← 0
    FI
FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VPMOVZXWQ (EVEX encoded versions)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

Packed\_Zero\_Extend\_WORD\_to\_QWORD(TMP\_DEST[127:0], SRC[31:0])

IF VL &gt;= 256

Packed\_Zero\_Extend\_WORD\_to\_QWORD(TMP\_DEST[255:128], SRC[63:32])

FI;

IF VL &gt;= 512

Packed\_Zero\_Extend\_WORD\_to\_QWORD(TMP\_DEST[383:256], SRC[95:64])

Packed\_Zero\_Extend\_WORD\_to\_QWORD(TMP\_DEST[511:384], SRC[127:96])

FI;

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN DEST[j+63:i] ← TEMP\_DEST[j+63:i]

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[j+63:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[j+63:i] ← 0

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VPMOVZXDQ (EVEX encoded versions)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

Packed\_Zero\_Extend\_DWORD\_to\_QWORD(TEMP\_DEST[127:0], SRC[63:0])

IF VL &gt;= 256

Packed\_Zero\_Extend\_DWORD\_to\_QWORD(TEMP\_DEST[255:128], SRC[127:64])

FI;

IF VL &gt;= 512

Packed\_Zero\_Extend\_DWORD\_to\_QWORD(TEMP\_DEST[383:256], SRC[191:128])

Packed\_Zero\_Extend\_DWORD\_to\_QWORD(TEMP\_DEST[511:384], SRC[255:192])

FI;

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN DEST[j+63:i] ← TEMP\_DEST[j+63:i]

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[j+63:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[j+63:i] ← 0

FI

FI;  
 ENDFOR  
 DEST[MAX\_VL-1:VL] ← 0

**VPMOVZXBW (VEX.256 encoded version)**

Packed\_Zero\_Extend\_BYTE\_to\_WORD(DEST[127:0], SRC[63:0])  
 Packed\_Zero\_Extend\_BYTE\_to\_WORD(DEST[255:128], SRC[127:64])  
 DEST[MAX\_VL-1:256] ← 0

**VPMOVZXBW (VEX.256 encoded version)**

Packed\_Zero\_Extend\_BYTE\_to\_DWORD(DEST[127:0], SRC[31:0])  
 Packed\_Zero\_Extend\_BYTE\_to\_DWORD(DEST[255:128], SRC[63:32])  
 DEST[MAX\_VL-1:256] ← 0

**VPMOVZXBQ (VEX.256 encoded version)**

Packed\_Zero\_Extend\_BYTE\_to\_QWORD(DEST[127:0], SRC[15:0])  
 Packed\_Zero\_Extend\_BYTE\_to\_QWORD(DEST[255:128], SRC[31:16])  
 DEST[MAX\_VL-1:256] ← 0

**VPMOVZXWD (VEX.256 encoded version)**

Packed\_Zero\_Extend\_WORD\_to\_DWORD(DEST[127:0], SRC[63:0])  
 Packed\_Zero\_Extend\_WORD\_to\_DWORD(DEST[255:128], SRC[127:64])  
 DEST[MAX\_VL-1:256] ← 0

**VPMOVZXWQ (VEX.256 encoded version)**

Packed\_Zero\_Extend\_WORD\_to\_QWORD(DEST[127:0], SRC[31:0])  
 Packed\_Zero\_Extend\_WORD\_to\_QWORD(DEST[255:128], SRC[63:32])  
 DEST[MAX\_VL-1:256] ← 0

**VPMOVZXDQ (VEX.256 encoded version)**

Packed\_Zero\_Extend\_DWORD\_to\_QWORD(DEST[127:0], SRC[63:0])  
 Packed\_Zero\_Extend\_DWORD\_to\_QWORD(DEST[255:128], SRC[127:64])  
 DEST[MAX\_VL-1:256] ← 0

**VPMOVZXBW (VEX.128 encoded version)**

Packed\_Zero\_Extend\_BYTE\_to\_WORD()  
 DEST[MAX\_VL-1:128] ← 0

**VPMOVZXBW (VEX.128 encoded version)**

Packed\_Zero\_Extend\_BYTE\_to\_DWORD()  
 DEST[MAX\_VL-1:128] ← 0

**VPMOVZXBQ (VEX.128 encoded version)**

Packed\_Zero\_Extend\_BYTE\_to\_QWORD()  
 DEST[MAX\_VL-1:128] ← 0

**VPMOVZXWD (VEX.128 encoded version)**

Packed\_Zero\_Extend\_WORD\_to\_DWORD()  
 DEST[MAX\_VL-1:128] ← 0

**VPMOVZXWQ (VEX.128 encoded version)**

Packed\_Zero\_Extend\_WORD\_to\_QWORD()  
 DEST[MAX\_VL-1:128] ← 0

**VPMOVZXDQ (VEX.128 encoded version)**

Packed\_Zero\_Extend\_DWORD\_to\_QWORD()

DEST[MAX\_VL-1:128] ← 0

**PMOVZXBW**

Packed\_Zero\_Extend\_BYTE\_to\_WORD()

DEST[MAX\_VL-1:128] (Unmodified)

**PMOVZXBQ**

Packed\_Zero\_Extend\_BYTE\_to\_DWORD()

DEST[MAX\_VL-1:128] (Unmodified)

**PMOVZXBQ**

Packed\_Zero\_Extend\_BYTE\_to\_QWORD()

DEST[MAX\_VL-1:128] (Unmodified)

**PMOVZXWD**

Packed\_Zero\_Extend\_WORD\_to\_DWORD()

DEST[MAX\_VL-1:128] (Unmodified)

**PMOVZXWQ**

Packed\_Zero\_Extend\_WORD\_to\_QWORD()

DEST[MAX\_VL-1:128] (Unmodified)

**PMOVZXDQ**

Packed\_Zero\_Extend\_DWORD\_to\_QWORD()

DEST[MAX\_VL-1:128] (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**

VPMOVZXBW \_\_m512i \_\_mm512\_cvtepu8\_epi16(\_\_m256i a);  
 VPMOVZXBW \_\_m512i \_\_mm512\_mask\_cvtepu8\_epi16(\_\_m512i a, \_\_mmask32 k, \_\_m256i b);  
 VPMOVZXBW \_\_m512i \_\_mm512\_maskz\_cvtepu8\_epi16(\_\_mmask32 k, \_\_m256i b);  
 VPMOVZXBQ \_\_m512i \_\_mm512\_cvtepu8\_epi32(\_\_m128i a);  
 VPMOVZXBQ \_\_m512i \_\_mm512\_mask\_cvtepu8\_epi32(\_\_m512i a, \_\_mmask16 k, \_\_m128i b);  
 VPMOVZXBQ \_\_m512i \_\_mm512\_maskz\_cvtepu8\_epi32(\_\_mmask16 k, \_\_m128i b);  
 VPMOVZXBQ \_\_m512i \_\_mm512\_cvtepu8\_epi64(\_\_m128i a);  
 VPMOVZXBQ \_\_m512i \_\_mm512\_mask\_cvtepu8\_epi64(\_\_m512i a, \_\_mmask8 k, \_\_m128i b);  
 VPMOVZXBQ \_\_m512i \_\_mm512\_maskz\_cvtepu8\_epi64(\_\_mmask8 k, \_\_m128i a);  
 VPMOVZXDQ \_\_m512i \_\_mm512\_cvtepu32\_epi64(\_\_m256i a);  
 VPMOVZXDQ \_\_m512i \_\_mm512\_mask\_cvtepu32\_epi64(\_\_m512i a, \_\_mmask8 k, \_\_m256i b);  
 VPMOVZXDQ \_\_m512i \_\_mm512\_maskz\_cvtepu32\_epi64(\_\_mmask8 k, \_\_m256i a);  
 VPMOVZXWD \_\_m512i \_\_mm512\_cvtepu16\_epi32(\_\_m128i a);  
 VPMOVZXWD \_\_m512i \_\_mm512\_mask\_cvtepu16\_epi32(\_\_m512i a, \_\_mmask16 k, \_\_m128i b);  
 VPMOVZXWD \_\_m512i \_\_mm512\_maskz\_cvtepu16\_epi32(\_\_mmask16 k, \_\_m128i a);  
 VPMOVZXWQ \_\_m512i \_\_mm512\_cvtepu16\_epi64(\_\_m256i a);  
 VPMOVZXWQ \_\_m512i \_\_mm512\_mask\_cvtepu16\_epi64(\_\_m512i a, \_\_mmask8 k, \_\_m256i b);  
 VPMOVZXWQ \_\_m512i \_\_mm512\_maskz\_cvtepu16\_epi64(\_\_mmask8 k, \_\_m256i a);  
 VPMOVZXBW \_\_m256i \_\_mm256\_cvtepu8\_epi16(\_\_m256i a);  
 VPMOVZXBW \_\_m256i \_\_mm256\_mask\_cvtepu8\_epi16(\_\_m256i a, \_\_mmask16 k, \_\_m128i b);  
 VPMOVZXBW \_\_m256i \_\_mm256\_maskz\_cvtepu8\_epi16(\_\_mmask16 k, \_\_m128i b);  
 VPMOVZXBQ \_\_m256i \_\_mm256\_cvtepu8\_epi32(\_\_m128i a);  
 VPMOVZXBQ \_\_m256i \_\_mm256\_mask\_cvtepu8\_epi32(\_\_m256i a, \_\_mmask8 k, \_\_m128i b);  
 VPMOVZXBQ \_\_m256i \_\_mm256\_maskz\_cvtepu8\_epi32(\_\_mmask8 k, \_\_m128i b);  
 VPMOVZXBQ \_\_m256i \_\_mm256\_cvtepu8\_epi64(\_\_m128i a);

VPMOVZXBQ \_\_m256i \_\_mm256\_mask\_cvtepu8\_epi64(\_\_m256i a, \_\_mmask8 k, \_\_m128i b);  
 VPMOVZXBQ \_\_m256i \_\_mm256\_maskz\_cvtepu8\_epi64( \_\_mmask8 k, \_\_m128i a);  
 VPMOVZXDQ \_\_m256i \_\_mm256\_cvtepu32\_epi64(\_\_m128i a);  
 VPMOVZXDQ \_\_m256i \_\_mm256\_mask\_cvtepu32\_epi64(\_\_m256i a, \_\_mmask8 k, \_\_m128i b);  
 VPMOVZXDQ \_\_m256i \_\_mm256\_maskz\_cvtepu32\_epi64( \_\_mmask8 k, \_\_m128i a);  
 VPMOVZXWD \_\_m256i \_\_mm256\_cvtepu16\_epi32(\_\_m128i a);  
 VPMOVZXWD \_\_m256i \_\_mm256\_mask\_cvtepu16\_epi32(\_\_m256i a, \_\_mmask16 k, \_\_m128i b);  
 VPMOVZXWD \_\_m256i \_\_mm256\_maskz\_cvtepu16\_epi32(\_\_mmask16 k, \_\_m128i a);  
 VPMOVZXWQ \_\_m256i \_\_mm256\_cvtepu16\_epi64(\_\_m128i a);  
 VPMOVZXWQ \_\_m256i \_\_mm256\_mask\_cvtepu16\_epi64(\_\_m256i a, \_\_mmask8 k, \_\_m128i b);  
 VPMOVZXWQ \_\_m256i \_\_mm256\_maskz\_cvtepu16\_epi64( \_\_mmask8 k, \_\_m128i a);  
 VPMOVZXBW \_\_m128i \_\_mm\_mask\_cvtepu8\_epi16(\_\_m128i a, \_\_mmask8 k, \_\_m128i b);  
 VPMOVZXBW \_\_m128i \_\_mm\_maskz\_cvtepu8\_epi16( \_\_mmask8 k, \_\_m128i b);  
 VPMOVZXBW \_\_m128i \_\_mm\_mask\_cvtepu8\_epi32(\_\_m128i a, \_\_mmask8 k, \_\_m128i b);  
 VPMOVZXBW \_\_m128i \_\_mm\_maskz\_cvtepu8\_epi32( \_\_mmask8 k, \_\_m128i b);  
 VPMOVZXBQ \_\_m128i \_\_mm\_mask\_cvtepu8\_epi64(\_\_m128i a, \_\_mmask8 k, \_\_m128i b);  
 VPMOVZXBQ \_\_m128i \_\_mm\_maskz\_cvtepu8\_epi64( \_\_mmask8 k, \_\_m128i a);  
 VPMOVZXDQ \_\_m128i \_\_mm\_mask\_cvtepu32\_epi64(\_\_m128i a, \_\_mmask8 k, \_\_m128i b);  
 VPMOVZXDQ \_\_m128i \_\_mm\_maskz\_cvtepu32\_epi64( \_\_mmask8 k, \_\_m128i a);  
 VPMOVZXWD \_\_m128i \_\_mm\_mask\_cvtepu16\_epi32(\_\_m128i a, \_\_mmask16 k, \_\_m128i b);  
 VPMOVZXWD \_\_m128i \_\_mm\_maskz\_cvtepu16\_epi32(\_\_mmask8 k, \_\_m128i a);  
 VPMOVZXWQ \_\_m128i \_\_mm\_mask\_cvtepu16\_epi64(\_\_m128i a, \_\_mmask8 k, \_\_m128i b);  
 VPMOVZXWQ \_\_m128i \_\_mm\_maskz\_cvtepu16\_epi64( \_\_mmask8 k, \_\_m128i a);  
 PMOVZXBW \_\_m128i \_\_mm\_ cvtepu8\_epi16 ( \_\_m128i a);  
 PMOVZXBW \_\_m128i \_\_mm\_ cvtepu8\_epi32 ( \_\_m128i a);  
 PMOVZXBQ \_\_m128i \_\_mm\_ cvtepu8\_epi64 ( \_\_m128i a);  
 PMOVZXWD \_\_m128i \_\_mm\_ cvtepu16\_epi32 ( \_\_m128i a);  
 PMOVZXWQ \_\_m128i \_\_mm\_ cvtepu16\_epi64 ( \_\_m128i a);  
 PMOVZXDQ \_\_m128i \_\_mm\_ cvtepu32\_epi64 ( \_\_m128i a);

### SIMD Floating-Point Exceptions

None

### Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 5.

EVEX-encoded instruction, see Exceptions Type E5.

#UD If VEX.vvvv != 1111B, or EVEX.vvvv != 1111B.

## PMULDQ—Multiply Packed Doubleword Integers

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 28 /r PMULDQ xmm1, xmm2/m128	RM	V/V	SSE4_1	Multiply packed signed doubleword integers in xmm1 by packed signed doubleword integers in xmm2/m128, and store the quadword results in xmm1.
VEX.NDS.128.66.0F38.WIG 28 /r VPMULDQ xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Multiply packed signed doubleword integers in xmm2 by packed signed doubleword integers in xmm3/m128, and store the quadword results in xmm1.
VEX.NDS.256.66.0F38.WIG 28 /r VPMULDQ ymm1, ymm2, ymm3/m256	RVM	V/V	AVX2	Multiply packed signed doubleword integers in ymm2 by packed signed doubleword integers in ymm3/m256, and store the quadword results in ymm1.
EVEX.NDS.128.66.0F38.W1 28 /r VPMULDQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	FV	V/V	AVX512VL AVX512F	Multiply packed signed doubleword integers in xmm2 by packed signed doubleword integers in xmm3/m128/m64bcst, and store the quadword results in xmm1 using writemask k1.
EVEX.NDS.256.66.0F38.W1 28 /r VPMULDQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	FV	V/V	AVX512VL AVX512F	Multiply packed signed doubleword integers in ymm2 by packed signed doubleword integers in ymm3/m256/m64bcst, and store the quadword results in ymm1 using writemask k1.
EVEX.NDS.512.66.0F38.W1 28 /r VPMULDQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	FV	V/V	AVX512F	Multiply packed signed doubleword integers in zmm2 by packed signed doubleword integers in zmm3/m512/m64bcst, and store the quadword results in zmm1 using writemask k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FV	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Multiplies packed signed doubleword integers in the even-numbered (zero-based reference) elements of the first source operand with the packed signed doubleword integers in the corresponding elements of the second source operand and stores packed signed quadword results in the destination operand.

128-bit Legacy SSE version: The input signed doubleword integers are taken from the even-numbered elements of the source operands, i.e. the first (low) and third doubleword element. For 128-bit memory operands, 128 bits are fetched from memory, but only the first and third doublewords are used in the computation. The first source operand and the destination XMM operand is the same. The second source operand can be an XMM register or 128-bit memory location. Bits (MAX\_VL-1:128) of the corresponding destination register remain unchanged.

VEX.128 encoded version: The input signed doubleword integers are taken from the even-numbered elements of the source operands, i.e., the first (low) and third doubleword element. For 128-bit memory operands, 128 bits are fetched from memory, but only the first and third doublewords are used in the computation. The first source operand and the destination operand are XMM registers. The second source operand can be an XMM register or 128-bit memory location. Bits (MAX\_VL-1:128) of the corresponding destination register are zeroed.

VEX.256 encoded version: The input signed doubleword integers are taken from the even-numbered elements of the source operands, i.e. the first, 3rd, 5th, 7th doubleword element. For 256-bit memory operands, 256 bits are fetched from memory, but only the four even-numbered doublewords are used in the computation. The first source

operand and the destination operand are YMM registers. The second source operand can be a YMM register or 256-bit memory location. Bits (MAX\_VL-1:256) of the corresponding destination ZMM register are zeroed.

EVEX encoded version: The input signed doubleword integers are taken from the even-numbered elements of the source operands. The first source operand is a ZMM/YMM/XMM registers. The second source operand can be an ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination is a ZMM/YMM/XMM register, and updated according to the writemask at 64-bit granularity.

### Operation

#### VPMULDQ (EVEX encoded versions)

(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) AND (SRC2 *is memory*)
        THEN DEST[i+63:i] ← SignExtend64( SRC1[i+31:i]) * SignExtend64( SRC2[31:0])
        ELSE DEST[i+63:i] ← SignExtend64( SRC1[j+31:i]) * SignExtend64( SRC2[i+31:i])
      FI;
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+63:i] remains unchanged*
        ELSE *zeroing-masking* ; zeroing-masking
          DEST[i+63:i] ← 0
        FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0
```

#### VPMULDQ (VEX.256 encoded version)

```
DEST[63:0] ← SignExtend64( SRC1[31:0]) * SignExtend64( SRC2[31:0])
DEST[127:64] ← SignExtend64( SRC1[95:64]) * SignExtend64( SRC2[95:64])
DEST[191:128] ← SignExtend64( SRC1[159:128]) * SignExtend64( SRC2[159:128])
DEST[255:192] ← SignExtend64( SRC1[223:192]) * SignExtend64( SRC2[223:192])
DEST[MAX_VL-1:256] ← 0
```

#### VPMULDQ (VEX.128 encoded version)

```
DEST[63:0] ← SignExtend64( SRC1[31:0]) * SignExtend64( SRC2[31:0])
DEST[127:64] ← SignExtend64( SRC1[95:64]) * SignExtend64( SRC2[95:64])
DEST[MAX_VL-1:128] ← 0
```

#### PMULDQ (128-bit Legacy SSE version)

```
DEST[63:0] ← SignExtend64( DEST[31:0]) * SignExtend64( SRC[31:0])
DEST[127:64] ← SignExtend64( DEST[95:64]) * SignExtend64( SRC[95:64])
DEST[MAX_VL-1:128] (Unmodified)
```

### Intel C/C++ Compiler Intrinsic Equivalent

```
VPMULDQ __m512i __mm512_mul_epi32(__m512i a, __m512i b);
VPMULDQ __m512i __mm512_mask_mul_epi32(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPMULDQ __m512i __mm512_maskz_mul_epi32(__mmask8 k, __m512i a, __m512i b);
VPMULDQ __m256i __mm256_mask_mul_epi32(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPMULDQ __m256i __mm256_mask_mul_epi32(__mmask8 k, __m256i a, __m256i b);
VPMULDQ __m128i __mm_mask_mul_epi32(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMULDQ __m128i __mm_mask_mul_epi32(__mmask8 k, __m128i a, __m128i b);
```

(V)PMULDQ \_\_m128i \_mm\_mul\_epi32(\_\_m128i a, \_\_m128i b);  
VPMULDQ \_\_m256i \_mm256\_mul\_epi32(\_\_m256i a, \_\_m256i b);

#### **SIMD Floating-Point Exceptions**

None

#### **Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.

## PMULHRW — Packed Multiply High with Round and Scale

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
OF 3B 0B /r <sup>1</sup> PMULHRW <i>mm1, mm2/m64</i>	RM	V/V	SSSE3	Multiply 16-bit signed words, scale and round signed doublewords, pack high 16 bits to <i>mm1</i> .
66 OF 3B 0B /r PMULHRW <i>xmm1, xmm2/m128</i>	RM	V/V	SSSE3	Multiply 16-bit signed words, scale and round signed doublewords, pack high 16 bits to <i>xmm1</i> .
VEX.NDS.128.66.OF3B.WIG 0B /r VPMULHRW <i>xmm1, xmm2, xmm3/m128</i>	RVM	V/V	AVX	Multiply 16-bit signed words, scale and round signed doublewords, pack high 16 bits to <i>xmm1</i> .
VEX.NDS.256.66.OF3B.WIG 0B /r VPMULHRW <i>ymm1, ymm2, ymm3/m256</i>	RVM	V/V	AVX2	Multiply 16-bit signed words, scale and round signed doublewords, pack high 16 bits to <i>ymm1</i> .
EVEX.NDS.128.66.OF3B.WIG 0B /r VPMULHRW <i>xmm1 {k1}{z}, xmm2, xmm3/m128</i>	FVM	V/V	AVX512VL AVX512BW	Multiply 16-bit signed words, scale and round signed doublewords, pack high 16 bits to <i>xmm1</i> under writemask <i>k1</i> .
EVEX.NDS.256.66.OF3B.WIG 0B /r VPMULHRW <i>ymm1 {k1}{z}, ymm2, ymm3/m256</i>	FVM	V/V	AVX512VL AVX512BW	Multiply 16-bit signed words, scale and round signed doublewords, pack high 16 bits to <i>ymm1</i> under writemask <i>k1</i> .
EVEX.NDS.512.66.OF3B.WIG 0B /r VPMULHRW <i>zmm1 {k1}{z}, zmm2, zmm3/m512</i>	FVM	V/V	AVX512BW	Multiply 16-bit signed words, scale and round signed doublewords, pack high 16 bits to <i>zmm1</i> under writemask <i>k1</i> .

### NOTES:

1. See note in Section 2.4, "AVX and SSE Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FVM	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

PMULHRW multiplies vertically each signed 16-bit integer from the destination operand (first operand) with the corresponding signed 16-bit integer of the source operand (second operand), producing intermediate, signed 32-bit integers. Each intermediate 32-bit integer is truncated to the 18 most significant bits. Rounding is always performed by adding 1 to the least significant bit of the 18-bit intermediate result. The final result is obtained by selecting the 16 bits immediately to the right of the most significant bit of each 18-bit intermediate result and packed to the destination operand.

When the source operand is a 128-bit memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

In 64-bit mode and not encoded with VEX/EVEX, use the REX prefix to access XMM8-XMM15 registers.

Legacy SSE version 64-bit operand: Both operands can be MMX registers. The second source operand is an MMX register or a 64-bit memory location.



128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers.

EVEX encoded versions: The first source 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. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

## Operation

### PMULHRSW (with 64-bit operands)

```
temp0[31:0] = INT32 ((DEST[15:0] * SRC[15:0]) >>14) + 1;
temp1[31:0] = INT32 ((DEST[31:16] * SRC[31:16]) >>14) + 1;
temp2[31:0] = INT32 ((DEST[47:32] * SRC[47:32]) >>14) + 1;
temp3[31:0] = INT32 ((DEST[63:48] * SRC[63:48]) >>14) + 1;
DEST[15:0] = temp0[16:1];
DEST[31:16] = temp1[16:1];
DEST[47:32] = temp2[16:1];
DEST[63:48] = temp3[16:1];
```

### PMULHRSW (with 128-bit operand)

```
temp0[31:0] = INT32 ((DEST[15:0] * SRC[15:0]) >>14) + 1;
temp1[31:0] = INT32 ((DEST[31:16] * SRC[31:16]) >>14) + 1;
temp2[31:0] = INT32 ((DEST[47:32] * SRC[47:32]) >>14) + 1;
temp3[31:0] = INT32 ((DEST[63:48] * SRC[63:48]) >>14) + 1;
temp4[31:0] = INT32 ((DEST[79:64] * SRC[79:64]) >>14) + 1;
temp5[31:0] = INT32 ((DEST[95:80] * SRC[95:80]) >>14) + 1;
temp6[31:0] = INT32 ((DEST[111:96] * SRC[111:96]) >>14) + 1;
temp7[31:0] = INT32 ((DEST[127:112] * SRC[127:112]) >>14) + 1;
DEST[15:0] = temp0[16:1];
DEST[31:16] = temp1[16:1];
DEST[47:32] = temp2[16:1];
DEST[63:48] = temp3[16:1];
DEST[79:64] = temp4[16:1];
DEST[95:80] = temp5[16:1];
DEST[111:96] = temp6[16:1];
DEST[127:112] = temp7[16:1];
```

### VPMULHRSW (VEX.128 encoded version)

```
temp0[31:0] ← INT32 ((SRC1[15:0] * SRC2[15:0]) >>14) + 1
temp1[31:0] ← INT32 ((SRC1[31:16] * SRC2[31:16]) >>14) + 1
temp2[31:0] ← INT32 ((SRC1[47:32] * SRC2[47:32]) >>14) + 1
temp3[31:0] ← INT32 ((SRC1[63:48] * SRC2[63:48]) >>14) + 1
temp4[31:0] ← INT32 ((SRC1[79:64] * SRC2[79:64]) >>14) + 1
temp5[31:0] ← INT32 ((SRC1[95:80] * SRC2[95:80]) >>14) + 1
temp6[31:0] ← INT32 ((SRC1[111:96] * SRC2[111:96]) >>14) + 1
temp7[31:0] ← INT32 ((SRC1[127:112] * SRC2[127:112]) >>14) + 1
DEST[15:0] ← temp0[16:1]
DEST[31:16] ← temp1[16:1]
DEST[47:32] ← temp2[16:1]
```

```

DEST[63:48] ← temp3[16:1]
DEST[79:64] ← temp4[16:1]
DEST[95:80] ← temp5[16:1]
DEST[111:96] ← temp6[16:1]
DEST[127:112] ← temp7[16:1]
DEST[VLMAX-1:128] ← 0

```

**VPMULHRWSW (VEX.256 encoded version)**

```

temp0[31:0] ← INT32 ((SRC1[15:0] * SRC2[15:0]) >>14) + 1
temp1[31:0] ← INT32 ((SRC1[31:16] * SRC2[31:16]) >>14) + 1
temp2[31:0] ← INT32 ((SRC1[47:32] * SRC2[47:32]) >>14) + 1
temp3[31:0] ← INT32 ((SRC1[63:48] * SRC2[63:48]) >>14) + 1
temp4[31:0] ← INT32 ((SRC1[79:64] * SRC2[79:64]) >>14) + 1
temp5[31:0] ← INT32 ((SRC1[95:80] * SRC2[95:80]) >>14) + 1
temp6[31:0] ← INT32 ((SRC1[111:96] * SRC2[111:96]) >>14) + 1
temp7[31:0] ← INT32 ((SRC1[127:112] * SRC2[127:112]) >>14) + 1
temp8[31:0] ← INT32 ((SRC1[143:128] * SRC2[143:128]) >>14) + 1
temp9[31:0] ← INT32 ((SRC1[159:144] * SRC2[159:144]) >>14) + 1
temp10[31:0] ← INT32 ((SRC1[175:160] * SRC2[175:160]) >>14) + 1
temp11[31:0] ← INT32 ((SRC1[191:176] * SRC2[191:176]) >>14) + 1
temp12[31:0] ← INT32 ((SRC1[207:192] * SRC2[207:192]) >>14) + 1
temp13[31:0] ← INT32 ((SRC1[223:208] * SRC2[223:208]) >>14) + 1
temp14[31:0] ← INT32 ((SRC1[239:224] * SRC2[239:224]) >>14) + 1
temp15[31:0] ← INT32 ((SRC1[255:240] * SRC2[255:240]) >>14) + 1

```

```

DEST[15:0] ← temp0[16:1]
DEST[31:16] ← temp1[16:1]
DEST[47:32] ← temp2[16:1]
DEST[63:48] ← temp3[16:1]
DEST[79:64] ← temp4[16:1]
DEST[95:80] ← temp5[16:1]
DEST[111:96] ← temp6[16:1]
DEST[127:112] ← temp7[16:1]
DEST[143:128] ← temp8[16:1]
DEST[159:144] ← temp9[16:1]
DEST[175:160] ← temp10[16:1]
DEST[191:176] ← temp11[16:1]
DEST[207:192] ← temp12[16:1]
DEST[223:208] ← temp13[16:1]
DEST[239:224] ← temp14[16:1]
DEST[255:240] ← temp15[16:1]
DEST[MAX_VL-1:256] ← 0

```

**VPMULHRWSW (EVEX encoded version)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j ← 0 TO KL-1

  i ← j \* 16

  IF k1[j] OR \*no writemask\*

    THEN

      temp[31:0] ← ((SRC1[i+15:i] \* SRC2[i+15:i]) >>14) + 1

      DEST[i+15:i] ← tmp[16:1]

    ELSE

      IF \*merging-masking\* ; merging-masking

        THEN \*DEST[i+15:i] remains unchanged\*

```

        ELSE *zeroing-masking*          ; zeroing-masking
            DEST[i+15:i] ← 0
        FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

### Intel C/C++ Compiler Intrinsic Equivalents

```

VPMULHRSW __m512i __mm512_mulhrs_epi16(__m512i a, __m512i b);
VPMULHRSW __m512i __mm512_mask_mulhrs_epi16(__m512i s, __mmask32 k, __m512i a, __m512i b);
VPMULHRSW __m512i __mm512_maskz_mulhrs_epi16(__mmask32 k, __m512i a, __m512i b);
VPMULHRSW __m256i __mm256_mask_mulhrs_epi16(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPMULHRSW __m256i __mm256_maskz_mulhrs_epi16(__mmask16 k, __m256i a, __m256i b);
VPMULHRSW __m128i __mm_mask_mulhrs_epi16(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMULHRSW __m128i __mm_maskz_mulhrs_epi16(__mmask8 k, __m128i a, __m128i b);
PMULHRSW: __m64 __mm_mulhrs_pi16(__m64 a, __m64 b)
(V)PMULHRSW: __m128i __mm_mulhrs_epi16(__m128i a, __m128i b)
VPMULHRSW: __m256i __mm256_mulhrs_epi16(__m256i a, __m256i b)

```

### SIMD Floating-Point Exceptions

None.

### Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.  
 EVEX-encoded instruction, see Exceptions Type E4.nb.

## PMULHUW—Multiply Packed Unsigned Integers and Store High Result

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F E4 /r <sup>1</sup> PMULHUW <i>mm1, mm2/m64</i>	RM	V/V	SSE	Multiply the packed unsigned word integers in <i>mm1</i> register and <i>mm2/m64</i> , and store the high 16 bits of the results in <i>mm1</i> .
66 0F E4 /r PMULHUW <i>xmm1, xmm2/m128</i>	RM	V/V	SSE2	Multiply the packed unsigned word integers in <i>xmm1</i> and <i>xmm2/m128</i> , and store the high 16 bits of the results in <i>xmm1</i> .
VEX.NDS.128.66.0F.WIG E4 /r VPMULHUW <i>xmm1, xmm2, xmm3/m128</i>	RVM	V/V	AVX	Multiply the packed unsigned word integers in <i>xmm2</i> and <i>xmm3/m128</i> , and store the high 16 bits of the results in <i>xmm1</i> .
VEX.NDS.256.66.0F.WIG E4 /r VPMULHUW <i>ymm1, ymm2, ymm3/m256</i>	RVM	V/V	AVX2	Multiply the packed unsigned word integers in <i>ymm2</i> and <i>ymm3/m256</i> , and store the high 16 bits of the results in <i>ymm1</i> .
EVEX.NDS.128.66.0F.WIG E4 /r VPMULHUW <i>xmm1 {k1}{z}, xmm2, xmm3/m128</i>	FVM	V/V	AVX512VL AVX512BW	Multiply the packed unsigned word integers in <i>xmm2</i> and <i>xmm3/m128</i> , and store the high 16 bits of the results in <i>xmm1</i> under writemask <i>k1</i> .
EVEX.NDS.256.66.0F.WIG E4 /r VPMULHUW <i>ymm1 {k1}{z}, ymm2, ymm3/m256</i>	FVM	V/V	AVX512VL AVX512BW	Multiply the packed unsigned word integers in <i>ymm2</i> and <i>ymm3/m256</i> , and store the high 16 bits of the results in <i>ymm1</i> under writemask <i>k1</i> .
EVEX.NDS.512.66.0F.WIG E4 /r VPMULHUW <i>zmm1 {k1}{z}, zmm2, zmm3/m512</i>	FVM	V/V	AVX512BW	Multiply the packed unsigned word integers in <i>zmm2</i> and <i>zmm3/m512</i> , and store the high 16 bits of the results in <i>zmm1</i> under writemask <i>k1</i> .

### NOTES:

1. See note in Section 2.4, “AVX and SSE Instruction Exception Specification” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FVM	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Performs a SIMD unsigned multiply of the packed unsigned word integers in the destination operand (first operand) and the source operand (second operand), and stores the high 16 bits of each 32-bit intermediate results in the destination operand. (Figure 4-8 shows this operation when using 64-bit operands.)

- In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).
- Legacy SSE version 64-bit operand: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand is an MMX technology register.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers.

EVEX encoded versions: The first source 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. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

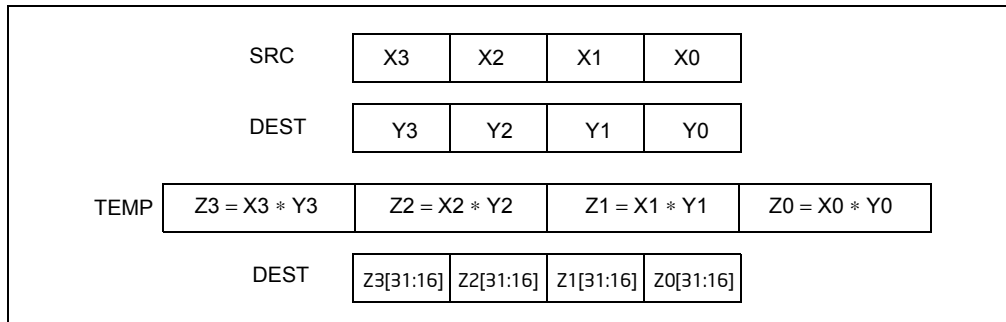


Figure 4-8. PMULHUW and PMULHW Instruction Operation Using 64-bit Operands

## Operation

### PMULHUW (with 64-bit operands)

```

TEMP0[31:0] ← DEST[15:0] * SRC[15:0]; (* Unsigned multiplication *)
TEMP1[31:0] ← DEST[31:16] * SRC[31:16];
TEMP2[31:0] ← DEST[47:32] * SRC[47:32];
TEMP3[31:0] ← DEST[63:48] * SRC[63:48];
DEST[15:0] ← TEMP0[31:16];
DEST[31:16] ← TEMP1[31:16];
DEST[47:32] ← TEMP2[31:16];
DEST[63:48] ← TEMP3[31:16];

```

### PMULHUW (with 128-bit operands)

```

TEMP0[31:0] ← DEST[15:0] * SRC[15:0]; (* Unsigned multiplication *)
TEMP1[31:0] ← DEST[31:16] * SRC[31:16];
TEMP2[31:0] ← DEST[47:32] * SRC[47:32];
TEMP3[31:0] ← DEST[63:48] * SRC[63:48];
TEMP4[31:0] ← DEST[79:64] * SRC[79:64];
TEMP5[31:0] ← DEST[95:80] * SRC[95:80];
TEMP6[31:0] ← DEST[111:96] * SRC[111:96];
TEMP7[31:0] ← DEST[127:112] * SRC[127:112];
DEST[15:0] ← TEMP0[31:16];
DEST[31:16] ← TEMP1[31:16];
DEST[47:32] ← TEMP2[31:16];
DEST[63:48] ← TEMP3[31:16];
DEST[79:64] ← TEMP4[31:16];
DEST[95:80] ← TEMP5[31:16];
DEST[111:96] ← TEMP6[31:16];
DEST[127:112] ← TEMP7[31:16];

```

### VPMULHUW (VEX.128 encoded version)

```

TEMP0[31:0] ← SRC1[15:0] * SRC2[15:0]

```

TEMP1[31:0] ← SRC1[31:16] \* SRC2[31:16]  
 TEMP2[31:0] ← SRC1[47:32] \* SRC2[47:32]  
 TEMP3[31:0] ← SRC1[63:48] \* SRC2[63:48]  
 TEMP4[31:0] ← SRC1[79:64] \* SRC2[79:64]  
 TEMP5[31:0] ← SRC1[95:80] \* SRC2[95:80]  
 TEMP6[31:0] ← SRC1[111:96] \* SRC2[111:96]  
 TEMP7[31:0] ← SRC1[127:112] \* SRC2[127:112]  
 DEST[15:0] ← TEMPO[31:16]  
 DEST[31:16] ← TEMP1[31:16]  
 DEST[47:32] ← TEMP2[31:16]  
 DEST[63:48] ← TEMP3[31:16]  
 DEST[79:64] ← TEMP4[31:16]  
 DEST[95:80] ← TEMP5[31:16]  
 DEST[111:96] ← TEMP6[31:16]  
 DEST[127:112] ← TEMP7[31:16]  
 DEST[VLMAX-1:128] ← 0

**PMULHUW (VEX.256 encoded version)**

TEMPO[31:0] ← SRC1[15:0] \* SRC2[15:0]  
 TEMP1[31:0] ← SRC1[31:16] \* SRC2[31:16]  
 TEMP2[31:0] ← SRC1[47:32] \* SRC2[47:32]  
 TEMP3[31:0] ← SRC1[63:48] \* SRC2[63:48]  
 TEMP4[31:0] ← SRC1[79:64] \* SRC2[79:64]  
 TEMP5[31:0] ← SRC1[95:80] \* SRC2[95:80]  
 TEMP6[31:0] ← SRC1[111:96] \* SRC2[111:96]  
 TEMP7[31:0] ← SRC1[127:112] \* SRC2[127:112]  
 TEMP8[31:0] ← SRC1[143:128] \* SRC2[143:128]  
 TEMP9[31:0] ← SRC1[159:144] \* SRC2[159:144]  
 TEMP10[31:0] ← SRC1[175:160] \* SRC2[175:160]  
 TEMP11[31:0] ← SRC1[191:176] \* SRC2[191:176]  
 TEMP12[31:0] ← SRC1[207:192] \* SRC2[207:192]  
 TEMP13[31:0] ← SRC1[223:208] \* SRC2[223:208]  
 TEMP14[31:0] ← SRC1[239:224] \* SRC2[239:224]  
 TEMP15[31:0] ← SRC1[255:240] \* SRC2[255:240]  
 DEST[15:0] ← TEMPO[31:16]  
 DEST[31:16] ← TEMP1[31:16]  
 DEST[47:32] ← TEMP2[31:16]  
 DEST[63:48] ← TEMP3[31:16]  
 DEST[79:64] ← TEMP4[31:16]  
 DEST[95:80] ← TEMP5[31:16]  
 DEST[111:96] ← TEMP6[31:16]  
 DEST[127:112] ← TEMP7[31:16]  
 DEST[143:128] ← TEMP8[31:16]  
 DEST[159:144] ← TEMP9[31:16]  
 DEST[175:160] ← TEMP10[31:16]  
 DEST[191:176] ← TEMP11[31:16]  
 DEST[207:192] ← TEMP12[31:16]  
 DEST[223:208] ← TEMP13[31:16]  
 DEST[239:224] ← TEMP14[31:16]  
 DEST[255:240] ← TEMP15[31:16]  
 DEST[MAX\_VL-1:256] ← 0

**PMULHUW (EVEX encoded versions)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 16
  IF k1[j] OR *no writemask*
    THEN
      temp[31:0] ← SRC1[j+15:i] * SRC2[j+15:i]
      DEST[j+15:i] ← tmp[31:16]
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[j+15:i] remains unchanged*
      ELSE *zeroing-masking* ; zeroing-masking
        DEST[j+15:i] ← 0
      FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

### Intel C/C++ Compiler Intrinsic Equivalent

```

VPMULHUW __m512i __mm512_mulhi_epu16(__m512i a, __m512i b);
VPMULHUW __m512i __mm512_mask_mulhi_epu16(__m512i s, __mmask32 k, __m512i a, __m512i b);
VPMULHUW __m512i __mm512_maskz_mulhi_epu16(__mmask32 k, __m512i a, __m512i b);
VPMULHUW __m256i __mm256_mask_mulhi_epu16(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPMULHUW __m256i __mm256_maskz_mulhi_epu16(__mmask16 k, __m256i a, __m256i b);
VPMULHUW __m128i __mm_mask_mulhi_epu16(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMULHUW __m128i __mm_maskz_mulhi_epu16(__mmask8 k, __m128i a, __m128i b);
PMULHUW: __m64 __mm_mulhi_epu16(__m64 a, __m64 b)
(V)PMULHUW: __m128i __mm_mulhi_epu16 (__m128i a, __m128i b)
VPMULHUW: __m256i __mm256_mulhi_epu16 (__m256i a, __m256i b)

```

### Flags Affected

None.

### Numeric Exceptions

None.

### Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.nb.

## PMULHW—Multiply Packed Signed Integers and Store High Result

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F E5 /r <sup>1</sup> PMULHW <i>mm</i> , <i>mm/m64</i>	RM	V/V	MMX	Multiply the packed signed word integers in <i>mm1</i> register and <i>mm2/m64</i> , and store the high 16 bits of the results in <i>mm1</i> .
66 0F E5 /r PMULHW <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE2	Multiply the packed signed word integers in <i>xmm1</i> and <i>xmm2/m128</i> , and store the high 16 bits of the results in <i>xmm1</i> .
VEX.NDS.128.66.0F.WIG E5 /r VPMULHW <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Multiply the packed signed word integers in <i>xmm2</i> and <i>xmm3/m128</i> , and store the high 16 bits of the results in <i>xmm1</i> .
VEX.NDS.256.66.0F.WIG E5 /r VPMULHW <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	RVM	V/V	AVX2	Multiply the packed signed word integers in <i>ymm2</i> and <i>ymm3/m256</i> , and store the high 16 bits of the results in <i>ymm1</i> .
EVEX.NDS.128.66.0F.WIG E5 /r VPMULHW <i>xmm1</i> {k1}{z}, <i>xmm2</i> , <i>xmm3/m128</i>	FVM	V/V	AVX512VL AVX512BW	Multiply the packed signed word integers in <i>xmm2</i> and <i>xmm3/m128</i> , and store the high 16 bits of the results in <i>xmm1</i> under writemask k1.
EVEX.NDS.256.66.0F.WIG E5 /r VPMULHW <i>ymm1</i> {k1}{z}, <i>ymm2</i> , <i>ymm3/m256</i>	FVM	V/V	AVX512VL AVX512BW	Multiply the packed signed word integers in <i>ymm2</i> and <i>ymm3/m256</i> , and store the high 16 bits of the results in <i>ymm1</i> under writemask k1.
EVEX.NDS.512.66.0F.WIG E5 /r VPMULHW <i>zmm1</i> {k1}{z}, <i>zmm2</i> , <i>zmm3/m512</i>	FVM	V/V	AVX512BW	Multiply the packed signed word integers in <i>zmm2</i> and <i>zmm3/m512</i> , and store the high 16 bits of the results in <i>zmm1</i> under writemask k1.

### NOTES:

1. See note in Section 2.4, “AVX and SSE Instruction Exception Specification” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FVM	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Performs a SIMD signed multiply of the packed signed word integers in the destination operand (first operand) and the source operand (second operand), and stores the high 16 bits of each intermediate 32-bit result in the destination operand. (Figure 4-8 shows this operation when using 64-bit operands.)

■ In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

■ Legacy SSE version 64-bit operand: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand is an MMX technology register.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.



VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers.

EVEX encoded versions: The first source 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. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

## Operation

### PMULHW (with 64-bit operands)

```

TEMP0[31:0] ← DEST[15:0] * SRC[15:0]; (* Signed multiplication *)
TEMP1[31:0] ← DEST[31:16] * SRC[31:16];
TEMP2[31:0] ← DEST[47:32] * SRC[47:32];
TEMP3[31:0] ← DEST[63:48] * SRC[63:48];
DEST[15:0] ← TEMP0[31:16];
DEST[31:16] ← TEMP1[31:16];
DEST[47:32] ← TEMP2[31:16];
DEST[63:48] ← TEMP3[31:16];

```

### PMULHW (with 128-bit operands)

```

TEMP0[31:0] ← DEST[15:0] * SRC[15:0]; (* Signed multiplication *)
TEMP1[31:0] ← DEST[31:16] * SRC[31:16];
TEMP2[31:0] ← DEST[47:32] * SRC[47:32];
TEMP3[31:0] ← DEST[63:48] * SRC[63:48];
TEMP4[31:0] ← DEST[79:64] * SRC[79:64];
TEMP5[31:0] ← DEST[95:80] * SRC[95:80];
TEMP6[31:0] ← DEST[111:96] * SRC[111:96];
TEMP7[31:0] ← DEST[127:112] * SRC[127:112];
DEST[15:0] ← TEMP0[31:16];
DEST[31:16] ← TEMP1[31:16];
DEST[47:32] ← TEMP2[31:16];
DEST[63:48] ← TEMP3[31:16];
DEST[79:64] ← TEMP4[31:16];
DEST[95:80] ← TEMP5[31:16];
DEST[111:96] ← TEMP6[31:16];
DEST[127:112] ← TEMP7[31:16];

```

### VPMULHW (VEX.128 encoded version)

```

TEMP0[31:0] ← SRC1[15:0] * SRC2[15:0] (*Signed Multiplication*)
TEMP1[31:0] ← SRC1[31:16] * SRC2[31:16]
TEMP2[31:0] ← SRC1[47:32] * SRC2[47:32]
TEMP3[31:0] ← SRC1[63:48] * SRC2[63:48]
TEMP4[31:0] ← SRC1[79:64] * SRC2[79:64]
TEMP5[31:0] ← SRC1[95:80] * SRC2[95:80]
TEMP6[31:0] ← SRC1[111:96] * SRC2[111:96]
TEMP7[31:0] ← SRC1[127:112] * SRC2[127:112]
DEST[15:0] ← TEMP0[31:16]
DEST[31:16] ← TEMP1[31:16]
DEST[47:32] ← TEMP2[31:16]
DEST[63:48] ← TEMP3[31:16]
DEST[79:64] ← TEMP4[31:16]
DEST[95:80] ← TEMP5[31:16]
DEST[111:96] ← TEMP6[31:16]

```

DEST[127:112] ← TEMP7[31:16]  
 DEST[VLMAX-1:128] ← 0

#### PMULHW (VEX.256 encoded version)

TEMP0[31:0] ← SRC1[15:0] \* SRC2[15:0] (\*Signed Multiplication\*)  
 TEMP1[31:0] ← SRC1[31:16] \* SRC2[31:16]  
 TEMP2[31:0] ← SRC1[47:32] \* SRC2[47:32]  
 TEMP3[31:0] ← SRC1[63:48] \* SRC2[63:48]  
 TEMP4[31:0] ← SRC1[79:64] \* SRC2[79:64]  
 TEMP5[31:0] ← SRC1[95:80] \* SRC2[95:80]  
 TEMP6[31:0] ← SRC1[111:96] \* SRC2[111:96]  
 TEMP7[31:0] ← SRC1[127:112] \* SRC2[127:112]  
 TEMP8[31:0] ← SRC1[143:128] \* SRC2[143:128]  
 TEMP9[31:0] ← SRC1[159:144] \* SRC2[159:144]  
 TEMP10[31:0] ← SRC1[175:160] \* SRC2[175:160]  
 TEMP11[31:0] ← SRC1[191:176] \* SRC2[191:176]  
 TEMP12[31:0] ← SRC1[207:192] \* SRC2[207:192]  
 TEMP13[31:0] ← SRC1[223:208] \* SRC2[223:208]  
 TEMP14[31:0] ← SRC1[239:224] \* SRC2[239:224]  
 TEMP15[31:0] ← SRC1[255:240] \* SRC2[255:240]  
 DEST[15:0] ← TEMP0[31:16]  
 DEST[31:16] ← TEMP1[31:16]  
 DEST[47:32] ← TEMP2[31:16]  
 DEST[63:48] ← TEMP3[31:16]  
 DEST[79:64] ← TEMP4[31:16]  
 DEST[95:80] ← TEMP5[31:16]  
 DEST[111:96] ← TEMP6[31:16]  
 DEST[127:112] ← TEMP7[31:16]  
 DEST[143:128] ← TEMP8[31:16]  
 DEST[159:144] ← TEMP9[31:16]  
 DEST[175:160] ← TEMP10[31:16]  
 DEST[191:176] ← TEMP11[31:16]  
 DEST[207:192] ← TEMP12[31:16]  
 DEST[223:208] ← TEMP13[31:16]  
 DEST[239:224] ← TEMP14[31:16]  
 DEST[255:240] ← TEMP15[31:16]  
 DEST[VLMAX-1:256] ← 0

#### PMULHW (EVEX encoded versions)

(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j ← 0 TO KL-1

  i ← j \* 16

  IF k1[j] OR \*no writemask\*

    THEN

      temp[31:0] ← SRC1[j+15:i] \* SRC2[j+15:i]

      DEST[j+15:i] ← temp[31:16]

    ELSE

      IF \*merging-masking\* ; merging-masking

        THEN \*DEST[j+15:i] remains unchanged\*

        ELSE \*zeroing-masking\* ; zeroing-masking

          DEST[j+15:i] ← 0

    FI

  FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

### Intel C/C++ Compiler Intrinsic Equivalent

VPMULHW \_\_m512i \_mm512\_mulhi\_epi16(\_\_m512i a, \_\_m512i b);  
 VPMULHW \_\_m512i \_mm512\_mask\_mulhi\_epi16(\_\_m512i s, \_\_mmask32 k, \_\_m512i a, \_\_m512i b);  
 VPMULHW \_\_m512i \_mm512\_maskz\_mulhi\_epi16(\_\_mmask32 k, \_\_m512i a, \_\_m512i b);  
 VPMULHW \_\_m256i \_mm256\_mask\_mulhi\_epi16(\_\_m256i s, \_\_mmask16 k, \_\_m256i a, \_\_m256i b);  
 VPMULHW \_\_m256i \_mm256\_maskz\_mulhi\_epi16(\_\_mmask16 k, \_\_m256i a, \_\_m256i b);  
 VPMULHW \_\_m128i \_mm\_mask\_mulhi\_epi16(\_\_m128i s, \_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
 VPMULHW \_\_m128i \_mm\_maskz\_mulhi\_epi16(\_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
 PMULHW: \_\_m64 \_mm\_mulhi\_pi16(\_\_m64 m1, \_\_m64 m2)  
 (V)PMULHW: \_\_m128i \_mm\_mulhi\_epi16(\_\_m128i a, \_\_m128i b)  
 VPMULHW: \_\_m256i \_mm256\_mulhi\_epi16(\_\_m256i a, \_\_m256i b)

### Flags Affected

None.

### SIMD Floating-Point Exceptions

None.

### Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.nb.

## PMULLD/PMULLQ—Multiply Packed Integers and Store Low Result

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 40 /r PMULLD xmm1, xmm2/m128	RM	V/V	SSE4_1	Multiply the packed dword signed integers in xmm1 and xmm2/m128 and store the low 32 bits of each product in xmm1.
VEX.NDS.128.66.0F38.WIG 40 /r VPMULLD xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Multiply the packed dword signed integers in xmm2 and xmm3/m128 and store the low 32 bits of each product in xmm1.
VEX.NDS.256.66.0F38.WIG 40 /r VPMULLD ymm1, ymm2, ymm3/m256	RVM	V/V	AVX2	Multiply the packed dword signed integers in ymm2 and ymm3/m256 and store the low 32 bits of each product in ymm1.
EVEX.NDS.128.66.0F38.W0 40 /r VPMULLD xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	FV	V/V	AVX512VL AVX512F	Multiply the packed dword signed integers in xmm2 and xmm3/m128/m32bcst and store the low 32 bits of each product in xmm1 under writemask k1.
EVEX.NDS.256.66.0F38.W0 40 /r VPMULLD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	FV	V/V	AVX512VL AVX512F	Multiply the packed dword signed integers in ymm2 and ymm3/m256/m32bcst and store the low 32 bits of each product in ymm1 under writemask k1.
EVEX.NDS.512.66.0F38.W0 40 /r VPMULLD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	FV	V/V	AVX512F	Multiply the packed dword signed integers in zmm2 and zmm3/m512/m32bcst and store the low 32 bits of each product in zmm1 under writemask k1.
EVEX.NDS.128.66.0F38.W1 40 /r VPMULLQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	FV	V/V	AVX512VL AVX512DQ	Multiply the packed qword signed integers in xmm2 and xmm3/m128/m64bcst and store the low 64 bits of each product in xmm1 under writemask k1.
EVEX.NDS.256.66.0F38.W1 40 /r VPMULLQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	FV	V/V	AVX512VL AVX512DQ	Multiply the packed qword signed integers in ymm2 and ymm3/m256/m64bcst and store the low 64 bits of each product in ymm1 under writemask k1.
EVEX.NDS.512.66.0F38.W1 40 /r VPMULLQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	FV	V/V	AVX512DQ	Multiply the packed qword signed integers in zmm2 and zmm3/m512/m64bcst and store the low 64 bits of each product in zmm1 under writemask k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FV	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Performs a SIMD signed multiply of the packed signed dword/qword integers from each element of the first source operand with the corresponding element in the second source operand. The low 32/64 bits of each 64/128-bit intermediate results are stored to the destination operand.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAX\_VL-1:128) of the corresponding ZMM destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAX\_VL-1:128) of the corresponding ZMM register are zeroed.

VEX.256 encoded version: The first source operand is a YMM register; The second source operand is a YMM register or 256-bit memory location. Bits (MAX\_VL-1:256) of the corresponding destination ZMM register are zeroed.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second 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 32/64-bit memory location. The destination operand is conditionally updated based on writemask k1.

## Operation

### VPMULLQ (EVEX encoded versions)

(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) AND (SRC2 \*is memory\*)

      THEN Temp[127:0] ← SRC1[i+63:i] \* SRC2[63:0]

      ELSE Temp[127:0] ← SRC1[i+63:i] \* SRC2[i+63:i]

    FI;

    DEST[i+63:i] ← Temp[63:0]

  ELSE

    IF \*merging-masking\* ; merging-masking

      THEN \*DEST[i+63:i] remains unchanged\*

      ELSE ; zeroing-masking

        DEST[i+63:i] ← 0

    FI

  FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

### VPMULLD (EVEX encoded versions)

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

  i ← j \* 32

  IF k1[j] OR \*no writemask\* THEN

    IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

      THEN Temp[63:0] ← SRC1[i+31:i] \* SRC2[31:0]

      ELSE Temp[63:0] ← SRC1[i+31:i] \* SRC2[i+31:i]

    FI;

    DEST[i+31:i] ← Temp[31:0]

  ELSE

    IF \*merging-masking\* ; merging-masking

      THEN \*DEST[i+31:i] remains unchanged\*

      ELSE ; zeroing-masking

        DEST[i+31:i] ← 0

    FI

  FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VPMULLD (VEX.256 encoded version)**

Temp0[63:0] ← SRC1[31:0] \* SRC2[31:0]  
 Temp1[63:0] ← SRC1[63:32] \* SRC2[63:32]  
 Temp2[63:0] ← SRC1[95:64] \* SRC2[95:64]  
 Temp3[63:0] ← SRC1[127:96] \* SRC2[127:96]  
 Temp4[63:0] ← SRC1[159:128] \* SRC2[159:128]  
 Temp5[63:0] ← SRC1[191:160] \* SRC2[191:160]  
 Temp6[63:0] ← SRC1[223:192] \* SRC2[223:192]  
 Temp7[63:0] ← SRC1[255:224] \* SRC2[255:224]

DEST[31:0] ← Temp0[31:0]  
 DEST[63:32] ← Temp1[31:0]  
 DEST[95:64] ← Temp2[31:0]  
 DEST[127:96] ← Temp3[31:0]  
 DEST[159:128] ← Temp4[31:0]  
 DEST[191:160] ← Temp5[31:0]  
 DEST[223:192] ← Temp6[31:0]  
 DEST[255:224] ← Temp7[31:0]  
 DEST[MAX\_VL-1:256] ← 0

**VPMULLD (VEX.128 encoded version)**

Temp0[63:0] ← SRC1[31:0] \* SRC2[31:0]  
 Temp1[63:0] ← SRC1[63:32] \* SRC2[63:32]  
 Temp2[63:0] ← SRC1[95:64] \* SRC2[95:64]  
 Temp3[63:0] ← SRC1[127:96] \* SRC2[127:96]  
 DEST[31:0] ← Temp0[31:0]  
 DEST[63:32] ← Temp1[31:0]  
 DEST[95:64] ← Temp2[31:0]  
 DEST[127:96] ← Temp3[31:0]  
 DEST[MAX\_VL-1:128] ← 0

**PMULLD (128-bit Legacy SSE version)**

Temp0[63:0] ← DEST[31:0] \* SRC[31:0]  
 Temp1[63:0] ← DEST[63:32] \* SRC[63:32]  
 Temp2[63:0] ← DEST[95:64] \* SRC[95:64]  
 Temp3[63:0] ← DEST[127:96] \* SRC[127:96]  
 DEST[31:0] ← Temp0[31:0]  
 DEST[63:32] ← Temp1[31:0]  
 DEST[95:64] ← Temp2[31:0]  
 DEST[127:96] ← Temp3[31:0]  
 DEST[MAX\_VL-1:128] (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**

VPMULLD \_\_m512i\_mm512\_mullo\_epi32(\_\_m512i a, \_\_m512i b);  
 VPMULLD \_\_m512i\_mm512\_mask\_mullo\_epi32(\_\_m512i s, \_\_mmask16 k, \_\_m512i a, \_\_m512i b);  
 VPMULLD \_\_m512i\_mm512\_maskz\_mullo\_epi32(\_\_mmask16 k, \_\_m512i a, \_\_m512i b);  
 VPMULLD \_\_m256i\_mm256\_mask\_mullo\_epi32(\_\_m256i s, \_\_mmask8 k, \_\_m256i a, \_\_m256i b);  
 VPMULLD \_\_m256i\_mm256\_maskz\_mullo\_epi32(\_\_mmask8 k, \_\_m256i a, \_\_m256i b);  
 VPMULLD \_\_m128i\_mm\_mask\_mullo\_epi32(\_\_m128i s, \_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
 VPMULLD \_\_m128i\_mm\_maskz\_mullo\_epi32(\_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
 VPMULLD \_\_m256i\_mm256\_mullo\_epi32(\_\_m256i a, \_\_m256i b);  
 PMULLD \_\_m128i\_mm\_mullo\_epi32(\_\_m128i a, \_\_m128i b);  
 VPMULLQ \_\_m512i\_mm512\_mullo\_epi64(\_\_m512i a, \_\_m512i b);  
 VPMULLQ \_\_m512i\_mm512\_mask\_mullo\_epi64(\_\_m512i s, \_\_mmask8 k, \_\_m512i a, \_\_m512i b);

VPMULLQ \_\_m512i \_mm512\_maskz\_mullo\_epi64( \_\_mmask8 k, \_\_m512i a, \_\_m512i b);  
VPMULLQ \_\_m256i \_mm256\_mullo\_epi64(\_\_m256i a, \_\_m256i b);  
VPMULLQ \_\_m256i \_mm256\_mask\_mullo\_epi64(\_\_m256i s, \_\_mmask8 k, \_\_m256i a, \_\_m256i b);  
VPMULLQ \_\_m256i \_mm256\_maskz\_mullo\_epi64( \_\_mmask8 k, \_\_m256i a, \_\_m256i b);  
VPMULLQ \_\_m128i \_mm\_mullo\_epi64(\_\_m128i a, \_\_m128i b);  
VPMULLQ \_\_m128i \_mm\_mask\_mullo\_epi64(\_\_m128i s, \_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
VPMULLQ \_\_m128i \_mm\_maskz\_mullo\_epi64( \_\_mmask8 k, \_\_m128i a, \_\_m128i b);

### SIMD Floating-Point Exceptions

None

### Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.

## PMULLW—Multiply Packed Signed Integers and Store Low Result

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F D5 /r <sup>1</sup> PMULLW <i>mm</i> , <i>mm/m64</i>	RM	V/V	MMX	Multiply the packed signed word integers in <i>mm1</i> register and <i>mm2/m64</i> , and store the low 16 bits of the results in <i>mm1</i> .
66 0F D5 /r PMULLW <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE2	Multiply the packed signed word integers in <i>xmm1</i> and <i>xmm2/m128</i> , and store the low 16 bits of the results in <i>xmm1</i> .
VEX.NDS.128.66.0F.WIG D5 /r VPMULLW <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Multiply the packed dword signed integers in <i>xmm2</i> and <i>xmm3/m128</i> and store the low 32 bits of each product in <i>xmm1</i> .
VEX.NDS.256.66.0F.WIG D5 /r VPMULLW <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	RVM	V/V	AVX2	Multiply the packed signed word integers in <i>ymm2</i> and <i>ymm3/m256</i> , and store the low 16 bits of the results in <i>ymm1</i> .
EVEX.NDS.128.66.0F.WIG D5 /r VPMULLW <i>xmm1</i> {k1}{z}, <i>xmm2</i> , <i>xmm3/m128</i>	FVM	V/V	AVX512VL AVX512BW	Multiply the packed signed word integers in <i>xmm2</i> and <i>xmm3/m128</i> , and store the low 16 bits of the results in <i>xmm1</i> under writemask k1.
EVEX.NDS.256.66.0F.WIG D5 /r VPMULLW <i>ymm1</i> {k1}{z}, <i>ymm2</i> , <i>ymm3/m256</i>	FVM	V/V	AVX512VL AVX512BW	Multiply the packed signed word integers in <i>ymm2</i> and <i>ymm3/m256</i> , and store the low 16 bits of the results in <i>ymm1</i> under writemask k1.
EVEX.NDS.512.66.0F.WIG D5 /r VPMULLW <i>zmm1</i> {k1}{z}, <i>zmm2</i> , <i>zmm3/m512</i>	FVM	V/V	AVX512BW	Multiply the packed signed word integers in <i>zmm2</i> and <i>zmm3/m512</i> , and store the low 16 bits of the results in <i>zmm1</i> under writemask k1.

### NOTES:

1. See note in Section 2.4, “AVX and SSE Instruction Exception Specification” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FVM	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Performs a SIMD signed multiply of the packed signed word integers in the destination operand (first operand) and the source operand (second operand), and stores the low 16 bits of each intermediate 32-bit result in the destination operand. (Figure 4-8 shows this operation when using 64-bit operands.)

- In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).
  - Legacy SSE version 64-bit operand: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand is an MMX technology register.
- 128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.



VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination operand is conditionally updated based on writemask k1.

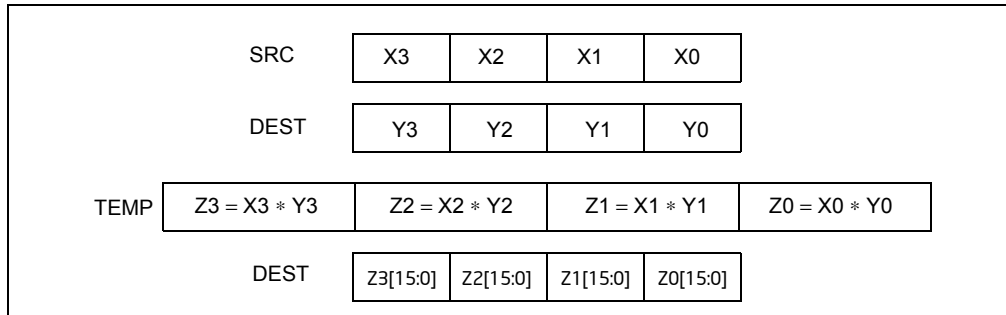


Figure 4-9. PMULLU Instruction Operation Using 64-bit Operands

## Operation

### PMULLW (with 64-bit operands)

```

TEMP0[31:0] ← DEST[15:0] * SRC[15:0]; (* Signed multiplication *)
TEMP1[31:0] ← DEST[31:16] * SRC[31:16];
TEMP2[31:0] ← DEST[47:32] * SRC[47:32];
TEMP3[31:0] ← DEST[63:48] * SRC[63:48];
DEST[15:0] ← TEMP0[15:0];
DEST[31:16] ← TEMP1[15:0];
DEST[47:32] ← TEMP2[15:0];
DEST[63:48] ← TEMP3[15:0];

```

### PMULLW (with 128-bit operands)

```

TEMP0[31:0] ← DEST[15:0] * SRC[15:0]; (* Signed multiplication *)
TEMP1[31:0] ← DEST[31:16] * SRC[31:16];
TEMP2[31:0] ← DEST[47:32] * SRC[47:32];
TEMP3[31:0] ← DEST[63:48] * SRC[63:48];
TEMP4[31:0] ← DEST[79:64] * SRC[79:64];
TEMP5[31:0] ← DEST[95:80] * SRC[95:80];
TEMP6[31:0] ← DEST[111:96] * SRC[111:96];
TEMP7[31:0] ← DEST[127:112] * SRC[127:112];
DEST[15:0] ← TEMP0[15:0];
DEST[31:16] ← TEMP1[15:0];
DEST[47:32] ← TEMP2[15:0];
DEST[63:48] ← TEMP3[15:0];
DEST[79:64] ← TEMP4[15:0];
DEST[95:80] ← TEMP5[15:0];
DEST[111:96] ← TEMP6[15:0];
DEST[127:112] ← TEMP7[15:0];
DEST[VLMAX-1:256] ← 0

```

**VPMULLW (VEX.128 encoded version)**

```

Temp0[31:0] ← SRC1[15:0] * SRC2[15:0]
Temp1[31:0] ← SRC1[31:16] * SRC2[31:16]
Temp2[31:0] ← SRC1[47:32] * SRC2[47:32]
Temp3[31:0] ← SRC1[63:48] * SRC2[63:48]
Temp4[31:0] ← SRC1[79:64] * SRC2[79:64]
Temp5[31:0] ← SRC1[95:80] * SRC2[95:80]
Temp6[31:0] ← SRC1[111:96] * SRC2[111:96]
Temp7[31:0] ← SRC1[127:112] * SRC2[127:112]
DEST[15:0] ← Temp0[15:0]
DEST[31:16] ← Temp1[15:0]
DEST[47:32] ← Temp2[15:0]
DEST[63:48] ← Temp3[15:0]
DEST[79:64] ← Temp4[15:0]
DEST[95:80] ← Temp5[15:0]
DEST[111:96] ← Temp6[15:0]
DEST[127:112] ← Temp7[15:0]
DEST[VLMAX-1:128] ← 0

```

**PMULLW (EVEX encoded versions)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j ← 0 TO KL-1

i ← j \* 16

IF k1[j] OR \*no writemask\*

THEN

temp[31:0] ← SRC1[j+15:i] \* SRC2[j+15:i]

DEST[j+15:i] ← temp[15:0]

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[j+15:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[j+15:i] ← 0

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

VPMULLW \_\_m512i \_\_mm512\_mullo\_epi16(\_\_m512i a, \_\_m512i b);

VPMULLW \_\_m512i \_\_mm512\_mask\_mullo\_epi16(\_\_m512i s, \_\_mmask32 k, \_\_m512i a, \_\_m512i b);

VPMULLW \_\_m512i \_\_mm512\_maskz\_mullo\_epi16(\_\_mmask32 k, \_\_m512i a, \_\_m512i b);

VPMULLW \_\_m256i \_\_mm256\_mask\_mullo\_epi16(\_\_m256i s, \_\_mmask16 k, \_\_m256i a, \_\_m256i b);

VPMULLW \_\_m256i \_\_mm256\_maskz\_mullo\_epi16(\_\_mmask16 k, \_\_m256i a, \_\_m256i b);

VPMULLW \_\_m128i \_\_mm\_mask\_mullo\_epi16(\_\_m128i s, \_\_mmask8 k, \_\_m128i a, \_\_m128i b);

VPMULLW \_\_m128i \_\_mm\_maskz\_mullo\_epi16(\_\_mmask8 k, \_\_m128i a, \_\_m128i b);

PMULLW: \_\_m64 \_\_mm\_mullo\_pi16(\_\_m64 m1, \_\_m64 m2)

(V)PMULLW: \_\_m128i \_\_mm\_mullo\_epi16(\_\_m128i a, \_\_m128i b)

VPMULLW: \_\_m256i \_\_mm256\_mullo\_epi16(\_\_m256i a, \_\_m256i b);

**Flags Affected**

None.

## SIMD Floating-Point Exceptions

None.

## Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.nb.

## PMULUDQ—Multiply Packed Unsigned Doubleword Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F F4 /r <sup>1</sup> PMULUDQ mm1, mm2/m64	RM	V/V	SSE2	Multiply unsigned doubleword integer in mm1 by unsigned doubleword integer in mm2/m64, and store the quadword result in mm1.
66 0F F4 /r PMULUDQ xmm1, xmm2/m128	RM	V/V	SSE2	Multiply packed unsigned doubleword integers in xmm1 by packed unsigned doubleword integers in xmm2/m128, and store the quadword results in xmm1.
VEX.NDS.128.66.0F.WIG F4 /r VPMULUDQ xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Multiply packed unsigned doubleword integers in xmm2 by packed unsigned doubleword integers in xmm3/m128, and store the quadword results in xmm1.
VEX.NDS.256.66.0F.WIG F4 /r VPMULUDQ ymm1, ymm2, ymm3/m256	RVM	V/V	AVX2	Multiply packed unsigned doubleword integers in ymm2 by packed unsigned doubleword integers in ymm3/m256, and store the quadword results in ymm1.
EVEX.NDS.128.66.0F.W1 F4 /r VPMULUDQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	FV	V/V	AVX512VL AVX512F	Multiply packed unsigned doubleword integers in xmm2 by packed unsigned doubleword integers in xmm3/m128/m64bcst, and store the quadword results in xmm1 under writemask k1.
EVEX.NDS.256.66.0F.W1 F4 /r VPMULUDQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	FV	V/V	AVX512VL AVX512F	Multiply packed unsigned doubleword integers in ymm2 by packed unsigned doubleword integers in ymm3/m256/m64bcst, and store the quadword results in ymm1 under writemask k1.
EVEX.NDS.512.66.0F.W1 F4 /r VPMULUDQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	FV	V/V	AVX512F	Multiply packed unsigned doubleword integers in zmm2 by packed unsigned doubleword integers in zmm3/m512/m64bcst, and store the quadword results in zmm1 under writemask k1.

### NOTES:

1. See note in Section 2.4, “AVX and SSE Instruction Exception Specification” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FV	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Multiplies the first operand (destination operand) by the second operand (source operand) and stores the result in the destination operand.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE version 64-bit operand: The source operand can be an unsigned doubleword integer stored in the low doubleword of an MMX technology register or a 64-bit memory location. The destination operand can be an unsigned doubleword integer stored in the low doubleword an MMX technology register. The result is an unsigned

quadword integer stored in the destination an MMX technology register. When a quadword result is too large to be represented in 64 bits (overflow), the result is wrapped around and the low 64 bits are written to the destination element (that is, the carry is ignored).

For 64-bit memory operands, 64 bits are fetched from memory, but only the low doubleword is used in the computation.

128-bit Legacy SSE version: The second source operand is two packed unsigned doubleword integers stored in the first (low) and third doublewords of an XMM register or a 128-bit memory location. For 128-bit memory operands, 128 bits are fetched from memory, but only the first and third doublewords are used in the computation. The first source operand is two packed unsigned doubleword integers stored in the first and third doublewords of an XMM register. The destination contains two packed unsigned quadword integers stored in an XMM register. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The second source operand is two packed unsigned doubleword integers stored in the first (low) and third doublewords of an XMM register or a 128-bit memory location. For 128-bit memory operands, 128 bits are fetched from memory, but only the first and third doublewords are used in the computation. The first source operand is two packed unsigned doubleword integers stored in the first and third doublewords of an XMM register. The destination contains two packed unsigned quadword integers stored in an XMM register. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The second source operand is four packed unsigned doubleword integers stored in the first (low), third, fifth and seventh doublewords of a YMM register or a 256-bit memory location. For 256-bit memory operands, 256 bits are fetched from memory, but only the first, third, fifth and seventh doublewords are used in the computation. The first source operand is four packed unsigned doubleword integers stored in the first, third, fifth and seventh doublewords of an YMM register. The destination contains four packed unaligned quadword integers stored in an YMM register.

EVEX encoded version: The input unsigned doubleword integers are taken from the even-numbered elements of the source operands. The first source operand is a ZMM/YMM/XMM registers. The second source operand can be an ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination is a ZMM/YMM/XMM register, and updated according to the writemask at 64-bit granularity.

## Operation

### PMULUDQ (with 64-Bit operands)

$$\text{DEST}[63:0] \leftarrow \text{DEST}[31:0] * \text{SRC}[31:0];$$

### PMULUDQ (with 128-Bit operands)

$$\begin{aligned} \text{DEST}[63:0] &\leftarrow \text{DEST}[31:0] * \text{SRC}[31:0]; \\ \text{DEST}[127:64] &\leftarrow \text{DEST}[95:64] * \text{SRC}[95:64]; \end{aligned}$$

### VPMULUDQ (VEX.128 encoded version)

$$\begin{aligned} \text{DEST}[63:0] &\leftarrow \text{SRC1}[31:0] * \text{SRC2}[31:0] \\ \text{DEST}[127:64] &\leftarrow \text{SRC1}[95:64] * \text{SRC2}[95:64] \\ \text{DEST}[\text{VLMAX}-1:128] &\leftarrow 0 \end{aligned}$$

### VPMULUDQ (VEX.256 encoded version)

$$\begin{aligned} \text{DEST}[63:0] &\leftarrow \text{SRC1}[31:0] * \text{SRC2}[31:0] \\ \text{DEST}[127:64] &\leftarrow \text{SRC1}[95:64] * \text{SRC2}[95:64] \\ \text{DEST}[191:128] &\leftarrow \text{SRC1}[159:128] * \text{SRC2}[159:128] \\ \text{DEST}[255:192] &\leftarrow \text{SRC1}[223:192] * \text{SRC2}[223:192] \\ \text{DEST}[\text{VLMAX}-1:256] &\leftarrow 0 \end{aligned}$$

### VPMULUDQ (EVEX encoded versions)

(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) AND (SRC2 \*is memory\*)

```

        THEN DEST[j+63:i] ← ZeroExtend64( SRC1[j+31:i]) * ZeroExtend64( SRC2[31:0])
        ELSE DEST[j+63:i] ← ZeroExtend64( SRC1[j+31:i]) * ZeroExtend64( SRC2[j+31:i])
    FI;
ELSE
    IF *merging-masking*           ; merging-masking
        THEN *DEST[j+63:i] remains unchanged*
        ELSE *zeroing-masking*     ; zeroing-masking
            DEST[j+63:i] ← 0
    FI
FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

### Intel C/C++ Compiler Intrinsic Equivalent

```

VPMULUDQ __m512i __mm512_mul_epu32(__m512i a, __m512i b);
VPMULUDQ __m512i __mm512_mask_mul_epu32(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPMULUDQ __m512i __mm512_maskz_mul_epu32(__mmask8 k, __m512i a, __m512i b);
VPMULUDQ __m256i __mm256_mask_mul_epu32(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPMULUDQ __m256i __mm256_maskz_mul_epu32(__mmask8 k, __m256i a, __m256i b);
VPMULUDQ __m128i __mm_mask_mul_epu32(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMULUDQ __m128i __mm_maskz_mul_epu32(__mmask8 k, __m128i a, __m128i b);
PMULUDQ: __m64 __mm_mul_su32(__m64 a, __m64 b)
(V)PMULUDQ: __m128i __mm_mul_epu32(__m128i a, __m128i b)
VPMULUDQ: __m256i __mm256_mul_epu32(__m256i a, __m256i b);

```

### Flags Affected

None.

### SIMD Floating-Point Exceptions

None.

### Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.

## POR—Bitwise Logical OR

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F EB /r <sup>1</sup> POR mm, mm/m64	RM	V/V	MMX	Bitwise OR of mm/m64 and mm.
66 0F EB /r POR xmm1, xmm2/m128	RM	V/V	SSE2	Bitwise OR of xmm2/m128 and xmm1.
VEX.NDS.128.66.0F.WIG EB /r VPOR xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Bitwise OR of xmm2/m128 and xmm3.
VEX.NDS.256.66.0F.WIG EB /r VPOR ymm1, ymm2, ymm3/m256	RVM	V/V	AVX2	Bitwise OR of ymm2/m256 and ymm3.
EVEX.NDS.128.66.0F.W0 EB /r VPORD xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	FV	V/V	AVX512VL AVX512F	Bitwise OR of packed doubleword integers in xmm2 and xmm3/m128/m32bcst using writemask k1.
EVEX.NDS.256.66.0F.W0 EB /r VPORD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	FV	V/V	AVX512VL AVX512F	Bitwise OR of packed doubleword integers in ymm2 and ymm3/m256/m32bcst using writemask k1.
EVEX.NDS.512.66.0F.W0 EB /r VPORD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	FV	V/V	AVX512F	Bitwise OR of packed doubleword integers in zmm2 and zmm3/m512/m32bcst using writemask k1.
EVEX.NDS.128.66.0F.W1 EB /r VPORQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	FV	V/V	AVX512VL AVX512F	Bitwise OR of packed quadword integers in xmm2 and xmm3/m128/m64bcst using writemask k1.
EVEX.NDS.256.66.0F.W1 EB /r VPORQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	FV	V/V	AVX512VL AVX512F	Bitwise OR of packed quadword integers in ymm2 and ymm3/m256/m64bcst using writemask k1.
EVEX.NDS.512.66.0F.W1 EB /r VPORQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	FV	V/V	AVX512F	Bitwise OR of packed quadword integers in zmm2 and zmm3/m512/m64bcst using writemask k1.

### NOTES:

1. See note in Section 2.4, “AVX and SSE Instruction Exception Specification” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FV	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Performs a bitwise logical OR operation on the source operand (second operand) and the destination operand (first operand) and stores the result in the destination operand. Each bit of the result is set to 1 if either or both of the corresponding bits of the first and second operands are 1; otherwise, it is set to 0.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE version: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand is an MMX technology register.

128-bit Legacy SSE version: The second source operand is an XMM register or a 128-bit memory location. The first source and destination operands can be XMM registers. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The second source operand is an XMM register or a 128-bit memory location. The first source and destination operands can be XMM registers. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The second source operand is an YMM register or a 256-bit memory location. The first source and destination operands can be YMM registers.

EVEX encoded version: The first source 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 32/64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1 at 32/64-bit granularity.

## Operation

### POR (64-bit operand)

DEST  $\leftarrow$  DEST OR SRC

### POR (128-bit Legacy SSE version)

DEST  $\leftarrow$  DEST OR SRC

DEST[VLMAX-1:128] (Unmodified)

### VPOR (VEX.128 encoded version)

DEST  $\leftarrow$  SRC1 OR SRC2

DEST[VLMAX-1:128]  $\leftarrow$  0

### VPOR (VEX.256 encoded version)

DEST  $\leftarrow$  SRC1 OR SRC2

DEST[VLMAX-1:256]  $\leftarrow$  0

### VPORD (EVEX encoded versions)

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j  $\leftarrow$  0 TO KL-1

  i  $\leftarrow$  j \* 32

  IF k1[j] OR \*no writemask\* THEN

    IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

      THEN DEST[i+31:i]  $\leftarrow$  SRC1[i+31:i] BITWISE OR SRC2[31:0]

      ELSE DEST[i+31:i]  $\leftarrow$  SRC1[i+31:i] BITWISE OR SRC2[i+31:i]

    FI;

  ELSE

    IF \*merging-masking\* ; merging-masking

      \*DEST[i+31:i] remains unchanged\*

    ELSE ; zeroing-masking

      DEST[i+31:i]  $\leftarrow$  0

    FI;

  FI;

ENDFOR;

DEST[MAX\_VL-1:VL]  $\leftarrow$  0



**Intel C/C++ Compiler Intrinsic Equivalent**

VPORD \_\_m512i \_mm512\_or\_epi32(\_\_m512i a, \_\_m512i b);  
 VPORD \_\_m512i \_mm512\_mask\_or\_epi32(\_\_m512i s, \_\_mmask16 k, \_\_m512i a, \_\_m512i b);  
 VPORD \_\_m512i \_mm512\_maskz\_or\_epi32(\_\_mmask16 k, \_\_m512i a, \_\_m512i b);  
 VPORD \_\_m256i \_mm256\_or\_epi32(\_\_m256i a, \_\_m256i b);  
 VPORD \_\_m256i \_mm256\_mask\_or\_epi32(\_\_m256i s, \_\_mmask8 k, \_\_m256i a, \_\_m256i b);  
 VPORD \_\_m256i \_mm256\_maskz\_or\_epi32(\_\_mmask8 k, \_\_m256i a, \_\_m256i b);  
 VPORD \_\_m128i \_mm\_or\_epi32(\_\_m128i a, \_\_m128i b);  
 VPORD \_\_m128i \_mm\_mask\_or\_epi32(\_\_m128i s, \_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
 VPORD \_\_m128i \_mm\_maskz\_or\_epi32(\_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
 VPORQ \_\_m512i \_mm512\_or\_epi64(\_\_m512i a, \_\_m512i b);  
 VPORQ \_\_m512i \_mm512\_mask\_or\_epi64(\_\_m512i s, \_\_mmask8 k, \_\_m512i a, \_\_m512i b);  
 VPORQ \_\_m512i \_mm512\_maskz\_or\_epi64(\_\_mmask8 k, \_\_m512i a, \_\_m512i b);  
 VPORQ \_\_m256i \_mm256\_or\_epi64(\_\_m256i a, int imm);  
 VPORQ \_\_m256i \_mm256\_mask\_or\_epi64(\_\_m256i s, \_\_mmask8 k, \_\_m256i a, \_\_m256i b);  
 VPORQ \_\_m256i \_mm256\_maskz\_or\_epi64(\_\_mmask8 k, \_\_m256i a, \_\_m256i b);  
 VPORQ \_\_m128i \_mm\_or\_epi64(\_\_m128i a, \_\_m128i b);  
 VPORQ \_\_m128i \_mm\_mask\_or\_epi64(\_\_m128i s, \_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
 VPORQ \_\_m128i \_mm\_maskz\_or\_epi64(\_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
 POR \_\_m64 \_mm\_or\_si64(\_\_m64 m1, \_\_m64 m2)  
 (V)POR: \_\_m128i \_mm\_or\_si128(\_\_m128i m1, \_\_m128i m2)  
 VPOR: \_\_m256i \_mm256\_or\_si256(\_\_m256i a, \_\_m256i b)

**Flags Affected**

None.

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.

## PSADBW—Compute Sum of Absolute Differences

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F F6 /r <sup>1</sup> PSADBW mm1, mm2/m64	RM	V/V	SSE	Computes the absolute differences of the packed unsigned byte integers from mm2/m64 and mm1; differences are then summed to produce an unsigned word integer result.
66 0F F6 /r PSADBW xmm1, xmm2/m128	RM	V/V	SSE2	Computes the absolute differences of the packed unsigned byte integers from xmm2/m128 and xmm1; the 8 low differences and 8 high differences are then summed separately to produce two unsigned word integer results.
VEX.NDS.128.66.0F.WIG F6 /r VPSADBW xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Computes the absolute differences of the packed unsigned byte integers from xmm3/m128 and xmm2; the 8 low differences and 8 high differences are then summed separately to produce two unsigned word integer results.
VEX.NDS.256.66.0F.WIG F6 /r VPSADBW ymm1, ymm2, ymm3/m256	RVM	V/V	AVX2	Computes the absolute differences of the packed unsigned byte integers from ymm3/m256 and ymm2; then each consecutive 8 differences are summed separately to produce four unsigned word integer results.
EVEX.NDS.128.66.0F.WIG F6 /r VPSADBW xmm1, xmm2, xmm3/m128	FVM	V/V	AVX512VL AVX512BW	Computes the absolute differences of the packed unsigned byte integers from xmm3/m128 and xmm2; then each consecutive 8 differences are summed separately to produce four unsigned word integer results.
EVEX.NDS.256.66.0F.WIG F6 /r VPSADBW ymm1, ymm2, ymm3/m256	FVM	V/V	AVX512VL AVX512BW	Computes the absolute differences of the packed unsigned byte integers from ymm3/m256 and ymm2; then each consecutive 8 differences are summed separately to produce four unsigned word integer results.
EVEX.NDS.512.66.0F.WIG F6 /r VPSADBW zmm1, zmm2, zmm3/m512	FVM	V/V	AVX512BW	Computes the absolute differences of the packed unsigned byte integers from zmm3/m512 and zmm2; then each consecutive 8 differences are summed separately to produce four unsigned word integer results.

### NOTES:

1. See note in Section 2.4, “AVX and SSE Instruction Exception Specification” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FVM	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

### Description

Computes the absolute value of the difference of 8 unsigned byte integers from the source operand (second

operand) and from the destination operand (first operand). These 8 differences are then summed to produce an unsigned word integer result that is stored in the destination operand. Figure 4-10 shows the operation of the PSADBW instruction when using 64-bit operands.

When operating on 64-bit operands, the word integer result is stored in the low word of the destination operand, and the remaining bytes in the destination operand are cleared to all 0s.

When operating on 128-bit operands, two packed results are computed. Here, the 8 low-order bytes of the source and destination operands are operated on to produce a word result that is stored in the low word of the destination operand, and the 8 high-order bytes are operated on to produce a word result that is stored in bits 64 through 79 of the destination operand. The remaining bytes of the destination operand are cleared.

For 256-bit version, the third group of 8 differences are summed to produce an unsigned word in bits[143:128] of the destination register and the fourth group of 8 differences are summed to produce an unsigned word in bits[207:192] of the destination register. The remaining words of the destination are set to 0.

For 512-bit version, the fifth group result is stored in bits [271:256] of the destination. The result from the sixth group is stored in bits [335:320]. The results for the seventh and eighth group are stored respectively in bits [399:384] and bits [463:447], respectively. The remaining bits in the destination are set to 0.

In 64-bit mode and not encoded by VEX/EVEX prefix, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE version: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand is an MMX technology register.

128-bit Legacy SSE version: The first source operand and destination register are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAX\_VL-1:128) of the corresponding ZMM destination register remain unchanged.

VEX.128 and EVEX.128 encoded versions: The first source operand and destination register are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAX\_VL-1:128) of the corresponding ZMM register are zeroed.

VEX.256 and EVEX.256 encoded versions: The first source operand and destination register are YMM registers. The second source operand is an YMM register or a 256-bit memory location. Bits (MAX\_VL-1:256) of the corresponding ZMM register are zeroed.

EVEX.512 encoded version: The first source operand and destination register are ZMM registers. The second source operand is a ZMM register or a 512-bit memory location.

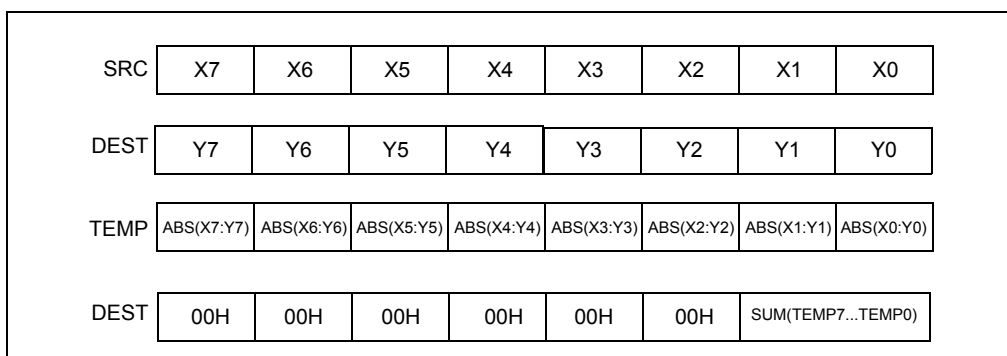


Figure 4-10. PSADBW Instruction Operation Using 64-bit Operands

### Operation

#### VPSADBW (EVEX encoded versions)

VL = 128, 256, 512

TEMP0 ← ABS(SRC1[7:0] - SRC2[7:0])

(\* Repeat operation for bytes 1 through 15 \*)

TEMP15 ← ABS(SRC1[127:120] - SRC2[127:120])

DEST[15:0] ← SUM(TEMP0:TEMP7)

DEST[63:16] ← 000000000000H

DEST[79:64] ← SUM(TEMP8:TEMP15)  
 DEST[127:80] ← 000000000000H

IF VL >= 256

(\* Repeat operation for bytes 16 through 31\*)

TEMP31 ← ABS(SRC1[255:248] - SRC2[255:248])

DEST[143:128] ← SUM(TEMP16:TEMP23)

DEST[191:144] ← 000000000000H

DEST[207:192] ← SUM(TEMP24:TEMP31)

DEST[223:208] ← 000000000000H

FI;

IF VL >= 512

(\* Repeat operation for bytes 32 through 63\*)

TEMP63 ← ABS(SRC1[511:504] - SRC2[511:504])

DEST[271:256] ← SUM(TEMP0:TEMP7)

DEST[319:272] ← 000000000000H

DEST[335:320] ← SUM(TEMP8:TEMP15)

DEST[383:336] ← 000000000000H

DEST[399:384] ← SUM(TEMP16:TEMP23)

DEST[447:400] ← 000000000000H

DEST[463:448] ← SUM(TEMP24:TEMP31)

DEST[511:464] ← 000000000000H

FI;

DEST[MAX\_VL-1:VL] ← 0

#### **VPSADBW (VEX.256 encoded version)**

TEMP0 ← ABS(SRC1[7:0] - SRC2[7:0])

(\* Repeat operation for bytes 2 through 30\*)

TEMP31 ← ABS(SRC1[255:248] - SRC2[255:248])

DEST[15:0] ← SUM(TEMP0:TEMP7)

DEST[63:16] ← 000000000000H

DEST[79:64] ← SUM(TEMP8:TEMP15)

DEST[127:80] ← 000000000000H

DEST[143:128] ← SUM(TEMP16:TEMP23)

DEST[191:144] ← 000000000000H

DEST[207:192] ← SUM(TEMP24:TEMP31)

DEST[223:208] ← 000000000000H

DEST[MAX\_VL-1:256] ← 0

**VPSADBW (VEX.128 encoded version)**

$TEMP0 \leftarrow ABS(SRC1[7:0] - SRC2[7:0])$   
 (\* Repeat operation for bytes 2 through 14 \*)  
 $TEMP15 \leftarrow ABS(SRC1[127:120] - SRC2[127:120])$   
 $DEST[15:0] \leftarrow SUM(TEMP0:TEMP7)$   
 $DEST[63:16] \leftarrow 000000000000H$   
 $DEST[79:64] \leftarrow SUM(TEMP8:TEMP15)$   
 $DEST[127:80] \leftarrow 000000000000H$   
 $DEST[MAX\_VL-1:128] \leftarrow 0$

**PSADBW (128-bit Legacy SSE version)**

$TEMP0 \leftarrow ABS(DEST[7:0] - SRC[7:0])$   
 (\* Repeat operation for bytes 2 through 14 \*)  
 $TEMP15 \leftarrow ABS(DEST[127:120] - SRC[127:120])$   
 $DEST[15:0] \leftarrow SUM(TEMP0:TEMP7)$   
 $DEST[63:16] \leftarrow 000000000000H$   
 $DEST[79:64] \leftarrow SUM(TEMP8:TEMP15)$   
 $DEST[127:80] \leftarrow 000000000000$   
 $DEST[MAX\_VL-1:128]$  (Unmodified)

**PSADBW (64-bit operand)**

$TEMP0 \leftarrow ABS(DEST[7:0] - SRC[7:0])$   
 (\* Repeat operation for bytes 2 through 6 \*)  
 $TEMP7 \leftarrow ABS(DEST[63:56] - SRC[63:56])$   
 $DEST[15:0] \leftarrow SUM(TEMP0:TEMP7)$   
 $DEST[63:16] \leftarrow 000000000000H$

**Intel C/C++ Compiler Intrinsic Equivalent**

VPSADBW \_\_m512i \_mm512\_sad\_epu8(\_\_m512i a, \_\_m512i b)  
 PSADBW: \_\_m64 \_mm\_sad\_pu8(\_\_m64 a, \_\_m64 b)  
 (V)PSADBW: \_\_m128i \_mm\_sad\_epu8(\_\_m128i a, \_\_m128i b)  
 VPSADBW: \_\_m256i \_mm256\_sad\_epu8(\_\_m256i a, \_\_m256i b)

**Flags Affected**

None.

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 4.  
 EVEX-encoded instruction, see Exceptions Type E4NF.nb.

## PSHUFB — Packed Shuffle Bytes

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F 38 00 /r <sup>1</sup> PSHUFB <i>mm1</i> , <i>mm2/m64</i>	RM	V/V	SSSE3	Shuffle bytes in <i>mm1</i> according to contents of <i>mm2/m64</i> .
66 0F 38 00 /r PSHUFB <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSSE3	Shuffle bytes in <i>xmm1</i> according to contents of <i>xmm2/m128</i> .
VEX.NDS.128.66.0F38.WIG 00 /r VPSHUFB <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Shuffle bytes in <i>xmm2</i> according to contents of <i>xmm3/m128</i> .
VEX.NDS.256.66.0F38.WIG 00 /r VPSHUFB <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	RVM	V/V	AVX2	Shuffle bytes in <i>ymm2</i> according to contents of <i>ymm3/m256</i> .
EVEX.NDS.128.66.0F38.WIG 00 /r VPSHUFB <i>xmm1</i> {k1}{z}, <i>xmm2</i> , <i>xmm3/m128</i>	FVM	V/V	AVX512VL AVX512BW	Shuffle bytes in <i>xmm2</i> according to contents of <i>xmm3/m128</i> under write mask k1.
EVEX.NDS.256.66.0F38.WIG 00 /r VPSHUFB <i>ymm1</i> {k1}{z}, <i>ymm2</i> , <i>ymm3/m256</i>	FVM	V/V	AVX512VL AVX512BW	Shuffle bytes in <i>ymm2</i> according to contents of <i>ymm3/m256</i> under write mask k1.
EVEX.NDS.512.66.0F38.WIG 00 /r VPSHUFB <i>zmm1</i> {k1}{z}, <i>zmm2</i> , <i>zmm3/m512</i>	FVM	V/V	AVX512BW	Shuffle bytes in <i>zmm2</i> according to contents of <i>zmm3/m512</i> under write mask k1.

### NOTES:

1. See note in Section 2.4, “AVX and SSE Instruction Exception Specification” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FVM	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

PSHUFB performs in-place shuffles of bytes in the destination operand (the first operand) according to the shuffle control mask in the source operand (the second operand). The instruction permutes the data in the destination operand, leaving the shuffle mask unaffected. If the most significant bit (bit[7]) of each byte of the shuffle control mask is set, then constant zero is written in the result byte. Each byte in the shuffle control mask forms an index to permute the corresponding byte in the destination operand. The value of each index is the least significant 4 bits (128-bit operation) or 3 bits (64-bit operation) of the shuffle control byte. When the source operand is a 128-bit memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

In 64-bit mode and not encoded with VEX/EVEX, use the REX prefix to access XMM8-XMM15 registers.

Legacy SSE version 64-bit operand: Both operands can be MMX registers.

128-bit Legacy SSE version: The first source operand and the destination operand are the same. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The destination operand is the first operand, the first source operand is the second operand, the second source operand is the third operand. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: Bits (255:128) of the destination YMM register stores the 16-byte shuffle result of the upper 16 bytes of the first source operand, using the upper 16-bytes of the second source operand as control mask. The value of each index is for the high 128-bit lane is the least significant 4 bits of the respective shuffle control byte. The index value selects a source data element within each 128-bit lane.

EVEX encoded version: The second source operand is an ZMM/YMM/XMM register or an 512/256/128-bit memory location. The first source operand and destination operands are ZMM/YMM/XMM registers. The destination is conditionally updated with writemask k1.

EVEX and VEX encoded version: Four/two in-lane 128-bit shuffles.

## Operation

### PSHUFB (with 64 bit operands)

TEMP ← DEST

for i = 0 to 7 {

if (SRC[(i \* 8)+7] = 1 ) then

DEST[(i\*8)+7...(i\*8)+0] ← 0;

else

index[2..0] ← SRC[(i\*8)+2 .. (i\*8)+0];

DEST[(i\*8)+7...(i\*8)+0] ← TEMP[(index\*8+7)..(index\*8+0)];

endif;

}

### PSHUFB (with 128 bit operands)

TEMP ← DEST

for i = 0 to 15 {

if (SRC[(i \* 8)+7] = 1 ) then

DEST[(i\*8)+7...(i\*8)+0] ← 0;

else

index[3..0] ← SRC[(i\*8)+3 .. (i\*8)+0];

DEST[(i\*8)+7...(i\*8)+0] ← TEMP[(index\*8+7)..(index\*8+0)];

endif

}

### VPSHUFB (VEX.128 encoded version)

for i = 0 to 15 {

if (SRC2[(i \* 8)+7] = 1 ) then

DEST[(i\*8)+7...(i\*8)+0] ← 0;

else

index[3..0] ← SRC2[(i\*8)+3 .. (i\*8)+0];

DEST[(i\*8)+7...(i\*8)+0] ← SRC1[(index\*8+7)..(index\*8+0)];

endif

}

DEST[VLMAX-1:128] ← 0

### VPSHUFB (VEX.256 encoded version)

for i = 0 to 15 {

if (SRC2[(i \* 8)+7] == 1 ) then

DEST[(i\*8)+7...(i\*8)+0] ← 0;

else

index[3..0] ← SRC2[(i\*8)+3 .. (i\*8)+0];

DEST[(i\*8)+7...(i\*8)+0] ← SRC1[(index\*8+7)..(index\*8+0)];

endif

if (SRC2[128 + (i \* 8)+7] == 1 ) then

DEST[128 + (i\*8)+7...(i\*8)+0] ← 0;

else

```

    index[3..0] ← SRC2[128 + (i*8)+3 .. (i*8)+0];
    DEST[128 + (i*8)+7 .. (i*8)+0] ← SRC1[128 + (index*8+7)..(index*8+0)];
  endif
}
VPSHUFB (EVEX encoded versions)
(KL, VL) = (16, 128), (32, 256), (64, 512)
jmask ← (KL-1) & ~0xF // 0x00, 0x10, 0x30 depending on the VL
FOR j = 0 TO KL-1 // dest
  IF kl[ i ] or no_masking
    index ← src.byte[ j ];
    IF index & 0x80
      Dest.byte[ j ] ← 0;
    ELSE
      index ← (index & 0xF) + (j & jmask); // 16-element in-lane lookup
      Dest.byte[ j ] ← src.byte[ index ];
    ELSE if zeroing
      Dest.byte[ j ] ← 0;
    DEST[MAX_VL-1:VL] ← 0;

```

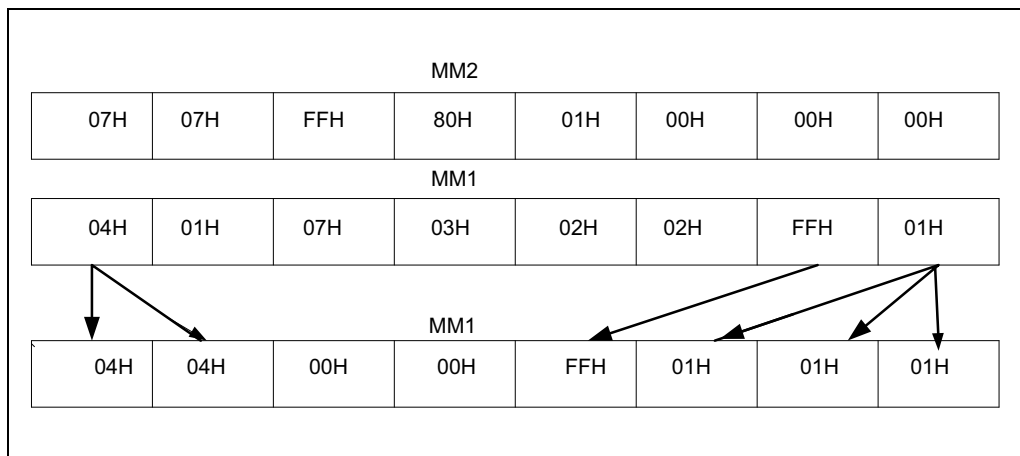


Figure 4-11. PSHUFB with 64-Bit Operands

### Intel C/C++ Compiler Intrinsic Equivalent

```

VPSHUFB __m512i_mm512_shuffle_epi8(__m512i a, __m512i b);
VPSHUFB __m512i_mm512_mask_shuffle_epi8(__m512i s, __mmask64 k, __m512i a, __m512i b);
VPSHUFB __m512i_mm512_maskz_shuffle_epi8(__mmask64 k, __m512i a, __m512i b);
VPSHUFB __m256i_mm256_mask_shuffle_epi8(__m256i s, __mmask32 k, __m256i a, __m256i b);
VPSHUFB __m256i_mm256_maskz_shuffle_epi8(__mmask32 k, __m256i a, __m256i b);
VPSHUFB __m128i_mm_mask_shuffle_epi8(__m128i s, __mmask16 k, __m128i a, __m128i b);
VPSHUFB __m128i_mm_maskz_shuffle_epi8(__mmask16 k, __m128i a, __m128i b);
PSHUFB: __m64_mm_shuffle_pi8(__m64 a, __m64 b)
(V)PSHUFB: __m128i_mm_shuffle_epi8(__m128i a, __m128i b)
VPSHUFB: __m256i_mm256_shuffle_epi8(__m256i a, __m256i b)

```

### SIMD Floating-Point Exceptions

None.





### Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4NF.nb.

## PSHUFD—Shuffle Packed Doublewords

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 70 /r ib PSHUFD <i>xmm1</i> , <i>xmm2/m128</i> , <i>imm8</i>	RMI	V/V	SSE2	Shuffle the doublewords in <i>xmm2/m128</i> based on the encoding in <i>imm8</i> and store the result in <i>xmm1</i> .
VEX.128.66.0F.WIG 70 /r ib VPSHUFD <i>xmm1</i> , <i>xmm2/m128</i> , <i>imm8</i>	RMI	V/V	AVX	Shuffle the doublewords in <i>xmm2/m128</i> based on the encoding in <i>imm8</i> and store the result in <i>xmm1</i> .
VEX.256.66.0F.WIG 70 /r ib VPSHUFD <i>ymm1</i> , <i>ymm2/m256</i> , <i>imm8</i>	RMI	V/V	AVX2	Shuffle the doublewords in <i>ymm2/m256</i> based on the encoding in <i>imm8</i> and store the result in <i>ymm1</i> .
EVEX.128.66.0F.W0 70 /r ib VPSHUFD <i>xmm1</i> { <i>k1</i> }{ <i>z</i> }, <i>xmm2/m128/m32bcst</i> , <i>imm8</i>	FV	V/V	AVX512VL AVX512F	Shuffle the doublewords in <i>xmm2/m128/m32bcst</i> based on the encoding in <i>imm8</i> and store the result in <i>xmm1</i> using writemask <i>k1</i> .
EVEX.256.66.0F.W0 70 /r ib VPSHUFD <i>ymm1</i> { <i>k1</i> }{ <i>z</i> }, <i>ymm2/m256/m32bcst</i> , <i>imm8</i>	FV	V/V	AVX512VL AVX512F	Shuffle the doublewords in <i>ymm2/m256/m32bcst</i> based on the encoding in <i>imm8</i> and store the result in <i>ymm1</i> using writemask <i>k1</i> .
EVEX.512.66.0F.W0 70 /r ib VPSHUFD <i>zmm1</i> { <i>k1</i> }{ <i>z</i> }, <i>zmm2/m512/m32bcst</i> , <i>imm8</i>	FV	V/V	AVX512F	Shuffle the doublewords in <i>zmm2/m512/m32bcst</i> based on the encoding in <i>imm8</i> and store the result in <i>zmm1</i> using writemask <i>k1</i> .

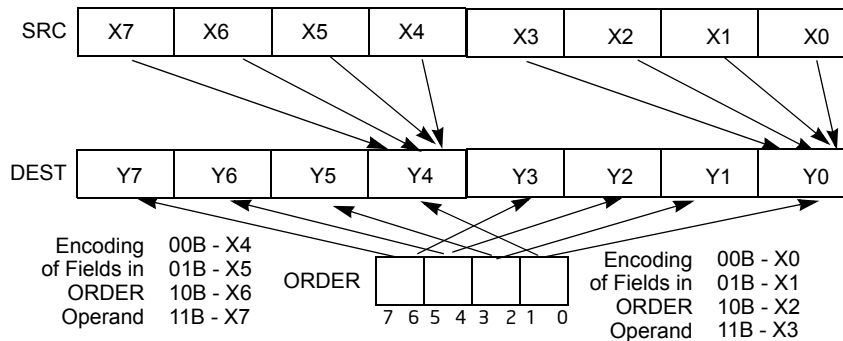
### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (w)	ModRM:r/m (r)	imm8	NA
FV	ModRM:reg (w)	ModRM:r/m (r)	Imm8	NA

### Description

Copies doublewords from source operand (second operand) and inserts them in the destination operand (first operand) at the locations selected with the order operand (third operand). Figure 4-12 shows the operation of the 256-bit VPSHUFD instruction and the encoding of the order operand. Each 2-bit field in the order operand selects the contents of one doubleword location within a 128-bit lane and copy to the target element in the destination operand. For example, bits 0 and 1 of the order operand targets the first doubleword element in the low and high 128-bit lane of the destination operand for 256-bit VPSHUFD. The encoded value of bits 1:0 of the order operand (see the field encoding in Figure 4-12) determines which doubleword element (from the respective 128-bit lane) of the source operand will be copied to doubleword 0 of the destination operand.

For 128-bit operation, only the low 128-bit lane are operative. The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. The order operand is an 8-bit immediate. Note that this instruction permits a doubleword in the source operand to be copied to more than one doubleword location in the destination operand.



**Figure 4-12. 256-bit VPSHUFD Instruction Operation**

The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. The order operand is an 8-bit immediate. Note that this instruction permits a doubleword in the source operand to be copied to more than one doubleword location in the destination operand.

In 64-bit mode and not encoded in VEX/EVEX, using REX.R permits this instruction to access XMM8-XMM15.

128-bit Legacy SSE version: Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. Bits (MAX\_VL-1:128) of the corresponding ZMM register are zeroed.

VEX.256 encoded version: The source operand can be an YMM register or a 256-bit memory location. The destination operand is an YMM register. Bits (MAX\_VL-1:256) of the corresponding ZMM register are zeroed. Bits (255-1:128) of the destination stores the shuffled results of the upper 16 bytes of the source operand using the immediate byte as the order operand.

EVEX encoded version: The source operand can be an ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register updated according to the writemask.

Each 128-bit lane of the destination stores the shuffled results of the respective lane of the source operand using the immediate byte as the order operand.

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

## Operation

### PSHUFD (128-bit Legacy SSE version)

```
DEST[31:0] ← (SRC >> (ORDER[1:0] * 32))[31:0];
DEST[63:32] ← (SRC >> (ORDER[3:2] * 32))[31:0];
DEST[95:64] ← (SRC >> (ORDER[5:4] * 32))[31:0];
DEST[127:96] ← (SRC >> (ORDER[7:6] * 32))[31:0];
DEST[VLMAX-1:128] (Unmodified)
```

### VPSHUFD (VEX.128 encoded version)

```
DEST[31:0] ← (SRC >> (ORDER[1:0] * 32))[31:0];
DEST[63:32] ← (SRC >> (ORDER[3:2] * 32))[31:0];
DEST[95:64] ← (SRC >> (ORDER[5:4] * 32))[31:0];
DEST[127:96] ← (SRC >> (ORDER[7:6] * 32))[31:0];
DEST[VLMAX-1:128] ← 0
```

**VPSHUFD (VEX.256 encoded version)**

```

DEST[31:0] ← (SRC[127:0] >> (ORDER[1:0] * 32))[31:0];
DEST[63:32] ← (SRC[127:0] >> (ORDER[3:2] * 32))[31:0];
DEST[95:64] ← (SRC[127:0] >> (ORDER[5:4] * 32))[31:0];
DEST[127:96] ← (SRC[127:0] >> (ORDER[7:6] * 32))[31:0];
DEST[159:128] ← (SRC[255:128] >> (ORDER[1:0] * 32))[31:0];
DEST[191:160] ← (SRC[255:128] >> (ORDER[3:2] * 32))[31:0];
DEST[223:192] ← (SRC[255:128] >> (ORDER[5:4] * 32))[31:0];
DEST[255:224] ← (SRC[255:128] >> (ORDER[7:6] * 32))[31:0];
DEST[VLMAX-1:256] ← 0

```

**VPSHUFD (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 32
  IF (EVEX.b = 1) AND (SRC *is memory*)
    THEN TMP_SRC[i+31:i] ← SRC[31:0]
    ELSE TMP_SRC[i+31:i] ← SRC[i+31:i]
  FI;
ENDFOR;
IF VL >= 128
  TMP_DEST[31:0] ← (TMP_SRC[127:0] >> (ORDER[1:0] * 32))[31:0];
  TMP_DEST[63:32] ← (TMP_SRC[127:0] >> (ORDER[3:2] * 32))[31:0];
  TMP_DEST[95:64] ← (TMP_SRC[127:0] >> (ORDER[5:4] * 32))[31:0];
  TMP_DEST[127:96] ← (TMP_SRC[127:0] >> (ORDER[7:6] * 32))[31:0];
FI;
IF VL >= 256
  TMP_DEST[159:128] ← (TMP_SRC[255:128] >> (ORDER[1:0] * 32))[31:0];
  TMP_DEST[191:160] ← (TMP_SRC[255:128] >> (ORDER[3:2] * 32))[31:0];
  TMP_DEST[223:192] ← (TMP_SRC[255:128] >> (ORDER[5:4] * 32))[31:0];
  TMP_DEST[255:224] ← (TMP_SRC[255:128] >> (ORDER[7:6] * 32))[31:0];
FI;
IF VL >= 512
  TMP_DEST[287:256] ← (TMP_SRC[383:256] >> (ORDER[1:0] * 32))[31:0];
  TMP_DEST[319:288] ← (TMP_SRC[383:256] >> (ORDER[3:2] * 32))[31:0];
  TMP_DEST[351:320] ← (TMP_SRC[383:256] >> (ORDER[5:4] * 32))[31:0];
  TMP_DEST[383:352] ← (TMP_SRC[383:256] >> (ORDER[7:6] * 32))[31:0];
  TMP_DEST[415:384] ← (TMP_SRC[511:384] >> (ORDER[1:0] * 32))[31:0];
  TMP_DEST[447:416] ← (TMP_SRC[511:384] >> (ORDER[3:2] * 32))[31:0];
  TMP_DEST[479:448] ← (TMP_SRC[511:384] >> (ORDER[5:4] * 32))[31:0];
  TMP_DEST[511:480] ← (TMP_SRC[511:384] >> (ORDER[7:6] * 32))[31:0];
FI;
FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+31:i] remains unchanged*
      ELSE *zeroing-masking* ; zeroing-masking
        DEST[i+31:i] ← 0
      FI
    FI;
  FI;
ENDFOR

```

DEST[MAX\_VL-1:VL] ← 0

### Intel C/C++ Compiler Intrinsic Equivalent

VPSHUFD \_\_m512i \_mm512\_shuffle\_epi32(\_\_m512i a, int n);  
 VPSHUFD \_\_m512i \_mm512\_mask\_shuffle\_epi32(\_\_m512i s, \_\_mmask16 k, \_\_m512i a, int n);  
 VPSHUFD \_\_m512i \_mm512\_maskz\_shuffle\_epi32(\_\_mmask16 k, \_\_m512i a, int n);  
 VPSHUFD \_\_m256i \_mm256\_mask\_shuffle\_epi32(\_\_m256i s, \_\_mmask8 k, \_\_m256i a, int n);  
 VPSHUFD \_\_m256i \_mm256\_maskz\_shuffle\_epi32(\_\_mmask8 k, \_\_m256i a, int n);  
 VPSHUFD \_\_m128i \_mm\_mask\_shuffle\_epi32(\_\_m128i s, \_\_mmask8 k, \_\_m128i a, int n);  
 VPSHUFD \_\_m128i \_mm\_maskz\_shuffle\_epi32(\_\_mmask8 k, \_\_m128i a, int n);  
 (V)PSHUFD: \_\_m128i \_mm\_shuffle\_epi32(\_\_m128i a, int n)  
 VPSHUFD: \_\_m256i \_mm256\_shuffle\_epi32(\_\_m256i a, const int n)

### Flags Affected

None.

### SIMD Floating-Point Exceptions

None.

### Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4NF.

#UD If VEX.vvvv ≠ 1111B or EVEX.vvvv ≠ 1111B.

## PSHUFHW—Shuffle Packed High Words

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 70 /r ib PSHUFHW <i>xmm1</i> , <i>xmm2/m128</i> , <i>imm8</i>	RMI	V/V	SSE2	Shuffle the high words in <i>xmm2/m128</i> based on the encoding in <i>imm8</i> and store the result in <i>xmm1</i> .
VEX.128.F3.0F.WIG 70 /r ib VPSHUFHW <i>xmm1</i> , <i>xmm2/m128</i> , <i>imm8</i>	RMI	V/V	AVX	Shuffle the high words in <i>xmm2/m128</i> based on the encoding in <i>imm8</i> and store the result in <i>xmm1</i> .
VEX.256.F3.0F.WIG 70 /r ib VPSHUFHW <i>ymm1</i> , <i>ymm2/m256</i> , <i>imm8</i>	RMI	V/V	AVX2	Shuffle the high words in <i>ymm2/m256</i> based on the encoding in <i>imm8</i> and store the result in <i>ymm1</i> .
EVEX.128.F3.0F.WIG 70 /r ib VPSHUFHW <i>xmm1</i> { <i>k1</i> }{ <i>z</i> }, <i>xmm2/m128</i> , <i>imm8</i>	FVM	V/V	AVX512VL AVX512BW	Shuffle the high words in <i>xmm2/m128</i> based on the encoding in <i>imm8</i> and store the result in <i>xmm1</i> under write mask <i>k1</i> .
EVEX.256.F3.0F.WIG 70 /r ib VPSHUFHW <i>ymm1</i> { <i>k1</i> }{ <i>z</i> }, <i>ymm2/m256</i> , <i>imm8</i>	FVM	V/V	AVX512VL AVX512BW	Shuffle the high words in <i>ymm2/m256</i> based on the encoding in <i>imm8</i> and store the result in <i>ymm1</i> under write mask <i>k1</i> .
EVEX.512.F3.0F.WIG 70 /r ib VPSHUFHW <i>zmm1</i> { <i>k1</i> }{ <i>z</i> }, <i>zmm2/m512</i> , <i>imm8</i>	FVM	V/V	AVX512BW	Shuffle the high words in <i>zmm2/m512</i> based on the encoding in <i>imm8</i> and store the result in <i>zmm1</i> under write mask <i>k1</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg ( <i>w</i> )	ModRM:r/m ( <i>r</i> )	<i>imm8</i>	NA
FVM	ModRM:reg ( <i>w</i> )	ModRM:r/m ( <i>r</i> )	<i>Imm8</i>	NA

### Description

Copies words from the high quadword of a 128-bit lane of the source operand and inserts them in the high quadword of the destination operand at word locations (of the respective lane) selected with the immediate operand. This 256-bit operation is similar to the in-lane operation used by the 256-bit VPSHUFD instruction, which is illustrated in Figure 4-12. For 128-bit operation, only the low 128-bit lane is operative. Each 2-bit field in the immediate operand selects the contents of one word location in the high quadword of the destination operand. The binary encodings of the immediate operand fields select words (0, 1, 2 or 3, 4) from the high quadword of the source operand to be copied to the destination operand. The low quadword of the source operand is copied to the low quadword of the destination operand, for each 128-bit lane.

Note that this instruction permits a word in the high quadword of the source operand to be copied to more than one word location in the high quadword of the destination operand.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The destination operand is an XMM register. The source operand can be an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The destination operand is an XMM register. The source operand can be an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the destination YMM register are zeroed. VEX.vvvv is reserved and must be 1111b, VEX.L must be 0, otherwise the instruction will #UD.

VEX.256 encoded version: The destination operand is an YMM register. The source operand can be an YMM register or a 256-bit memory location.

EVEX encoded version: The destination operand is a ZMM/YMM/XMM registers. The source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination is updated according to the writemask.

Note: In VEX encoded versions, VEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

## Operation

### PSHUFHW (128-bit Legacy SSE version)

```
DEST[63:0] ← SRC[63:0]
DEST[79:64] ← (SRC >> (imm[1:0] * 16))[79:64]
DEST[95:80] ← (SRC >> (imm[3:2] * 16))[79:64]
DEST[111:96] ← (SRC >> (imm[5:4] * 16))[79:64]
DEST[127:112] ← (SRC >> (imm[7:6] * 16))[79:64]
DEST[VLMAX-1:128] (Unmodified)
```

### VPSHUFHW (VEX.128 encoded version)

```
DEST[63:0] ← SRC1[63:0]
DEST[79:64] ← (SRC1 >> (imm[1:0] * 16))[79:64]
DEST[95:80] ← (SRC1 >> (imm[3:2] * 16))[79:64]
DEST[111:96] ← (SRC1 >> (imm[5:4] * 16))[79:64]
DEST[127:112] ← (SRC1 >> (imm[7:6] * 16))[79:64]
DEST[VLMAX-1:128] ← 0
```

### VPSHUFHW (VEX.256 encoded version)

```
DEST[63:0] ← SRC1[63:0]
DEST[79:64] ← (SRC1 >> (imm[1:0] * 16))[79:64]
DEST[95:80] ← (SRC1 >> (imm[3:2] * 16))[79:64]
DEST[111:96] ← (SRC1 >> (imm[5:4] * 16))[79:64]
DEST[127:112] ← (SRC1 >> (imm[7:6] * 16))[79:64]
DEST[191:128] ← SRC1[191:128]
DEST[207:192] ← (SRC1 >> (imm[1:0] * 16))[207:192]
DEST[223:208] ← (SRC1 >> (imm[3:2] * 16))[207:192]
DEST[239:224] ← (SRC1 >> (imm[5:4] * 16))[207:192]
DEST[255:240] ← (SRC1 >> (imm[7:6] * 16))[207:192]
DEST[VLMAX-1:256] ← 0
```

### VPSHUFHW (EVEX encoded versions)

(KL, VL) = (8, 128), (16, 256), (32, 512)

IF VL >= 128

```
TMP_DEST[63:0] ← SRC1[63:0]
TMP_DEST[79:64] ← (SRC1 >> (imm[1:0] * 16))[79:64]
TMP_DEST[95:80] ← (SRC1 >> (imm[3:2] * 16))[79:64]
TMP_DEST[111:96] ← (SRC1 >> (imm[5:4] * 16))[79:64]
TMP_DEST[127:112] ← (SRC1 >> (imm[7:6] * 16))[79:64]
```

FI;

IF VL >= 256

```
TMP_DEST[191:128] ← SRC1[191:128]
TMP_DEST[207:192] ← (SRC1 >> (imm[1:0] * 16))[207:192]
TMP_DEST[223:208] ← (SRC1 >> (imm[3:2] * 16))[207:192]
TMP_DEST[239:224] ← (SRC1 >> (imm[5:4] * 16))[207:192]
TMP_DEST[255:240] ← (SRC1 >> (imm[7:6] * 16))[207:192]
```

FI;

IF VL >= 512

```
TMP_DEST[319:256] ← SRC1[319:256]
TMP_DEST[335:320] ← (SRC1 >> (imm[1:0] * 16))[335:320]
```

```

TMP_DEST[351:336] ← (SRC1 >> (imm[3:2] * 16))[335:320]
TMP_DEST[367:352] ← (SRC1 >> (imm[5:4] * 16))[335:320]
TMP_DEST[383:368] ← (SRC1 >> (imm[7:6] * 16))[335:320]
TMP_DEST[447:384] ← SRC1[447:384]
TMP_DEST[463:448] ← (SRC1 >> (imm[1:0] * 16))[463:448]
TMP_DEST[479:464] ← (SRC1 >> (imm[3:2] * 16))[463:448]
TMP_DEST[495:480] ← (SRC1 >> (imm[5:4] * 16))[463:448]
TMP_DEST[511:496] ← (SRC1 >> (imm[7:6] * 16))[463:448]
FI;

FOR j ← 0 TO KL-1
  i ← j * 16
  IF k1[j] OR *no writemask*
    THEN DEST[i+15:i] ← TMP_DEST[i+15:i];
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+15:i] remains unchanged*
    ELSE *zeroing-masking* ; zeroing-masking
      DEST[i+15:i] ← 0
    FI
  FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

### Intel C/C++ Compiler Intrinsic Equivalent

```

VPSHUFWH __m512i _mm512_shufflehi_epi16(__m512i a, int n);
VPSHUFWH __m512i _mm512_mask_shufflehi_epi16(__m512i s, __mmask16 k, __m512i a, int n);
VPSHUFWH __m512i _mm512_maskz_shufflehi_epi16(__mmask16 k, __m512i a, int n);
VPSHUFWH __m256i _mm256_mask_shufflehi_epi16(__m256i s, __mmask8 k, __m256i a, int n);
VPSHUFWH __m256i _mm256_maskz_shufflehi_epi16(__mmask8 k, __m256i a, int n);
VPSHUFWH __m128i _mm_mask_shufflehi_epi16(__m128i s, __mmask8 k, __m128i a, int n);
VPSHUFWH __m128i _mm_maskz_shufflehi_epi16(__mmask8 k, __m128i a, int n);
(V)PSHUFWH: __m128i _mm_shufflehi_epi16(__m128i a, int n)
VPSHUFWH: __m256i _mm256_shufflehi_epi16(__m256i a, const int n)

```

### Flags Affected

None.

### SIMD Floating-Point Exceptions

None.

### Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4;

EVEX-encoded instruction, see Exceptions Type E4NF.nb

#UD If VEX.vvvv != 1111B, or EVEX.vvvv != 1111B.



## PSHUFLW—Shuffle Packed Low Words

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 70 /r ib PSHUFLW <i>xmm1</i> , <i>xmm2/m128</i> , <i>imm8</i>	RMI	V/V	SSE2	Shuffle the low words in <i>xmm2/m128</i> based on the encoding in <i>imm8</i> and store the result in <i>xmm1</i> .
VEX.128.F2.0F.WIG 70 /r ib VPSHUFLW <i>xmm1</i> , <i>xmm2/m128</i> , <i>imm8</i>	RMI	V/V	AVX	Shuffle the low words in <i>xmm2/m128</i> based on the encoding in <i>imm8</i> and store the result in <i>xmm1</i> .
VEX.256.F2.0F.WIG 70 /r ib VPSHUFLW <i>ymm1</i> , <i>ymm2/m256</i> , <i>imm8</i>	RMI	V/V	AVX2	Shuffle the low words in <i>ymm2/m256</i> based on the encoding in <i>imm8</i> and store the result in <i>ymm1</i> .
EVEX.128.F2.0F.WIG 70 /r ib VPSHUFLW <i>xmm1</i> { <i>k1</i> }{ <i>z</i> }, <i>xmm2/m128</i> , <i>imm8</i>	FVM	V/V	AVX512VL AVX512BW	Shuffle the low words in <i>xmm2/m128</i> based on the encoding in <i>imm8</i> and store the result in <i>xmm1</i> under write mask <i>k1</i> .
EVEX.256.F2.0F.WIG 70 /r ib VPSHUFLW <i>ymm1</i> { <i>k1</i> }{ <i>z</i> }, <i>ymm2/m256</i> , <i>imm8</i>	FVM	V/V	AVX512VL AVX512BW	Shuffle the low words in <i>ymm2/m256</i> based on the encoding in <i>imm8</i> and store the result in <i>ymm1</i> under write mask <i>k1</i> .
EVEX.512.F2.0F.WIG 70 /r ib VPSHUFLW <i>zmm1</i> { <i>k1</i> }{ <i>z</i> }, <i>zmm2/m512</i> , <i>imm8</i>	FVM	V/V	AVX512BW	Shuffle the low words in <i>zmm2/m512</i> based on the encoding in <i>imm8</i> and store the result in <i>zmm1</i> under write mask <i>k1</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg ( <i>w</i> )	ModRM:r/m ( <i>r</i> )	<i>imm8</i>	NA
FVM	ModRM:reg ( <i>w</i> )	ModRM:r/m ( <i>r</i> )	<i>Imm8</i>	NA

### Description

Copies words from the low quadword of a 128-bit lane of the source operand and inserts them in the low quadword of the destination operand at word locations (of the respective lane) selected with the immediate operand. The 256-bit operation is similar to the in-lane operation used by the 256-bit VPSHUFD instruction, which is illustrated in Figure 4-12. For 128-bit operation, only the low 128-bit lane is operative. Each 2-bit field in the immediate operand selects the contents of one word location in the low quadword of the destination operand. The binary encodings of the immediate operand fields select words (0, 1, 2 or 3) from the low quadword of the source operand to be copied to the destination operand. The high quadword of the source operand is copied to the high quadword of the destination operand, for each 128-bit lane.

Note that this instruction permits a word in the low quadword of the source operand to be copied to more than one word location in the low quadword of the destination operand.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The destination operand is an XMM register. The source operand can be an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The destination operand is an XMM register. The source operand can be an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The destination operand is an YMM register. The source operand can be an YMM register or a 256-bit memory location.

EVEX encoded version: The destination operand is a ZMM/YMM/XMM registers. The source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination is updated according to the writemask.

Note: In VEX encoded versions, VEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

## Operation

### PSHUFLW (128-bit Legacy SSE version)

```
DEST[15:0] ← (SRC >> (imm[1:0] * 16))[15:0]
DEST[31:16] ← (SRC >> (imm[3:2] * 16))[15:0]
DEST[47:32] ← (SRC >> (imm[5:4] * 16))[15:0]
DEST[63:48] ← (SRC >> (imm[7:6] * 16))[15:0]
DEST[127:64] ← SRC[127:64]
DEST[VLMAX-1:128] (Unmodified)
```

### VPSHUFLW (VEX.128 encoded version)

```
DEST[15:0] ← (SRC1 >> (imm[1:0] * 16))[15:0]
DEST[31:16] ← (SRC1 >> (imm[3:2] * 16))[15:0]
DEST[47:32] ← (SRC1 >> (imm[5:4] * 16))[15:0]
DEST[63:48] ← (SRC1 >> (imm[7:6] * 16))[15:0]
DEST[127:64] ← SRC[127:64]
DEST[VLMAX-1:128] ← 0
```

### VPSHUFLW (VEX.256 encoded version)

```
DEST[15:0] ← (SRC1 >> (imm[1:0] * 16))[15:0]
DEST[31:16] ← (SRC1 >> (imm[3:2] * 16))[15:0]
DEST[47:32] ← (SRC1 >> (imm[5:4] * 16))[15:0]
DEST[63:48] ← (SRC1 >> (imm[7:6] * 16))[15:0]
DEST[127:64] ← SRC1[127:64]
DEST[143:128] ← (SRC1 >> (imm[1:0] * 16))[143:128]
DEST[159:144] ← (SRC1 >> (imm[3:2] * 16))[143:128]
DEST[175:160] ← (SRC1 >> (imm[5:4] * 16))[143:128]
DEST[191:176] ← (SRC1 >> (imm[7:6] * 16))[143:128]
DEST[255:192] ← SRC1[255:192]
DEST[VLMAX-1:256] ← 0
```

### VPSHUFLW (EVEX.U1.512 encoded version)

(KL, VL) = (8, 128), (16, 256), (32, 512)

IF VL >= 128

```
TMP_DEST[15:0] ← (SRC1 >> (imm[1:0] * 16))[15:0]
TMP_DEST[31:16] ← (SRC1 >> (imm[3:2] * 16))[15:0]
TMP_DEST[47:32] ← (SRC1 >> (imm[5:4] * 16))[15:0]
TMP_DEST[63:48] ← (SRC1 >> (imm[7:6] * 16))[15:0]
TMP_DEST[127:64] ← SRC1[127:64]
```

FI;

IF VL >= 256

```
TMP_DEST[143:128] ← (SRC1 >> (imm[1:0] * 16))[143:128]
TMP_DEST[159:144] ← (SRC1 >> (imm[3:2] * 16))[143:128]
TMP_DEST[175:160] ← (SRC1 >> (imm[5:4] * 16))[143:128]
TMP_DEST[191:176] ← (SRC1 >> (imm[7:6] * 16))[143:128]
TMP_DEST[255:192] ← SRC1[255:192]
```

FI;

IF VL >= 512

```
TMP_DEST[271:256] ← (SRC1 >> (imm[1:0] * 16))[271:256]
TMP_DEST[287:272] ← (SRC1 >> (imm[3:2] * 16))[271:256]
```

```

TMP_DEST[303:288] ← (SRC1 >> (imm[5:4] * 16))[271:256]
TMP_DEST[319:304] ← (SRC1 >> (imm[7:6] * 16))[271:256]
TMP_DEST[383:320] ← SRC1[383:320]
TMP_DEST[399:384] ← (SRC1 >> (imm[1:0] * 16))[399:384]
TMP_DEST[415:400] ← (SRC1 >> (imm[3:2] * 16))[399:384]
TMP_DEST[431:416] ← (SRC1 >> (imm[5:4] * 16))[399:384]
TMP_DEST[447:432] ← (SRC1 >> (imm[7:6] * 16))[399:384]
TMP_DEST[511:448] ← SRC1[511:448]
FI;

FOR j ← 0 TO KL-1
  i ← j * 16
  IF k1[j] OR *no writemask*
    THEN DEST[i+15:i] ← TMP_DEST[i+15:i];
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+15:i] remains unchanged*
    ELSE *zeroing-masking* ; zeroing-masking
      DEST[i+15:i] ← 0
    FI
  FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

### Intel C/C++ Compiler Intrinsic Equivalent

```

VPSHUFLW __m512i _mm512_shufflelo_epi16(__m512i a, int n);
VPSHUFLW __m512i _mm512_mask_shufflelo_epi16(__m512i s, __mmask16 k, __m512i a, int n);
VPSHUFLW __m512i _mm512_maskz_shufflelo_epi16(__mmask16 k, __m512i a, int n);
VPSHUFLW __m256i _mm256_mask_shufflelo_epi16(__m256i s, __mmask8 k, __m256i a, int n);
VPSHUFLW __m256i _mm256_maskz_shufflelo_epi16(__mmask8 k, __m256i a, int n);
VPSHUFLW __m128i _mm_mask_shufflelo_epi16(__m128i s, __mmask8 k, __m128i a, int n);
VPSHUFLW __m128i _mm_maskz_shufflelo_epi16(__mmask8 k, __m128i a, int n);
(V)PSHUFLW: __m128i _mm_shufflelo_epi16(__m128i a, int n)
VPSHUFLW: __m256i _mm256_shufflelo_epi16(__m256i a, const int n)

```

### Flags Affected

None.

### SIMD Floating-Point Exceptions

None.

### Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4;

EVEX-encoded instruction, see Exceptions Type E4NF.nb

#UD If VEX.vvvv != 1111B, or EVEX.vvvv != 1111B.

## PSLLDQ—Shift Double Quadword Left Logical

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 73 /7 ib PSLLDQ <i>xmm1</i> , <i>imm8</i>	MI	V/V	SSE2	Shift <i>xmm1</i> left by <i>imm8</i> bytes while shifting in 0s.
VEX.NDD.128.66.0F.WIG 73 /7 ib VPSLLDQ <i>xmm1</i> , <i>xmm2</i> , <i>imm8</i>	VMI	V/V	AVX	Shift <i>xmm2</i> left by <i>imm8</i> bytes while shifting in 0s and store result in <i>xmm1</i> .
VEX.NDD.256.66.0F.WIG 73 /7 ib VPSLLDQ <i>ymm1</i> , <i>ymm2</i> , <i>imm8</i>	VMI	V/V	AVX2	Shift <i>ymm2</i> left by <i>imm8</i> bytes while shifting in 0s and store result in <i>ymm1</i> .
EVEX.NDD.128.66.0F.WIG 73 /7 ib VPSLLDQ <i>xmm1</i> , <i>xmm2</i> / <i>m128</i> , <i>imm8</i>	FVMI	V/V	AVX512VL AVX512BW	Shift <i>xmm2</i> / <i>m128</i> left by <i>imm8</i> bytes while shifting in 0s and store result in <i>xmm1</i> .
EVEX.NDD.256.66.0F.WIG 73 /7 ib VPSLLDQ <i>ymm1</i> , <i>ymm2</i> / <i>m256</i> , <i>imm8</i>	FVMI	V/V	AVX512VL AVX512BW	Shift <i>ymm2</i> / <i>m256</i> left by <i>imm8</i> bytes while shifting in 0s and store result in <i>ymm1</i> .
EVEX.NDD.512.66.0F.WIG 73 /7 ib VPSLLDQ <i>zmm1</i> , <i>zmm2</i> / <i>m512</i> , <i>imm8</i>	FVMI	V/V	AVX512BW	Shift <i>zmm2</i> / <i>m512</i> left by <i>imm8</i> bytes while shifting in 0s and store result in <i>zmm1</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MI	ModRM:r/m (r, w)	<i>imm8</i>	NA	NA
VMI	VEX.vvvv (w)	ModRM:r/m (r)	<i>imm8</i>	NA
FVMI	EVEX.vvvv (w)	ModRM:r/m (R)	<i>Imm8</i>	NA

### Description

Shifts the destination operand (first operand) to the left by the number of bytes specified in the count operand (second operand). The empty low-order bytes are cleared (set to all 0s). If the value specified by the count operand is greater than 15, the destination operand is set to all 0s. The count operand is an 8-bit immediate.

128-bit Legacy SSE version: The source and destination operands are the same. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The source and destination operands are XMM registers. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The source operand is YMM register. The destination operand is an YMM register. Bits (MAX\_VL-1:256) of the corresponding ZMM register are zeroed. The count operand applies to both the low and high 128-bit lanes.

EVEX encoded versions: 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. The count operand applies to each 128-bit lanes.

### Operation

#### VPSLLDQ (EVEX.U1.512 encoded version)

TEMP ← COUNT

IF (TEMP > 15) THEN TEMP ← 16; FI

DEST[127:0] ← SRC[127:0] << (TEMP \* 8)

DEST[255:128] ← SRC[255:128] << (TEMP \* 8)

DEST[383:256] ← SRC[383:256] << (TEMP \* 8)

DEST[511:384] ← SRC[511:384] << (TEMP \* 8)

DEST[MAX\_VL-1:512] ← 0

**VPSLLDQ (VEX.256 and EVEX.256 encoded version)**TEMP  $\leftarrow$  COUNTIF (TEMP > 15) THEN TEMP  $\leftarrow$  16; FIDEST[127:0]  $\leftarrow$  SRC[127:0]  $\ll$  (TEMP \* 8)DEST[255:128]  $\leftarrow$  SRC[255:128]  $\ll$  (TEMP \* 8)DEST[MAX\_VL-1:256]  $\leftarrow$  0**VPSLLDQ (VEX.128 and EVEX.128 encoded version)**TEMP  $\leftarrow$  COUNTIF (TEMP > 15) THEN TEMP  $\leftarrow$  16; FIDEST  $\leftarrow$  SRC  $\ll$  (TEMP \* 8)DEST[MAX\_VL-1:128]  $\leftarrow$  0**PSLLDQ(128-bit Legacy SSE version)**TEMP  $\leftarrow$  COUNTIF (TEMP > 15) THEN TEMP  $\leftarrow$  16; FIDEST  $\leftarrow$  DEST  $\ll$  (TEMP \* 8)

DEST[VLMAX-1:128] (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**(V)PSLLDQ: `__m128i _mm_slli_si128 ( __m128i a, int imm)`VPSLLDQ: `__m256i _mm256_slli_si256 ( __m256i a, const int imm)`VPSLLDQ: `__m512i _mm512_bslli_epi128 ( __m512i a, const int imm)`**Flags Affected**

None.

**Numeric Exceptions**

None.

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 7.

EVEX-encoded instruction, see Exceptions Type E4NF.nb.

## PSLLW/PSLLD/PSLLQ—Shift Packed Data Left Logical

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F F1 /r <sup>1</sup> PSLLW <i>mm</i> , <i>mm/m64</i>	RM	V/V	MMX	Shift words in <i>mm</i> left <i>mm/m64</i> while shifting in 0s.
66 0F F1 /r PSLLW <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE2	Shift words in <i>xmm1</i> left by <i>xmm2/m128</i> while shifting in 0s.
0F 71 /6 ib PSLLW <i>mm1</i> , <i>imm8</i>	MI	V/V	MMX	Shift words in <i>mm</i> left by <i>imm8</i> while shifting in 0s.
66 0F 71 /6 ib PSLLW <i>xmm1</i> , <i>imm8</i>	MI	V/V	SSE2	Shift words in <i>xmm1</i> left by <i>imm8</i> while shifting in 0s.
0F F2 /r <sup>1</sup> PSLLD <i>mm</i> , <i>mm/m64</i>	RM	V/V	MMX	Shift doublewords in <i>mm</i> left by <i>mm/m64</i> while shifting in 0s.
66 0F F2 /r PSLLD <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE2	Shift doublewords in <i>xmm1</i> left by <i>xmm2/m128</i> while shifting in 0s.
0F 72 /6 ib <sup>1</sup> PSLLD <i>mm</i> , <i>imm8</i>	MI	V/V	MMX	Shift doublewords in <i>mm</i> left by <i>imm8</i> while shifting in 0s.
66 0F 72 /6 ib PSLLD <i>xmm1</i> , <i>imm8</i>	MI	V/V	SSE2	Shift doublewords in <i>xmm1</i> left by <i>imm8</i> while shifting in 0s.
0F F3 /r <sup>1</sup> PSLLQ <i>mm</i> , <i>mm/m64</i>	RM	V/V	MMX	Shift quadword in <i>mm</i> left by <i>mm/m64</i> while shifting in 0s.
66 0F F3 /r PSLLQ <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE2	Shift quadwords in <i>xmm1</i> left by <i>xmm2/m128</i> while shifting in 0s.
0F 73 /6 ib <sup>1</sup> PSLLQ <i>mm</i> , <i>imm8</i>	MI	V/V	MMX	Shift quadword in <i>mm</i> left by <i>imm8</i> while shifting in 0s.
66 0F 73 /6 ib PSLLQ <i>xmm1</i> , <i>imm8</i>	MI	V/V	SSE2	Shift quadwords in <i>xmm1</i> left by <i>imm8</i> while shifting in 0s.
VEX.NDS.128.66.0F.WIG F1 /r VPSLLW <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Shift words in <i>xmm2</i> left by amount specified in <i>xmm3/m128</i> while shifting in 0s.
VEX.NDD.128.66.0F.WIG 71 /6 ib VPSLLW <i>xmm1</i> , <i>xmm2</i> , <i>imm8</i>	VMI	V/V	AVX	Shift words in <i>xmm2</i> left by <i>imm8</i> while shifting in 0s.
VEX.NDS.128.66.0F.WIG F2 /r VPSLLD <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Shift doublewords in <i>xmm2</i> left by amount specified in <i>xmm3/m128</i> while shifting in 0s.
VEX.NDD.128.66.0F.WIG 72 /6 ib VPSLLD <i>xmm1</i> , <i>xmm2</i> , <i>imm8</i>	VMI	V/V	AVX	Shift doublewords in <i>xmm2</i> left by <i>imm8</i> while shifting in 0s.
VEX.NDS.128.66.0F.WIG F3 /r VPSLLQ <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Shift quadwords in <i>xmm2</i> left by amount specified in <i>xmm3/m128</i> while shifting in 0s.
VEX.NDD.128.66.0F.WIG 73 /6 ib VPSLLQ <i>xmm1</i> , <i>xmm2</i> , <i>imm8</i>	VMI	V/V	AVX	Shift quadwords in <i>xmm2</i> left by <i>imm8</i> while shifting in 0s.
VEX.NDS.256.66.0F.WIG F1 /r VPSLLW <i>ymm1</i> , <i>ymm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX2	Shift words in <i>ymm2</i> left by amount specified in <i>xmm3/m128</i> while shifting in 0s.
VEX.NDD.256.66.0F.WIG 71 /6 ib VPSLLW <i>ymm1</i> , <i>ymm2</i> , <i>imm8</i>	VMI	V/V	AVX2	Shift words in <i>ymm2</i> left by <i>imm8</i> while shifting in 0s.

VEX.NDS.256.66.0F.WIG F2 /r VPSLLD <i>ymm1, ymm2, xmm3/m128</i>	RVM V/V	AVX2	Shift doublewords in <i>ymm2</i> left by amount specified in <i>xmm3/m128</i> while shifting in 0s.
VEX.NDD.256.66.0F.WIG 72 /6 ib VPSLLD <i>ymm1, ymm2, imm8</i>	VMI V/V	AVX2	Shift doublewords in <i>ymm2</i> left by <i>imm8</i> while shifting in 0s.
VEX.NDS.256.66.0F.WIG F3 /r VPSLLQ <i>ymm1, ymm2, xmm3/m128</i>	RVM V/V	AVX2	Shift quadwords in <i>ymm2</i> left by amount specified in <i>xmm3/m128</i> while shifting in 0s.
VEX.NDD.256.66.0F.WIG 73 /6 ib VPSLLQ <i>ymm1, ymm2, imm8</i>	VMI V/V	AVX2	Shift quadwords in <i>ymm2</i> left by <i>imm8</i> while shifting in 0s.
EVEX.NDS.128.66.0F.WIG F1 /r VPSLLW <i>xmm1 {k1}{z}, xmm2, xmm3/m128</i>	M128 V/V	AVX512VL AVX512BW	Shift words in <i>xmm2</i> left by amount specified in <i>xmm3/m128</i> while shifting in 0s using writemask <i>k1</i> .
EVEX.NDS.256.66.0F.WIG F1 /r VPSLLW <i>ymm1 {k1}{z}, ymm2, xmm3/m128</i>	M128 V/V	AVX512VL AVX512BW	Shift words in <i>ymm2</i> left by amount specified in <i>xmm3/m128</i> while shifting in 0s using writemask <i>k1</i> .
EVEX.NDS.512.66.0F.WIG F1 /r VPSLLW <i>zmm1 {k1}{z}, zmm2, xmm3/m128</i>	M128 V/V	AVX512BW	Shift words in <i>zmm2</i> left by amount specified in <i>xmm3/m128</i> while shifting in 0s using writemask <i>k1</i> .
EVEX.NDD.128.66.0F.WIG 71 /6 ib VPSLLW <i>xmm1 {k1}{z}, xmm2/m128, imm8</i>	FVMI V/V	AVX512VL AVX512BW	Shift words in <i>xmm2/m128</i> left by <i>imm8</i> while shifting in 0s using writemask <i>k1</i> .
EVEX.NDD.256.66.0F.WIG 71 /6 ib VPSLLW <i>ymm1 {k1}{z}, ymm2/m256, imm8</i>	FVMI V/V	AVX512VL AVX512BW	Shift words in <i>ymm2/m256</i> left by <i>imm8</i> while shifting in 0s using writemask <i>k1</i> .
EVEX.NDD.512.66.0F.WIG 71 /6 ib VPSLLW <i>zmm1 {k1}{z}, zmm2/m512, imm8</i>	FVMI V/V	AVX512BW	Shift words in <i>zmm2/m512</i> left by <i>imm8</i> while shifting in 0 using writemask <i>k1</i> .
EVEX.NDS.128.66.0F.W0 F2 /r VPSLLD <i>xmm1 {k1}{z}, xmm2, xmm3/m128</i>	M128 V/V	AVX512VL AVX512F	Shift doublewords in <i>xmm2</i> left by amount specified in <i>xmm3/m128</i> while shifting in 0s under writemask <i>k1</i> .
EVEX.NDS.256.66.0F.W0 F2 /r VPSLLD <i>ymm1 {k1}{z}, ymm2, xmm3/m128</i>	M128 V/V	AVX512VL AVX512F	Shift doublewords in <i>ymm2</i> left by amount specified in <i>xmm3/m128</i> while shifting in 0s under writemask <i>k1</i> .
EVEX.NDS.512.66.0F.W0 F2 /r VPSLLD <i>zmm1 {k1}{z}, zmm2, xmm3/m128</i>	M128 V/V	AVX512F	Shift doublewords in <i>zmm2</i> left by amount specified in <i>xmm3/m128</i> while shifting in 0s under writemask <i>k1</i> .
EVEX.NDD.128.66.0F.W0 72 /6 ib VPSLLD <i>xmm1 {k1}{z}, xmm2/m128/m32bcst, imm8</i>	FVI V/V	AVX512VL AVX512F	Shift doublewords in <i>xmm2/m128/m32bcst</i> left by <i>imm8</i> while shifting in 0s using writemask <i>k1</i> .
EVEX.NDD.256.66.0F.W0 72 /6 ib VPSLLD <i>ymm1 {k1}{z}, ymm2/m256/m32bcst, imm8</i>	FVI V/V	AVX512VL AVX512F	Shift doublewords in <i>ymm2/m256/m32bcst</i> left by <i>imm8</i> while shifting in 0s using writemask <i>k1</i> .
EVEX.NDD.512.66.0F.W0 72 /6 ib VPSLLD <i>zmm1 {k1}{z}, zmm2/m512/m32bcst, imm8</i>	FVI V/V	AVX512F	Shift doublewords in <i>zmm2/m512/m32bcst</i> left by <i>imm8</i> while shifting in 0s using writemask <i>k1</i> .
EVEX.NDS.128.66.0F.W1 F3 /r VPSLLQ <i>xmm1 {k1}{z}, xmm2, xmm3/m128</i>	M128 V/V	AVX512VL AVX512F	Shift quadwords in <i>xmm2</i> left by amount specified in <i>xmm3/m128</i> while shifting in 0s using writemask <i>k1</i> .

EVEX.NDS.256.66.0F.W1 F3 /r VPSLLQ ymm1 {k1}{z}, ymm2, xmm3/m128	M128 V/V	AVX512VL AVX512F	Shift quadwords in ymm2 left by amount specified in xmm3/m128 while shifting in 0s using writemask k1.
EVEX.NDS.512.66.0F.W1 F3 /r VPSLLQ zmm1 {k1}{z}, zmm2, xmm3/m128	M128 V/V	AVX512F	Shift quadwords in zmm2 left by amount specified in xmm3/m128 while shifting in 0s using writemask k1.
EVEX.NDD.128.66.0F.W1 73 /6 ib VPSLLQ xmm1 {k1}{z}, xmm2/m128/m64bcst, imm8	FVI V/V	AVX512VL AVX512F	Shift quadwords in xmm2/m128/m64bcst left by imm8 while shifting in 0s using writemask k1.
EVEX.NDD.256.66.0F.W1 73 /6 ib VPSLLQ ymm1 {k1}{z}, ymm2/m256/m64bcst, imm8	FVI V/V	AVX512VL AVX512F	Shift quadwords in ymm2/m256/m64bcst left by imm8 while shifting in 0s using writemask k1.
EVEX.NDD.512.66.0F.W1 73 /6 ib VPSLLQ zmm1 {k1}{z}, zmm2/m512/m64bcst, imm8	FVI V/V	AVX512F	Shift quadwords in zmm2/m512/m64bcst left by imm8 while shifting in 0s using writemask k1.

**NOTES:**

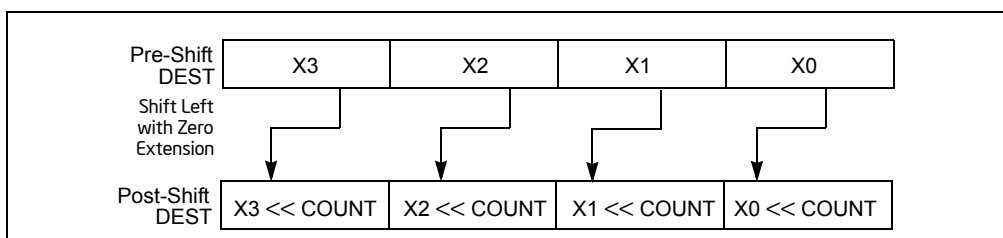
1. See note in Section 2.4, “AVX and SSE Instruction Exception Specification” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
MI	ModRM:r/m (r, w)	imm8	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
VMI	VEX.vvvv (w)	ModRM:r/m (r)	imm8	NA
FVMI	EVEX.vvvv (w)	ModRM:r/m (R)	Imm8	NA
FVI	EVEX.vvvv (w)	ModRM:r/m (R)	Imm8	NA
M128	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

**Description**

Shifts the bits in the individual data elements (words, doublewords, or quadword) in the destination operand (first operand) to the left by the number of bits specified in the count operand (second operand). As the bits in the data elements are shifted left, the empty low-order bits are cleared (set to 0). If the value specified by the count operand is greater than 15 (for words), 31 (for doublewords), or 63 (for a quadword), then the destination operand is set to all 0s. Figure 4-13 gives an example of shifting words in a 64-bit operand.



**Figure 4-13. PSLLW, PSLLD, and PSLLQ Instruction Operation Using 64-bit Operand**



The (V)PSLLW instruction shifts each of the words in the destination operand to the left by the number of bits specified in the count operand; the (V)PSLLD instruction shifts each of the doublewords in the destination operand; and the (V)PSLLQ instruction shifts the quadword (or quadwords) in the destination operand.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE instructions 64-bit operand: The destination operand is an MMX technology register; the count operand can be either an MMX technology register or a 64-bit memory location.

128-bit Legacy SSE version: The destination and first source operands are XMM registers. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged. The count operand can be either an XMM register or a 128-bit memory location or an 8-bit immediate. If the count operand is a memory address, 128 bits are loaded but the upper 64 bits are ignored.

VEX.128 encoded version: The destination and first source operands are XMM registers. Bits (VLMAX-1:128) of the destination YMM register are zeroed. The count operand can be either an XMM register or a 128-bit memory location or an 8-bit immediate. If the count operand is a memory address, 128 bits are loaded but the upper 64 bits are ignored.

VEX.256 encoded version: The destination operand is a YMM register. The source operand is a YMM register or a memory location. The count operand can come either from an XMM register or a memory location or an 8-bit immediate. Bits (MAX\_VL-1:256) of the corresponding ZMM register are zeroed.

EVEX encoded versions: The destination operand is a ZMM register updated according to the writemask. The count operand is either an 8-bit immediate (the immediate count version) or an 8-bit value from an XMM register or a memory location (the variable count version). For the immediate count version, the source operand (the second operand) can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32/64-bit memory location. For the variable count version, the first source operand (the second operand) is a ZMM register, the second source operand (the third operand, 8-bit variable count) can be an XMM register or a memory location.

Note: In VEX/EVEX encoded versions of shifts with an immediate count, vvvv of VEX/EVEX encode the destination register, and VEX.B/EVEX.B + ModRM.r/m encodes the source register.

Note: For shifts with an immediate count (VEX.128.66.0F 71-73 /6, or EVEX.128.66.0F 71-73 /6), VEX.vvvv/EVEX.vvvv encodes the destination register.

## Operation

### PSLLW (with 64-bit operand)

```
IF (COUNT > 15)
  THEN
    DEST[64:0] ← 0000000000000000H;
  ELSE
    DEST[15:0] ← ZeroExtend(DEST[15:0] << COUNT);
    (* Repeat shift operation for 2nd and 3rd words *)
    DEST[63:48] ← ZeroExtend(DEST[63:48] << COUNT);
  FI;
```

### PSLLD (with 64-bit operand)

```
IF (COUNT > 31)
  THEN
    DEST[64:0] ← 0000000000000000H;
  ELSE
    DEST[31:0] ← ZeroExtend(DEST[31:0] << COUNT);
    DEST[63:32] ← ZeroExtend(DEST[63:32] << COUNT);
  FI;
```

### PSLLQ (with 64-bit operand)

```
IF (COUNT > 63)
  THEN
    DEST[64:0] ← 0000000000000000H;
```

```

ELSE
    DEST ← ZeroExtend(DEST << COUNT);
FI;

```

```

LOGICAL_LEFT_SHIFT_WORDS(SRC, COUNT_SRC)
COUNT ← COUNT_SRC[63:0];
IF (COUNT > 15)
THEN
    DEST[127:0] ← 00000000000000000000000000000000H
ELSE
    DEST[15:0] ← ZeroExtend(SRC[15:0] << COUNT);
    (* Repeat shift operation for 2nd through 7th words *)
    DEST[127:112] ← ZeroExtend(SRC[127:112] << COUNT);
FI;

```

```

LOGICAL_LEFT_SHIFT_DWORDS1(SRC, COUNT_SRC)
COUNT ← COUNT_SRC[63:0];
IF (COUNT > 31)
THEN
    DEST[31:0] ← 0
ELSE
    DEST[31:0] ← ZeroExtend(SRC[31:0] << COUNT);
FI;

```

```

LOGICAL_LEFT_SHIFT_DWORDS(SRC, COUNT_SRC)
COUNT ← COUNT_SRC[63:0];
IF (COUNT > 31)
THEN
    DEST[127:0] ← 00000000000000000000000000000000H
ELSE
    DEST[31:0] ← ZeroExtend(SRC[31:0] << COUNT);
    (* Repeat shift operation for 2nd through 3rd words *)
    DEST[127:96] ← ZeroExtend(SRC[127:96] << COUNT);
FI;

```

```

LOGICAL_LEFT_SHIFT_QWORDS1(SRC, COUNT_SRC)
COUNT ← COUNT_SRC[63:0];
IF (COUNT > 63)
THEN
    DEST[63:0] ← 0
ELSE
    DEST[63:0] ← ZeroExtend(SRC[63:0] << COUNT);
FI;

```

```

LOGICAL_LEFT_SHIFT_QWORDS(SRC, COUNT_SRC)
COUNT ← COUNT_SRC[63:0];
IF (COUNT > 63)
THEN
    DEST[127:0] ← 00000000000000000000000000000000H
ELSE
    DEST[63:0] ← ZeroExtend(SRC[63:0] << COUNT);
    DEST[127:64] ← ZeroExtend(SRC[127:64] << COUNT);
FI;
LOGICAL_LEFT_SHIFT_WORDS_256b(SRC, COUNT_SRC)

```

```

COUNT ←COUNT_SRC[63:0];
IF (COUNT > 15)
THEN
    DEST[127:0] ←00000000000000000000000000000000H
    DEST[255:128] ←00000000000000000000000000000000H
ELSE
    DEST[15:0] ←ZeroExtend(SRC[15:0] << COUNT);
    (* Repeat shift operation for 2nd through 15th words *)
    DEST[255:240] ←ZeroExtend(SRC[255:240] << COUNT);
FI;

```

```

LOGICAL_LEFT_SHIFT_DWORDS_256b(SRC, COUNT_SRC)
COUNT ←COUNT_SRC[63:0];
IF (COUNT > 31)
THEN
    DEST[127:0] ←00000000000000000000000000000000H
    DEST[255:128] ←00000000000000000000000000000000H
ELSE
    DEST[31:0] ←ZeroExtend(SRC[31:0] << COUNT);
    (* Repeat shift operation for 2nd through 7th words *)
    DEST[255:224] ←ZeroExtend(SRC[255:224] << COUNT);
FI;

```

```

LOGICAL_LEFT_SHIFT_QWORDS_256b(SRC, COUNT_SRC)
COUNT ←COUNT_SRC[63:0];
IF (COUNT > 63)
THEN
    DEST[127:0] ←00000000000000000000000000000000H
    DEST[255:128] ←00000000000000000000000000000000H
ELSE
    DEST[63:0] ←ZeroExtend(SRC[63:0] << COUNT);
    DEST[127:64] ←ZeroExtend(SRC[127:64] << COUNT);
    DEST[191:128] ←ZeroExtend(SRC[191:128] << COUNT);
    DEST[255:192] ←ZeroExtend(SRC[255:192] << COUNT);
FI;

```

**VPSLLW (EVEX versions, xmm/m128)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

IF VL = 128

```

    TMP_DEST[127:0] ← LOGICAL_LEFT_SHIFT_WORDS_128b(SRC1[127:0], SRC2)

```

FI;

IF VL = 256

```

    TMP_DEST[255:0] ← LOGICAL_LEFT_SHIFT_WORDS_256b(SRC1[255:0], SRC2)

```

FI;

IF VL = 512

```

    TMP_DEST[255:0] ← LOGICAL_LEFT_SHIFT_WORDS_256b(SRC1[255:0], SRC2)

```

```

    TMP_DEST[511:256] ← LOGICAL_LEFT_SHIFT_WORDS_256b(SRC1[511:256], SRC2)

```

FI;

FOR j ← 0 TO KL-1

i ← j \* 16

IF k1[j] OR \*no writemask\*

```

    THEN DEST[i+15:i] ← TMP_DEST[i+15:i]

```

```

    ELSE

```

```

        IF *merging-masking*           ; merging-masking
            THEN *DEST[i+15:i] remains unchanged*
            ELSE *zeroing-masking*     ; zeroing-masking
                DEST[i+15:i] = 0
        FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VPSLLW (EVEX versions, imm8)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

IF VL = 128

```

    TMP_DEST[127:0] ← LOGICAL_LEFT_SHIFT_WORDS_128b(SRC1[127:0], imm8)

```

FI;

IF VL = 256

```

    TMP_DEST[255:0] ← LOGICAL_RIGHT_SHIFT_WORDS_256b(SRC1[255:0], imm8)

```

FI;

IF VL = 512

```

    TMP_DEST[255:0] ← LOGICAL_LEFT_SHIFT_WORDS_256b(SRC1[255:0], imm8)

```

```

    TMP_DEST[511:256] ← LOGICAL_LEFT_SHIFT_WORDS_256b(SRC1[511:256], imm8)

```

FI;

FOR j ← 0 TO KL-1

i ← j \* 16

IF k1[j] OR \*no writemask\*

```

    THEN DEST[i+15:i] ← TMP_DEST[i+15:i]

```

ELSE

```

    IF *merging-masking*           ; merging-masking

```

```

        THEN *DEST[i+15:i] remains unchanged*

```

```

        ELSE *zeroing-masking*     ; zeroing-masking

```

```

            DEST[i+15:i] = 0

```

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VPSLLW (ymm, ymm, xmm/m128) - VEX.256 encoding**

DEST[255:0] ← LOGICAL\_LEFT\_SHIFT\_WORDS\_256b(SRC1, SRC2)

DEST[MAX\_VL-1:256] ← 0;

**VPSLLW (ymm, imm8) - VEX.256 encoding**

DEST[255:0] ← LOGICAL\_LEFT\_SHIFT\_WORD\_256b(SRC1, imm8)

DEST[MAX\_VL-1:256] ← 0;

**VPSLLW (xmm, xmm, xmm/m128) - VEX.128 encoding**

DEST[127:0] ← LOGICAL\_LEFT\_SHIFT\_WORDS(SRC1, SRC2)

DEST[MAX\_VL-1:128] ← 0

**VPSLLW (xmm, imm8) - VEX.128 encoding**

DEST[127:0] ← LOGICAL\_LEFT\_SHIFT\_WORDS(SRC1, imm8)

DEST[MAX\_VL-1:128] ← 0

**PSLLW (xmm, xmm, xmm/m128)**

DEST[127:0] ← LOGICAL\_LEFT\_SHIFT\_WORDS(DEST, SRC)

DEST[MAX\_VL-1:128] (Unmodified)

**PSLLW (xmm, imm8)**

DEST[127:0] ← LOGICAL\_LEFT\_SHIFT\_WORDS(DEST, imm8)

DEST[MAX\_VL-1:128] (Unmodified)

**VPSLLD (EVEX versions, imm8)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\* THEN

IF (EVEX.b = 1) AND (SRC1 \*is memory\*)

THEN DEST[i+31:i] ← LOGICAL\_LEFT\_SHIFT\_DWORDS1(SRC1[31:0], imm8)

ELSE DEST[i+31:i] ← LOGICAL\_LEFT\_SHIFT\_DWORDS1(SRC1[i+31:i], imm8)

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+31:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VPSLLD (EVEX versions, xmm/m128)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

IF VL = 128

TMP\_DEST[127:0] ← LOGICAL\_LEFT\_SHIFT\_DWORDS\_128b(SRC1[127:0], SRC2)

FI;

IF VL = 256

TMP\_DEST[255:0] ← LOGICAL\_LEFT\_SHIFT\_DWORDS\_256b(SRC1[255:0], SRC2)

FI;

IF VL = 512

TMP\_DEST[255:0] ← LOGICAL\_LEFT\_SHIFT\_DWORDS\_256b(SRC1[255:0], SRC2)

TMP\_DEST[511:256] ← LOGICAL\_LEFT\_SHIFT\_DWORDS\_256b(SRC1[511:256], SRC2)

FI;

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\*

THEN DEST[i+31:i] ← TMP\_DEST[i+31:i]

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+31:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VPSLLD (ymm, ymm, xmm/m128) - VEX.256 encoding**

DEST[255:0] ← LOGICAL\_LEFT\_SHIFT\_DWORDS\_256b(SRC1, SRC2)  
 DEST[MAX\_VL-1:256] ← 0;

**VPSLLD (ymm, imm8) - VEX.256 encoding**

DEST[255:0] ← LOGICAL\_LEFT\_SHIFT\_DWORDS\_256b(SRC1, imm8)  
 DEST[MAX\_VL-1:256] ← 0;

**VPSLLD (xmm, xmm, xmm/m128) - VEX.128 encoding**

DEST[127:0] ← LOGICAL\_LEFT\_SHIFT\_DWORDS(SRC1, SRC2)  
 DEST[MAX\_VL-1:128] ← 0

**VPSLLD (xmm, imm8) - VEX.128 encoding**

DEST[127:0] ← LOGICAL\_LEFT\_SHIFT\_DWORDS(SRC1, imm8)  
 DEST[MAX\_VL-1:128] ← 0

**PSLLD (xmm, xmm, xmm/m128)**

DEST[127:0] ← LOGICAL\_LEFT\_SHIFT\_DWORDS(DEST, SRC)  
 DEST[MAX\_VL-1:128] (Unmodified)

**PSLLD (xmm, imm8)**

DEST[127:0] ← LOGICAL\_LEFT\_SHIFT\_DWORDS(DEST, imm8)  
 DEST[MAX\_VL-1:128] (Unmodified)

**VPSLLQ (EVEX versions, imm8)**

(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) AND (SRC1 \*is memory\*)

      THEN DEST[i+63:i] ← LOGICAL\_LEFT\_SHIFT\_QWORDS1(SRC1[63:0], imm8)

      ELSE DEST[i+63:i] ← LOGICAL\_LEFT\_SHIFT\_QWORDS1(SRC1[i+63:i], imm8)

    FI;

  ELSE

    IF \*merging-masking\* ; merging-masking

      THEN \*DEST[i+63:i] remains unchanged\*

      ELSE \*zeroing-masking\* ; zeroing-masking

        DEST[i+63:i] ← 0

    FI

  FI;

ENDFOR

**VPSLLQ (EVEX versions, xmm/m128)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

IF VL = 128

  TMP\_DEST[127:0] ← LOGICAL\_LEFT\_SHIFT\_QWORDS\_128b(SRC1[127:0], SRC2)

  FI;

IF VL = 256

  TMP\_DEST[255:0] ← LOGICAL\_LEFT\_SHIFT\_QWORDS\_256b(SRC1[255:0], SRC2)

  FI;

IF VL = 512

  TMP\_DEST[255:0] ← LOGICAL\_LEFT\_SHIFT\_QWORDS\_256b(SRC1[255:0], SRC2)

  TMP\_DEST[511:256] ← LOGICAL\_LEFT\_SHIFT\_QWORDS\_256b(SRC1[511:256], SRC2)

  FI;

```

FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[j+63:i] ← TMP_DEST[j+63:i]
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[j+63:i] remains unchanged*
    ELSE *zeroing-masking* ; zeroing-masking
      DEST[j+63:i] ← 0
    FI
  FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VPSLLQ (ymm, ymm, xmm/m128) - VEX.256 encoding**

```

DEST[255:0] ← LOGICAL_LEFT_SHIFT_QWORDS_256b(SRC1, SRC2)
DEST[MAX_VL-1:256] ← 0;

```

**VPSLLQ (ymm, imm8) - VEX.256 encoding**

```

DEST[255:0] ← LOGICAL_LEFT_SHIFT_QWORDS_256b(SRC1, imm8)
DEST[MAX_VL-1:256] ← 0;

```

**VPSLLQ (xmm, xmm, xmm/m128) - VEX.128 encoding**

```

DEST[127:0] ← LOGICAL_LEFT_SHIFT_QWORDS(SRC1, SRC2)
DEST[MAX_VL-1:128] ← 0

```

**VPSLLQ (xmm, imm8) - VEX.128 encoding**

```

DEST[127:0] ← LOGICAL_LEFT_SHIFT_QWORDS(SRC1, imm8)
DEST[MAX_VL-1:128] ← 0

```

**PSLLQ (xmm, xmm, xmm/m128)**

```

DEST[127:0] ← LOGICAL_LEFT_SHIFT_QWORDS(DEST, SRC)
DEST[MAX_VL-1:128] (Unmodified)

```

**PSLLQ (xmm, imm8)**

```

DEST[127:0] ← LOGICAL_LEFT_SHIFT_QWORDS(DEST, imm8)
DEST[MAX_VL-1:128] (Unmodified)

```

**Intel C/C++ Compiler Intrinsic Equivalents**

```

VPSLLD __m512i __mm512_slli_epi32(__m512i a, unsigned int imm);
VPSLLD __m512i __mm512_mask_slli_epi32(__m512i s, __mmask16 k, __m512i a, unsigned int imm);
VPSLLD __m512i __mm512_maskz_slli_epi32(__mmask16 k, __m512i a, unsigned int imm);
VPSLLD __m256i __mm256_mask_slli_epi32(__m256i s, __mmask8 k, __m256i a, unsigned int imm);
VPSLLD __m256i __mm256_maskz_slli_epi32(__mmask8 k, __m256i a, unsigned int imm);
VPSLLD __m128i __mm_mask_slli_epi32(__m128i s, __mmask8 k, __m128i a, unsigned int imm);
VPSLLD __m128i __mm_maskz_slli_epi32(__mmask8 k, __m128i a, unsigned int imm);
VPSLLD __m512i __mm512_sll_epi32(__m512i a, __m128i cnt);
VPSLLD __m512i __mm512_mask_sll_epi32(__m512i s, __mmask16 k, __m512i a, __m128i cnt);
VPSLLD __m512i __mm512_maskz_sll_epi32(__mmask16 k, __m512i a, __m128i cnt);
VPSLLD __m256i __mm256_mask_sll_epi32(__m256i s, __mmask8 k, __m256i a, __m128i cnt);
VPSLLD __m256i __mm256_maskz_sll_epi32(__mmask8 k, __m256i a, __m128i cnt);
VPSLLD __m128i __mm_mask_sll_epi32(__m128i s, __mmask8 k, __m128i a, __m128i cnt);
VPSLLD __m128i __mm_maskz_sll_epi32(__mmask8 k, __m128i a, __m128i cnt);

```

VPSLLQ \_\_m512i \_mm512\_mask\_slli\_epi64(\_\_m512i a, unsigned int imm);  
 VPSLLQ \_\_m512i \_mm512\_mask\_slli\_epi64(\_\_m512i s, \_\_mmask8 k, \_\_m512i a, unsigned int imm);  
 VPSLLQ \_\_m512i \_mm512\_maskz\_slli\_epi64(\_\_mmask8 k, \_\_m512i a, unsigned int imm);  
 VPSLLQ \_\_m256i \_mm256\_mask\_slli\_epi64(\_\_m256i s, \_\_mmask8 k, \_\_m256i a, unsigned int imm);  
 VPSLLQ \_\_m256i \_mm256\_maskz\_slli\_epi64(\_\_mmask8 k, \_\_m256i a, unsigned int imm);  
 VPSLLQ \_\_m128i \_mm\_mask\_slli\_epi64(\_\_m128i s, \_\_mmask8 k, \_\_m128i a, unsigned int imm);  
 VPSLLQ \_\_m128i \_mm\_maskz\_slli\_epi64(\_\_mmask8 k, \_\_m128i a, unsigned int imm);  
 VPSLLQ \_\_m512i \_mm512\_mask\_sll\_epi64(\_\_m512i a, \_\_m128i cnt);  
 VPSLLQ \_\_m512i \_mm512\_mask\_sll\_epi64(\_\_m512i s, \_\_mmask8 k, \_\_m512i a, \_\_m128i cnt);  
 VPSLLQ \_\_m512i \_mm512\_maskz\_sll\_epi64(\_\_mmask8 k, \_\_m512i a, \_\_m128i cnt);  
 VPSLLQ \_\_m256i \_mm256\_mask\_sll\_epi64(\_\_m256i s, \_\_mmask8 k, \_\_m256i a, \_\_m128i cnt);  
 VPSLLQ \_\_m256i \_mm256\_maskz\_sll\_epi64(\_\_mmask8 k, \_\_m256i a, \_\_m128i cnt);  
 VPSLLQ \_\_m128i \_mm\_mask\_sll\_epi64(\_\_m128i s, \_\_mmask8 k, \_\_m128i a, \_\_m128i cnt);  
 VPSLLQ \_\_m128i \_mm\_maskz\_sll\_epi64(\_\_mmask8 k, \_\_m128i a, \_\_m128i cnt);  
 VPSLLW \_\_m512i \_mm512\_slli\_epi16(\_\_m512i a, unsigned int imm);  
 VPSLLW \_\_m512i \_mm512\_mask\_slli\_epi16(\_\_m512i s, \_\_mmask32 k, \_\_m512i a, unsigned int imm);  
 VPSLLW \_\_m512i \_mm512\_maskz\_slli\_epi16(\_\_mmask32 k, \_\_m512i a, unsigned int imm);  
 VPSLLW \_\_m256i \_mm256\_mask\_sllii\_epi16(\_\_m256i s, \_\_mmask16 k, \_\_m256i a, unsigned int imm);  
 VPSLLW \_\_m256i \_mm256\_maskz\_sllii\_epi16(\_\_mmask16 k, \_\_m256i a, unsigned int imm);  
 VPSLLW \_\_m128i \_mm\_mask\_slli\_epi16(\_\_m128i s, \_\_mmask8 k, \_\_m128i a, unsigned int imm);  
 VPSLLW \_\_m128i \_mm\_maskz\_slli\_epi16(\_\_mmask8 k, \_\_m128i a, unsigned int imm);  
 VPSLLW \_\_m512i \_mm512\_sll\_epi16(\_\_m512i a, \_\_m128i cnt);  
 VPSLLW \_\_m512i \_mm512\_mask\_sll\_epi16(\_\_m512i s, \_\_mmask32 k, \_\_m512i a, \_\_m128i cnt);  
 VPSLLW \_\_m512i \_mm512\_maskz\_sll\_epi16(\_\_mmask32 k, \_\_m512i a, \_\_m128i cnt);  
 VPSLLW \_\_m256i \_mm256\_mask\_sll\_epi16(\_\_m256i s, \_\_mmask16 k, \_\_m256i a, \_\_m128i cnt);  
 VPSLLW \_\_m256i \_mm256\_maskz\_sll\_epi16(\_\_mmask16 k, \_\_m256i a, \_\_m128i cnt);  
 VPSLLW \_\_m128i \_mm\_mask\_sll\_epi16(\_\_m128i s, \_\_mmask8 k, \_\_m128i a, \_\_m128i cnt);  
 VPSLLW \_\_m128i \_mm\_maskz\_sll\_epi16(\_\_mmask8 k, \_\_m128i a, \_\_m128i cnt);  
 PSLLW: \_\_m64 \_mm\_slli\_pi16(\_\_m64 m, int count)  
 PSLLW: \_\_m64 \_mm\_sll\_pi16(\_\_m64 m, \_\_m64 count)  
 (V)PSLLW: \_\_m128i \_mm\_slli\_pi16(\_\_m64 m, int count)  
 (V)PSLLW: \_\_m128i \_mm\_slli\_pi16(\_\_m128i m, \_\_m128i count)  
 VPSLLW: \_\_m256i \_mm256\_slli\_epi16(\_\_m256i m, int count)  
 VPSLLW: \_\_m256i \_mm256\_sll\_epi16(\_\_m256i m, \_\_m128i count)  
 PSLLD: \_\_m64 \_mm\_slli\_pi32(\_\_m64 m, int count)  
 PSLLD: \_\_m64 \_mm\_sll\_pi32(\_\_m64 m, \_\_m64 count)  
 (V)PSLLD: \_\_m128i \_mm\_slli\_epi32(\_\_m128i m, int count)  
 (V)PSLLD: \_\_m128i \_mm\_sll\_epi32(\_\_m128i m, \_\_m128i count)  
 VPSLLD: \_\_m256i \_mm256\_slli\_epi32(\_\_m256i m, int count)  
 VPSLLD: \_\_m256i \_mm256\_sll\_epi32(\_\_m256i m, \_\_m128i count)  
 PSLLQ: \_\_m64 \_mm\_slli\_si64(\_\_m64 m, int count)  
 PSLLQ: \_\_m64 \_mm\_sll\_si64(\_\_m64 m, \_\_m64 count)  
 (V)PSLLQ: \_\_m128i \_mm\_slli\_epi64(\_\_m128i m, int count)  
 (V)PSLLQ: \_\_m128i \_mm\_sll\_epi64(\_\_m128i m, \_\_m128i count)  
 VPSLLQ: \_\_m256i \_mm256\_slli\_epi64(\_\_m256i m, int count)  
 VPSLLQ: \_\_m256i \_mm256\_sll\_epi64(\_\_m256i m, \_\_m128i count)

### Flags Affected

None.

### Numeric Exceptions

None.



## Other Exceptions

VEX-encoded instructions:

- Syntax with RM/RVM operand encoding, see Exceptions Type 4.

- Syntax with MI/VMI operand encoding, see Exceptions Type 7.

EVEX-encoded VPSLLW, see Exceptions Type E4NF.nb.

EVEX-encoded VPSLLD/Q:

- Syntax with M128 operand encoding, see Exceptions Type E4NF.nb.

- Syntax with FVI operand encoding, see Exceptions Type E4.

## PSRAW/PSRAD/PSRAQ—Shift Packed Data Right Arithmetic

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F E1 /r <sup>1</sup> PSRAW <i>mm</i> , <i>mm/m64</i>	RM	V/V	MMX	Shift words in <i>mm</i> right by <i>mm/m64</i> while shifting in sign bits.
66 0F E1 /r PSRAW <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE2	Shift words in <i>xmm1</i> right by <i>xmm2/m128</i> while shifting in sign bits.
0F 71 /4 ib <sup>1</sup> PSRAW <i>mm</i> , <i>imm8</i>	MI	V/V	MMX	Shift words in <i>mm</i> right by <i>imm8</i> while shifting in sign bits.
66 0F 71 /4 ib PSRAW <i>xmm1</i> , <i>imm8</i>	MI	V/V	SSE2	Shift words in <i>xmm1</i> right by <i>imm8</i> while shifting in sign bits.
0F E2 /r <sup>1</sup> PSRAD <i>mm</i> , <i>mm/m64</i>	RM	V/V	MMX	Shift doublewords in <i>mm</i> right by <i>mm/m64</i> while shifting in sign bits.
66 0F E2 /r PSRAD <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE2	Shift doubleword in <i>xmm1</i> right by <i>xmm2/m128</i> while shifting in sign bits.
0F 72 /4 ib <sup>1</sup> PSRAD <i>mm</i> , <i>imm8</i>	MI	V/V	MMX	Shift doublewords in <i>mm</i> right by <i>imm8</i> while shifting in sign bits.
66 0F 72 /4 ib PSRAD <i>xmm1</i> , <i>imm8</i>	MI	V/V	SSE2	Shift doublewords in <i>xmm1</i> right by <i>imm8</i> while shifting in sign bits.
VEX.NDS.128.66.0F.WIG E1 /r VPSRAW <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Shift words in <i>xmm2</i> right by amount specified in <i>xmm3/m128</i> while shifting in sign bits.
VEX.NDD.128.66.0F.WIG 71 /4 ib VPSRAW <i>xmm1</i> , <i>xmm2</i> , <i>imm8</i>	VMI	V/V	AVX	Shift words in <i>xmm2</i> right by <i>imm8</i> while shifting in sign bits.
VEX.NDS.128.66.0F.WIG E2 /r VPSRAD <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Shift doublewords in <i>xmm2</i> right by amount specified in <i>xmm3/m128</i> while shifting in sign bits.
VEX.NDD.128.66.0F.WIG 72 /4 ib VPSRAD <i>xmm1</i> , <i>xmm2</i> , <i>imm8</i>	VMI	V/V	AVX	Shift doublewords in <i>xmm2</i> right by <i>imm8</i> while shifting in sign bits.
VEX.NDS.256.66.0F.WIG E1 /r VPSRAW <i>ymm1</i> , <i>ymm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX2	Shift words in <i>ymm2</i> right by amount specified in <i>xmm3/m128</i> while shifting in sign bits.
VEX.NDD.256.66.0F.WIG 71 /4 ib VPSRAW <i>ymm1</i> , <i>ymm2</i> , <i>imm8</i>	VMI	V/V	AVX2	Shift words in <i>ymm2</i> right by <i>imm8</i> while shifting in sign bits.
VEX.NDS.256.66.0F.WIG E2 /r VPSRAD <i>ymm1</i> , <i>ymm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX2	Shift doublewords in <i>ymm2</i> right by amount specified in <i>xmm3/m128</i> while shifting in sign bits.
VEX.NDD.256.66.0F.WIG 72 /4 ib VPSRAD <i>ymm1</i> , <i>ymm2</i> , <i>imm8</i>	VMI	V/V	AVX2	Shift doublewords in <i>ymm2</i> right by <i>imm8</i> while shifting in sign bits.
EVEX.NDS.128.66.0F.WIG E1 /r VPSRAW <i>xmm1</i> {k1}{z}, <i>xmm2</i> , <i>xmm3/m128</i>	M128	V/V	AVX512VL AVX512BW	Shift words in <i>xmm2</i> right by amount specified in <i>xmm3/m128</i> while shifting in sign bits using writemask k1.
EVEX.NDS.256.66.0F.WIG E1 /r VPSRAW <i>ymm1</i> {k1}{z}, <i>ymm2</i> , <i>xmm3/m128</i>	M128	V/V	AVX512VL AVX512BW	Shift words in <i>ymm2</i> right by amount specified in <i>xmm3/m128</i> while shifting in sign bits using writemask k1.

EVEX.NDS.512.66.0F.WIG E1 /r VPSRAW zmm1 {k1}{z}, zmm2, xmm3/m128	M128 V/V	AVX512BW	Shift words in zmm2 right by amount specified in xmm3/m128 while shifting in sign bits using writemask k1.
EVEX.NDD.128.66.0F.WIG 71 /4 ib VPSRAW xmm1 {k1}{z}, xmm2/m128, imm8	FVMI V/V	AVX512VL AVX512BW	Shift words in xmm2/m128 right by imm8 while shifting in sign bits using writemask k1.
EVEX.NDD.256.66.0F.WIG 71 /4 ib VPSRAW ymm1 {k1}{z}, ymm2/m256, imm8	FVMI V/V	AVX512VL AVX512BW	Shift words in ymm2/m256 right by imm8 while shifting in sign bits using writemask k1.
EVEX.NDD.512.66.0F.WIG 71 /4 ib VPSRAW zmm1 {k1}{z}, zmm2/m512, imm8	FVMI V/V	AVX512BW	Shift words in zmm2/m512 right by imm8 while shifting in sign bits using writemask k1.
EVEX.NDS.128.66.0F.W0 E2 /r VPSRAD xmm1 {k1}{z}, xmm2, xmm3/m128	M128 V/V	AVX512VL AVX512F	Shift doublewords in xmm2 right by amount specified in xmm3/m128 while shifting in sign bits using writemask k1.
EVEX.NDS.256.66.0F.W0 E2 /r VPSRAD ymm1 {k1}{z}, ymm2, xmm3/m128	M128 V/V	AVX512VL AVX512F	Shift doublewords in ymm2 right by amount specified in xmm3/m128 while shifting in sign bits using writemask k1.
EVEX.NDS.512.66.0F.W0 E2 /r VPSRAD zmm1 {k1}{z}, zmm2, xmm3/m128	M128 V/V	AVX512F	Shift doublewords in zmm2 right by amount specified in xmm3/m128 while shifting in sign bits using writemask k1.
EVEX.NDD.128.66.0F.W0 72 /4 ib VPSRAD xmm1 {k1}{z}, xmm2/m128/m32bcst, imm8	FVI V/V	AVX512VL AVX512F	Shift doublewords in xmm2/m128/m32bcst right by imm8 while shifting in sign bits using writemask k1.
EVEX.NDD.256.66.0F.W0 72 /4 ib VPSRAD ymm1 {k1}{z}, ymm2/m256/m32bcst, imm8	FVI V/V	AVX512VL AVX512F	Shift doublewords in ymm2/m256/m32bcst right by imm8 while shifting in sign bits using writemask k1.
EVEX.NDD.512.66.0F.W0 72 /4 ib VPSRAD zmm1 {k1}{z}, zmm2/m512/m32bcst, imm8	FVI V/V	AVX512F	Shift doublewords in zmm2/m512/m32bcst right by imm8 while shifting in sign bits using writemask k1.
EVEX.NDS.128.66.0F.W1 E2 /r VPSRAQ xmm1 {k1}{z}, xmm2, xmm3/m128	M128 V/V	AVX512VL AVX512F	Shift quadwords in xmm2 right by amount specified in xmm3/m128 while shifting in sign bits using writemask k1.
EVEX.NDS.256.66.0F.W1 E2 /r VPSRAQ ymm1 {k1}{z}, ymm2, xmm3/m128	M128 V/V	AVX512VL AVX512F	Shift quadwords in ymm2 right by amount specified in xmm3/m128 while shifting in sign bits using writemask k1.
EVEX.NDS.512.66.0F.W1 E2 /r VPSRAQ zmm1 {k1}{z}, zmm2, xmm3/m128	M128 V/V	AVX512F	Shift quadwords in zmm2 right by amount specified in xmm3/m128 while shifting in sign bits using writemask k1.
EVEX.NDD.128.66.0F.W1 72 /4 ib VPSRAQ xmm1 {k1}{z}, xmm2/m128/m64bcst, imm8	FVI V/V	AVX512VL AVX512F	Shift quadwords in xmm2/m128/m64bcst right by imm8 while shifting in sign bits using writemask k1.
EVEX.NDD.256.66.0F.W1 72 /4 ib VPSRAQ ymm1 {k1}{z}, ymm2/m256/m64bcst, imm8	FVI V/V	AVX512VL AVX512F	Shift quadwords in ymm2/m256/m64bcst right by imm8 while shifting in sign bits using writemask k1.

EVEX.NDD.512.66.0F.W1 72 /4 ib	FVI	V/V	AVX512F	Shift quadwords in zmm2/m512/m64bcst right by imm8 while shifting in sign bits using writemask k1.
VPSRAQ zmm1 {k1}{z}, zmm2/m512/m64bcst, imm8				

**NOTES:**

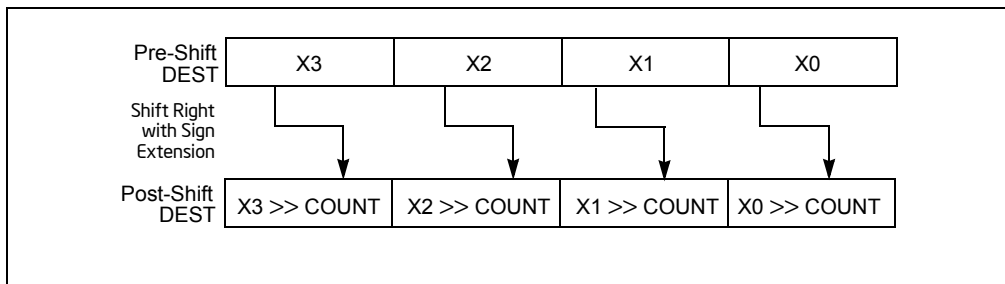
1. See note in Section 2.4, “AVX and SSE Instruction Exception Specification” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
MI	ModRM:r/m (r, w)	imm8	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
VMI	VEX.vvvv (w)	ModRM:r/m (r)	imm8	NA
FVMI	EVEX.vvvv (w)	ModRM:r/m (R)	Imm8	NA
FVI	EVEX.vvvv (w)	ModRM:r/m (R)	Imm8	NA
M128	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

**Description**

Shifts the bits in the individual data elements (words, doublewords or quadwords) in the destination operand (first operand) to the right by the number of bits specified in the count operand (second operand). As the bits in the data elements are shifted right, the empty high-order bits are filled with the initial value of the sign bit of the data element. If the value specified by the count operand is greater than 15 (for words), 31 (for doublewords), or 63 (for quadwords), each destination data element is filled with the initial value of the sign bit of the element. (Figure 4-14 gives an example of shifting words in a 64-bit operand.)



**Figure 4-14. PSRAW and PSRAD Instruction Operation Using a 64-bit Operand**

Note that only the first 64-bits of a 128-bit count operand are checked to compute the count. If the second source operand is a memory address, 128 bits are loaded.

The (V)PSRAW instruction shifts each of the words in the destination operand to the right by the number of bits specified in the count operand, and the (V)PSRAD instruction shifts each of the doublewords in the destination operand.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE instructions 64-bit operand: The destination operand is an MMX technology register; the count operand can be either an MMX technology register or a 64-bit memory location.

128-bit Legacy SSE version: The destination and first source operands are XMM registers. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged. The count operand can be either an XMM register or a 128-bit memory location or an 8-bit immediate. If the count operand is a memory address, 128 bits are loaded but the upper 64 bits are ignored.

VEX.128 encoded version: The destination and first source operands are XMM registers. Bits (VLMAX-1:128) of the destination YMM register are zeroed. The count operand can be either an XMM register or a 128-bit memory location or an 8-bit immediate. If the count operand is a memory address, 128 bits are loaded but the upper 64 bits are ignored.

VEX.256 encoded version: The destination operand is a YMM register. The source operand is a YMM register or a memory location. The count operand can come either from an XMM register or a memory location or an 8-bit immediate. Bits (MAX\_VL-1:256) of the corresponding ZMM register are zeroed.

EVEX encoded versions: The destination operand is a ZMM register updated according to the writemask. The count operand is either an 8-bit immediate (the immediate count version) or an 8-bit value from an XMM register or a memory location (the variable count version). For the immediate count version, the source operand (the second operand) can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32/64-bit memory location. For the variable count version, the first source operand (the second operand) is a ZMM register, the second source operand (the third operand, 8-bit variable count) can be an XMM register or a memory location.

Note: In VEX/EVEX encoded versions of shifts with an immediate count, vvvv of VEX/EVEX encode the destination register, and VEX.B/EVEX.B + ModRM.r/m encodes the source register.

Note: For shifts with an immediate count (VEX.128.66.0F 71-73 /4, EVEX.128.66.0F 71-73 /4), VEX.vvvv/EVEX.vvvv encodes the destination register.

## Operation

### PSRAW (with 64-bit operand)

```
IF (COUNT > 15)
    THEN COUNT ← 16;
FI;
DEST[15:0] ← SignExtend(DEST[15:0] >> COUNT);
(* Repeat shift operation for 2nd and 3rd words *)
DEST[63:48] ← SignExtend(DEST[63:48] >> COUNT);
```

### PSRAD (with 64-bit operand)

```
IF (COUNT > 31)
    THEN COUNT ← 32;
FI;
DEST[31:0] ← SignExtend(DEST[31:0] >> COUNT);
DEST[63:32] ← SignExtend(DEST[63:32] >> COUNT);
```

ARITHMETIC\_RIGHT\_SHIFT\_DWORDS1(SRC, COUNT\_SRC)

COUNT ← COUNT\_SRC[63:0];

IF (COUNT > 31)

THEN

DEST[31:0] ← SignBit

ELSE

DEST[31:0] ← SignExtend(SRC[31:0] >> COUNT);

FI;

ARITHMETIC\_RIGHT\_SHIFT\_QWORDS1(SRC, COUNT\_SRC)

COUNT ← COUNT\_SRC[63:0];

IF (COUNT > 63)

THEN

DEST[63:0] ← SignBit

ELSE

```

DEST[63:0] ← SignExtend(SRC[63:0] >> COUNT);
FI;

```

```

ARITHMETIC_RIGHT_SHIFT_WORDS_256b(SRC, COUNT_SRC)
COUNT ← COUNT_SRC[63:0];
IF (COUNT > 15)
    THEN    COUNT ← 16;

```

```

FI;
DEST[15:0] ← SignExtend(SRC[15:0] >> COUNT);
(* Repeat shift operation for 2nd through 15th words *)
DEST[255:240] ← SignExtend(SRC[255:240] >> COUNT);

```

```

ARITHMETIC_RIGHT_SHIFT_DWORDS_256b(SRC, COUNT_SRC)
COUNT ← COUNT_SRC[63:0];
IF (COUNT > 31)
    THEN    COUNT ← 32;

```

```

FI;
DEST[31:0] ← SignExtend(SRC[31:0] >> COUNT);
(* Repeat shift operation for 2nd through 7th words *)
DEST[255:224] ← SignExtend(SRC[255:224] >> COUNT);

```

```

ARITHMETIC_RIGHT_SHIFT_QWORDS(SRC, COUNT_SRC, VL)           ; VL: 128b, 256b or 512b
COUNT ← COUNT_SRC[63:0];
IF (COUNT > 63)
    THEN    COUNT ← 64;

```

```

FI;
DEST[63:0] ← SignExtend(SRC[63:0] >> COUNT);
(* Repeat shift operation for 2nd through 7th words *)
DEST[VL-1:VL-64] ← SignExtend(SRC[VL-1:VL-64] >> COUNT);

```

```

ARITHMETIC_RIGHT_SHIFT_WORDS(SRC, COUNT_SRC)
COUNT ← COUNT_SRC[63:0];
IF (COUNT > 15)
    THEN    COUNT ← 16;

```

```

FI;
DEST[15:0] ← SignExtend(SRC[15:0] >> COUNT);
(* Repeat shift operation for 2nd through 7th words *)
DEST[127:112] ← SignExtend(SRC[127:112] >> COUNT);

```

```

ARITHMETIC_RIGHT_SHIFT_DWORDS(SRC, COUNT_SRC)
COUNT ← COUNT_SRC[63:0];
IF (COUNT > 31)
    THEN    COUNT ← 32;

```

```

FI;
DEST[31:0] ← SignExtend(SRC[31:0] >> COUNT);
(* Repeat shift operation for 2nd through 3rd words *)
DEST[127:96] ← SignExtend(SRC[127:96] >> COUNT);

```

**VPSRAW (EVEX versions, xmm/m128)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

IF VL = 128

TMP\_DEST[127:0] ← ARITHMETIC\_RIGHT\_SHIFT\_WORDS\_128b(SRC1[127:0], SRC2)

FI;

IF VL = 256

TMP\_DEST[255:0] ← ARITHMETIC\_RIGHT\_SHIFT\_WORDS\_256b(SRC1[255:0], SRC2)

FI;

IF VL = 512

TMP\_DEST[255:0] ← ARITHMETIC\_RIGHT\_SHIFT\_WORDS\_256b(SRC1[255:0], SRC2)

TMP\_DEST[511:256] ← ARITHMETIC\_RIGHT\_SHIFT\_WORDS\_256b(SRC1[511:256], SRC2)

FI;

FOR j ← 0 TO KL-1

i ← j \* 16

IF k1[j] OR \*no writemask\*

THEN DEST[i+15:i] ← TMP\_DEST[i+15:i]

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+15:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[i+15:i] = 0

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VPSRAW (EVEX versions, imm8)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

IF VL = 128

TMP\_DEST[127:0] ← ARITHMETIC\_RIGHT\_SHIFT\_WORDS\_128b(SRC1[127:0], imm8)

FI;

IF VL = 256

TMP\_DEST[255:0] ← ARITHMETIC\_RIGHT\_SHIFT\_WORDS\_256b(SRC1[255:0], imm8)

FI;

IF VL = 512

TMP\_DEST[255:0] ← ARITHMETIC\_RIGHT\_SHIFT\_WORDS\_256b(SRC1[255:0], imm8)

TMP\_DEST[511:256] ← ARITHMETIC\_RIGHT\_SHIFT\_WORDS\_256b(SRC1[511:256], imm8)

FI;

FOR j ← 0 TO KL-1

i ← j \* 16

IF k1[j] OR \*no writemask\*

THEN DEST[i+15:i] ← TMP\_DEST[i+15:i]

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+15:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[i+15:i] = 0

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VPSRAW (ymm, ymm, xmm/m128) - VEX**

DEST[255:0] ← ARITHMETIC\_RIGHT\_SHIFT\_WORDS\_256b(SRC1, SRC2)

DEST[MAX\_VL-1:256] ← 0

**VPSRAW (ymm, imm8) - VEX**

DEST[255:0] ← ARITHMETIC\_RIGHT\_SHIFT\_WORDS\_256b(SRC1, imm8)

DEST[MAX\_VL-1:256] ← 0

**VPSRAW (xmm, xmm, xmm/m128) - VEX**

DEST[127:0] ← ARITHMETIC\_RIGHT\_SHIFT\_WORDS(SRC1, SRC2)

DEST[MAX\_VL-1:128] ← 0

**VPSRAW (xmm, imm8) - VEX**

DEST[127:0] ← ARITHMETIC\_RIGHT\_SHIFT\_WORDS(SRC1, imm8)

DEST[MAX\_VL-1:128] ← 0

**PSRAW (xmm, xmm, xmm/m128)**

DEST[127:0] ← ARITHMETIC\_RIGHT\_SHIFT\_WORDS(DEST, SRC)

DEST[MAX\_VL-1:128] (Unmodified)

**PSRAW (xmm, imm8)**

DEST[127:0] ← ARITHMETIC\_RIGHT\_SHIFT\_WORDS(DEST, imm8)

DEST[MAX\_VL-1:128] (Unmodified)

**VPSRAD (EVEX versions, imm8)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\* THEN

IF (EVEX.b = 1) AND (SRC1 \*is memory\*)

THEN DEST[j+31:i] ← ARITHMETIC\_RIGHT\_SHIFT\_DWORDS1(SRC1[31:0], imm8)

ELSE DEST[j+31:i] ← ARITHMETIC\_RIGHT\_SHIFT\_DWORDS1(SRC1[j+31:i], imm8)

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[j+31:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[j+31:i] ← 0

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VPSRAD (EVEX versions, xmm/m128)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

IF VL = 128

TMP\_DEST[127:0] ← ARITHMETIC\_RIGHT\_SHIFT\_DWORDS\_128b(SRC1[127:0], SRC2)

FI;

IF VL = 256

TMP\_DEST[255:0] ← ARITHMETIC\_RIGHT\_SHIFT\_DWORDS\_256b(SRC1[255:0], SRC2)

FI;

IF VL = 512

TMP\_DEST[255:0] ← ARITHMETIC\_RIGHT\_SHIFT\_DWORDS\_256b(SRC1[255:0], SRC2)

TMP\_DEST[511:256] ← ARITHMETIC\_RIGHT\_SHIFT\_DWORDS\_256b(SRC1[511:256], SRC2)



```

FI;
FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[j+31:i] ← TMP_DEST[j+31:i]
  ELSE
    IF *merging-masking*           ; merging-masking
      THEN *DEST[j+31:i] remains unchanged*
    ELSE *zeroing-masking*       ; zeroing-masking
      DEST[j+31:i] ← 0
    FI
  FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VPSRAD (ymm, ymm, xmm/m128) - VEX**

```

DEST[255:0] ← ARITHMETIC_RIGHT_SHIFT_DWORDS_256b(SRC1, SRC2)
DEST[MAX_VL-1:256] ← 0

```

**VPSRAD (ymm, imm8) - VEX**

```

DEST[255:0] ← ARITHMETIC_RIGHT_SHIFT_DWORDS_256b(SRC1, imm8)
DEST[MAX_VL-1:256] ← 0

```

**VPSRAD (xmm, xmm, xmm/m128) - VEX**

```

DEST[127:0] ← ARITHMETIC_RIGHT_SHIFT_DWORDS(SRC1, SRC2)
DEST[MAX_VL-1:128] ← 0

```

**VPSRAD (xmm, imm8) - VEX**

```

DEST[127:0] ← ARITHMETIC_RIGHT_SHIFT_DWORDS(SRC1, imm8)
DEST[MAX_VL-1:128] ← 0

```

**PSRAD (xmm, xmm, xmm/m128)**

```

DEST[127:0] ← ARITHMETIC_RIGHT_SHIFT_DWORDS(DEST, SRC)
DEST[MAX_VL-1:128] (Unmodified)

```

**PSRAD (xmm, imm8)**

```

DEST[127:0] ← ARITHMETIC_RIGHT_SHIFT_DWORDS(DEST, imm8)
DEST[MAX_VL-1:128] (Unmodified)

```

**VPSRAQ (EVEX versions, imm8)**

(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) AND (SRC1 *is memory*)
      THEN DEST[j+63:i] ← ARITHMETIC_RIGHT_SHIFT_QWORDS1(SRC1[63:0], imm8)
    ELSE DEST[j+63:i] ← ARITHMETIC_RIGHT_SHIFT_QWORDS1(SRC1[j+63:i], imm8)
  FI;
  ELSE
    IF *merging-masking*           ; merging-masking
      THEN *DEST[j+63:i] remains unchanged*
    ELSE *zeroing-masking*       ; zeroing-masking
      DEST[j+63:i] ← 0
    FI
  FI;
ENDFOR

```

```

    FI;
  ENDFOR
  DEST[MAX_VL-1:VL] ← 0

```

### VPSRAQ (EVEX versions, xmm/m128)

(KL, VL) = (2, 128), (4, 256), (8, 512)

TMP\_DEST[VL-1:0] ← ARITHMETIC\_RIGHT\_SHIFT\_QWORDS(SRC1[VL-1:0], SRC2, VL)

```

FOR j ← 0 TO 7
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
  ELSE
    IF *merging-masking*           ; merging-masking
      THEN *DEST[i+63:i] remains unchanged*
    ELSE *zeroing-masking*        ; zeroing-masking
      DEST[i+63:i] ← 0
    FI
  FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

### Intel C/C++ Compiler Intrinsic Equivalents

```

VPSRAD __m512i __mm512_srai_epi32(__m512i a, unsigned int imm);
VPSRAD __m512i __mm512_mask_srai_epi32(__m512i s, __mmask16 k, __m512i a, unsigned int imm);
VPSRAD __m512i __mm512_maskz_srai_epi32(__mmask16 k, __m512i a, unsigned int imm);
VPSRAD __m256i __mm256_mask_srai_epi32(__m256i s, __mmask8 k, __m256i a, unsigned int imm);
VPSRAD __m256i __mm256_maskz_srai_epi32(__mmask8 k, __m256i a, unsigned int imm);
VPSRAD __m128i __mm_mask_srai_epi32(__m128i s, __mmask8 k, __m128i a, unsigned int imm);
VPSRAD __m128i __mm_maskz_srai_epi32(__mmask8 k, __m128i a, unsigned int imm);
VPSRAD __m512i __mm512_sra_epi32(__m512i a, __m128i cnt);
VPSRAD __m512i __mm512_mask_sra_epi32(__m512i s, __mmask16 k, __m512i a, __m128i cnt);
VPSRAD __m512i __mm512_maskz_sra_epi32(__mmask16 k, __m512i a, __m128i cnt);
VPSRAD __m256i __mm256_mask_sra_epi32(__m256i s, __mmask8 k, __m256i a, __m128i cnt);
VPSRAD __m256i __mm256_maskz_sra_epi32(__mmask8 k, __m256i a, __m128i cnt);
VPSRAD __m128i __mm_mask_sra_epi32(__m128i s, __mmask8 k, __m128i a, __m128i cnt);
VPSRAD __m128i __mm_maskz_sra_epi32(__mmask8 k, __m128i a, __m128i cnt);
VPSRAQ __m512i __mm512_srai_epi64(__m512i a, unsigned int imm);
VPSRAQ __m512i __mm512_mask_srai_epi64(__m512i s, __mmask8 k, __m512i a, unsigned int imm)
VPSRAQ __m512i __mm512_maskz_srai_epi64(__mmask8 k, __m512i a, unsigned int imm)
VPSRAQ __m256i __mm256_mask_srai_epi64(__m256i s, __mmask8 k, __m256i a, unsigned int imm);
VPSRAQ __m256i __mm256_maskz_srai_epi64(__mmask8 k, __m256i a, unsigned int imm);
VPSRAQ __m128i __mm_mask_srai_epi64(__m128i s, __mmask8 k, __m128i a, unsigned int imm);
VPSRAQ __m128i __mm_maskz_srai_epi64(__mmask8 k, __m128i a, unsigned int imm);
VPSRAQ __m512i __mm512_sra_epi64(__m512i a, __m128i cnt);
VPSRAQ __m512i __mm512_mask_sra_epi64(__m512i s, __mmask8 k, __m512i a, __m128i cnt)
VPSRAQ __m512i __mm512_maskz_sra_epi64(__mmask8 k, __m512i a, __m128i cnt)
VPSRAQ __m256i __mm256_mask_sra_epi64(__m256i s, __mmask8 k, __m256i a, __m128i cnt);
VPSRAQ __m256i __mm256_maskz_sra_epi64(__mmask8 k, __m256i a, __m128i cnt);
VPSRAQ __m128i __mm_mask_sra_epi64(__m128i s, __mmask8 k, __m128i a, __m128i cnt);
VPSRAQ __m128i __mm_maskz_sra_epi64(__mmask8 k, __m128i a, __m128i cnt);
VPSRAW __m512i __mm512_srai_epi16(__m512i a, unsigned int imm);
VPSRAW __m512i __mm512_mask_srai_epi16(__m512i s, __mmask32 k, __m512i a, unsigned int imm);

```

VPSRAW \_\_m512i \_\_mm512\_maskz\_srai\_epi16(\_\_mmask32 k, \_\_m512i a, unsigned int imm);  
 VPSRAW \_\_m256i \_\_mm256\_mask\_srai\_epi16(\_\_m256i s, \_\_mmask16 k, \_\_m256i a, unsigned int imm);  
 VPSRAW \_\_m256i \_\_mm256\_maskz\_srai\_epi16(\_\_mmask16 k, \_\_m256i a, unsigned int imm);  
 VPSRAW \_\_m128i \_\_mm\_mask\_srai\_epi16(\_\_m128i s, \_\_mmask8 k, \_\_m128i a, unsigned int imm);  
 VPSRAW \_\_m128i \_\_mm\_maskz\_srai\_epi16(\_\_mmask8 k, \_\_m128i a, unsigned int imm);  
 VPSRAW \_\_m512i \_\_mm512\_sra\_epi16(\_\_m512i a, \_\_m128i cnt);  
 VPSRAW \_\_m512i \_\_mm512\_mask\_sra\_epi16(\_\_m512i s, \_\_mmask16 k, \_\_m512i a, \_\_m128i cnt);  
 VPSRAW \_\_m512i \_\_mm512\_maskz\_sra\_epi16(\_\_mmask16 k, \_\_m512i a, \_\_m128i cnt);  
 VPSRAW \_\_m256i \_\_mm256\_mask\_sra\_epi16(\_\_m256i s, \_\_mmask8 k, \_\_m256i a, \_\_m128i cnt);  
 VPSRAW \_\_m256i \_\_mm256\_maskz\_sra\_epi16(\_\_mmask8 k, \_\_m256i a, \_\_m128i cnt);  
 VPSRAW \_\_m128i \_\_mm\_mask\_sra\_epi16(\_\_m128i s, \_\_mmask8 k, \_\_m128i a, \_\_m128i cnt);  
 VPSRAW \_\_m128i \_\_mm\_maskz\_sra\_epi16(\_\_mmask8 k, \_\_m128i a, \_\_m128i cnt);  
 PSRAW: \_\_m64 \_\_mm\_srai\_pi16 (\_\_m64 m, int count)  
 PSRAW: \_\_m64 \_\_mm\_sra\_pi16 (\_\_m64 m, \_\_m64 count)  
 (V)PSRAW: \_\_m128i \_\_mm\_srai\_epi16(\_\_m128i m, int count)  
 (V)PSRAW: \_\_m128i \_\_mm\_sra\_epi16(\_\_m128i m, \_\_m128i count)  
 VPSRAW: \_\_m256i \_\_mm256\_srai\_epi16 (\_\_m256i m, int count)  
 VPSRAW: \_\_m256i \_\_mm256\_sra\_epi16 (\_\_m256i m, \_\_m128i count)  
 PSRAD: \_\_m64 \_\_mm\_srai\_pi32 (\_\_m64 m, int count)  
 PSRAD: \_\_m64 \_\_mm\_sra\_pi32 (\_\_m64 m, \_\_m64 count)  
 (V)PSRAD: \_\_m128i \_\_mm\_srai\_epi32 (\_\_m128i m, int count)  
 (V)PSRAD: \_\_m128i \_\_mm\_sra\_epi32 (\_\_m128i m, \_\_m128i count)  
 VPSRAD: \_\_m256i \_\_mm256\_srai\_epi32 (\_\_m256i m, int count)  
 VPSRAD: \_\_m256i \_\_mm256\_sra\_epi32 (\_\_m256i m, \_\_m128i count)

### Flags Affected

None.

### Numeric Exceptions

None.

### Other Exceptions

VEX-encoded instructions:

Syntax with RM/RVM operand encoding, see Exceptions Type 4.

Syntax with MI/VMI operand encoding, see Exceptions Type 7.

EVEX-encoded VPSRAW, see Exceptions Type E4NF.nb.

EVEX-encoded VPSRAD/Q:

Syntax with M128 operand encoding, see Exceptions Type E4NF.nb.

Syntax with FVI operand encoding, see Exceptions Type E4.

## PSRLDQ—Shift Double Quadword Right Logical

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 73 /3 ib PSRLDQ <i>xmm1</i> , <i>imm8</i>	MI	V/V	SSE2	Shift <i>xmm1</i> right by <i>imm8</i> while shifting in 0s.
VEX.NDD.128.66.0F.WIG 73 /3 ib VPSRLDQ <i>xmm1</i> , <i>xmm2</i> , <i>imm8</i>	VMI	V/V	AVX	Shift <i>xmm2</i> right by <i>imm8</i> bytes while shifting in 0s.
VEX.NDD.256.66.0F.WIG 73 /3 ib VPSRLDQ <i>ymm1</i> , <i>ymm2</i> , <i>imm8</i>	VMI	V/V	AVX2	Shift <i>ymm1</i> right by <i>imm8</i> bytes while shifting in 0s.
EVEX.NDD.128.66.0F.WIG 73 /3 ib VPSRLDQ <i>xmm1</i> , <i>xmm2/m128</i> , <i>imm8</i>	FVM	V/V	AVX512VL AVX512BW	Shift <i>xmm2/m128</i> right by <i>imm8</i> bytes while shifting in 0s and store result in <i>xmm1</i> .
EVEX.NDD.256.66.0F.WIG 73 /3 ib VPSRLDQ <i>ymm1</i> , <i>ymm2/m256</i> , <i>imm8</i>	FVM	V/V	AVX512VL AVX512BW	Shift <i>ymm2/m256</i> right by <i>imm8</i> bytes while shifting in 0s and store result in <i>ymm1</i> .
EVEX.NDD.512.66.0F.WIG 73 /3 ib VPSRLDQ <i>zmm1</i> , <i>zmm2/m512</i> , <i>imm8</i>	FVM	V/V	AVX512BW	Shift <i>zmm2/m512</i> right by <i>imm8</i> bytes while shifting in 0s and store result in <i>zmm1</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MI	ModRM:r/m (r, w)	<i>imm8</i>	NA	NA
VMI	VEX.vvvv (w)	ModRM:r/m (r)	<i>imm8</i>	NA
FVM	EVEX.vvvv (w)	ModRM:r/m (R)	<i>Imm8</i>	NA

### Description

Shifts the destination operand (first operand) to the right by the number of bytes specified in the count operand (second operand). The empty high-order bytes are cleared (set to all 0s). If the value specified by the count operand is greater than 15, the destination operand is set to all 0s. The count operand is an 8-bit immediate.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The source and destination operands are the same. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The source and destination operands are XMM registers. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The source operand is a YMM register. The destination operand is a YMM register. The count operand applies to both the low and high 128-bit lanes.

VEX.256 encoded version: The source operand is YMM register. The destination operand is an YMM register. Bits (MAX\_VL-1:256) of the corresponding ZMM register are zeroed. The count operand applies to both the low and high 128-bit lanes.

EVEX encoded versions: 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. The count operand applies to each 128-bit lanes.

Note: VEX.vvvv/EVEX.vvvv encodes the destination register.

**Operation****VPSRLDQ (EVEX.512 encoded version)**

```
TEMP ← COUNT
IF (TEMP > 15) THEN TEMP ← 16; FI
DEST[127:0] ← SRC[127:0] >> (TEMP * 8)
DEST[255:128] ← SRC[255:128] >> (TEMP * 8)
DEST[383:256] ← SRC[383:256] >> (TEMP * 8)
DEST[511:384] ← SRC[511:384] >> (TEMP * 8)
DEST[MAX_VL-1:512] ← 0;
```

**VPSRLDQ (VEX.256 and EVEX.256 encoded version)**

```
TEMP ← COUNT
IF (TEMP > 15) THEN TEMP ← 16; FI
DEST[127:0] ← SRC[127:0] >> (TEMP * 8)
DEST[255:128] ← SRC[255:128] >> (TEMP * 8)
DEST[MAX_VL-1:256] ← 0;
```

**VPSRLDQ (VEX.128 and EVEX.128 encoded version)**

```
TEMP ← COUNT
IF (TEMP > 15) THEN TEMP ← 16; FI
DEST ← SRC >> (TEMP * 8)
DEST[MAX_VL-1:128] ← 0;
```

**PSRLDQ(128-bit Legacy SSE version)**

```
TEMP ← COUNT
IF (TEMP > 15) THEN TEMP ← 16; FI
DEST ← DEST >> (TEMP * 8)
DEST[MAX_VL-1:128] (Unmodified)
```

**Intel C/C++ Compiler Intrinsic Equivalents**

```
(V)PSRLDQ __m128i _mm_srli_si128 ( __m128i a, int imm)
VPSRLDQ __m256i _mm256_bsrl_i_epi128 ( __m256i, const int)
VPSRLDQ __m512i _mm512_bsrl_i_epi128 ( __m512i, int)
```

**Flags Affected**

None.

**Numeric Exceptions**

None.

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 7.  
EVEX-encoded instruction, see Exceptions Type E4NF.nb.

## PSRLW/PSRLD/PSRLQ—Shift Packed Data Right Logical

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F D1 /r <sup>1</sup> PSRLW <i>mm</i> , <i>mm/m64</i>	RM	V/V	MMX	Shift words in <i>mm</i> right by amount specified in <i>mm/m64</i> while shifting in 0s.
66 0F D1 /r PSRLW <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE2	Shift words in <i>xmm1</i> right by amount specified in <i>xmm2/m128</i> while shifting in 0s.
0F 71 /2 ib <sup>1</sup> PSRLW <i>mm</i> , <i>imm8</i>	MI	V/V	MMX	Shift words in <i>mm</i> right by <i>imm8</i> while shifting in 0s.
66 0F 71 /2 ib PSRLW <i>xmm1</i> , <i>imm8</i>	MI	V/V	SSE2	Shift words in <i>xmm1</i> right by <i>imm8</i> while shifting in 0s.
0F D2 /r <sup>1</sup> PSRLD <i>mm</i> , <i>mm/m64</i>	RM	V/V	MMX	Shift doublewords in <i>mm</i> right by amount specified in <i>mm/m64</i> while shifting in 0s.
66 0F D2 /r PSRLD <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE2	Shift doublewords in <i>xmm1</i> right by amount specified in <i>xmm2/m128</i> while shifting in 0s.
0F 72 /2 ib <sup>1</sup> PSRLD <i>mm</i> , <i>imm8</i>	MI	V/V	MMX	Shift doublewords in <i>mm</i> right by <i>imm8</i> while shifting in 0s.
66 0F 72 /2 ib PSRLD <i>xmm1</i> , <i>imm8</i>	MI	V/V	SSE2	Shift doublewords in <i>xmm1</i> right by <i>imm8</i> while shifting in 0s.
0F D3 /r <sup>1</sup> PSRLQ <i>mm</i> , <i>mm/m64</i>	RM	V/V	MMX	Shift <i>mm</i> right by amount specified in <i>mm/m64</i> while shifting in 0s.
66 0F D3 /r PSRLQ <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE2	Shift quadwords in <i>xmm1</i> right by amount specified in <i>xmm2/m128</i> while shifting in 0s.
0F 73 /2 ib <sup>1</sup> PSRLQ <i>mm</i> , <i>imm8</i>	MI	V/V	MMX	Shift <i>mm</i> right by <i>imm8</i> while shifting in 0s.
66 0F 73 /2 ib PSRLQ <i>xmm1</i> , <i>imm8</i>	MI	V/V	SSE2	Shift quadwords in <i>xmm1</i> right by <i>imm8</i> while shifting in 0s.
VEX.NDS.128.66.0F.WIG D1 /r VPSRLW <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Shift words in <i>xmm2</i> right by amount specified in <i>xmm3/m128</i> while shifting in 0s.
VEX.NDD.128.66.0F.WIG 71 /2 ib VPSRLW <i>xmm1</i> , <i>xmm2</i> , <i>imm8</i>	VMI	V/V	AVX	Shift words in <i>xmm2</i> right by <i>imm8</i> while shifting in 0s.
VEX.NDS.128.66.0F.WIG D2 /r VPSRLD <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Shift doublewords in <i>xmm2</i> right by amount specified in <i>xmm3/m128</i> while shifting in 0s.
VEX.NDD.128.66.0F.WIG 72 /2 ib VPSRLD <i>xmm1</i> , <i>xmm2</i> , <i>imm8</i>	VMI	V/V	AVX	Shift doublewords in <i>xmm2</i> right by <i>imm8</i> while shifting in 0s.
VEX.NDS.128.66.0F.WIG D3 /r VPSRLQ <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Shift quadwords in <i>xmm2</i> right by amount specified in <i>xmm3/m128</i> while shifting in 0s.
VEX.NDD.128.66.0F.WIG 73 /2 ib VPSRLQ <i>xmm1</i> , <i>xmm2</i> , <i>imm8</i>	VMI	V/V	AVX	Shift quadwords in <i>xmm2</i> right by <i>imm8</i> while shifting in 0s.
VEX.NDS.256.66.0F.WIG D1 /r VPSRLW <i>ymm1</i> , <i>ymm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX2	Shift words in <i>ymm2</i> right by amount specified in <i>xmm3/m128</i> while shifting in 0s.
VEX.NDD.256.66.0F.WIG 71 /2 ib VPSRLW <i>ymm1</i> , <i>ymm2</i> , <i>imm8</i>	VMI	V/V	AVX2	Shift words in <i>ymm2</i> right by <i>imm8</i> while shifting in 0s.

VEEX.NDS.256.66.0F.WIG D2 /r VPSRLD <i>ymm1, ymm2, xmm3/m128</i>	RVM V/V	AVX2	Shift doublewords in <i>ymm2</i> right by amount specified in <i>xmm3/m128</i> while shifting in 0s.
VEEX.NDD.256.66.0F.WIG 72 /2 ib VPSRLD <i>ymm1, ymm2, imm8</i>	VMI V/V	AVX2	Shift doublewords in <i>ymm2</i> right by <i>imm8</i> while shifting in 0s.
VEEX.NDS.256.66.0F.WIG D3 /r VPSRLQ <i>ymm1, ymm2, xmm3/m128</i>	RVM V/V	AVX2	Shift quadwords in <i>ymm2</i> right by amount specified in <i>xmm3/m128</i> while shifting in 0s.
VEEX.NDD.256.66.0F.WIG 73 /2 ib VPSRLQ <i>ymm1, ymm2, imm8</i>	VMI V/V	AVX2	Shift quadwords in <i>ymm2</i> right by <i>imm8</i> while shifting in 0s.
EVEX.NDS.128.66.0F.WIG D1 /r VPSRLW <i>xmm1 {k1}{z}, xmm2, xmm3/m128</i>	M128 V/V	AVX512VL AVX512BW	Shift words in <i>xmm2</i> right by amount specified in <i>xmm3/m128</i> while shifting in 0s using writemask <i>k1</i> .
EVEX.NDS.256.66.0F.WIG D1 /r VPSRLW <i>ymm1 {k1}{z}, ymm2, xmm3/m128</i>	M128 V/V	AVX512VL AVX512BW	Shift words in <i>ymm2</i> right by amount specified in <i>xmm3/m128</i> while shifting in 0s using writemask <i>k1</i> .
EVEX.NDS.512.66.0F.WIG D1 /r VPSRLW <i>zmm1 {k1}{z}, zmm2, xmm3/m128</i>	M128 V/V	AVX512BW	Shift words in <i>zmm2</i> right by amount specified in <i>xmm3/m128</i> while shifting in 0s using writemask <i>k1</i> .
EVEX.NDD.128.66.0F.WIG 71 /2 ib VPSRLW <i>xmm1 {k1}{z}, xmm2/m128, imm8</i>	FVM V/V	AVX512VL AVX512BW	Shift words in <i>xmm2/m128</i> right by <i>imm8</i> while shifting in 0s using writemask <i>k1</i> .
EVEX.NDD.256.66.0F.WIG 71 /2 ib VPSRLW <i>ymm1 {k1}{z}, ymm2/m256, imm8</i>	FVM V/V	AVX512VL AVX512BW	Shift words in <i>ymm2/m256</i> right by <i>imm8</i> while shifting in 0s using writemask <i>k1</i> .
EVEX.NDD.512.66.0F.WIG 71 /2 ib VPSRLW <i>zmm1 {k1}{z}, zmm2/m512, imm8</i>	FVM V/V	AVX512BW	Shift words in <i>zmm2/m512</i> right by <i>imm8</i> while shifting in 0s using writemask <i>k1</i> .
EVEX.NDS.128.66.0F.WO D2 /r VPSRLD <i>xmm1 {k1}{z}, xmm2, xmm3/m128</i>	M128 V/V	AVX512VL AVX512F	Shift doublewords in <i>xmm2</i> right by amount specified in <i>xmm3/m128</i> while shifting in 0s using writemask <i>k1</i> .
EVEX.NDS.256.66.0F.WO D2 /r VPSRLD <i>ymm1 {k1}{z}, ymm2, xmm3/m128</i>	M128 V/V	AVX512VL AVX512F	Shift doublewords in <i>ymm2</i> right by amount specified in <i>xmm3/m128</i> while shifting in 0s using writemask <i>k1</i> .
EVEX.NDS.512.66.0F.WO D2 /r VPSRLD <i>zmm1 {k1}{z}, zmm2, xmm3/m128</i>	M128 V/V	AVX512F	Shift doublewords in <i>zmm2</i> right by amount specified in <i>xmm3/m128</i> while shifting in 0s using writemask <i>k1</i> .
EVEX.NDD.128.66.0F.WO 72 /2 ib VPSRLD <i>xmm1 {k1}{z}, xmm2/m128/m32bcst, imm8</i>	FV V/V	AVX512VL AVX512F	Shift doublewords in <i>xmm2/m128/m32bcst</i> right by <i>imm8</i> while shifting in 0s using writemask <i>k1</i> .
EVEX.NDD.256.66.0F.WO 72 /2 ib VPSRLD <i>ymm1 {k1}{z}, ymm2/m256/m32bcst, imm8</i>	FV V/V	AVX512VL AVX512F	Shift doublewords in <i>ymm2/m256/m32bcst</i> right by <i>imm8</i> while shifting in 0s using writemask <i>k1</i> .
EVEX.NDD.512.66.0F.WO 72 /2 ib VPSRLD <i>zmm1 {k1}{z}, zmm2/m512/m32bcst, imm8</i>	FVI V/V	AVX512F	Shift doublewords in <i>zmm2/m512/m32bcst</i> right by <i>imm8</i> while shifting in 0s using writemask <i>k1</i> .

EVEX.NDS.128.66.0F.W1 D3 /r VPSRLQ xmm1 {k1}{z}, xmm2, xmm3/m128	M128 V/V	AVX512VL AVX512F	Shift quadwords in xmm2 right by amount specified in xmm3/m128 while shifting in 0s using writemask k1.
EVEX.NDS.256.66.0F.W1 D3 /r VPSRLQ ymm1 {k1}{z}, ymm2, xmm3/m128	M128 V/V	AVX512VL AVX512F	Shift quadwords in ymm2 right by amount specified in xmm3/m128 while shifting in 0s using writemask k1.
EVEX.NDS.512.66.0F.W1 D3 /r VPSRLQ zmm1 {k1}{z}, zmm2, xmm3/m128	M128 V/V	AVX512F	Shift quadwords in zmm2 right by amount specified in xmm3/m128 while shifting in 0s using writemask k1.
EVEX.NDD.128.66.0F.W1 73 /2 ib VPSRLQ xmm1 {k1}{z}, xmm2/m128/m64bcst, imm8	FV V/V	AVX512VL AVX512F	Shift quadwords in xmm2/m128/m64bcst right by imm8 while shifting in 0s using writemask k1.
EVEX.NDD.256.66.0F.W1 73 /2 ib VPSRLQ ymm1 {k1}{z}, ymm2/m256/m64bcst, imm8	FV V/V	AVX512VL AVX512F	Shift quadwords in ymm2/m256/m64bcst right by imm8 while shifting in 0s using writemask k1.
EVEX.NDD.512.66.0F.W1 73 /2 ib VPSRLQ zmm1 {k1}{z}, zmm2/m512/m64bcst, imm8	FVI V/V	AVX512F	Shift quadwords in zmm2/m512/m64bcst right by imm8 while shifting in 0s using writemask k1.

**NOTES:**

1. See note in Section 2.4, “AVX and SSE Instruction Exception Specification” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
MI	ModRM:r/m (r, w)	imm8	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
VMI	VEX.vvvv (w)	ModRM:r/m (r)	imm8	NA
FVM	EVEX.vvvv (w)	ModRM:r/m (R)	Imm8	NA
FVI	EVEX.vvvv (w)	ModRM:r/m (R)	Imm8	NA
M128	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

**Description**

Shifts the bits in the individual data elements (words, doublewords, or quadword) in the destination operand (first operand) to the right by the number of bits specified in the count operand (second operand). As the bits in the data elements are shifted right, the empty high-order bits are cleared (set to 0). If the value specified by the count operand is greater than 15 (for words), 31 (for doublewords), or 63 (for a quadword), then the destination operand is set to all 0s. Figure 4-15 gives an example of shifting words in a 64-bit operand.

Note that only the low 64-bits of a 128-bit count operand are checked to compute the count.



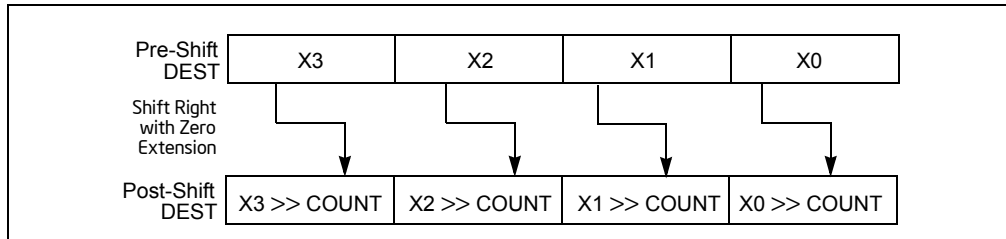


Figure 4-15. PSRLW, PSRLD, and PSRLQ Instruction Operation Using 64-bit Operand

The (V)PSRLW instruction shifts each of the words in the destination operand to the right by the number of bits specified in the count operand; the (V)PSRLD instruction shifts each of the doublewords in the destination operand; and the PSRLQ instruction shifts the quadword (or quadwords) in the destination operand.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE instruction 64-bit operand: The destination operand is an MMX technology register; the count operand can be either an MMX technology register or a 64-bit memory location.

128-bit Legacy SSE version: The destination operand is an XMM register; the count operand can be either an XMM register or a 128-bit memory location, or an 8-bit immediate. If the count operand is a memory address, 128 bits are loaded but the upper 64 bits are ignored. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The destination operand is an XMM register; the count operand can be either an XMM register or a 128-bit memory location, or an 8-bit immediate. If the count operand is a memory address, 128 bits are loaded but the upper 64 bits are ignored. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The destination operand is a YMM register. The source operand is a YMM register or a memory location. The count operand can come either from an XMM register or a memory location or an 8-bit immediate. Bits (MAX\_VL-1:256) of the corresponding ZMM register are zeroed.

EVEX encoded versions: The destination operand is a ZMM register updated according to the writemask. The count operand is either an 8-bit immediate (the immediate count version) or an 8-bit value from an XMM register or a memory location (the variable count version). For the immediate count version, the source operand (the second operand) can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32/64-bit memory location. For the variable count version, the first source operand (the second operand) is a ZMM register, the second source operand (the third operand, 8-bit variable count) can be an XMM register or a memory location.

Note: In VEX/EVEX encoded versions of shifts with an immediate count, vvvv of VEX/EVEX encode the destination register, and VEX.B/EVEX.B + ModRM.r/m encodes the source register.

Note: For shifts with an immediate count (VEX.128.66.0F 71-73 /2, or EVEX.128.66.0F 71-73 /2), VEX.vvvv/EVEX.vvvv encodes the destination register.

## Operation

### PSRLW (with 64-bit operand)

```
IF (COUNT > 15)
  THEN
    DEST[64:0] ← 0000000000000000H
  ELSE
    DEST[15:0] ← ZeroExtend(DEST[15:0] >> COUNT);
    (* Repeat shift operation for 2nd and 3rd words *)
    DEST[63:48] ← ZeroExtend(DEST[63:48] >> COUNT);
  FI;
```

### PSRLD (with 64-bit operand)

```
IF (COUNT > 31)
  THEN
```

```

    DEST[64:0] ← 0000000000000000H
ELSE
    DEST[31:0] ← ZeroExtend(DEST[31:0] >> COUNT);
    DEST[63:32] ← ZeroExtend(DEST[63:32] >> COUNT);
FI;

```

**PSRLQ (with 64-bit operand)**

```

IF (COUNT > 63)
THEN
    DEST[64:0] ← 0000000000000000H
ELSE
    DEST ← ZeroExtend(DEST >> COUNT);
FI;

```

```

LOGICAL_RIGHT_SHIFT_DWORDS1(SRC, COUNT_SRC)
COUNT ← COUNT_SRC[63:0];
IF (COUNT > 31)
THEN
    DEST[31:0] ← 0
ELSE
    DEST[31:0] ← ZeroExtend(SRC[31:0] >> COUNT);
FI;

```

```

LOGICAL_RIGHT_SHIFT_QWORDS1(SRC, COUNT_SRC)
COUNT ← COUNT_SRC[63:0];
IF (COUNT > 63)
THEN
    DEST[63:0] ← 0
ELSE
    DEST[63:0] ← ZeroExtend(SRC[63:0] >> COUNT);
FI;

```

```

LOGICAL_RIGHT_SHIFT_WORDS_256b(SRC, COUNT_SRC)
COUNT ← COUNT_SRC[63:0];
IF (COUNT > 15)
THEN
    DEST[255:0] ← 0
ELSE
    DEST[15:0] ← ZeroExtend(SRC[15:0] >> COUNT);
    (* Repeat shift operation for 2nd through 15th words *)
    DEST[255:240] ← ZeroExtend(SRC[255:240] >> COUNT);
FI;

```

```

LOGICAL_RIGHT_SHIFT_WORDS(SRC, COUNT_SRC)
COUNT ← COUNT_SRC[63:0];
IF (COUNT > 15)
THEN
    DEST[127:0] ← 00000000000000000000000000000000H
ELSE
    DEST[15:0] ← ZeroExtend(SRC[15:0] >> COUNT);
    (* Repeat shift operation for 2nd through 7th words *)
    DEST[127:112] ← ZeroExtend(SRC[127:112] >> COUNT);
FI;

```

```

LOGICAL_RIGHT_SHIFT_DWORDS_256b(SRC, COUNT_SRC)
COUNT ← COUNT_SRC[63:0];

```

```

IF (COUNT > 31)
THEN
    DEST[255:0] ← 0
ELSE
    DEST[31:0] ← ZeroExtend(SRC[31:0] >> COUNT);
    (* Repeat shift operation for 2nd through 3rd words *)
    DEST[255:224] ← ZeroExtend(SRC[255:224] >> COUNT);
FI;

LOGICAL_RIGHT_SHIFT_DWORDS(SRC, COUNT_SRC)
COUNT ← COUNT_SRC[63:0];
IF (COUNT > 31)
THEN
    DEST[127:0] ← 00000000000000000000000000000000H
ELSE
    DEST[31:0] ← ZeroExtend(SRC[31:0] >> COUNT);
    (* Repeat shift operation for 2nd through 3rd words *)
    DEST[127:96] ← ZeroExtend(SRC[127:96] >> COUNT);
FI;

LOGICAL_RIGHT_SHIFT_QWORDS_256b(SRC, COUNT_SRC)
COUNT ← COUNT_SRC[63:0];
IF (COUNT > 63)
THEN
    DEST[255:0] ← 0
ELSE
    DEST[63:0] ← ZeroExtend(SRC[63:0] >> COUNT);
    DEST[127:64] ← ZeroExtend(SRC[127:64] >> COUNT);
    DEST[191:128] ← ZeroExtend(SRC[191:128] >> COUNT);
    DEST[255:192] ← ZeroExtend(SRC[255:192] >> COUNT);
FI;

LOGICAL_RIGHT_SHIFT_QWORDS(SRC, COUNT_SRC)
COUNT ← COUNT_SRC[63:0];
IF (COUNT > 63)
THEN
    DEST[127:0] ← 00000000000000000000000000000000H
ELSE
    DEST[63:0] ← ZeroExtend(SRC[63:0] >> COUNT);
    DEST[127:64] ← ZeroExtend(SRC[127:64] >> COUNT);
FI;

```

**VPSRLW (EVEX versions, xmm/m128)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

IF VL = 128

TMP\_DEST[127:0] ← LOGICAL\_RIGHT\_SHIFT\_WORDS\_128b(SRC1[127:0], SRC2)

FI;

IF VL = 256

TMP\_DEST[255:0] ← LOGICAL\_RIGHT\_SHIFT\_WORDS\_256b(SRC1[255:0], SRC2)

FI;

IF VL = 512

TMP\_DEST[255:0] ← LOGICAL\_RIGHT\_SHIFT\_WORDS\_256b(SRC1[255:0], SRC2)

TMP\_DEST[511:256] ← LOGICAL\_RIGHT\_SHIFT\_WORDS\_256b(SRC1[511:256], SRC2)

FI;

FOR j ← 0 TO KL-1

i ← j \* 16

IF k1[j] OR \*no writemask\*

THEN DEST[i+15:i] ← TMP\_DEST[i+15:i]

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+15:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[i+15:i] = 0

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VPSRLW (EVEX versions, imm8)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

IF VL = 128

TMP\_DEST[127:0] ← LOGICAL\_RIGHT\_SHIFT\_WORDS\_128b(SRC1[127:0], imm8)

FI;

IF VL = 256

TMP\_DEST[255:0] ← LOGICAL\_RIGHT\_SHIFT\_WORDS\_256b(SRC1[255:0], imm8)

FI;

IF VL = 512

TMP\_DEST[255:0] ← LOGICAL\_RIGHT\_SHIFT\_WORDS\_256b(SRC1[255:0], imm8)

TMP\_DEST[511:256] ← LOGICAL\_RIGHT\_SHIFT\_WORDS\_256b(SRC1[511:256], imm8)

FI;

FOR j ← 0 TO KL-1

i ← j \* 16

IF k1[j] OR \*no writemask\*

THEN DEST[i+15:i] ← TMP\_DEST[i+15:i]

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+15:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[i+15:i] = 0

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VPSRLW (ymm, ymm, xmm/m128) - VEX.256 encoding**

DEST[255:0] ← LOGICAL\_RIGHT\_SHIFT\_WORDS\_256b(SRC1, SRC2)

DEST[MAX\_VL-1:256] ← 0;

**VPSRLW (ymm, imm8) - VEX.256 encoding**

DEST[255:0] ← LOGICAL\_RIGHT\_SHIFT\_WORDS\_256b(SRC1, imm8)

DEST[MAX\_VL-1:256] ← 0;

**VPSRLW (xmm, xmm, xmm/m128) - VEX.128 encoding**

DEST[127:0] ← LOGICAL\_RIGHT\_SHIFT\_WORDS(SRC1, SRC2)

DEST[MAX\_VL-1:128] ← 0

**VPSRLW (xmm, imm8) - VEX.128 encoding**

DEST[127:0] ← LOGICAL\_RIGHT\_SHIFT\_WORDS(SRC1, imm8)

DEST[MAX\_VL-1:128] ← 0

**PSRLW (xmm, xmm, xmm/m128)**

DEST[127:0] ← LOGICAL\_RIGHT\_SHIFT\_WORDS(DEST, SRC)

DEST[MAX\_VL-1:128] (Unmodified)

**PSRLW (xmm, imm8)**

DEST[127:0] ← LOGICAL\_RIGHT\_SHIFT\_WORDS(DEST, imm8)

DEST[MAX\_VL-1:128] (Unmodified)

**VPSRLD (EVEX versions, xmm/m128)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

IF VL = 128

TMP\_DEST[127:0] ← LOGICAL\_RIGHT\_SHIFT\_DWORDS\_128b(SRC1[127:0], SRC2)

FI;

IF VL = 256

TMP\_DEST[255:0] ← LOGICAL\_RIGHT\_SHIFT\_DWORDS\_256b(SRC1[255:0], SRC2)

FI;

IF VL = 512

TMP\_DEST[255:0] ← LOGICAL\_RIGHT\_SHIFT\_DWORDS\_256b(SRC1[255:0], SRC2)

TMP\_DEST[511:256] ← LOGICAL\_RIGHT\_SHIFT\_DWORDS\_256b(SRC1[511:256], SRC2)

FI;

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\*

THEN DEST[i+31:i] ← TMP\_DEST[i+31:i]

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+31:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VPSRLD (EVEX versions, imm8)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask* THEN
    IF (EVEX.b = 1) AND (SRC1 *is memory*)
      THEN DEST[i+31:i] ← LOGICAL_RIGHT_SHIFT_DWORDS1(SRC1[31:0], imm8)
      ELSE DEST[i+31:i] ← LOGICAL_RIGHT_SHIFT_DWORDS1(SRC1[i+31:i], imm8)
    FI;
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+31:i] remains unchanged*
      ELSE *zeroing-masking* ; zeroing-masking
        DEST[i+31:i] ← 0
    FI
  FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VPSRLD (ymm, ymm, xmm/m128) - VEX.256 encoding**

```

DEST[255:0] ← LOGICAL_RIGHT_SHIFT_DWORDS_256b(SRC1, SRC2)
DEST[MAX_VL-1:256] ← 0;

```

**VPSRLD (ymm, imm8) - VEX.256 encoding**

```

DEST[255:0] ← LOGICAL_RIGHT_SHIFT_DWORDS_256b(SRC1, imm8)
DEST[MAX_VL-1:256] ← 0;

```

**VPSRLD (xmm, xmm, xmm/m128) - VEX.128 encoding**

```

DEST[127:0] ← LOGICAL_RIGHT_SHIFT_DWORDS(SRC1, SRC2)
DEST[MAX_VL-1:128] ← 0

```

**VPSRLD (xmm, imm8) - VEX.128 encoding**

```

DEST[127:0] ← LOGICAL_RIGHT_SHIFT_DWORDS(SRC1, imm8)
DEST[MAX_VL-1:128] ← 0

```

**PSRLD (xmm, xmm, xmm/m128)**

```

DEST[127:0] ← LOGICAL_RIGHT_SHIFT_DWORDS(DEST, SRC)
DEST[MAX_VL-1:128] (Unmodified)

```

**PSRLD (xmm, imm8)**

```

DEST[127:0] ← LOGICAL_RIGHT_SHIFT_DWORDS(DEST, imm8)
DEST[MAX_VL-1:128] (Unmodified)

```

**VPSRLQ (EVEX versions, xmm/m128)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

TMP\_DEST[255:0] ← LOGICAL\_RIGHT\_SHIFT\_QWORDS\_256b(SRC1[255:0], SRC2)

TMP\_DEST[511:256] ← LOGICAL\_RIGHT\_SHIFT\_QWORDS\_256b(SRC1[511:256], SRC2)

IF VL = 128

TMP\_DEST[127:0] ← LOGICAL\_RIGHT\_SHIFT\_QWORDS\_128b(SRC1[127:0], SRC2)

FI;

IF VL = 256

TMP\_DEST[255:0] ← LOGICAL\_RIGHT\_SHIFT\_QWORDS\_256b(SRC1[255:0], SRC2)

FI;

IF VL = 512

TMP\_DEST[255:0] ← LOGICAL\_RIGHT\_SHIFT\_QWORDS\_256b(SRC1[255:0], SRC2)

TMP\_DEST[511:256] ← LOGICAL\_RIGHT\_SHIFT\_QWORDS\_256b(SRC1[511:256], SRC2)

FI;

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN DEST[j+63:i] ← TMP\_DEST[j+63:i]

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[j+63:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[j+63:i] ← 0

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VPSRLQ (EVEX versions, imm8)**

(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) AND (SRC1 \*is memory\*)

THEN DEST[j+63:i] ← LOGICAL\_RIGHT\_SHIFT\_QWORDS1(SRC1[63:0], imm8)

ELSE DEST[j+63:i] ← LOGICAL\_RIGHT\_SHIFT\_QWORDS1(SRC1[j+63:i], imm8)

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[j+63:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[j+63:i] ← 0

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VPSRLQ (ymm, ymm, xmm/m128) - VEX.256 encoding**

DEST[255:0] ← LOGICAL\_RIGHT\_SHIFT\_QWORDS\_256b(SRC1, SRC2)

DEST[MAX\_VL-1:256] ← 0;

**VPSRLQ (ymm, imm8) - VEX.256 encoding**

DEST[255:0] ← LOGICAL\_RIGHT\_SHIFT\_QWORDS\_256b(SRC1, imm8)

DEST[MAX\_VL-1:256] ← 0;

**VPSRLQ (xmm, xmm, xmm/m128) - VEX.128 encoding**

DEST[127:0] ← LOGICAL\_RIGHT\_SHIFT\_QWORDS(SRC1, SRC2)

DEST[MAX\_VL-1:128] ← 0

**VPSRLQ (xmm, imm8) - VEX.128 encoding**

DEST[127:0] ← LOGICAL\_RIGHT\_SHIFT\_QWORDS(SRC1, imm8)

DEST[MAX\_VL-1:128] ← 0

**PSRLQ (xmm, xmm, xmm/m128)**

DEST[127:0] ← LOGICAL\_RIGHT\_SHIFT\_QWORDS(DEST, SRC)

DEST[MAX\_VL-1:128] (Unmodified)

**PSRLQ (xmm, imm8)**

DEST[127:0] ← LOGICAL\_RIGHT\_SHIFT\_QWORDS(DEST, imm8)

DEST[MAX\_VL-1:128] (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalents**

VPSRLD \_\_m512i \_mm512\_srl\_epi32(\_\_m512i a, unsigned int imm);

VPSRLD \_\_m512i \_mm512\_mask\_srl\_epi32(\_\_m512i s, \_\_mmask16 k, \_\_m512i a, unsigned int imm);

VPSRLD \_\_m512i \_mm512\_maskz\_srl\_epi32(\_\_mmask16 k, \_\_m512i a, unsigned int imm);

VPSRLD \_\_m256i \_mm256\_mask\_srl\_epi32(\_\_m256i s, \_\_mmask8 k, \_\_m256i a, unsigned int imm);

VPSRLD \_\_m256i \_mm256\_maskz\_srl\_epi32(\_\_mmask8 k, \_\_m256i a, unsigned int imm);

VPSRLD \_\_m128i \_mm\_mask\_srl\_epi32(\_\_m128i s, \_\_mmask8 k, \_\_m128i a, unsigned int imm);

VPSRLD \_\_m128i \_mm\_maskz\_srl\_epi32(\_\_mmask8 k, \_\_m128i a, unsigned int imm);

VPSRLD \_\_m512i \_mm512\_srl\_epi32(\_\_m512i a, \_\_m128i cnt);

VPSRLD \_\_m512i \_mm512\_mask\_srl\_epi32(\_\_m512i s, \_\_mmask16 k, \_\_m512i a, \_\_m128i cnt);

VPSRLD \_\_m512i \_mm512\_maskz\_srl\_epi32(\_\_mmask16 k, \_\_m512i a, \_\_m128i cnt);

VPSRLD \_\_m256i \_mm256\_mask\_srl\_epi32(\_\_m256i s, \_\_mmask8 k, \_\_m256i a, \_\_m128i cnt);

VPSRLD \_\_m256i \_mm256\_maskz\_srl\_epi32(\_\_mmask8 k, \_\_m256i a, \_\_m128i cnt);

VPSRLD \_\_m128i \_mm\_mask\_srl\_epi32(\_\_m128i s, \_\_mmask8 k, \_\_m128i a, \_\_m128i cnt);

VPSRLD \_\_m128i \_mm\_maskz\_srl\_epi32(\_\_mmask8 k, \_\_m128i a, \_\_m128i cnt);

VPSRLQ \_\_m512i \_mm512\_srl\_epi64(\_\_m512i a, unsigned int imm);

VPSRLQ \_\_m512i \_mm512\_mask\_srl\_epi64(\_\_m512i s, \_\_mmask8 k, \_\_m512i a, unsigned int imm);

VPSRLQ \_\_m512i \_mm512\_maskz\_srl\_epi64(\_\_mmask8 k, \_\_m512i a, unsigned int imm);

VPSRLQ \_\_m256i \_mm256\_mask\_srl\_epi64(\_\_m256i s, \_\_mmask8 k, \_\_m256i a, unsigned int imm);

VPSRLQ \_\_m256i \_mm256\_maskz\_srl\_epi64(\_\_mmask8 k, \_\_m256i a, unsigned int imm);

VPSRLQ \_\_m128i \_mm\_mask\_srl\_epi64(\_\_m128i s, \_\_mmask8 k, \_\_m128i a, unsigned int imm);

VPSRLQ \_\_m128i \_mm\_maskz\_srl\_epi64(\_\_mmask8 k, \_\_m128i a, unsigned int imm);

VPSRLQ \_\_m512i \_mm512\_srl\_epi64(\_\_m512i a, \_\_m128i cnt);

VPSRLQ \_\_m512i \_mm512\_mask\_srl\_epi64(\_\_m512i s, \_\_mmask8 k, \_\_m512i a, \_\_m128i cnt);

VPSRLQ \_\_m512i \_mm512\_maskz\_srl\_epi64(\_\_mmask8 k, \_\_m512i a, \_\_m128i cnt);

VPSRLQ \_\_m256i \_mm256\_mask\_srl\_epi64(\_\_m256i s, \_\_mmask8 k, \_\_m256i a, \_\_m128i cnt);

VPSRLQ \_\_m256i \_mm256\_maskz\_srl\_epi64(\_\_mmask8 k, \_\_m256i a, \_\_m128i cnt);

VPSRLQ \_\_m128i \_mm\_mask\_srl\_epi64(\_\_m128i s, \_\_mmask8 k, \_\_m128i a, \_\_m128i cnt);

VPSRLQ \_\_m128i \_mm\_maskz\_srl\_epi64(\_\_mmask8 k, \_\_m128i a, \_\_m128i cnt);

VPSRLW \_\_m512i \_mm512\_srl\_epi16(\_\_m512i a, unsigned int imm);

VPSRLW \_\_m512i \_mm512\_mask\_srl\_epi16(\_\_m512i s, \_\_mmask32 k, \_\_m512i a, unsigned int imm);

VPSRLW \_\_m512i \_mm512\_maskz\_srl\_epi16(\_\_mmask32 k, \_\_m512i a, unsigned int imm);

VPSRLW \_\_m256i \_mm256\_mask\_srl\_epi16(\_\_m256i s, \_\_mmask16 k, \_\_m256i a, unsigned int imm);

VPSRLW \_\_m256i \_mm256\_maskz\_srl\_epi16(\_\_mmask16 k, \_\_m256i a, unsigned int imm);

VPSRLW \_\_m128i \_mm\_mask\_srl\_epi16(\_\_m128i s, \_\_mmask8 k, \_\_m128i a, unsigned int imm);

VPSRLW \_\_m128i \_mm\_maskz\_srl\_epi16(\_\_mmask8 k, \_\_m128i a, unsigned int imm);

VPSRLW \_\_m512i \_mm512\_srl\_epi16(\_\_m512i a, \_\_m128i cnt);

VPSRLW \_\_m512i \_mm512\_mask\_srl\_epi16(\_\_m512i s, \_\_mmask32 k, \_\_m512i a, \_\_m128i cnt);



VPSRLW \_\_m512i \_\_mm512\_maskz\_srl\_epi16(\_\_mmask32 k, \_\_m512i a, \_\_m128i cnt);  
 VPSRLW \_\_m256i \_\_mm256\_mask\_srl\_epi16(\_\_m256i s, \_\_mmask16 k, \_\_m256i a, \_\_m128i cnt);  
 VPSRLW \_\_m256i \_\_mm256\_maskz\_srl\_epi16(\_\_mmask8 k, \_\_mmask16 a, \_\_m128i cnt);  
 VPSRLW \_\_m128i \_\_mm\_mask\_srl\_epi16(\_\_m128i s, \_\_mmask8 k, \_\_m128i a, \_\_m128i cnt);  
 VPSRLW \_\_m128i \_\_mm\_maskz\_srl\_epi16(\_\_mmask8 k, \_\_m128i a, \_\_m128i cnt);  
 PSRLW: \_\_m64 \_\_mm\_srli\_pi16(\_\_m64 m, int count)  
 PSRLW: \_\_m64 \_\_mm\_srl\_pi16(\_\_m64 m, \_\_m64 count)  
 (V)PSRLW: \_\_m128i \_\_mm\_srli\_epi16(\_\_m128i m, int count)  
 (V)PSRLW: \_\_m128i \_\_mm\_srl\_epi16(\_\_m128i m, \_\_m128i count)  
 VPSRLW: \_\_m256i \_\_mm256\_srli\_epi16(\_\_m256i m, int count)  
 VPSRLW: \_\_m256i \_\_mm256\_srl\_epi16(\_\_m256i m, \_\_m128i count)  
 PSRLD: \_\_m64 \_\_mm\_srli\_pi32(\_\_m64 m, int count)  
 PSRLD: \_\_m64 \_\_mm\_srl\_pi32(\_\_m64 m, \_\_m64 count)  
 (V)PSRLD: \_\_m128i \_\_mm\_srli\_epi32(\_\_m128i m, int count)  
 (V)PSRLD: \_\_m128i \_\_mm\_srl\_epi32(\_\_m128i m, \_\_m128i count)  
 VPSRLD: \_\_m256i \_\_mm256\_srli\_epi32(\_\_m256i m, int count)  
 VPSRLD: \_\_m256i \_\_mm256\_srl\_epi32(\_\_m256i m, \_\_m128i count)  
 PSRLQ: \_\_m64 \_\_mm\_srli\_si64(\_\_m64 m, int count)  
 PSRLQ: \_\_m64 \_\_mm\_srl\_si64(\_\_m64 m, \_\_m64 count)  
 (V)PSRLQ: \_\_m128i \_\_mm\_srli\_epi64(\_\_m128i m, int count)  
 (V)PSRLQ: \_\_m128i \_\_mm\_srl\_epi64(\_\_m128i m, \_\_m128i count)  
 VPSRLQ: \_\_m256i \_\_mm256\_srli\_epi64(\_\_m256i m, int count)  
 VPSRLQ: \_\_m256i \_\_mm256\_srl\_epi64(\_\_m256i m, \_\_m128i count)

### Flags Affected

None.

### Numeric Exceptions

None.

### Other Exceptions

VEX-encoded instructions:

Syntax with RM/RVM operand encoding, see Exceptions Type 4.

Syntax with MI/VMI operand encoding, see Exceptions Type 7.

EVEX-encoded VPSRLW, see Exceptions Type E4NF.nb.

EVEX-encoded VPSRLD/Q:

Syntax with M128 operand encoding, see Exceptions Type E4NF.nb.

Syntax with FVI operand encoding, see Exceptions Type E4.

## PSUBB/PSUBW/PSUBD—Subtract Packed Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F F8 /r <sup>1</sup> PSUBB <i>mm</i> , <i>mm/m64</i>	RM	V/V	MMX	Subtract packed byte integers in <i>mm/m64</i> from packed byte integers in <i>mm</i> .
66 0F F8 /r PSUBB <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE2	Subtract packed byte integers in <i>xmm2/m128</i> from packed byte integers in <i>xmm1</i> .
0F F9 /r <sup>1</sup> PSUBW <i>mm</i> , <i>mm/m64</i>	RM	V/V	MMX	Subtract packed word integers in <i>mm/m64</i> from packed word integers in <i>mm</i> .
66 0F F9 /r PSUBW <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE2	Subtract packed word integers in <i>xmm2/m128</i> from packed word integers in <i>xmm1</i> .
0F FA /r <sup>1</sup> PSUBD <i>mm</i> , <i>mm/m64</i>	RM	V/V	MMX	Subtract packed doubleword integers in <i>mm/m64</i> from packed doubleword integers in <i>mm</i> .
66 0F FA /r PSUBD <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE2	Subtract packed doubleword integers in <i>xmm2/mem128</i> from packed doubleword integers in <i>xmm1</i> .
VEX.NDS.128.66.0F.WIG F8 /r VPSUBB <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Subtract packed byte integers in <i>xmm3/m128</i> from <i>xmm2</i> .
VEX.NDS.128.66.0F.WIG F9 /r VPSUBW <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Subtract packed word integers in <i>xmm3/m128</i> from <i>xmm2</i> .
VEX.NDS.128.66.0F.WIG FA /r VPSUBD <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Subtract packed doubleword integers in <i>xmm3/m128</i> from <i>xmm2</i> .
VEX.NDS.256.66.0F.WIG F8 /r VPSUBB <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	RVM	V/V	AVX2	Subtract packed byte integers in <i>ymm3/m256</i> from <i>ymm2</i> .
VEX.NDS.256.66.0F.WIG F9 /r VPSUBW <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	RVM	V/V	AVX2	Subtract packed word integers in <i>ymm3/m256</i> from <i>ymm2</i> .
VEX.NDS.256.66.0F.WIG FA /r VPSUBD <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	RVM	V/V	AVX2	Subtract packed doubleword integers in <i>ymm3/m256</i> from <i>ymm2</i> .
EVEX.NDS.128.66.0F.WIG F8 /r VPSUBB <i>xmm1</i> {k1}{z}, <i>xmm2</i> , <i>xmm3/m128</i>	FVM	V/V	AVX512VL AVX512BW	Subtract packed byte integers in <i>xmm3/m128</i> from <i>xmm2</i> and store in <i>xmm1</i> using writemask k1.
EVEX.NDS.256.66.0F.WIG F8 /r VPSUBB <i>ymm1</i> {k1}{z}, <i>ymm2</i> , <i>ymm3/m256</i>	FVM	V/V	AVX512VL AVX512BW	Subtract packed byte integers in <i>ymm3/m256</i> from <i>ymm2</i> and store in <i>ymm1</i> using writemask k1.
EVEX.NDS.512.66.0F.WIG F8 /r VPSUBB <i>zmm1</i> {k1}{z}, <i>zmm2</i> , <i>zmm3/m512</i>	FVM	V/V	AVX512BW	Subtract packed byte integers in <i>zmm3/m512</i> from <i>zmm2</i> and store in <i>zmm1</i> using writemask k1.
EVEX.NDS.128.66.0F.WIG F9 /r VPSUBW <i>xmm1</i> {k1}{z}, <i>xmm2</i> , <i>xmm3/m128</i>	FVM	V/V	AVX512VL AVX512BW	Subtract packed word integers in <i>xmm3/m128</i> from <i>xmm2</i> and store in <i>xmm1</i> using writemask k1.
EVEX.NDS.256.66.0F.WIG F9 /r VPSUBW <i>ymm1</i> {k1}{z}, <i>ymm2</i> , <i>ymm3/m256</i>	FVM	V/V	AVX512VL AVX512BW	Subtract packed word integers in <i>ymm3/m256</i> from <i>ymm2</i> and store in <i>ymm1</i> using writemask k1.
EVEX.NDS.512.66.0F.WIG F9 /r VPSUBW <i>zmm1</i> {k1}{z}, <i>zmm2</i> , <i>zmm3/m512</i>	FVM	V/V	AVX512BW	Subtract packed word integers in <i>zmm3/m512</i> from <i>zmm2</i> and store in <i>zmm1</i> using writemask k1.

EVEX.NDS.128.66.0F.W0 FA /r VPSUBD xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	FV	V/V	AVX512VL AVX512F	Subtract packed doubleword integers in xmm3/m128/m32bcst from xmm2 and store in xmm1 using writemask k1.
EVEX.NDS.256.66.0F.W0 FA /r VPSUBD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	FV	V/V	AVX512VL AVX512F	Subtract packed doubleword integers in ymm3/m256/m32bcst from ymm2 and store in ymm1 using writemask k1.
EVEX.NDS.512.66.0F.W0 FA /r VPSUBD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	FV	V/V	AVX512F	Subtract packed doubleword integers in zmm3/m512/m32bcst from zmm2 and store in zmm1 using writemask k1.

**NOTES:**

1. See note in Section 2.4, “AVX and SSE Instruction Exception Specification” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FVM	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA
FV	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

**Description**

Performs a SIMD subtract of the packed integers of the source operand (second operand) from the packed integers of the destination operand (first operand), and stores the packed integer results in the destination operand. See Figure 9-4 in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for an illustration of a SIMD operation. Overflow is handled with wraparound, as described in the following paragraphs.

The (V)PSUBB instruction subtracts packed byte integers. When an individual result is too large or too small to be represented in a byte, the result is wrapped around and the low 8 bits are written to the destination element.

The (V)PSUBW instruction subtracts packed word integers. When an individual result is too large or too small to be represented in a word, the result is wrapped around and the low 16 bits are written to the destination element.

The (V)PSUBD instruction subtracts packed doubleword integers. When an individual result is too large or too small to be represented in a doubleword, the result is wrapped around and the low 32 bits are written to the destination element.

Note that the (V)PSUBB, (V)PSUBW, and (V)PSUBD instructions can operate on either unsigned or signed (two's complement notation) packed integers; however, it does not set bits in the EFLAGS register to indicate overflow and/or a carry. To prevent undetected overflow conditions, software must control the ranges of values upon which it operates.

- In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).
- Legacy SSE version 64-bit operand: The destination operand must be an MMX technology register and the source operand can be either an MMX technology register or a 64-bit memory location.  
128-bit Legacy SSE version: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.
- VEX.128 encoded version: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

VEX.256 encoded versions: The second source operand is an YMM register or an 256-bit memory location. The first source operand and destination operands are YMM registers. Bits (MAX\_VL-1:256) of the corresponding ZMM register are zeroed.

EVEX encoded VPSUBD: The second 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 32/64-bit memory location. The first source operand and destination operands are ZMM/YMM/XMM registers. The destination is conditionally updated with writemask k1.

EVEX encoded VPSUBB/W: The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The first source operand and destination operands are ZMM/YMM/XMM registers. The destination is conditionally updated with writemask k1.

## Operation

### PSUBB (with 64-bit operands)

```
DEST[7:0] ← DEST[7:0] – SRC[7:0];
(* Repeat subtract operation for 2nd through 7th byte *)
DEST[63:56] ← DEST[63:56] – SRC[63:56];
```

### PSUBW (with 64-bit operands)

```
DEST[15:0] ← DEST[15:0] – SRC[15:0];
(* Repeat subtract operation for 2nd and 3rd word *)
DEST[63:48] ← DEST[63:48] – SRC[63:48];
```

### PSUBD (with 64-bit operands)

```
DEST[31:0] ← DEST[31:0] – SRC[31:0];
DEST[63:32] ← DEST[63:32] – SRC[63:32];
```

### PSUBD (with 128-bit operands)

```
DEST[31:0] ← DEST[31:0] – SRC[31:0];
(* Repeat subtract operation for 2nd and 3rd doubleword *)
DEST[127:96] ← DEST[127:96] – SRC[127:96];
```

### VPSUBB (EVEX encoded versions)

(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] ← SRC1[i+7:i] - SRC2[i+7:i]

    ELSE

      IF \*merging-masking\* ; merging-masking

        THEN \*DEST[i+7:i] remains unchanged\*

        ELSE \*zeroing-masking\* ; zeroing-masking

          DEST[i+7:i] = 0

    FI

  FI;

ENDFOR;

DEST[MAX\_VL-1:VL] ← 0

### VPSUBW (EVEX encoded versions)

(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] ← SRC1[i+15:i] - SRC2[i+15:i]

    ELSE

      IF \*merging-masking\* ; merging-masking

```

        THEN *DEST[j+15:i] remains unchanged*
        ELSE *zeroing-masking*           ; zeroing-masking
          DEST[j+15:i] = 0
      FI
  FI;
ENDFOR;
DEST[MAX_VL-1:VL] ← 0

```

**VPSUBD (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

```

  i ← j * 32
  IF k1[j] OR *no writemask* THEN
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
      THEN DEST[j+31:i] ← SRC1[j+31:i] - SRC2[31:0]
      ELSE DEST[j+31:i] ← SRC1[j+31:i] - SRC2[j+31:i]
    FI;
  ELSE
    IF *merging-masking*           ; merging-masking
      THEN *DEST[j+31:i] remains unchanged*
      ELSE *zeroing-masking*       ; zeroing-masking
        DEST[j+31:i] ← 0
    FI
  FI;
ENDFOR;
DEST[MAX_VL-1:VL] ← 0

```

**VPSUBB (VEX.256 encoded version)**

```

DEST[7:0] ← SRC1[7:0]-SRC2[7:0]
DEST[15:8] ← SRC1[15:8]-SRC2[15:8]
DEST[23:16] ← SRC1[23:16]-SRC2[23:16]
DEST[31:24] ← SRC1[31:24]-SRC2[31:24]
DEST[39:32] ← SRC1[39:32]-SRC2[39:32]
DEST[47:40] ← SRC1[47:40]-SRC2[47:40]
DEST[55:48] ← SRC1[55:48]-SRC2[55:48]
DEST[63:56] ← SRC1[63:56]-SRC2[63:56]
DEST[71:64] ← SRC1[71:64]-SRC2[71:64]
DEST[79:72] ← SRC1[79:72]-SRC2[79:72]
DEST[87:80] ← SRC1[87:80]-SRC2[87:80]
DEST[95:88] ← SRC1[95:88]-SRC2[95:88]
DEST[103:96] ← SRC1[103:96]-SRC2[103:96]
DEST[111:104] ← SRC1[111:104]-SRC2[111:104]
DEST[119:112] ← SRC1[119:112]-SRC2[119:112]
DEST[127:120] ← SRC1[127:120]-SRC2[127:120]
DEST[135:128] ← SRC1[135:128]-SRC2[135:128]
DEST[143:136] ← SRC1[143:136]-SRC2[143:136]
DEST[151:144] ← SRC1[151:144]-SRC2[151:144]
DEST[159:152] ← SRC1[159:152]-SRC2[159:152]
DEST[167:160] ← SRC1[167:160]-SRC2[167:160]
DEST[175:168] ← SRC1[175:168]-SRC2[175:168]
DEST[183:176] ← SRC1[183:176]-SRC2[183:176]
DEST[191:184] ← SRC1[191:184]-SRC2[191:184]
DEST[199:192] ← SRC1[199:192]-SRC2[199:192]
DEST[207:200] ← SRC1[207:200]-SRC2[207:200]

```

DEST[215:208] ← SRC1[215:208]-SRC2[215:208]  
 DEST[223:216] ← SRC1[223:216]-SRC2[223:216]  
 DEST[231:224] ← SRC1[231:224]-SRC2[231:224]  
 DEST[239:232] ← SRC1[239:232]-SRC2[239:232]  
 DEST[247:240] ← SRC1[247:240]-SRC2[247:240]  
 DEST[255:248] ← SRC1[255:248]-SRC2[255:248]  
 DEST[MAX\_VL-1:256] ← 0

#### **VPSUBB (VEX.128 encoded version)**

DEST[7:0] ← SRC1[7:0]-SRC2[7:0]  
 DEST[15:8] ← SRC1[15:8]-SRC2[15:8]  
 DEST[23:16] ← SRC1[23:16]-SRC2[23:16]  
 DEST[31:24] ← SRC1[31:24]-SRC2[31:24]  
 DEST[39:32] ← SRC1[39:32]-SRC2[39:32]  
 DEST[47:40] ← SRC1[47:40]-SRC2[47:40]  
 DEST[55:48] ← SRC1[55:48]-SRC2[55:48]  
 DEST[63:56] ← SRC1[63:56]-SRC2[63:56]  
 DEST[71:64] ← SRC1[71:64]-SRC2[71:64]  
 DEST[79:72] ← SRC1[79:72]-SRC2[79:72]  
 DEST[87:80] ← SRC1[87:80]-SRC2[87:80]  
 DEST[95:88] ← SRC1[95:88]-SRC2[95:88]  
 DEST[103:96] ← SRC1[103:96]-SRC2[103:96]  
 DEST[111:104] ← SRC1[111:104]-SRC2[111:104]  
 DEST[119:112] ← SRC1[119:112]-SRC2[119:112]  
 DEST[127:120] ← SRC1[127:120]-SRC2[127:120]  
 DEST[MAX\_VL-1:128] ← 0

#### **PSUBB (128-bit Legacy SSE version)**

DEST[7:0] ← DEST[7:0]-SRC[7:0]  
 DEST[15:8] ← DEST[15:8]-SRC[15:8]  
 DEST[23:16] ← DEST[23:16]-SRC[23:16]  
 DEST[31:24] ← DEST[31:24]-SRC[31:24]  
 DEST[39:32] ← DEST[39:32]-SRC[39:32]  
 DEST[47:40] ← DEST[47:40]-SRC[47:40]  
 DEST[55:48] ← DEST[55:48]-SRC[55:48]  
 DEST[63:56] ← DEST[63:56]-SRC[63:56]  
 DEST[71:64] ← DEST[71:64]-SRC[71:64]  
 DEST[79:72] ← DEST[79:72]-SRC[79:72]  
 DEST[87:80] ← DEST[87:80]-SRC[87:80]  
 DEST[95:88] ← DEST[95:88]-SRC[95:88]  
 DEST[103:96] ← DEST[103:96]-SRC[103:96]  
 DEST[111:104] ← DEST[111:104]-SRC[111:104]  
 DEST[119:112] ← DEST[119:112]-SRC[119:112]  
 DEST[127:120] ← DEST[127:120]-SRC[127:120]  
 DEST[MAX\_VL-1:128] (Unmodified)

#### **VPSUBW (VEX.256 encoded version)**

DEST[15:0] ← SRC1[15:0]-SRC2[15:0]  
 DEST[31:16] ← SRC1[31:16]-SRC2[31:16]  
 DEST[47:32] ← SRC1[47:32]-SRC2[47:32]  
 DEST[63:48] ← SRC1[63:48]-SRC2[63:48]  
 DEST[79:64] ← SRC1[79:64]-SRC2[79:64]  
 DEST[95:80] ← SRC1[95:80]-SRC2[95:80]  
 DEST[111:96] ← SRC1[111:96]-SRC2[111:96]

DEST[127:112] ← SRC1[127:112]-SRC2[127:112]  
 DEST[143:128] ← SRC1[143:128]-SRC2[143:128]  
 DEST[159:144] ← SRC1[159:144]-SRC2[159:144]  
 DEST[175:160] ← SRC1[175:160]-SRC2[175:160]  
 DEST[191:176] ← SRC1[191:176]-SRC2[191:176]  
 DEST[207:192] ← SRC1[207:192]-SRC2[207:192]  
 DEST[223:208] ← SRC1[223:208]-SRC2[223:208]  
 DEST[239:224] ← SRC1[239:224]-SRC2[239:224]  
 DEST[255:240] ← SRC1[255:240]-SRC2[255:240]  
 DEST[MAX\_VL-1:256] ← 0

**VPSUBW (VEX.128 encoded version)**

DEST[15:0] ← SRC1[15:0]-SRC2[15:0]  
 DEST[31:16] ← SRC1[31:16]-SRC2[31:16]  
 DEST[47:32] ← SRC1[47:32]-SRC2[47:32]  
 DEST[63:48] ← SRC1[63:48]-SRC2[63:48]  
 DEST[79:64] ← SRC1[79:64]-SRC2[79:64]  
 DEST[95:80] ← SRC1[95:80]-SRC2[95:80]  
 DEST[111:96] ← SRC1[111:96]-SRC2[111:96]  
 DEST[127:112] ← SRC1[127:112]-SRC2[127:112]  
 DEST[MAX\_VL-1:128] ← 0

**PSUBW (128-bit Legacy SSE version)**

DEST[15:0] ← DEST[15:0]-SRC[15:0]  
 DEST[31:16] ← DEST[31:16]-SRC[31:16]  
 DEST[47:32] ← DEST[47:32]-SRC[47:32]  
 DEST[63:48] ← DEST[63:48]-SRC[63:48]  
 DEST[79:64] ← DEST[79:64]-SRC[79:64]  
 DEST[95:80] ← DEST[95:80]-SRC[95:80]  
 DEST[111:96] ← DEST[111:96]-SRC[111:96]  
 DEST[127:112] ← DEST[127:112]-SRC[127:112]  
 DEST[MAX\_VL-1:128] (Unmodified)

**VPSUBD (VEX.256 encoded version)**

DEST[31:0] ← SRC1[31:0]-SRC2[31:0]  
 DEST[63:32] ← SRC1[63:32]-SRC2[63:32]  
 DEST[95:64] ← SRC1[95:64]-SRC2[95:64]  
 DEST[127:96] ← SRC1[127:96]-SRC2[127:96]  
 DEST[159:128] ← SRC1[159:128]-SRC2[159:128]  
 DEST[191:160] ← SRC1[191:160]-SRC2[191:160]  
 DEST[223:192] ← SRC1[223:192]-SRC2[223:192]  
 DEST[255:224] ← SRC1[255:224]-SRC2[255:224]  
 DEST[MAX\_VL-1:256] ← 0

**VPSUBD (VEX.128 encoded version)**

DEST[31:0] ← SRC1[31:0]-SRC2[31:0]  
 DEST[63:32] ← SRC1[63:32]-SRC2[63:32]  
 DEST[95:64] ← SRC1[95:64]-SRC2[95:64]  
 DEST[127:96] ← SRC1[127:96]-SRC2[127:96]  
 DEST[MAX\_VL-1:128] ← 0

**PSUBD (128-bit Legacy SSE version)**

DEST[31:0] ← DEST[31:0]-SRC[31:0]  
 DEST[63:32] ← DEST[63:32]-SRC[63:32]

DEST[95:64] ←DEST[95:64]-SRC[95:64]  
 DEST[127:96] ←DEST[127:96]-SRC[127:96]  
 DEST[MAX\_VL-1:128] (Unmodified)

### Intel C/C++ Compiler Intrinsic Equivalents

VPSUBB \_\_m512i \_\_mm512\_sub\_epi8(\_\_m512i a, \_\_m512i b);  
 VPSUBB \_\_m512i \_\_mm512\_mask\_sub\_epi8(\_\_m512i s, \_\_mmask64 k, \_\_m512i a, \_\_m512i b);  
 VPSUBB \_\_m512i \_\_mm512\_maskz\_sub\_epi8(\_\_mmask64 k, \_\_m512i a, \_\_m512i b);  
 VPSUBB \_\_m256i \_\_mm256\_mask\_sub\_epi8(\_\_m256i s, \_\_mmask32 k, \_\_m256i a, \_\_m256i b);  
 VPSUBB \_\_m256i \_\_mm256\_maskz\_sub\_epi8(\_\_mmask32 k, \_\_m256i a, \_\_m256i b);  
 VPSUBB \_\_m128i \_\_mm\_mask\_sub\_epi8(\_\_m128i s, \_\_mmask16 k, \_\_m128i a, \_\_m128i b);  
 VPSUBB \_\_m128i \_\_mm\_maskz\_sub\_epi8(\_\_mmask16 k, \_\_m128i a, \_\_m128i b);  
 VPSUBW \_\_m512i \_\_mm512\_sub\_epi16(\_\_m512i a, \_\_m512i b);  
 VPSUBW \_\_m512i \_\_mm512\_mask\_sub\_epi16(\_\_m512i s, \_\_mmask32 k, \_\_m512i a, \_\_m512i b);  
 VPSUBW \_\_m512i \_\_mm512\_maskz\_sub\_epi16(\_\_mmask32 k, \_\_m512i a, \_\_m512i b);  
 VPSUBW \_\_m256i \_\_mm256\_mask\_sub\_epi16(\_\_m256i s, \_\_mmask16 k, \_\_m256i a, \_\_m256i b);  
 VPSUBW \_\_m256i \_\_mm256\_maskz\_sub\_epi16(\_\_mmask16 k, \_\_m256i a, \_\_m256i b);  
 VPSUBW \_\_m128i \_\_mm\_mask\_sub\_epi16(\_\_m128i s, \_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
 VPSUBW \_\_m128i \_\_mm\_maskz\_sub\_epi16(\_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
 VPSUBD \_\_m512i \_\_mm512\_sub\_epi32(\_\_m512i a, \_\_m512i b);  
 VPSUBD \_\_m512i \_\_mm512\_mask\_sub\_epi32(\_\_m512i s, \_\_mmask16 k, \_\_m512i a, \_\_m512i b);  
 VPSUBD \_\_m512i \_\_mm512\_maskz\_sub\_epi32(\_\_mmask16 k, \_\_m512i a, \_\_m512i b);  
 VPSUBD \_\_m256i \_\_mm256\_mask\_sub\_epi32(\_\_m256i s, \_\_mmask8 k, \_\_m256i a, \_\_m256i b);  
 VPSUBD \_\_m256i \_\_mm256\_maskz\_sub\_epi32(\_\_mmask8 k, \_\_m256i a, \_\_m256i b);  
 VPSUBD \_\_m128i \_\_mm\_mask\_sub\_epi32(\_\_m128i s, \_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
 VPSUBD \_\_m128i \_\_mm\_maskz\_sub\_epi32(\_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
 PSUBB: \_\_m64 \_\_mm\_sub\_pi8(\_\_m64 m1, \_\_m64 m2)  
 (V)PSUBB: \_\_m128i \_\_mm\_sub\_epi8 (\_\_m128i a, \_\_m128i b)  
 VPSUBB: \_\_m256i \_\_mm256\_sub\_epi8 (\_\_m256i a, \_\_m256i b)  
 PSUBW: \_\_m64 \_\_mm\_sub\_pi16(\_\_m64 m1, \_\_m64 m2)  
 (V)PSUBW: \_\_m128i \_\_mm\_sub\_epi16 (\_\_m128i a, \_\_m128i b)  
 VPSUBW: \_\_m256i \_\_mm256\_sub\_epi16 (\_\_m256i a, \_\_m256i b)  
 PSUBD: \_\_m64 \_\_mm\_sub\_pi32(\_\_m64 m1, \_\_m64 m2)  
 (V)PSUBD: \_\_m128i \_\_mm\_sub\_epi32 (\_\_m128i a, \_\_m128i b)  
 VPSUBD: \_\_m256i \_\_mm256\_sub\_epi32 (\_\_m256i a, \_\_m256i b)

### Flags Affected

None.

### Numeric Exceptions

None.

### Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded VPSUBD, see Exceptions Type E4.

EVEX-encoded VPSUBB/W, see Exceptions Type E4.nb.



## PSUBQ—Subtract Packed Quadword Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F FB /r <sup>1</sup> PSUBQ mm1, mm2/m64	RM	V/V	SSE2	Subtract quadword integer in mm1 from mm2 /m64.
66 0F FB /r PSUBQ xmm1, xmm2/m128	RM	V/V	SSE2	Subtract packed quadword integers in xmm1 from xmm2 /m128.
VEX.NDS.128.66.0F.WIG FB/r VPSUBQ xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Subtract packed quadword integers in xmm3/m128 from xmm2.
VEX.NDS.256.66.0F.WIG FB /r VPSUBQ ymm1, ymm2, ymm3/m256	RVM	V/V	AVX2	Subtract packed quadword integers in ymm3/m256 from ymm2.
EVEX.NDS.128.66.0F.W1 FB /r VPSUBQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	FV	V/V	AVX512VL AVX512F	Subtract packed quadword integers in xmm3/m128/m64bcst from xmm2 and store in xmm1 using writemask k1.
EVEX.NDS.256.66.0F.W1 FB /r VPSUBQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	FV	V/V	AVX512VL AVX512F	Subtract packed quadword integers in ymm3/m256/m64bcst from ymm2 and store in ymm1 using writemask k1.
EVEX.NDS.512.66.0F.W1 FB/r VPSUBQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	FV	V/V	AVX512F	Subtract packed quadword integers in zmm3/m512/m64bcst from zmm2 and store in zmm1 using writemask k1.

### NOTES:

1. See note in Section 2.4, “AVX and SSE Instruction Exception Specification” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FV	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Subtracts the second operand (source operand) from the first operand (destination operand) and stores the result in the destination operand. When packed quadword operands are used, a SIMD subtract is performed. When a quadword result is too large to be represented in 64 bits (overflow), the result is wrapped around and the low 64 bits are written to the destination element (that is, the carry is ignored).

Note that the (V)PSUBQ instruction can operate on either unsigned or signed (two’s complement notation) integers; however, it does not set bits in the EFLAGS register to indicate overflow and/or a carry. To prevent undetected overflow conditions, software must control the ranges of the values upon which it operates.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE version 64-bit operand: The source operand can be a quadword integer stored in an MMX technology register or a 64-bit memory location.

128-bit Legacy SSE version: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

VEX.256 encoded versions: The second source operand is an YMM register or an 256-bit memory location. The first source operand and destination operands are YMM registers. Bits (MAX\_VL-1:256) of the corresponding ZMM register are zeroed.

EVEX encoded VPSUBQ: The second 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 32/64-bit memory location. The first source operand and destination operands are ZMM/YMM/XMM registers. The destination is conditionally updated with writemask k1.

## Operation

### PSUBQ (with 64-Bit operands)

```
DEST[63:0] ← DEST[63:0] – SRC[63:0];
```

### PSUBQ (with 128-Bit operands)

```
DEST[63:0] ← DEST[63:0] – SRC[63:0];
DEST[127:64] ← DEST[127:64] – SRC[127:64];
```

### VPSUBQ (VEX.128 encoded version)

```
DEST[63:0] ← SRC1[63:0]-SRC2[63:0]
DEST[127:64] ← SRC1[127:64]-SRC2[127:64]
DEST[VLMAX-1:128] ← 0
```

### VPSUBQ (VEX.256 encoded version)

```
DEST[63:0] ← SRC1[63:0]-SRC2[63:0]
DEST[127:64] ← SRC1[127:64]-SRC2[127:64]
DEST[191:128] ← SRC1[191:128]-SRC2[191:128]
DEST[255:192] ← SRC1[255:192]-SRC2[255:192]
DEST[VLMAX-1:256] ← 0
```

### VPSUBQ (EVEX encoded versions)

(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) AND (SRC2 \*is memory\*)

      THEN DEST[i+63:i] ← SRC1[i+63:i] - SRC2[63:0]

      ELSE DEST[i+63:i] ← SRC1[i+63:i] - SRC2[i+63:i]

    FI;

  ELSE

    IF \*merging-masking\* ; merging-masking

      THEN \*DEST[i+63:i] remains unchanged\*

      ELSE \*zeroing-masking\* ; zeroing-masking

        DEST[i+63:i] ← 0

    FI

  FI;

ENDFOR;

DEST[MAX\_VL-1:VL] ← 0

## Intel C/C++ Compiler Intrinsic Equivalents

```
VPSUBQ __m512i __mm512_sub_epi64(__m512i a, __m512i b);
```

```
VPSUBQ __m512i __mm512_mask_sub_epi64(__m512i s, __mmask8 k, __m512i a, __m512i b);
```

```
VPSUBQ __m512i __mm512_maskz_sub_epi64(__mmask8 k, __m512i a, __m512i b);
```

```
VPSUBQ __m256i __mm256_mask_sub_epi64(__m256i s, __mmask8 k, __m256i a, __m256i b);
```

VPSUBQ \_\_m256i \_\_mm256\_maskz\_sub\_epi64( \_\_mmask8 k, \_\_m256i a, \_\_m256i b);  
VPSUBQ \_\_m128i \_\_mm\_mask\_sub\_epi64(\_\_m128i s, \_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
VPSUBQ \_\_m128i \_\_mm\_maskz\_sub\_epi64( \_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
PSUBQ: \_\_m64 \_\_mm\_sub\_si64(\_\_m64 m1, \_\_m64 m2)  
(V)PSUBQ: \_\_m128i \_\_mm\_sub\_epi64(\_\_m128i m1, \_\_m128i m2)  
VPSUBQ: \_\_m256i \_\_mm256\_sub\_epi64(\_\_m256i m1, \_\_m256i m2)

### Flags Affected

None.

### Numeric Exceptions

None.

### Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded VPSUBQ, see Exceptions Type E4.

## PSUBSB/PSUBSW—Subtract Packed Signed Integers with Signed Saturation

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F E8 /r <sup>1</sup> PSUBSB <i>mm</i> , <i>mm/m64</i>	RM	V/V	MMX	Subtract signed packed bytes in <i>mm/m64</i> from signed packed bytes in <i>mm</i> and saturate results.
66 0F E8 /r PSUBSB <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE2	Subtract packed signed byte integers in <i>xmm2/m128</i> from packed signed byte integers in <i>xmm1</i> and saturate results.
0F E9 /r <sup>1</sup> PSUBSW <i>mm</i> , <i>mm/m64</i>	RM	V/V	MMX	Subtract signed packed words in <i>mm/m64</i> from signed packed words in <i>mm</i> and saturate results.
66 0F E9 /r PSUBSW <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE2	Subtract packed signed word integers in <i>xmm2/m128</i> from packed signed word integers in <i>xmm1</i> and saturate results.
VEX.NDS.128.66.0F.WIG E8 /r VPSUBSB <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Subtract packed signed byte integers in <i>xmm3/m128</i> from packed signed byte integers in <i>xmm2</i> and saturate results.
VEX.NDS.128.66.0F.WIG E9 /r VPSUBSW <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Subtract packed signed word integers in <i>xmm3/m128</i> from packed signed word integers in <i>xmm2</i> and saturate results.
VEX.NDS.256.66.0F.WIG E8 /r VPSUBSB <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	RVM	V/V	AVX2	Subtract packed signed byte integers in <i>ymm3/m256</i> from packed signed byte integers in <i>ymm2</i> and saturate results.
VEX.NDS.256.66.0F.WIG E9 /r VPSUBSW <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	RVM	V/V	AVX2	Subtract packed signed word integers in <i>ymm3/m256</i> from packed signed word integers in <i>ymm2</i> and saturate results.
EVEX.NDS.128.66.0F.WIG E8 /r VPSUBSB <i>xmm1</i> { <i>k1</i> }{ <i>z</i> }, <i>xmm2</i> , <i>xmm3/m128</i>	FVM	V/V	AVX512VL AVX512BW	Subtract packed signed byte integers in <i>xmm3/m128</i> from packed signed byte integers in <i>xmm2</i> and saturate results and store in <i>xmm1</i> using writemask <i>k1</i> .
EVEX.NDS.256.66.0F.WIG E8 /r VPSUBSB <i>ymm1</i> { <i>k1</i> }{ <i>z</i> }, <i>ymm2</i> , <i>ymm3/m256</i>	FVM	V/V	AVX512VL AVX512BW	Subtract packed signed byte integers in <i>ymm3/m256</i> from packed signed byte integers in <i>ymm2</i> and saturate results and store in <i>ymm1</i> using writemask <i>k1</i> .
EVEX.NDS.512.66.0F.WIG E8 /r VPSUBSB <i>zmm1</i> { <i>k1</i> }{ <i>z</i> }, <i>zmm2</i> , <i>zmm3/m512</i>	FVM	V/V	AVX512BW	Subtract packed signed byte integers in <i>zmm3/m512</i> from packed signed byte integers in <i>zmm2</i> and saturate results and store in <i>zmm1</i> using writemask <i>k1</i> .
EVEX.NDS.128.66.0F.WIG E9 /r VPSUBSW <i>xmm1</i> { <i>k1</i> }{ <i>z</i> }, <i>xmm2</i> , <i>xmm3/m128</i>	FVM	V/V	AVX512VL AVX512BW	Subtract packed signed word integers in <i>xmm3/m128</i> from packed signed word integers in <i>xmm2</i> and saturate results and store in <i>xmm1</i> using writemask <i>k1</i> .
EVEX.NDS.256.66.0F.WIG E9 /r VPSUBSW <i>ymm1</i> { <i>k1</i> }{ <i>z</i> }, <i>ymm2</i> , <i>ymm3/m256</i>	FVM	V/V	AVX512VL AVX512BW	Subtract packed signed word integers in <i>ymm3/m256</i> from packed signed word integers in <i>ymm2</i> and saturate results and store in <i>ymm1</i> using writemask <i>k1</i> .

EVEX.NDS.512.66.0F.WIG E9 /r VPSUBSW zmm1 {k1}{z}, zmm2, zmm3/m512	FVM V/V	AVX512BW	Subtract packed signed word integers in zmm3/m512 from packed signed word integers in zmm2 and saturate results and store in zmm1 using writemask k1.
-----------------------------------------------------------------------	---------	----------	-------------------------------------------------------------------------------------------------------------------------------------------------------

**NOTES:**

1. See note in Section 2.4, “AVX and SSE Instruction Exception Specification” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FVM	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

**Description**

Performs a SIMD subtract of the packed signed integers of the source operand (second operand) from the packed signed integers of the destination operand (first operand), and stores the packed integer results in the destination operand. See Figure 9-4 in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for an illustration of a SIMD operation. Overflow is handled with signed saturation, as described in the following paragraphs.

The (V)PSUBSB instruction subtracts packed signed byte integers. When an individual byte result is beyond the range of a signed byte integer (that is, greater than 7FH or less than 80H), the saturated value of 7FH or 80H, respectively, is written to the destination operand.

The (V)PSUBSW instruction subtracts packed signed word integers. When an individual word result is beyond the range of a signed word integer (that is, greater than 7FFFH or less than 8000H), the saturated value of 7FFFH or 8000H, respectively, is written to the destination operand.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE version 64-bit operand: The destination operand must be an MMX technology register and the source operand can be either an MMX technology register or a 64-bit memory location.

128-bit Legacy SSE version: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

VEX.256 encoded versions: The second source operand is an YMM register or a 256-bit memory location. The first source operand and destination operands are YMM registers. Bits (MAX\_VL-1:256) of the corresponding ZMM register are zeroed.

EVEX encoded version: The second source operand is an ZMM/YMM/XMM register or a 512/256/128-bit memory location. The first source operand and destination operands are ZMM/YMM/XMM registers. The destination is conditionally updated with writemask k1.

**Operation****PSUBSB (with 64-bit operands)**

DEST[7:0] ← SaturateToSignedByte (DEST[7:0] – SRC (7:0));

(\* Repeat subtract operation for 2nd through 7th bytes \*)

DEST[63:56] ← SaturateToSignedByte (DEST[63:56] – SRC[63:56] );

**PSUBSW (with 64-bit operands)**

```
DEST[15:0] ← SaturateToSignedWord (DEST[15:0] – SRC[15:0]);
(* Repeat subtract operation for 2nd and 7th words *)
DEST[63:48] ← SaturateToSignedWord (DEST[63:48] – SRC[63:48]);
```

**VPSUBSB (EVEX encoded versions)**

(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] ← SaturateToSignedByte (SRC1[i+7:i] - SRC2[i+7:i])
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+7:i] remains unchanged*
        ELSE *zeroing-masking* ; zeroing-masking
          DEST[i+7:i] ← 0;
      FI
  FI;
ENDFOR;
DEST[MAX_VL-1:VL] ← 0
```

**VPSUBSW (EVEX encoded versions)**

(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] ← SaturateToSignedWord (SRC1[i+15:i] - SRC2[i+15:i])
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+15:i] remains unchanged*
        ELSE *zeroing-masking* ; zeroing-masking
          DEST[i+15:i] ← 0;
      FI
  FI;
ENDFOR;
DEST[MAX_VL-1:VL] ← 0;
```

**VPSUBSB (VEX.256 encoded version)**

```
DEST[7:0] ← SaturateToSignedByte (SRC1[7:0] - SRC2[7:0]);
(* Repeat subtract operation for 2nd through 31th bytes *)
DEST[255:248] ← SaturateToSignedByte (SRC1[255:248] - SRC2[255:248]);
DEST[MAX_VL-1:256] ← 0;
```

**VPSUBSB (VEX.128 encoded version)**

```
DEST[7:0] ← SaturateToSignedByte (SRC1[7:0] - SRC2[7:0]);
(* Repeat subtract operation for 2nd through 14th bytes *)
DEST[127:120] ← SaturateToSignedByte (SRC1[127:120] - SRC2[127:120]);
DEST[MAX_VL-1:128] ← 0;
```

**PSUBSB (128-bit Legacy SSE Version)**

```
DEST[7:0] ← SaturateToSignedByte (DEST[7:0] - SRC[7:0]);
(* Repeat subtract operation for 2nd through 14th bytes *)
DEST[127:120] ← SaturateToSignedByte (DEST[127:120] - SRC[127:120]);
DEST[MAX_VL-1:128] (Unmodified);
```

**VPSUBSW (VEX.256 encoded version)**

DEST[15:0] ← SaturateToSignedWord (SRC1[15:0] - SRC2[15:0]);  
 (\* Repeat subtract operation for 2nd through 15th words \*)  
 DEST[255:240] ← SaturateToSignedWord (SRC1[255:240] - SRC2[255:240]);  
 DEST[MAX\_VL-1:256] ← 0;

**VPSUBSW (VEX.128 encoded version)**

DEST[15:0] ← SaturateToSignedWord (SRC1[15:0] - SRC2[15:0]);  
 (\* Repeat subtract operation for 2nd through 7th words \*)  
 DEST[127:112] ← SaturateToSignedWord (SRC1[127:112] - SRC2[127:112]);  
 DEST[MAX\_VL-1:128] ← 0;

**PSUBSW (128-bit Legacy SSE Version)**

DEST[15:0] ← SaturateToSignedWord (DEST[15:0] - SRC[15:0]);  
 (\* Repeat subtract operation for 2nd through 7th words \*)  
 DEST[127:112] ← SaturateToSignedWord (DEST[127:112] - SRC[127:112]);  
 DEST[MAX\_VL-1:128] (Unmodified);

**Intel C/C++ Compiler Intrinsic Equivalents**

VPSUBSB \_\_m512i\_mm512\_subs\_epi8(\_\_m512i a, \_\_m512i b);  
 VPSUBSB \_\_m512i\_mm512\_mask\_subs\_epi8(\_\_m512i s, \_\_mmask64 k, \_\_m512i a, \_\_m512i b);  
 VPSUBSB \_\_m512i\_mm512\_maskz\_subs\_epi8(\_\_mmask64 k, \_\_m512i a, \_\_m512i b);  
 VPSUBSB \_\_m256i\_mm256\_mask\_subs\_epi8(\_\_m256i s, \_\_mmask32 k, \_\_m256i a, \_\_m256i b);  
 VPSUBSB \_\_m256i\_mm256\_maskz\_subs\_epi8(\_\_mmask32 k, \_\_m256i a, \_\_m256i b);  
 VPSUBSB \_\_m128i\_mm\_mask\_subs\_epi8(\_\_m128i s, \_\_mmask16 k, \_\_m128i a, \_\_m128i b);  
 VPSUBSB \_\_m128i\_mm\_maskz\_subs\_epi8(\_\_mmask16 k, \_\_m128i a, \_\_m128i b);  
 VPSUBSW \_\_m512i\_mm512\_subs\_epi16(\_\_m512i a, \_\_m512i b);  
 VPSUBSW \_\_m512i\_mm512\_mask\_subs\_epi16(\_\_m512i s, \_\_mmask32 k, \_\_m512i a, \_\_m512i b);  
 VPSUBSW \_\_m512i\_mm512\_maskz\_subs\_epi16(\_\_mmask32 k, \_\_m512i a, \_\_m512i b);  
 VPSUBSW \_\_m256i\_mm256\_mask\_subs\_epi16(\_\_m256i s, \_\_mmask16 k, \_\_m256i a, \_\_m256i b);  
 VPSUBSW \_\_m256i\_mm256\_maskz\_subs\_epi16(\_\_mmask16 k, \_\_m256i a, \_\_m256i b);  
 VPSUBSW \_\_m128i\_mm\_mask\_subs\_epi16(\_\_m128i s, \_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
 VPSUBSW \_\_m128i\_mm\_maskz\_subs\_epi16(\_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
 PSUBSB: \_\_m64\_mm\_subs\_pi8(\_\_m64 m1, \_\_m64 m2)  
 (V)PSUBSB: \_\_m128i\_mm\_subs\_epi8(\_\_m128i m1, \_\_m128i m2)  
 VPSUBSB: \_\_m256i\_mm256\_subs\_epi8(\_\_m256i m1, \_\_m256i m2)  
 PSUBSW: \_\_m64\_mm\_subs\_pi16(\_\_m64 m1, \_\_m64 m2)  
 (V)PSUBSW: \_\_m128i\_mm\_subs\_epi16(\_\_m128i m1, \_\_m128i m2)  
 VPSUBSW: \_\_m256i\_mm256\_subs\_epi16(\_\_m256i m1, \_\_m256i m2)

**Flags Affected**

None.

**Numeric Exceptions**

None.

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.nb.

## PSUBUSB/PSUBUSW—Subtract Packed Unsigned Integers with Unsigned Saturation

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F D8 /r <sup>1</sup> PSUBUSB <i>mm</i> , <i>mm/m64</i>	RM	V/V	MMX	Subtract unsigned packed bytes in <i>mm/m64</i> from unsigned packed bytes in <i>mm</i> and saturate result.
66 0F D8 /r PSUBUSB <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE2	Subtract packed unsigned byte integers in <i>xmm2/m128</i> from packed unsigned byte integers in <i>xmm1</i> and saturate result.
0F D9 /r <sup>1</sup> PSUBUSW <i>mm</i> , <i>mm/m64</i>	RM	V/V	MMX	Subtract unsigned packed words in <i>mm/m64</i> from unsigned packed words in <i>mm</i> and saturate result.
66 0F D9 /r PSUBUSW <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE2	Subtract packed unsigned word integers in <i>xmm2/m128</i> from packed unsigned word integers in <i>xmm1</i> and saturate result.
VEX.NDS.128.66.0F.WIG D8 /r VPSUBUSB <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Subtract packed unsigned byte integers in <i>xmm3/m128</i> from packed unsigned byte integers in <i>xmm2</i> and saturate result.
VEX.NDS.128.66.0F.WIG D9 /r VPSUBUSW <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Subtract packed unsigned word integers in <i>xmm3/m128</i> from packed unsigned word integers in <i>xmm2</i> and saturate result.
VEX.NDS.256.66.0F.WIG D8 /r VPSUBUSB <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	RVM	V/V	AVX2	Subtract packed unsigned byte integers in <i>ymm3/m256</i> from packed unsigned byte integers in <i>ymm2</i> and saturate result.
VEX.NDS.256.66.0F.WIG D9 /r VPSUBUSW <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	RVM	V/V	AVX2	Subtract packed unsigned word integers in <i>ymm3/m256</i> from packed unsigned word integers in <i>ymm2</i> and saturate result.
EVEX.NDS.128.66.0F.WIG D8 /r VPSUBUSB <i>xmm1</i> {k1}{z}, <i>xmm2</i> , <i>xmm3/m128</i>	FVM	V/V	AVX512VL AVX512BW	Subtract packed unsigned byte integers in <i>xmm3/m128</i> from packed unsigned byte integers in <i>xmm2</i> , saturate results and store in <i>xmm1</i> using writemask <i>k1</i> .
EVEX.NDS.256.66.0F.WIG D8 /r VPSUBUSB <i>ymm1</i> {k1}{z}, <i>ymm2</i> , <i>ymm3/m256</i>	FVM	V/V	AVX512VL AVX512BW	Subtract packed unsigned byte integers in <i>ymm3/m256</i> from packed unsigned byte integers in <i>ymm2</i> , saturate results and store in <i>ymm1</i> using writemask <i>k1</i> .
EVEX.NDS.512.66.0F.WIG D8 /r VPSUBUSB <i>zmm1</i> {k1}{z}, <i>zmm2</i> , <i>zmm3/m512</i>	FVM	V/V	AVX512BW	Subtract packed unsigned byte integers in <i>zmm3/m512</i> from packed unsigned byte integers in <i>zmm2</i> , saturate results and store in <i>zmm1</i> using writemask <i>k1</i> .
EVEX.NDS.128.66.0F.WIG D9 /r VPSUBUSW <i>xmm1</i> {k1}{z}, <i>xmm2</i> , <i>xmm3/m128</i>	FVM	V/V	AVX512VL AVX512BW	Subtract packed unsigned word integers in <i>xmm3/m128</i> from packed unsigned word integers in <i>xmm2</i> and saturate results and store in <i>xmm1</i> using writemask <i>k1</i> .
EVEX.NDS.256.66.0F.WIG D9 /r VPSUBUSW <i>ymm1</i> {k1}{z}, <i>ymm2</i> , <i>ymm3/m256</i>	FVM	V/V	AVX512VL AVX512BW	Subtract packed unsigned word integers in <i>ymm3/m256</i> from packed unsigned word integers in <i>ymm2</i> , saturate results and store in <i>ymm1</i> using writemask <i>k1</i> .



EVEX.NDS.512.66.0F.WIG D9 /r VPSUBUSW zmm1 {k1}{z}, zmm2, zmm3/m512	FVM V/V	AVX512BW	Subtract packed unsigned word integers in zmm3/m512 from packed unsigned word integers in zmm2, saturate results and store in zmm1 using writemask k1.
------------------------------------------------------------------------	---------	----------	--------------------------------------------------------------------------------------------------------------------------------------------------------

**NOTES:**

1. See note in Section 2.4, “AVX and SSE Instruction Exception Specification” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FVM	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

**Description**

Performs a SIMD subtract of the packed unsigned integers of the source operand (second operand) from the packed unsigned integers of the destination operand (first operand), and stores the packed unsigned integer results in the destination operand. See Figure 9-4 in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for an illustration of a SIMD operation. Overflow is handled with unsigned saturation, as described in the following paragraphs.

These instructions can operate on either 64-bit or 128-bit operands.

The (V)PSUBUSB instruction subtracts packed unsigned byte integers. When an individual byte result is less than zero, the saturated value of 00H is written to the destination operand.

The (V)PSUBUSW instruction subtracts packed unsigned word integers. When an individual word result is less than zero, the saturated value of 0000H is written to the destination operand.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE version 64-bit operand: The destination operand must be an MMX technology register and the source operand can be either an MMX technology register or a 64-bit memory location.

128-bit Legacy SSE version: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

VEX.256 encoded versions: The second source operand is an YMM register or a 256-bit memory location. The first source operand and destination operands are YMM registers. Bits (MAX\_VL-1:256) of the corresponding ZMM register are zeroed.

EVEX encoded version: The second source operand is an ZMM/YMM/XMM register or an 512/256/128-bit memory location. The first source operand and destination operands are ZMM/YMM/XMM registers. The destination is conditionally updated with writemask k1.

**Operation****PSUBUSB (with 64-bit operands)**

DEST[7:0] ← SaturateToUnsignedByte (DEST[7:0] – SRC (7:0));

(\* Repeat add operation for 2nd through 7th bytes \*)

DEST[63:56] ← SaturateToUnsignedByte (DEST[63:56] – SRC[63:56]);

**PSUBUSW (with 64-bit operands)**

```
DEST[15:0] ← SaturateToUnsignedWord (DEST[15:0] – SRC[15:0] );
(* Repeat add operation for 2nd and 3rd words *)
DEST[63:48] ← SaturateToUnsignedWord (DEST[63:48] – SRC[63:48] );
```

**VPSUBUSB (EVEX encoded versions)**

(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] ← SaturateToUnsignedByte (SRC1[i+7:i] - SRC2[i+7:i])

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+7:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[i+7:i] ← 0;

FI

FI;

ENDFOR;

DEST[MAX\_VL-1:VL] ← 0;

**VPSUBUSW (EVEX encoded versions)**

(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] ← SaturateToUnsignedWord (SRC1[i+15:i] - SRC2[i+15:i])

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+15:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[i+15:i] ← 0;

FI

FI;

ENDFOR;

DEST[MAX\_VL-1:VL] ← 0;

**VPSUBUSB (VEX.256 encoded version)**

DEST[7:0] ← SaturateToUnsignedByte (SRC1[7:0] - SRC2[7:0]);

(\* Repeat subtract operation for 2nd through 31st bytes \*)

DEST[255:148] ← SaturateToUnsignedByte (SRC1[255:248] - SRC2[255:248]);

DEST[MAX\_VL-1:256] ← 0;

**VPSUBUSB (VEX.128 encoded version)**

DEST[7:0] ← SaturateToUnsignedByte (SRC1[7:0] - SRC2[7:0]);

(\* Repeat subtract operation for 2nd through 14th bytes \*)

DEST[127:120] ← SaturateToUnsignedByte (SRC1[127:120] - SRC2[127:120]);

DEST[MAX\_VL-1:128] ← 0

**PSUBUSB (128-bit Legacy SSE Version)**

DEST[7:0] ← SaturateToUnsignedByte (DEST[7:0] - SRC[7:0]);

(\* Repeat subtract operation for 2nd through 14th bytes \*)

DEST[127:120] ← SaturateToUnsignedByte (DEST[127:120] - SRC[127:120]);

DEST[MAX\_VL-1:128] (Unmodified)

**VPSUBUSW (VEX.256 encoded version)**

DEST[15:0] ← SaturateToUnsignedWord (SRC1[15:0] - SRC2[15:0]);  
 (\* Repeat subtract operation for 2nd through 15th words \*)  
 DEST[255:240] ← SaturateToUnsignedWord (SRC1[255:240] - SRC2[255:240]);  
 DEST[MAX\_VL-1:256] ← 0;

**VPSUBUSW (VEX.128 encoded version)**

DEST[15:0] ← SaturateToUnsignedWord (SRC1[15:0] - SRC2[15:0]);  
 (\* Repeat subtract operation for 2nd through 7th words \*)  
 DEST[127:112] ← SaturateToUnsignedWord (SRC1[127:112] - SRC2[127:112]);  
 DEST[MAX\_VL-1:128] ← 0

**PSUBUSW (128-bit Legacy SSE Version)**

DEST[15:0] ← SaturateToUnsignedWord (DEST[15:0] - SRC[15:0]);  
 (\* Repeat subtract operation for 2nd through 7th words \*)  
 DEST[127:112] ← SaturateToUnsignedWord (DEST[127:112] - SRC[127:112]);  
 DEST[MAX\_VL-1:128] (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalents**

VPSUBUSB \_\_m512i\_mm512\_subs\_epu8(\_\_m512i a, \_\_m512i b);  
 VPSUBUSB \_\_m512i\_mm512\_mask\_subs\_epu8(\_\_m512i s, \_\_mmask64 k, \_\_m512i a, \_\_m512i b);  
 VPSUBUSB \_\_m512i\_mm512\_maskz\_subs\_epu8(\_\_mmask64 k, \_\_m512i a, \_\_m512i b);  
 VPSUBUSB \_\_m256i\_mm256\_mask\_subs\_epu8(\_\_m256i s, \_\_mmask32 k, \_\_m256i a, \_\_m256i b);  
 VPSUBUSB \_\_m256i\_mm256\_maskz\_subs\_epu8(\_\_mmask32 k, \_\_m256i a, \_\_m256i b);  
 VPSUBUSB \_\_m128i\_mm\_mask\_subs\_epu8(\_\_m128i s, \_\_mmask16 k, \_\_m128i a, \_\_m128i b);  
 VPSUBUSB \_\_m128i\_mm\_maskz\_subs\_epu8(\_\_mmask16 k, \_\_m128i a, \_\_m128i b);  
 VPSUBUSW \_\_m512i\_mm512\_subs\_epu16(\_\_m512i a, \_\_m512i b);  
 VPSUBUSW \_\_m512i\_mm512\_mask\_subs\_epu16(\_\_m512i s, \_\_mmask32 k, \_\_m512i a, \_\_m512i b);  
 VPSUBUSW \_\_m512i\_mm512\_maskz\_subs\_epu16(\_\_mmask32 k, \_\_m512i a, \_\_m512i b);  
 VPSUBUSW \_\_m256i\_mm256\_mask\_subs\_epu16(\_\_m256i s, \_\_mmask16 k, \_\_m256i a, \_\_m256i b);  
 VPSUBUSW \_\_m256i\_mm256\_maskz\_subs\_epu16(\_\_mmask16 k, \_\_m256i a, \_\_m256i b);  
 VPSUBUSW \_\_m128i\_mm\_mask\_subs\_epu16(\_\_m128i s, \_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
 VPSUBUSW \_\_m128i\_mm\_maskz\_subs\_epu16(\_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
 PSUBUSB: \_\_m64\_mm\_subs\_pu8(\_\_m64 m1, \_\_m64 m2)  
 (V)PSUBUSB: \_\_m128i\_mm\_subs\_epu8(\_\_m128i m1, \_\_m128i m2)  
 VPSUBUSB: \_\_m256i\_mm256\_subs\_epu8(\_\_m256i m1, \_\_m256i m2)  
 PSUBUSW: \_\_m64\_mm\_subs\_pu16(\_\_m64 m1, \_\_m64 m2)  
 (V)PSUBUSW: \_\_m128i\_mm\_subs\_epu16(\_\_m128i m1, \_\_m128i m2)  
 VPSUBUSW: \_\_m256i\_mm256\_subs\_epu16(\_\_m256i m1, \_\_m256i m2)

**Flags Affected**

None.

**Numeric Exceptions**

None.

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.

## PTWRITE - Write Data to a Processor Trace Packet

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 REX.W OF AE /4 PTWRITE r64/m64	RM	V/N.E		Reads the data from r64/m64 to encode into a PTW packet if dependencies are met (see details below).
F3 OF AE /4 PTWRITE r32/m32	RM	V/V		Reads the data from r32/m32 to encode into a PTW packet if dependencies are met (see details below).

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:rm (r)	NA	NA	NA

### Description

This instruction reads data in the source operand and sends it to the Intel Processor Trace hardware to be encoded in a PTW packet if TriggerEn, ContextEn, FilterEn, and PTWEn are all set to 1. For more details on these values, see *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C, Section 36.2.3, "Power Event Tracing"*. The size of data is 64-bit if using REX.W in 64-bit mode, otherwise 32-bits of data are copied from the source operand.

Note: The instruction will #UD if prefix 66H is used.

### Operation

IF (IA32\_RTIT\_STATUS.TriggerEn & IA32\_RTIT\_STATUS.ContextEn & IA32\_RTIT\_STATUS.FilterEn & IA32\_RTIT\_CTL.PTWEn) = 1

PTW.PayloadBytes ← Encoded payload size;

PTW.IP ← IA32\_RTIT\_CTL.FUPonPTW

IF IA32\_RTIT\_CTL.FUPonPTW = 1

Insert FUP packet with IP of PTWRITE;

FI;

FI;

### Flags Affected

None.

### Other Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS or GS segments.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF (fault-code)	For a page fault.
#AC(0)	If an unaligned memory reference is made while the current privilege level is 3 and alignment checking is enabled.
#UD	If CPUID.(EAX=14H, ECX=0):EBX.PTWRITE [Bit 4] = 0. If LOCK prefix is used. If 66H prefix is used.

**Real-Address Mode Exceptions**

#GP(0)	If any part of the operand lies outside of the effective address space from 0 to 0FFFFH.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If CPUID.(EAX=14H, ECX=0):EBX.PTWRITE [Bit 4] = 0. If LOCK prefix is used. If 66H prefix is used.

**Virtual 8086 Mode Exceptions**

#GP(0)	If any part of the operand lies outside of the effective address space from 0 to 0FFFFH.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF (fault-code)	For a page fault.
#AC(0)	If an unaligned memory reference is made while alignment checking is enabled.
#UD	If CPUID.(EAX=14H, ECX=0):EBX.PTWRITE [Bit 4] = 0. If LOCK prefix is used. If 66H prefix is used.

**Compatibility Mode Exceptions**

Same exceptions as in Protected Mode.

**64-Bit Mode Exceptions**

#GP(0)	If the memory address is in a non-canonical form.
#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#PF (fault-code)	For a page fault.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If CPUID.(EAX=14H, ECX=0):EBX.PTWRITE [Bit 4] = 0. If LOCK prefix is used. If 66H prefix is used.

## PUNPCKHBW/PUNPCKHWD/PUNPCKHDQ/PUNPCKHQDQ— Unpack High Data

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F 68 /r <sup>1</sup> PUNPCKHBW <i>mm, mm/m64</i>	RM	V/V	MMX	Unpack and interleave high-order bytes from <i>mm</i> and <i>mm/m64</i> into <i>mm</i> .
66 0F 68 /r PUNPCKHBW <i>xmm1, xmm2/m128</i>	RM	V/V	SSE2	Unpack and interleave high-order bytes from <i>xmm1</i> and <i>xmm2/m128</i> into <i>xmm1</i> .
0F 69 /r <sup>1</sup> PUNPCKHWD <i>mm, mm/m64</i>	RM	V/V	MMX	Unpack and interleave high-order words from <i>mm</i> and <i>mm/m64</i> into <i>mm</i> .
66 0F 69 /r PUNPCKHWD <i>xmm1, xmm2/m128</i>	RM	V/V	SSE2	Unpack and interleave high-order words from <i>xmm1</i> and <i>xmm2/m128</i> into <i>xmm1</i> .
0F 6A /r <sup>1</sup> PUNPCKHDQ <i>mm, mm/m64</i>	RM	V/V	MMX	Unpack and interleave high-order doublewords from <i>mm</i> and <i>mm/m64</i> into <i>mm</i> .
66 0F 6A /r PUNPCKHDQ <i>xmm1, xmm2/m128</i>	RM	V/V	SSE2	Unpack and interleave high-order doublewords from <i>xmm1</i> and <i>xmm2/m128</i> into <i>xmm1</i> .
66 0F 6D /r PUNPCKHQDQ <i>xmm1, xmm2/m128</i>	RM	V/V	SSE2	Unpack and interleave high-order quadwords from <i>xmm1</i> and <i>xmm2/m128</i> into <i>xmm1</i> .
VEX.NDS.128.66.0F.WIG 68/r VPUNPCKHBW <i>xmm1, xmm2, xmm3/m128</i>	RVM	V/V	AVX	Interleave high-order bytes from <i>xmm2</i> and <i>xmm3/m128</i> into <i>xmm1</i> .
VEX.NDS.128.66.0F.WIG 69/r VPUNPCKHWD <i>xmm1, xmm2, xmm3/m128</i>	RVM	V/V	AVX	Interleave high-order words from <i>xmm2</i> and <i>xmm3/m128</i> into <i>xmm1</i> .
VEX.NDS.128.66.0F.WIG 6A/r VPUNPCKHDQ <i>xmm1, xmm2, xmm3/m128</i>	RVM	V/V	AVX	Interleave high-order doublewords from <i>xmm2</i> and <i>xmm3/m128</i> into <i>xmm1</i> .
VEX.NDS.128.66.0F.WIG 6D/r VPUNPCKHQDQ <i>xmm1, xmm2, xmm3/m128</i>	RVM	V/V	AVX	Interleave high-order quadword from <i>xmm2</i> and <i>xmm3/m128</i> into <i>xmm1</i> register.
VEX.NDS.256.66.0F.WIG 68 /r VPUNPCKHBW <i>ymm1, ymm2, ymm3/m256</i>	RVM	V/V	AVX2	Interleave high-order bytes from <i>ymm2</i> and <i>ymm3/m256</i> into <i>ymm1</i> register.
VEX.NDS.256.66.0F.WIG 69 /r VPUNPCKHWD <i>ymm1, ymm2, ymm3/m256</i>	RVM	V/V	AVX2	Interleave high-order words from <i>ymm2</i> and <i>ymm3/m256</i> into <i>ymm1</i> register.
VEX.NDS.256.66.0F.WIG 6A /r VPUNPCKHDQ <i>ymm1, ymm2, ymm3/m256</i>	RVM	V/V	AVX2	Interleave high-order doublewords from <i>ymm2</i> and <i>ymm3/m256</i> into <i>ymm1</i> register.
VEX.NDS.256.66.0F.WIG 6D /r VPUNPCKHQDQ <i>ymm1, ymm2, ymm3/m256</i>	RVM	V/V	AVX2	Interleave high-order quadword from <i>ymm2</i> and <i>ymm3/m256</i> into <i>ymm1</i> register.
EVEX.NDS.128.66.0F.WIG 68 /r VPUNPCKHBW <i>xmm1 {k1}{z}, xmm2, xmm3/m128</i>	FVM	V/V	AVX512VL AVX512BW	Interleave high-order bytes from <i>xmm2</i> and <i>xmm3/m128</i> into <i>xmm1</i> register using <i>k1</i> write mask.
EVEX.NDS.128.66.0F.WIG 69 /r VPUNPCKHWD <i>xmm1 {k1}{z}, xmm2, xmm3/m128</i>	FVM	V/V	AVX512VL AVX512BW	Interleave high-order words from <i>xmm2</i> and <i>xmm3/m128</i> into <i>xmm1</i> register using <i>k1</i> write mask.
EVEX.NDS.128.66.0F.WO 6A /r VPUNPCKHDQ <i>xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst</i>	FV	V/V	AVX512VL AVX512F	Interleave high-order doublewords from <i>xmm2</i> and <i>xmm3/m128/m32bcst</i> into <i>xmm1</i> register using <i>k1</i> write mask.

EVEX.NDS.128.66.0F.W1 6D /r VPUNPCKHQDQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	FV V/V	AVX512VL AVX512F	Interleave high-order quadword from xmm2 and xmm3/m128/m64bcst into xmm1 register using k1 write mask.
EVEX.NDS.256.66.0F.WIG 68 /r VPUNPCKHBW ymm1 {k1}{z}, ymm2, ymm3/m256	FVM V/V	AVX512VL AVX512BW	Interleave high-order bytes from ymm2 and ymm3/m256 into ymm1 register using k1 write mask.
EVEX.NDS.256.66.0F.WIG 69 /r VPUNPCKHWD ymm1 {k1}{z}, ymm2, ymm3/m256	FVM V/V	AVX512VL AVX512BW	Interleave high-order words from ymm2 and ymm3/m256 into ymm1 register using k1 write mask.
EVEX.NDS.256.66.0F.W0 6A /r VPUNPCKHDQ ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	FV V/V	AVX512VL AVX512F	Interleave high-order doublewords from ymm2 and ymm3/m256/m32bcst into ymm1 register using k1 write mask.
EVEX.NDS.256.66.0F.W1 6D /r VPUNPCKHQDQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	FV V/V	AVX512VL AVX512F	Interleave high-order quadword from ymm2 and ymm3/m256/m64bcst into ymm1 register using k1 write mask.
EVEX.NDS.512.66.0F.WIG 68/r VPUNPCKHBW zmm1 {k1}{z}, zmm2, zmm3/m512	FVM V/V	AVX512BW	Interleave high-order bytes from zmm2 and zmm3/m512 into zmm1 register.
EVEX.NDS.512.66.0F.WIG 69/r VPUNPCKHWD zmm1 {k1}{z}, zmm2, zmm3/m512	FVM V/V	AVX512BW	Interleave high-order words from zmm2 and zmm3/m512 into zmm1 register.
EVEX.NDS.512.66.0F.W0 6A /r VPUNPCKHDQ zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	FV V/V	AVX512F	Interleave high-order doublewords from zmm2 and zmm3/m512/m32bcst into zmm1 register using k1 write mask.
EVEX.NDS.512.66.0F.W1 6D /r VPUNPCKHQDQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	FV V/V	AVX512F	Interleave high-order quadword from zmm2 and zmm3/m512/m64bcst into zmm1 register using k1 write mask.

**NOTES:**

1. See note in Section 2.4, “AVX and SSE Instruction Exception Specification” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FVM	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA
FV	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

**Description**

Unpacks and interleaves the high-order data elements (bytes, words, doublewords, or quadwords) of the destination operand (first operand) and source operand (second operand) into the destination operand. Figure 4-16 shows the unpack operation for bytes in 64-bit operands. The low-order data elements are ignored.

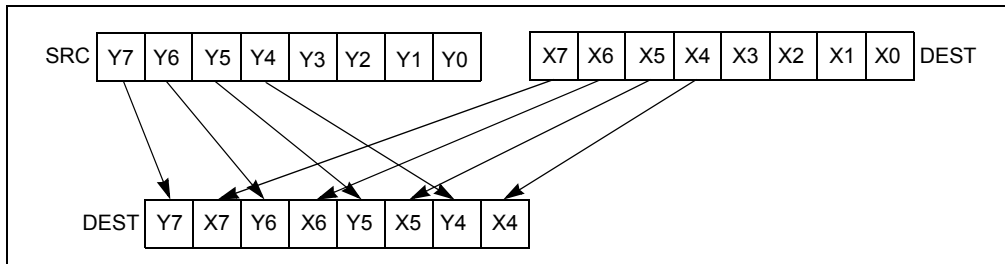


Figure 4-16. PUNPCKHBW Instruction Operation Using 64-bit Operands

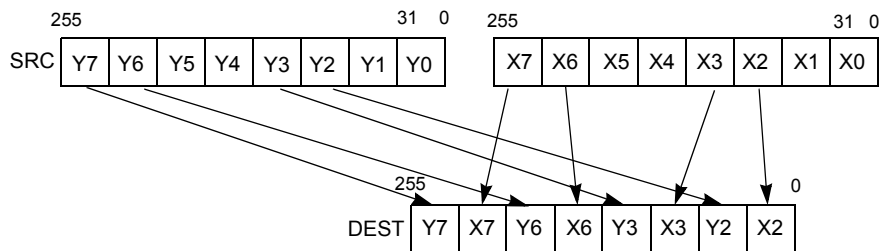


Figure 4-17. 256-bit VPUNPCKHDQ Instruction Operation

When the source data comes from a 64-bit memory operand, the full 64-bit operand is accessed from memory, but the instruction uses only the high-order 32 bits. When the source data comes from a 128-bit memory operand, an implementation may fetch only the appropriate 64 bits; however, alignment to a 16-byte boundary and normal segment checking will still be enforced.

The (V)PUNPCKHBW instruction interleaves the high-order bytes of the source and destination operands, the (V)PUNPCKHWD instruction interleaves the high-order words of the source and destination operands, the (V)PUNPCKHDQ instruction interleaves the high-order doubleword (or doublewords) of the source and destination operands, and the (V)PUNPCKHQDQ instruction interleaves the high-order quadwords of the source and destination operands.

These instructions can be used to convert bytes to words, words to doublewords, doublewords to quadwords, and quadwords to double quadwords, respectively, by placing all 0s in the source operand. Here, if the source operand contains all 0s, the result (stored in the destination operand) contains zero extensions of the high-order data elements from the original value in the destination operand. For example, with the (V)PUNPCKHBW instruction the high-order bytes are zero extended (that is, unpacked into unsigned word integers), and with the (V)PUNPCKHWD instruction, the high-order words are zero extended (unpacked into unsigned doubleword integers).

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE versions 64-bit operand: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand is an MMX technology register.

128-bit Legacy SSE versions: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded versions: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The second source operand is an YMM register or an 256-bit memory location. The first source operand and destination operands are YMM registers.



EVEX encoded VPUNPCKHDQ/QDQ: The second 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 32/64-bit memory location. The first source operand and destination operands are ZMM/YMM/XMM registers. The destination is conditionally updated with writemask k1.

EVEX encoded VPUNPCKHWD/BW: The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The first source operand and destination operands are ZMM/YMM/XMM registers. The destination is conditionally updated with writemask k1.

## Operation

### PUNPCKHBW instruction with 64-bit operands:

```
DEST[7:0] ← DEST[39:32];
DEST[15:8] ← SRC[39:32];
DEST[23:16] ← DEST[47:40];
DEST[31:24] ← SRC[47:40];
DEST[39:32] ← DEST[55:48];
DEST[47:40] ← SRC[55:48];
DEST[55:48] ← DEST[63:56];
DEST[63:56] ← SRC[63:56];
```

### PUNPCKHW instruction with 64-bit operands:

```
DEST[15:0] ← DEST[47:32];
DEST[31:16] ← SRC[47:32];
DEST[47:32] ← DEST[63:48];
DEST[63:48] ← SRC[63:48];
```

### PUNPCKHDQ instruction with 64-bit operands:

```
DEST[31:0] ← DEST[63:32];
DEST[63:32] ← SRC[63:32];
```

INTERLEAVE\_HIGH\_BYTES\_512b (SRC1, SRC2)

TMP\_DEST[255:0] ← INTERLEAVE\_HIGH\_BYTES\_256b(SRC1[255:0], SRC[255:0])

TMP\_DEST[511:256] ← INTERLEAVE\_HIGH\_BYTES\_256b(SRC1[511:256], SRC[511:256])

INTERLEAVE\_HIGH\_BYTES\_256b (SRC1, SRC2)

```
DEST[7:0] ← SRC1[71:64]
DEST[15:8] ← SRC2[71:64]
DEST[23:16] ← SRC1[79:72]
DEST[31:24] ← SRC2[79:72]
DEST[39:32] ← SRC1[87:80]
DEST[47:40] ← SRC2[87:80]
DEST[55:48] ← SRC1[95:88]
DEST[63:56] ← SRC2[95:88]
DEST[71:64] ← SRC1[103:96]
DEST[79:72] ← SRC2[103:96]
DEST[87:80] ← SRC1[111:104]
DEST[95:88] ← SRC2[111:104]
DEST[103:96] ← SRC1[119:112]
DEST[111:104] ← SRC2[119:112]
DEST[119:112] ← SRC1[127:120]
DEST[127:120] ← SRC2[127:120]
DEST[135:128] ← SRC1[199:192]
DEST[143:136] ← SRC2[199:192]
DEST[151:144] ← SRC1[207:200]
DEST[159:152] ← SRC2[207:200]
```

DEST[167:160] ← SRC1[215:208]  
 DEST[175:168] ← SRC2[215:208]  
 DEST[183:176] ← SRC1[223:216]  
 DEST[191:184] ← SRC2[223:216]  
 DEST[199:192] ← SRC1[231:224]  
 DEST[207:200] ← SRC2[231:224]  
 DEST[215:208] ← SRC1[239:232]  
 DEST[223:216] ← SRC2[239:232]  
 DEST[231:224] ← SRC1[247:240]  
 DEST[239:232] ← SRC2[247:240]  
 DEST[247:240] ← SRC1[255:248]  
 DEST[255:248] ← SRC2[255:248]

INTERLEAVE\_HIGH\_BYTES (SRC1, SRC2)

DEST[7:0] ← SRC1[71:64]  
 DEST[15:8] ← SRC2[71:64]  
 DEST[23:16] ← SRC1[79:72]  
 DEST[31:24] ← SRC2[79:72]  
 DEST[39:32] ← SRC1[87:80]  
 DEST[47:40] ← SRC2[87:80]  
 DEST[55:48] ← SRC1[95:88]  
 DEST[63:56] ← SRC2[95:88]  
 DEST[71:64] ← SRC1[103:96]  
 DEST[79:72] ← SRC2[103:96]  
 DEST[87:80] ← SRC1[111:104]  
 DEST[95:88] ← SRC2[111:104]  
 DEST[103:96] ← SRC1[119:112]  
 DEST[111:104] ← SRC2[119:112]  
 DEST[119:112] ← SRC1[127:120]  
 DEST[127:120] ← SRC2[127:120]

INTERLEAVE\_HIGH\_WORDS\_512b (SRC1, SRC2)

TMP\_DEST[255:0] ← INTERLEAVE\_HIGH\_WORDS\_256b(SRC1[255:0], SRC[255:0])  
 TMP\_DEST[511:256] ← INTERLEAVE\_HIGH\_WORDS\_256b(SRC1[511:256], SRC[511:256])

INTERLEAVE\_HIGH\_WORDS\_256b(SRC1, SRC2)

DEST[15:0] ← SRC1[79:64]  
 DEST[31:16] ← SRC2[79:64]  
 DEST[47:32] ← SRC1[95:80]  
 DEST[63:48] ← SRC2[95:80]  
 DEST[79:64] ← SRC1[111:96]  
 DEST[95:80] ← SRC2[111:96]  
 DEST[111:96] ← SRC1[127:112]  
 DEST[127:112] ← SRC2[127:112]  
 DEST[143:128] ← SRC1[207:192]  
 DEST[159:144] ← SRC2[207:192]  
 DEST[175:160] ← SRC1[223:208]  
 DEST[191:176] ← SRC2[223:208]  
 DEST[207:192] ← SRC1[239:224]  
 DEST[223:208] ← SRC2[239:224]  
 DEST[239:224] ← SRC1[255:240]  
 DEST[255:240] ← SRC2[255:240]

INTERLEAVE\_HIGH\_WORDS (SRC1, SRC2)

DEST[15:0] ← SRC1[79:64]  
 DEST[31:16] ← SRC2[79:64]  
 DEST[47:32] ← SRC1[95:80]  
 DEST[63:48] ← SRC2[95:80]  
 DEST[79:64] ← SRC1[111:96]  
 DEST[95:80] ← SRC2[111:96]  
 DEST[111:96] ← SRC1[127:112]  
 DEST[127:112] ← SRC2[127:112]

INTERLEAVE\_HIGH\_DWORDS\_512b (SRC1, SRC2)  
 TMP\_DEST[255:0] ← INTERLEAVE\_HIGH\_DWORDS\_256b(SRC1[255:0], SRC2[255:0])  
 TMP\_DEST[511:256] ← INTERLEAVE\_HIGH\_DWORDS\_256b(SRC1[511:256], SRC2[511:256])

INTERLEAVE\_HIGH\_DWORDS\_256b(SRC1, SRC2)  
 DEST[31:0] ← SRC1[95:64]  
 DEST[63:32] ← SRC2[95:64]  
 DEST[95:64] ← SRC1[127:96]  
 DEST[127:96] ← SRC2[127:96]  
 DEST[159:128] ← SRC1[223:192]  
 DEST[191:160] ← SRC2[223:192]  
 DEST[223:192] ← SRC1[255:224]  
 DEST[255:224] ← SRC2[255:224]

INTERLEAVE\_HIGH\_DWORDS(SRC1, SRC2)  
 DEST[31:0] ← SRC1[95:64]  
 DEST[63:32] ← SRC2[95:64]  
 DEST[95:64] ← SRC1[127:96]  
 DEST[127:96] ← SRC2[127:96]

INTERLEAVE\_HIGH\_QWORDS\_512b (SRC1, SRC2)  
 TMP\_DEST[255:0] ← INTERLEAVE\_HIGH\_QWORDS\_256b(SRC1[255:0], SRC2[255:0])  
 TMP\_DEST[511:256] ← INTERLEAVE\_HIGH\_QWORDS\_256b(SRC1[511:256], SRC2[511:256])

INTERLEAVE\_HIGH\_QWORDS\_256b(SRC1, SRC2)  
 DEST[63:0] ← SRC1[127:64]  
 DEST[127:64] ← SRC2[127:64]  
 DEST[191:128] ← SRC1[255:192]  
 DEST[255:192] ← SRC2[255:192]

INTERLEAVE\_HIGH\_QWORDS(SRC1, SRC2)  
 DEST[63:0] ← SRC1[127:64]  
 DEST[127:64] ← SRC2[127:64]

#### **PUNPCKHBW (128-bit Legacy SSE Version)**

DEST[127:0] ← INTERLEAVE\_HIGH\_BYTES(DEST, SRC)  
 DEST[255:127] (Unmodified)

#### **VPUNPCKHBW (VEX.128 encoded version)**

DEST[127:0] ← INTERLEAVE\_HIGH\_BYTES(SRC1, SRC2)  
 DEST[511:127] ← 0

#### **VPUNPCKHBW (VEX.256 encoded version)**

DEST[255:0] ← INTERLEAVE\_HIGH\_BYTES\_256b(SRC1, SRC2)  
 DEST[MAX\_VL-1:256] ← 0

**VPUNPCKHBW (EVEX encoded versions)**

(KL, VL) = (16, 128), (32, 256), (64, 512)

IF VL = 128

TMP\_DEST[VL-1:0] ← INTERLEAVE\_HIGH\_BYTES(SRC1[VL-1:0], SRC2[VL-1:0])

FI;

IF VL = 256

TMP\_DEST[VL-1:0] ← INTERLEAVE\_HIGH\_BYTES\_256b(SRC1[VL-1:0], SRC2[VL-1:0])

FI;

IF VL = 512

TMP\_DEST[VL-1:0] ← INTERLEAVE\_HIGH\_BYTES\_512b(SRC1[VL-1:0], SRC2[VL-1:0])

FI;

FOR j ← 0 TO KL-1

i ← j \* 8

IF k1[j] OR \*no writemask\*

THEN DEST[i+7:i] ← TMP\_DEST[i+7:i]

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+7:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[i+7:i] ← 0

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**PUNPCKHWD (128-bit Legacy SSE Version)**

DEST[127:0] ← INTERLEAVE\_HIGH\_WORDS(DEST, SRC)

DEST[255:127] (Unmodified)

**VPUNPCKHWD (VEX.128 encoded version)**

DEST[127:0] ← INTERLEAVE\_HIGH\_WORDS(SRC1, SRC2)

DEST[511:127] ← 0

**VPUNPCKHWD (VEX.256 encoded version)**

DEST[255:0] ← INTERLEAVE\_HIGH\_WORDS\_256b(SRC1, SRC2)

DEST[MAX\_VL-1:256] ← 0

VPUNPCKHWD (EVEX encoded versions)

(KL, VL) = (8, 128), (16, 256), (32, 512)

IF VL = 128

TMP\_DEST[VL-1:0] ← INTERLEAVE\_HIGH\_WORDS(SRC1[VL-1:0], SRC2[VL-1:0])

FI;

IF VL = 256

TMP\_DEST[VL-1:0] ← INTERLEAVE\_HIGH\_WORDS\_256b(SRC1[VL-1:0], SRC2[VL-1:0])

FI;

IF VL = 512

TMP\_DEST[VL-1:0] ← INTERLEAVE\_HIGH\_WORDS\_512b(SRC1[VL-1:0], SRC2[VL-1:0])

FI;

FOR j ← 0 TO KL-1

i ← j \* 16

IF k1[j] OR \*no writemask\*

THEN DEST[i+15:i] ← TMP\_DEST[i+15:i]

```

    ELSE
        IF *merging-masking*           ; merging-masking
            THEN *DEST[j+15:i] remains unchanged*
        ELSE *zeroing-masking*         ; zeroing-masking
            DEST[j+15:i] ← 0
        FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**PUNPCKHDQ (128-bit Legacy SSE Version)**

```

DEST[127:0] ← INTERLEAVE_HIGH_DWORDS(DEST, SRC)
DEST[255:127] (Unmodified)

```

**VPUNPCKHDQ (VEX.128 encoded version)**

```

DEST[127:0] ← INTERLEAVE_HIGH_DWORDS(SRC1, SRC2)
DEST[511:127] ← 0

```

**VPUNPCKHDQ (VEX.256 encoded version)**

```

DEST[255:0] ← INTERLEAVE_HIGH_DWORDS_256b(SRC1, SRC2)
DEST[MAX_VL-1:256] ← 0

```

**VPUNPCKHDQ (EVEX.512 encoded version)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

```

FOR j ← 0 TO KL-1
    i ← j * 32
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
        THEN TMP_SRC2[j+31:i] ← SRC2[31:0]
        ELSE TMP_SRC2[j+31:i] ← SRC2[j+31:i]
    FI;
ENDFOR;
IF VL = 128
    TMP_DEST[VL-1:0] ← INTERLEAVE_HIGH_DWORDS(SRC1[VL-1:0], TMP_SRC2[VL-1:0])
FI;
IF VL = 256
    TMP_DEST[VL-1:0] ← INTERLEAVE_HIGH_DWORDS_256b(SRC1[VL-1:0], TMP_SRC2[VL-1:0])
FI;
IF VL = 512
    TMP_DEST[VL-1:0] ← INTERLEAVE_HIGH_DWORDS_512b(SRC1[VL-1:0], TMP_SRC2[VL-1:0])
FI;

```

```

FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[j+31:i] ← TMP_DEST[j+31:i]
    ELSE
        IF *merging-masking*           ; merging-masking
            THEN *DEST[j+31:i] remains unchanged*
        ELSE *zeroing-masking*         ; zeroing-masking
            DEST[j+31:i] ← 0
        FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**PUNPCKHQDQ (128-bit Legacy SSE Version)**

DEST[127:0] ← INTERLEAVE\_HIGH\_QWORDS(DEST, SRC)  
 DEST[MAX\_VL-1:128] (Unmodified)

**VPUNPCKHQDQ (VEX.128 encoded version)**

DEST[127:0] ← INTERLEAVE\_HIGH\_QWORDS(SRC1, SRC2)  
 DEST[MAX\_VL-1:128] ← 0

**VPUNPCKHQDQ (VEX.256 encoded version)**

DEST[255:0] ← INTERLEAVE\_HIGH\_QWORDS\_256b(SRC1, SRC2)  
 DEST[MAX\_VL-1:256] ← 0

**VPUNPCKHQDQ (EVEX encoded versions)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

  i ← j \* 64

  IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

    THEN TMP\_SRC2[i+63:i] ← SRC2[63:0]

    ELSE TMP\_SRC2[i+63:i] ← SRC2[i+63:i]

  FI;

ENDFOR;

IF VL = 128

  TMP\_DEST[VL-1:0] ← INTERLEAVE\_HIGH\_QWORDS(SRC1[VL-1:0], TMP\_SRC2[VL-1:0])

FI;

IF VL = 256

  TMP\_DEST[VL-1:0] ← INTERLEAVE\_HIGH\_QWORDS\_256b(SRC1[VL-1:0], TMP\_SRC2[VL-1:0])

FI;

IF VL = 512

  TMP\_DEST[VL-1:0] ← INTERLEAVE\_HIGH\_QWORDS\_512b(SRC1[VL-1:0], TMP\_SRC2[VL-1:0])

FI;

FOR j ← 0 TO KL-1

  i ← j \* 64

  IF k1[j] OR \*no writemask\*

    THEN DEST[i+63:i] ← TMP\_DEST[i+63:i]

    ELSE

      IF \*merging-masking\* ; merging-masking

        THEN \*DEST[i+63:i] remains unchanged\*

        ELSE \*zeroing-masking\* ; zeroing-masking

          DEST[i+63:i] ← 0

    FI

  FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalents**

VPUNPCKHBW \_\_m512i \_\_mm512\_unpackhi\_epi8(\_\_m512i a, \_\_m512i b);

VPUNPCKHBW \_\_m512i \_\_mm512\_mask\_unpackhi\_epi8(\_\_m512i s, \_\_mmask64 k, \_\_m512i a, \_\_m512i b);

VPUNPCKHBW \_\_m512i \_\_mm512\_maskz\_unpackhi\_epi8(\_\_mmask64 k, \_\_m512i a, \_\_m512i b);

VPUNPCKHBW \_\_m256i \_\_mm256\_mask\_unpackhi\_epi8(\_\_m256i s, \_\_mmask32 k, \_\_m256i a, \_\_m256i b);

VPUNPCKHBW \_\_m256i \_\_mm256\_maskz\_unpackhi\_epi8(\_\_mmask32 k, \_\_m256i a, \_\_m256i b);

VPUNPCKHBW \_\_m128i \_\_mm\_mask\_unpackhi\_epi8(v s, \_\_mmask16 k, \_\_m128i a, \_\_m128i b);

VPUNPCKHBW \_\_m128i \_\_mm\_maskz\_unpackhi\_epi8(\_\_mmask16 k, \_\_m128i a, \_\_m128i b);

VPUNPCKHWD \_\_m512i \_\_mm512\_unpackhi\_epi16(\_\_m512i a, \_\_m512i b);  
 VPUNPCKHWD \_\_m512i \_\_mm512\_mask\_unpackhi\_epi16(\_\_m512i s, \_\_mmask32 k, \_\_m512i a, \_\_m512i b);  
 VPUNPCKHWD \_\_m512i \_\_mm512\_maskz\_unpackhi\_epi16(\_\_mmask32 k, \_\_m512i a, \_\_m512i b);  
 VPUNPCKHWD \_\_m256i \_\_mm256\_mask\_unpackhi\_epi16(\_\_m256i s, \_\_mmask16 k, \_\_m256i a, \_\_m256i b);  
 VPUNPCKHWD \_\_m256i \_\_mm256\_maskz\_unpackhi\_epi16(\_\_mmask16 k, \_\_m256i a, \_\_m256i b);  
 VPUNPCKHWD \_\_m128i \_\_mm\_mask\_unpackhi\_epi16(v s, \_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
 VPUNPCKHWD \_\_m128i \_\_mm\_maskz\_unpackhi\_epi16(\_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
 VPUNPCKHDQ \_\_m512i \_\_mm512\_unpackhi\_epi32(\_\_m512i a, \_\_m512i b);  
 VPUNPCKHDQ \_\_m512i \_\_mm512\_mask\_unpackhi\_epi32(\_\_m512i s, \_\_mmask16 k, \_\_m512i a, \_\_m512i b);  
 VPUNPCKHDQ \_\_m512i \_\_mm512\_maskz\_unpackhi\_epi32(\_\_mmask16 k, \_\_m512i a, \_\_m512i b);  
 VPUNPCKHDQ \_\_m256i \_\_mm256\_mask\_unpackhi\_epi32(\_\_m512i s, \_\_mmask8 k, \_\_m512i a, \_\_m512i b);  
 VPUNPCKHDQ \_\_m256i \_\_mm256\_maskz\_unpackhi\_epi32(\_\_mmask8 k, \_\_m512i a, \_\_m512i b);  
 VPUNPCKHDQ \_\_m128i \_\_mm\_mask\_unpackhi\_epi32(\_\_m512i s, \_\_mmask8 k, \_\_m512i a, \_\_m512i b);  
 VPUNPCKHDQ \_\_m128i \_\_mm\_maskz\_unpackhi\_epi32(\_\_mmask8 k, \_\_m512i a, \_\_m512i b);  
 VPUNPCKHQDQ \_\_m512i \_\_mm512\_unpackhi\_epi64(\_\_m512i a, \_\_m512i b);  
 VPUNPCKHQDQ \_\_m512i \_\_mm512\_mask\_unpackhi\_epi64(\_\_m512i s, \_\_mmask8 k, \_\_m512i a, \_\_m512i b);  
 VPUNPCKHQDQ \_\_m512i \_\_mm512\_maskz\_unpackhi\_epi64(\_\_mmask8 k, \_\_m512i a, \_\_m512i b);  
 VPUNPCKHQDQ \_\_m256i \_\_mm256\_mask\_unpackhi\_epi64(\_\_m512i s, \_\_mmask8 k, \_\_m512i a, \_\_m512i b);  
 VPUNPCKHQDQ \_\_m256i \_\_mm256\_maskz\_unpackhi\_epi64(\_\_mmask8 k, \_\_m512i a, \_\_m512i b);  
 VPUNPCKHQDQ \_\_m128i \_\_mm\_mask\_unpackhi\_epi64(\_\_m512i s, \_\_mmask8 k, \_\_m512i a, \_\_m512i b);  
 VPUNPCKHQDQ \_\_m128i \_\_mm\_maskz\_unpackhi\_epi64(\_\_mmask8 k, \_\_m512i a, \_\_m512i b);  
 PUNPCKHBW: \_\_m64 \_\_mm\_unpackhi\_pi8(\_\_m64 m1, \_\_m64 m2)  
 (V)PUNPCKHBW: \_\_m128i \_\_mm\_unpackhi\_epi8(\_\_m128i m1, \_\_m128i m2)  
 VPUNPCKHBW: \_\_m256i \_\_mm256\_unpackhi\_epi8(\_\_m256i m1, \_\_m256i m2)  
 PUNPCKHWD: \_\_m64 \_\_mm\_unpackhi\_pi16(\_\_m64 m1, \_\_m64 m2)  
 (V)PUNPCKHWD: \_\_m128i \_\_mm\_unpackhi\_epi16(\_\_m128i m1, \_\_m128i m2)  
 VPUNPCKHWD: \_\_m256i \_\_mm256\_unpackhi\_epi16(\_\_m256i m1, \_\_m256i m2)  
 PUNPCKHDQ: \_\_m64 \_\_mm\_unpackhi\_pi32(\_\_m64 m1, \_\_m64 m2)  
 (V)PUNPCKHDQ: \_\_m128i \_\_mm\_unpackhi\_epi32(\_\_m128i m1, \_\_m128i m2)  
 VPUNPCKHDQ: \_\_m256i \_\_mm256\_unpackhi\_epi32(\_\_m256i m1, \_\_m256i m2)  
 (V)PUNPCKHQDQ: \_\_m128i \_\_mm\_unpackhi\_epi64 (\_\_m128i a, \_\_m128i b)  
 VPUNPCKHQDQ: \_\_m256i \_\_mm256\_unpackhi\_epi64 (\_\_m256i a, \_\_m256i b)

### Flags Affected

None.

### Numeric Exceptions

None.

### Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded VPUNPCKHQDQ/QDQ, see Exceptions Type E4NF.

EVEX-encoded VPUNPCKHBW/WD, see Exceptions Type E4NF.nb.

## PUNPCKLBW/PUNPCKLWD/PUNPCKLDQ/PUNPCKLQDQ—Unpack Low Data

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
OF 60 /r <sup>1</sup> PUNPCKLBW <i>mm</i> , <i>mm/m32</i>	RM	V/V	MMX	Interleave low-order bytes from <i>mm</i> and <i>mm/m32</i> into <i>mm</i> .
66 OF 60 /r PUNPCKLBW <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE2	Interleave low-order bytes from <i>xmm1</i> and <i>xmm2/m128</i> into <i>xmm1</i> .
OF 61 /r <sup>1</sup> PUNPCKLWD <i>mm</i> , <i>mm/m32</i>	RM	V/V	MMX	Interleave low-order words from <i>mm</i> and <i>mm/m32</i> into <i>mm</i> .
66 OF 61 /r PUNPCKLWD <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE2	Interleave low-order words from <i>xmm1</i> and <i>xmm2/m128</i> into <i>xmm1</i> .
OF 62 /r <sup>1</sup> PUNPCKLDQ <i>mm</i> , <i>mm/m32</i>	RM	V/V	MMX	Interleave low-order doublewords from <i>mm</i> and <i>mm/m32</i> into <i>mm</i> .
66 OF 62 /r PUNPCKLDQ <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE2	Interleave low-order doublewords from <i>xmm1</i> and <i>xmm2/m128</i> into <i>xmm1</i> .
66 OF 6C /r PUNPCKLQDQ <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE2	Interleave low-order quadword from <i>xmm1</i> and <i>xmm2/m128</i> into <i>xmm1</i> register.
VEX.NDS.128.66.OF.WIG 60/r VPUNPCKLBW <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Interleave low-order bytes from <i>xmm2</i> and <i>xmm3/m128</i> into <i>xmm1</i> .
VEX.NDS.128.66.OF.WIG 61/r VPUNPCKLWD <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Interleave low-order words from <i>xmm2</i> and <i>xmm3/m128</i> into <i>xmm1</i> .
VEX.NDS.128.66.OF.WIG 62/r VPUNPCKLDQ <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Interleave low-order doublewords from <i>xmm2</i> and <i>xmm3/m128</i> into <i>xmm1</i> .
VEX.NDS.128.66.OF.WIG 6C/r VPUNPCKLQDQ <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Interleave low-order quadword from <i>xmm2</i> and <i>xmm3/m128</i> into <i>xmm1</i> register.
VEX.NDS.256.66.OF.WIG 60 /r VPUNPCKLBW <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	RVM	V/V	AVX2	Interleave low-order bytes from <i>ymm2</i> and <i>ymm3/m256</i> into <i>ymm1</i> register.
VEX.NDS.256.66.OF.WIG 61 /r VPUNPCKLWD <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	RVM	V/V	AVX2	Interleave low-order words from <i>ymm2</i> and <i>ymm3/m256</i> into <i>ymm1</i> register.
VEX.NDS.256.66.OF.WIG 62 /r VPUNPCKLDQ <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	RVM	V/V	AVX2	Interleave low-order doublewords from <i>ymm2</i> and <i>ymm3/m256</i> into <i>ymm1</i> register.
VEX.NDS.256.66.OF.WIG 6C /r VPUNPCKLQDQ <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	RVM	V/V	AVX2	Interleave low-order quadword from <i>ymm2</i> and <i>ymm3/m256</i> into <i>ymm1</i> register.
EVEX.NDS.128.66.OF.WIG 60 /r VPUNPCKLBW <i>xmm1</i> {k1}{z}, <i>xmm2</i> , <i>xmm3/m128</i>	FVM	V/V	AVX512VL AVX512BW	Interleave low-order bytes from <i>xmm2</i> and <i>xmm3/m128</i> into <i>xmm1</i> register subject to write mask k1.
EVEX.NDS.128.66.OF.WIG 61 /r VPUNPCKLWD <i>xmm1</i> {k1}{z}, <i>xmm2</i> , <i>xmm3/m128</i>	FVM	V/V	AVX512VL AVX512BW	Interleave low-order words from <i>xmm2</i> and <i>xmm3/m128</i> into <i>xmm1</i> register subject to write mask k1.
EVEX.NDS.128.66.OF.WO 62 /r VPUNPCKLDQ <i>xmm1</i> {k1}{z}, <i>xmm2</i> , <i>xmm3/m128/m32bcst</i>	FV	V/V	AVX512VL AVX512F	Interleave low-order doublewords from <i>xmm2</i> and <i>xmm3/m128/m32bcst</i> into <i>xmm1</i> register subject to write mask k1.
EVEX.NDS.128.66.OF.W1 6C /r VPUNPCKLQDQ <i>xmm1</i> {k1}{z}, <i>xmm2</i> , <i>xmm3/m128/m64bcst</i>	FV	V/V	AVX512VL AVX512F	Interleave low-order quadword from <i>zmm2</i> and <i>zmm3/m512/m64bcst</i> into <i>zmm1</i> register subject to write mask k1.



EVEX.NDS.256.66.0F.WIG 60 /r VPUNPCKLBW ymm1 {k1}{z}, ymm2, ymm3/m256	FVM V/V	AVX512VL AVX512BW	Interleave low-order bytes from ymm2 and ymm3/m256 into ymm1 register subject to write mask k1.
EVEX.NDS.256.66.0F.WIG 61 /r VPUNPCKLWD ymm1 {k1}{z}, ymm2, ymm3/m256	FVM V/V	AVX512VL AVX512BW	Interleave low-order words from ymm2 and ymm3/m256 into ymm1 register subject to write mask k1.
EVEX.NDS.256.66.0F.W0 62 /r VPUNPCKLDQ ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	FV V/V	AVX512VL AVX512F	Interleave low-order doublewords from ymm2 and ymm3/m256/m32bcst into ymm1 register subject to write mask k1.
EVEX.NDS.256.66.0F.W1 6C /r VPUNPCKLQDQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	FV V/V	AVX512VL AVX512F	Interleave low-order quadword from ymm2 and ymm3/m256/m64bcst into ymm1 register subject to write mask k1.
EVEX.NDS.512.66.0F.WIG 60/r VPUNPCKLBW zmm1 {k1}{z}, zmm2, zmm3/m512	FVM V/V	AVX512BW	Interleave low-order bytes from zmm2 and zmm3/m512 into zmm1 register subject to write mask k1.
EVEX.NDS.512.66.0F.WIG 61/r VPUNPCKLWD zmm1 {k1}{z}, zmm2, zmm3/m512	FVM V/V	AVX512BW	Interleave low-order words from zmm2 and zmm3/m512 into zmm1 register subject to write mask k1.
EVEX.NDS.512.66.0F.W0 62 /r VPUNPCKLDQ zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	FV V/V	AVX512F	Interleave low-order doublewords from zmm2 and zmm3/m512/m32bcst into zmm1 register subject to write mask k1.
EVEX.NDS.512.66.0F.W1 6C /r VPUNPCKLQDQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	FV V/V	AVX512F	Interleave low-order quadword from zmm2 and zmm3/m512/m64bcst into zmm1 register subject to write mask k1.

**NOTES:**

1. See note in Section 2.4, “AVX and SSE Instruction Exception Specification” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FVM	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA
FV	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

**Description**

Unpacks and interleaves the low-order data elements (bytes, words, doublewords, and quadwords) of the destination operand (first operand) and source operand (second operand) into the destination operand. (Figure 4-18 shows the unpack operation for bytes in 64-bit operands.). The high-order data elements are ignored.

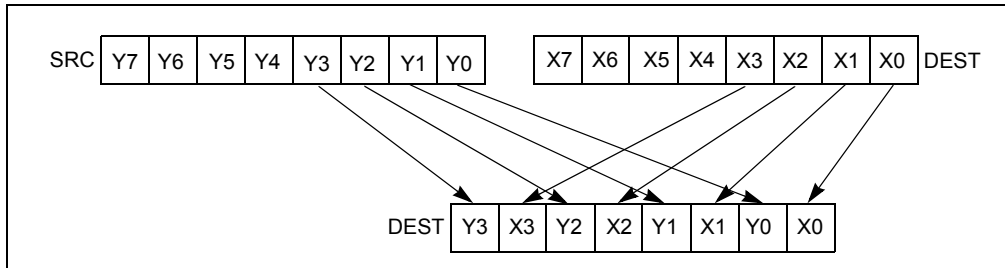


Figure 4-18. PUNPCKLBW Instruction Operation Using 64-bit Operands

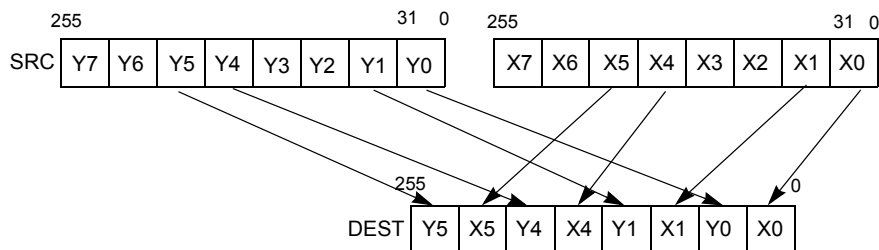


Figure 4-19. 256-bit VPUNPCKLDQ Instruction Operation

When the source data comes from a 128-bit memory operand, an implementation may fetch only the appropriate 64 bits; however, alignment to a 16-byte boundary and normal segment checking will still be enforced.

The (V)PUNPCKLBW instruction interleaves the low-order bytes of the source and destination operands, the (V)PUNPCKLWD instruction interleaves the low-order words of the source and destination operands, the (V)PUNPCKLDQ instruction interleaves the low-order doubleword (or doublewords) of the source and destination operands, and the (V)PUNPCKLQDQ instruction interleaves the low-order quadwords of the source and destination operands.

These instructions can be used to convert bytes to words, words to doublewords, doublewords to quadwords, and quadwords to double quadwords, respectively, by placing all 0s in the source operand. Here, if the source operand contains all 0s, the result (stored in the destination operand) contains zero extensions of the high-order data elements from the original value in the destination operand. For example, with the (V)PUNPCKLBW instruction the high-order bytes are zero extended (that is, unpacked into unsigned word integers), and with the (V)PUNPCKLWD instruction, the high-order words are zero extended (unpacked into unsigned doubleword integers).

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE versions 64-bit operand: The source operand can be an MMX technology register or a 32-bit memory location. The destination operand is an MMX technology register.

128-bit Legacy SSE versions: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded versions: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The second source operand is an YMM register or an 256-bit memory location. The first source operand and destination operands are YMM registers. Bits (MAX\_VL-1:256) of the corresponding ZMM register are zeroed.

EVEX encoded VPUNPCKLDQ/QDQ: The second 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 32/64-bit memory location. The first source operand and destination operands are ZMM/YMM/XMM registers. The destination is conditionally updated with writemask k1.

EVEX encoded VPUNPCKLWD/BW: The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The first source operand and destination operands are ZMM/YMM/XMM registers. The destination is conditionally updated with writemask k1.

## Operation

### PUNPCKLBW instruction with 64-bit operands:

```
DEST[63:56] ← SRC[31:24];
DEST[55:48] ← DEST[31:24];
DEST[47:40] ← SRC[23:16];
DEST[39:32] ← DEST[23:16];
DEST[31:24] ← SRC[15:8];
DEST[23:16] ← DEST[15:8];
DEST[15:8] ← SRC[7:0];
DEST[7:0] ← DEST[7:0];
```

### PUNPCKLWD instruction with 64-bit operands:

```
DEST[63:48] ← SRC[31:16];
DEST[47:32] ← DEST[31:16];
DEST[31:16] ← SRC[15:0];
DEST[15:0] ← DEST[15:0];
```

### PUNPCKLDQ instruction with 64-bit operands:

```
DEST[63:32] ← SRC[31:0];
DEST[31:0] ← DEST[31:0];
```

INTERLEAVE\_BYTES\_512b(SRC1, SRC2)

TMP\_DEST[255:0] ← INTERLEAVE\_BYTES\_256b(SRC1[255:0], SRC[255:0])

TMP\_DEST[511:256] ← INTERLEAVE\_BYTES\_256b(SRC1[511:256], SRC[511:256])

INTERLEAVE\_BYTES\_256b(SRC1, SRC2)

DEST[7:0] ← SRC1[7:0]

DEST[15:8] ← SRC2[7:0]

DEST[23:16] ← SRC1[15:8]

DEST[31:24] ← SRC2[15:8]

DEST[39:32] ← SRC1[23:16]

DEST[47:40] ← SRC2[23:16]

DEST[55:48] ← SRC1[31:24]

DEST[63:56] ← SRC2[31:24]

DEST[71:64] ← SRC1[39:32]

DEST[79:72] ← SRC2[39:32]

DEST[87:80] ← SRC1[47:40]

DEST[95:88] ← SRC2[47:40]

DEST[103:96] ← SRC1[55:48]

DEST[111:104] ← SRC2[55:48]

DEST[119:112] ← SRC1[63:56]

DEST[127:120] ← SRC2[63:56]

DEST[135:128] ← SRC1[135:128]

DEST[143:136] ← SRC2[135:128]

DEST[151:144] ← SRC1[143:136]

DEST[159:152] ← SRC2[143:136]

DEST[167:160] ← SRC1[151:144]

DEST[175:168] ← SRC2[151:144]  
 DEST[183:176] ← SRC1[159:152]  
 DEST[191:184] ← SRC2[159:152]  
 DEST[199:192] ← SRC1[167:160]  
 DEST[207:200] ← SRC2[167:160]  
 DEST[215:208] ← SRC1[175:168]  
 DEST[223:216] ← SRC2[175:168]  
 DEST[231:224] ← SRC1[183:176]  
 DEST[239:232] ← SRC2[183:176]  
 DEST[247:240] ← SRC1[191:184]  
 DEST[255:248] ← SRC2[191:184]

INTERLEAVE\_BYTES (SRC1, SRC2)

DEST[7:0] ← SRC1[7:0]  
 DEST[15:8] ← SRC2[7:0]  
 DEST[23:16] ← SRC2[15:8]  
 DEST[31:24] ← SRC2[15:8]  
 DEST[39:32] ← SRC1[23:16]  
 DEST[47:40] ← SRC2[23:16]  
 DEST[55:48] ← SRC1[31:24]  
 DEST[63:56] ← SRC2[31:24]  
 DEST[71:64] ← SRC1[39:32]  
 DEST[79:72] ← SRC2[39:32]  
 DEST[87:80] ← SRC1[47:40]  
 DEST[95:88] ← SRC2[47:40]  
 DEST[103:96] ← SRC1[55:48]  
 DEST[111:104] ← SRC2[55:48]  
 DEST[119:112] ← SRC1[63:56]  
 DEST[127:120] ← SRC2[63:56]

INTERLEAVE\_WORDS\_512b (SRC1, SRC2)

TMP\_DEST[255:0] ← INTERLEAVE\_WORDS\_256b(SRC1[255:0], SRC[255:0])  
 TMP\_DEST[511:256] ← INTERLEAVE\_WORDS\_256b(SRC1[511:256], SRC[511:256])

INTERLEAVE\_WORDS\_256b(SRC1, SRC2)

DEST[15:0] ← SRC1[15:0]  
 DEST[31:16] ← SRC2[15:0]  
 DEST[47:32] ← SRC1[31:16]  
 DEST[63:48] ← SRC2[31:16]  
 DEST[79:64] ← SRC1[47:32]  
 DEST[95:80] ← SRC2[47:32]  
 DEST[111:96] ← SRC1[63:48]  
 DEST[127:112] ← SRC2[63:48]  
 DEST[143:128] ← SRC1[143:128]  
 DEST[159:144] ← SRC2[143:128]  
 DEST[175:160] ← SRC1[159:144]  
 DEST[191:176] ← SRC2[159:144]  
 DEST[207:192] ← SRC1[175:160]  
 DEST[223:208] ← SRC2[175:160]  
 DEST[239:224] ← SRC1[191:176]  
 DEST[255:240] ← SRC2[191:176]

INTERLEAVE\_WORDS (SRC1, SRC2)

DEST[15:0] ← SRC1[15:0]

DEST[31:16] ← SRC2[15:0]  
 DEST[47:32] ← SRC1[31:16]  
 DEST[63:48] ← SRC2[31:16]  
 DEST[79:64] ← SRC1[47:32]  
 DEST[95:80] ← SRC2[47:32]  
 DEST[111:96] ← SRC1[63:48]  
 DEST[127:112] ← SRC2[63:48]

INTERLEAVE\_DWORDS\_512b (SRC1, SRC2)  
 TMP\_DEST[255:0] ← INTERLEAVE\_DWORDS\_256b(SRC1[255:0], SRC2[255:0])  
 TMP\_DEST[511:256] ← INTERLEAVE\_DWORDS\_256b(SRC1[511:256], SRC2[511:256])

INTERLEAVE\_DWORDS\_256b(SRC1, SRC2)  
 DEST[31:0] ← SRC1[31:0]  
 DEST[63:32] ← SRC2[31:0]  
 DEST[95:64] ← SRC1[63:32]  
 DEST[127:96] ← SRC2[63:32]  
 DEST[159:128] ← SRC1[159:128]  
 DEST[191:160] ← SRC2[159:128]  
 DEST[223:192] ← SRC1[191:160]  
 DEST[255:224] ← SRC2[191:160]

INTERLEAVE\_DWORDS(SRC1, SRC2)  
 DEST[31:0] ← SRC1[31:0]  
 DEST[63:32] ← SRC2[31:0]  
 DEST[95:64] ← SRC1[63:32]  
 DEST[127:96] ← SRC2[63:32]  
 INTERLEAVE\_QWORDS\_512b (SRC1, SRC2)  
 TMP\_DEST[255:0] ← INTERLEAVE\_QWORDS\_256b(SRC1[255:0], SRC2[255:0])  
 TMP\_DEST[511:256] ← INTERLEAVE\_QWORDS\_256b(SRC1[511:256], SRC2[511:256])

INTERLEAVE\_QWORDS\_256b(SRC1, SRC2)  
 DEST[63:0] ← SRC1[63:0]  
 DEST[127:64] ← SRC2[63:0]  
 DEST[191:128] ← SRC1[191:128]  
 DEST[255:192] ← SRC2[191:128]

INTERLEAVE\_QWORDS(SRC1, SRC2)  
 DEST[63:0] ← SRC1[63:0]  
 DEST[127:64] ← SRC2[63:0]

**PUNPCKLBW**

DEST[127:0] ← INTERLEAVE\_BYTES(DEST, SRC)  
 DEST[255:127] (Unmodified)

**VPUNPCKLBW (VEX.128 encoded instruction)**

DEST[127:0] ← INTERLEAVE\_BYTES(SRC1, SRC2)  
 DEST[511:127] ← 0

**VPUNPCKLBW (VEX.256 encoded instruction)**

DEST[255:0] ← INTERLEAVE\_BYTES\_256b(SRC1, SRC2)  
 DEST[MAX\_VL-1:256] ← 0

**VPUNPCKLBW (EVEX.512 encoded instruction)**

(KL, VL) = (16, 128), (32, 256), (64, 512)

IF VL = 128

TMP\_DEST[VL-1:0] ← INTERLEAVE\_BYTES(SRC1[VL-1:0], SRC2[VL-1:0])

FI;

IF VL = 256

TMP\_DEST[VL-1:0] ← INTERLEAVE\_BYTES\_256b(SRC1[VL-1:0], SRC2[VL-1:0])

FI;

IF VL = 512

TMP\_DEST[VL-1:0] ← INTERLEAVE\_BYTES\_512b(SRC1[VL-1:0], SRC2[VL-1:0])

FI;

FOR j ← 0 TO KL-1

i ← j \* 8

IF k1[j] OR \*no writemask\*

THEN DEST[i+7:i] ← TMP\_DEST[i+7:i]

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+7:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[i+7:i] ← 0

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

DEST[511:0] ← INTERLEAVE\_BYTES\_512b(SRC1, SRC2)

**PUNPCKLWD**

DEST[127:0] ← INTERLEAVE\_WORDS(DEST, SRC)

DEST[255:127] (Unmodified)

**VPUNPCKLWD (VEX.128 encoded instruction)**

DEST[127:0] ← INTERLEAVE\_WORDS(SRC1, SRC2)

DEST[511:127] ← 0

**VPUNPCKLWD (VEX.256 encoded instruction)**

DEST[255:0] ← INTERLEAVE\_WORDS\_256b(SRC1, SRC2)

DEST[MAX\_VL-1:256] ← 0

**VPUNPCKLWD (EVEX.512 encoded instruction)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

IF VL = 128

TMP\_DEST[VL-1:0] ← INTERLEAVE\_WORDS(SRC1[VL-1:0], SRC2[VL-1:0])

FI;

IF VL = 256

TMP\_DEST[VL-1:0] ← INTERLEAVE\_WORDS\_256b(SRC1[VL-1:0], SRC2[VL-1:0])

FI;

IF VL = 512

TMP\_DEST[VL-1:0] ← INTERLEAVE\_WORDS\_512b(SRC1[VL-1:0], SRC2[VL-1:0])

FI;

FOR j ← 0 TO KL-1

i ← j \* 16

IF k1[j] OR \*no writemask\*

```

    THEN DEST[j+15:i] ← TMP_DEST[j+15:i]
  ELSE
    IF *merging-masking*           ; merging-masking
      THEN *DEST[j+15:i] remains unchanged*
    ELSE *zeroing-masking*         ; zeroing-masking
      DEST[j+15:i] ← 0
    FI
  FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0
DEST[511:0] ← INTERLEAVE_WORDS_512b(SRC1, SRC2)

```

**PUNPCKLDQ**

```

DEST[127:0] ← INTERLEAVE_DWORDS(DEST, SRC)
DEST[MAX_VL-1:128] (Unmodified)

```

**VPUNPCKLDQ (VEX.128 encoded instruction)**

```

DEST[127:0] ← INTERLEAVE_DWORDS(SRC1, SRC2)
DEST[MAX_VL-1:128] ← 0

```

**VPUNPCKLDQ (VEX.256 encoded instruction)**

```

DEST[255:0] ← INTERLEAVE_DWORDS_256b(SRC1, SRC2)
DEST[MAX_VL-1:256] ← 0

```

**VPUNPCKLDQ (EVEX encoded instructions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 32
  IF (EVEX.b = 1) AND (SRC2 *is memory*)
    THEN TMP_SRC2[j+31:i] ← SRC2[31:0]
  ELSE TMP_SRC2[j+31:i] ← SRC2[j+31:i]
  FI;
ENDFOR;
IF VL = 128
  TMP_DEST[VL-1:0] ← INTERLEAVE_DWORDS(SRC1[VL-1:0], TMP_SRC2[VL-1:0])
FI;
IF VL = 256
  TMP_DEST[VL-1:0] ← INTERLEAVE_DWORDS_256b(SRC1[VL-1:0], TMP_SRC2[VL-1:0])
FI;
IF VL = 512
  TMP_DEST[VL-1:0] ← INTERLEAVE_DWORDS_512b(SRC1[VL-1:0], TMP_SRC2[VL-1:0])
FI;

```

```

FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[j+31:i] ← TMP_DEST[j+31:i]
  ELSE
    IF *merging-masking*           ; merging-masking
      THEN *DEST[j+31:i] remains unchanged*
    ELSE *zeroing-masking*         ; zeroing-masking
      DEST[j+31:i] ← 0
    FI
  FI;
ENDFOR;

```

```

ENDFOR
DEST511:0] ← INTERLEAVE_DWORDS_512b(SRC1, SRC2)
DEST[MAX_VL-1:VL] ← 0

```

**PUNPCKLQDQ**

```

DEST[127:0] ← INTERLEAVE_QWORDS(DEST, SRC)
DEST[MAX_VL-1:128] (Unmodified)

```

**VPUNPCKLQDQ (VEX.128 encoded instruction)**

```

DEST[127:0] ← INTERLEAVE_QWORDS(SRC1, SRC2)
DEST[MAX_VL-1:128] ← 0

```

**VPUNPCKLQDQ (VEX.256 encoded instruction)**

```

DEST[255:0] ← INTERLEAVE_QWORDS_256b(SRC1, SRC2)
DEST[MAX_VL-1:256] ← 0

```

**VPUNPCKLQDQ (EVEX encoded instructions)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

```

FOR j ← 0 TO KL-1
    i ← j * 64
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
        THEN TMP_SRC2[i+63:i] ← SRC2[63:0]
        ELSE TMP_SRC2[i+63:i] ← SRC2[i+63:i]
    FI;
ENDFOR;
IF VL = 128
    TMP_DEST[VL-1:0] ← INTERLEAVE_QWORDS(SRC1[VL-1:0], TMP_SRC2[VL-1:0])
FI;
IF VL = 256
    TMP_DEST[VL-1:0] ← INTERLEAVE_QWORDS_256b(SRC1[VL-1:0], TMP_SRC2[VL-1:0])
FI;
IF VL = 512
    TMP_DEST[VL-1:0] ← INTERLEAVE_QWORDS_512b(SRC1[VL-1:0], TMP_SRC2[VL-1:0])
FI;

FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
            ELSE *zeroing-masking* ; zeroing-masking
                DEST[i+63:i] ← 0
            FI
        FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalents**

```

VPUNPCKLBW __m512i __mm512_unpacklo_epi8(__m512i a, __m512i b);
VPUNPCKLBW __m512i __mm512_mask_unpacklo_epi8(__m512i s, __mmask64 k, __m512i a, __m512i b);
VPUNPCKLBW __m512i __mm512_maskz_unpacklo_epi8(__mmask64 k, __m512i a, __m512i b);
VPUNPCKLBW __m256i __mm256_mask_unpacklo_epi8(__m256i s, __mmask32 k, __m256i a, __m256i b);

```



VPUNPCKLBW \_\_m256i \_\_mm256\_maskz\_unpacklo\_epi8( \_\_mmask32 k, \_\_m256i a, \_\_m256i b);  
 VPUNPCKLBW \_\_m128i \_\_mm\_mask\_unpacklo\_epi8(v s, \_\_mmask16 k, \_\_m128i a, \_\_m128i b);  
 VPUNPCKLBW \_\_m128i \_\_mm\_maskz\_unpacklo\_epi8( \_\_mmask16 k, \_\_m128i a, \_\_m128i b);  
 VPUNPCKLWD \_\_m512i \_\_mm512\_unpacklo\_epi16(\_\_m512i a, \_\_m512i b);  
 VPUNPCKLWD \_\_m512i \_\_mm512\_mask\_unpacklo\_epi16(\_\_m512i s, \_\_mmask32 k, \_\_m512i a, \_\_m512i b);  
 VPUNPCKLWD \_\_m512i \_\_mm512\_maskz\_unpacklo\_epi16( \_\_mmask32 k, \_\_m512i a, \_\_m512i b);  
 VPUNPCKLWD \_\_m256i \_\_mm256\_mask\_unpacklo\_epi16(\_\_m256i s, \_\_mmask16 k, \_\_m256i a, \_\_m256i b);  
 VPUNPCKLWD \_\_m256i \_\_mm256\_maskz\_unpacklo\_epi16( \_\_mmask16 k, \_\_m256i a, \_\_m256i b);  
 VPUNPCKLWD \_\_m128i \_\_mm\_mask\_unpacklo\_epi16(v s, \_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
 VPUNPCKLWD \_\_m128i \_\_mm\_maskz\_unpacklo\_epi16( \_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
 VPUNPCKLDQ \_\_m512i \_\_mm512\_unpacklo\_epi32(\_\_m512i a, \_\_m512i b);  
 VPUNPCKLDQ \_\_m512i \_\_mm512\_mask\_unpacklo\_epi32(\_\_m512i s, \_\_mmask16 k, \_\_m512i a, \_\_m512i b);  
 VPUNPCKLDQ \_\_m512i \_\_mm512\_maskz\_unpacklo\_epi32( \_\_mmask16 k, \_\_m512i a, \_\_m512i b);  
 VPUNPCKLDQ \_\_m256i \_\_mm256\_mask\_unpacklo\_epi32(\_\_m256i s, \_\_mmask8 k, \_\_m256i a, \_\_m256i b);  
 VPUNPCKLDQ \_\_m256i \_\_mm256\_maskz\_unpacklo\_epi32( \_\_mmask8 k, \_\_m256i a, \_\_m256i b);  
 VPUNPCKLDQ \_\_m128i \_\_mm\_mask\_unpacklo\_epi32(v s, \_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
 VPUNPCKLDQ \_\_m128i \_\_mm\_maskz\_unpacklo\_epi32( \_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
 VPUNPCKLQDQ \_\_m512i \_\_mm512\_unpacklo\_epi64(\_\_m512i a, \_\_m512i b);  
 VPUNPCKLQDQ \_\_m512i \_\_mm512\_mask\_unpacklo\_epi64(\_\_m512i s, \_\_mmask8 k, \_\_m512i a, \_\_m512i b);  
 VPUNPCKLQDQ \_\_m512i \_\_mm512\_maskz\_unpacklo\_epi64( \_\_mmask8 k, \_\_m512i a, \_\_m512i b);  
 VPUNPCKLQDQ \_\_m256i \_\_mm256\_mask\_unpacklo\_epi64(\_\_m256i s, \_\_mmask8 k, \_\_m256i a, \_\_m256i b);  
 VPUNPCKLQDQ \_\_m256i \_\_mm256\_maskz\_unpacklo\_epi64( \_\_mmask8 k, \_\_m256i a, \_\_m256i b);  
 VPUNPCKLQDQ \_\_m128i \_\_mm\_mask\_unpacklo\_epi64(\_\_m128i s, \_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
 VPUNPCKLQDQ \_\_m128i \_\_mm\_maskz\_unpacklo\_epi64( \_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
 PUNPCKLBW: \_\_m64 \_\_mm\_unpacklo\_pi8 ( \_\_m64 m1, \_\_m64 m2)  
 (V)PUNPCKLBW: \_\_m128i \_\_mm\_unpacklo\_epi8 ( \_\_m128i m1, \_\_m128i m2)  
 VPUNPCKLBW: \_\_m256i \_\_mm256\_unpacklo\_epi8 ( \_\_m256i m1, \_\_m256i m2)  
 PUNPCKLWD: \_\_m64 \_\_mm\_unpacklo\_pi16 ( \_\_m64 m1, \_\_m64 m2)  
 (V)PUNPCKLWD: \_\_m128i \_\_mm\_unpacklo\_epi16 ( \_\_m128i m1, \_\_m128i m2)  
 VPUNPCKLWD: \_\_m256i \_\_mm256\_unpacklo\_epi16 ( \_\_m256i m1, \_\_m256i m2)  
 PUNPCKLDQ: \_\_m64 \_\_mm\_unpacklo\_pi32 ( \_\_m64 m1, \_\_m64 m2)  
 (V)PUNPCKLDQ: \_\_m128i \_\_mm\_unpacklo\_epi32 ( \_\_m128i m1, \_\_m128i m2)  
 VPUNPCKLDQ: \_\_m256i \_\_mm256\_unpacklo\_epi32 ( \_\_m256i m1, \_\_m256i m2)  
 (V)PUNPCKLDQ: \_\_m128i \_\_mm\_unpacklo\_epi64 ( \_\_m128i m1, \_\_m128i m2)  
 VPUNPCKLQDQ: \_\_m256i \_\_mm256\_unpacklo\_epi64 ( \_\_m256i m1, \_\_m256i m2)

### Flags Affected

None.

### Numeric Exceptions

None.

### Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded VPUNPCKLDQ/QDQ, see Exceptions Type E4NF.

EVEX-encoded VPUNPCKLBW/WD, see Exceptions Type E4NF.nb.

## PXOR—Logical Exclusive OR

Opcode*/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F EF /r <sup>1</sup> PXOR mm, mm/m64	RM	V/V	MMX	Bitwise XOR of mm/m64 and mm.
66 0F EF /r PXOR xmm1, xmm2/m128	RM	V/V	SSE2	Bitwise XOR of xmm2/m128 and xmm1.
VEX.NDS.128.66.0F.WIG EF /r VPXOR xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Bitwise XOR of xmm3/m128 and xmm2.
VEX.NDS.256.66.0F.WIG EF /r VPXOR ymm1, ymm2, ymm3/m256	RVM	V/V	AVX2	Bitwise XOR of ymm3/m256 and ymm2.
EVEX.NDS.128.66.0F.W0 EF /r VPXORD xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	FV	V/V	AVX512VL AVX512F	Bitwise XOR of packed doubleword integers in xmm2 and xmm3/m128 using writemask k1.
EVEX.NDS.256.66.0F.W0 EF /r VPXORD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	FV	V/V	AVX512VL AVX512F	Bitwise XOR of packed doubleword integers in ymm2 and ymm3/m256 using writemask k1.
EVEX.NDS.512.66.0F.W0 EF /r VPXORD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	FV	V/V	AVX512F	Bitwise XOR of packed doubleword integers in zmm2 and zmm3/m512/m32bcst using writemask k1.
EVEX.NDS.128.66.0F.W1 EF /r VPXORQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	FV	V/V	AVX512VL AVX512F	Bitwise XOR of packed quadword integers in xmm2 and xmm3/m128 using writemask k1.
EVEX.NDS.256.66.0F.W1 EF /r VPXORQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	FV	V/V	AVX512VL AVX512F	Bitwise XOR of packed quadword integers in ymm2 and ymm3/m256 using writemask k1.
EVEX.NDS.512.66.0F.W1 EF /r VPXORQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	FV	V/V	AVX512F	Bitwise XOR of packed quadword integers in zmm2 and zmm3/m512/m64bcst using writemask k1.

### NOTES:

1. See note in Section 2.4, “AVX and SSE Instruction Exception Specification” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FV	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Performs a bitwise logical exclusive-OR (XOR) operation on the source operand (second operand) and the destination operand (first operand) and stores the result in the destination operand. Each bit of the result is 1 if the corresponding bits of the two operands are different; each bit is 0 if the corresponding bits of the operands are the same.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE instructions 64-bit operand: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand is an MMX technology register.

128-bit Legacy SSE version: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAX\_VL-1:256) of the corresponding register destination are zeroed.

EVEX encoded versions: The first source 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 32/64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

## Operation

### PXOR (64-bit operand)

DEST  $\leftarrow$  DEST XOR SRC

### PXOR (128-bit Legacy SSE version)

DEST  $\leftarrow$  DEST XOR SRC

DEST[VLMAX-1:128] (Unmodified)

### VPXOR (VEX.128 encoded version)

DEST  $\leftarrow$  SRC1 XOR SRC2

DEST[VLMAX-1:128]  $\leftarrow$  0

### VPXOR (VEX.256 encoded version)

DEST  $\leftarrow$  SRC1 XOR SRC2

DEST[VLMAX-1:256]  $\leftarrow$  0

### VPXORD (EVEX encoded versions)

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j  $\leftarrow$  0 TO KL-1

  i  $\leftarrow$  j \* 32

  IF k1[j] OR \*no writemask\* THEN

    IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

      THEN DEST[i+31:i]  $\leftarrow$  SRC1[i+31:i] BITWISE XOR SRC2[31:0]

      ELSE DEST[i+31:i]  $\leftarrow$  SRC1[i+31:i] BITWISE XOR SRC2[i+31:i]

    FI;

  ELSE

    IF \*merging-masking\* ; merging-masking

      THEN \*DEST[31:0] remains unchanged\*

    ELSE ; zeroing-masking

      DEST[31:0]  $\leftarrow$  0

    FI;

  FI;

ENDFOR;

DEST[MAX\_VL-1:VL]  $\leftarrow$  0

**VPXORQ (EVEX encoded versions)**

(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) AND (SRC2 \*is memory\*)

THEN DEST[i+63:i] ← SRC1[i+63:i] BITWISE XOR SRC2[63:0]

ELSE DEST[i+63:i] ← SRC1[i+63:i] BITWISE XOR SRC2[i+63:i]

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[63:0] remains unchanged\*

ELSE ; zeroing-masking

DEST[63:0] ← 0

FI;

FI;

ENDFOR;

DEST[MAX\_VL-1:VL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

VPXORD \_\_m512i \_\_mm512\_xor\_epi32(\_\_m512i a, \_\_m512i b)

VPXORD \_\_m512i \_\_mm512\_mask\_xor\_epi32(\_\_m512i s, \_\_mmask16 m, \_\_m512i a, \_\_m512i b)

VPXORD \_\_m512i \_\_mm512\_maskz\_xor\_epi32(\_\_mmask16 m, \_\_m512i a, \_\_m512i b)

VPXORD \_\_m256i \_\_mm256\_xor\_epi32(\_\_m256i a, \_\_m256i b)

VPXORD \_\_m256i \_\_mm256\_mask\_xor\_epi32(\_\_m256i s, \_\_mmask8 m, \_\_m256i a, \_\_m256i b)

VPXORD \_\_m256i \_\_mm256\_maskz\_xor\_epi32(\_\_mmask8 m, \_\_m256i a, \_\_m256i b)

VPXORD \_\_m128i \_\_mm\_xor\_epi32(\_\_m128i a, \_\_m128i b)

VPXORD \_\_m128i \_\_mm\_mask\_xor\_epi32(\_\_m128i s, \_\_mmask8 m, \_\_m128i a, \_\_m128i b)

VPXORD \_\_m128i \_\_mm\_maskz\_xor\_epi32(\_\_mmask16 m, \_\_m128i a, \_\_m128i b)

VPXORQ \_\_m512i \_\_mm512\_xor\_epi64(\_\_m512i a, \_\_m512i b);

VPXORQ \_\_m512i \_\_mm512\_mask\_xor\_epi64(\_\_m512i s, \_\_mmask8 m, \_\_m512i a, \_\_m512i b);

VPXORQ \_\_m512i \_\_mm512\_maskz\_xor\_epi64(\_\_mmask8 m, \_\_m512i a, \_\_m512i b);

VPXORQ \_\_m256i \_\_mm256\_xor\_epi64(\_\_m256i a, \_\_m256i b);

VPXORQ \_\_m256i \_\_mm256\_mask\_xor\_epi64(\_\_m256i s, \_\_mmask8 m, \_\_m256i a, \_\_m256i b);

VPXORQ \_\_m256i \_\_mm256\_maskz\_xor\_epi64(\_\_mmask8 m, \_\_m256i a, \_\_m256i b);

VPXORQ \_\_m128i \_\_mm\_xor\_epi64(\_\_m128i a, \_\_m128i b);

VPXORQ \_\_m128i \_\_mm\_mask\_xor\_epi64(\_\_m128i s, \_\_mmask8 m, \_\_m128i a, \_\_m128i b);

VPXORQ \_\_m128i \_\_mm\_maskz\_xor\_epi64(\_\_mmask8 m, \_\_m128i a, \_\_m128i b);

PXOR: \_\_m64 \_\_mm\_xor\_si64(\_\_m64 m1, \_\_m64 m2)

(V)PXOR: \_\_m128i \_\_mm\_xor\_si128(\_\_m128i a, \_\_m128i b)

VPXOR: \_\_m256i \_\_mm256\_xor\_si256(\_\_m256i a, \_\_m256i b)

**Flags Affected**

None.

**Numeric Exceptions**

None.

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.

## RCL/RCR/ROL/ROR—Rotate

Opcode**	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
D0 /2	RCL <i>r/m8</i> , 1	M1	Valid	Valid	Rotate 9 bits (CF, <i>r/m8</i> ) left once.
REX + D0 /2	RCL <i>r/m8*</i> , 1	M1	Valid	N.E.	Rotate 9 bits (CF, <i>r/m8</i> ) left once.
D2 /2	RCL <i>r/m8</i> , CL	MC	Valid	Valid	Rotate 9 bits (CF, <i>r/m8</i> ) left CL times.
REX + D2 /2	RCL <i>r/m8*</i> , CL	MC	Valid	N.E.	Rotate 9 bits (CF, <i>r/m8</i> ) left CL times.
C0 /2 <i>ib</i>	RCL <i>r/m8</i> , <i>imm8</i>	MI	Valid	Valid	Rotate 9 bits (CF, <i>r/m8</i> ) left <i>imm8</i> times.
REX + C0 /2 <i>ib</i>	RCL <i>r/m8*</i> , <i>imm8</i>	MI	Valid	N.E.	Rotate 9 bits (CF, <i>r/m8</i> ) left <i>imm8</i> times.
D1 /2	RCL <i>r/m16</i> , 1	M1	Valid	Valid	Rotate 17 bits (CF, <i>r/m16</i> ) left once.
D3 /2	RCL <i>r/m16</i> , CL	MC	Valid	Valid	Rotate 17 bits (CF, <i>r/m16</i> ) left CL times.
C1 /2 <i>ib</i>	RCL <i>r/m16</i> , <i>imm8</i>	MI	Valid	Valid	Rotate 17 bits (CF, <i>r/m16</i> ) left <i>imm8</i> times.
D1 /2	RCL <i>r/m32</i> , 1	M1	Valid	Valid	Rotate 33 bits (CF, <i>r/m32</i> ) left once.
REX.W + D1 /2	RCL <i>r/m64</i> , 1	M1	Valid	N.E.	Rotate 65 bits (CF, <i>r/m64</i> ) left once. Uses a 6 bit count.
D3 /2	RCL <i>r/m32</i> , CL	MC	Valid	Valid	Rotate 33 bits (CF, <i>r/m32</i> ) left CL times.
REX.W + D3 /2	RCL <i>r/m64</i> , CL	MC	Valid	N.E.	Rotate 65 bits (CF, <i>r/m64</i> ) left CL times. Uses a 6 bit count.
C1 /2 <i>ib</i>	RCL <i>r/m32</i> , <i>imm8</i>	MI	Valid	Valid	Rotate 33 bits (CF, <i>r/m32</i> ) left <i>imm8</i> times.
REX.W + C1 /2 <i>ib</i>	RCL <i>r/m64</i> , <i>imm8</i>	MI	Valid	N.E.	Rotate 65 bits (CF, <i>r/m64</i> ) left <i>imm8</i> times. Uses a 6 bit count.
D0 /3	RCR <i>r/m8</i> , 1	M1	Valid	Valid	Rotate 9 bits (CF, <i>r/m8</i> ) right once.
REX + D0 /3	RCR <i>r/m8*</i> , 1	M1	Valid	N.E.	Rotate 9 bits (CF, <i>r/m8</i> ) right once.
D2 /3	RCR <i>r/m8</i> , CL	MC	Valid	Valid	Rotate 9 bits (CF, <i>r/m8</i> ) right CL times.
REX + D2 /3	RCR <i>r/m8*</i> , CL	MC	Valid	N.E.	Rotate 9 bits (CF, <i>r/m8</i> ) right CL times.
C0 /3 <i>ib</i>	RCR <i>r/m8</i> , <i>imm8</i>	MI	Valid	Valid	Rotate 9 bits (CF, <i>r/m8</i> ) right <i>imm8</i> times.
REX + C0 /3 <i>ib</i>	RCR <i>r/m8*</i> , <i>imm8</i>	MI	Valid	N.E.	Rotate 9 bits (CF, <i>r/m8</i> ) right <i>imm8</i> times.
D1 /3	RCR <i>r/m16</i> , 1	M1	Valid	Valid	Rotate 17 bits (CF, <i>r/m16</i> ) right once.
D3 /3	RCR <i>r/m16</i> , CL	MC	Valid	Valid	Rotate 17 bits (CF, <i>r/m16</i> ) right CL times.
C1 /3 <i>ib</i>	RCR <i>r/m16</i> , <i>imm8</i>	MI	Valid	Valid	Rotate 17 bits (CF, <i>r/m16</i> ) right <i>imm8</i> times.
D1 /3	RCR <i>r/m32</i> , 1	M1	Valid	Valid	Rotate 33 bits (CF, <i>r/m32</i> ) right once. Uses a 6 bit count.
REX.W + D1 /3	RCR <i>r/m64</i> , 1	M1	Valid	N.E.	Rotate 65 bits (CF, <i>r/m64</i> ) right once. Uses a 6 bit count.
D3 /3	RCR <i>r/m32</i> , CL	MC	Valid	Valid	Rotate 33 bits (CF, <i>r/m32</i> ) right CL times.
REX.W + D3 /3	RCR <i>r/m64</i> , CL	MC	Valid	N.E.	Rotate 65 bits (CF, <i>r/m64</i> ) right CL times. Uses a 6 bit count.
C1 /3 <i>ib</i>	RCR <i>r/m32</i> , <i>imm8</i>	MI	Valid	Valid	Rotate 33 bits (CF, <i>r/m32</i> ) right <i>imm8</i> times.
REX.W + C1 /3 <i>ib</i>	RCR <i>r/m64</i> , <i>imm8</i>	MI	Valid	N.E.	Rotate 65 bits (CF, <i>r/m64</i> ) right <i>imm8</i> times. Uses a 6 bit count.
D0 /0	ROL <i>r/m8</i> , 1	M1	Valid	Valid	Rotate 8 bits <i>r/m8</i> left once.
REX + D0 /0	ROL <i>r/m8*</i> , 1	M1	Valid	N.E.	Rotate 8 bits <i>r/m8</i> left once
D2 /0	ROL <i>r/m8</i> , CL	MC	Valid	Valid	Rotate 8 bits <i>r/m8</i> left CL times.
REX + D2 /0	ROL <i>r/m8*</i> , CL	MC	Valid	N.E.	Rotate 8 bits <i>r/m8</i> left CL times.
C0 /0 <i>ib</i>	ROL <i>r/m8</i> , <i>imm8</i>	MI	Valid	Valid	Rotate 8 bits <i>r/m8</i> left <i>imm8</i> times.

Opcode**	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
REX + CO /0 <i>ib</i>	ROL <i>r/m8*</i> , <i>imm8</i>	MI	Valid	N.E.	Rotate 8 bits <i>r/m8</i> left <i>imm8</i> times.
D1 /0	ROL <i>r/m16</i> , 1	M1	Valid	Valid	Rotate 16 bits <i>r/m16</i> left once.
D3 /0	ROL <i>r/m16</i> , CL	MC	Valid	Valid	Rotate 16 bits <i>r/m16</i> left CL times.
C1 /0 <i>ib</i>	ROL <i>r/m16</i> , <i>imm8</i>	MI	Valid	Valid	Rotate 16 bits <i>r/m16</i> left <i>imm8</i> times.
D1 /0	ROL <i>r/m32</i> , 1	M1	Valid	Valid	Rotate 32 bits <i>r/m32</i> left once.
REX.W + D1 /0	ROL <i>r/m64</i> , 1	M1	Valid	N.E.	Rotate 64 bits <i>r/m64</i> left once. Uses a 6 bit count.
D3 /0	ROL <i>r/m32</i> , CL	MC	Valid	Valid	Rotate 32 bits <i>r/m32</i> left CL times.
REX.W + D3 /0	ROL <i>r/m64</i> , CL	MC	Valid	N.E.	Rotate 64 bits <i>r/m64</i> left CL times. Uses a 6 bit count.
C1 /0 <i>ib</i>	ROL <i>r/m32</i> , <i>imm8</i>	MI	Valid	Valid	Rotate 32 bits <i>r/m32</i> left <i>imm8</i> times.
REX.W + C1 /0 <i>ib</i>	ROL <i>r/m64</i> , <i>imm8</i>	MI	Valid	N.E.	Rotate 64 bits <i>r/m64</i> left <i>imm8</i> times. Uses a 6 bit count.
D0 /1	ROR <i>r/m8</i> , 1	M1	Valid	Valid	Rotate 8 bits <i>r/m8</i> right once.
REX + D0 /1	ROR <i>r/m8*</i> , 1	M1	Valid	N.E.	Rotate 8 bits <i>r/m8</i> right once.
D2 /1	ROR <i>r/m8</i> , CL	MC	Valid	Valid	Rotate 8 bits <i>r/m8</i> right CL times.
REX + D2 /1	ROR <i>r/m8*</i> , CL	MC	Valid	N.E.	Rotate 8 bits <i>r/m8</i> right CL times.
C0 /1 <i>ib</i>	ROR <i>r/m8</i> , <i>imm8</i>	MI	Valid	Valid	Rotate 8 bits <i>r/m16</i> right <i>imm8</i> times.
REX + C0 /1 <i>ib</i>	ROR <i>r/m8*</i> , <i>imm8</i>	MI	Valid	N.E.	Rotate 8 bits <i>r/m16</i> right <i>imm8</i> times.
D1 /1	ROR <i>r/m16</i> , 1	M1	Valid	Valid	Rotate 16 bits <i>r/m16</i> right once.
D3 /1	ROR <i>r/m16</i> , CL	MC	Valid	Valid	Rotate 16 bits <i>r/m16</i> right CL times.
C1 /1 <i>ib</i>	ROR <i>r/m16</i> , <i>imm8</i>	MI	Valid	Valid	Rotate 16 bits <i>r/m16</i> right <i>imm8</i> times.
D1 /1	ROR <i>r/m32</i> , 1	M1	Valid	Valid	Rotate 32 bits <i>r/m32</i> right once.
REX.W + D1 /1	ROR <i>r/m64</i> , 1	M1	Valid	N.E.	Rotate 64 bits <i>r/m64</i> right once. Uses a 6 bit count.
D3 /1	ROR <i>r/m32</i> , CL	MC	Valid	Valid	Rotate 32 bits <i>r/m32</i> right CL times.
REX.W + D3 /1	ROR <i>r/m64</i> , CL	MC	Valid	N.E.	Rotate 64 bits <i>r/m64</i> right CL times. Uses a 6 bit count.
C1 /1 <i>ib</i>	ROR <i>r/m32</i> , <i>imm8</i>	MI	Valid	Valid	Rotate 32 bits <i>r/m32</i> right <i>imm8</i> times.
REX.W + C1 /1 <i>ib</i>	ROR <i>r/m64</i> , <i>imm8</i>	MI	Valid	N.E.	Rotate 64 bits <i>r/m64</i> right <i>imm8</i> times. Uses a 6 bit count.

**NOTES:**

\* In 64-bit mode, *r/m8* can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

\*\* See IA-32 Architecture Compatibility section below.

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M1	ModRM:r/m (w)	1	NA	NA
MC	ModRM:r/m (w)	CL	NA	NA
MI	ModRM:r/m (w)	<i>imm8</i>	NA	NA

## Description

Shifts (rotates) the bits of the first operand (destination operand) the number of bit positions specified in the second operand (count operand) and stores the result in the destination operand. The destination operand can be a register or a memory location; the count operand is an unsigned integer that can be an immediate or a value in the CL register. The count is masked to 5 bits (or 6 bits if in 64-bit mode and REX.W = 1).

The rotate left (ROL) and rotate through carry left (RCL) instructions shift all the bits toward more-significant bit positions, except for the most-significant bit, which is rotated to the least-significant bit location. The rotate right (ROR) and rotate through carry right (RCR) instructions shift all the bits toward less significant bit positions, except for the least-significant bit, which is rotated to the most-significant bit location.

The RCL and RCR instructions include the CF flag in the rotation. The RCL instruction shifts the CF flag into the least-significant bit and shifts the most-significant bit into the CF flag. The RCR instruction shifts the CF flag into the most-significant bit and shifts the least-significant bit into the CF flag. For the ROL and ROR instructions, the original value of the CF flag is not a part of the result, but the CF flag receives a copy of the bit that was shifted from one end to the other.

The OF flag is defined only for the 1-bit rotates; it is undefined in all other cases (except RCL and RCR instructions only: a zero-bit rotate does nothing, that is affects no flags). For left rotates, the OF flag is set to the exclusive OR of the CF bit (after the rotate) and the most-significant bit of the result. For right rotates, the OF flag is set to the exclusive OR of the two most-significant bits of the result.

In 64-bit mode, using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Use of REX.W promotes the first operand to 64 bits and causes the count operand to become a 6-bit counter.

## IA-32 Architecture Compatibility

The 8086 does not mask the rotation count. However, all other IA-32 processors (starting with the Intel 286 processor) do mask the rotation count to 5 bits, resulting in a maximum count of 31. This masking is done in all operating modes (including the virtual-8086 mode) to reduce the maximum execution time of the instructions.

## Operation

(\* RCL and RCR instructions \*)

SIZE ← OperandSize;

CASE (determine count) OF

SIZE ← 8: tempCOUNT ← (COUNT AND 1FH) MOD 9;  
 SIZE ← 16: tempCOUNT ← (COUNT AND 1FH) MOD 17;  
 SIZE ← 32: tempCOUNT ← COUNT AND 1FH;  
 SIZE ← 64: tempCOUNT ← COUNT AND 3FH;

ESAC;

(\* RCL instruction operation \*)

WHILE (tempCOUNT ≠ 0)

DO

tempCF ← MSB(DEST);  
 DEST ← (DEST \* 2) + CF;  
 CF ← tempCF;  
 tempCOUNT ← tempCOUNT - 1;

OD;

ELIHW;

IF (COUNT & COUNTMASK) = 1

THEN OF ← MSB(DEST) XOR CF;

ELSE OF is undefined;

FI;

```

(* RCR instruction operation *)
IF (COUNT & COUNTMASK) = 1
  THEN OF ← MSB(DEST) XOR CF;
  ELSE OF is undefined;
FI;
WHILE (tempCOUNT ≠ 0)
  DO
    tempCF ← LSB(SRC);
    DEST ← (DEST / 2) + (CF * 2SIZE);
    CF ← tempCF;
    tempCOUNT ← tempCOUNT - 1;
  OD;

(* ROL and ROR instructions *)
IF OperandSize = 64
  THEN COUNTMASK = 3FH;
  ELSE COUNTMASK = 1FH;
FI;

(* ROL instruction operation *)
tempCOUNT ← (COUNT & COUNTMASK) MOD SIZE

WHILE (tempCOUNT ≠ 0)
  DO
    tempCF ← MSB(DEST);
    DEST ← (DEST * 2) + tempCF;
    tempCOUNT ← tempCOUNT - 1;
  OD;
ELIHW;
IF (COUNT & COUNTMASK) ≠ 0
  THEN CF ← LSB(DEST);
FI;
IF (COUNT & COUNTMASK) = 1
  THEN OF ← MSB(DEST) XOR CF;
  ELSE OF is undefined;
FI;

(* ROR instruction operation *)
tempCOUNT ← (COUNT & COUNTMASK) MOD SIZE
WHILE (tempCOUNT ≠ 0)
  DO
    tempCF ← LSB(SRC);
    DEST ← (DEST / 2) + (tempCF * 2SIZE);
    tempCOUNT ← tempCOUNT - 1;
  OD;
ELIHW;
IF (COUNT & COUNTMASK) ≠ 0
  THEN CF ← MSB(DEST);
FI;
IF (COUNT & COUNTMASK) = 1
  THEN OF ← MSB(DEST) XOR MSB - 1(DEST);
  ELSE OF is undefined;
FI;

```



## Flags Affected

If the masked count is 0, the flags are not affected. If the masked count is 1, then the OF flag is affected, otherwise (masked count is greater than 1) the OF flag is undefined. The CF flag is affected when the masked count is non-zero. The SF, ZF, AF, and PF flags are always unaffected.

## Protected Mode Exceptions

#GP(0)	If the source operand is located in a non-writable segment. If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

## Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used.

## Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

## Compatibility Mode Exceptions

Same exceptions as in protected mode.

## 64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the source operand is located in a nonwritable segment. If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

## SHUFDP—Packed Interleave Shuffle of Pairs of Double-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F C6 /r ib SHUFDP xmm1, xmm2/m128, imm8	RMI	V/V	SSE2	Shuffle two pairs of double-precision floating-point values from xmm1 and xmm2/m128 using imm8 to select from each pair, interleaved result is stored in xmm1.
VEX.NDS.128.66.0F.WIG C6 /r ib VSHUFDP xmm1, xmm2, xmm3/m128, imm8	RVMI	V/V	AVX	Shuffle two pairs of double-precision floating-point values from xmm2 and xmm3/m128 using imm8 to select from each pair, interleaved result is stored in xmm1.
VEX.NDS.256.66.0F.WIG C6 /r ib VSHUFDP ymm1, ymm2, ymm3/m256, imm8	RVMI	V/V	AVX	Shuffle four pairs of double-precision floating-point values from ymm2 and ymm3/m256 using imm8 to select from each pair, interleaved result is stored in xmm1.
EVEX.NDS.128.66.0F.W1 C6 /r ib VSHUFDP xmm1{k1}{z}, xmm2, xmm3/m128/m64bcst, imm8	FV	V/V	AVX512VL AVX512F	Shuffle two pairs of double-precision floating-point values from xmm2 and xmm3/m128/m64bcst using imm8 to select from each pair. store interleaved results in xmm1 subject to writemask k1.
EVEX.NDS.256.66.0F.W1 C6 /r ib VSHUFDP ymm1{k1}{z}, ymm2, ymm3/m256/m64bcst, imm8	FV	V/V	AVX512VL AVX512F	Shuffle four pairs of double-precision floating-point values from ymm2 and ymm3/m256/m64bcst using imm8 to select from each pair. store interleaved results in ymm1 subject to writemask k1.
EVEX.NDS.512.66.0F.W1 C6 /r ib VSHUFDP zmm1{k1}{z}, zmm2, zmm3/m512/m64bcst, imm8	FV	V/V	AVX512F	Shuffle eight pairs of double-precision floating-point values from zmm2 and zmm3/m512/m64bcst using imm8 to select from each pair. store interleaved results in zmm1 subject to writemask k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (r, w)	ModRM:r/m (r)	Imm8	NA
RVMI	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	Imm8
FV	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	Imm8

### Description

Selects a double-precision floating-point value of an input pair using a bit control and move to a designated element of the destination operand. The low-to-high order of double-precision element of the destination operand is interleaved between the first source operand and the second source operand at the granularity of input pair of 128 bits. Each bit in the imm8 byte, starting from bit 0, is the select control of the corresponding element of the destination to received the shuffled result of an input pair.

EVEX encoded versions: The first source 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 64-bit memory location. The destination operand is a ZMM/YMM/XMM register updated according to the writemask. The select controls are the lower 8/4/2 bits of the imm8 byte.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register. The select controls are the bit 3:0 of the imm8 byte, imm8[7:4] are ignored.

VEX.128 encoded version: The first source operand is a XMM register. The second source operand can be a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper bits (MAX\_VL-1:128) of the corresponding ZMM register destination are zeroed. The select controls are the bit 1:0 of the imm8 byte, imm8[7:2] are ignored.

128-bit Legacy SSE version: The second source can be an XMM register or a 128-bit memory location. The destination operand and the first source operand is the same and is an XMM register. The upper bits (MAX\_VL-1:128) of the corresponding ZMM register destination are unmodified. The select controls are the bit 1:0 of the imm8 byte, imm8[7:2) are ignored.

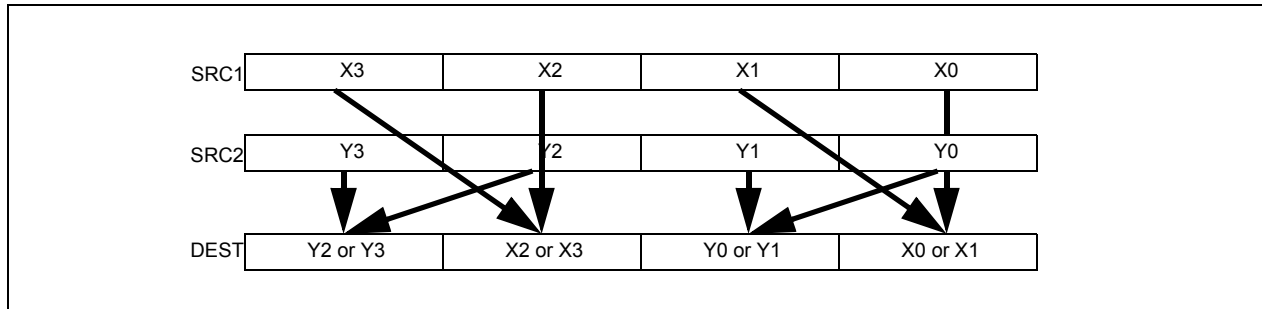


Figure 4-20. 256-bit VSHUFPD Operation of Four Pairs of DP FP Values

### Operation

#### VSHUFPD (EVEX encoded versions when SRC2 is a vector register)

(KL, VL) = (2, 128), (4, 256), (8, 512)

IF IMMO[0] = 0

THEN TMP\_DEST[63:0] ← SRC1[63:0]

ELSE TMP\_DEST[63:0] ← SRC1[127:64] FI;

IF IMMO[1] = 0

THEN TMP\_DEST[127:64] ← SRC2[63:0]

ELSE TMP\_DEST[127:64] ← SRC2[127:64] FI;

IF VL ≥ 256

IF IMMO[2] = 0

THEN TMP\_DEST[191:128] ← SRC1[191:128]

ELSE TMP\_DEST[191:128] ← SRC1[255:192] FI;

IF IMMO[3] = 0

THEN TMP\_DEST[255:192] ← SRC2[191:128]

ELSE TMP\_DEST[255:192] ← SRC2[255:192] FI;

FI;

IF VL ≥ 512

IF IMMO[4] = 0

THEN TMP\_DEST[319:256] ← SRC1[319:256]

ELSE TMP\_DEST[319:256] ← SRC1[383:320] FI;

IF IMMO[5] = 0

THEN TMP\_DEST[383:320] ← SRC2[319:256]

ELSE TMP\_DEST[383:320] ← SRC2[383:320] FI;

IF IMMO[6] = 0

THEN TMP\_DEST[447:384] ← SRC1[447:384]

ELSE TMP\_DEST[447:384] ← SRC1[511:448] FI;

IF IMMO[7] = 0

THEN TMP\_DEST[511:448] ← SRC2[447:384]

ELSE TMP\_DEST[511:448] ← SRC2[511:448] FI;

FI;

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN DEST[i+63:i] ← TMP\_DEST[i+63:i]

ELSE

```

        IF *merging-masking*           ; merging-masking
            THEN *DEST[i+63:i] remains unchanged*
            ELSE *zeroing-masking*     ; zeroing-masking
                DEST[i+63:i] ← 0
        FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VSHUFPD (EVEX encoded versions when SRC2 is memory)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

```

FOR j ← 0 TO KL-1
    i ← j * 64
    IF (EVEX.b = 1)
        THEN TMP_SRC2[i+63:i] ← SRC2[63:0]
        ELSE TMP_SRC2[i+63:i] ← SRC2[i+63:i]
    FI;
ENDFOR;
IF IMMO[0] = 0
    THEN TMP_DEST[63:0] ← SRC1[63:0]
    ELSE TMP_DEST[63:0] ← SRC1[127:64] FI;
IF IMMO[1] = 0
    THEN TMP_DEST[127:64] ← TMP_SRC2[63:0]
    ELSE TMP_DEST[127:64] ← TMP_SRC2[127:64] FI;
IF VL >= 256
    IF IMMO[2] = 0
        THEN TMP_DEST[191:128] ← SRC1[191:128]
        ELSE TMP_DEST[191:128] ← SRC1[255:192] FI;
    IF IMMO[3] = 0
        THEN TMP_DEST[255:192] ← TMP_SRC2[191:128]
        ELSE TMP_DEST[255:192] ← TMP_SRC2[255:192] FI;
    FI;
IF VL >= 512
    IF IMMO[4] = 0
        THEN TMP_DEST[319:256] ← SRC1[319:256]
        ELSE TMP_DEST[319:256] ← SRC1[383:320] FI;
    IF IMMO[5] = 0
        THEN TMP_DEST[383:320] ← TMP_SRC2[319:256]
        ELSE TMP_DEST[383:320] ← TMP_SRC2[383:320] FI;
    IF IMMO[6] = 0
        THEN TMP_DEST[447:384] ← SRC1[447:384]
        ELSE TMP_DEST[447:384] ← SRC1[511:448] FI;
    IF IMMO[7] = 0
        THEN TMP_DEST[511:448] ← TMP_SRC2[447:384]
        ELSE TMP_DEST[511:448] ← TMP_SRC2[511:448] FI;
    FI;
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*           ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
            FI
        FI

```

```

        ELSE *zeroing-masking*          ; zeroing-masking
          DEST[i+63:i] ← 0
      FI
  FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VSHUFPD (VEX.256 encoded version)**

```

IF IMMO[0] = 0
  THEN DEST[63:0] ← SRC1[63:0]
  ELSE DEST[63:0] ← SRC1[127:64] FI;
IF IMMO[1] = 0
  THEN DEST[127:64] ← SRC2[63:0]
  ELSE DEST[127:64] ← SRC2[127:64] FI;
IF IMMO[2] = 0
  THEN DEST[191:128] ← SRC1[191:128]
  ELSE DEST[191:128] ← SRC1[255:192] FI;
IF IMMO[3] = 0
  THEN DEST[255:192] ← SRC2[191:128]
  ELSE DEST[255:192] ← SRC2[255:192] FI;
DEST[MAX_VL-1:256] (Unmodified)

```

**VSHUFPD (VEX.128 encoded version)**

```

IF IMMO[0] = 0
  THEN DEST[63:0] ← SRC1[63:0]
  ELSE DEST[63:0] ← SRC1[127:64] FI;
IF IMMO[1] = 0
  THEN DEST[127:64] ← SRC2[63:0]
  ELSE DEST[127:64] ← SRC2[127:64] FI;
DEST[MAX_VL-1:128] ← 0

```

**VSHUFPD (128-bit Legacy SSE version)**

```

IF IMMO[0] = 0
  THEN DEST[63:0] ← SRC1[63:0]
  ELSE DEST[63:0] ← SRC1[127:64] FI;
IF IMMO[1] = 0
  THEN DEST[127:64] ← SRC2[63:0]
  ELSE DEST[127:64] ← SRC2[127:64] FI;
DEST[MAX_VL-1:128] (Unmodified)

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VSHUFPD __m512d __mm512_shuffle_pd(__m512d a, __m512d b, int imm);
VSHUFPD __m512d __mm512_mask_shuffle_pd(__m512d s, __mmask8 k, __m512d a, __m512d b, int imm);
VSHUFPD __m512d __mm512_maskz_shuffle_pd(__mmask8 k, __m512d a, __m512d b, int imm);
VSHUFPD __m256d __mm256_shuffle_pd(__m256d a, __m256d b, const int select);
VSHUFPD __m256d __mm256_mask_shuffle_pd(__m256d s, __mmask8 k, __m256d a, __m256d b, int imm);
VSHUFPD __m256d __mm256_maskz_shuffle_pd(__mmask8 k, __m256d a, __m256d b, int imm);
SHUFPD __m128d __mm_shuffle_pd(__m128d a, __m128d b, const int select);
VSHUFPD __m128d __mm_mask_shuffle_pd(__m128d s, __mmask8 k, __m128d a, __m128d b, int imm);
VSHUFPD __m128d __mm_maskz_shuffle_pd(__mmask8 k, __m128d a, __m128d b, int imm);

```

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4NF.

## SHUFPS—Packed Interleave Shuffle of Quadruplets of Single-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
OF C6 /r ib SHUFPS xmm1, xmm3/m128, imm8	RMI	V/V	SSE	Select from quadruplet of single-precision floating-point values in xmm1 and xmm2/m128 using imm8, interleaved result pairs are stored in xmm1.
VEX.NDS.128.OF.WIG C6 /r ib VSHUFPS xmm1, xmm2, xmm3/m128, imm8	RVMI	V/V	AVX	Select from quadruplet of single-precision floating-point values in xmm1 and xmm2/m128 using imm8, interleaved result pairs are stored in xmm1.
VEX.NDS.256.OF.WIG C6 /r ib VSHUFPS ymm1, ymm2, ymm3/m256, imm8	RVMI	V/V	AVX	Select from quadruplet of single-precision floating-point values in ymm2 and ymm3/m256 using imm8, interleaved result pairs are stored in ymm1.
EVEX.NDS.128.OF.W0 C6 /r ib VSHUFPS xmm1{k1}{z}, xmm2, xmm3/m128/m32bcst, imm8	FV	V/V	AVX512VL AVX512F	Select from quadruplet of single-precision floating-point values in xmm1 and xmm2/m128 using imm8, interleaved result pairs are stored in xmm1, subject to writemask k1.
EVEX.NDS.256.OF.W0 C6 /r ib VSHUFPS ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst, imm8	FV	V/V	AVX512VL AVX512F	Select from quadruplet of single-precision floating-point values in ymm2 and ymm3/m256 using imm8, interleaved result pairs are stored in ymm1, subject to writemask k1.
EVEX.NDS.512.OF.W0 C6 /r ib VSHUFPS zmm1{k1}{z}, zmm2, zmm3/m512/m32bcst, imm8	FV	V/V	AVX512F	Select from quadruplet of single-precision floating-point values in zmm2 and zmm3/m512 using imm8, interleaved result pairs are stored in zmm1, subject to writemask k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (r, w)	ModRM:r/m (r)	Imm8	NA
RVMI	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	Imm8
FV	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	Imm8

### Description

Selects a single-precision floating-point value of an input quadruplet using a two-bit control and move to a designated element of the destination operand. Each 64-bit element-pair of a 128-bit lane of the destination operand is interleaved between the corresponding lane of the first source operand and the second source operand at the granularity 128 bits. Each two bits in the imm8 byte, starting from bit 0, is the select control of the corresponding element of a 128-bit lane of the destination to received the shuffled result of an input quadruplet. The two lower elements of a 128-bit lane in the destination receives shuffle results from the quadruple of the first source operand. The next two elements of the destination receives shuffle results from the quadruple of the second source operand.

EVEX encoded versions: The first source 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 32-bit memory location. The destination operand is a ZMM/YMM/XMM register updated according to the writemask. Imm8[7:0] provides 4 select controls for each applicable 128-bit lane of the destination.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register. Imm8[7:0] provides 4 select controls for the high and low 128-bit of the destination.

VEX.128 encoded version: The first source operand is a XMM register. The second source operand can be a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper bits (MAX\_VL-1:128)

of the corresponding ZMM register destination are zeroed. Imm8[7:0] provides 4 select controls for each element of the destination.

128-bit Legacy SSE version: The source can be an XMM register or a 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAX\_VL-1:128) of the corresponding ZMM register destination are unmodified. Imm8[7:0] provides 4 select controls for each element of the destination.

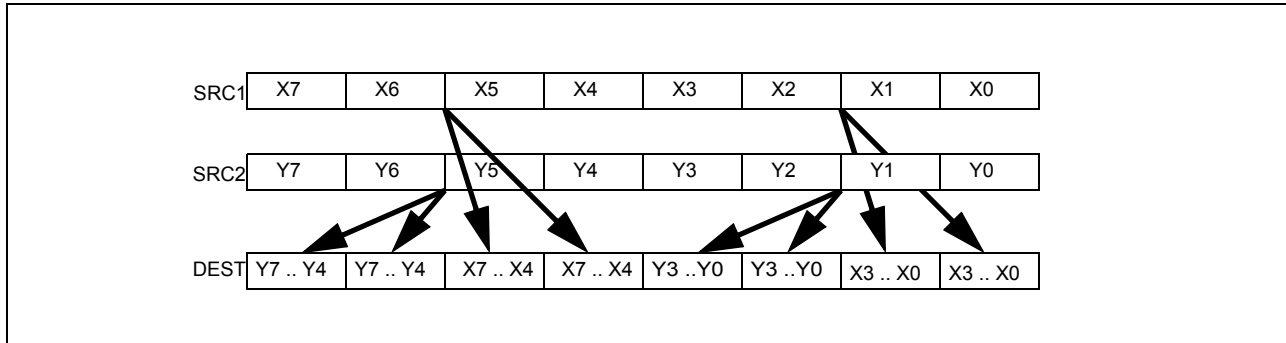


Figure 4-21. 256-bit VSHUFPS Operation of Selection from Input Quadruplet and Pair-wise Interleaved Result

### Operation

```
Select4(SRC, control) {
CASE (control[1:0]) OF
0: TMP ← SRC[31:0];
1: TMP ← SRC[63:32];
2: TMP ← SRC[95:64];
3: TMP ← SRC[127:96];
ESAC;
RETURN TMP
}
```

### VPSHUFPS (EVEX encoded versions when SRC2 is a vector register)

(KL, VL) = (4, 128), (8, 256), (16, 512)

```
TMP_DEST[31:0] ← Select4(SRC1[127:0], imm8[1:0]);
TMP_DEST[63:32] ← Select4(SRC1[127:0], imm8[3:2]);
TMP_DEST[95:64] ← Select4(SRC2[127:0], imm8[5:4]);
TMP_DEST[127:96] ← Select4(SRC2[127:0], imm8[7:6]);
IF VL >= 256
TMP_DEST[159:128] ← Select4(SRC1[255:128], imm8[1:0]);
TMP_DEST[191:160] ← Select4(SRC1[255:128], imm8[3:2]);
TMP_DEST[223:192] ← Select4(SRC2[255:128], imm8[5:4]);
TMP_DEST[255:224] ← Select4(SRC2[255:128], imm8[7:6]);
FI;
IF VL >= 512
TMP_DEST[287:256] ← Select4(SRC1[383:256], imm8[1:0]);
TMP_DEST[319:288] ← Select4(SRC1[383:256], imm8[3:2]);
TMP_DEST[351:320] ← Select4(SRC2[383:256], imm8[5:4]);
TMP_DEST[383:352] ← Select4(SRC2[383:256], imm8[7:6]);
TMP_DEST[415:384] ← Select4(SRC1[511:384], imm8[1:0]);
TMP_DEST[447:416] ← Select4(SRC1[511:384], imm8[3:2]);
TMP_DEST[479:448] ← Select4(SRC2[511:384], imm8[5:4]);
TMP_DEST[511:480] ← Select4(SRC2[511:384], imm8[7:6]);
```



```

FI;
FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
  ELSE
    IF *merging-masking*           ; merging-masking
      THEN *DEST[i+31:i] remains unchanged*
    ELSE *zeroing-masking*         ; zeroing-masking
      DEST[i+31:i] ← 0
    FI
  FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VPSHUFPS (EVEX encoded versions when SRC2 is memory)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 32
  IF (EVEX.b = 1)
    THEN TMP_SRC2[i+31:i] ← SRC2[31:0]
  ELSE TMP_SRC2[i+31:i] ← SRC2[i+31:i]
  FI;
ENDFOR;
TMP_DEST[31:0] ← Select4(SRC1[127:0], imm8[1:0]);
TMP_DEST[63:32] ← Select4(SRC1[127:0], imm8[3:2]);
TMP_DEST[95:64] ← Select4(TMP_SRC2[127:0], imm8[5:4]);
TMP_DEST[127:96] ← Select4(TMP_SRC2[127:0], imm8[7:6]);
IF VL >= 256
  TMP_DEST[159:128] ← Select4(SRC1[255:128], imm8[1:0]);
  TMP_DEST[191:160] ← Select4(SRC1[255:128], imm8[3:2]);
  TMP_DEST[223:192] ← Select4(TMP_SRC2[255:128], imm8[5:4]);
  TMP_DEST[255:224] ← Select4(TMP_SRC2[255:128], imm8[7:6]);
FI;
IF VL >= 512
  TMP_DEST[287:256] ← Select4(SRC1[383:256], imm8[1:0]);
  TMP_DEST[319:288] ← Select4(SRC1[383:256], imm8[3:2]);
  TMP_DEST[351:320] ← Select4(TMP_SRC2[383:256], imm8[5:4]);
  TMP_DEST[383:352] ← Select4(TMP_SRC2[383:256], imm8[7:6]);
  TMP_DEST[415:384] ← Select4(SRC1[511:384], imm8[1:0]);
  TMP_DEST[447:416] ← Select4(SRC1[511:384], imm8[3:2]);
  TMP_DEST[479:448] ← Select4(TMP_SRC2[511:384], imm8[5:4]);
  TMP_DEST[511:480] ← Select4(TMP_SRC2[511:384], imm8[7:6]);
FI;
FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
  ELSE
    IF *merging-masking*           ; merging-masking
      THEN *DEST[i+31:i] remains unchanged*
    ELSE *zeroing-masking*         ; zeroing-masking
      DEST[i+31:i] ← 0
    FI
  FI;
ENDFOR

```

```

FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0
VSHUFPS (VEX.256 encoded version)
DEST[31:0] ← Select4(SRC1[127:0], imm8[1:0]);
DEST[63:32] ← Select4(SRC1[127:0], imm8[3:2]);
DEST[95:64] ← Select4(SRC2[127:0], imm8[5:4]);
DEST[127:96] ← Select4(SRC2[127:0], imm8[7:6]);
DEST[159:128] ← Select4(SRC1[255:128], imm8[1:0]);
DEST[191:160] ← Select4(SRC1[255:128], imm8[3:2]);
DEST[223:192] ← Select4(SRC2[255:128], imm8[5:4]);
DEST[255:224] ← Select4(SRC2[255:128], imm8[7:6]);
DEST[MAX_VL-1:256] ← 0

```

```

VSHUFPS (VEX.128 encoded version)
DEST[31:0] ← Select4(SRC1[127:0], imm8[1:0]);
DEST[63:32] ← Select4(SRC1[127:0], imm8[3:2]);
DEST[95:64] ← Select4(SRC2[127:0], imm8[5:4]);
DEST[127:96] ← Select4(SRC2[127:0], imm8[7:6]);
DEST[MAX_VL-1:128] ← 0

```

```

SHUFPS (128-bit Legacy SSE version)
DEST[31:0] ← Select4(SRC1[127:0], imm8[1:0]);
DEST[63:32] ← Select4(SRC1[127:0], imm8[3:2]);
DEST[95:64] ← Select4(SRC2[127:0], imm8[5:4]);
DEST[127:96] ← Select4(SRC2[127:0], imm8[7:6]);
DEST[MAX_VL-1:128] (Unmodified)

```

#### Intel C/C++ Compiler Intrinsic Equivalent

```

VSHUFPS __m512 _mm512_shuffle_ps(__m512 a, __m512 b, int imm);
VSHUFPS __m512 _mm512_mask_shuffle_ps(__m512 s, __mmask16 k, __m512 a, __m512 b, int imm);
VSHUFPS __m512 _mm512_maskz_shuffle_ps(__mmask16 k, __m512 a, __m512 b, int imm);
VSHUFPS __m256 _mm256_shuffle_ps (__m256 a, __m256 b, const int select);
VSHUFPS __m256 _mm256_mask_shuffle_ps(__m256 s, __mmask8 k, __m256 a, __m256 b, int imm);
VSHUFPS __m256 _mm256_maskz_shuffle_ps(__mmask8 k, __m256 a, __m256 b, int imm);
SHUFPS __m128 _mm_shuffle_ps (__m128 a, __m128 b, const int select);
VSHUFPS __m128 _mm_mask_shuffle_ps(__m128 s, __mmask8 k, __m128 a, __m128 b, int imm);
VSHUFPS __m128 _mm_maskz_shuffle_ps(__mmask8 k, __m128 a, __m128 b, int imm);

```

#### SIMD Floating-Point Exceptions

None

#### Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4NF.

## SQRTPD—Square Root of Double-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 51 /r SQRTPD xmm1, xmm2/m128	RM	V/V	SSE2	Computes Square Roots of the packed double-precision floating-point values in xmm2/m128 and stores the result in xmm1.
VEX.128.66.0F.WIG 51 /r VSQRTPD xmm1, xmm2/m128	RM	V/V	AVX	Computes Square Roots of the packed double-precision floating-point values in xmm2/m128 and stores the result in xmm1.
VEX.256.66.0F.WIG 51 /r VSQRTPD ymm1, ymm2/m256	RM	V/V	AVX	Computes Square Roots of the packed double-precision floating-point values in ymm2/m256 and stores the result in ymm1.
EVEX.128.66.0F.W1 51 /r VSQRTPD xmm1 {k1}{z}, xmm2/m128/m32bcst	FV	V/V	AVX512VL AVX512F	Computes Square Roots of the packed double-precision floating-point values in xmm2/m128/m64bcst and stores the result in xmm1 subject to writemask k1.
EVEX.256.66.0F.W1 51 /r VSQRTPD ymm1 {k1}{z}, ymm2/m256/m32bcst	FV	V/V	AVX512VL AVX512F	Computes Square Roots of the packed double-precision floating-point values in ymm2/m256/m64bcst and stores the result in ymm1 subject to writemask k1.
EVEX.512.66.0F.W1 51 /r VSQRTPD zmm1 {k1}{z}, zmm2/m512/m64bcst{er}	FV	V/V	AVX512F	Computes Square Roots of the packed double-precision floating-point values in zmm2/m512/m64bcst and stores the result in zmm1 subject to writemask k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
FV	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Performs a SIMD computation of the square roots of the two, four or eight packed double-precision floating-point values in the source operand (the second operand) stores the packed double-precision floating-point results in the destination operand (the first operand).

EVEX encoded versions: 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 64-bit memory location. The destination operand is a ZMM/YMM/XMM register updated according to the writemask.

VEX.256 encoded version: The source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAX\_VL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: the source operand second source operand or a 128-bit memory location. The destination operand is an XMM register. The upper bits (MAX\_VL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAX\_VL-1:128) of the corresponding ZMM register destination are unmodified.

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

**Operation****VSQRTPD (EVEX encoded versions)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

IF (VL = 512) AND (EVEX.b = 1) AND (SRC \*is register\*)

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\* THEN

IF (EVEX.b = 1) AND (SRC \*is memory\*)

THEN DEST[i+63:i] ← SQRT(SRC[63:0])

ELSE DEST[i+63:i] ← SQRT(SRC[i+63:i])

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VSQRTPD (VEX.256 encoded version)**

DEST[63:0] ← SQRT(SRC[63:0])

DEST[127:64] ← SQRT(SRC[127:64])

DEST[191:128] ← SQRT(SRC[191:128])

DEST[255:192] ← SQRT(SRC[255:192])

DEST[MAX\_VL-1:256] ← 0

**VSQRTPD (VEX.128 encoded version)**

DEST[63:0] ← SQRT(SRC[63:0])

DEST[127:64] ← SQRT(SRC[127:64])

DEST[MAX\_VL-1:128] ← 0

**SQRTPD (128-bit Legacy SSE version)**

DEST[63:0] ← SQRT(SRC[63:0])

DEST[127:64] ← SQRT(SRC[127:64])

DEST[MAX\_VL-1:128] (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**

VSQRTPD \_\_m512d \_\_mm512\_sqrt\_round\_pd(\_\_m512d a, int r);

VSQRTPD \_\_m512d \_\_mm512\_mask\_sqrt\_round\_pd(\_\_m512d s, \_\_mmask8 k, \_\_m512d a, int r);

VSQRTPD \_\_m512d \_\_mm512\_maskz\_sqrt\_round\_pd(\_\_mmask8 k, \_\_m512d a, int r);

VSQRTPD \_\_m256d \_\_mm256\_sqrt\_pd(\_\_m256d a);

VSQRTPD \_\_m256d \_\_mm256\_mask\_sqrt\_pd(\_\_m256d s, \_\_mmask8 k, \_\_m256d a, int r);

VSQRTPD \_\_m256d \_\_mm256\_maskz\_sqrt\_pd(\_\_mmask8 k, \_\_m256d a, int r);

SQRTPD \_\_m128d \_\_mm\_sqrt\_pd(\_\_m128d a);

VSQRTPD \_\_m128d \_\_mm\_mask\_sqrt\_pd(\_\_m128d s, \_\_mmask8 k, \_\_m128d a, int r);

VSQRTPD \_\_m128d \_\_mm\_maskz\_sqrt\_pd(\_\_mmask8 k, \_\_m128d a, int r);

**SIMD Floating-Point Exceptions**

Invalid, Precision, Denormal

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 2; additionally

#UD                    If VEX.vvvv != 1111B.

EVEX-encoded instruction, see Exceptions Type E2.

#UD                    If EVEX.vvvv != 1111B.

## SQRTPS—Square Root of Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
OF 51 /r SQRTPS xmm1, xmm2/m128	RM	V/V	SSE	Computes Square Roots of the packed single-precision floating-point values in xmm2/m128 and stores the result in xmm1.
VEX.128.OF.WIG 51 /r VSQRTPS xmm1, xmm2/m128	RM	V/V	AVX	Computes Square Roots of the packed single-precision floating-point values in xmm2/m128 and stores the result in xmm1.
VEX.256.OF.WIG 51/r VSQRTPS ymm1, ymm2/m256	RM	V/V	AVX	Computes Square Roots of the packed single-precision floating-point values in ymm2/m256 and stores the result in ymm1.
EVEX.128.OF.W0 51 /r VSQRTPS xmm1 {k1}{z}, xmm2/m128/m32bcst	FV	V/V	AVX512VL AVX512F	Computes Square Roots of the packed single-precision floating-point values in xmm2/m128/m32bcst and stores the result in xmm1 subject to writemask k1.
EVEX.256.OF.W0 51 /r VSQRTPS ymm1 {k1}{z}, ymm2/m256/m32bcst	FV	V/V	AVX512VL AVX512F	Computes Square Roots of the packed single-precision floating-point values in ymm2/m256/m32bcst and stores the result in ymm1 subject to writemask k1.
EVEX.512.OF.W0 51/r VSQRTPS zmm1 {k1}{z}, zmm2/m512/m32bcst{er}	FV	V/V	AVX512F	Computes Square Roots of the packed single-precision floating-point values in zmm2/m512/m32bcst and stores the result in zmm1 subject to writemask k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
FV	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Performs a SIMD computation of the square roots of the four, eight or sixteen packed single-precision floating-point values in the source operand (second operand) stores the packed single-precision floating-point results in the destination operand.

EVEX.512 encoded versions: 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 32-bit memory location. The destination operand is a ZMM/YMM/XMM register updated according to the writemask.

VEX.256 encoded version: The source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAX\_VL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: the source operand second source operand or a 128-bit memory location. The destination operand is an XMM register. The upper bits (MAX\_VL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAX\_VL-1:128) of the corresponding ZMM register destination are unmodified.

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

**Operation****VSQRTPS (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

IF (VL = 512) AND (EVEX.b = 1) AND (SRC \*is register\*)

```

THEN
    SET_RM(EVEX.RC);
ELSE
    SET_RM(MXCSR.RM);
FI;
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask* THEN
        IF (EVEX.b = 1) AND (SRC *is memory*)
            THEN DEST[i+31:i] ← SQRT(SRC[31:0])
            ELSE DEST[i+31:i] ← SQRT(SRC[i+31:i])
        FI;
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+31:i] remains unchanged*
        ELSE ; zeroing-masking
            DEST[i+31:i] ← 0
        FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VSQRTPS (VEX.256 encoded version)**

DEST[31:0] ← SQRT(SRC[31:0])

DEST[63:32] ← SQRT(SRC[63:32])

DEST[95:64] ← SQRT(SRC[95:64])

DEST[127:96] ← SQRT(SRC[127:96])

DEST[159:128] ← SQRT(SRC[159:128])

DEST[191:160] ← SQRT(SRC[191:160])

DEST[223:192] ← SQRT(SRC[223:192])

DEST[255:224] ← SQRT(SRC[255:224])

**VSQRTPS (VEX.128 encoded version)**

DEST[31:0] ← SQRT(SRC[31:0])

DEST[63:32] ← SQRT(SRC[63:32])

DEST[95:64] ← SQRT(SRC[95:64])

DEST[127:96] ← SQRT(SRC[127:96])

DEST[MAX\_VL-1:128] ← 0

**SQRTPS (128-bit Legacy SSE version)**

DEST[31:0] ← SQRT(SRC[31:0])

DEST[63:32] ← SQRT(SRC[63:32])

DEST[95:64] ← SQRT(SRC[95:64])

DEST[127:96] ← SQRT(SRC[127:96])

DEST[MAX\_VL-1:128] (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VSQRTPS __m512 _mm512_sqrt_round_ps(__m512 a, int r);
VSQRTPS __m512 _mm512_mask_sqrt_round_ps(__m512 s, __mmask16 k, __m512 a, int r);
VSQRTPS __m512 _mm512_maskz_sqrt_round_ps(__mmask16 k, __m512 a, int r);
VSQRTPS __m256 _mm256_sqrt_ps (__m256 a);
VSQRTPS __m256 _mm256_mask_sqrt_ps(__m256 s, __mmask8 k, __m256 a, int r);
VSQRTPS __m256 _mm256_maskz_sqrt_ps(__mmask8 k, __m256 a, int r);
SQRTPS __m128 _mm_sqrt_ps (__m128 a);
VSQRTPS __m128 _mm_mask_sqrt_ps(__m128 s, __mmask8 k, __m128 a, int r);
VSQRTPS __m128 _mm_maskz_sqrt_ps(__mmask8 k, __m128 a, int r);

```

**SIMD Floating-Point Exceptions**

Invalid, Precision, Denormal

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 2; additionally

#UD If VEX.vvvv != 1111B.

EVEX-encoded instruction, see Exceptions Type E2.

#UD If EVEX.vvvv != 1111B.



## SQRTSD—Compute Square Root of Scalar Double-Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 51/r SQRTSD xmm1,xmm2/m64	RM	V/V	SSE2	Computes square root of the low double-precision floating-point value in xmm2/m64 and stores the results in xmm1.
VEX.NDS.128.F2.0F.WIG 51/r VSQRTSD xmm1,xmm2, xmm3/m64	RVM	V/V	AVX	Computes square root of the low double-precision floating-point value in xmm3/m64 and stores the results in xmm1. Also, upper double-precision floating-point value (bits[127:64]) from xmm2 is copied to xmm1[127:64].
EVEX.NDS.LIG.F2.0F.W1 51/r VSQRTSD xmm1 {k1}{z}, xmm2, xmm3/m64{er}	T1S	V/V	AVX512F	Computes square root of the low double-precision floating-point value in xmm3/m64 and stores the results in xmm1 under writemask k1. Also, upper double-precision floating-point value (bits[127:64]) from xmm2 is copied to xmm1[127:64].

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
T1S	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Computes the square root of the low double-precision floating-point value in the second source operand and stores the double-precision floating-point result in the destination operand. The second source operand can be an XMM register or a 64-bit memory location. The first source and destination operands are XMM registers.

128-bit Legacy SSE version: The first source operand and the destination operand are the same. The quadword at bits 127:64 of the destination operand remains unchanged. Bits (MAX\_VL-1:64) of the corresponding destination register remain unchanged.

VEX.128 and EVEX encoded versions: Bits 127:64 of the destination operand are copied from the corresponding bits of the first source operand. Bits (MAX\_VL-1:128) of the destination register are zeroed.

EVEX encoded version: The low quadword element of the destination operand is updated according to the writemask.

Software should ensure VSQRTSD is encoded with VEX.L=0. Encoding VSQRTSD with VEX.L=1 may encounter unpredictable behavior across different processor generations.

**Operation****VSQRTSD (EVEX encoded version)**

```

IF (EVEX.b = 1) AND (SRC2 *is register*)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
    THEN DEST[63:0] ← SQRT(SRC2[63:0])
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE ; zeroing-masking
                THEN DEST[63:0] ← 0
        FI;
FI;
DEST[127:64] ← SRC1[127:64]
DEST[MAX_VL-1:128] ← 0

```

**VSQRTSD (VEX.128 encoded version)**

```

DEST[63:0] ← SQRT(SRC2[63:0])
DEST[127:64] ← SRC1[127:64]
DEST[MAX_VL-1:128] ← 0

```

**SQRTSD (128-bit Legacy SSE version)**

```

DEST[63:0] ← SQRT(SRC[63:0])
DEST[MAX_VL-1:64] (Unmodified)

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VSQRTSD __m128d __mm_sqrt_round_sd(__m128d a, __m128d b, int r);
VSQRTSD __m128d __mm_mask_sqrt_round_sd(__m128d s, __mmask8 k, __m128d a, __m128d b, int r);
VSQRTSD __m128d __mm_maskz_sqrt_round_sd(__mmask8 k, __m128d a, __m128d b, int r);
SQRTSD __m128d __mm_sqrt_sd (__m128d a, __m128d b)

```

**SIMD Floating-Point Exceptions**

Invalid, Precision, Denormal

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 3.  
 EVEX-encoded instruction, see Exceptions Type E3.

## SQRTSS—Compute Square Root of Scalar Single-Precision Value

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 51 /r SQRTSS xmm1, xmm2/m32	RM	V/V	SSE	Computes square root of the low single-precision floating-point value in xmm2/m32 and stores the results in xmm1.
VEX.NDS.128.F3.0F.WIG 51 /r VSQRTSS xmm1, xmm2, xmm3/m32	RVM	V/V	AVX	Computes square root of the low single-precision floating-point value in xmm3/m32 and stores the results in xmm1. Also, upper single-precision floating-point values (bits[127:32]) from xmm2 are copied to xmm1[127:32].
EVEX.NDS.LIG.F3.0F.W0 51 /r VSQRTSS xmm1 {k1}{z}, xmm2, xmm3/m32{er}	T1S	V/V	AVX512F	Computes square root of the low single-precision floating-point value in xmm3/m32 and stores the results in xmm1 under writemask k1. Also, upper single-precision floating-point values (bits[127:32]) from xmm2 are copied to xmm1[127:32].

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
T1S	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Computes the square root of the low single-precision floating-point value in the second source operand and stores the single-precision floating-point result in the destination operand. The second source operand can be an XMM register or a 32-bit memory location. The first source and destination operands is an XMM register.

128-bit Legacy SSE version: The first source operand and the destination operand are the same. Bits (MAX\_VL-1:32) of the corresponding YMM destination register remain unchanged.

VEX.128 and EVEX encoded versions: Bits 127:32 of the destination operand are copied from the corresponding bits of the first source operand. Bits (MAX\_VL-1:128) of the destination ZMM register are zeroed.

EVEX encoded version: The low doubleword element of the destination operand is updated according to the writemask.

Software should ensure VSQRTSS is encoded with VEX.L=0. Encoding VSQRTSS with VEX.L=1 may encounter unpredictable behavior across different processor generations.

**Operation****VSQRTSS (EVEX encoded version)**

```

IF (EVEX.b = 1) AND (SRC2 *is register*)
  THEN
    SET_RM(EVEX.RC);
  ELSE
    SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
  THEN  DEST[31:0] ← SQRT(SRC2[31:0])
  ELSE
    IF *merging-masking*           ; merging-masking
      THEN *DEST[31:0] remains unchanged*
      ELSE                           ; zeroing-masking
        DEST[31:0] ← 0
    FI;
FI;
DEST[127:31] ← SRC1[127:31]
DEST[MAX_VL-1:128] ← 0

```

**VSQRTSS (VEX.128 encoded version)**

```

DEST[31:0] ← SQRT(SRC2[31:0])
DEST[127:32] ← SRC1[127:32]
DEST[MAX_VL-1:128] ← 0

```

**SQRTSS (128-bit Legacy SSE version)**

```

DEST[31:0] ← SQRT(SRC2[31:0])
DEST[MAX_VL-1:32] (Unmodified)

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VSQRTSS __m128 _mm_sqrt_round_ss(__m128 a, __m128 b, int r);
VSQRTSS __m128 _mm_mask_sqrt_round_ss(__m128 s, __mmask8 k, __m128 a, __m128 b, int r);
VSQRTSS __m128 _mm_maskz_sqrt_round_ss(__mmask8 k, __m128 a, __m128 b, int r);
SQRTSS __m128 _mm_sqrt_ss(__m128 a)

```

**SIMD Floating-Point Exceptions**

Invalid, Precision, Denormal

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 3.  
 EVEX-encoded instruction, see Exceptions Type E3.

## SUBPD—Subtract Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 5C /r SUBPD xmm1, xmm2/m128	RM	V/V	SSE2	Subtract packed double-precision floating-point values in xmm2/mem from xmm1 and store result in xmm1.
VEX.NDS.128.66.0F.WIG 5C /r VSUBPD xmm1,xmm2, xmm3/m128	RVM	V/V	AVX	Subtract packed double-precision floating-point values in xmm3/mem from xmm2 and store result in xmm1.
VEX.NDS.256.66.0F.WIG 5C /r VSUBPD ymm1, ymm2, ymm3/m256	RVM	V/V	AVX	Subtract packed double-precision floating-point values in ymm3/mem from ymm2 and store result in ymm1.
EVEX.NDS.128.66.0F.W1 5C /r VSUBPD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	FV	V/V	AVX512VL AVX512F	Subtract packed double-precision floating-point values from xmm3/m128/m64bcst to xmm2 and store result in xmm1 with writemask k1.
EVEX.NDS.256.66.0F.W1 5C /r VSUBPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	FV	V/V	AVX512VL AVX512F	Subtract packed double-precision floating-point values from ymm3/m256/m64bcst to ymm2 and store result in ymm1 with writemask k1.
EVEX.NDS.512.66.0F.W1 5C /r VSUBPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	FV	V/V	AVX512F	Subtract packed double-precision floating-point values from zmm3/m512/m64bcst to zmm2 and store result in zmm1 with writemask k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FV	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Performs a SIMD subtract of the two, four or eight packed double-precision floating-point values of the second Source operand from the first Source operand, and stores the packed double-precision floating-point results in the destination operand.

VEX.128 and EVEX.128 encoded versions: The second source operand is an XMM register or an 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAX\_VL-1:128) of the corresponding destination register are zeroed.

VEX.256 and EVEX.256 encoded versions: The second source operand is an YMM register or an 256-bit memory location. The first source operand and destination operands are YMM registers. Bits (MAX\_VL-1:256) of the corresponding destination register are zeroed.

EVEX.512 encoded version: The second source operand is a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 64-bit memory location. The first source operand and destination operands are ZMM registers. The destination operand is conditionally updated according to the writemask.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper Bits (MAX\_VL-1:128) of the corresponding register destination are unmodified.

**Operation****VSUBPD (EVEX encoded versions) when src2 operand is a vector register**

(KL, VL) = (2, 128), (4, 256), (8, 512)

IF (VL = 512) AND (EVEX.b = 1)

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN DEST[i+63:i] ← SRC1[i+63:i] - SRC2[i+63:i]

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[63:0] remains unchanged\*

ELSE ; zeroing-masking

DEST[63:0] ← 0

FI;

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VSUBPD (EVEX encoded versions) when src2 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] ← SRC1[i+63:i] - SRC2[63:0];

ELSE EST[i+63:i] ← SRC1[i+63:i] - SRC2[i+63:i];

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[63:0] remains unchanged\*

ELSE ; zeroing-masking

DEST[63:0] ← 0

FI;

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VSUBPD (VEX.256 encoded version)**

DEST[63:0] ← SRC1[63:0] - SRC2[63:0]

DEST[127:64] ← SRC1[127:64] - SRC2[127:64]

DEST[191:128] ← SRC1[191:128] - SRC2[191:128]

DEST[255:192] ← SRC1[255:192] - SRC2[255:192]

DEST[MAX\_VL-1:256] ← 0

**VSUBPD (VEX.128 encoded version)**

DEST[63:0] ← SRC1[63:0] - SRC2[63:0]  
 DEST[127:64] ← SRC1[127:64] - SRC2[127:64]  
 DEST[MAX\_VL-1:128] ← 0

**SUBPD (128-bit Legacy SSE version)**

DEST[63:0] ← DEST[63:0] - SRC[63:0]  
 DEST[127:64] ← DEST[127:64] - SRC[127:64]  
 DEST[MAX\_VL-1:128] (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**

VSUBPD \_\_m512d\_mm512\_sub\_pd (\_\_m512d a, \_\_m512d b);  
 VSUBPD \_\_m512d\_mm512\_mask\_sub\_pd (\_\_m512d s, \_\_mmask8 k, \_\_m512d a, \_\_m512d b);  
 VSUBPD \_\_m512d\_mm512\_maskz\_sub\_pd (\_\_mmask8 k, \_\_m512d a, \_\_m512d b);  
 VSUBPD \_\_m512d\_mm512\_sub\_round\_pd (\_\_m512d a, \_\_m512d b, int);  
 VSUBPD \_\_m512d\_mm512\_mask\_sub\_round\_pd (\_\_m512d s, \_\_mmask8 k, \_\_m512d a, \_\_m512d b, int);  
 VSUBPD \_\_m512d\_mm512\_maskz\_sub\_round\_pd (\_\_mmask8 k, \_\_m512d a, \_\_m512d b, int);  
 VSUBPD \_\_m256d\_mm256\_sub\_pd (\_\_m256d a, \_\_m256d b);  
 VSUBPD \_\_m256d\_mm256\_mask\_sub\_pd (\_\_m256d s, \_\_mmask8 k, \_\_m256d a, \_\_m256d b);  
 VSUBPD \_\_m256d\_mm256\_maskz\_sub\_pd (\_\_mmask8 k, \_\_m256d a, \_\_m256d b);  
 SUBPD \_\_m128d\_mm\_sub\_pd (\_\_m128d a, \_\_m128d b);  
 VSUBPD \_\_m128d\_mm\_mask\_sub\_pd (\_\_m128d s, \_\_mmask8 k, \_\_m128d a, \_\_m128d b);  
 VSUBPD \_\_m128d\_mm\_maskz\_sub\_pd (\_\_mmask8 k, \_\_m128d a, \_\_m128d b);

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal

**Other Exceptions**

VEX-encoded instructions, see Exceptions Type 2.

EVEX-encoded instructions, see Exceptions Type E2.

## SUBPS—Subtract Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
OF 5C /r SUBPS xmm1, xmm2/m128	RM	V/V	SSE	Subtract packed single-precision floating-point values in xmm2/mem from xmm1 and store result in xmm1.
VEX.NDS.128.OF.WIG 5C /r VSUBPS xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Subtract packed single-precision floating-point values in xmm3/mem from xmm2 and stores result in xmm1.
VEX.NDS.256.OF.WIG 5C /r VSUBPS ymm1, ymm2, ymm3/m256	RVM	V/V	AVX	Subtract packed single-precision floating-point values in ymm3/mem from ymm2 and stores result in ymm1.
EVEX.NDS.128.OF.WO 5C /r VSUBPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	FV	V/V	AVX512VL AVX512F	Subtract packed single-precision floating-point values from xmm3/m128/m32bcst to xmm2 and stores result in xmm1 with writemask k1.
EVEX.NDS.256.OF.WO 5C /r VSUBPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	FV	V/V	AVX512VL AVX512F	Subtract packed single-precision floating-point values from ymm3/m256/m32bcst to ymm2 and stores result in ymm1 with writemask k1.
EVEX.NDS.512.OF.WO 5C /r VSUBPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}	FV	V/V	AVX512F	Subtract packed single-precision floating-point values in zmm3/m512/m32bcst from zmm2 and stores result in zmm1 with writemask k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FV	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Performs a SIMD subtract of the packed single-precision floating-point values in the second Source operand from the First Source operand, and stores the packed single-precision floating-point results in the destination operand.

VEX.128 and EVEX.128 encoded versions: The second source operand is an XMM register or an 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAX\_VL-1:128) of the corresponding destination register are zeroed.

VEX.256 and EVEX.256 encoded versions: The second source operand is an YMM register or an 256-bit memory location. The first source operand and destination operands are YMM registers. Bits (MAX\_VL-1:256) of the corresponding destination register are zeroed.

EVEX.512 encoded version: The second source operand is a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32-bit memory location. The first source operand and destination operands are ZMM registers. The destination operand is conditionally updated according to the writemask.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper Bits (MAX\_VL-1:128) of the corresponding register destination are unmodified.



**Operation****VSUBPS (EVEX encoded versions) when src2 operand is a vector register**

(KL, VL) = (4, 128), (8, 256), (16, 512)

IF (VL = 512) AND (EVEX.b = 1)

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\*

THEN DEST[i+31:i] ← SRC1[i+31:i] - SRC2[i+31:i]

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[31:0] remains unchanged\*

ELSE ; zeroing-masking

DEST[31:0] ← 0

FI;

FI;

ENDFOR;

DEST[MAX\_VL-1:VL] ← 0

**VSUBPS (EVEX encoded versions) when src2 operand is a memory source**

(KL, VL) = (4, 128), (8, 256), (16, 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] ← SRC1[i+31:i] - SRC2[31:0];

ELSE DEST[i+31:i] ← SRC1[i+31:i] - SRC2[i+31:i];

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[31:0] remains unchanged\*

ELSE ; zeroing-masking

DEST[31:0] ← 0

FI;

FI;

ENDFOR;

DEST[MAX\_VL-1:VL] ← 0

**VSUBPS (VEX.256 encoded version)**

DEST[31:0] ← SRC1[31:0] - SRC2[31:0]

DEST[63:32] ← SRC1[63:32] - SRC2[63:32]

DEST[95:64] ← SRC1[95:64] - SRC2[95:64]

DEST[127:96] ← SRC1[127:96] - SRC2[127:96]

DEST[159:128] ← SRC1[159:128] - SRC2[159:128]

DEST[191:160] ← SRC1[191:160] - SRC2[191:160]

DEST[223:192] ← SRC1[223:192] - SRC2[223:192]

DEST[255:224] ← SRC1[255:224] - SRC2[255:224].

DEST[MAX\_VL-1:256] ← 0

**VSUBPS (VEX.128 encoded version)**

$\text{DEST}[31:0] \leftarrow \text{SRC1}[31:0] - \text{SRC2}[31:0]$   
 $\text{DEST}[63:32] \leftarrow \text{SRC1}[63:32] - \text{SRC2}[63:32]$   
 $\text{DEST}[95:64] \leftarrow \text{SRC1}[95:64] - \text{SRC2}[95:64]$   
 $\text{DEST}[127:96] \leftarrow \text{SRC1}[127:96] - \text{SRC2}[127:96]$   
 $\text{DEST}[\text{MAX\_VL}-1:128] \leftarrow 0$

**SUBPS (128-bit Legacy SSE version)**

$\text{DEST}[31:0] \leftarrow \text{SRC1}[31:0] - \text{SRC2}[31:0]$   
 $\text{DEST}[63:32] \leftarrow \text{SRC1}[63:32] - \text{SRC2}[63:32]$   
 $\text{DEST}[95:64] \leftarrow \text{SRC1}[95:64] - \text{SRC2}[95:64]$   
 $\text{DEST}[127:96] \leftarrow \text{SRC1}[127:96] - \text{SRC2}[127:96]$   
 $\text{DEST}[\text{MAX\_VL}-1:128]$  (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**

`VSUBPS __m512 __mm512_sub_ps (__m512 a, __m512 b);`  
`VSUBPS __m512 __mm512_mask_sub_ps (__m512 s, __mmask16 k, __m512 a, __m512 b);`  
`VSUBPS __m512 __mm512_maskz_sub_ps (__mmask16 k, __m512 a, __m512 b);`  
`VSUBPS __m512 __mm512_sub_round_ps (__m512 a, __m512 b, int);`  
`VSUBPS __m512 __mm512_mask_sub_round_ps (__m512 s, __mmask16 k, __m512 a, __m512 b, int);`  
`VSUBPS __m512 __mm512_maskz_sub_round_ps (__mmask16 k, __m512 a, __m512 b, int);`  
`VSUBPS __m256 __mm256_sub_ps (__m256 a, __m256 b);`  
`VSUBPS __m256 __mm256_mask_sub_ps (__m256 s, __mmask8 k, __m256 a, __m256 b);`  
`VSUBPS __m256 __mm256_maskz_sub_ps (__mmask16 k, __m256 a, __m256 b);`  
`SUBPS __m128 __mm_sub_ps (__m128 a, __m128 b);`  
`VSUBPS __m128 __mm_mask_sub_ps (__m128 s, __mmask8 k, __m128 a, __m128 b);`  
`VSUBPS __m128 __mm_maskz_sub_ps (__mmask16 k, __m128 a, __m128 b);`

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal

**Other Exceptions**

VEX-encoded instructions, see Exceptions Type 2.

EVEX-encoded instructions, see Exceptions Type E2.

## SUBSD—Subtract Scalar Double-Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 5C /r SUBSD xmm1, xmm2/m64	RM	V/V	SSE2	Subtract the low double-precision floating-point value in xmm2/m64 from xmm1 and store the result in xmm1.
VEX.NDS.128.F2.0F.WIG 5C /r VSUBSD xmm1, xmm2, xmm3/m64	RVM	V/V	AVX	Subtract the low double-precision floating-point value in xmm3/m64 from xmm2 and store the result in xmm1.
EVEX.NDS.LIG.F2.0F.W1 5C /r VSUBSD xmm1 {k1}{z}, xmm2, xmm3/m64{er}	T1S	V/V	AVX512F	Subtract the low double-precision floating-point value in xmm3/m64 from xmm2 and store the result in xmm1 under writemask k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
T1S	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Subtract the low double-precision floating-point value in the second source operand from the first source operand and stores the double-precision floating-point result in the low quadword of the destination operand.

The second source operand can be an XMM register or a 64-bit memory location. The first source and destination operands are XMM registers.

128-bit Legacy SSE version: The destination and first source operand are the same. Bits (MAX\_VL-1:64) of the corresponding destination register remain unchanged.

VEX.128 and EVEX encoded versions: Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (MAX\_VL-1:128) of the destination register are zeroed.

EVEX encoded version: The low quadword element of the destination operand is updated according to the writemask.

Software should ensure VSUBSD is encoded with VEX.L=0. Encoding VSUBSD with VEX.L=1 may encounter unpredictable behavior across different processor generations.

**Operation****VSUBSD (EVEX encoded version)**

```

IF (SRC2 *is register*) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
    THEN  DEST[63:0] ← SRC1[63:0] - SRC2[63:0]
    ELSE
        IF *merging-masking*           ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                         ; zeroing-masking
                THEN DEST[63:0] ← 0
        FI;
FI;
DEST[127:64] ← SRC1[127:64]
DEST[MAX_VL-1:128] ← 0

```

**VSUBSD (VEX.128 encoded version)**

```

DEST[63:0] ← SRC1[63:0] - SRC2[63:0]
DEST[127:64] ← SRC1[127:64]
DEST[MAX_VL-1:128] ← 0

```

**SUBSD (128-bit Legacy SSE version)**

```

DEST[63:0] ← DEST[63:0] - SRC[63:0]
DEST[MAX_VL-1:64] (Unmodified)

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VSUBSD __m128d __mm_mask_sub_sd (__m128d s, __mmask8 k, __m128d a, __m128d b);
VSUBSD __m128d __mm_maskz_sub_sd (__mmask8 k, __m128d a, __m128d b);
VSUBSD __m128d __mm_sub_round_sd (__m128d a, __m128d b, int);
VSUBSD __m128d __mm_mask_sub_round_sd (__m128d s, __mmask8 k, __m128d a, __m128d b, int);
VSUBSD __m128d __mm_maskz_sub_round_sd (__mmask8 k, __m128d a, __m128d b, int);
SUBSD __m128d __mm_sub_sd (__m128d a, __m128d b);

```

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal

**Other Exceptions**

VEX-encoded instructions, see Exceptions Type 3.

EVEX-encoded instructions, see Exceptions Type E3.

## SUBSS—Subtract Scalar Single-Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 5C /r SUBSS xmm1, xmm2/m32	RM	V/V	SSE	Subtract the low single-precision floating-point value in xmm2/m32 from xmm1 and store the result in xmm1.
VEX.NDS.128.F3.0F.WIG 5C /r VSUBSS xmm1, xmm2, xmm3/m32	RVM	V/V	AVX	Subtract the low single-precision floating-point value in xmm3/m32 from xmm2 and store the result in xmm1.
EVEX.NDS.LIG.F3.0F.W0 5C /r VSUBSS xmm1 {k1}{z}, xmm2, xmm3/m32{er}	T1S	V/V	AVX512F	Subtract the low single-precision floating-point value in xmm3/m32 from xmm2 and store the result in xmm1 under writemask k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
T1S	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Subtract the low single-precision floating-point value from the second source operand and the first source operand and store the double-precision floating-point result in the low doubleword of the destination operand.

The second source operand can be an XMM register or a 32-bit memory location. The first source and destination operands are XMM registers.

128-bit Legacy SSE version: The destination and first source operand are the same. Bits (MAX\_VL-1:32) of the corresponding destination register remain unchanged.

VEX.128 and EVEX encoded versions: Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (MAX\_VL-1:128) of the destination register are zeroed.

EVEX encoded version: The low doubleword element of the destination operand is updated according to the writemask.

Software should ensure VSUBSS is encoded with VEX.L=0. Encoding VSUBSD with VEX.L=1 may encounter unpredictable behavior across different processor generations.

**Operation****VSUBSS (EVEX encoded version)**

```

IF (SRC2 *is register*) AND (EVEX.b = 1)
  THEN
    SET_RM(EVEX.RC);
  ELSE
    SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
  THEN  DEST[31:0] ← SRC1[31:0] - SRC2[31:0]
  ELSE
    IF *merging-masking*           ; merging-masking
      THEN *DEST[31:0] remains unchanged*
    ELSE                             ; zeroing-masking
      THEN DEST[31:0] ← 0
    FI;
FI;
DEST[127:32] ← SRC1[127:32]
DEST[MAX_VL-1:128] ← 0

```

**VSUBSS (VEX.128 encoded version)**

```

DEST[31:0] ← SRC1[31:0] - SRC2[31:0]
DEST[127:32] ← SRC1[127:32]
DEST[MAX_VL-1:128] ← 0

```

**SUBSS (128-bit Legacy SSE version)**

```

DEST[31:0] ← DEST[31:0] - SRC[31:0]
DEST[MAX_VL-1:32] (Unmodified)

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VSUBSS __m128 _mm_mask_sub_ss (__m128 s, __mmask8 k, __m128 a, __m128 b);
VSUBSS __m128 _mm_maskz_sub_ss (__mmask8 k, __m128 a, __m128 b);
VSUBSS __m128 _mm_sub_round_ss (__m128 a, __m128 b, int);
VSUBSS __m128 _mm_mask_sub_round_ss (__m128 s, __mmask8 k, __m128 a, __m128 b, int);
VSUBSS __m128 _mm_maskz_sub_round_ss (__mmask8 k, __m128 a, __m128 b, int);
SUBSS __m128 _mm_sub_ss (__m128 a, __m128 b);

```

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal

**Other Exceptions**

VEX-encoded instructions, see Exceptions Type 3.

EVEX-encoded instructions, see Exceptions Type E3.

## UCOMISD—Unordered Compare Scalar Double-Precision Floating-Point Values and Set EFLAGS

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 2E /r UCOMISD xmm1, xmm2/m64	RM	V/V	SSE2	Compare low double-precision floating-point values in xmm1 and xmm2/mem64 and set the EFLAGS flags accordingly.
VEX.128.66.0F.WIG 2E /r VUCOMISD xmm1, xmm2/m64	RM	V/V	AVX	Compare low double-precision floating-point values in xmm1 and xmm2/mem64 and set the EFLAGS flags accordingly.
EVEX.LIG.66.0F.W1 2E /r VUCOMISD xmm1, xmm2/m64{sae}	T1S	V/V	AVX512F	Compare low double-precision floating-point values in xmm1 and xmm2/m64 and set the EFLAGS flags accordingly.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r)	ModRM:r/m (r)	NA	NA
T1S	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

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 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).

Operand 1 is an XMM register; operand 2 can be an XMM register or a 64 bit memory location.

The UCOMISD instruction differs from the COMISD instruction in that it signals a SIMD floating-point invalid operation exception (#I) only when a source operand is an SNaN. The COMISD instruction signals an invalid numeric exception only if a source operand is either an SNaN or a QNaN.

The EFLAGS register is not updated if an unmasked SIMD floating-point exception is generated.

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

Software should ensure VCOMISD is encoded with VEX.L=0. Encoding VCOMISD with VEX.L=1 may encounter unpredictable behavior across different processor generations.

### Operation

#### (V)UCOMISD (all versions)

```
RESULT ← UnorderedCompare(DEST[63:0] <> SRC[63:0]) {
```

```
(* Set EFLAGS *) CASE (RESULT) OF
```

```
  UNORDERED: ZF,PF,CF ← 111;
```

```
  GREATER_THAN: ZF,PF,CF ← 000;
```

```
  LESS_THAN: ZF,PF,CF ← 001;
```

```
  EQUAL: ZF,PF,CF ← 100;
```

```
ESAC;
```

```
OF, AF, SF ← 0; }
```

**Intel C/C++ Compiler Intrinsic Equivalent**

VUCOMISD int \_\_mm\_comi\_round\_sd(\_\_m128d a, \_\_m128d b, int imm, int sae);  
UCOMISD int \_\_mm\_ucomieq\_sd(\_\_m128d a, \_\_m128d b)  
UCOMISD int \_\_mm\_ucomilt\_sd(\_\_m128d a, \_\_m128d b)  
UCOMISD int \_\_mm\_ucomile\_sd(\_\_m128d a, \_\_m128d b)  
UCOMISD int \_\_mm\_ucomigt\_sd(\_\_m128d a, \_\_m128d b)  
UCOMISD int \_\_mm\_ucomige\_sd(\_\_m128d a, \_\_m128d b)  
UCOMISD int \_\_mm\_ucomineq\_sd(\_\_m128d a, \_\_m128d b)

**SIMD Floating-Point Exceptions**

Invalid (if SNaN operands), Denormal

**Other Exceptions**

VEX-encoded instructions, see Exceptions Type 3; additionally

#UD                    If VEX.vvvv != 1111B.

EVEX-encoded instructions, see Exceptions Type E3NF.



## UCOMISS—Unordered Compare Scalar Single-Precision Floating-Point Values and Set EFLAGS

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
OF 2E /r UCOMISS xmm1, xmm2/m32	RM	V/V	SSE	Compare low single-precision floating-point values in xmm1 and xmm2/mem32 and set the EFLAGS flags accordingly.
VEX.128.OF.WIG 2E /r VUCOMISS xmm1, xmm2/m32	RM	V/V	AVX	Compare low single-precision floating-point values in xmm1 and xmm2/mem32 and set the EFLAGS flags accordingly.
EVEX.LIG.OF.WO 2E /r VUCOMISS xmm1, xmm2/m32{sae}	T1S	V/V	AVX512F	Compare low single-precision floating-point values in xmm1 and xmm2/mem32 and set the EFLAGS flags accordingly.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r)	ModRM:r/m (r)	NA	NA
T1S	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Compares the single-precision floating-point values in the low doublewords 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).

Operand 1 is an XMM register; operand 2 can be an XMM register or a 32 bit memory location.

The UCOMISS instruction differs from the COMISS instruction in that it signals a SIMD floating-point invalid operation exception (#1) only if a source operand is an SNaN. The COMISS instruction signals an invalid numeric 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: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b, otherwise instructions will #UD.

Software should ensure VCOMISS is encoded with VEX.L=0. Encoding VCOMISS with VEX.L=1 may encounter unpredictable behavior across different processor generations.

### Operation

#### (V)UCOMISS (all versions)

```

RESULT ← UnorderedCompare(DEST[31:0] <> SRC[31:0]) {
(* Set EFLAGS *) CASE (RESULT) OF
  UNORDERED: ZF,PF,CF ← 111;
  GREATER_THAN: ZF,PF,CF ← 000;
  LESS_THAN: ZF,PF,CF ← 001;
  EQUAL: ZF,PF,CF ← 100;
ESAC;
OF, AF, SF ← 0; }

```

**Intel C/C++ Compiler Intrinsic Equivalent**

VUCOMISS int \_mm\_comi\_round\_ss(\_\_m128 a, \_\_m128 b, int imm, int sae);  
 UCOMISS int \_mm\_ucomieq\_ss(\_\_m128 a, \_\_m128 b);  
 UCOMISS int \_mm\_ucomilt\_ss(\_\_m128 a, \_\_m128 b);  
 UCOMISS int \_mm\_ucomile\_ss(\_\_m128 a, \_\_m128 b);  
 UCOMISS int \_mm\_ucomigt\_ss(\_\_m128 a, \_\_m128 b);  
 UCOMISS int \_mm\_ucomige\_ss(\_\_m128 a, \_\_m128 b);  
 UCOMISS int \_mm\_ucomineq\_ss(\_\_m128 a, \_\_m128 b);

**SIMD Floating-Point Exceptions**

Invalid (if SNaN Operands), Denormal

**Other Exceptions**

VEX-encoded instructions, see Exceptions Type 3; additionally

#UD If VEX.vvvv != 1111B.

EVEX-encoded instructions, see Exceptions Type E3NF.

## UNPCKHPD—Unpack and Interleave High Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 15 /r UNPCKHPD xmm1, xmm2/m128	RM	V/V	SSE2	Unpacks and Interleaves double-precision floating-point values from high quadwords of xmm1 and xmm2/m128.
VEX.NDS.128.66.0F.WIG 15 /r VUNPCKHPD xmm1,xmm2, xmm3/m128	RVM	V/V	AVX	Unpacks and Interleaves double-precision floating-point values from high quadwords of xmm2 and xmm3/m128.
VEX.NDS.256.66.0F.WIG 15 /r VUNPCKHPD ymm1,ymm2, ymm3/m256	RVM	V/V	AVX	Unpacks and Interleaves double-precision floating-point values from high quadwords of ymm2 and ymm3/m256.
EVEX.NDS.128.66.0F.W1 15 /r VUNPCKHPD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	FV	V/V	AVX512VL AVX512F	Unpacks and Interleaves double precision floating-point values from high quadwords of xmm2 and xmm3/m128/m64bcst subject to writemask k1.
EVEX.NDS.256.66.0F.W1 15 /r VUNPCKHPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	FV	V/V	AVX512VL AVX512F	Unpacks and Interleaves double precision floating-point values from high quadwords of ymm2 and ymm3/m256/m64bcst subject to writemask k1.
EVEX.NDS.512.66.0F.W1 15 /r VUNPCKHPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	FV	V/V	AVX512F	Unpacks and Interleaves double-precision floating-point values from high quadwords of zmm2 and zmm3/m512/m64bcst subject to writemask k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FV	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Performs an interleaved unpack of the high double-precision floating-point values from the first source operand and the second source operand. See Figure 4-15 in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B.

**128-bit Legacy SSE version:** The second source can be an XMM register or a 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAX\_VL-1:128) of the corresponding ZMM register destination are unmodified. When unpacking from a memory operand, an implementation may fetch only the appropriate 64 bits; however, alignment to 16-byte boundary and normal segment checking will still be enforced.

**VEX.128 encoded version:** The first source operand is a XMM register. The second source operand can be a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper bits (MAX\_VL-1:128) of the corresponding ZMM register destination are zeroed.

**VEX.256 encoded version:** The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.

**EVEX.512 encoded version:** The first source operand is a ZMM register. The second source operand is a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

EVEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register, a 256-bit memory location, or a 256-bit vector broadcasted from a 64-bit memory location. The destination operand is a YMM register, conditionally updated using writemask k1.

EVEX.128 encoded version: The first source operand is a XMM register. The second source operand is a XMM register, a 128-bit memory location, or a 128-bit vector broadcasted from a 64-bit memory location. The destination operand is a XMM register, conditionally updated using writemask k1.

### Operation

#### VUNPCKHPD (EVEX encoded versions when SRC2 is a register)

(KL, VL) = (2, 128), (4, 256), (8, 512)

IF VL >= 128

    TMP\_DEST[63:0] ← SRC1[127:64]

    TMP\_DEST[127:64] ← SRC2[127:64]

FI;

IF VL >= 256

    TMP\_DEST[191:128] ← SRC1[255:192]

    TMP\_DEST[255:192] ← SRC2[255:192]

FI;

IF VL >= 512

    TMP\_DEST[319:256] ← SRC1[383:320]

    TMP\_DEST[383:320] ← SRC2[383:320]

    TMP\_DEST[447:384] ← SRC1[511:448]

    TMP\_DEST[511:448] ← SRC2[511:448]

FI;

FOR j ← 0 TO KL-1

    i ← j \* 64

    IF k1[j] OR \*no writemask\*

        THEN DEST[i+63:i] ← TMP\_DEST[i+63:i]

    ELSE

        IF \*merging-masking\* ; merging-masking

            THEN \*DEST[i+63:i] remains unchanged\*

        ELSE \*zeroing-masking\* ; zeroing-masking

            DEST[i+63:i] ← 0

    FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VUNPCKHPD (EVEX encoded version when SRC2 is memory)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 64
  IF (EVEX.b = 1)
    THEN TMP_SRC2[i+63:i] ← SRC2[63:0]
    ELSE TMP_SRC2[i+63:i] ← SRC2[i+63:i]
  FI;
ENDFOR;
IF VL >= 128
  TMP_DEST[63:0] ← SRC1[127:64]
  TMP_DEST[127:64] ← TMP_SRC2[127:64]
FI;
IF VL >= 256
  TMP_DEST[191:128] ← SRC1[255:192]
  TMP_DEST[255:192] ← TMP_SRC2[255:192]
FI;
IF VL >= 512
  TMP_DEST[319:256] ← SRC1[383:320]
  TMP_DEST[383:320] ← TMP_SRC2[383:320]
  TMP_DEST[447:384] ← SRC1[511:448]
  TMP_DEST[511:448] ← TMP_SRC2[511:448]
FI;

FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+63:i] remains unchanged*
        ELSE *zeroing-masking* ; zeroing-masking
          DEST[i+63:i] ← 0
      FI
  FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VUNPCKHPD (VEX.256 encoded version)**

```

DEST[63:0] ← SRC1[127:64]
DEST[127:64] ← SRC2[127:64]
DEST[191:128] ← SRC1[255:192]
DEST[255:192] ← SRC2[255:192]
DEST[MAX_VL-1:256] ← 0

```

**VUNPCKHPD (VEX.128 encoded version)**

```

DEST[63:0] ← SRC1[127:64]
DEST[127:64] ← SRC2[127:64]
DEST[MAX_VL-1:128] ← 0

```

**UNPCKHPD (128-bit Legacy SSE version)**

```

DEST[63:0] ← SRC1[127:64]
DEST[127:64] ← SRC2[127:64]
DEST[MAX_VL-1:128] (Unmodified)

```

**Intel C/C++ Compiler Intrinsic Equivalent**

VUNPCKHPD \_\_m512d \_\_mm512\_unpackhi\_pd(\_\_m512d a, \_\_m512d b);  
 VUNPCKHPD \_\_m512d \_\_mm512\_mask\_unpackhi\_pd(\_\_m512d s, \_\_mmask8 k, \_\_m512d a, \_\_m512d b);  
 VUNPCKHPD \_\_m512d \_\_mm512\_maskz\_unpackhi\_pd(\_\_mmask8 k, \_\_m512d a, \_\_m512d b);  
 VUNPCKHPD \_\_m256d \_\_mm256\_unpackhi\_pd(\_\_m256d a, \_\_m256d b);  
 VUNPCKHPD \_\_m256d \_\_mm256\_mask\_unpackhi\_pd(\_\_m256d s, \_\_mmask8 k, \_\_m256d a, \_\_m256d b);  
 VUNPCKHPD \_\_m256d \_\_mm256\_maskz\_unpackhi\_pd(\_\_mmask8 k, \_\_m256d a, \_\_m256d b);  
 UNPCKHPD \_\_m128d \_\_mm\_unpackhi\_pd(\_\_m128d a, \_\_m128d b);  
 VUNPCKHPD \_\_m128d \_\_mm\_mask\_unpackhi\_pd(\_\_m128d s, \_\_mmask8 k, \_\_m128d a, \_\_m128d b);  
 VUNPCKHPD \_\_m128d \_\_mm\_maskz\_unpackhi\_pd(\_\_mmask8 k, \_\_m128d a, \_\_m128d b);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

Non-EVEX-encoded instructions, see Exceptions Type 4.

EVEX-encoded instructions, see Exceptions Type E4NF.

## UNPCKHPS—Unpack and Interleave High Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
OF 15 /r UNPCKHPS xmm1, xmm2/m128	RM	V/V	SSE	Unpacks and Interleaves single-precision floating-point values from high quadwords of xmm1 and xmm2/m128.
VEX.NDS.128.OF.WIG 15 /r VUNPCKHPS xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Unpacks and Interleaves single-precision floating-point values from high quadwords of xmm2 and xmm3/m128.
VEX.NDS.256.OF.WIG 15 /r VUNPCKHPS ymm1, ymm2, ymm3/m256	RVM	V/V	AVX	Unpacks and Interleaves single-precision floating-point values from high quadwords of ymm2 and ymm3/m256.
EVEX.NDS.128.OF.W0 15 /r VUNPCKHPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	FV	V/V	AVX512VL AVX512F	Unpacks and Interleaves single-precision floating-point values from high quadwords of xmm2 and xmm3/m128/m32bcst and write result to xmm1 subject to writemask k1.
EVEX.NDS.256.OF.W0 15 /r VUNPCKHPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	FV	V/V	AVX512VL AVX512F	Unpacks and Interleaves single-precision floating-point values from high quadwords of ymm2 and ymm3/m256/m32bcst and write result to ymm1 subject to writemask k1.
EVEX.NDS.512.OF.W0 15 /r VUNPCKHPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	FV	V/V	AVX512F	Unpacks and Interleaves single-precision floating-point values from high quadwords of zmm2 and zmm3/m512/m32bcst and write result to zmm1 subject to writemask k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FV	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Performs an interleaved unpack of the high single-precision floating-point values from the first source operand and the second source operand.

128-bit Legacy SSE version: The second source can be an XMM register or a 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAX\_VL-1:128) of the corresponding ZMM register destination are unmodified. When unpacking from a memory operand, an implementation may fetch only the appropriate 64 bits; however, alignment to 16-byte boundary and normal segment checking will still be enforced.

VEX.128 encoded version: The first source operand is a XMM register. The second source operand can be a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper bits (MAX\_VL-1:128) of the corresponding ZMM register destination are zeroed.

VEX.256 encoded version: The second source operand is an YMM register or a 256-bit memory location. The first source operand and destination operands are YMM registers.

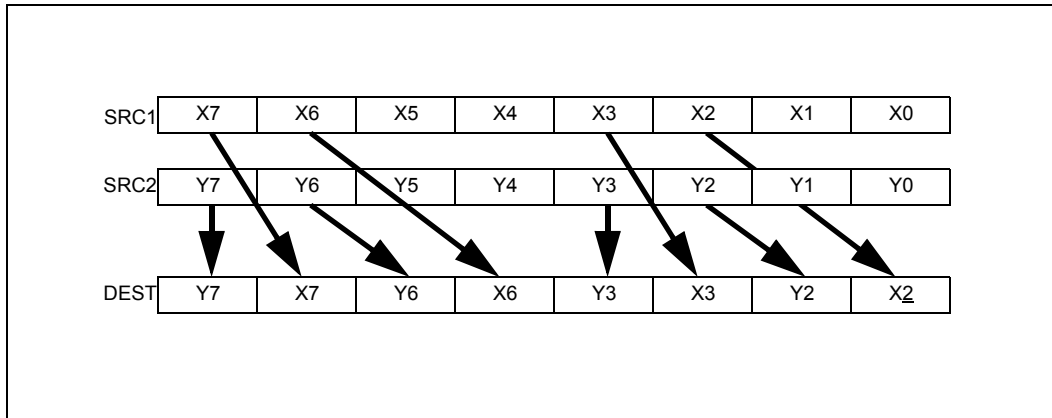


Figure 4-22. VUNPCKHPS Operation

EVEX.512 encoded version: The first source operand is a ZMM register. The second source operand is a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

EVEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register, a 256-bit memory location, or a 256-bit vector broadcasted from a 32-bit memory location. The destination operand is a YMM register, conditionally updated using writemask k1.

EVEX.128 encoded version: The first source operand is a XMM register. The second source operand is a XMM register, a 128-bit memory location, or a 128-bit vector broadcasted from a 32-bit memory location. The destination operand is a XMM register, conditionally updated using writemask k1.

### Operation

#### VUNPCKHPS (EVEX encoded version when SRC2 is a register)

(KL, VL) = (4, 128), (8, 256), (16, 512)

IF VL >= 128

```
TMP_DEST[31:0] ← SRC1[95:64]
TMP_DEST[63:32] ← SRC2[95:64]
TMP_DEST[95:64] ← SRC1[127:96]
TMP_DEST[127:96] ← SRC2[127:96]
```

FI;

IF VL >= 256

```
TMP_DEST[159:128] ← SRC1[223:192]
TMP_DEST[191:160] ← SRC2[223:192]
TMP_DEST[223:192] ← SRC1[255:224]
TMP_DEST[255:224] ← SRC2[255:224]
```

FI;

IF VL >= 512

```
TMP_DEST[287:256] ← SRC1[351:320]
TMP_DEST[319:288] ← SRC2[351:320]
TMP_DEST[351:320] ← SRC1[383:352]
TMP_DEST[383:352] ← SRC2[383:352]
TMP_DEST[415:384] ← SRC1[479:448]
TMP_DEST[447:416] ← SRC2[479:448]
TMP_DEST[479:448] ← SRC1[511:480]
TMP_DEST[511:480] ← SRC2[511:480]
```

FI;



```

FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
  ELSE
    IF *merging-masking*           ; merging-masking
      THEN *DEST[i+31:i] remains unchanged*
    ELSE *zeroing-masking*         ; zeroing-masking
      DEST[i+31:i] ← 0
    FI
  FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VUNPCKHPS (EVEX encoded version when SRC2 is memory)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 32
  IF (EVEX.b = 1)
    THEN TMP_SRC2[i+31:i] ← SRC2[31:0]
    ELSE TMP_SRC2[i+31:i] ← SRC2[i+31:i]
  FI;
ENDFOR;
IF VL >= 128
  TMP_DEST[31:0] ← SRC1[95:64]
  TMP_DEST[63:32] ← TMP_SRC2[95:64]
  TMP_DEST[95:64] ← SRC1[127:96]
  TMP_DEST[127:96] ← TMP_SRC2[127:96]
FI;
IF VL >= 256
  TMP_DEST[159:128] ← SRC1[223:192]
  TMP_DEST[191:160] ← TMP_SRC2[223:192]
  TMP_DEST[223:192] ← SRC1[255:224]
  TMP_DEST[255:224] ← TMP_SRC2[255:224]
FI;
IF VL >= 512
  TMP_DEST[287:256] ← SRC1[351:320]
  TMP_DEST[319:288] ← TMP_SRC2[351:320]
  TMP_DEST[351:320] ← SRC1[383:352]
  TMP_DEST[383:352] ← TMP_SRC2[383:352]
  TMP_DEST[415:384] ← SRC1[479:448]
  TMP_DEST[447:416] ← TMP_SRC2[479:448]
  TMP_DEST[479:448] ← SRC1[511:480]
  TMP_DEST[511:480] ← TMP_SRC2[511:480]
FI;
FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
  ELSE
    IF *merging-masking*           ; merging-masking
      THEN *DEST[i+31:i] remains unchanged*
    ELSE *zeroing-masking*         ; zeroing-masking
      DEST[i+31:i] ← 0
    FI
  FI;
ENDFOR

```

FI

```

FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VUNPCKHPS (VEX.256 encoded version)**

```

DEST[31:0] ← SRC1[95:64]
DEST[63:32] ← SRC2[95:64]
DEST[95:64] ← SRC1[127:96]
DEST[127:96] ← SRC2[127:96]
DEST[159:128] ← SRC1[223:192]
DEST[191:160] ← SRC2[223:192]
DEST[223:192] ← SRC1[255:224]
DEST[255:224] ← SRC2[255:224]
DEST[MAX_VL-1:256] ← 0

```

**VUNPCKHPS (VEX.128 encoded version)**

```

DEST[31:0] ← SRC1[95:64]
DEST[63:32] ← SRC2[95:64]
DEST[95:64] ← SRC1[127:96]
DEST[127:96] ← SRC2[127:96]
DEST[MAX_VL-1:128] ← 0

```

**UNPCKHPS (128-bit Legacy SSE version)**

```

DEST[31:0] ← SRC1[95:64]
DEST[63:32] ← SRC2[95:64]
DEST[95:64] ← SRC1[127:96]
DEST[127:96] ← SRC2[127:96]
DEST[MAX_VL-1:128] (Unmodified)

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VUNPCKHPS __m512 _mm512_unpackhi_ps( __m512 a, __m512 b);
VUNPCKHPS __m512 _mm512_mask_unpackhi_ps(__m512 s, __mmask16 k, __m512 a, __m512 b);
VUNPCKHPS __m512 _mm512_maskz_unpackhi_ps(__mmask16 k, __m512 a, __m512 b);
VUNPCKHPS __m256 _mm256_unpackhi_ps( __m256 a, __m256 b);
VUNPCKHPS __m256 _mm256_mask_unpackhi_ps(__m256 s, __mmask8 k, __m256 a, __m256 b);
VUNPCKHPS __m256 _mm256_maskz_unpackhi_ps(__mmask8 k, __m256 a, __m256 b);
UNPCKHPS __m128 _mm_unpackhi_ps( __m128 a, __m128 b);
VUNPCKHPS __m128 _mm_mask_unpackhi_ps(__m128 s, __mmask8 k, __m128 a, __m128 b);
VUNPCKHPS __m128 _mm_maskz_unpackhi_ps(__mmask8 k, __m128 a, __m128 b);

```

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

Non-EVEX-encoded instructions, see Exceptions Type 4.

EVEX-encoded instructions, see Exceptions Type E4NF.

## UNPCKLPD—Unpack and Interleave Low Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 14 /r UNPCKLPD xmm1, xmm2/m128	RM	V/V	SSE2	Unpacks and interleaves double-precision floating-point values from low quadwords of xmm1 and xmm2/m128.
VEX.NDS.128.66.0F.WIG 14 /r VUNPCKLPD xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Unpacks and interleaves double-precision floating-point values from low quadwords of xmm2 and xmm3/m128.
VEX.NDS.256.66.0F.WIG 14 /r VUNPCKLPD ymm1, ymm2, ymm3/m256	RVM	V/V	AVX	Unpacks and interleaves double-precision floating-point values from low quadwords of ymm2 and ymm3/m256.
EVEX.NDS.128.66.0F.W1 14 /r VUNPCKLPD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	FV	V/V	AVX512VL AVX512F	Unpacks and interleaves double precision floating-point values from low quadwords of xmm2 and xmm3/m128/m64bcst subject to write mask k1.
EVEX.NDS.256.66.0F.W1 14 /r VUNPCKLPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	FV	V/V	AVX512VL AVX512F	Unpacks and interleaves double precision floating-point values from low quadwords of ymm2 and ymm3/m256/m64bcst subject to write mask k1.
EVEX.NDS.512.66.0F.W1 14 /r VUNPCKLPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	FV	V/V	AVX512F	Unpacks and interleaves double-precision floating-point values from low quadwords of zmm2 and zmm3/m512/m64bcst subject to write mask k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FV	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Performs an interleaved unpack of the low double-precision floating-point values from the first source operand and the second source operand.

128-bit Legacy SSE version: The second source can be an XMM register or a 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAX\_VL-1:128) of the corresponding ZMM register destination are unmodified. When unpacking from a memory operand, an implementation may fetch only the appropriate 64 bits; however, alignment to 16-byte boundary and normal segment checking will still be enforced.

VEX.128 encoded version: The first source operand is a XMM register. The second source operand can be a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper bits (MAX\_VL-1:128) of the corresponding ZMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.

EVEX.512 encoded version: The first source operand is a ZMM register. The second source operand is a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

EVEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register, a 256-bit memory location, or a 256-bit vector broadcasted from a 64-bit memory location. The destination operand is a YMM register, conditionally updated using writemask k1.

EVEX.128 encoded version: The first source operand is an XMM register. The second source operand is a XMM register, a 128-bit memory location, or a 128-bit vector broadcasted from a 64-bit memory location. The destination operand is a XMM register, conditionally updated using writemask k1.

### Operation

#### VUNPCKLPD (EVEX encoded versions when SRC2 is a register)

(KL, VL) = (2, 128), (4, 256), (8, 512)

IF VL >= 128

    TMP\_DEST[63:0] ← SRC1[63:0]

    TMP\_DEST[127:64] ← SRC2[63:0]

FI;

IF VL >= 256

    TMP\_DEST[191:128] ← SRC1[191:128]

    TMP\_DEST[255:192] ← SRC2[191:128]

FI;

IF VL >= 512

    TMP\_DEST[319:256] ← SRC1[319:256]

    TMP\_DEST[383:320] ← SRC2[319:256]

    TMP\_DEST[447:384] ← SRC1[447:384]

    TMP\_DEST[511:448] ← SRC2[447:384]

FI;

FOR j ← 0 TO KL-1

    i ← j \* 64

    IF k1[j] OR \*no writemask\*

        THEN DEST[i+63:i] ← TMP\_DEST[i+63:i]

    ELSE

        IF \*merging-masking\* ; merging-masking

            THEN \*DEST[i+63:i] remains unchanged\*

        ELSE \*zeroing-masking\* ; zeroing-masking

            DEST[i+63:i] ← 0

    FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VUNPCKLPD (EVEX encoded version when SRC2 is memory)**

```

(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
  i ← j * 64
  IF (EVEX.b = 1)
    THEN TMP_SRC2[i+63:i] ← SRC2[63:0]
    ELSE TMP_SRC2[i+63:i] ← SRC2[i+63:i]
  FI;
ENDFOR;
IF VL >= 128
  TMP_DEST[63:0] ← SRC1[63:0]
  TMP_DEST[127:64] ← TMP_SRC2[63:0]
FI;
IF VL >= 256
  TMP_DEST[191:128] ← SRC1[191:128]
  TMP_DEST[255:192] ← TMP_SRC2[191:128]
FI;
IF VL >= 512
  TMP_DEST[319:256] ← SRC1[319:256]
  TMP_DEST[383:320] ← TMP_SRC2[319:256]
  TMP_DEST[447:384] ← SRC1[447:384]
  TMP_DEST[511:448] ← TMP_SRC2[447:384]
FI;
FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+63:i] remains unchanged*
        ELSE *zeroing-masking* ; zeroing-masking
          DEST[i+63:i] ← 0
      FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VUNPCKLPD (VEX.256 encoded version)**

```

DEST[63:0] ← SRC1[63:0]
DEST[127:64] ← SRC2[63:0]
DEST[191:128] ← SRC1[191:128]
DEST[255:192] ← SRC2[191:128]
DEST[MAX_VL-1:256] ← 0

```

**VUNPCKLPD (VEX.128 encoded version)**

```

DEST[63:0] ← SRC1[63:0]
DEST[127:64] ← SRC2[63:0]
DEST[MAX_VL-1:128] ← 0

```

**UNPCKLPD (128-bit Legacy SSE version)**

```

DEST[63:0] ← SRC1[63:0]
DEST[127:64] ← SRC2[63:0]
DEST[MAX_VL-1:128] (Unmodified)

```

**Intel C/C++ Compiler Intrinsic Equivalent**

VUNPCKLPD \_\_m512d \_\_mm512\_unpacklo\_pd( \_\_m512d a, \_\_m512d b);  
 VUNPCKLPD \_\_m512d \_\_mm512\_mask\_unpacklo\_pd(\_\_m512d s, \_\_mmask8 k, \_\_m512d a, \_\_m512d b);  
 VUNPCKLPD \_\_m512d \_\_mm512\_maskz\_unpacklo\_pd(\_\_mmask8 k, \_\_m512d a, \_\_m512d b);  
 VUNPCKLPD \_\_m256d \_\_mm256\_unpacklo\_pd(\_\_m256d a, \_\_m256d b)  
 VUNPCKLPD \_\_m256d \_\_mm256\_mask\_unpacklo\_pd(\_\_m256d s, \_\_mmask8 k, \_\_m256d a, \_\_m256d b);  
 VUNPCKLPD \_\_m256d \_\_mm256\_maskz\_unpacklo\_pd(\_\_mmask8 k, \_\_m256d a, \_\_m256d b);  
 UNPCKLPD \_\_m128d \_\_mm\_unpacklo\_pd(\_\_m128d a, \_\_m128d b)  
 VUNPCKLPD \_\_m128d \_\_mm\_mask\_unpacklo\_pd(\_\_m128d s, \_\_mmask8 k, \_\_m128d a, \_\_m128d b);  
 VUNPCKLPD \_\_m128d \_\_mm\_maskz\_unpacklo\_pd(\_\_mmask8 k, \_\_m128d a, \_\_m128d b);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

Non-EVEX-encoded instructions, see Exceptions Type 4.

EVEX-encoded instructions, see Exceptions Type E4NF.

## UNPCKLPS—Unpack and Interleave Low Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
OF 14 /r UNPCKLPS xmm1, xmm2/m128	RM	V/V	SSE	Unpacks and Interleaves single-precision floating-point values from low quadwords of xmm1 and xmm2/m128.
VEX.NDS.128.OF.WIG 14 /r VUNPCKLPS xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Unpacks and Interleaves single-precision floating-point values from low quadwords of xmm2 and xmm3/m128.
VEX.NDS.256.OF.WIG 14 /r VUNPCKLPS ymm1, ymm2, ymm3/m256	RVM	V/V	AVX	Unpacks and Interleaves single-precision floating-point values from low quadwords of ymm2 and ymm3/m256.
EVEX.NDS.128.OF.W0 14 /r VUNPCKLPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	FV	V/V	AVX512VL AVX512F	Unpacks and Interleaves single-precision floating-point values from low quadwords of xmm2 and xmm3/mem and write result to xmm1 subject to write mask k1.
EVEX.NDS.256.OF.W0 14 /r VUNPCKLPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	FV	V/V	AVX512VL AVX512F	Unpacks and Interleaves single-precision floating-point values from low quadwords of ymm2 and ymm3/mem and write result to ymm1 subject to write mask k1.
EVEX.NDS.512.OF.W0 14 /r VUNPCKLPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	FV	V/V	AVX512F	Unpacks and Interleaves single-precision floating-point values from low quadwords of zmm2 and zmm3/m512/m32bcst and write result to zmm1 subject to write mask k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FV	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Performs an interleaved unpack of the low single-precision floating-point values from the first source operand and the second source operand.

128-bit Legacy SSE version: The second source can be an XMM register or a 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAX\_VL-1:128) of the corresponding ZMM register destination are unmodified. When unpacking from a memory operand, an implementation may fetch only the appropriate 64 bits; however, alignment to 16-byte boundary and normal segment checking will still be enforced.

VEX.128 encoded version: The first source operand is a XMM register. The second source operand can be a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper bits (MAX\_VL-1:128) of the corresponding ZMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.

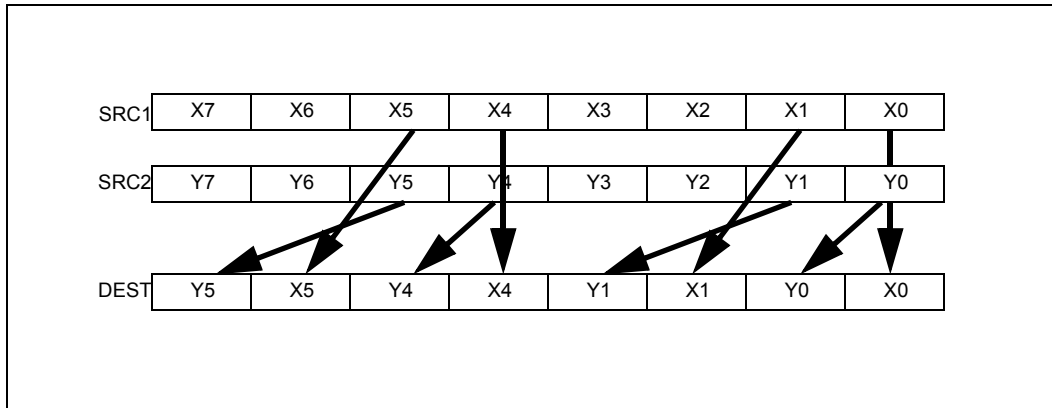


Figure 4-23. VUNPCKLPS Operation

EVEX.512 encoded version: The first source operand is a ZMM register. The second source operand is a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

EVEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register, a 256-bit memory location, or a 256-bit vector broadcasted from a 32-bit memory location. The destination operand is a YMM register, conditionally updated using writemask k1.

EVEX.128 encoded version: The first source operand is an XMM register. The second source operand is a XMM register, a 128-bit memory location, or a 128-bit vector broadcasted from a 32-bit memory location. The destination operand is a XMM register, conditionally updated using writemask k1.

### Operation

#### VUNPCKLPS (EVEX encoded version when SRC2 is a ZMM register)

(KL, VL) = (4, 128), (8, 256), (16, 512)

IF VL >= 128

```
TMP_DEST[31:0] ← SRC1[31:0]
TMP_DEST[63:32] ← SRC2[31:0]
TMP_DEST[95:64] ← SRC1[63:32]
TMP_DEST[127:96] ← SRC2[63:32]
```

FI;

IF VL >= 256

```
TMP_DEST[159:128] ← SRC1[159:128]
TMP_DEST[191:160] ← SRC2[159:128]
TMP_DEST[223:192] ← SRC1[191:160]
TMP_DEST[255:224] ← SRC2[191:160]
```

FI;

IF VL >= 512

```
TMP_DEST[287:256] ← SRC1[287:256]
TMP_DEST[319:288] ← SRC2[287:256]
TMP_DEST[351:320] ← SRC1[319:288]
TMP_DEST[383:352] ← SRC2[319:288]
TMP_DEST[415:384] ← SRC1[415:384]
TMP_DEST[447:416] ← SRC2[415:384]
TMP_DEST[479:448] ← SRC1[447:416]
TMP_DEST[511:480] ← SRC2[447:416]
```

FI;

FOR j ← 0 TO KL-1

i ← j \* 32



```

IF k1[j] OR *no writemask*
  THEN DEST[j+31:i] ← TMP_DEST[j+31:i]
  ELSE
    IF *merging-masking*           ; merging-masking
      THEN *DEST[j+31:i] remains unchanged*
    ELSE *zeroing-masking*         ; zeroing-masking
      DEST[j+31:i] ← 0
    FI
  FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VUNPCKLPS (EVEX encoded version when SRC2 is memory)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 31
  IF (EVEX.b = 1)
    THEN TMP_SRC2[j+31:i] ← SRC2[31:0]
    ELSE TMP_SRC2[j+31:i] ← SRC2[j+31:i]
  FI;
ENDFOR;
IF VL >= 128
  TMP_DEST[31:0] ← SRC1[31:0]
  TMP_DEST[63:32] ← TMP_SRC2[31:0]
  TMP_DEST[95:64] ← SRC1[63:32]
  TMP_DEST[127:96] ← TMP_SRC2[63:32]
FI;
IF VL >= 256
  TMP_DEST[159:128] ← SRC1[159:128]
  TMP_DEST[191:160] ← TMP_SRC2[159:128]
  TMP_DEST[223:192] ← SRC1[191:160]
  TMP_DEST[255:224] ← TMP_SRC2[191:160]
FI;
IF VL >= 512
  TMP_DEST[287:256] ← SRC1[287:256]
  TMP_DEST[319:288] ← TMP_SRC2[287:256]
  TMP_DEST[351:320] ← SRC1[319:288]
  TMP_DEST[383:352] ← TMP_SRC2[319:288]
  TMP_DEST[415:384] ← SRC1[415:384]
  TMP_DEST[447:416] ← TMP_SRC2[415:384]
  TMP_DEST[479:448] ← SRC1[447:416]
  TMP_DEST[511:480] ← TMP_SRC2[447:416]
FI;
FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[j+31:i] ← TMP_DEST[j+31:i]
    ELSE
      IF *merging-masking*           ; merging-masking
        THEN *DEST[j+31:i] remains unchanged*
      ELSE *zeroing-masking*         ; zeroing-masking
        DEST[j+31:i] ← 0
      FI
    FI;
  FI;
ENDFOR

```

ENDFOR  
 DEST[MAX\_VL-1:VL] ← 0

#### UNPCKLPS (VEX.256 encoded version)

DEST[31:0] ← SRC1[31:0]  
 DEST[63:32] ← SRC2[31:0]  
 DEST[95:64] ← SRC1[63:32]  
 DEST[127:96] ← SRC2[63:32]  
 DEST[159:128] ← SRC1[159:128]  
 DEST[191:160] ← SRC2[159:128]  
 DEST[223:192] ← SRC1[191:160]  
 DEST[255:224] ← SRC2[191:160]  
 DEST[MAX\_VL-1:256] ← 0

#### VUNPCKLPS (VEX.128 encoded version)

DEST[31:0] ← SRC1[31:0]  
 DEST[63:32] ← SRC2[31:0]  
 DEST[95:64] ← SRC1[63:32]  
 DEST[127:96] ← SRC2[63:32]  
 DEST[MAX\_VL-1:128] ← 0

#### UNPCKLPS (128-bit Legacy SSE version)

DEST[31:0] ← SRC1[31:0]  
 DEST[63:32] ← SRC2[31:0]  
 DEST[95:64] ← SRC1[63:32]  
 DEST[127:96] ← SRC2[63:32]  
 DEST[MAX\_VL-1:128] (Unmodified)

#### Intel C/C++ Compiler Intrinsic Equivalent

VUNPCKLPS \_\_m512 \_\_mm512\_unpacklo\_ps(\_\_m512 a, \_\_m512 b);  
 VUNPCKLPS \_\_m512 \_\_mm512\_mask\_unpacklo\_ps(\_\_m512 s, \_\_mmask16 k, \_\_m512 a, \_\_m512 b);  
 VUNPCKLPS \_\_m512 \_\_mm512\_maskz\_unpacklo\_ps(\_\_mmask16 k, \_\_m512 a, \_\_m512 b);  
 VUNPCKLPS \_\_m256 \_\_mm256\_unpacklo\_ps (\_\_m256 a, \_\_m256 b);  
 VUNPCKLPS \_\_m256 \_\_mm256\_mask\_unpacklo\_ps(\_\_m256 s, \_\_mmask8 k, \_\_m256 a, \_\_m256 b);  
 VUNPCKLPS \_\_m256 \_\_mm256\_maskz\_unpacklo\_ps(\_\_mmask8 k, \_\_m256 a, \_\_m256 b);  
 UNPCKLPS \_\_m128 \_\_mm\_unpacklo\_ps (\_\_m128 a, \_\_m128 b);  
 VUNPCKLPS \_\_m128 \_\_mm\_mask\_unpacklo\_ps(\_\_m128 s, \_\_mmask8 k, \_\_m128 a, \_\_m128 b);  
 VUNPCKLPS \_\_m128 \_\_mm\_maskz\_unpacklo\_ps(\_\_mmask8 k, \_\_m128 a, \_\_m128 b);

#### SIMD Floating-Point Exceptions

None

#### Other Exceptions

Non-EVEX-encoded instructions, see Exceptions Type 4.  
 EVEX-encoded instructions, see Exceptions Type E4NF.



## 10. Updates to Chapter 5, Volume 2C

Change bars show changes to Chapter 5 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2C: Instruction Set Reference, V-Z*.

Changes include minor updates to reflect volume addition, arrangement of information and document order numbers. Added Ternary Bit Vector Logic Table for reference at beginning of the chapter. Addition of Intel® AVX-512 instructions. The following instructions have been added and/or updated: VALIGND/VALIGNQ, VBLENDMPD/VBLENDMPS, VBROADCAST, VPBROADCASTM, VCOMPRESSPD, VCOMPRESSPS, VCVTPD2QQ, VCVTPD2UDQ, VCVTPD2UQQ, VCVTPH2PS, VCVTPS2PH, VCVTPS2UDQ, VCVTPS2QQ, VCVTPS2UQQ, VCVTQQ2PD, VCVTQQ2PS, VCVTSD2USI, VCVTSS2USI, VCVTTPD2QQ, VCVTTPD2UDQ, VCVTTPD2UQQ, VCVTTPS2UDQ, VCVTTPS2QQ, VCVTTPS2UQQ, VCVTSS2USI, VCVTTSS2USI, VCVTUDQ2PD, VCVTUDQ2PS, VCVTUQQ2PD, VCVTUQQ2PS, VCVTUSI2SD, VCVTUSI2SS, VDBPSADBW, VEXPANDPD, VEXPANDPS, VEXP2PD, VEXP2PS, VEXTRACTF128/VEXTRACTF32x4/VEXTRACTF64x2/VEXTRACTF32x8/VEXTRACTF64x4, VFIXUPIMMPD, VFIXUPIMMPS, VFIXUPIMMSD, VFIXUPIMMSS, VFMADD132PD/VFMADD213PD/VFMADD231PD, VFMADD132PS/VFMADD213PS/VFMADD231PS, VFMADD132SD/VFMADD213SD/VFMADD231SD, VFMADD132SS/VFMADD213SS/VFMADD231SS, VFMADDSUB132PD/VFMADDSUB213PD/VFMADDSUB231PD, VFMADDSUB132PS/VFMADDSUB213PS/VFMADDSUB231PS, VFMSSUB132PD/VFMSSUB213PD/VFMSSUB231PD, VFMSSUB132PS/VFMSSUB213PS/VFMSSUB231PS, VFMSSUB132SD/VFMSSUB213SD/VFMSSUB231SD, VFMSSUB132SS/VFMSSUB213SS/VFMSSUB231SS, VFNMADD132PD/VFNMADD213PD/VFNMADD231PD, VFNMADD132PS/VFNMADD213PS/VFNMADD231PS, VFNMADD132SD/VFNMADD213SD/VFNMADD231SD, VFNMADD132SS/VFNMADD213SS/VFNMADD231SS, VFNMSSUB132PD/VFNMSSUB213PD/VFNMSSUB231PD, VFNMSSUB132PS/VFNMSSUB213PS/VFNMSSUB231PS, VFNMSSUB132SD/VFNMSSUB213SD/VFNMSSUB231SD, VFNMSSUB132SS/VFNMSSUB213SS/VFNMSSUB231SS, VFPCLASSPD, VFPCLASSPS, VFPCLASSSD, VFPCLASSSS, VGATHERDPS/VGATHERDPD, VGATHERPF0DPS/VGATHERPF0QPS/VGATHERPF0DPD/VGATHERPF0QPD, VGATHERPF1DPS/VGATHERPF1QPS/VGATHERPF1DPD/VGATHERPF1QPD, VGATHERQPS/VGATHERQPD, VPGATHERDD/VPGATHERDQ, VPGATHERQD/VPGATHERQQ, VGETEXPPD, VGETEXPPS, VGETEXPSD, VGETEXSS, VGETMANTPD, VGETMANTPS, VGETMANTSD, VGETMANTSS, VINSERTF128/VINSERTF32x4/VINSERTF64x2/VINSERTF32x8/VINSERTF64x4, VINSERTI128/VINSERTI32x4/VINSERTI64x2/VINSERTI32x8/VINSERTI64x4, VPBLENDMB/VPBLENDMW, VPBLENDMD/VPBLENDMQ, VPBROADCASTB/W/D/Q, VPBROADCAST, VPCMPB/VPCMPUB, VPCMPD/VPCMPUD, VPCMPQ/VPCMPUQ, VPCMPW/VPCMPUW, VPCOMPRESSD, VPCOMPRESSQ, VPCONFLICTD/Q, VPERMD/VPERMW, VPERMI2W/D/Q/PS/PD, VPERMILPD, VPERMILPS, VPERMPD, VPERMPS, VPERMQ, VEXPANDD, VEXPANDQ, VPLZCNTD/Q, VPMOVM2B/VPMOVM2W/VPMOVM2D/VPMOVM2Q, VPMOVB2M/VPMOVW2M/VPMOVD2M/VPMOVQ2M, VPMOVQB/VPMOVSQB/VPMOVUSQB, VPMOVQW/VPMOVQSW/VPMOVUSQW, VPMOVQD/VPMOVQSD/VPMOVUSQD, VPMOVDB/VPMOVSDB/VPMOVUSDB, VPMOVDW/VPMOVSDW/VPMOVUSDW, VPMOVWB/VPMOVSWB/VPMOVUSWB, PROLD/PROLVD/PROLQ/PROLVQ, PRORD/PRORVD/PRORQ/PRORVQ, VPSCATTERDD/VPSCATTERDQ/VPSCATTERQD/VPSCATTERQQ, VPSLLVW/VPSLLVD/VPSLLVQ, VPSRAVW/VPSRAVD/VPSRAVQ, VPSRLVW/VPSRLVD/VPSRLVQ, VPTERNLOGD/VPTERNLOGQ, VPTSTMB/VPTSTMW/VPTSTMD/VPTSTMQ, VPTSTNMB/W/D/Q, VRANGEPD, VRANGEPS, VRANGESD, VRANGESS, VRCP14PD, VRCP14SD, VRCP14PS, VRCP14SS, VRCP28PD, VRCP28SD, VRCP28PS, VRCP28SS, VREDUCEPD, VREDUCESD, VREDUCEPS, VREDUCESS, VRNDSCALEPD, VRNDSCALESD, VRNDSCALEPS, VRNDSCALESS, VRSQRT14PD, VRSQRT14SD, VRSQRT14PS, VRSQRT14SS, VRSQRT28PD, VRSQRT28SD, VRSQRT28PS, VRSQRT28SS, VSCALEFPD, VSCALEFSD, VSCALEFPS, VSCALEFSS, VSCATTERDPS/VSCATTERDPD/VSCATTERQPS/VSCATTERQPD, VSCATTERPF0DPS/VSCATTERPF0QPS/VSCATTERPF0DPD/VSCATTERPF0QPD, VSCATTERPF1DPS/VSCATTERPF1QPS/VSCATTERPF1DPD/VSCATTERPF1QPD, VSHUFF32x4/VSHUFF64x2/VSHUFFI32x4/VSHUFFI64x2, XADD, XORPD, XORPS.

## 5.1 TERNARY BIT VECTOR LOGIC TABLE

VPTERNLOGD/VPTERNLOGQ instructions operate on dword/qword elements and take three bit vectors of the respective input data elements to form a set of 32/64 indices, where each 3-bit value provides an index into an 8-bit lookup table represented by the imm8 byte of the instruction. The 256 possible values of the imm8 byte is constructed as a 16x16 boolean logic table. The 16 rows of the table uses the lower 4 bits of imm8 as row index. The 16 columns are referenced by imm8[7:4]. The 16 columns of the table are present in two halves, with 8 columns shown in Table 5-1 for the column index value between 0:7, followed by Table 5-2 showing the 8 columns corresponding to column index 8:15. This section presents the two-halves of the 256-entry table using a shorthand notation representing simple or compound boolean logic expressions with three input bit source data.

The three input bit source data will be denoted with the capital letters: A, B, C; where A represents a bit from the first source operand (also the destination operand), B and C represent a bit from the 2nd and 3rd source operands.

Each map entry takes the form of a logic expression consisting of one or more component expressions. Each component expression consists of either a unary or binary boolean operator and associated operands. Each binary boolean operator is expressed in lowercase letters, and operands concatenated after the logic operator. The unary operator 'not' is expressed using '!'. Additionally, the conditional expression "A?B:C" expresses a result returning B if A is set, returning C otherwise.

A binary boolean operator is followed by two operands, e.g. andAB. For a compound binary expression that contain commutative components and comprising the same logic operator, the 2nd logic operator is omitted and three operands can be concatenated in sequence, e.g. andABC. When the 2nd operand of the first binary boolean expression comes from the result of another boolean expression, the 2nd boolean expression is concatenated after the uppercase operand of the first logic expression, e.g. norBnandAC. When the result is independent of an operand, that operand is omitted in the logic expression, e.g. zeros or norCB.

The 3-input expression "majorABC" returns 0 if two or more input bits are 0, returns 1 if two or more input bits are 1. The 3-input expression "minorABC" returns 1 if two or more input bits are 0, returns 0 if two or more input bits are 1.

The building-block bit logic functions used in Table 5-1 and Table 5-2 include;

- Constants: TRUE (1), FALSE (0);
- Unary function: Not (!);
- Binary functions: and, nand, or, nor, xor, xnor;
- Conditional function: Select (?:);
- Tertiary functions: major, minor.

Table 5-1. Low 8 columns of the 16x16 Map of VPTERNLOG Boolean Logic Operations

Imm	[7:4]							
[3:0]	0H	1H	2H	3H	4H	5H	6H	7H
00H	FALSE	andAnorBC	norBnandAC	andA!B	norCnandBA	andA!C	andAxorBC	andAnandBC
01H	norABC	norCB	norBxorAC	A?!B:norBC	norCxorBA	A?!C:norBC	A?xorBC:norBC	A?nandBC:norBC
02H	andCnorBA	norBxnorAC	andC!B	norBnorAC	C?norBA:andBA	C?norBA:A	C?!B:andBA	C?!B:A
03H	norBA	norBandAC	C?!B:norBA	!B	C?norBA:xnorBA	A?!C:!B	A?xorBC:!B	A?nandBC:!B
04H	andBnorAC	norCxnorBA	B?norAC:andAC	B?norAC:A	andB!C	norCnorBA	B?!C:andAC	B?!C:A
05H	norCA	norCandBA	B?norAC:xnorAC	A?!B:!C	B?!C:norAC	!C	A?xorBC:!C	A?nandBC:!C
06H	norAxnorBC	A?norBC:xorBC	B?norAC:C	xorBorAC	C?norBA:B	xorCorBA	xorCB	B?!C:orAC
07H	norAandBC	minorABC	C?!B:!A	nandBorAC	B?!C:!A	nandCorBA	A?xorBC:nandBC	nandCB
08H	norAnandBC	A?norBC:andBC	andCxorBA	A?!B:andBC	andBxorAC	A?!C:andBC	A?xorBC:andBC	xorAandBC
09H	norAxorBC	A?norBC:xnorBC	C?xorBA:norBA	A?!B:xnorBC	B?xorAC:norAC	A?!C:xnorBC	xnorABC	A?nandBC:xnorBC
0AH	andCIA	A?norBC:C	andCnandBA	A?!B:C	C?!A:andBA	xorCA	xorCandBA	A?nandBC:C
0BH	C?!A:norBA	C?!A:!B	C?nandBA:norBA	C?nandBA:!B	B?xorAC:!A	B?xorAC:nandAC	C?nandBA:xnorBA	nandBxnorAC
0CH	andB!A	A?norBC:B	B?!A:andAC	xorBA	andBnandAC	A?!C:B	xorBandAC	A?nandBC:B
0DH	B?!A:norAC	B?!A:!C	B?!A:xnorAC	C?xorBA:nandBA	B?nandAC:norAC	B?nandAC:!C	B?nandAC:xnorAC	nandCxnorBA
0EH	norAnorBC	xorAorBC	B?!A:C	A?!B:orBC	C?!A:B	A?!C:orBC	B?nandAC:C	A?nandBC:orBC
0FH	!A	nandAorBC	C?nandBA:!A	nandBA	B?nandAC:!A	nandCA	nandAxnorBC	nandABC

Table 5-2 shows the half of 256-entry map corresponding to column index values 8:15.

Table 5-2. Low 8 columns of the 16x16 Map of VPTERNLOG Boolean Logic Operations

Imm	[7:4]							
[3:0]	08H	09H	0AH	0BH	0CH	0DH	0EH	0FH
00H	<i>andABC</i>	<i>andAxnorBC</i>	<i>andCA</i>	<i>B?andAC:A</i>	<i>andBA</i>	<i>C?andBA:A</i>	<i>andAorBC</i>	<i>A</i>
01H	<i>A?andBC:norBC</i>	<i>B?andAC:!C</i>	<i>A?C:norBC</i>	<i>C?A:!B</i>	<i>A?B:norBC</i>	<i>B?A:!C</i>	<i>xnorAorBC</i>	<i>orAnorBC</i>
02H	<i>andCxnorBA</i>	<i>B?andAC:xorAC</i>	<i>B?andAC:C</i>	<i>B?andAC:orAC</i>	<i>C?xnorBA:andBA</i>	<i>B?A:xorAC</i>	<i>B?A:C</i>	<i>B?A:orAC</i>
03H	<i>A?andBC:!B</i>	<i>xnorBandAC</i>	<i>A?C:!B</i>	<i>nandBnandAC</i>	<i>xnorBA</i>	<i>B?A:nandAC</i>	<i>A?orBC:!B</i>	<i>orA!B</i>
04H	<i>andBxnorAC</i>	<i>C?andBA:xorBA</i>	<i>B?xnorAC:andAC</i>	<i>B?xnorAC:A</i>	<i>C?andBA:B</i>	<i>C?andBA:orBA</i>	<i>C?A:B</i>	<i>C?A:orBA</i>
05H	<i>A?andBC:!C</i>	<i>xnorCandBA</i>	<i>xnorCA</i>	<i>C?A:nandBA</i>	<i>A?B:!C</i>	<i>nandCnandBA</i>	<i>A?orBC:!C</i>	<i>orA!C</i>
06H	<i>A?andBC:xorBC</i>	<i>xorABC</i>	<i>A?C:xorBC</i>	<i>B?xnorAC:orAC</i>	<i>A?B:xorBC</i>	<i>C?xnorBA:orBA</i>	<i>A?orBC:xorBC</i>	<i>orAxorBC</i>
07H	<i>xnorAandBC</i>	<i>A?xnorBC:nandBC</i>	<i>A?C:nandBC</i>	<i>nandBxorAC</i>	<i>A?B:nandBC</i>	<i>nandCxorBA</i>	<i>A?orBCnandBC</i>	<i>orAnandBC</i>
08H	<i>andCB</i>	<i>A?xnorBC:andBC</i>	<i>andCorAB</i>	<i>B?C:A</i>	<i>andBorAC</i>	<i>C?B:A</i>	<i>majorABC</i>	<i>orAandBC</i>
09H	<i>B?C:norAC</i>	<i>xnorCB</i>	<i>xnorCorBA</i>	<i>C?orBA:!B</i>	<i>xnorBorAC</i>	<i>B?orAC:!C</i>	<i>A?orBC:xnorBC</i>	<i>orAxnorBC</i>
0AH	<i>A?andBC:C</i>	<i>A?xnorBC:C</i>	<i>C</i>	<i>B?C:orAC</i>	<i>A?B:C</i>	<i>B?orAC:xorAC</i>	<i>orCandBA</i>	<i>orCA</i>
0BH	<i>B?C:!A</i>	<i>B?C:nandAC</i>	<i>orCnorBA</i>	<i>orC!B</i>	<i>B?orAC:!A</i>	<i>B?orAC:nandAC</i>	<i>orCxnorBA</i>	<i>nandBnorAC</i>
0CH	<i>A?andBC:B</i>	<i>A?xnorBC:B</i>	<i>A?C:B</i>	<i>C?orBA:xorBA</i>	<i>B</i>	<i>C?B:orBA</i>	<i>orBandAC</i>	<i>orBA</i>
0DH	<i>C?B!A</i>	<i>C?B:nandBA</i>	<i>C?orBA:!A</i>	<i>C?orBA:nandBA</i>	<i>orBnorAC</i>	<i>orB!C</i>	<i>orBxnorAC</i>	<i>nandCnorBA</i>
0EH	<i>A?andBC:orBC</i>	<i>A?xnorBC:orBC</i>	<i>A?C:orBC</i>	<i>orCxorBA</i>	<i>A?B:orBC</i>	<i>orBxorAC</i>	<i>orCB</i>	<i>orABC</i>
0FH	<i>nandAnandBC</i>	<i>nandAxorBC</i>	<i>orC!A</i>	<i>orCnandBA</i>	<i>orB!A</i>	<i>orBnandAC</i>	<i>nandAnorBC</i>	<i>TRUE</i>

Table 5-1 and Table 5-2 translate each of the possible value of the imm8 byte to a Boolean expression. These tables can also be used by software to translate Boolean expressions to numerical constants to form the imm8 value needed to construct the VPTERNLOG syntax. There is a unique set of three byte constants (F0H, CCH, AAH) that can be used for this purpose as input operands in conjunction with the Boolean expressions defined in those tables. The reverse mapping can be expressed as:

Result\_imm8 = Table\_Lookup\_Entry( 0F0H, 0CCH, 0AAH)

Table\_Lookup\_Entry is the Boolean expression defined in Table 5-1 and Table 5-2.

## 5.2 INSTRUCTIONS (V-Z)

Chapter 5 continues an alphabetical discussion of Intel® 64 and IA-32 instructions (V-Z). See also: Chapter 3, “Instruction Set Reference, A-L,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A*, and Chapter 4, “Instruction Set Reference, M-U,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2B*.



## VALIGND/VALIGNQ—Align Doubleword/Quadword Vectors

Opcode/ Instruction	Op / En	64/32 bitMode Support	CPUID Feature Flag	Description
EVEX.NDS.128.66.0F3A.W0 03 /r ib VALIGND xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst, imm8	FV	V/V	AVX512VL AVX512F	Shift right and merge vectors xmm2 and xmm3/m128/m32bcst with double-word granularity using imm8 as number of elements to shift, and store the final result in xmm1, under writemask.
EVEX.NDS.128.66.0F3A.W1 03 /r ib VALIGNQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst, imm8	FV	V/V	AVX512VL AVX512F	Shift right and merge vectors xmm2 and xmm3/m128/m64bcst with quad-word granularity using imm8 as number of elements to shift, and store the final result in xmm1, under writemask.
EVEX.NDS.256.66.0F3A.W0 03 /r ib VALIGND ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst, imm8	FV	V/V	AVX512VL AVX512F	Shift right and merge vectors ymm2 and ymm3/m256/m32bcst with double-word granularity using imm8 as number of elements to shift, and store the final result in ymm1, under writemask.
EVEX.NDS.256.66.0F3A.W1 03 /r ib VALIGNQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst, imm8	FV	V/V	AVX512VL AVX512F	Shift right and merge vectors ymm2 and ymm3/m256/m64bcst with quad-word granularity using imm8 as number of elements to shift, and store the final result in ymm1, under writemask.
EVEX.NDS.512.66.0F3A.W0 03 /r ib VALIGND zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst, imm8	FV	V/V	AVX512F	Shift right and merge vectors zmm2 and zmm3/m512/m32bcst with double-word granularity using imm8 as number of elements to shift, and store the final result in zmm1, under writemask.
EVEX.NDS.512.66.0F3A.W1 03 /r ib VALIGNQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst, imm8	FV	V/V	AVX512F	Shift right and merge vectors zmm2 and zmm3/m512/m64bcst with quad-word granularity using imm8 as number of elements to shift, and store the final result in zmm1, under writemask.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
FV	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

### Description

Concatenates and shifts right doubleword/quadword elements of the first source operand (the second operand) and the second source operand (the third operand) into a 1024/512/256-bit intermediate vector. The low 512/256/128-bit of the intermediate vector is written to the destination operand (the first operand) using the writemask k1. The destination and first source operands are ZMM/YMM/XMM registers. 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 32/64-bit memory location.

This instruction is writemasked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values (merging-masking) or are set to 0 (zeroing-masking).

### Operation

#### VALIGND (EVEX encoded versions)

(KL, VL) = (4, 128), (8, 256), (16, 512)

IF (SRC2 \*is memory\*) (AND EVEX.b = 1)

THEN

FOR j ← 0 TO KL-1

i ← j \* 32

```

        src[j+31:i] ← SRC2[31:0]
    ENDFOR;
    ELSE src ← SRC2
FI
; Concatenate sources
tmp[VL-1:0] ← src[VL-1:0]
tmp[2VL-1:VL] ← SRC1[VL-1:0]
; Shift right doubleword elements
IF VL = 128
    THEN SHIFT = imm8[1:0]
    ELSE
        IF VL = 256
            THEN SHIFT = imm8[2:0]
            ELSE SHIFT = imm8[3:0]
        FI
FI;
tmp[2VL-1:0] ← tmp[2VL-1:0] >> (32*SHIFT)
; Apply writemask
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[j+31:i] ← tmp[j+31:i]
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[j+31:i] remains unchanged*
        ELSE                                  ; zeroing-masking
            DEST[j+31:i] ← 0
        FI
    FI;
ENDFOR;
DEST[MAX_VL-1:VL] ← 0

```

**VALIGNQ (EVEX encoded versions)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

IF (SRC2 \*is memory\*) (AND EVEX.b = 1)

```

    THEN
        FOR j ← 0 TO KL-1
            i ← j * 64
            src[j+63:i] ← SRC2[63:0]
        ENDFOR;
    ELSE src ← SRC2
FI

```

```

; Concatenate sources
tmp[VL-1:0] ← src[VL-1:0]
tmp[2VL-1:VL] ← SRC1[VL-1:0]
; Shift right quadword elements
IF VL = 128
    THEN SHIFT = imm8[0]
    ELSE
        IF VL = 256
            THEN SHIFT = imm8[1:0]
            ELSE SHIFT = imm8[2:0]
        FI
FI;

```

FI;

```

tmp[2VL-1:0] ← tmp[2VL-1:0] >> (64*SHIFT)
; Apply writemask
FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] ← tmp[i+63:i]
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+63:i] remains unchanged*
    ELSE ; zeroing-masking
      DEST[i+63:i] ← 0
    FI
  FI;
ENDFOR;
DEST[MAX_VL-1:VL] ← 0

```

### Intel C/C++ Compiler Intrinsic Equivalent

```

VALIGND __m512i __mm512_alignr_epi32( __m512i a, __m512i b, int cnt);
VALIGND __m512i __mm512_mask_alignr_epi32( __m512i s, __mmask16 k, __m512i a, __m512i b, int cnt);
VALIGND __m512i __mm512_maskz_alignr_epi32( __mmask16 k, __m512i a, __m512i b, int cnt);
VALIGND __m256i __mm256_mask_alignr_epi32( __m256i s, __mmask8 k, __m256i a, __m256i b, int cnt);
VALIGND __m256i __mm256_maskz_alignr_epi32( __mmask8 k, __m256i a, __m256i b, int cnt);
VALIGND __m128i __mm_mask_alignr_epi32( __m128i s, __mmask8 k, __m128i a, __m128i b, int cnt);
VALIGND __m128i __mm_maskz_alignr_epi32( __mmask8 k, __m128i a, __m128i b, int cnt);
VALIGNQ __m512i __mm512_alignr_epi64( __m512i a, __m512i b, int cnt);
VALIGNQ __m512i __mm512_mask_alignr_epi64( __m512i s, __mmask8 k, __m512i a, __m512i b, int cnt);
VALIGNQ __m512i __mm512_maskz_alignr_epi64( __mmask8 k, __m512i a, __m512i b, int cnt);
VALIGNQ __m256i __mm256_mask_alignr_epi64( __m256i s, __mmask8 k, __m256i a, __m256i b, int cnt);
VALIGNQ __m256i __mm256_maskz_alignr_epi64( __mmask8 k, __m256i a, __m256i b, int cnt);
VALIGNQ __m128i __mm_mask_alignr_epi64( __m128i s, __mmask8 k, __m128i a, __m128i b, int cnt);
VALIGNQ __m128i __mm_maskz_alignr_epi64( __mmask8 k, __m128i a, __m128i b, int cnt);

```

### Exceptions

See Exceptions Type E4NF.

## VBLENDMPD/VBLENDMPS—Blend Float64/Float32 Vectors Using an OpMask Control

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.128.66.0F38.W1 65 /r VBLENDMPD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	FV	V/V	AVX512VL AVX512F	Blend double-precision vector xmm2 and double-precision vector xmm3/m128/m64bcst and store the result in xmm1, under control mask.
EVEX.NDS.256.66.0F38.W1 65 /r VBLENDMPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	FV	V/V	AVX512VL AVX512F	Blend double-precision vector ymm2 and double-precision vector ymm3/m256/m64bcst and store the result in ymm1, under control mask.
EVEX.NDS.512.66.0F38.W1 65 /r VBLENDMPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	FV	V/V	AVX512F	Blend double-precision vector zmm2 and double-precision vector zmm3/m512/m64bcst and store the result in zmm1, under control mask.
EVEX.NDS.128.66.0F38.W0 65 /r VBLENDMPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	FV	V/V	AVX512VL AVX512F	Blend single-precision vector xmm2 and single-precision vector xmm3/m128/m32bcst and store the result in xmm1, under control mask.
EVEX.NDS.256.66.0F38.W0 65 /r VBLENDMPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	FV	V/V	AVX512VL AVX512F	Blend single-precision vector ymm2 and single-precision vector ymm3/m256/m32bcst and store the result in ymm1, under control mask.
EVEX.NDS.512.66.0F38.W0 65 /r VBLENDMPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	FV	V/V	AVX512F	Blend single-precision vector zmm2 and single-precision vector zmm3/m512/m32bcst using k1 as select control and store the result in zmm1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
FV	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

### Description

Performs an element-by-element blending between float64/float32 elements in the first source operand (the second operand) with the elements in the second source operand (the third operand) using an opmask register as select control. The blended result is written to the destination register.

The destination and first source operands are ZMM/YMM/XMM registers. 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 64-bit memory location.

The opmask register is not used as a writemask for this instruction. Instead, the mask is used as an element selector: every element of the destination is conditionally selected between first source or second source using the value of the related mask bit (0 for first source operand, 1 for second source operand).

If EVEX.z is set, the elements with corresponding mask bit value of 0 in the destination operand are zeroed.

### Operation

#### VBLENDMPD (EVEX encoded versions)

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

  i ← j \* 64

  IF k1[j] OR \*no controlmask\*

    THEN

      IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

        THEN

```

        DEST[i+63:i] ← SRC2[63:0]
    ELSE
        DEST[i+63:i] ← SRC2[i+63:i]
    FI;
ELSE
    IF *merging-masking*           ; merging-masking
        THEN DEST[i+63:i] ← SRC1[i+63:i]
    ELSE                           ; zeroing-masking
        DEST[i+63:i] ← 0
    FI;
FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0
VBLENDMPS (EVEX encoded versions)
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no controlmask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN
                    DEST[i+31:i] ← SRC2[31:0]
                ELSE
                    DEST[i+31:i] ← SRC2[i+31:i]
            FI;
        ELSE
            IF *merging-masking*           ; merging-masking
                THEN DEST[i+31:i] ← SRC1[i+31:i]
            ELSE                           ; zeroing-masking
                DEST[i+31:i] ← 0
            FI;
        FI;
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

#### Intel C/C++ Compiler Intrinsic Equivalent

```

VBLENDMPD __m512d __mm512_mask_blend_pd(__mmask8 k, __m512d a, __m512d b);
VBLENDMPD __m256d __mm256_mask_blend_pd(__mmask8 k, __m256d a, __m256d b);
VBLENDMPD __m128d __mm_mask_blend_pd(__mmask8 k, __m128d a, __m128d b);
VBLENDMPS __m512 __mm512_mask_blend_ps(__mmask16 k, __m512 a, __m512 b);
VBLENDMPS __m256 __mm256_mask_blend_ps(__mmask8 k, __m256 a, __m256 b);
VBLENDMPS __m128 __mm_mask_blend_ps(__mmask8 k, __m128 a, __m128 b);

```

#### SIMD Floating-Point Exceptions

None

#### Other Exceptions

See Exceptions Type E4.

## VBROADCAST—Load with Broadcast Floating-Point Data

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F38.W0 18 /r VBROADCASTSS xmm1, m32	RM	V/V	AVX	Broadcast single-precision floating-point element in mem to four locations in xmm1.
VEX.256.66.0F38.W0 18 /r VBROADCASTSS ymm1, m32	RM	V/V	AVX	Broadcast single-precision floating-point element in mem to eight locations in ymm1.
VEX.256.66.0F38.W0 19 /r VBROADCASTSD ymm1, m64	RM	V/V	AVX	Broadcast double-precision floating-point element in mem to four locations in ymm1.
VEX.256.66.0F38.W0 1A /r VBROADCASTF128 ymm1, m128	RM	V/V	AVX	Broadcast 128 bits of floating-point data in mem to low and high 128-bits in ymm1.
EVEX.256.66.0F38.W1 19 /r VBROADCASTSD xmm1 {k1}{z}, xmm2/m64	T1S	V/V	AVX512VL AVX512F	Broadcast low double-precision floating-point element in xmm2/m64 to four locations in ymm1 using writemask k1.
EVEX.512.66.0F38.W1 19 /r VBROADCASTSD zmm1 {k1}{z}, xmm2/m64	T1S	V/V	AVX512F	Broadcast low double-precision floating-point element in xmm2/m64 to eight locations in zmm1 using writemask k1.
EVEX.256.66.0F38.W0 19 /r VBROADCASTF32X2 ymm1 {k1}{z}, xmm2/m64	T2	V/V	AVX512VL AVX512DQ	Broadcast two single-precision floating-point elements in xmm2/m64 to locations in ymm1 using writemask k1.
EVEX.512.66.0F38.W0 19 /r VBROADCASTF32X2 zmm1 {k1}{z}, xmm2/m64	T2	V/V	AVX512DQ	Broadcast two single-precision floating-point elements in xmm2/m64 to locations in zmm1 using writemask k1.
EVEX.128.66.0F38.W0 18 /r VBROADCASTSS xmm1 {k1}{z}, xmm2/m32	T1S	V/V	AVX512VL AVX512F	Broadcast low single-precision floating-point element in xmm2/m32 to all locations in xmm1 using writemask k1.
EVEX.256.66.0F38.W0 18 /r VBROADCASTSS ymm1 {k1}{z}, xmm2/m32	T1S	V/V	AVX512VL AVX512F	Broadcast low single-precision floating-point element in xmm2/m32 to all locations in ymm1 using writemask k1.
EVEX.512.66.0F38.W0 18 /r VBROADCASTSS zmm1 {k1}{z}, xmm2/m32	T1S	V/V	AVX512F	Broadcast low single-precision floating-point element in xmm2/m32 to all locations in zmm1 using writemask k1.
EVEX.256.66.0F38.W0 1A /r VBROADCASTF32X4 ymm1 {k1}{z}, m128	T4	V/V	AVX512VL AVX512F	Broadcast 128 bits of 4 single-precision floating-point data in mem to locations in ymm1 using writemask k1.
EVEX.512.66.0F38.W0 1A /r VBROADCASTF32X4 zmm1 {k1}{z}, m128	T4	V/V	AVX512F	Broadcast 128 bits of 4 single-precision floating-point data in mem to locations in zmm1 using writemask k1.

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.256.66.0F38.W1 1A /r VBROADCASTF64X2 ymm1 {k1}{z}, m128	T2	V/V	AVX512VL AVX512DQ	Broadcast 128 bits of 2 double-precision floating-point data in mem to locations in ymm1 using writemask k1.
EVEX.512.66.0F38.W1 1A /r VBROADCASTF64X2 zmm1 {k1}{z}, m128	T2	V/V	AVX512DQ	Broadcast 128 bits of 2 double-precision floating-point data in mem to locations in zmm1 using writemask k1.
EVEX.512.66.0F38.W0 1B /r VBROADCASTF32X8 zmm1 {k1}{z}, m256	T8	V/V	AVX512DQ	Broadcast 256 bits of 8 single-precision floating-point data in mem to locations in zmm1 using writemask k1.
EVEX.512.66.0F38.W1 1B /r VBROADCASTF64X4 zmm1 {k1}{z}, m256	T4	V/V	AVX512F	Broadcast 256 bits of 4 double-precision floating-point data in mem to locations in zmm1 using writemask k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
T1S, T2, T4, T8	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

#### Description

VBROADCASTSD/VBROADCASTSS/VBROADCASTF128 load floating-point values as one tuple from the source operand (second operand) in memory and broadcast to all elements of the destination operand (first operand).

VEX256-encoded versions: The destination operand is a YMM register. The source operand is either a 32-bit, 64-bit, or 128-bit memory location. Register source encodings are reserved and will #UD. Bits (MAX\_VL-1:256) of the destination register are zeroed.

EVEX-encoded versions: The destination operand is a ZMM/YMM/XMM register and updated according to the writemask k1. The source operand is either a 32-bit, 64-bit memory location or the low doubleword/quadword element of an XMM register.

VBROADCASTF32X2/VBROADCASTF32X4/VBROADCASTF64X2/VBROADCASTF32X8/VBROADCASTF64X4 load floating-point values as tuples from the source operand (the second operand) in memory or register and broadcast to all elements of the destination operand (the first operand). The destination operand is a YMM/ZMM register updated according to the writemask k1. The source operand is either a register or 64-bit/128-bit/256-bit memory location.

VBROADCASTSD and VBROADCASTF128,F32x4 and F64x2 are only supported as 256-bit and 512-bit wide versions and up. VBROADCASTSS is supported in 128-bit, 256-bit and 512-bit wide versions. F32x8 and F64x4 are only supported as 512-bit wide versions.

VBROADCASTF32X2/VBROADCASTF32X4/VBROADCASTF32X8 have 32-bit granularity. VBROADCASTF64X2 and VBROADCASTF64X4 have 64-bit granularity.

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

If VBROADCASTSD or VBROADCASTF128 is encoded with VEX.L= 0, an attempt to execute the instruction encoded with VEX.L= 0 will cause an #UD exception.

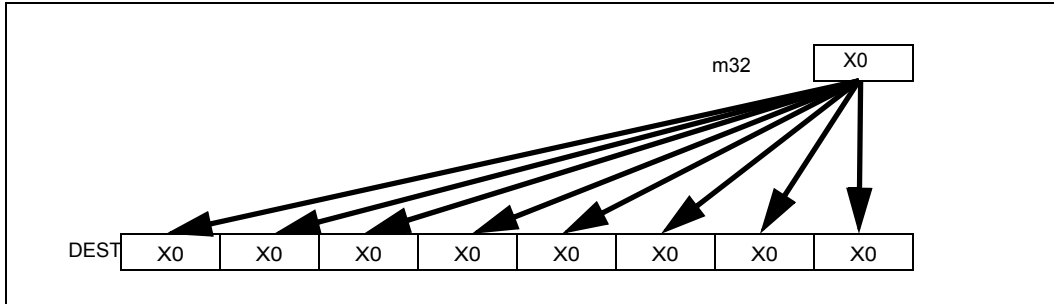


Figure 5-1. VBROADCASTSS Operation (VEX.256 encoded version)

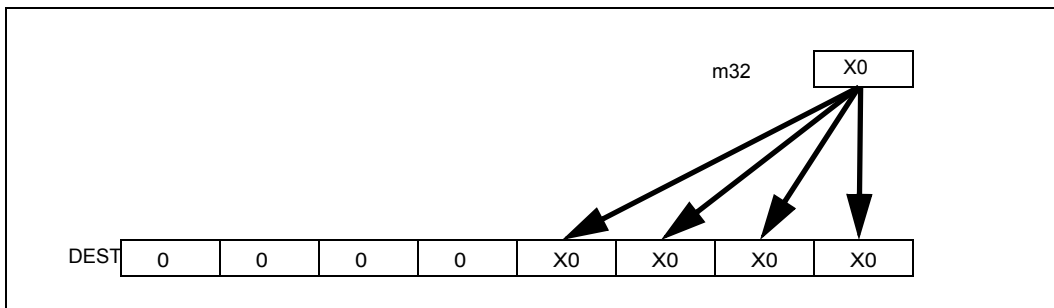


Figure 5-2. VBROADCASTSS Operation (VEX.128-bit version)

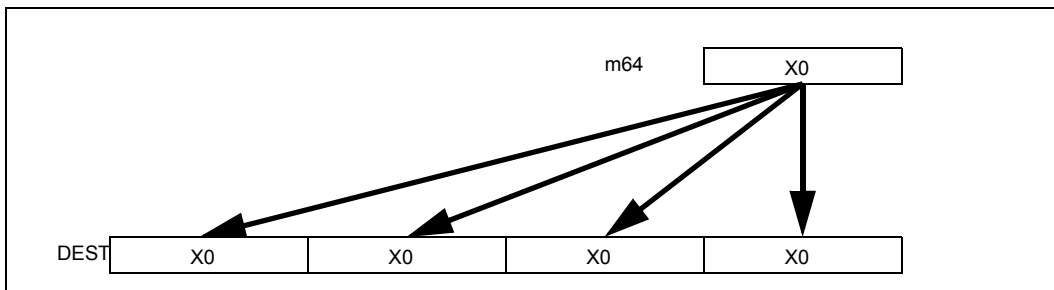


Figure 5-3. VBROADCASTSD Operation (VEX.256-bit version)

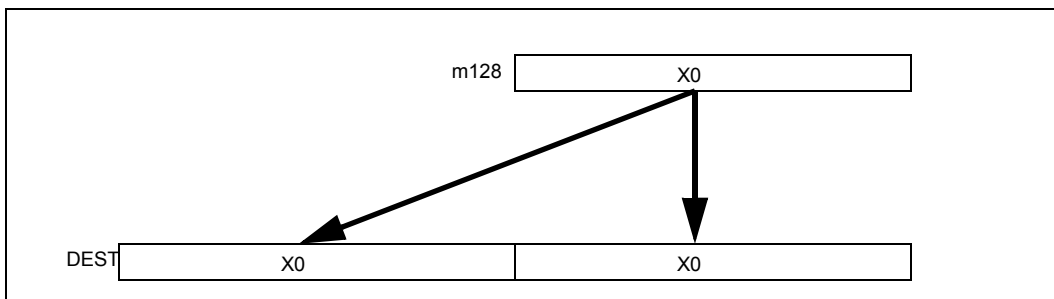


Figure 5-4. VBROADCASTF128 Operation (VEX.256-bit version)



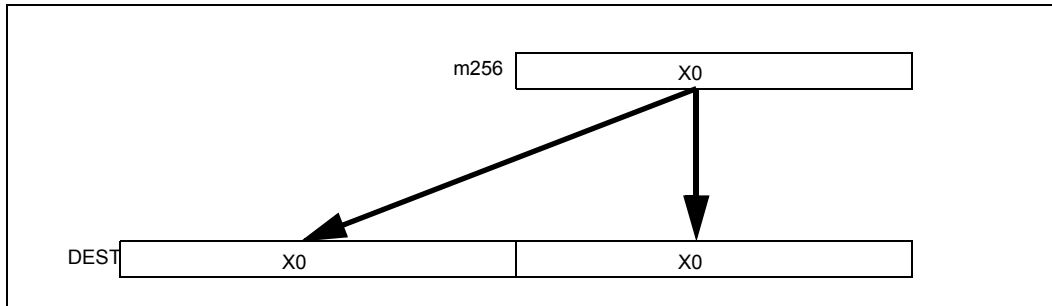


Figure 5-5. VBROADCASTF64X4 Operation (512-bit version with writemask all 1s)

### Operation

#### VBROADCASTSS (128 bit version VEX and legacy)

```
temp ← SRC[31:0]
DEST[31:0] ← temp
DEST[63:32] ← temp
DEST[95:64] ← temp
DEST[127:96] ← temp
DEST[MAX_VL-1:128] ← 0
```

#### VBROADCASTSS (VEX.256 encoded version)

```
temp ← SRC[31:0]
DEST[31:0] ← temp
DEST[63:32] ← temp
DEST[95:64] ← temp
DEST[127:96] ← temp
DEST[159:128] ← temp
DEST[191:160] ← temp
DEST[223:192] ← temp
DEST[255:224] ← temp
DEST[MAX_VL-1:256] ← 0
```

#### VBROADCASTSS (EVEX encoded versions)

(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[31:0]
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+31:i] remains unchanged*
    ELSE ; zeroing-masking
      DEST[i+31:i] ← 0
    FI
  FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0
```

#### VBROADCASTSD (VEX.256 encoded version)

```
temp ← SRC[63:0]
DEST[63:0] ← temp
```

```

DEST[127:64] ← temp
DEST[191:128] ← temp
DEST[255:192] ← temp
DEST[MAX_VL-1:256] ← 0

```

**VBROADCASTSD (EVEX encoded versions)**

(KL, VL) = (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[63:0]
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+63:i] remains unchanged*
    ELSE ; zeroing-masking
      DEST[i+63:i] ← 0
    FI
  FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VBROADCASTF32x2 (EVEX encoded versions)**

(KL, VL) = (8, 256), (16, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 32
  n ← (j mod 2) * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ← SRC[n+31:n]
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+31:i] remains unchanged*
    ELSE ; zeroing-masking
      DEST[i+31:i] ← 0
    FI
  FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VBROADCASTF128 (VEX.256 encoded version)**

```

temp ← SRC[127:0]
DEST[127:0] ← temp
DEST[255:128] ← temp
DEST[MAX_VL-1:256] ← 0

```

**VBROADCASTF32X4 (EVEX encoded versions)**

(KL, VL) = (8, 256), (16, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 32
  n ← (j modulo 4) * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ← SRC[n+31:n]
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+31:i] remains unchanged*

```

```

        ELSE                                ; zeroing-masking
            DEST[i+31:i] ← 0
        FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VBROADCASTF64X2 (EVEX encoded versions)**

(KL, VL) = (4, 256), (8, 512)

```

FOR j ← 0 TO KL-1
    i ← j * 64
    n ← (j modulo 2) * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← SRC[n+63:n]
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[i+63:i] remains unchanged*
        ELSE                                  ; zeroing-masking
            DEST[i+63:i] = 0
        FI
    FI;
ENDFOR;

```

**VBROADCASTF32X8 (EVEX.U1.512 encoded version)**

```

FOR j ← 0 TO 15
    i ← j * 32
    n ← (j modulo 8) * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← SRC[n+31:n]
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[i+31:i] remains unchanged*
        ELSE                                  ; zeroing-masking
            DEST[i+31:i] ← 0
        FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VBROADCASTF64X4 (EVEX.512 encoded version)**

```

FOR j ← 0 TO 7
    i ← j * 64
    n ← (j modulo 4) * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← SRC[n+63:n]
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[i+63:i] remains unchanged*
        ELSE                                  ; zeroing-masking
            DEST[i+63:i] ← 0
        FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

VBROADCASTF32x2 \_\_m512 \_\_mm512\_broadcast\_f32x2( \_\_m128 a);  
 VBROADCASTF32x2 \_\_m512 \_\_mm512\_mask\_broadcast\_f32x2(\_\_m512 s, \_\_mmask16 k, \_\_m128 a);  
 VBROADCASTF32x2 \_\_m512 \_\_mm512\_maskz\_broadcast\_f32x2( \_\_mmask16 k, \_\_m128 a);  
 VBROADCASTF32x2 \_\_m256 \_\_mm256\_broadcast\_f32x2( \_\_m128 a);  
 VBROADCASTF32x2 \_\_m256 \_\_mm256\_mask\_broadcast\_f32x2(\_\_m256 s, \_\_mmask8 k, \_\_m128 a);  
 VBROADCASTF32x2 \_\_m256 \_\_mm256\_maskz\_broadcast\_f32x2( \_\_mmask8 k, \_\_m128 a);  
 VBROADCASTF32x4 \_\_m512 \_\_mm512\_broadcast\_f32x4( \_\_m128 a);  
 VBROADCASTF32x4 \_\_m512 \_\_mm512\_mask\_broadcast\_f32x4(\_\_m512 s, \_\_mmask16 k, \_\_m128 a);  
 VBROADCASTF32x4 \_\_m512 \_\_mm512\_maskz\_broadcast\_f32x4( \_\_mmask16 k, \_\_m128 a);  
 VBROADCASTF32x4 \_\_m256 \_\_mm256\_broadcast\_f32x4( \_\_m128 a);  
 VBROADCASTF32x4 \_\_m256 \_\_mm256\_mask\_broadcast\_f32x4(\_\_m256 s, \_\_mmask8 k, \_\_m128 a);  
 VBROADCASTF32x4 \_\_m256 \_\_mm256\_maskz\_broadcast\_f32x4( \_\_mmask8 k, \_\_m128 a);  
 VBROADCASTF32x8 \_\_m512 \_\_mm512\_broadcast\_f32x8( \_\_m256 a);  
 VBROADCASTF32x8 \_\_m512 \_\_mm512\_mask\_broadcast\_f32x8(\_\_m512 s, \_\_mmask16 k, \_\_m256 a);  
 VBROADCASTF32x8 \_\_m512 \_\_mm512\_maskz\_broadcast\_f32x8( \_\_mmask16 k, \_\_m256 a);  
 VBROADCASTF64x2 \_\_m512d \_\_mm512\_broadcast\_f64x2( \_\_m128d a);  
 VBROADCASTF64x2 \_\_m512d \_\_mm512\_mask\_broadcast\_f64x2(\_\_m512d s, \_\_mmask8 k, \_\_m128d a);  
 VBROADCASTF64x2 \_\_m512d \_\_mm512\_maskz\_broadcast\_f64x2( \_\_mmask8 k, \_\_m128d a);  
 VBROADCASTF64x2 \_\_m256d \_\_mm256\_broadcast\_f64x2( \_\_m128d a);  
 VBROADCASTF64x2 \_\_m256d \_\_mm256\_mask\_broadcast\_f64x2(\_\_m256d s, \_\_mmask8 k, \_\_m128d a);  
 VBROADCASTF64x2 \_\_m256d \_\_mm256\_maskz\_broadcast\_f64x2( \_\_mmask8 k, \_\_m128d a);  
 VBROADCASTF64x4 \_\_m512d \_\_mm512\_broadcast\_f64x4( \_\_m256d a);  
 VBROADCASTF64x4 \_\_m512d \_\_mm512\_mask\_broadcast\_f64x4(\_\_m512d s, \_\_mmask8 k, \_\_m256d a);  
 VBROADCASTF64x4 \_\_m512d \_\_mm512\_maskz\_broadcast\_f64x4( \_\_mmask8 k, \_\_m256d a);  
 VBROADCASTSD \_\_m512d \_\_mm512\_broadcastsd\_pd( \_\_m128d a);  
 VBROADCASTSD \_\_m512d \_\_mm512\_mask\_broadcastsd\_pd(\_\_m512d s, \_\_mmask8 k, \_\_m128d a);  
 VBROADCASTSD \_\_m512d \_\_mm512\_maskz\_broadcastsd\_pd(\_\_mmask8 k, \_\_m128d a);  
 VBROADCASTSD \_\_m256d \_\_mm256\_broadcastsd\_pd(\_\_m128d a);  
 VBROADCASTSD \_\_m256d \_\_mm256\_mask\_broadcastsd\_pd(\_\_m256d s, \_\_mmask8 k, \_\_m128d a);  
 VBROADCASTSD \_\_m256d \_\_mm256\_maskz\_broadcastsd\_pd( \_\_mmask8 k, \_\_m128d a);  
 VBROADCASTSD \_\_m256d \_\_mm256\_broadcast\_sd(double \*a);  
 VBROADCASTSS \_\_m512 \_\_mm512\_broadcastss\_ps( \_\_m128 a);  
 VBROADCASTSS \_\_m512 \_\_mm512\_mask\_broadcastss\_ps(\_\_m512 s, \_\_mmask16 k, \_\_m128 a);  
 VBROADCASTSS \_\_m512 \_\_mm512\_maskz\_broadcastss\_ps( \_\_mmask16 k, \_\_m128 a);  
 VBROADCASTSS \_\_m256 \_\_mm256\_broadcastss\_ps(\_\_m128 a);  
 VBROADCASTSS \_\_m256 \_\_mm256\_mask\_broadcast\_ss(\_\_m256 s, \_\_mmask8 k, \_\_m128 a);  
 VBROADCASTSS \_\_m256 \_\_mm256\_maskz\_broadcast\_ss( \_\_mmask8 k, \_\_m128 a);  
 VBROADCASTSS \_\_m128 \_\_mm\_broadcastss\_ps(\_\_m128 a);  
 VBROADCASTSS \_\_m128 \_\_mm\_mask\_broadcast\_ss(\_\_m128 s, \_\_mmask8 k, \_\_m128 a);  
 VBROADCASTSS \_\_m128 \_\_mm\_maskz\_broadcast\_ss( \_\_mmask8 k, \_\_m128 a);  
 VBROADCASTSS \_\_m128 \_\_mm\_broadcast\_ss(float \*a);  
 VBROADCASTSS \_\_m256 \_\_mm256\_broadcast\_ss(float \*a);  
 VBROADCASTF128 \_\_m256 \_\_mm256\_broadcast\_ps(\_\_m128 \* a);  
 VBROADCASTF128 \_\_m256d \_\_mm256\_broadcast\_pd(\_\_m128d \* a);

**Exceptions**

VEX-encoded instructions, see Exceptions Type 6;

EVEX-encoded instructions, see Exceptions Type E6.

#UD                      If VEX.L = 0 for VBROADCASTSD or VBROADCASTF128.  
                           If EVEX.L'L = 0 for VBROADCASTSD/VBROADCASTF32X2/VBROADCASTF32X4/VBROADCASTF64X2.  
                           If EVEX.L'L < 10b for VBROADCASTF32X8/VBROADCASTF64X4.

## VPBROADCASTM—Broadcast Mask to Vector Register

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.F3.0F38.W1 2A /r VPBROADCASTMB2Q xmm1, k1	RM	V/V	AVX512VL AVX512CD	Broadcast low byte value in k1 to two locations in xmm1.
EVEX.256.F3.0F38.W1 2A /r VPBROADCASTMB2Q ymm1, k1	RM	V/V	AVX512VL AVX512CD	Broadcast low byte value in k1 to four locations in ymm1.
EVEX.512.F3.0F38.W1 2A /r VPBROADCASTMB2Q zmm1, k1	RM	V/V	AVX512CD	Broadcast low byte value in k1 to eight locations in zmm1.
EVEX.128.F3.0F38.W0 3A /r VPBROADCASTMW2D xmm1, k1	RM	V/V	AVX512VL AVX512CD	Broadcast low word value in k1 to four locations in xmm1.
EVEX.256.F3.0F38.W0 3A /r VPBROADCASTMW2D ymm1, k1	RM	V/V	AVX512VL AVX512CD	Broadcast low word value in k1 to eight locations in ymm1.
EVEX.512.F3.0F38.W0 3A /r VPBROADCASTMW2D zmm1, k1	RM	V/V	AVX512CD	Broadcast low word value in k1 to sixteen locations in zmm1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Broadcasts the zero-extended 64/32 bit value of the low byte/word of the source operand (the second operand) to each 64/32 bit element of the destination operand (the first operand). The source operand is an opmask register. The destination operand is a ZMM register (EVEX.512), YMM register (EVEX.256), or XMM register (EVEX.128).

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

### Operation

#### VPBROADCASTMB2Q

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

    i ← j\*64

    DEST[i+63:i] ← ZeroExtend(SRC[7:0])

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

#### VPBROADCASTMW2D

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

    i ← j\*32

    DEST[i+31:i] ← ZeroExtend(SRC[15:0])

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

### Intel C/C++ Compiler Intrinsic Equivalent

VPBROADCASTMB2Q \_\_m512i \_\_mm512\_broadcastmb\_epi64( \_\_mmask8);

VPBROADCASTMW2D \_\_m512i \_\_mm512\_broadcastmw\_epi32( \_\_mmask16);

VPBROADCASTMB2Q \_\_m256i \_\_mm256\_broadcastmb\_epi64(\_\_mmask8);  
VPBROADCASTMW2D \_\_m256i \_\_mm256\_broadcastmw\_epi32(\_\_mmask8);  
VPBROADCASTMB2Q \_\_m128i \_\_mm\_broadcastmb\_epi64(\_\_mmask8);  
VPBROADCASTMW2D \_\_m128i \_\_mm\_broadcastmw\_epi32(\_\_mmask8);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

EVEX-encoded instruction, see Exceptions Type E6NF.

## VCOMPRESSPD—Store Sparse Packed Double-Precision Floating-Point Values into Dense Memory

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W1 8A /r VCOMPRESSPD xmm1/m128 {k1}{z}, xmm2	T1S	V/V	AVX512VL AVX512F	Compress packed double-precision floating-point values from xmm2 to xmm1/m128 using writemask k1.
EVEX.256.66.0F38.W1 8A /r VCOMPRESSPD ymm1/m256 {k1}{z}, ymm2	T1S	V/V	AVX512VL AVX512F	Compress packed double-precision floating-point values from ymm2 to ymm1/m256 using writemask k1.
EVEX.512.66.0F38.W1 8A /r VCOMPRESSPD zmm1/m512 {k1}{z}, zmm2	T1S	V/V	AVX512F	Compress packed double-precision floating-point values from zmm2 using control mask k1 to zmm1/m512.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
T1S	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

### Description

Compress (store) up to 8 double-precision floating-point values from the source operand (the second operand) as a contiguous vector to the destination operand (the first operand). The source operand is a ZMM/YMM/XMM register, the destination operand can be a ZMM/YMM/XMM register or a 512/256/128-bit memory location.

The opmask register k1 selects the active elements (partial vector or possibly non-contiguous if less than 8 active elements) from the source operand to compress into a contiguous vector. The contiguous vector is written to the destination starting from the low element of the destination operand.

Memory destination version: Only the contiguous vector is written to the destination memory location. EVEX.z must be zero.

Register destination version: If the vector length of the contiguous vector is less than that of the input vector in the source operand, the upper bits of the destination register are unmodified if EVEX.z is not set, otherwise the upper bits are zeroed.

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

Note that the compressed displacement assumes a pre-scaling (N) corresponding to the size of one single element instead of the size of the full vector.

### Operation

**VCOMPRESSPD (EVEX encoded versions) store form**

(KL, VL) = (2, 128), (4, 256), (8, 512)

SIZE ← 64

k ← 0

FOR j ← 0 TO KL-1

  i ← j \* 64

  IF k1[j] OR \*no writemask\*

  THEN

    DEST[k+SIZE-1:k] ← SRC[i+63:i]

    k ← k + SIZE

  FI;

ENDFOR

**VCOMPRESSPD (EVEX encoded versions) reg-reg form**

(KL, VL) = (2, 128), (4, 256), (8, 512)

SIZE ← 64

k ← 0

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN

DEST[k+SIZE-1:k] ← SRC[i+63:i]

k ← k + SIZE

FI;

ENDFOR

IF \*merging-masking\*

THEN \*DEST[VL-1:k] remains unchanged\*

ELSE DEST[VL-1:k] ← 0

FI

DEST[MAX\_VL-1:VL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

VCOMPRESSPD \_\_m512d \_\_mm512\_mask\_compress\_pd( \_\_m512d s, \_\_mmask8 k, \_\_m512d a);

VCOMPRESSPD \_\_m512d \_\_mm512\_maskz\_compress\_pd( \_\_mmask8 k, \_\_m512d a);

VCOMPRESSPD void \_\_mm512\_mask\_compressstoreu\_pd( void \* d, \_\_mmask8 k, \_\_m512d a);

VCOMPRESSPD \_\_m256d \_\_mm256\_mask\_compress\_pd( \_\_m256d s, \_\_mmask8 k, \_\_m256d a);

VCOMPRESSPD \_\_m256d \_\_mm256\_maskz\_compress\_pd( \_\_mmask8 k, \_\_m256d a);

VCOMPRESSPD void \_\_mm256\_mask\_compressstoreu\_pd( void \* d, \_\_mmask8 k, \_\_m256d a);

VCOMPRESSPD \_\_m128d \_\_mm\_mask\_compress\_pd( \_\_m128d s, \_\_mmask8 k, \_\_m128d a);

VCOMPRESSPD \_\_m128d \_\_mm\_maskz\_compress\_pd( \_\_mmask8 k, \_\_m128d a);

VCOMPRESSPD void \_\_mm\_mask\_compressstoreu\_pd( void \* d, \_\_mmask8 k, \_\_m128d a);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

EVEX-encoded instructions, see Exceptions Type E4.nb.

#UD                   If EVEX.vvvv != 1111B.



## VCOMPRESSPS—Store Sparse Packed Single-Precision Floating-Point Values into Dense Memory

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 8A /r VCOMPRESSPS xmm1/m128 {k1}{z}, xmm2	T1S	V/V	AVX512VL AVX512F	Compress packed single-precision floating-point values from xmm2 to xmm1/m128 using writemask k1.
EVEX.256.66.0F38.W0 8A /r VCOMPRESSPS ymm1/m256 {k1}{z}, ymm2	T1S	V/V	AVX512VL AVX512F	Compress packed single-precision floating-point values from ymm2 to ymm1/m256 using writemask k1.
EVEX.512.66.0F38.W0 8A /r VCOMPRESSPS zmm1/m512 {k1}{z}, zmm2	T1S	V/V	AVX512F	Compress packed single-precision floating-point values from zmm2 using control mask k1 to zmm1/m512.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
T1S	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

### Description

Compress (stores) up to 16 single-precision floating-point values from the source operand (the second operand) to the destination operand (the first operand). The source operand is a ZMM/YMM/XMM register, the destination operand can be a ZMM/YMM/XMM register or a 512/256/128-bit memory location.

The opmask register k1 selects the active elements (a partial vector or possibly non-contiguous if less than 16 active elements) from the source operand to compress into a contiguous vector. The contiguous vector is written to the destination starting from the low element of the destination operand.

Memory destination version: Only the contiguous vector is written to the destination memory location. EVEX.z must be zero.

Register destination version: If the vector length of the contiguous vector is less than that of the input vector in the source operand, the upper bits of the destination register are unmodified if EVEX.z is not set, otherwise the upper bits are zeroed.

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

Note that the compressed displacement assumes a pre-scaling (N) corresponding to the size of one single element instead of the size of the full vector.

### Operation

**VCOMPRESSPS (EVEX encoded versions) store form**

(KL, VL) = (4, 128), (8, 256), (16, 512)

SIZE ← 32

k ← 0

FOR j ← 0 TO KL-1

  i ← j \* 32

  IF k1[j] OR \*no writemask\*

    THEN

      DEST[k+SIZE-1:k] ← SRC[i+31:i]

      k ← k + SIZE

  FI;

ENDFOR;

**VCOMPRESSPS (EVEX encoded versions) reg-reg form**

(KL, VL) = (4, 128), (8, 256), (16, 512)

SIZE ← 32

k ← 0

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\*

THEN

DEST[k+SIZE-1:k] ← SRC[i+31:i]

k ← k + SIZE

FI;

ENDFOR

IF \*merging-masking\*

THEN \*DEST[VL-1:k] remains unchanged\*

ELSE DEST[VL-1:k] ← 0

FI

DEST[MAX\_VL-1:VL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

VCOMPRESSPS \_\_m512 \_\_mm512\_mask\_compress\_ps( \_\_m512 s, \_\_mmask16 k, \_\_m512 a);

VCOMPRESSPS \_\_m512 \_\_mm512\_maskz\_compress\_ps( \_\_mmask16 k, \_\_m512 a);

VCOMPRESSPS void \_\_mm512\_mask\_compressstoreu\_ps( void \* d, \_\_mmask16 k, \_\_m512 a);

VCOMPRESSPS \_\_m256 \_\_mm256\_mask\_compress\_ps( \_\_m256 s, \_\_mmask8 k, \_\_m256 a);

VCOMPRESSPS \_\_m256 \_\_mm256\_maskz\_compress\_ps( \_\_mmask8 k, \_\_m256 a);

VCOMPRESSPS void \_\_mm256\_mask\_compressstoreu\_ps( void \* d, \_\_mmask8 k, \_\_m256 a);

VCOMPRESSPS \_\_m128 \_\_mm\_mask\_compress\_ps( \_\_m128 s, \_\_mmask8 k, \_\_m128 a);

VCOMPRESSPS \_\_m128 \_\_mm\_maskz\_compress\_ps( \_\_mmask8 k, \_\_m128 a);

VCOMPRESSPS void \_\_mm\_mask\_compressstoreu\_ps( void \* d, \_\_mmask8 k, \_\_m128 a);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

EVEX-encoded instructions, see Exceptions Type E4.nb.

#UD                    If EVEX.vvvv != 1111B.

## VCVTPD2QQ—Convert Packed Double-Precision Floating-Point Values to Packed Quadword Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F.W1 7B /r VCVTPD2QQ xmm1 {k1}{z}, xmm2/m128/m64bcst	FV	V/V	AVX512VL AVX512DQ	Convert two packed double-precision floating-point values from xmm2/m128/m64bcst to two packed quadword integers in xmm1 with writemask k1.
EVEX.256.66.0F.W1 7B /r VCVTPD2QQ ymm1 {k1}{z}, ymm2/m256/m64bcst	FV	V/V	AVX512VL AVX512DQ	Convert four packed double-precision floating-point values from ymm2/m256/m64bcst to four packed quadword integers in ymm1 with writemask k1.
EVEX.512.66.0F.W1 7B /r VCVTPD2QQ zmm1 {k1}{z}, zmm2/m512/m64bcst{er}	FV	V/V	AVX512DQ	Convert eight packed double-precision floating-point values from zmm2/m512/m64bcst to eight packed quadword integers in zmm1 with writemask k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
FV	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Converts packed double-precision floating-point values in the source operand (second operand) to packed quadword integers in the destination operand (first operand).

EVEX encoded versions: The source operand is a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operation is a ZMM/YMM/XMM register conditionally updated with writemask k1.

When a conversion is inexact, 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, the indefinite integer value ( $2^{w-1}$ , where w represents the number of bits in the destination format) is returned.

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

### Operation

**VCVTPD2QQ (EVEX encoded version) when src operand is a register**

(KL, VL) = (2, 128), (4, 256), (8, 512)

IF (VL == 512) AND (EVEX.b == 1)

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN DEST[i+63:i] ←

Convert\_Double\_Precision\_Floating\_Point\_To\_QuadInteger(SRC[i+63:i])

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE ; zeroing-masking

```

                DEST[i+63:i] ← 0
            FI
        FI;
    ENDFOR
    DEST[MAX_VL-1:VL] ← 0

```

**VCVTPD2QQ (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_Double_Precision_Floating_Point_To_QuadInteger(SRC[63:0])
                ELSE
                    DEST[i+63:i] ← Convert_Double_Precision_Floating_Point_To_QuadInteger(SRC[i+63:i])
            FI;
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+63:i] ← 0
            FI
        FI;
    ENDFOR
    DEST[MAX_VL-1:VL] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VCVTPD2QQ __m512i __mm512_cvtpd_epi64( __m512d a);
VCVTPD2QQ __m512i __mm512_mask_cvtpd_epi64( __m512i s, __mmask8 k, __m512d a);
VCVTPD2QQ __m512i __mm512_maskz_cvtpd_epi64( __mmask8 k, __m512d a);
VCVTPD2QQ __m512i __mm512_cvt_roundpd_epi64( __m512d a, int r);
VCVTPD2QQ __m512i __mm512_mask_cvt_roundpd_epi64( __m512i s, __mmask8 k, __m512d a, int r);
VCVTPD2QQ __m512i __mm512_maskz_cvt_roundpd_epi64( __mmask8 k, __m512d a, int r);
VCVTPD2QQ __m256i __mm256_mask_cvtpd_epi64( __m256i s, __mmask8 k, __m256d a);
VCVTPD2QQ __m256i __mm256_maskz_cvtpd_epi64( __mmask8 k, __m256d a);
VCVTPD2QQ __m128i __mm_mask_cvtpd_epi64( __m128i s, __mmask8 k, __m128d a);
VCVTPD2QQ __m128i __mm_maskz_cvtpd_epi64( __mmask8 k, __m128d a);
VCVTPD2QQ __m256i __mm256_cvtpd_epi64( __m256d src)
VCVTPD2QQ __m128i __mm_cvtpd_epi64( __m128d src)

```

**SIMD Floating-Point Exceptions**

Invalid, Precision

**Other Exceptions**

EVEX-encoded instructions, see Exceptions Type E2

#UD If EVEX.vvvv != 1111B.

## VCVTPD2UDQ—Convert Packed Double-Precision Floating-Point Values to Packed Unsigned Doubleword Integers

Opcode Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.0F.W1 79 /r VCVTPD2UDQ xmm1 {k1}{z}, xmm2/m128/m64bcst	FV	V/V	AVX512VL AVX512F	Convert two packed double-precision floating-point values in xmm2/m128/m64bcst to two unsigned doubleword integers in xmm1 subject to writemask k1.
EVEX.256.0F.W1 79 /r VCVTPD2UDQ xmm1 {k1}{z}, ymm2/m256/m64bcst	FV	V/V	AVX512VL AVX512F	Convert four packed double-precision floating-point values in ymm2/m256/m64bcst to four unsigned doubleword integers in xmm1 subject to writemask k1.
EVEX.512.0F.W1 79 /r VCVTPD2UDQ ymm1 {k1}{z}, zmm2/m512/m64bcst{er}	FV	V/V	AVX512F	Convert eight packed double-precision floating-point values in zmm2/m512/m64bcst to eight unsigned doubleword integers in ymm1 subject to writemask k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
FV	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Converts packed double-precision floating-point values in the source operand (the second operand) to packed unsigned doubleword integers in the destination operand (the first operand).

When a conversion is inexact, 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, the integer value  $2^w - 1$  is returned, where  $w$  represents the number of bits in the destination format.

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 64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask  $k1$ . The upper bits (MAX\_VL-1:256) of the corresponding destination are zeroed.

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

### Operation

#### VCVTPD2UDQ (EVEX encoded versions) when src2 operand is a register

(KL, VL) = (2, 128), (4, 256), (8, 512)

IF (VL = 512) AND (EVEX.b = 1)

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR  $j \leftarrow 0$  TO KL-1

$i \leftarrow j * 32$

$k \leftarrow j * 64$

IF  $k1[j]$  OR \*no writemask\*

THEN

DEST[ $i+31:i$ ] ←

Convert\_Double\_Precision\_Floating\_Point\_To\_UInteger(SRC[ $k+63:k$ ])

ELSE

```

        IF *merging-masking*           ; merging-masking
            THEN *DEST[j+31:i] remains unchanged*
        ELSE                             ; zeroing-masking
            DEST[j+31:i] ← 0
        FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL/2] ← 0

```

#### VCVTPD2UDQ (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[j+31:i] ←
                    Convert_Double_Precision_Floating_Point_To_UInteger(SRC[63:0])
                ELSE
                    DEST[j+31:i] ←
                    Convert_Double_Precision_Floating_Point_To_UInteger(SRC[k+63:k])
            FI;
        ELSE
            IF *merging-masking*           ; merging-masking
                THEN *DEST[j+31:i] remains unchanged*
            ELSE                             ; zeroing-masking
                DEST[j+31:i] ← 0
            FI
        FI;
    ENDFOR
    DEST[MAX_VL-1:VL/2] ← 0

```

#### Intel C/C++ Compiler Intrinsic Equivalent

```

VCVTPD2UDQ __m256i __mm512_cvtpd_epu32( __m512d a);
VCVTPD2UDQ __m256i __mm512_mask_cvtpd_epu32( __m256i s, __mmask8 k, __m512d a);
VCVTPD2UDQ __m256i __mm512_maskz_cvtpd_epu32( __mmask8 k, __m512d a);
VCVTPD2UDQ __m256i __mm512_cvt_roundpd_epu32( __m512d a, int r);
VCVTPD2UDQ __m256i __mm512_mask_cvt_roundpd_epu32( __m256i s, __mmask8 k, __m512d a, int r);
VCVTPD2UDQ __m256i __mm512_maskz_cvt_roundpd_epu32( __mmask8 k, __m512d a, int r);
VCVTPD2UDQ __m128i __mm256_mask_cvtpd_epu32( __m128i s, __mmask8 k, __m256d a);
VCVTPD2UDQ __m128i __mm256_maskz_cvtpd_epu32( __mmask8 k, __m256d a);
VCVTPD2UDQ __m128i __mm_mask_cvtpd_epu32( __m128i s, __mmask8 k, __m128d a);
VCVTPD2UDQ __m128i __mm_maskz_cvtpd_epu32( __mmask8 k, __m128d a);

```

#### SIMD Floating-Point Exceptions

Invalid, Precision

#### Other Exceptions

EVEX-encoded instructions, see Exceptions Type E2.

#UD If EVEX.vvvv != 1111B.

## VCVTPD2UQQ—Convert Packed Double-Precision Floating-Point Values to Packed Unsigned Quadword Integers

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F.W1 79 /r VCVTPD2UQQ xmm1 {k1}{z}, xmm2/m128/m64bcst	FV	V/V	AVX512VL AVX512DQ	Convert two packed double-precision floating-point values from xmm2/mem to two packed unsigned quadword integers in xmm1 with writemask k1.
EVEX.256.66.0F.W1 79 /r VCVTPD2UQQ ymm1 {k1}{z}, ymm2/m256/m64bcst	FV	V/V	AVX512VL AVX512DQ	Convert fourth packed double-precision floating-point values from ymm2/mem to four packed unsigned quadword integers in ymm1 with writemask k1.
EVEX.512.66.0F.W1 79 /r VCVTPD2UQQ zmm1 {k1}{z}, zmm2/m512/m64bcst{er}	FV	V/V	AVX512DQ	Convert eight packed double-precision floating-point values from zmm2/mem to eight packed unsigned quadword integers in zmm1 with writemask k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
FV	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Converts packed double-precision floating-point values in the source operand (second operand) to packed unsigned quadword integers in the destination operand (first operand).

When a conversion is inexact, 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, the integer value  $2^w - 1$  is returned, where  $w$  represents the number of bits in the destination format.

The source operand is a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operation is a ZMM/YMM/XMM register conditionally updated with writemask k1.

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

### Operation

#### VCVTPD2UQQ (EVEX encoded versions) when src operand is a register

(KL, VL) = (2, 128), (4, 256), (8, 512)

IF (VL == 512) AND (EVEX.b == 1)

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR  $j \leftarrow 0$  TO  $KL-1$

$i \leftarrow j * 64$

IF  $k1[j]$  OR \*no writemask\*

THEN  $DEST[i+63:i] \leftarrow$

Convert\_Double\_Precision\_Floating\_Point\_To\_UQuadInteger(SRC[i+63:i])

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE ; zeroing-masking

```

                DEST[i+63:i] ← 0
            FI
        FI;
    ENDFOR
    DEST[MAX_VL-1:VL] ← 0

```

**VCVTPD2UQQ (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
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b == 1)
                THEN
                    DEST[i+63:i] ←
                    Convert_Double_Precision_Floating_Point_To_UQuadInteger(SRC[63:0])
                ELSE
                    DEST[i+63:i] ←
                    Convert_Double_Precision_Floating_Point_To_UQuadInteger(SRC[i+63:i])
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
            ELSE                                  ; zeroing-masking
                DEST[i+63:i] ← 0
            FI
        FI;
    ENDFOR
    DEST[MAX_VL-1:VL] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VCVTPD2UQQ __m512i __mm512_cvtpd_epu64( __m512d a);
VCVTPD2UQQ __m512i __mm512_mask_cvtpd_epu64( __m512i s, __mmask8 k, __m512d a);
VCVTPD2UQQ __m512i __mm512_maskz_cvtpd_epu64( __mmask8 k, __m512d a);
VCVTPD2UQQ __m512i __mm512_cvt_roundpd_epu64( __m512d a, int r);
VCVTPD2UQQ __m512i __mm512_mask_cvt_roundpd_epu64( __m512i s, __mmask8 k, __m512d a, int r);
VCVTPD2UQQ __m512i __mm512_maskz_cvt_roundpd_epu64( __mmask8 k, __m512d a, int r);
VCVTPD2UQQ __m256i __mm256_mask_cvtpd_epu64( __m256i s, __mmask8 k, __m256d a);
VCVTPD2UQQ __m256i __mm256_maskz_cvtpd_epu64( __mmask8 k, __m256d a);
VCVTPD2UQQ __m128i __mm_mask_cvtpd_epu64( __m128i s, __mmask8 k, __m128d a);
VCVTPD2UQQ __m128i __mm_maskz_cvtpd_epu64( __mmask8 k, __m128d a);
VCVTPD2UQQ __m256i __mm256_cvtpd_epu64( __m256d src)
VCVTPD2UQQ __m128i __mm_cvtpd_epu64( __m128d src)

```

**SIMD Floating-Point Exceptions**

Invalid, Precision

**Other Exceptions**

EVEX-encoded instructions, see Exceptions Type E2

#UD If EVEX.vvvv != 1111B.



## VCVTPH2PS—Convert 16-bit FP values to Single-Precision FP values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F38.W0 1313 /r VCVTPH2PS xmm1, xmm2/m64	RM	V/V	F16C	Convert four packed half precision (16-bit) floating-point values in xmm2/m64 to packed single-precision floating-point value in xmm1.
VEX.256.66.0F38.W0 1313 /r VCVTPH2PS ymm1, xmm2/m128	RM	V/V	F16C	Convert eight packed half precision (16-bit) floating-point values in xmm2/m128 to packed single-precision floating-point value in ymm1.
EVEX.128.66.0F38.W0 1313 /r VCVTPH2PS xmm1 {k1}{z}, xmm2/m64	HVM	V/V	AVX512VL AVX512F	Convert four packed half precision (16-bit) floating-point values in xmm2/m64 to packed single-precision floating-point values in xmm1.
EVEX.256.66.0F38.W0 1313 /r VCVTPH2PS ymm1 {k1}{z}, xmm2/m128	HVM	V/V	AVX512VL AVX512F	Convert eight packed half precision (16-bit) floating-point values in xmm2/m128 to packed single-precision floating-point values in ymm1.
EVEX.512.66.0F38.W0 1313 /r VCVTPH2PS zmm1 {k1}{z}, ymm2/m256 {sae}	HVM	V/V	AVX512F	Convert sixteen packed half precision (16-bit) floating-point values in ymm2/m256 to packed single-precision floating-point values in zmm1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
HVM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Converts packed half precision (16-bits) floating-point values in the low-order bits of the source operand (the second operand) to packed single-precision floating-point values and writes the converted values into the destination operand (the first operand).

If case of a denormal operand, the correct normal result is returned. MXCSR.DAZ is ignored and is treated as if it 0. No denormal exception is reported on MXCSR.

VEX.128 version: The source operand is a XMM register or 64-bit memory location. The destination operand is a XMM register. The upper bits (MAX\_VL-1:128) of the corresponding destination register are zeroed.

VEX.256 version: The source operand is a XMM register or 128-bit memory location. The destination operand is a YMM register. Bits (MAX\_VL-1:256) of the corresponding destination register are zeroed.

EVEX encoded versions: The source operand is a YMM/XMM/XMM (low 64-bits) register or a 256/128/64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

The diagram below illustrates how data is converted from four packed half precision (in 64 bits) to four single precision (in 128 bits) FP values.

Note: VEX.vvvv and EVEX.vvvv are reserved (must be 1111b).

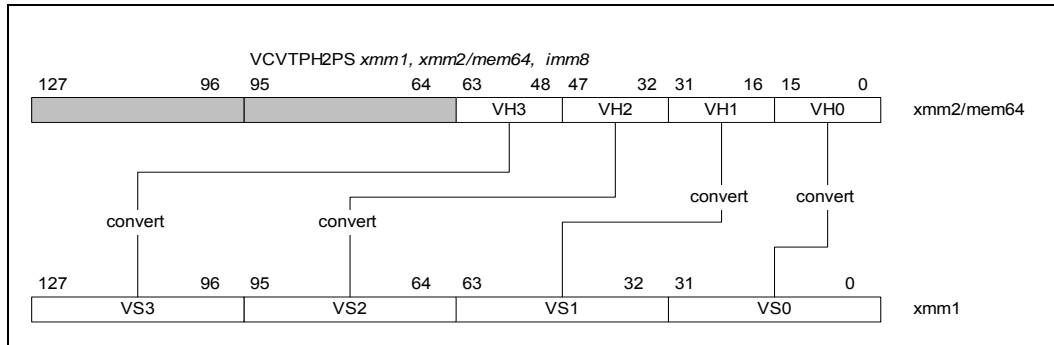


Figure 5-6. VCVTPH2PS (128-bit Version)

**Operation**

```

vCvt_h2s(SRC1[15:0])
{
RETURN Cvt_Half_Precision_To_Single_Precision(SRC1[15:0]);
}

```

**VCVTPH2PS (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

  i ← j \* 32

  k ← j \* 16

  IF k1[j] OR \*no writemask\*

    THEN DEST[i+31:i] ←

      vCvt\_h2s(SRC[k+15:k])

  ELSE

    IF \*merging-masking\* ; merging-masking

      THEN \*DEST[i+31:i] remains unchanged\*

    ELSE ; zeroing-masking

      DEST[i+31:i] ← 0

  FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VCVTPH2PS (VEX.256 encoded version)**

DEST[31:0] ← vCvt\_h2s(SRC1[15:0]);

DEST[63:32] ← vCvt\_h2s(SRC1[31:16]);

DEST[95:64] ← vCvt\_h2s(SRC1[47:32]);

DEST[127:96] ← vCvt\_h2s(SRC1[63:48]);

DEST[159:128] ← vCvt\_h2s(SRC1[79:64]);

DEST[191:160] ← vCvt\_h2s(SRC1[95:80]);

DEST[223:192] ← vCvt\_h2s(SRC1[111:96]);

DEST[255:224] ← vCvt\_h2s(SRC1[127:112]);

DEST[MAX\_VL-1:256] ← 0

**VCVTPH2PS (VEX.128 encoded version)**

```

DEST[31:0] ← vCvt_h2s(SRC1[15:0]);
DEST[63:32] ← vCvt_h2s(SRC1[31:16]);
DEST[95:64] ← vCvt_h2s(SRC1[47:32]);
DEST[127:96] ← vCvt_h2s(SRC1[63:48]);
DEST[MAX_VL-1:128] ← 0

```

**Flags Affected**

None

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VCVTPH2PS __m512 __mm512_cvtph_ps( __m256i a);
VCVTPH2PS __m512 __mm512_mask_cvtph_ps(__m512 s, __mmask16 k, __m256i a);
VCVTPH2PS __m512 __mm512_maskz_cvtph_ps(__mmask16 k, __m256i a);
VCVTPH2PS __m512 __mm512_cvt_roundph_ps( __m256i a, int sae);
VCVTPH2PS __m512 __mm512_mask_cvt_roundph_ps(__m512 s, __mmask16 k, __m256i a, int sae);
VCVTPH2PS __m512 __mm512_maskz_cvt_roundph_ps( __mmask16 k, __m256i a, int sae);
VCVTPH2PS __m256 __mm256_mask_cvtph_ps(__m256 s, __mmask8 k, __m128i a);
VCVTPH2PS __m256 __mm256_maskz_cvtph_ps(__mmask8 k, __m128i a);
VCVTPH2PS __m128 __mm_mask_cvtph_ps(__m128 s, __mmask8 k, __m128i a);
VCVTPH2PS __m128 __mm_maskz_cvtph_ps(__mmask8 k, __m128i a);
VCVTPH2PS __m128 __mm_cvtph_ps( __m128i m1);
VCVTPH2PS __m256 __mm256_cvtph_ps( __m128i m1)

```

**SIMD Floating-Point Exceptions**

Invalid

**Other Exceptions**

VEX-encoded instructions, see Exceptions Type 11 (do not report #AC);

EVEX-encoded instructions, see Exceptions Type E11.

#UD If VEX.W=1.

#UD If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

## VCVTPS2PH—Convert Single-Precision FP value to 16-bit FP value

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F3A.W0 1D 1D/r ib VCVTPS2PH xmm1/m64, xmm2, imm8	MRI	V/V	F16C	Convert four packed single-precision floating-point values in xmm2 to packed half-precision (16-bit) floating-point values in xmm1/m64. Imm8 provides rounding controls.
VEX.256.66.0F3A.W0 1D1D /r ib VCVTPS2PH xmm1/m128, ymm2, imm8	MRI	V/V	F16C	Convert eight packed single-precision floating-point values in ymm2 to packed half-precision (16-bit) floating-point values in xmm1/m128. Imm8 provides rounding controls.
EVEX.128.66.0F3A.W0 1D1D /r ib VCVTPS2PH xmm1/m64 {k1}{z}, xmm2, imm8	HVM	V/V	AVX512VL AVX512F	Convert four packed single-precision floating-point values in xmm2 to packed half-precision (16-bit) floating-point values in xmm1/m64. Imm8 provides rounding controls.
EVEX.256.66.0F3A.W0 1D1D /r ib VCVTPS2PH xmm1/m128 {k1}{z}, ymm2, imm8	HVM	V/V	AVX512VL AVX512F	Convert eight packed single-precision floating-point values in ymm2 to packed half-precision (16-bit) floating-point values in xmm1/m128. Imm8 provides rounding controls.
EVEX.512.66.0F3A.W0 1D1D /r ib VCVTPS2PH ymm1/m256 {k1}{z}, zmm2{sae}, imm8	HVM	V/V	AVX512F	Convert sixteen packed single-precision floating-point values in zmm2 to packed half-precision (16-bit) floating-point values in ymm1/m256. Imm8 provides rounding controls.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MRI	ModRM:r/m (w)	ModRM:reg (r)	Imm8	NA
HVM	ModRM:r/m (w)	ModRM:reg (r)	Imm8	NA

### Description

Convert packed single-precision floating values in the source operand to half-precision (16-bit) floating-point values and store to the destination operand. The rounding mode is specified using the immediate field (imm8).

Underflow results (i.e., tiny results) are converted to denormals. MXCSR.FTZ is ignored. If a source element is denormal relative to the input format with DM masked and at least one of PM or UM unmasked; a SIMD exception will be raised with DE, UE and PE set.

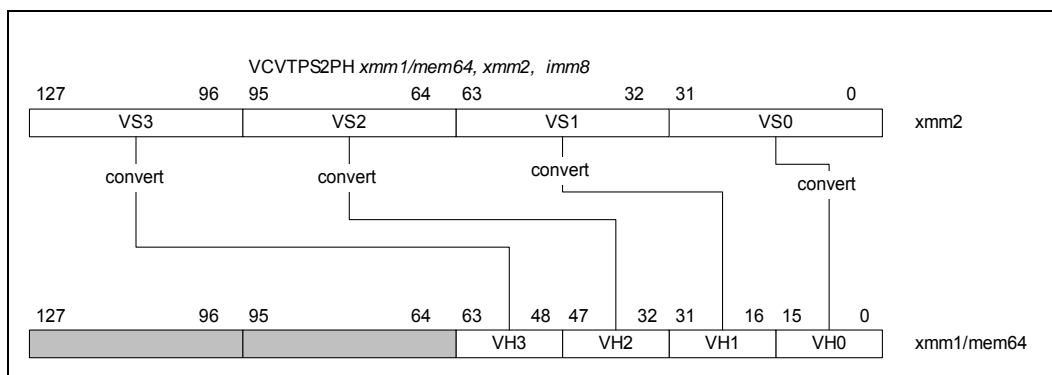


Figure 5-7. VCVTPS2PH (128-bit Version)

The immediate byte defines several bit fields that control rounding operation. The effect and encoding of the RC field are listed in Table 5-3.

**Table 5-3. Immediate Byte Encoding for 16-bit Floating-Point Conversion Instructions**

Bits	Field Name/value	Description	Comment
Imm[1:0]	RC=00B	Round to nearest even	If Imm[2] = 0
	RC=01B	Round down	
	RC=10B	Round up	
	RC=11B	Truncate	
Imm[2]	MS1=0	Use Imm[1:0] for rounding	Ignore MXCSR.RC
	MS1=1	Use MXCSR.RC for rounding	
Imm[7:3]	Ignored	Ignored by processor	

VEX.128 version: The source operand is a XMM register. The destination operand is a XMM register or 64-bit memory location. If the destination operand is a register then the upper bits (MAX\_VL-1:64) of corresponding register are zeroed.

VEX.256 version: The source operand is a YMM register. The destination operand is a XMM register or 128-bit memory location. If the destination operand is a register, the upper bits (MAX\_VL-1:128) of the corresponding destination register are zeroed.

Note: VEX.vvvv and EVEX.vvvv are reserved (must be 1111b).

EVEX encoded versions: The source operand is a ZMM/YMM/XMM register. The destination operand is a YMM/XMM/XMM (low 64-bits) register or a 256/128/64-bit memory location, conditionally updated with writemask k1. Bits (MAX\_VL-1:256/128/64) of the corresponding destination register are zeroed.

### Operation

```
vCvt_s2h(SRC1[31:0])
{
  IF Imm[2] = 0
  THEN ; using Imm[1:0] for rounding control, see Table 5-3
    RETURN Cvt_Single_Precision_To_Half_Precision_FP_Imm(SRC1[31:0]);
  ELSE ; using MXCSR.RC for rounding control
    RETURN Cvt_Single_Precision_To_Half_Precision_FP_Mxcsr(SRC1[31:0]);
  FI;
}
```

### VCVTPS2PH (EVEX encoded versions) when dest is a register

(KL, VL) = (4, 128), (8, 256), (16, 512)

```
FOR j ← 0 TO KL-1
  i ← j * 16
  k ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+15:j] ←
      vCvt_s2h(SRC[k+31:k])
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+15:j] remains unchanged*
      ELSE ; zeroing-masking
        DEST[i+15:j] ← 0
      FI
  FI;
ENDFOR
DEST[MAX_VL-1:VL/2] ← 0
```

**VCVTPS2PH (EVEX encoded versions) when dest is memory**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

i ← j \* 16

k ← j \* 32

IF k1[j] OR \*no writemask\*

THEN DEST[i+15:i] ←

vCvt\_s2h(SRC[k+31:k])

ELSE

\*DEST[i+15:i] remains unchanged\* ; merging-masking

FI;

ENDFOR

**VCVTPS2PH (VEX.256 encoded version)**

DEST[15:0] ← vCvt\_s2h(SRC1[31:0]);

DEST[31:16] ← vCvt\_s2h(SRC1[63:32]);

DEST[47:32] ← vCvt\_s2h(SRC1[95:64]);

DEST[63:48] ← vCvt\_s2h(SRC1[127:96]);

DEST[79:64] ← vCvt\_s2h(SRC1[159:128]);

DEST[95:80] ← vCvt\_s2h(SRC1[191:160]);

DEST[111:96] ← vCvt\_s2h(SRC1[223:192]);

DEST[127:112] ← vCvt\_s2h(SRC1[255:224]);

DEST[MAX\_VL-1:128] ← 0

**VCVTPS2PH (VEX.128 encoded version)**

DEST[15:0] ← vCvt\_s2h(SRC1[31:0]);

DEST[31:16] ← vCvt\_s2h(SRC1[63:32]);

DEST[47:32] ← vCvt\_s2h(SRC1[95:64]);

DEST[63:48] ← vCvt\_s2h(SRC1[127:96]);

DEST[MAX\_VL-1:64] ← 0

**Flags Affected**

None

**Intel C/C++ Compiler Intrinsic Equivalent**

VCVTPS2PH \_\_m256i \_\_mm512\_cvtps\_ph(\_\_m512 a);

VCVTPS2PH \_\_m256i \_\_mm512\_mask\_cvtps\_ph(\_\_m256i s, \_\_mmask16 k, \_\_m512 a);

VCVTPS2PH \_\_m256i \_\_mm512\_maskz\_cvtps\_ph(\_\_mmask16 k, \_\_m512 a);

VCVTPS2PH \_\_m256i \_\_mm512\_cvt\_roundps\_ph(\_\_m512 a, const int imm);

VCVTPS2PH \_\_m256i \_\_mm512\_mask\_cvt\_roundps\_ph(\_\_m256i s, \_\_mmask16 k, \_\_m512 a, const int imm);

VCVTPS2PH \_\_m256i \_\_mm512\_maskz\_cvt\_roundps\_ph(\_\_mmask16 k, \_\_m512 a, const int imm);

VCVTPS2PH \_\_m128i \_\_mm256\_mask\_cvtps\_ph(\_\_m128i s, \_\_mmask8 k, \_\_m256 a);

VCVTPS2PH \_\_m128i \_\_mm256\_maskz\_cvtps\_ph(\_\_mmask8 k, \_\_m256 a);

VCVTPS2PH \_\_m128i \_\_mm\_mask\_cvtps\_ph(\_\_m128i s, \_\_mmask8 k, \_\_m128 a);

VCVTPS2PH \_\_m128i \_\_mm\_maskz\_cvtps\_ph(\_\_mmask8 k, \_\_m128 a);

VCVTPS2PH \_\_m128i \_\_mm\_cvtps\_ph (\_\_m128 m1, const int imm);

VCVTPS2PH \_\_m128i \_\_mm256\_cvtps\_ph(\_\_m256 m1, const int imm);

**SIMD Floating-Point Exceptions**

Invalid, Underflow, Overflow, Precision, Denormal (if MXCSR.DAZ=0);

**Other Exceptions**

VEX-encoded instructions, see Exceptions Type 11 (do not report #AC);

EVEX-encoded instructions, see Exceptions Type E11.

#UD                    If VEX.W=1.

#UD                    If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

## VCVTSP2UDQ—Convert Packed Single-Precision Floating-Point Values to Packed Unsigned Doubleword Integer Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.OF.W0 79 /r VCVTSP2UDQ xmm1 {k1}{z}, xmm2/m128/m32bcst	FV	V/V	AVX512VL AVX512F	Convert four packed single precision floating-point values from xmm2/m128/m32bcst to four packed unsigned doubleword values in xmm1 subject to writemask k1.
EVEX.256.OF.W0 79 /r VCVTSP2UDQ ymm1 {k1}{z}, ymm2/m256/m32bcst	FV	V/V	AVX512VL AVX512F	Convert eight packed single precision floating-point values from ymm2/m256/m32bcst to eight packed unsigned doubleword values in ymm1 subject to writemask k1.
EVEX.512.OF.W0 79 /r VCVTSP2UDQ zmm1 {k1}{z}, zmm2/m512/m32bcst{er}	FV	V/V	AVX512F	Convert sixteen packed single-precision floating-point values from zmm2/m512/m32bcst to sixteen packed unsigned doubleword values in zmm1 subject to writemask k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
FV	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Converts sixteen packed single-precision floating-point values in the source operand to sixteen unsigned doubleword integers in the destination operand.

When a conversion is inexact, 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, the integer value  $2^w - 1$  is returned, where  $w$  represents the number of bits in the destination format.

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 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask  $k1$ .

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

### Operation

**VCVTSP2UDQ (EVEX encoded versions) when src operand is a register**

(KL, VL) = (4, 128), (8, 256), (16, 512)

IF (VL = 512) AND (EVEX.b = 1)

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR  $j \leftarrow 0$  TO  $KL-1$

$i \leftarrow j * 32$

IF  $k1[j]$  OR \*no writemask\*

THEN  $DEST[i+31:i] \leftarrow$

Convert\_Single\_Precision\_Floating\_Point\_To\_UInteger(SRC[i+31:i])

ELSE

IF \*merging-masking\* ;merging-masking

THEN \*DEST[i+31:i] remains unchanged\*



```

        ELSE                                ; zeroing-masking
            DEST[i+31:i] ← 0
        FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

### VCVTPS2UDQ (EVEX encoded versions) when src operand is a memory source

(KL, VL) = (4, 128), (8, 256), (16, 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_Single_Precision_Floating_Point_To_UInteger(SRC[31:0])
                ELSE
                    DEST[i+31:i] ←
                    Convert_Single_Precision_Floating_Point_To_UInteger(SRC[i+31:i])
            FI;
        ELSE
            IF *merging-masking*           ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
            ELSE                             ; zeroing-masking
                DEST[i+31:i] ← 0
            FI
        FI;
    ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

### Intel C/C++ Compiler Intrinsic Equivalent

```

VCVTPS2UDQ __m512i __mm512_cvtps_epu32( __m512 a);
VCVTPS2UDQ __m512i __mm512_mask_cvtps_epu32( __m512i s, __mmask16 k, __m512 a);
VCVTPS2UDQ __m512i __mm512_maskz_cvtps_epu32( __mmask16 k, __m512 a);
VCVTPS2UDQ __m512i __mm512_cvt_roundps_epu32( __m512 a, int r);
VCVTPS2UDQ __m512i __mm512_mask_cvt_roundps_epu32( __m512i s, __mmask16 k, __m512 a, int r);
VCVTPS2UDQ __m512i __mm512_maskz_cvt_roundps_epu32( __mmask16 k, __m512 a, int r);
VCVTPS2UDQ __m256i __mm256_cvtps_epu32( __m256d a);
VCVTPS2UDQ __m256i __mm256_mask_cvtps_epu32( __m256i s, __mmask8 k, __m256 a);
VCVTPS2UDQ __m256i __mm256_maskz_cvtps_epu32( __mmask8 k, __m256 a);
VCVTPS2UDQ __m128i __mm_cvtps_epu32( __m128 a);
VCVTPS2UDQ __m128i __mm_mask_cvtps_epu32( __m128i s, __mmask8 k, __m128 a);
VCVTPS2UDQ __m128i __mm_maskz_cvtps_epu32( __mmask8 k, __m128 a);

```

### SIMD Floating-Point Exceptions

Invalid, Precision

### Other Exceptions

EVEX-encoded instructions, see Exceptions Type E2.

#UD If EVEX.vvvv != 1111B.

## VCVTQPS2Q—Convert Packed Single Precision Floating-Point Values to Packed Signed Quadword Integer Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F.W0 7B /r VCVTQPS2Q xmm1 {k1}{z}, xmm2/m64/m32bcst	HV	V/V	AVX512VL AVX512DQ	Convert two packed single precision floating-point values from xmm2/m64/m32bcst to two packed signed quadword values in xmm1 subject to writemask k1.
EVEX.256.66.0F.W0 7B /r VCVTQPS2Q ymm1 {k1}{z}, xmm2/m128/m32bcst	HV	V/V	AVX512VL AVX512DQ	Convert four packed single precision floating-point values from xmm2/m128/m32bcst to four packed signed quadword values in ymm1 subject to writemask k1.
EVEX.512.66.0F.W0 7B /r VCVTQPS2Q zmm1 {k1}{z}, ymm2/m256/m32bcst{er}	HV	V/V	AVX512DQ	Convert eight packed single precision floating-point values from ymm2/m256/m32bcst to eight packed signed quadword values in zmm1 subject to writemask k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
HV	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Converts eight packed single-precision floating-point values in the source operand to eight signed quadword integers in the destination operand.

When a conversion is inexact, 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, the indefinite integer value ( $2^{w-1}$ , where  $w$  represents the number of bits in the destination format) is returned.

The source operand is a YMM/XMM/XMM (low 64-bit) register or a 256/128/64-bit memory location. The destination operation is a ZMM/YMM/XMM register conditionally updated with writemask  $k1$ .

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

### Operation

**VCVTQPS2Q (EVEX encoded versions) when src operand is a register**

(KL, VL) = (2, 128), (4, 256), (8, 512)

IF (VL == 512) AND (EVEX.b == 1)

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR  $j \leftarrow 0$  TO  $KL-1$

$i \leftarrow j * 64$

$k \leftarrow j * 32$

IF  $k1[j]$  OR \*no writemask\*

THEN  $DEST[i+63:i] \leftarrow$

Convert\_Single\_Precision\_To\_QuadInteger(SRC[k+31:k])

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE ; zeroing-masking

```

                DEST[i+63:i] ← 0
            FI
        FI;
    ENDFOR
    DEST[MAX_VL-1:VL] ← 0

```

#### VCVTPS2QQ (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_Single_Precision_To_QuadInteger(SRC[31:0])
                ELSE
                    DEST[i+63:i] ←
                    Convert_Single_Precision_To_QuadInteger(SRC[k+31:k])
                FI;
            ELSE
                IF *merging-masking*           ; merging-masking
                    THEN *DEST[i+63:i] remains unchanged*
                ELSE                             ; zeroing-masking
                    DEST[i+63:i] ← 0
                FI
            FI;
        ENDFOR
    DEST[MAX_VL-1:VL] ← 0

```

#### Intel C/C++ Compiler Intrinsic Equivalent

```

VCVTPS2QQ __m512i __mm512_cvtps_epi64( __m512 a);
VCVTPS2QQ __m512i __mm512_mask_cvtps_epi64( __m512i s, __mmask16 k, __m512 a);
VCVTPS2QQ __m512i __mm512_maskz_cvtps_epi64( __mmask16 k, __m512 a);
VCVTPS2QQ __m512i __mm512_cvt_roundps_epi64( __m512 a, int r);
VCVTPS2QQ __m512i __mm512_mask_cvt_roundps_epi64( __m512i s, __mmask16 k, __m512 a, int r);
VCVTPS2QQ __m512i __mm512_maskz_cvt_roundps_epi64( __mmask16 k, __m512 a, int r);
VCVTPS2QQ __m256i __mm256_cvtps_epi64( __m256 a);
VCVTPS2QQ __m256i __mm256_mask_cvtps_epi64( __m256i s, __mmask8 k, __m256 a);
VCVTPS2QQ __m256i __mm256_maskz_cvtps_epi64( __mmask8 k, __m256 a);
VCVTPS2QQ __m128i __mm_cvtps_epi64( __m128 a);
VCVTPS2QQ __m128i __mm_mask_cvtps_epi64( __m128i s, __mmask8 k, __m128 a);
VCVTPS2QQ __m128i __mm_maskz_cvtps_epi64( __mmask8 k, __m128 a);

```

#### SIMD Floating-Point Exceptions

Invalid, Precision

#### Other Exceptions

EVEX-encoded instructions, see Exceptions Type E3

#UD If EVEX.vvvv != 1111B.

## VCVTPS2UQQ—Convert Packed Single Precision Floating-Point Values to Packed Unsigned Quadword Integer Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F.W0 79 /r VCVTPS2UQQ xmm1 {k1}{z}, xmm2/m64/m32bcst	HV	V/V	AVX512VL AVX512DQ	Convert two packed single precision floating-point values from zmm2/m64/m32bcst to two packed unsigned quadword values in zmm1 subject to writemask k1.
EVEX.256.66.0F.W0 79 /r VCVTPS2UQQ ymm1 {k1}{z}, xmm2/m128/m32bcst	HV	V/V	AVX512VL AVX512DQ	Convert four packed single precision floating-point values from xmm2/m128/m32bcst to four packed unsigned quadword values in ymm1 subject to writemask k1.
EVEX.512.66.0F.W0 79 /r VCVTPS2UQQ zmm1 {k1}{z}, ymm2/m256/m32bcst{er}	HV	V/V	AVX512DQ	Convert eight packed single precision floating-point values from ymm2/m256/m32bcst to eight packed unsigned quadword values in zmm1 subject to writemask k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
HV	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Converts up to eight packed single-precision floating-point values in the source operand to unsigned quadword integers in the destination operand.

When a conversion is inexact, 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, the integer value  $2^w - 1$  is returned, where  $w$  represents the number of bits in the destination format.

The source operand is a YMM/XMM/XMM (low 64-bit) register or a 256/128/64-bit memory location. The destination operation is a ZMM/YMM/XMM register conditionally updated with writemask k1.

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

### Operation

#### VCVTPS2UQQ (EVEX encoded versions) when src operand is a register

(KL, VL) = (2, 128), (4, 256), (8, 512)

IF (VL == 512) AND (EVEX.b == 1)

```

THEN
    SET_RM(EVEX.RC);
ELSE
    SET_RM(MXCSR.RM);
FI;
FOR j ← 0 TO KL-1
    i ← j * 64
    k ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ←
            Convert_Single_Precision_To_UQuadInteger(SRC[k+31:k])
        ELSE
            IF *merging-masking*           ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
            ELSE                             ; zeroing-masking

```

```

                DEST[i+63:i] ← 0
            FI
        FI;
    ENDFOR
    DEST[MAX_VL-1:VL] ← 0

VCVTPS2UQQ (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_Single_Precision_To_UQuadInteger(SRC[31:0])
                    ELSE
                        DEST[i+63:i] ←
                        Convert_Single_Precision_To_UQuadInteger(SRC[k+31:k])
                    FI;
                ELSE
                    IF *merging-masking* ; merging-masking
                        THEN *DEST[i+63:i] remains unchanged*
                    ELSE ; zeroing-masking
                        DEST[i+63:i] ← 0
                    FI
                FI
            FI;
        ENDFOR
        DEST[MAX_VL-1:VL] ← 0

```

#### Intel C/C++ Compiler Intrinsic Equivalent

```

VCVTPS2UQQ __m512i __mm512_cvtps_epu64( __m512 a);
VCVTPS2UQQ __m512i __mm512_mask_cvtps_epu64( __m512i s, __mmask16 k, __m512 a);
VCVTPS2UQQ __m512i __mm512_maskz_cvtps_epu64( __mmask16 k, __m512 a);
VCVTPS2UQQ __m512i __mm512_cvt_roundps_epu64( __m512 a, int r);
VCVTPS2UQQ __m512i __mm512_mask_cvt_roundps_epu64( __m512i s, __mmask16 k, __m512 a, int r);
VCVTPS2UQQ __m512i __mm512_maskz_cvt_roundps_epu64( __mmask16 k, __m512 a, int r);
VCVTPS2UQQ __m256i __mm256_cvtps_epu64( __m256 a);
VCVTPS2UQQ __m256i __mm256_mask_cvtps_epu64( __m256i s, __mmask8 k, __m256 a);
VCVTPS2UQQ __m256i __mm256_maskz_cvtps_epu64( __mmask8 k, __m256 a);
VCVTPS2UQQ __m128i __mm_cvtps_epu64( __m128 a);
VCVTPS2UQQ __m128i __mm_mask_cvtps_epu64( __m128i s, __mmask8 k, __m128 a);
VCVTPS2UQQ __m128i __mm_maskz_cvtps_epu64( __mmask8 k, __m128 a);

```

#### SIMD Floating-Point Exceptions

Invalid, Precision

#### Other Exceptions

EVEX-encoded instructions, see Exceptions Type E3

#UD If EVEX.vvvv != 1111B.

## VCVTQQ2PD—Convert Packed Quadword Integers to Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.F3.0F.W1 E6 /r VCVTQQ2PD xmm1 {k1}{z}, xmm2/m128/m64bcst	FV	V/V	AVX512VL AVX512DQ	Convert two packed quadword integers from xmm2/m128/m64bcst to packed double-precision floating-point values in xmm1 with writemask k1.
EVEX.256.F3.0F.W1 E6 /r VCVTQQ2PD ymm1 {k1}{z}, ymm2/m256/m64bcst	FV	V/V	AVX512VL AVX512DQ	Convert four packed quadword integers from ymm2/m256/m64bcst to packed double-precision floating-point values in ymm1 with writemask k1.
EVEX.512.F3.0F.W1 E6 /r VCVTQQ2PD zmm1 {k1}{z}, zmm2/m512/m64bcst{er}	FV	V/V	AVX512DQ	Convert eight packed quadword integers from zmm2/m512/m64bcst to eight packed double-precision floating-point values in zmm1 with writemask k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
FV	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Converts packed quadword integers in the source operand (second operand) to packed double-precision floating-point values in the destination operand (first operand).

The source operand is a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operation is a ZMM/YMM/XMM register conditionally updated with writemask k1.

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

### Operation

#### VCVTQQ2PD (EVEX2 encoded versions) when src operand is a register

(KL, VL) = (2, 128), (4, 256), (8, 512)

IF (VL == 512) AND (EVEX.b == 1)

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN DEST[i+63:i] ←

Convert\_QuadInteger\_To\_Double\_Precision\_Floating\_Point(SRC[i+63:i])

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

#### VCVTQQ2PD (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

  IF k1[j] OR \*no writemask\*

    THEN

      IF (EVEX.b == 1)

        THEN

          DEST[i+63:i] ←

          Convert\_QuadInteger\_To\_Double\_Precision\_Floating\_Point(SRC[63:0])

        ELSE

          DEST[i+63:i] ←

          Convert\_QuadInteger\_To\_Double\_Precision\_Floating\_Point(SRC[i+63:i])

        FI;

    ELSE

      IF \*merging-masking\* ; merging-masking

        THEN \*DEST[i+63:i] remains unchanged\*

      ELSE ; zeroing-masking

        DEST[i+63:i] ← 0

    FI

  FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

#### Intel C/C++ Compiler Intrinsic Equivalent

VCVTQQ2PD \_\_m512d \_\_mm512\_cvtepi64\_pd( \_\_m512i a);

VCVTQQ2PD \_\_m512d \_\_mm512\_mask\_cvtepi64\_pd( \_\_m512d s, \_\_mmask16 k, \_\_m512i a);

VCVTQQ2PD \_\_m512d \_\_mm512\_maskz\_cvtepi64\_pd( \_\_mmask16 k, \_\_m512i a);

VCVTQQ2PD \_\_m512d \_\_mm512\_cvt\_roundepi64\_pd( \_\_m512i a, int r);

VCVTQQ2PD \_\_m512d \_\_mm512\_mask\_cvt\_roundepi\_ps( \_\_m512d s, \_\_mmask8 k, \_\_m512i a, int r);

VCVTQQ2PD \_\_m512d \_\_mm512\_maskz\_cvt\_roundepi64\_pd( \_\_mmask8 k, \_\_m512i a, int r);

VCVTQQ2PD \_\_m256d \_\_mm256\_mask\_cvtepi64\_pd( \_\_m256d s, \_\_mmask8 k, \_\_m256i a);

VCVTQQ2PD \_\_m256d \_\_mm256\_maskz\_cvtepi64\_pd( \_\_mmask8 k, \_\_m256i a);

VCVTQQ2PD \_\_m128d \_\_mm\_mask\_cvtepi64\_pd( \_\_m128d s, \_\_mmask8 k, \_\_m128i a);

VCVTQQ2PD \_\_m128d \_\_mm\_maskz\_cvtepi64\_pd( \_\_mmask8 k, \_\_m128i a);

#### SIMD Floating-Point Exceptions

Precision

#### Other Exceptions

EVEX-encoded instructions, see Exceptions Type E2

#UD If EVEX.vvvv != 1111B.

## VCVTQQ2PS—Convert Packed Quadword Integers to Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.0F.W1 5B /r VCVTQQ2PS xmm1 {k1}{z}, xmm2/m128/m64bcst	FV	V/V	AVX512VL AVX512DQ	Convert two packed quadword integers from xmm2/mem to packed single-precision floating-point values in xmm1 with writemask k1.
EVEX.256.0F.W1 5B /r VCVTQQ2PS xmm1 {k1}{z}, ymm2/m256/m64bcst	FV	V/V	AVX512VL AVX512DQ	Convert four packed quadword integers from ymm2/mem to packed single-precision floating-point values in xmm1 with writemask k1.
EVEX.512.0F.W1 5B /r VCVTQQ2PS ymm1 {k1}{z}, zmm2/m512/m64bcst{er}	FV	V/V	AVX512DQ	Convert eight packed quadword integers from zmm2/mem to eight packed single-precision floating-point values in ymm1 with writemask k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
FV	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Converts packed quadword integers in the source operand (second operand) to packed single-precision floating-point values in the destination operand (first operand).

The source operand is a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operation is a YMM/XMM/XMM (lower 64 bits) register conditionally updated with writemask k1.

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

### Operation

**VCVTQQ2PS (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[k+31:k] ←
      Convert_QuadInteger_To_Single_Precision_Floating_Point(SRC[i+63:i])
  ELSE
    IF *merging-masking*           ; merging-masking
      THEN *DEST[k+31:k] remains unchanged*
    ELSE                             ; zeroing-masking
      DEST[k+31:k] ← 0
  FI
FI;
ENDFOR
DEST[MAX_VL-1:VL/2] ← 0

```



**VCVTQQ2PS (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[k+31:k] ←
          Convert_QuadInteger_To_Single_Precision_Floating_Point(SRC[63:0])
        ELSE
          DEST[k+31:k] ←
          Convert_QuadInteger_To_Single_Precision_Floating_Point(SRC[i+63:i])
        FI;
      ELSE
        IF *merging-masking*           ; merging-masking
          THEN *DEST[k+31:k] remains unchanged*
        ELSE                             ; zeroing-masking
          DEST[k+31:k] ← 0
        FI
      FI;
    ENDFOR
  DEST[MAX_VL-1:VL/2] ← 0
  
```

#### Intel C/C++ Compiler Intrinsic Equivalent

```

VCVTQQ2PS __m256 __mm512_cvtepi64_ps( __m512i a);
VCVTQQ2PS __m256 __mm512_mask_cvtepi64_ps( __m256 s, __mmask16 k, __m512i a);
VCVTQQ2PS __m256 __mm512_maskz_cvtepi64_ps( __mmask16 k, __m512i a);
VCVTQQ2PS __m256 __mm512_cvt_roundepi64_ps( __m512i a, int r);
VCVTQQ2PS __m256 __mm512_mask_cvt_roundepi64_ps( __m256 s, __mmask8 k, __m512i a, int r);
VCVTQQ2PS __m256 __mm512_maskz_cvt_roundepi64_ps( __mmask8 k, __m512i a, int r);
VCVTQQ2PS __m128 __mm256_cvtepi64_ps( __m256i a);
VCVTQQ2PS __m128 __mm256_mask_cvtepi64_ps( __m128 s, __mmask8 k, __m256i a);
VCVTQQ2PS __m128 __mm256_maskz_cvtepi64_ps( __mmask8 k, __m256i a);
VCVTQQ2PS __m128 __mm_cvtepi64_ps( __m128i a);
VCVTQQ2PS __m128 __mm_mask_cvtepi64_ps( __m128 s, __mmask8 k, __m128i a);
VCVTQQ2PS __m128 __mm_maskz_cvtepi64_ps( __mmask8 k, __m128i a);
  
```

#### SIMD Floating-Point Exceptions

Precision

#### Other Exceptions

EVEX-encoded instructions, see Exceptions Type E2

#UD If EVEX.vvvv != 1111B.

## VCVTSD2USI—Convert Scalar Double-Precision Floating-Point Value to Unsigned Doubleword Integer

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LIG.F2.OF.W0 79 /r VCVTSD2USI r32, xmm1/m64{er}	T1F	V/V	AVX512F	Convert one double-precision floating-point value from xmm1/m64 to one unsigned doubleword integer r32.
EVEX.LIG.F2.OF.W1 79 /r VCVTSD2USI r64, xmm1/m64{er}	T1F	V/N.E. <sup>1</sup>	AVX512F	Convert one double-precision floating-point value from xmm1/m64 to one unsigned quadword integer zero-extended into r64.

### NOTES:

1. EVEX.W1 in non-64 bit is ignored; the instructions behaves as if the W0 version is used.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
T1F	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Converts a double-precision floating-point value in the source operand (the second operand) to an unsigned doubleword integer in the destination operand (the first operand). 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 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, the integer value  $2^w - 1$  is returned, where  $w$  represents the number of bits in the destination format.

### Operation

#### VCVTSD2USI (EVEX encoded version)

IF (SRC \*is register\*) AND (EVEX.b = 1)

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

IF 64-Bit Mode and OperandSize = 64

THEN DEST[63:0] ← Convert\_Double\_Precision\_Floating\_Point\_To\_UInteger(SRC[63:0]);

ELSE DEST[31:0] ← Convert\_Double\_Precision\_Floating\_Point\_To\_UInteger(SRC[63:0]);

FI

### Intel C/C++ Compiler Intrinsic Equivalent

VCVTSD2USI unsigned int \_\_mm\_cvtsd\_u32(\_\_m128d);

VCVTSD2USI unsigned int \_\_mm\_cvt\_roundsd\_u32(\_\_m128d, int r);

VCVTSD2USI unsigned \_\_int64 \_\_mm\_cvtsd\_u64(\_\_m128d);

VCVTSD2USI unsigned \_\_int64 \_\_mm\_cvt\_roundsd\_u64(\_\_m128d, int r);

### SIMD Floating-Point Exceptions

Invalid, Precision

 **Other Exceptions**

EVEX-encoded instructions, see Exceptions Type E3NF.

## VCVTSS2USI—Convert Scalar Single-Precision Floating-Point Value to Unsigned Doubleword Integer

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LIG.F3.OF.W0 79 /r VCVTSS2USI r32, xmm1/m32{er}	T1F	V/V	AVX512F	Convert one single-precision floating-point value from xmm1/m32 to one unsigned doubleword integer in r32.
EVEX.LIG.F3.OF.W1 79 /r VCVTSS2USI r64, xmm1/m32{er}	T1F	V/N.E. <sup>1</sup>	AVX512F	Convert one single-precision floating-point value from xmm1/m32 to one unsigned quadword integer in r64.

### NOTES:

1. EVEX.W1 in non-64 bit is ignored; the instructions behaves as if the W0 version is used.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
T1F	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Converts a single-precision floating-point value in the source operand (the second operand) to an unsigned doubleword integer (or unsigned quadword integer if operand size is 64 bits) in the destination operand (the first operand). 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 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, the integer value  $2^w - 1$  is returned, where  $w$  represents the number of bits in the destination format.

VEX.W1 and EVEX.W1 versions: 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.

### Operation

#### VCVTSS2USI (EVEX encoded version)

IF (SRC \*is register\*) AND (EVEX.b = 1)

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

IF 64-bit Mode and OperandSize = 64

THEN

DEST[63:0] ← Convert\_Single\_Precision\_Floating\_Point\_To\_UInteger(SRC[31:0]);

ELSE

DEST[31:0] ← Convert\_Single\_Precision\_Floating\_Point\_To\_UInteger(SRC[31:0]);

FI;

### Intel C/C++ Compiler Intrinsic Equivalent

VCVTSS2USI unsigned \_\_mm\_cvtss\_u32( \_\_m128 a);

VCVTSS2USI unsigned \_\_mm\_cvt\_roundss\_u32( \_\_m128 a, int r);

VCVTSS2USI unsigned \_\_int64 \_\_mm\_cvtss\_u64( \_\_m128 a);

VCVTSS2USI unsigned \_\_int64 \_\_mm\_cvt\_roundss\_u64( \_\_m128 a, int r);



## SIMD Floating-Point Exceptions

Invalid, Precision

## Other Exceptions

EVEX-encoded instructions, see Exceptions Type E3NF.

## VCVTTPD2QQ—Convert with Truncation Packed Double-Precision Floating-Point Values to Packed Quadword Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F.W1 7A /r VCVTTPD2QQ xmm1 {k1}{z}, xmm2/m128/m64bcst	FV	V/V	AVX512VL AVX512DQ	Convert two packed double-precision floating-point values from zmm2/m128/m64bcst to two packed quadword integers in zmm1 using truncation with writemask k1.
EVEX.256.66.0F.W1 7A /r VCVTTPD2QQ ymm1 {k1}{z}, ymm2/m256/m64bcst	FV	V/V	AVX512VL AVX512DQ	Convert four packed double-precision floating-point values from ymm2/m256/m64bcst to four packed quadword integers in ymm1 using truncation with writemask k1.
EVEX.512.66.0F.W1 7A /r VCVTTPD2QQ zmm1 {k1}{z}, zmm2/m512/m64bcst{sae}	FV	V/V	AVX512DQ	Convert eight packed double-precision floating-point values from zmm2/m512 to eight packed quadword integers in zmm1 using truncation with writemask k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
FV	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Converts with truncation packed double-precision floating-point values in the source operand (second operand) to packed quadword integers in the destination operand (first operand).

EVEX encoded versions: 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 value returned is rounded according to the rounding control bits in the MXCSR register. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value ( $2^{w-1}$ , where w represents the number of bits in the destination format) is returned.

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

### Operation

**VCVTTPD2QQ (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\_Double\_Precision\_Floating\_Point\_To\_QuadInteger\_Truncate(SRC[i+63:i])

  ELSE

    IF \*merging-masking\* ; merging-masking

      THEN \*DEST[i+63:i] remains unchanged\*

    ELSE ; zeroing-masking

      DEST[i+63:i] ← 0

  FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VCVTTPD2QQ (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_Double_Precision_Floating_Point_To_QuadInteger_Truncate(SRC[63:0])
        ELSE
          DEST[i+63:i] ← Convert_Double_Precision_Floating_Point_To_QuadInteger_Truncate(SRC[i+63:i])
        FI;
      ELSE
        IF *merging-masking* ; merging-masking
          THEN *DEST[i+63:i] remains unchanged*
        ELSE ; zeroing-masking
          DEST[i+63:i] ← 0
        FI
      FI;
    ENDFOR
  DEST[MAX_VL-1:VL] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VCVTTPD2QQ __m512i __mm512_cvttpd_epi64( __m512d a);
VCVTTPD2QQ __m512i __mm512_mask_cvttpd_epi64( __m512i s, __mmask8 k, __m512d a);
VCVTTPD2QQ __m512i __mm512_maskz_cvttpd_epi64( __mmask8 k, __m512d a);
VCVTTPD2QQ __m512i __mm512_cvtt_roundpd_epi64( __m512d a, int sae);
VCVTTPD2QQ __m512i __mm512_mask_cvtt_roundpd_epi64( __m512i s, __mmask8 k, __m512d a, int sae);
VCVTTPD2QQ __m512i __mm512_maskz_cvtt_roundpd_epi64( __mmask8 k, __m512d a, int sae);
VCVTTPD2QQ __m256i __mm256_mask_cvttpd_epi64( __m256i s, __mmask8 k, __m256d a);
VCVTTPD2QQ __m256i __mm256_maskz_cvttpd_epi64( __mmask8 k, __m256d a);
VCVTTPD2QQ __m128i __mm_mask_cvttpd_epi64( __m128i s, __mmask8 k, __m128d a);
VCVTTPD2QQ __m128i __mm_maskz_cvttpd_epi64( __mmask8 k, __m128d a);

```

**SIMD Floating-Point Exceptions**

Invalid, Precision

**Other Exceptions**

EVEX-encoded instructions, see Exceptions Type E2.

#UD If EVEX.vvvv != 1111B.

## VCVTTPD2UDQ—Convert with Truncation Packed Double-Precision Floating-Point Values to Packed Unsigned Doubleword Integers

Opcode Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.OF.W1 78 /r VCVTTPD2UDQ xmm1 {k1}{z}, xmm2/m128/m64bcst	FV	V/V	AVX512VL AVX512F	Convert two packed double-precision floating-point values in xmm2/m128/m64bcst to two unsigned doubleword integers in xmm1 using truncation subject to writemask k1.
EVEX.256.OF.W1 78 02 /r VCVTTPD2UDQ xmm1 {k1}{z}, ymm2/m256/m64bcst	FV	V/V	AVX512VL AVX512F	Convert four packed double-precision floating-point values in ymm2/m256/m64bcst to four unsigned doubleword integers in xmm1 using truncation subject to writemask k1.
EVEX.512.OF.W1 78 /r VCVTTPD2UDQ ymm1 {k1}{z}, zmm2/m512/m64bcst{sae}	FV	V/V	AVX512F	Convert eight packed double-precision floating-point values in zmm2/m512/m64bcst to eight unsigned doubleword integers in ymm1 using truncation subject to writemask k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
FV	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Converts with truncation packed double-precision floating-point values in the source operand (the second operand) to packed unsigned doubleword integers in the destination operand (the first operand).

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the integer value  $2^w - 1$  is returned, where  $w$  represents the number of bits in the destination format.

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 64-bit memory location. The destination operand is a YMM/XMM/XMM (low 64 bits) register conditionally updated with writemask  $k1$ . The upper bits (MAX\_VL-1:256) of the corresponding destination are zeroed.

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

### Operation

**VCVTTPD2UDQ (EVEX encoded versions) when src2 operand is a register**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR  $j \leftarrow 0$  TO  $KL-1$

$i \leftarrow j * 32$

$k \leftarrow j * 64$

  IF  $k1[j]$  OR \*no writemask\*

    THEN

      DEST[ $i+31:i$ ] ←

      Convert\_Double\_Precision\_Floating\_Point\_To\_UInteger\_Truncate(SRC[ $k+63:k$ ])

    ELSE

      IF \*merging-masking\* ; merging-masking

        THEN \*DEST[ $i+31:i$ ] remains unchanged\*

        ELSE ; zeroing-masking

          DEST[ $i+31:i$ ] ← 0

    FI

FI;



```

ENDFOR
DEST[MAX_VL-1:VL/2] ← 0

```

#### VCVTTPD2UDQ (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_Double_Precision_Floating_Point_To_UInteger_Truncate(SRC[63:0])
        ELSE
          DEST[i+31:i] ←
          Convert_Double_Precision_Floating_Point_To_UInteger_Truncate(SRC[k+63:k])
        FI;
      ELSE
        IF *merging-masking* ; merging-masking
          THEN *DEST[i+31:i] remains unchanged*
        ELSE ; zeroing-masking
          DEST[i+31:i] ← 0
        FI
      FI;
ENDFOR
DEST[MAX_VL-1:VL/2] ← 0

```

#### Intel C/C++ Compiler Intrinsic Equivalent

```

VCVTTPD2UDQ __m256i _mm512_cvttpd_epu32( __m512d a);
VCVTTPD2UDQ __m256i _mm512_mask_cvttpd_epu32( __m256i s, __mmask8 k, __m512d a);
VCVTTPD2UDQ __m256i _mm512_maskz_cvttpd_epu32( __mmask8 k, __m512d a);
VCVTTPD2UDQ __m256i _mm512_cvtt_roundpd_epu32( __m512d a, int sae);
VCVTTPD2UDQ __m256i _mm512_mask_cvtt_roundpd_epu32( __m256i s, __mmask8 k, __m512d a, int sae);
VCVTTPD2UDQ __m256i _mm512_maskz_cvtt_roundpd_epu32( __mmask8 k, __m512d a, int sae);
VCVTTPD2UDQ __m128i _mm256_mask_cvttpd_epu32( __m128i s, __mmask8 k, __m256d a);
VCVTTPD2UDQ __m128i _mm256_maskz_cvttpd_epu32( __mmask8 k, __m256d a);
VCVTTPD2UDQ __m128i _mm_mask_cvttpd_epu32( __m128i s, __mmask8 k, __m128d a);
VCVTTPD2UDQ __m128i _mm_maskz_cvttpd_epu32( __mmask8 k, __m128d a);

```

#### SIMD Floating-Point Exceptions

Invalid, Precision

#### Other Exceptions

EVEX-encoded instructions, see Exceptions Type E2.

#UD If EVEX.vvvv != 1111B.

## VCVTTPD2UQQ—Convert with Truncation Packed Double-Precision Floating-Point Values to Packed Unsigned Quadword Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F.W1 78 /r VCVTTPD2UQQ xmm1 {k1}{z}, xmm2/m128/m64bcst	FV	V/V	AVX512VL AVX512DQ	Convert two packed double-precision floating-point values from xmm2/m128/m64bcst to two packed unsigned quadword integers in xmm1 using truncation with writemask k1.
EVEX.256.66.0F.W1 78 /r VCVTTPD2UQQ ymm1 {k1}{z}, ymm2/m256/m64bcst	FV	V/V	AVX512VL AVX512DQ	Convert four packed double-precision floating-point values from ymm2/m256/m64bcst to four packed unsigned quadword integers in ymm1 using truncation with writemask k1.
EVEX.512.66.0F.W1 78 /r VCVTTPD2UQQ zmm1 {k1}{z}, zmm2/m512/m64bcst{sae}	FV	V/V	AVX512DQ	Convert eight packed double-precision floating-point values from zmm2/mem to eight packed unsigned quadword integers in zmm1 using truncation with writemask k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
FV	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Converts with truncation packed double-precision floating-point values in the source operand (second operand) to packed unsigned quadword integers in the destination operand (first operand).

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the integer value  $2^w - 1$  is returned, where  $w$  represents the number of bits in the destination format.

EVEX encoded versions: The source operand is a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operation is a ZMM/YMM/XMM register conditionally updated with writemask k1.

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

### Operation

**VCVTTPD2UQQ (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

  IF k1[j] OR \*no writemask\*

    THEN DEST[i+63:i] ←

      Convert\_Double\_Precision\_Floating\_Point\_To\_UQuadInteger\_Truncate(SRC[i+63:i])

  ELSE

    IF \*merging-masking\* ; merging-masking

      THEN \*DEST[i+63:i] remains unchanged\*

    ELSE ; zeroing-masking

      DEST[i+63:i] ← 0

  FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VCVTTPD2UQQ (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
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b == 1)
        THEN
          DEST[i+63:i] ←
          Convert_Double_Precision_Floating_Point_To_UQuadInteger_Truncate(SRC[63:0])
        ELSE
          DEST[i+63:i] ←
          Convert_Double_Precision_Floating_Point_To_UQuadInteger_Truncate(SRC[i+63:i])
      FI;
    ELSE
      IF *merging-masking*           ; merging-masking
        THEN *DEST[i+63:i] remains unchanged*
      ELSE                             ; zeroing-masking
        DEST[i+63:i] ← 0
      FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VCVTTPD2UQQ __mm<size>[_mask[z]]_cvtt[_round]pd_epu64
VCVTTPD2UQQ __m512i __mm512_cvttpd_epu64( __m512d a);
VCVTTPD2UQQ __m512i __mm512_mask_cvttpd_epu64( __m512i s, __mmask8 k, __m512d a);
VCVTTPD2UQQ __m512i __mm512_maskz_cvttpd_epu64( __mmask8 k, __m512d a);
VCVTTPD2UQQ __m512i __mm512_cvtt_roundpd_epu64( __m512d a, int sae);
VCVTTPD2UQQ __m512i __mm512_mask_cvtt_roundpd_epu64( __m512i s, __mmask8 k, __m512d a, int sae);
VCVTTPD2UQQ __m512i __mm512_maskz_cvtt_roundpd_epu64( __mmask8 k, __m512d a, int sae);
VCVTTPD2UQQ __m256i __mm256_mask_cvttpd_epu64( __m256i s, __mmask8 k, __m256d a);
VCVTTPD2UQQ __m256i __mm256_maskz_cvttpd_epu64( __mmask8 k, __m256d a);
VCVTTPD2UQQ __m128i __mm_mask_cvttpd_epu64( __m128i s, __mmask8 k, __m128d a);
VCVTTPD2UQQ __m128i __mm_maskz_cvttpd_epu64( __mmask8 k, __m128d a);

```

**SIMD Floating-Point Exceptions**

Invalid, Precision

**Other Exceptions**

EVEX-encoded instructions, see Exceptions Type E2.

#UD If EVEX.vvvv != 1111B.

## VCVTTPS2UDQ—Convert with Truncation Packed Single-Precision Floating-Point Values to Packed Unsigned Doubleword Integer Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.OF.W0 78 /r VCVTTPS2UDQ xmm1 {k1}{z}, xmm2/m128/m32bcst	FV	V/V	AVX512VL AVX512F	Convert four packed single precision floating-point values from xmm2/m128/m32bcst to four packed unsigned doubleword values in xmm1 using truncation subject to writemask k1.
EVEX.256.OF.W0 78 /r VCVTTPS2UDQ ymm1 {k1}{z}, ymm2/m256/m32bcst	FV	V/V	AVX512VL AVX512F	Convert eight packed single precision floating-point values from ymm2/m256/m32bcst to eight packed unsigned doubleword values in ymm1 using truncation subject to writemask k1.
EVEX.512.OF.W0 78 /r VCVTTPS2UDQ zmm1 {k1}{z}, zmm2/m512/m32bcst{sae}	FV	V/V	AVX512F	Convert sixteen packed single-precision floating-point values from zmm2/m512/m32bcst to sixteen packed unsigned doubleword values in zmm1 using truncation subject to writemask k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
FV	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Converts with truncation packed single-precision floating-point values in the source operand to sixteen unsigned doubleword integers in the destination operand.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the integer value  $2^w - 1$  is returned, where  $w$  represents the number of bits in the destination format.

EVEX encoded versions: 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 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

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

### Operation

**VCVTTPS2UDQ (EVEX encoded versions) when src operand is a register**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR  $j \leftarrow 0$  TO  $KL-1$

$i \leftarrow j * 32$

  IF  $k1[j]$  OR \*no writemask\*

    THEN  $DEST[i+31:i] \leftarrow$

      Convert\_Single\_Precision\_Floating\_Point\_To\_Integer\_Truncate(SRC[i+31:i])

  ELSE

    IF \*merging-masking\* ; merging-masking

      THEN \*DEST[i+31:i] remains unchanged\*

    ELSE ; zeroing-masking

$DEST[i+31:i] \leftarrow 0$

  FI

FI;

ENDFOR

$DEST[1:MAX\_VL-1:VL] \leftarrow 0$

**VCVTTPS2UDQ (EVEX encoded versions) when src operand is a memory source**

(KL, VL) = (4, 128), (8, 256), (16, 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_Single_Precision_Floating_Point_To_UInteger_Truncate(SRC[31:0])
        ELSE
          DEST[i+31:i] ←
          Convert_Single_Precision_Floating_Point_To_UInteger_Truncate(SRC[i+31:i])
        FI;
      ELSE
        IF *merging-masking*           ; merging-masking
          THEN *DEST[i+31:i] remains unchanged*
        ELSE                             ; zeroing-masking
          DEST[i+31:i] ← 0
        FI
      FI;
    ENDFOR
  DEST[MAX_VL-1:VL] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VCVTTPS2UDQ __m512i __mm512_cvttps_epu32( __m512 a);
VCVTTPS2UDQ __m512i __mm512_mask_cvttps_epu32( __m512i s, __mmask16 k, __m512 a);
VCVTTPS2UDQ __m512i __mm512_maskz_cvttps_epu32( __mmask16 k, __m512 a);
VCVTTPS2UDQ __m512i __mm512_cvtt_roundps_epu32( __m512 a, int sae);
VCVTTPS2UDQ __m512i __mm512_mask_cvtt_roundps_epu32( __m512i s, __mmask16 k, __m512 a, int sae);
VCVTTPS2UDQ __m512i __mm512_maskz_cvtt_roundps_epu32( __mmask16 k, __m512 a, int sae);
VCVTTPS2UDQ __m256i __mm256_mask_cvttps_epu32( __m256i s, __mmask8 k, __m256 a);
VCVTTPS2UDQ __m256i __mm256_maskz_cvttps_epu32( __mmask8 k, __m256 a);
VCVTTPS2UDQ __m128i __mm_mask_cvttps_epu32( __m128i s, __mmask8 k, __m128 a);
VCVTTPS2UDQ __m128i __mm_maskz_cvttps_epu32( __mmask8 k, __m128 a);

```

**SIMD Floating-Point Exceptions**

Invalid, Precision

**Other Exceptions**

EVEX-encoded instructions, see Exceptions Type E2.

#UD If EVEX.vvvv != 1111B.

## VCVTTPS2QQ—Convert with Truncation Packed Single Precision Floating-Point Values to Packed Signed Quadword Integer Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F.W0 7A /r VCVTTPS2QQ xmm1 {k1}{z}, xmm2/m64/m32bcst	HV	V/V	AVX512VL AVX512DQ	Convert two packed single precision floating-point values from xmm2/m64/m32bcst to two packed signed quadword values in xmm1 using truncation subject to writemask k1.
EVEX.256.66.0F.W0 7A /r VCVTTPS2QQ ymm1 {k1}{z}, xmm2/m128/m32bcst	HV	V/V	AVX512VL AVX512DQ	Convert four packed single precision floating-point values from xmm2/m128/m32bcst to four packed signed quadword values in ymm1 using truncation subject to writemask k1.
EVEX.512.66.0F.W0 7A /r VCVTTPS2QQ zmm1 {k1}{z}, ymm2/m256/m32bcst{sae}	HV	V/V	AVX512DQ	Convert eight packed single precision floating-point values from ymm2/m256/m32bcst to eight packed signed quadword values in zmm1 using truncation subject to writemask k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
HV	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Converts with truncation packed single-precision floating-point values in the source operand to eight signed quadword integers in the destination operand.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value ( $2^{w-1}$ , where  $w$  represents the number of bits in the destination format) is returned.

EVEX encoded versions: The source operand is a YMM/XMM/XMM (low 64 bits) register or a 256/128/64-bit memory location. The destination operation is a vector register conditionally updated with writemask k1.

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

### Operation

**VCVTTPS2QQ (EVEX encoded versions) when src operand is a register**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR  $j \leftarrow 0$  TO  $KL-1$

$i \leftarrow j * 64$

$k \leftarrow j * 32$

  IF  $k1[j]$  OR \*no writemask\*

    THEN  $DEST[i+63:i] \leftarrow$

      Convert\_Single\_Precision\_To\_QuadInteger\_Truncate(SRC[k+31:k])

  ELSE

    IF \*merging-masking\* ; merging-masking

      THEN \*DEST[i+63:i] remains unchanged\*

    ELSE ; zeroing-masking

$DEST[i+63:i] \leftarrow 0$

  FI

FI;

ENDFOR

$DEST[MAX\_VL-1:VL] \leftarrow 0$

**VCVTTPS2QQ (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_Single_Precision_To_QuadInteger_Truncate(SRC[31:0])
        ELSE
          DEST[i+63:i] ←
          Convert_Single_Precision_To_QuadInteger_Truncate(SRC[k+31:k])
      FI;
    ELSE
      IF *merging-masking*           ; merging-masking
        THEN *DEST[i+63:i] remains unchanged*
      ELSE                             ; zeroing-masking
        DEST[i+63:i] ← 0
      FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VCVTTPS2QQ __m512i __mm512_cvttps_epi64( __m256 a);
VCVTTPS2QQ __m512i __mm512_mask_cvttps_epi64( __m512i s, __mmask16 k, __m256 a);
VCVTTPS2QQ __m512i __mm512_maskz_cvttps_epi64( __mmask16 k, __m256 a);
VCVTTPS2QQ __m512i __mm512_cvtt_roundps_epi64( __m256 a, int sae);
VCVTTPS2QQ __m512i __mm512_mask_cvtt_roundps_epi64( __m512i s, __mmask16 k, __m256 a, int sae);
VCVTTPS2QQ __m512i __mm512_maskz_cvtt_roundps_epi64( __mmask16 k, __m256 a, int sae);
VCVTTPS2QQ __m256i __mm256_mask_cvttps_epi64( __m256i s, __mmask8 k, __m128 a);
VCVTTPS2QQ __m256i __mm256_maskz_cvttps_epi64( __mmask8 k, __m128 a);
VCVTTPS2QQ __m128i __mm_mask_cvttps_epi64( __m128i s, __mmask8 k, __m128 a);
VCVTTPS2QQ __m128i __mm_maskz_cvttps_epi64( __mmask8 k, __m128 a);

```

**SIMD Floating-Point Exceptions**

Invalid, Precision

**Other Exceptions**

EVEX-encoded instructions, see Exceptions Type E3.

#UD If EVEX.vvvv != 1111B.

## VCVTTPS2UQQ—Convert with Truncation Packed Single Precision Floating-Point Values to Packed Unsigned Quadword Integer Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F.W0 78 /r VCVTTPS2UQQ xmm1 {k1}{z}, xmm2/m64/m32bcst	HV	V/V	AVX512VL AVX512DQ	Convert two packed single precision floating-point values from xmm2/m64/m32bcst to two packed unsigned quadword values in xmm1 using truncation subject to writemask k1.
EVEX.256.66.0F.W0 78 /r VCVTTPS2UQQ ymm1 {k1}{z}, xmm2/m128/m32bcst	HV	V/V	AVX512VL AVX512DQ	Convert four packed single precision floating-point values from xmm2/m128/m32bcst to four packed unsigned quadword values in ymm1 using truncation subject to writemask k1.
EVEX.512.66.0F.W0 78 /r VCVTTPS2UQQ zmm1 {k1}{z}, ymm2/m256/m32bcst{sae}	HV	V/V	AVX512DQ	Convert eight packed single precision floating-point values from ymm2/m256/m32bcst to eight packed unsigned quadword values in zmm1 using truncation subject to writemask k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
HV	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Converts with truncation up to eight packed single-precision floating-point values in the source operand to unsigned quadword integers in the destination operand.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the integer value  $2^w - 1$  is returned, where  $w$  represents the number of bits in the destination format.

EVEX encoded versions: The source operand is a YMM/XMM/XMM (low 64 bits) register or a 256/128/64-bit memory location. The destination operation is a vector register conditionally updated with writemask  $k1$ .

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

### Operation

**VCVTTPS2UQQ (EVEX encoded versions) when src operand is a register**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR  $j \leftarrow 0$  TO  $KL-1$

$i \leftarrow j * 64$

$k \leftarrow j * 32$

  IF  $k1[j]$  OR \*no writemask\*

    THEN  $DEST[i+63:i] \leftarrow$

      Convert\_Single\_Precision\_To\_UQuadInteger\_Truncate(SRC[k+31:k])

  ELSE

    IF \*merging-masking\* ; merging-masking

      THEN \*DEST[i+63:i] remains unchanged\*

    ELSE ; zeroing-masking

$DEST[i+63:i] \leftarrow 0$

  FI

FI;

ENDFOR

$DEST[MAX\_VL-1:VL] \leftarrow 0$



**VCVTTPS2UQQ (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_Single_Precision_To_UQuadInteger_Truncate(SRC[31:0])
        ELSE
          DEST[i+63:i] ←
          Convert_Single_Precision_To_UQuadInteger_Truncate(SRC[k+31:k])
      FI;
    ELSE
      IF *merging-masking*           ; merging-masking
        THEN *DEST[i+63:i] remains unchanged*
      ELSE                             ; zeroing-masking
        DEST[i+63:i] ← 0
      FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VCVTTPS2UQQ __mm<size>[_mask[z]]_cvtt[_round]ps_epu64
VCVTTPS2UQQ __m512i __mm512_cvttps_epu64( __m256 a);
VCVTTPS2UQQ __m512i __mm512_mask_cvttps_epu64( __m512i s, __mmask16 k, __m256 a);
VCVTTPS2UQQ __m512i __mm512_maskz_cvttps_epu64( __mmask16 k, __m256 a);
VCVTTPS2UQQ __m512i __mm512_cvtt_roundps_epu64( __m256 a, int sae);
VCVTTPS2UQQ __m512i __mm512_mask_cvtt_roundps_epu64( __m512i s, __mmask16 k, __m256 a, int sae);
VCVTTPS2UQQ __m512i __mm512_maskz_cvtt_roundps_epu64( __mmask16 k, __m256 a, int sae);
VCVTTPS2UQQ __m256i __mm256_mask_cvttps_epu64( __m256i s, __mmask8 k, __m128 a);
VCVTTPS2UQQ __m256i __mm256_maskz_cvttps_epu64( __mmask8 k, __m128 a);
VCVTTPS2UQQ __m128i __mm_mask_cvttps_epu64( __m128i s, __mmask8 k, __m128 a);
VCVTTPS2UQQ __m128i __mm_maskz_cvttps_epu64( __mmask8 k, __m128 a);

```

**SIMD Floating-Point Exceptions**

Invalid, Precision

**Other Exceptions**

EVEX-encoded instructions, see Exceptions Type E3.

#UD If EVEX.vvvv != 1111B.

## VCVTTSD2USI—Convert with Truncation Scalar Double-Precision Floating-Point Value to Unsigned Integer

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LIG.F2.OF.W0 78 /r VCVTTSD2USI r32, xmm1/m64{sae}	T1F	V/V	AVX512F	Convert one double-precision floating-point value from xmm1/m64 to one unsigned doubleword integer r32 using truncation.
EVEX.LIG.F2.OF.W1 78 /r VCVTTSD2USI r64, xmm1/m64{sae}	T1F	V/N.E. <sup>1</sup>	AVX512F	Convert one double-precision floating-point value from xmm1/m64 to one unsigned quadword integer zero-extended into r64 using truncation.

### NOTES:

- For this specific instruction, EVEX.W in non-64 bit is ignored; the instructions behaves as if the W0 version is used.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
T1F	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Converts with truncation a double-precision floating-point value in the source operand (the second operand) to an unsigned doubleword integer (or unsigned quadword integer if operand size is 64 bits) in the destination operand (the first operand). 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 value returned is rounded according to the rounding control bits in the MXCSR register. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the integer value  $2^w - 1$  is returned, where  $w$  represents the number of bits in the destination format.

EVEX.W1 version: promotes the instruction to produce 64-bit data in 64-bit mode.

### Operation

#### VCVTTSD2USI (EVEX encoded version)

IF 64-Bit Mode and OperandSize = 64

```
THEN  DEST[63:0] ← Convert_Double_Precision_Floating_Point_To_UInteger_Truncate(SRC[63:0]);
ELSE  DEST[31:0] ← Convert_Double_Precision_Floating_Point_To_UInteger_Truncate(SRC[63:0]);
```

FI

### Intel C/C++ Compiler Intrinsic Equivalent

```
VCVTTSD2USI unsigned int _mm_cvttssd_u32(__m128d);
VCVTTSD2USI unsigned int _mm_cvtt_roundssd_u32(__m128d, int sae);
VCVTTSD2USI unsigned __int64 _mm_cvttssd_u64(__m128d);
VCVTTSD2USI unsigned __int64 _mm_cvtt_roundssd_u64(__m128d, int sae);
```

### SIMD Floating-Point Exceptions

Invalid, Precision

### Other Exceptions

EVEX-encoded instructions, see Exceptions Type E3NF.

## VCVTTSS2USI—Convert with Truncation Scalar Single-Precision Floating-Point Value to Unsigned Integer

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LIG.F3.OF.W0 78 /r VCVTTSS2USI r32, xmm1/m32{sae}	T1F	V/V	AVX512F	Convert one single-precision floating-point value from xmm1/m32 to one unsigned doubleword integer in r32 using truncation.
EVEX.LIG.F3.OF.W1 78 /r VCVTTSS2USI r64, xmm1/m32{sae}	T1F	V/N.E. <sup>1</sup>	AVX512F	Convert one single-precision floating-point value from xmm1/m32 to one unsigned quadword integer in r64 using truncation.

### NOTES:

- For this specific instruction, EVEX.W in non-64 bit is ignored; the instructions behaves as if the W0 version is used.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
T1F	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Converts with truncation a single-precision floating-point value in the source operand (the second operand) to an unsigned doubleword integer (or unsigned quadword integer if operand size is 64 bits) in the destination operand (the first operand). 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 value returned is rounded according to the rounding control bits in the MXCSR register. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the integer value  $2^w - 1$  is returned, where  $w$  represents the number of bits in the destination format.

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.

### Operation

#### VCVTTSS2USI (EVEX encoded version)

IF 64-bit Mode and OperandSize = 64

THEN

DEST[63:0] ← Convert\_Single\_Precision\_Floating\_Point\_To\_UInteger\_Truncate(SRC[31:0]);

ELSE

DEST[31:0] ← Convert\_Single\_Precision\_Floating\_Point\_To\_UInteger\_Truncate(SRC[31:0]);

FI;

### Intel C/C++ Compiler Intrinsic Equivalent

VCVTTSS2USI unsigned int \_\_mm\_cvtss\_u32( \_\_m128 a);

VCVTTSS2USI unsigned int \_\_mm\_cvtt\_roundss\_u32( \_\_m128 a, int sae);

VCVTTSS2USI unsigned \_\_int64 \_\_mm\_cvtss\_u64( \_\_m128 a);

VCVTTSS2USI unsigned \_\_int64 \_\_mm\_cvtt\_roundss\_u64( \_\_m128 a, int sae);

### SIMD Floating-Point Exceptions

Invalid, Precision

**Other Exceptions**

EVEX-encoded instructions, see Exceptions Type E3NF.

## VCVTUDQ2PD—Convert Packed Unsigned Doubleword Integers to Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.F3.0F.W0 7A /r VCVTUDQ2PD xmm1 {k1}{z}, xmm2/m64/m32bcst	HV	V/V	AVX512VL AVX512F	Convert two packed unsigned doubleword integers from ymm2/m64/m32bcst to packed double-precision floating-point values in zmm1 with writemask k1.
EVEX.256.F3.0F.W0 7A /r VCVTUDQ2PD ymm1 {k1}{z}, xmm2/m128/m32bcst	HV	V/V	AVX512VL AVX512F	Convert four packed unsigned doubleword integers from xmm2/m128/m32bcst to packed double-precision floating-point values in zmm1 with writemask k1.
EVEX.512.F3.0F.W0 7A /r VCVTUDQ2PD zmm1 {k1}{z}, ymm2/m256/m32bcst	HV	V/V	AVX512F	Convert eight packed unsigned doubleword integers from ymm2/m256/m32bcst to eight packed double-precision floating-point values in zmm1 with writemask k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
HV	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Converts packed unsigned doubleword integers in the source operand (second operand) to packed double-precision floating-point values in the destination operand (first operand).

The source operand is a YMM/XMM/XMM (low 64 bits) register, a 256/128/64-bit memory location or a 256/128/64-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

Attempt to encode this instruction with EVEX embedded rounding is ignored.

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

### Operation

#### VCVTUDQ2PD (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\_UIInteger\_To\_Double\_Precision\_Floating\_Point(SRC[k+31:k])

  ELSE

    IF \*merging-masking\* ; merging-masking

      THEN \*DEST[i+63:i] remains unchanged\*

    ELSE ; zeroing-masking

      DEST[i+63:i] ← 0

  FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VCVTUDQ2PD (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_UIInteger_To_Double_Precision_Floating_Point(SRC[31:0])
        ELSE
          DEST[i+63:i] ←
          Convert_UIInteger_To_Double_Precision_Floating_Point(SRC[k+31:k])
      FI;
    ELSE
      IF *merging-masking*           ; merging-masking
        THEN *DEST[i+63:i] remains unchanged*
      ELSE                             ; zeroing-masking
        DEST[i+63:i] ← 0
      FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

#### Intel C/C++ Compiler Intrinsic Equivalent

```

VCVTUDQ2PD __m512d __mm512_cvtepu32_pd( __m256i a);
VCVTUDQ2PD __m512d __mm512_mask_cvtepu32_pd( __m512d s, __mmask8 k, __m256i a);
VCVTUDQ2PD __m512d __mm512_maskz_cvtepu32_pd( __mmask8 k, __m256i a);
VCVTUDQ2PD __m256d __mm256_cvtepu32_pd( __m128i a);
VCVTUDQ2PD __m256d __mm256_mask_cvtepu32_pd( __m256d s, __mmask8 k, __m128i a);
VCVTUDQ2PD __m256d __mm256_maskz_cvtepu32_pd( __mmask8 k, __m128i a);
VCVTUDQ2PD __m128d __mm_cvtepu32_pd( __m128i a);
VCVTUDQ2PD __m128d __mm_mask_cvtepu32_pd( __m128d s, __mmask8 k, __m128i a);
VCVTUDQ2PD __m128d __mm_maskz_cvtepu32_pd( __mmask8 k, __m128i a);

```

#### SIMD Floating-Point Exceptions

None

#### Other Exceptions

EVEX-encoded instructions, see Exceptions Type E5.

#UD If EVEX.vvvv != 1111B.

## VCVTUDQ2PS—Convert Packed Unsigned Doubleword Integers to Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.F2.0F.W0 7A /r VCVTUDQ2PS xmm1 {k1}{z}, xmm2/m128/m32bcst	FV	V/V	AVX512VL AVX512F	Convert four packed unsigned doubleword integers from xmm2/m128/m32bcst to packed single-precision floating-point values in xmm1 with writemask k1.
EVEX.256.F2.0F.W0 7A /r VCVTUDQ2PS ymm1 {k1}{z}, ymm2/m256/m32bcst	FV	V/V	AVX512VL AVX512F	Convert eight packed unsigned doubleword integers from ymm2/m256/m32bcst to packed single-precision floating-point values in zmm1 with writemask k1.
EVEX.512.F2.0F.W0 7A /r VCVTUDQ2PS zmm1 {k1}{z}, zmm2/m512/m32bcst{er}	FV	V/V	AVX512F	Convert sixteen packed unsigned doubleword integers from zmm2/m512/m32bcst to sixteen packed single-precision floating-point values in zmm1 with writemask k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
FV	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Converts packed unsigned doubleword integers in the source operand (second operand) to single-precision floating-point values in the destination operand (first operand).

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 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

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

### Operation

**VCVTUDQ2PS (EVEX encoded version) when src operand is a register**

(KL, VL) = (4, 128), (8, 256), (16, 512)

IF (VL = 512) AND (EVEX.b = 1)

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\*

THEN DEST[i+31:i] ←

Convert\_UIInteger\_To\_Single\_Precision\_Floating\_Point(SRC[i+31:i])

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+31:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

```
ENDFOR
DEST[MAX_VL-1:VL] ← 0
```

**VCVTUDQ2PS (EVEX encoded version) when src operand is a memory source**  
 (KL, VL) = (4, 128), (8, 256), (16, 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_UInteger_To_Single_Precision_Floating_Point(SRC[31:0])
        ELSE
          DEST[i+31:i] ←
            Convert_UInteger_To_Single_Precision_Floating_Point(SRC[i+31:i])
      FI;
    ELSE
      IF *merging-masking*           ; merging-masking
        THEN *DEST[i+31:i] remains unchanged*
      ELSE                             ; zeroing-masking
        DEST[i+31:i] ← 0
      FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0
```

### Intel C/C++ Compiler Intrinsic Equivalent

```
VCVTUDQ2PS __m512 __mm512_cvtepu32_ps( __m512i a);
VCVTUDQ2PS __m512 __mm512_mask_cvtepu32_ps( __m512 s, __mmask16 k, __m512i a);
VCVTUDQ2PS __m512 __mm512_maskz_cvtepu32_ps( __mmask16 k, __m512i a);
VCVTUDQ2PS __m512 __mm512_cvt_roundepu32_ps( __m512i a, int r);
VCVTUDQ2PS __m512 __mm512_mask_cvt_roundepu32_ps( __m512 s, __mmask16 k, __m512i a, int r);
VCVTUDQ2PS __m512 __mm512_maskz_cvt_roundepu32_ps( __mmask16 k, __m512i a, int r);
VCVTUDQ2PS __m256 __mm256_cvtepu32_ps( __m256i a);
VCVTUDQ2PS __m256 __mm256_mask_cvtepu32_ps( __m256 s, __mmask8 k, __m256i a);
VCVTUDQ2PS __m256 __mm256_maskz_cvtepu32_ps( __mmask8 k, __m256i a);
VCVTUDQ2PS __m128 __mm_cvtepu32_ps( __m128i a);
VCVTUDQ2PS __m128 __mm_mask_cvtepu32_ps( __m128 s, __mmask8 k, __m128i a);
VCVTUDQ2PS __m128 __mm_maskz_cvtepu32_ps( __mmask8 k, __m128i a);
```

### SIMD Floating-Point Exceptions

Precision

### Other Exceptions

EVEX-encoded instructions, see Exceptions Type E2.

#UD If EVEX.vvvv != 1111B.



## VCVTUQQ2PD—Convert Packed Unsigned Quadword Integers to Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.F3.0F.W1 7A /r VCVTUQQ2PD xmm1 {k1}{z}, xmm2/m128/m64bcst	FV	V/V	AVX512VL AVX512DQ	Convert two packed unsigned quadword integers from xmm2/m128/m64bcst to two packed double-precision floating-point values in xmm1 with writemask k1.
EVEX.256.F3.0F.W1 7A /r VCVTUQQ2PD ymm1 {k1}{z}, ymm2/m256/m64bcst	FV	V/V	AVX512VL AVX512DQ	Convert four packed unsigned quadword integers from ymm2/m256/m64bcst to packed double-precision floating-point values in ymm1 with writemask k1.
EVEX.512.F3.0F.W1 7A /r VCVTUQQ2PD zmm1 {k1}{z}, zmm2/m512/m64bcst{er}	FV	V/V	AVX512DQ	Convert eight packed unsigned quadword integers from zmm2/m512/m64bcst to eight packed double-precision floating-point values in zmm1 with writemask k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
FV	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Converts packed unsigned quadword integers in the source operand (second operand) to packed double-precision floating-point values in the destination operand (first operand).

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 64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

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

### Operation

#### VCVTUQQ2PD (EVEX encoded version) when src operand is a register

(KL, VL) = (2, 128), (4, 256), (8, 512)

IF (VL == 512) AND (EVEX.b == 1)

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN DEST[i+63:i] ←

Convert\_UQuadInteger\_To\_Double\_Precision\_Floating\_Point(SRC[i+63:i])

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

```
ENDFOR
DEST[MAX_VL-1:VL] ← 0
```

#### VCVTUQQ2PD (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_UQuadInteger_To_Double_Precision_Floating_Point(SRC[63:0])
        ELSE
          DEST[i+63:i] ←
          Convert_UQuadInteger_To_Double_Precision_Floating_Point(SRC[i+63:i])
        FI;
      ELSE
        IF *merging-masking* ; merging-masking
          THEN *DEST[i+63:i] remains unchanged*
        ELSE ; zeroing-masking
          DEST[i+63:i] ← 0
        FI
      FI;
    FI;
  FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0
```

#### Intel C/C++ Compiler Intrinsic Equivalent

```
VCVTUQQ2PD __m512d __mm512_cvtepu64_ps( __m512i a);
VCVTUQQ2PD __m512d __mm512_mask_cvtepu64_ps( __m512d s, __mmask8 k, __m512i a);
VCVTUQQ2PD __m512d __mm512_maskz_cvtepu64_ps( __mmask8 k, __m512i a);
VCVTUQQ2PD __m512d __mm512_cvt_roundepu64_ps( __m512i a, int r);
VCVTUQQ2PD __m512d __mm512_mask_cvt_roundepu64_ps( __m512d s, __mmask8 k, __m512i a, int r);
VCVTUQQ2PD __m512d __mm512_maskz_cvt_roundepu64_ps( __mmask8 k, __m512i a, int r);
VCVTUQQ2PD __m256d __mm256_cvtepu64_ps( __m256i a);
VCVTUQQ2PD __m256d __mm256_mask_cvtepu64_ps( __m256d s, __mmask8 k, __m256i a);
VCVTUQQ2PD __m256d __mm256_maskz_cvtepu64_ps( __mmask8 k, __m256i a);
VCVTUQQ2PD __m128d __mm_cvtepu64_ps( __m128i a);
VCVTUQQ2PD __m128d __mm_mask_cvtepu64_ps( __m128d s, __mmask8 k, __m128i a);
VCVTUQQ2PD __m128d __mm_maskz_cvtepu64_ps( __mmask8 k, __m128i a);
```

#### SIMD Floating-Point Exceptions

Precision

#### Other Exceptions

EVEX-encoded instructions, see Exceptions Type E2.

#UD If EVEX.vvvv != 1111B.

## VCVTUQQ2PS—Convert Packed Unsigned Quadword Integers to Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.F2.0F.W1 7A /r VCVTUQQ2PS xmm1 {k1}{z}, xmm2/m128/m64bcst	FV	V/V	AVX512VL AVX512DQ	Convert two packed unsigned quadword integers from xmm2/m128/m64bcst to packed single-precision floating-point values in zmm1 with writemask k1.
EVEX.256.F2.0F.W1 7A /r VCVTUQQ2PS xmm1 {k1}{z}, ymm2/m256/m64bcst	FV	V/V	AVX512VL AVX512DQ	Convert four packed unsigned quadword integers from ymm2/m256/m64bcst to packed single-precision floating-point values in xmm1 with writemask k1.
EVEX.512.F2.0F.W1 7A /r VCVTUQQ2PS ymm1 {k1}{z}, zmm2/m512/m64bcst{er}	FV	V/V	AVX512DQ	Convert eight packed unsigned quadword integers from zmm2/m512/m64bcst to eight packed single-precision floating-point values in zmm1 with writemask k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
FV	ModRM:rreg (w)	ModRM:r/m (r)	NA	NA

### Description

Converts packed unsigned quadword integers in the source operand (second operand) to single-precision floating-point values in the destination operand (first operand).

EVEX encoded versions: The source operand is a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operand is a YMM/XMM/XMM (low 64 bits) register conditionally updated with writemask k1.

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

### Operation

#### VCVTUQQ2PS (EVEX encoded version) when src operand is a register

(KL, VL) = (2, 128), (4, 256), (8, 512)

IF (VL = 512) AND (EVEX.b = 1)

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j \* 32

k ← j \* 64

IF k1[j] OR \*no writemask\*

THEN DEST[i+31:i] ←

Convert\_UQuadInteger\_To\_Single\_Precision\_Floating\_Point(SRC[k+63:k])

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+31:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

```

ENDFOR
DEST[MAX_VL-1:VL/2] ← 0
VCVTUQQ2PS (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 * 32
  k ← j * 64
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1)
        THEN
          DEST[i+31:i] ←
          Convert_UQuadInteger_To_Single_Precision_Floating_Point(SRC[63:0])
        ELSE
          DEST[i+31:i] ←
          Convert_UQuadInteger_To_Single_Precision_Floating_Point(SRC[k+63:k])
        FI;
      ELSE
        IF *merging-masking* ; merging-masking
          THEN *DEST[i+31:i] remains unchanged*
        ELSE ; zeroing-masking
          DEST[i+31:i] ← 0
        FI
      FI;
    ENDIF
  ENDIF
ENDFOR
DEST[MAX_VL-1:VL/2] ← 0

```

### Intel C/C++ Compiler Intrinsic Equivalent

```

VCVTUQQ2PS __m256 __mm512_cvtepu64_ps( __m512i a);
VCVTUQQ2PS __m256 __mm512_mask_cvtepu64_ps( __m256 s, __mmask8 k, __m512i a);
VCVTUQQ2PS __m256 __mm512_maskz_cvtepu64_ps( __mmask8 k, __m512i a);
VCVTUQQ2PS __m256 __mm512_cvt_roundepu64_ps( __m512i a, int r);
VCVTUQQ2PS __m256 __mm512_mask_cvt_roundepu64_ps( __m256 s, __mmask8 k, __m512i a, int r);
VCVTUQQ2PS __m256 __mm512_maskz_cvt_roundepu64_ps( __mmask8 k, __m512i a, int r);
VCVTUQQ2PS __m128 __mm256_cvtepu64_ps( __m256i a);
VCVTUQQ2PS __m128 __mm256_mask_cvtepu64_ps( __m128 s, __mmask8 k, __m256i a);
VCVTUQQ2PS __m128 __mm256_maskz_cvtepu64_ps( __mmask8 k, __m256i a);
VCVTUQQ2PS __m128 __mm_cvtepu64_ps( __m128i a);
VCVTUQQ2PS __m128 __mm_mask_cvtepu64_ps( __m128 s, __mmask8 k, __m128i a);
VCVTUQQ2PS __m128 __mm_maskz_cvtepu64_ps( __mmask8 k, __m128i a);

```

### SIMD Floating-Point Exceptions

Precision

#### Other Exceptions

EVEX-encoded instructions, see Exceptions Type E2.

#UD If EVEX.vvvv != 1111B.

## VCVTUSI2SD—Convert Unsigned Integer to Scalar Double-Precision Floating-Point Value

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.LIG.F2.OF.W0 7B /r VCVTUSI2SD xmm1, xmm2, r/m32	T1S	V/V	AVX512F	Convert one unsigned doubleword integer from r/m32 to one double-precision floating-point value in xmm1.
EVEX.NDS.LIG.F2.OF.W1 7B /r VCVTUSI2SD xmm1, xmm2, r/m64{er}	T1S	V/N.E. <sup>1</sup>	AVX512F	Convert one unsigned quadword integer from r/m64 to one double-precision floating-point value in xmm1.

### NOTES:

- For this specific instruction, EVEX.W in non-64 bit is ignored; the instructions behaves as if the W0 version is used.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
T1S	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

### Description

Converts an unsigned doubleword integer (or unsigned quadword integer if operand size is 64 bits) in the second source operand to a double-precision floating-point value in the destination operand. The result is stored in the low quadword of the destination operand. When conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register.

The second source operand can be a general-purpose register or a 32/64-bit memory location. The first source and destination operands are XMM registers. Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (MAX\_VL-1:128) of the destination register are zeroed.

EVEX.W1 version: promotes the instruction to use 64-bit input value in 64-bit mode.

EVEX.W0 version: attempt to encode this instruction with EVEX embedded rounding is ignored.

### Operation

#### VCVTUSI2SD (EVEX encoded version)

IF (SRC2 \*is register\*) AND (EVEX.b = 1)

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

IF 64-Bit Mode And OperandSize = 64

THEN

DEST[63:0] ← Convert\_UInteger\_To\_Double\_Precision\_Floating\_Point(SRC2[63:0]);

ELSE

DEST[63:0] ← Convert\_UInteger\_To\_Double\_Precision\_Floating\_Point(SRC2[31:0]);

FI;

DEST[127:64] ← SRC1[127:64]

DEST[MAX\_VL-1:128] ← 0

### Intel C/C++ Compiler Intrinsic Equivalent

VCVTUSI2SD \_\_m128d \_\_mm\_cvtsd\_32\_sd( \_\_m128d s, unsigned a);

VCVTUSI2SD \_\_m128d \_\_mm\_cvtsd\_64\_sd( \_\_m128d s, unsigned \_\_int64 a);

VCVTUSI2SD \_\_m128d \_\_mm\_cvtsd\_round64\_sd( \_\_m128d s, unsigned \_\_int64 a, int r);



**SIMD Floating-Point Exceptions**

Precision

**Other Exceptions**

See Exceptions Type E3NF if W1, else type E10NF.

## VCVTUSI2SS—Convert Unsigned Integer to Scalar Single-Precision Floating-Point Value

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.LIG.F3.0F.W0 7B /r VCVTUSI2SS xmm1, xmm2, r/m32{er}	T1S	V/V	AVX512F	Convert one signed doubleword integer from r/m32 to one single-precision floating-point value in xmm1.
EVEX.NDS.LIG.F3.0F.W1 7B /r VCVTUSI2SS xmm1, xmm2, r/m64{er}	T1S	V/N.E. <sup>1</sup>	AVX512F	Convert one signed quadword integer from r/m64 to one single-precision floating-point value in xmm1.

### NOTES:

- For this specific instruction, EVEX.W in non-64 bit is ignored; the instructions behaves as if the W0 version is used.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
T1S	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA

### Description

Converts a unsigned doubleword integer (or unsigned quadword integer if operand size is 64 bits) in the source operand (second operand) to a single-precision floating-point value in the destination operand (first operand). The source operand can be a general-purpose register or a memory location. The destination operand is an XMM register. The result is stored in the low doubleword of the destination operand. When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits.

The second source operand can be a general-purpose register or a 32/64-bit memory location. The first source and destination operands are XMM registers. Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (MAX\_VL-1:128) of the destination register are zeroed.

EVEX.W1 version: promotes the instruction to use 64-bit input value in 64-bit mode.

### Operation

#### VCVTUSI2SS (EVEX encoded version)

IF (SRC2 \*is register\*) AND (EVEX.b = 1)

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

IF 64-Bit Mode And OperandSize = 64

THEN

DEST[31:0] ← Convert\_UInteger\_To\_Single\_Precision\_Floating\_Point(SRC[63:0]);

ELSE

DEST[31:0] ← Convert\_UInteger\_To\_Single\_Precision\_Floating\_Point(SRC[31:0]);

FI;

DEST[127:32] ← SRC1[127:32]

DEST[MAX\_VL-1:128] ← 0

### Intel C/C++ Compiler Intrinsic Equivalent

VCVTUSI2SS \_\_m128 \_mm\_cvtsi32\_ss( \_\_m128 s, unsigned a);

VCVTUSI2SS \_\_m128 \_mm\_cvt\_roundsi32\_ss( \_\_m128 s, unsigned a, int r);

VCVTUSI2SS \_\_m128 \_mm\_cvtsi64\_ss( \_\_m128 s, unsigned \_\_int64 a);

VCVTUSI2SS \_\_m128 \_mm\_cvt\_roundsi64\_ss( \_\_m128 s, unsigned \_\_int64 a, int r);



**SIMD Floating-Point Exceptions**

Precision

**Other Exceptions**

See Exceptions Type E3NF.



## VDBPSADBW—Double Block Packed Sum-Absolute-Differences (SAD) on Unsigned Bytes

Opcode/ Instruction	Op/ En	64/32 bitMode Support	CPUID Feature Flag	Description
EVEX.NDS.128.66.0F3A.W0 42 /r ib VDBPSADBW xmm1 {k1}{z}, xmm2, xmm3/m128, imm8	FVM	V/V	AVX512VL AVX512BW	Compute packed SAD word results of unsigned bytes in dword block from xmm2 with unsigned bytes of dword blocks transformed from xmm3/m128 using the shuffle controls in imm8. Results are written to xmm1 under the writemask k1.
EVEX.NDS.256.66.0F3A.W0 42 /r ib VDBPSADBW ymm1 {k1}{z}, ymm2, ymm3/m256, imm8	FVM	V/V	AVX512VL AVX512BW	Compute packed SAD word results of unsigned bytes in dword block from ymm2 with unsigned bytes of dword blocks transformed from ymm3/m256 using the shuffle controls in imm8. Results are written to ymm1 under the writemask k1.
EVEX.NDS.512.66.0F3A.W0 42 /r ib VDBPSADBW zmm1 {k1}{z}, zmm2, zmm3/m512, imm8	FVM	V/V	AVX512BW	Compute packed SAD word results of unsigned bytes in dword block from zmm2 with unsigned bytes of dword blocks transformed from zmm3/m512 using the shuffle controls in imm8. Results are written to zmm1 under the writemask k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
FVM	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	Imm8

### Description

Compute packed SAD (sum of absolute differences) word results of unsigned bytes from two 32-bit dword elements. Packed SAD word results are calculated in multiples of qword superblocks, producing 4 SAD word results in each 64-bit superblock of the destination register.

Within each super block of packed word results, the SAD results from two 32-bit dword elements are calculated as follows:

- The lower two word results are calculated each from the SAD operation between a sliding dword element within a qword superblock from an intermediate vector with a stationary dword element in the corresponding qword superblock of the first source operand. The intermediate vector, see “Tmp1” in Figure 5-8, is constructed from the second source operand the imm8 byte as shuffle control to select dword elements within a 128-bit lane of the second source operand. The two sliding dword elements in a qword superblock of Tmp1 are located at byte offset 0 and 1 within the superblock, respectively. The stationary dword element in the qword superblock from the first source operand is located at byte offset 0.
- The next two word results are calculated each from the SAD operation between a sliding dword element within a qword superblock from the intermediate vector Tmp1 with a second stationary dword element in the corresponding qword superblock of the first source operand. The two sliding dword elements in a qword superblock of Tmp1 are located at byte offset 2 and 3 within the superblock, respectively. The stationary dword element in the qword superblock from the first source operand is located at byte offset 4.
- The intermediate vector is constructed in 128-bit lanes. Within each 128-bit lane, each dword element of the intermediate vector is selected by a two-bit field within the imm8 byte on the corresponding 128-bits of the second source operand. The imm8 byte serves as dword shuffle control within each 128-bit lanes of the intermediate vector and the second source operand, similarly to PSHUFD.

The first source operand is a ZMM/YMM/XMM register. The second source operand is a ZMM/YMM/XMM register, or a 512/256/128-bit memory location. The destination operand is conditionally updated based on writemask k1 at 16-bit word granularity.

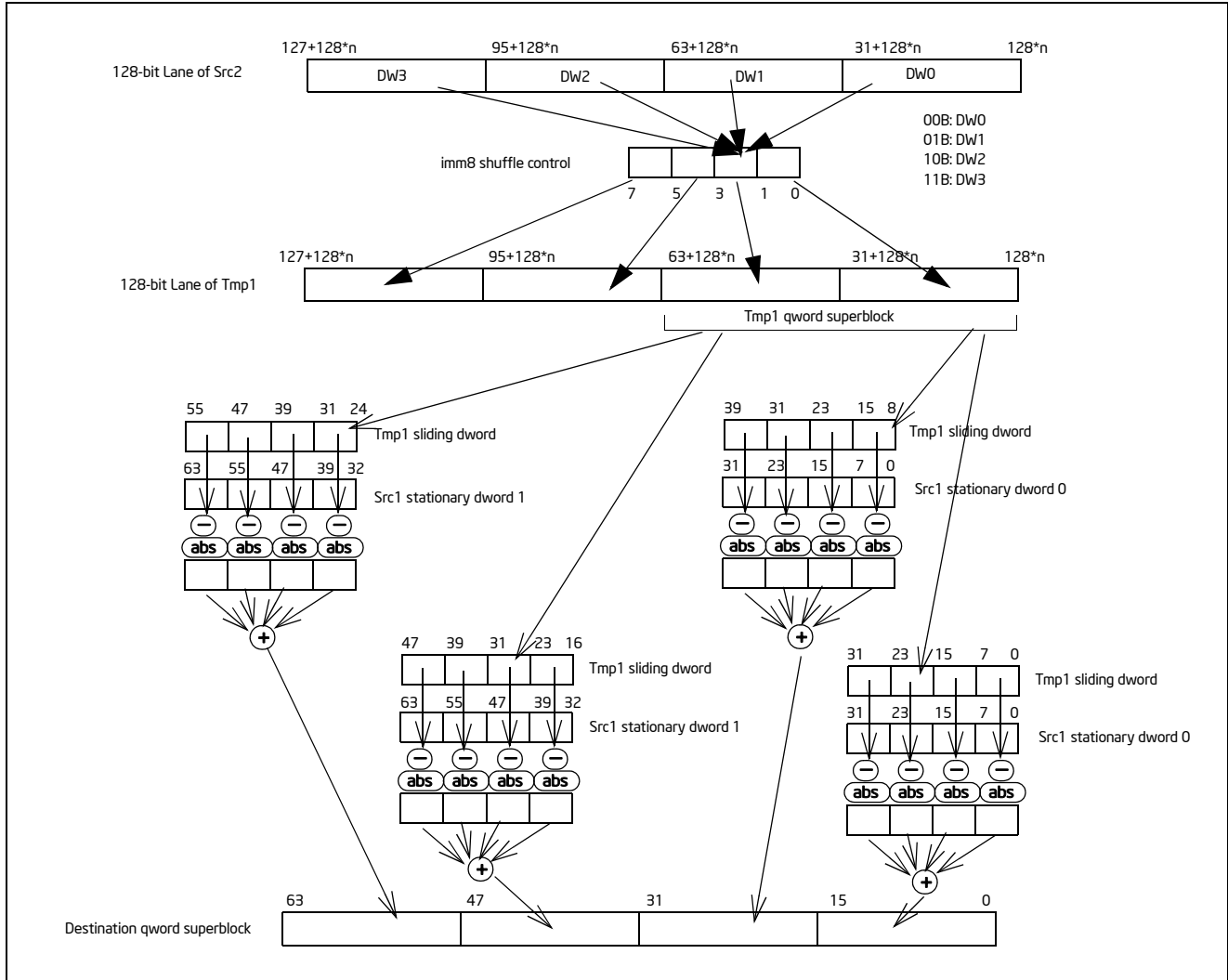


Figure 5-8. 64-bit Super Block of SAD Operation in VDBPSADBW

**Operation**

**VDBPSADBW (EVEX encoded versions)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

Selection of quadruplets:

FOR I = 0 to VL step 128

- TMP1[I+31:I] ← select (SRC2[I+127: I], imm8[1:0])
- TMP1[I+63: I+32] ← select (SRC2[I+127: I], imm8[3:2])
- TMP1[I+95: I+64] ← select (SRC2[I+127: I], imm8[5:4])
- TMP1[I+127: I+96] ← select (SRC2[I+127: I], imm8[7:6])

END FOR

SAD of quadruplets:

FOR I = 0 to VL step 64

- TMP\_DEST[I+15:I] ← ABS(SRC1[I+7: I] - TMP1[I+7: I]) +  
ABS(SRC1[I+15: I+8] - TMP1[I+15: I+8]) +

```
ABS(SRC1[I+23:I+16]- TMP1[I+23:I+16]) +
ABS(SRC1[I+31:I+24]- TMP1[I+31:I+24])
```

```
TMP_DEST[I+31:I+16] ← ABS(SRC1[I+7:I] - TMP1[I+15:I+8]) +
ABS(SRC1[I+15:I+8]- TMP1[I+23:I+16]) +
ABS(SRC1[I+23:I+16]- TMP1[I+31:I+24]) +
ABS(SRC1[I+31:I+24]- TMP1[I+39:I+32])
TMP_DEST[I+47:I+32] ← ABS(SRC1[I+39:I+32] - TMP1[I+23:I+16]) +
ABS(SRC1[I+47:I+40]- TMP1[I+31:I+24]) +
ABS(SRC1[I+55:I+48]- TMP1[I+39:I+32]) +
ABS(SRC1[I+63:I+56]- TMP1[I+47:I+40])
```

```
TMP_DEST[I+63:I+48] ← ABS(SRC1[I+39:I+32] - TMP1[I+31:I+24]) +
ABS(SRC1[I+47:I+40] - TMP1[I+39:I+32]) +
ABS(SRC1[I+55:I+48] - TMP1[I+47:I+40]) +
ABS(SRC1[I+63:I+56] - TMP1[I+55:I+48])
```

```
ENDFOR
```

```
FOR j ← 0 TO KL-1
```

```
  i ← j * 16
```

```
  IF k1[j] OR *no writemask*
```

```
    THEN DEST[i+15:i] ← TMP_DEST[i+15:i]
```

```
  ELSE
```

```
    IF *merging-masking* ; merging-masking
```

```
      THEN *DEST[i+15:i] remains unchanged*
```

```
    ELSE ; zeroing-masking
```

```
      DEST[i+15:i] ← 0
```

```
  FI
```

```
FI;
```

```
ENDFOR
```

```
DEST[MAX_VL-1:VL] ← 0
```

### Intel C/C++ Compiler Intrinsic Equivalent

```
VDBPSADBW __m512i _mm512_dbsad_epu8(__m512i a, __m512i b);
VDBPSADBW __m512i _mm512_mask_dbsad_epu8(__m512i s, __mmask32 m, __m512i a, __m512i b);
VDBPSADBW __m512i _mm512_maskz_dbsad_epu8(__mmask32 m, __m512i a, __m512i b);
VDBPSADBW __m256i _mm256_dbsad_epu8(__m256i a, __m256i b);
VDBPSADBW __m256i _mm256_mask_dbsad_epu8(__m256i s, __mmask16 m, __m256i a, __m256i b);
VDBPSADBW __m256i _mm256_maskz_dbsad_epu8(__mmask16 m, __m256i a, __m256i b);
VDBPSADBW __m128i _mm_dbsad_epu8(__m128i a, __m128i b);
VDBPSADBW __m128i _mm_mask_dbsad_epu8(__m128i s, __mmask8 m, __m128i a, __m128i b);
VDBPSADBW __m128i _mm_maskz_dbsad_epu8(__mmask8 m, __m128i a, __m128i b);
```

### SIMD Floating-Point Exceptions

None

### Other Exceptions

See Exceptions Type E4NF.nb.

## VEXPANDPD—Load Sparse Packed Double-Precision Floating-Point Values from Dense Memory

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W1 88 /r VEXPANDPD xmm1 {k1}{z}, xmm2/m128	T1S	V/V	AVX512VL AVX512F	Expand packed double-precision floating-point values from xmm2/m128 to xmm1 using writemask k1.
EVEX.256.66.0F38.W1 88 /r VEXPANDPD ymm1 {k1}{z}, ymm2/m256	T1S	V/V	AVX512VL AVX512F	Expand packed double-precision floating-point values from ymm2/m256 to ymm1 using writemask k1.
EVEX.512.66.0F38.W1 88 /r VEXPANDPD zmm1 {k1}{z}, zmm2/m512	T1S	V/V	AVX512F	Expand packed double-precision floating-point values from zmm2/m512 to zmm1 using writemask k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
T1S	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Expand (load) up to 8/4/2, contiguous, double-precision floating-point values of the input vector in the source operand (the second operand) to sparse elements in the destination operand (the first operand) selected by the writemask k1.

The destination operand is a ZMM/YMM/XMM register, the source operand can be a ZMM/YMM/XMM register or a 512/256/128-bit memory location.

The input vector starts from the lowest element in the source operand. The writemask register k1 selects the destination elements (a partial vector or sparse elements if less than 8 elements) to be replaced by the ascending elements in the input vector. Destination elements not selected by the writemask k1 are either unmodified or zeroed, depending on EVEX.z.

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

Note that the compressed displacement assumes a pre-scaling (N) corresponding to the size of one single element instead of the size of the full vector.

### Operation

#### VEXPANDPD (EVEX encoded versions)

(KL, VL) = (2, 128), (4, 256), (8, 512)

k ← 0

FOR j ← 0 TO KL-1

  i ← j \* 64

  IF k1[j] OR \*no writemask\*

    THEN

      DEST[i+63:i] ← SRC[k+63:k];

      k ← k + 64

    ELSE

      IF \*merging-masking\* ; merging-masking

        THEN \*DEST[i+63:i] remains unchanged\*

        ELSE ; zeroing-masking

          THEN DEST[i+63:i] ← 0

    FI

  FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VEXPANDPD __m512d __mm512_mask_expand_pd( __m512d s, __mmask8 k, __m512d a);
VEXPANDPD __m512d __mm512_maskz_expand_pd( __mmask8 k, __m512d a);
VEXPANDPD __m512d __mm512_mask_expandloadu_pd( __m512d s, __mmask8 k, void * a);
VEXPANDPD __m512d __mm512_maskz_expandloadu_pd( __mmask8 k, void * a);
VEXPANDPD __m256d __mm256_mask_expand_pd( __m256d s, __mmask8 k, __m256d a);
VEXPANDPD __m256d __mm256_maskz_expand_pd( __mmask8 k, __m256d a);
VEXPANDPD __m256d __mm256_mask_expandloadu_pd( __m256d s, __mmask8 k, void * a);
VEXPANDPD __m256d __mm256_maskz_expandloadu_pd( __mmask8 k, void * a);
VEXPANDPD __m128d __mm_mask_expand_pd( __m128d s, __mmask8 k, __m128d a);
VEXPANDPD __m128d __mm_maskz_expand_pd( __mmask8 k, __m128d a);
VEXPANDPD __m128d __mm_mask_expandloadu_pd( __m128d s, __mmask8 k, void * a);
VEXPANDPD __m128d __mm_maskz_expandloadu_pd( __mmask8 k, void * a);

```

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type E4.nb.

#UD                    If EVEX.vvvv != 1111B.

## VEXPANDPS—Load Sparse Packed Single-Precision Floating-Point Values from Dense Memory

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 88 /r VEXPANDPS xmm1 {k1}{z}, xmm2/m128	T1S	V/V	AVX512VL AVX512F	Expand packed single-precision floating-point values from xmm2/m128 to xmm1 using writemask k1.
EVEX.256.66.0F38.W0 88 /r VEXPANDPS ymm1 {k1}{z}, ymm2/m256	T1S	V/V	AVX512VL AVX512F	Expand packed single-precision floating-point values from ymm2/m256 to ymm1 using writemask k1.
EVEX.512.66.0F38.W0 88 /r VEXPANDPS zmm1 {k1}{z}, zmm2/m512	T1S	V/V	AVX512F	Expand packed single-precision floating-point values from zmm2/m512 to zmm1 using writemask k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
T1S	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Expand (load) up to 16/8/4, contiguous, single-precision floating-point values of the input vector in the source operand (the second operand) to sparse elements of the destination operand (the first operand) selected by the writemask k1.

The destination operand is a ZMM/YMM/XMM register, the source operand can be a ZMM/YMM/XMM register or a 512/256/128-bit memory location.

The input vector starts from the lowest element in the source operand. The writemask k1 selects the destination elements (a partial vector or sparse elements if less than 16 elements) to be replaced by the ascending elements in the input vector. Destination elements not selected by the writemask k1 are either unmodified or zeroed, depending on EVEX.z.

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

Note that the compressed displacement assumes a pre-scaling (N) corresponding to the size of one single element instead of the size of the full vector.

### Operation

#### VEXPANDPS (EVEX encoded versions)

(KL, VL) = (4, 128), (8, 256), (16, 512)

k ← 0

FOR j ← 0 TO KL-1

  i ← j \* 32

  IF k1[j] OR \*no writemask\*

    THEN

      DEST[i+31:i] ← SRC[k+31:k];

      k ← k + 32

    ELSE

      IF \*merging-masking\* ; merging-masking

        THEN \*DEST[i+31:i] remains unchanged\*

        ELSE ; zeroing-masking

          DEST[i+31:i] ← 0

    FI

  FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VEXPANDPS __m512 __mm512_mask_expand_ps( __m512 s, __mmask16 k, __m512 a);
VEXPANDPS __m512 __mm512_maskz_expand_ps( __mmask16 k, __m512 a);
VEXPANDPS __m512 __mm512_mask_expandloadu_ps( __m512 s, __mmask16 k, void * a);
VEXPANDPS __m512 __mm512_maskz_expandloadu_ps( __mmask16 k, void * a);
VEXPANDPD __m256 __mm256_mask_expand_ps( __m256 s, __mmask8 k, __m256 a);
VEXPANDPD __m256 __mm256_maskz_expand_ps( __mmask8 k, __m256 a);
VEXPANDPD __m256 __mm256_mask_expandloadu_ps( __m256 s, __mmask8 k, void * a);
VEXPANDPD __m256 __mm256_maskz_expandloadu_ps( __mmask8 k, void * a);
VEXPANDPD __m128 __mm_mask_expand_ps( __m128 s, __mmask8 k, __m128 a);
VEXPANDPD __m128 __mm_maskz_expand_ps( __mmask8 k, __m128 a);
VEXPANDPD __m128 __mm_mask_expandloadu_ps( __m128 s, __mmask8 k, void * a);
VEXPANDPD __m128 __mm_maskz_expandloadu_ps( __mmask8 k, void * a);

```

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type E4.nb.

#UD                    If EVEX.vvvv != 1111B.

## VEXP2PD—Approximation to the Exponential $2^x$ of Packed Double-Precision Floating-Point Values with Less Than $2^{-23}$ Relative Error

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.512.66.0F38.W1 C8 /r VEXP2PD zmm1 {k1}{z}, zmm2/m512/m64bcst {sae}	FV	V/V	AVX512ER	Computes approximations to the exponential $2^x$ (with less than $2^{-23}$ of maximum relative error) of the packed double-precision floating-point values from zmm2/m512/m64bcst and stores the floating-point result in zmm1 with writemask k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
FV	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA

### Description

Computes the approximate base-2 exponential evaluation of the double-precision floating-point values in the source operand (the second operand) and stores the results to the destination operand (the first operand) using the writemask k1. The approximate base-2 exponential is evaluated with less than  $2^{-23}$  of relative error.

Denormal input values are treated as zeros and do not signal #DE, irrespective of MXCSR.DAZ. Denormal results are flushed to zeros and do not signal #UE, irrespective of MXCSR.FZ.

The source operand is a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

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

A numerically exact implementation of VEXP2xx can be found at <https://software.intel.com/en-us/articles/reference-implementations-for-IA-approximation-instructions-vrcp14-vrsqrt14-vrcp28-vrsqrt28-vexp2>.

### Operation

#### VEXP2PD

(KL, VL) = (8, 512)

FOR j ← 0 TO KL-1

  i ← j \* 64

  IF k1[j] OR \*no writemask\* THEN

    IF (EVEX.b = 1) AND (SRC \*is memory\*)

      THEN DEST[i+63:i] ← EXP2\_23\_DP(SRC[63:0])

      ELSE DEST[i+63:i] ← EXP2\_23\_DP(SRC[i+63:i])

    FI;

  ELSE

    IF \*merging-masking\* ; merging-masking

      THEN \*DEST[i+63:i] remains unchanged\*

    ELSE ; zeroing-masking

      DEST[i+63:i] ← 0

    FI;

  FI;

ENDFOR;



Table 5-4. Special Values Behavior

Source Input	Result	Comments
NaN	QNaN(src)	If (SRC = SNaN) then #I
$+\infty$	$+\infty$	
$+/-0$	1.0f	<i>Exact result</i>
$-\infty$	+0.0f	
Integral value N	$2^N$	<i>Exact result</i>

**Intel C/C++ Compiler Intrinsic Equivalent**

VEXP2PD \_\_m512d \_\_mm512\_exp2a23\_round\_pd (\_\_m512d a, int sae);

VEXP2PD \_\_m512d \_\_mm512\_mask\_exp2a23\_round\_pd (\_\_m512d a, \_\_mmask8 m, \_\_m512d b, int sae);

VEXP2PD \_\_m512d \_\_mm512\_maskz\_exp2a23\_round\_pd (\_\_mmask8 m, \_\_m512d b, int sae);

**SIMD Floating-Point Exceptions**

Invalid (if SNaN input), Overflow

**Other Exceptions**

See Exceptions Type E2.

## VEXP2PS—Approximation to the Exponential $2^x$ of Packed Single-Precision Floating-Point Values with Less Than $2^{-23}$ Relative Error

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.512.66.0F38.W0 C8 /r VEXP2PS zmm1 {k1}{z}, zmm2/m512/m32bcst {sae}	FV	V/V	AVX512ER	Computes approximations to the exponential $2^x$ (with less than $2^{-23}$ of maximum relative error) of the packed single-precision floating-point values from zmm2/m512/m32bcst and stores the floating-point result in zmm1 with writemask k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
FV	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA

### Description

Computes the approximate base-2 exponential evaluation of the single-precision floating-point values in the source operand (the second operand) and store the results in the destination operand (the first operand) using the writemask k1. The approximate base-2 exponential is evaluated with less than  $2^{-23}$  of relative error.

Denormal input values are treated as zeros and do not signal #DE, irrespective of MXCSR.DAZ. Denormal results are flushed to zeros and do not signal #UE, irrespective of MXCSR.FZ.

The source operand is a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

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

A numerically exact implementation of VEXP2xx can be found at <https://software.intel.com/en-us/articles/reference-implementations-for-IA-approximation-instructions-vrcp14-vrsqrt14-vrcp28-vrsqrt28-vexp2>.

### Operation

#### VEXP2PS

(KL, VL) = (16, 512)

FOR j ← 0 TO KL-1

  i ← j \* 32

  IF k1[j] OR \*no writemask\* THEN

    IF (EVEX.b = 1) AND (SRC \*is memory\*)

      THEN DEST[i+31:i] ← EXP2\_23\_SP(SRC[31:0])

      ELSE DEST[i+31:i] ← EXP2\_23\_SP(SRC[i+31:i])

    FI;

  ELSE

    IF \*merging-masking\* ; merging-masking

      THEN \*DEST[i+31:i] remains unchanged\*

    ELSE ; zeroing-masking

      DEST[i+31:i] ← 0

    FI;

  FI;

ENDFOR;

Table 5-5. Special Values Behavior

Source Input	Result	Comments
NaN	QNaN(src)	If (SRC = SNaN) then #I
$+\infty$	$+\infty$	
+/-0	1.0f	<i>Exact result</i>
$-\infty$	+0.0f	
Integral value N	$2^N$	<i>Exact result</i>

**Intel C/C++ Compiler Intrinsic Equivalent**

VEXP2PS \_\_m512 \_\_mm512\_exp2a23\_round\_ps (\_\_m512 a, int sae);

VEXP2PS \_\_m512 \_\_mm512\_mask\_exp2a23\_round\_ps (\_\_m512 a, \_\_mmask16 m, \_\_m512 b, int sae);

VEXP2PS \_\_m512 \_\_mm512\_maskz\_exp2a23\_round\_ps (\_\_mmask16 m, \_\_m512 b, int sae);

**SIMD Floating-Point Exceptions**

Invalid (if SNaN input), Overflow

**Other Exceptions**

See Exceptions Type E2.

## VEEXTRACTF128/VEEXTRACTF32x4/VEEXTRACTF64x2/VEEXTRACTF32x8/VEEXTRACTF64x4—Extract Packed Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.256.66.0F3A.W0 19 /r ib VEEXTRACTF128 xmm1/m128, ymm2, imm8	RMI	V/V	AVX	Extract 128 bits of packed floating-point values from ymm2 and store results in xmm1/m128.
EVEEX.256.66.0F3A.W0 19 /r ib VEEXTRACTF32X4 xmm1/m128 {k1}{z}, ymm2, imm8	T4	V/V	AVX512VL AVX512F	Extract 128 bits of packed single-precision floating-point values from ymm2 and store results in xmm1/m128 subject to writemask k1.
EVEEX.512.66.0F3A.W0 19 /r ib VEEXTRACTF32x4 xmm1/m128 {k1}{z}, zmm2, imm8	T4	V/V	AVX512F	Extract 128 bits of packed single-precision floating-point values from zmm2 and store results in xmm1/m128 subject to writemask k1.
EVEEX.256.66.0F3A.W1 19 /r ib VEEXTRACTF64X2 xmm1/m128 {k1}{z}, ymm2, imm8	T2	V/V	AVX512VL AVX512DQ	Extract 128 bits of packed double-precision floating-point values from ymm2 and store results in xmm1/m128 subject to writemask k1.
EVEEX.512.66.0F3A.W1 19 /r ib VEEXTRACTF64X2 xmm1/m128 {k1}{z}, zmm2, imm8	T2	V/V	AVX512DQ	Extract 128 bits of packed double-precision floating-point values from zmm2 and store results in xmm1/m128 subject to writemask k1.
EVEEX.512.66.0F3A.W0 1B /r ib VEEXTRACTF32X8 ymm1/m256 {k1}{z}, zmm2, imm8	T8	V/V	AVX512DQ	Extract 256 bits of packed single-precision floating-point values from zmm2 and store results in ymm1/m256 subject to writemask k1.
EVEEX.512.66.0F3A.W1 1B /r ib VEEXTRACTF64x4 ymm1/m256 {k1}{z}, zmm2, imm8	T4	V/V	AVX512F	Extract 256 bits of packed double-precision floating-point values from zmm2 and store results in ymm1/m256 subject to writemask k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:r/m (w)	ModRM:reg (r)	Imm8	NA
T2, T4, T8	ModRM:r/m (w)	ModRM:reg (r)	Imm8	NA

### Description

VEEXTRACTF128/VEEXTRACTF32x4 and VEEXTRACTF64x2 extract 128-bits of single-precision floating-point values from the source operand (the second operand) and store to the low 128-bit of the destination operand (the first operand). The 128-bit data extraction occurs at an 128-bit granular offset specified by imm8[0] (256-bit) or imm8[1:0] as the multiply factor. The destination may be either a vector register or an 128-bit memory location.

VEEXTRACTF32x4: The low 128-bit of the destination operand is updated at 32-bit granularity according to the writemask.

VEEXTRACTF32x8 and VEEXTRACTF64x4 extract 256-bits of double-precision floating-point values from the source operand (second operand) and store to the low 256-bit of the destination operand (the first operand). The 256-bit data extraction occurs at an 256-bit granular offset specified by imm8[0] (256-bit) or imm8[0] as the multiply factor. The destination may be either a vector register or a 256-bit memory location.

VEEXTRACTF64x4: The low 256-bit of the destination operand is updated at 64-bit granularity according to the writemask.

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

The high 6 bits of the immediate are ignored.

If VEXTRACTF128 is encoded with VEX.L= 0, an attempt to execute the instruction encoded with VEX.L= 0 will cause an #UD exception.

### Operation

#### VEXTRACTF32x4 (EVEX encoded versions) when destination is a register

VL = 256, 512

IF VL = 256

CASE (imm8[0]) OF

0: TMP\_DEST[127:0] ← SRC1[127:0]

1: TMP\_DEST[127:0] ← SRC1[255:128]

ESAC.

FI;

IF VL = 512

CASE (imm8[1:0]) OF

00: TMP\_DEST[127:0] ← SRC1[127:0]

01: TMP\_DEST[127:0] ← SRC1[255:128]

10: TMP\_DEST[127:0] ← SRC1[383:256]

11: TMP\_DEST[127:0] ← SRC1[511:384]

ESAC.

FI;

FOR j ← 0 TO 3

i ← j \* 32

IF k1[j] OR \*no writemask\*

THEN DEST[i+31:i] ← TMP\_DEST[i+31:i]

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+31:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

ENDFOR

DEST[MAX\_VL-1:128] ← 0

#### VEXTRACTF32x4 (EVEX encoded versions) when destination is memory

VL = 256, 512

IF VL = 256

CASE (imm8[0]) OF

0: TMP\_DEST[127:0] ← SRC1[127:0]

1: TMP\_DEST[127:0] ← SRC1[255:128]

ESAC.

FI;

IF VL = 512

CASE (imm8[1:0]) OF

00: TMP\_DEST[127:0] ← SRC1[127:0]

01: TMP\_DEST[127:0] ← SRC1[255:128]

10: TMP\_DEST[127:0] ← SRC1[383:256]

11: TMP\_DEST[127:0] ← SRC1[511:384]

ESAC.

FI;

```

FOR j ← 0 TO 3
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
    ELSE *DEST[i+31:i] remains unchanged* ; merging-masking
  FI;
ENDFOR

```

**VEXTRACTF64x2 (EVEX encoded versions) when destination is a register**

VL = 256, 512

IF VL = 256

```

CASE (imm8[0]) OF
  0: TMP_DEST[127:0] ← SRC1[127:0]
  1: TMP_DEST[127:0] ← SRC1[255:128]
ESAC.

```

FI;

IF VL = 512

```

CASE (imm8[1:0]) OF
  00: TMP_DEST[127:0] ← SRC1[127:0]
  01: TMP_DEST[127:0] ← SRC1[255:128]
  10: TMP_DEST[127:0] ← SRC1[383:256]
  11: TMP_DEST[127:0] ← SRC1[511:384]
ESAC.

```

FI;

FOR j ← 0 TO 1

```

i ← j * 64
IF k1[j] OR *no writemask*
  THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+63:i] remains unchanged*
    ELSE *zeroing-masking* ; zeroing-masking
      DEST[i+63:i] ← 0
    FI
  FI

```

FI;

ENDFOR

DEST[MAX\_VL-1:128] ← 0

**VEXTRACTF64x2 (EVEX encoded versions) when destination is memory**

VL = 256, 512

IF VL = 256

```

CASE (imm8[0]) OF
  0: TMP_DEST[127:0] ← SRC1[127:0]
  1: TMP_DEST[127:0] ← SRC1[255:128]
ESAC.

```

FI;

IF VL = 512

```

CASE (imm8[1:0]) OF
  00: TMP_DEST[127:0] ← SRC1[127:0]
  01: TMP_DEST[127:0] ← SRC1[255:128]
  10: TMP_DEST[127:0] ← SRC1[383:256]
  11: TMP_DEST[127:0] ← SRC1[511:384]
ESAC.

```

```

FI;
FOR j ← 0 TO 1
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
    ELSE *DEST[i+63:i] remains unchanged*      ; merging-masking
  FI;
ENDFOR

```

#### **VEXTRACTF32x8 (EVEX.U1.512 encoded version) when destination is a register**

```

VL = 512
CASE (imm8[0]) OF
  0: TMP_DEST[255:0] ← SRC1[255:0]
  1: TMP_DEST[255:0] ← SRC1[511:256]
ESAC.

```

```

FOR j ← 0 TO 7
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
    ELSE
      IF *merging-masking*      ; merging-masking
        THEN *DEST[i+31:i] remains unchanged*
        ELSE *zeroing-masking*  ; zeroing-masking
          DEST[i+31:i] ← 0
      FI
    FI;
ENDFOR
DEST[MAX_VL-1:256] ← 0

```

#### **VEXTRACTF32x8 (EVEX.U1.512 encoded version) when destination is memory**

```

CASE (imm8[0]) OF
  0: TMP_DEST[255:0] ← SRC1[255:0]
  1: TMP_DEST[255:0] ← SRC1[511:256]
ESAC.

FOR j ← 0 TO 7
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
    ELSE *DEST[i+31:i] remains unchanged*      ; merging-masking
  FI;
ENDFOR

```

#### **VEXTRACTF64x4 (EVEX.512 encoded version) when destination is a register**

```

VL = 512
CASE (imm8[0]) OF
  0: TMP_DEST[255:0] ← SRC1[255:0]
  1: TMP_DEST[255:0] ← SRC1[511:256]
ESAC.

```

```

FOR j ← 0 TO 3
  i ← j * 64

```

```

IF k1[j] OR *no writemask*
  THEN DEST[j+63:i] ← TMP_DEST[j+63:i]
  ELSE
    IF *merging-masking*           ; merging-masking
      THEN *DEST[j+63:i] remains unchanged*
    ELSE *zeroing-masking*         ; zeroing-masking
      DEST[j+63:i] ← 0
    FI
  FI;
ENDFOR
DEST[MAX_VL-1:256] ← 0

```

**VEXTRACTF64x4 (EVEX.512 encoded version) when destination is memory**

```

CASE (imm8[0]) OF
  0: TMP_DEST[255:0] ← SRC1[255:0]
  1: TMP_DEST[255:0] ← SRC1[511:256]
ESAC.

```

```

FOR j ← 0 TO 3
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[j+63:i] ← TMP_DEST[j+63:i]
    ELSE           ; merging-masking
      *DEST[j+63:i] remains unchanged*
  FI;
ENDFOR

```

**VEXTRACTF128 (memory destination form)**

```

CASE (imm8[0]) OF
  0: DEST[127:0] ← SRC1[127:0]
  1: DEST[127:0] ← SRC1[255:128]
ESAC.

```

**VEXTRACTF128 (register destination form)**

```

CASE (imm8[0]) OF
  0: DEST[127:0] ← SRC1[127:0]
  1: DEST[127:0] ← SRC1[255:128]
ESAC.
DEST[MAX_VL-1:128] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VEXTRACTF32x4 __m128 _mm512_extractf32x4_ps(__m512 a, const int nid);
VEXTRACTF32x4 __m128 _mm512_mask_extractf32x4_ps(__m128 s, __mmask8 k, __m512 a, const int nid);
VEXTRACTF32x4 __m128 _mm512_maskz_extractf32x4_ps(__mmask8 k, __m512 a, const int nid);
VEXTRACTF32x4 __m128 _mm256_extractf32x4_ps(__m256 a, const int nid);
VEXTRACTF32x4 __m128 _mm256_mask_extractf32x4_ps(__m128 s, __mmask8 k, __m256 a, const int nid);
VEXTRACTF32x4 __m128 _mm256_maskz_extractf32x4_ps(__mmask8 k, __m256 a, const int nid);
VEXTRACTF32x8 __m256 _mm512_extractf32x8_ps(__m512 a, const int nid);
VEXTRACTF32x8 __m256 _mm512_mask_extractf32x8_ps(__m256 s, __mmask8 k, __m512 a, const int nid);
VEXTRACTF32x8 __m256 _mm512_maskz_extractf32x8_ps(__mmask8 k, __m512 a, const int nid);
VEXTRACTF64x2 __m128d _mm512_extractf64x2_pd(__m512d a, const int nid);
VEXTRACTF64x2 __m128d _mm512_mask_extractf64x2_pd(__m128d s, __mmask8 k, __m512d a, const int nid);
VEXTRACTF64x2 __m128d _mm512_maskz_extractf64x2_pd(__mmask8 k, __m512d a, const int nid);
VEXTRACTF64x2 __m128d _mm256_extractf64x2_pd(__m256d a, const int nid);

```



VEXTRACTF64x2 \_\_m128d \_\_mm256\_mask\_extractf64x2\_pd(\_\_m128d s, \_\_mmask8 k, \_\_m256d a, const int nidx);  
 VEXTRACTF64x2 \_\_m128d \_\_mm256\_maskz\_extractf64x2\_pd(\_\_mmask8 k, \_\_m256d a, const int nidx);  
 VEXTRACTF64x4 \_\_m256d \_\_mm512\_extractf64x4\_pd(\_\_m512d a, const int nidx);  
 VEXTRACTF64x4 \_\_m256d \_\_mm512\_mask\_extractf64x4\_pd(\_\_m256d s, \_\_mmask8 k, \_\_m512d a, const int nidx);  
 VEXTRACTF64x4 \_\_m256d \_\_mm512\_maskz\_extractf64x4\_pd(\_\_mmask8 k, \_\_m512d a, const int nidx);  
 VEXTRACTF128 \_\_m128 \_\_mm256\_extractf128\_ps(\_\_m256 a, int offset);  
 VEXTRACTF128 \_\_m128d \_\_mm256\_extractf128\_pd(\_\_m256d a, int offset);  
 VEXTRACTF128 \_\_m128i \_\_mm256\_extractf128\_si256(\_\_m256i a, int offset);

### SIMD Floating-Point Exceptions

None

### Other Exceptions

VEX-encoded instructions, see Exceptions Type 6;

EVEX-encoded instructions, see Exceptions Type E6NF.

#UD IF VEX.L = 0.

#UD If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

## VEXTRACTI128/VEXTRACTI32x4/VEXTRACTI64x2/VEXTRACTI32x8/VEXTRACTI64x4—Extract packed Integer Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.256.66.0F3A.W0 39 /r ib VEXTRACTI128 xmm1/m128, ymm2, imm8	RMI	V/V	AVX2	Extract 128 bits of integer data from ymm2 and store results in xmm1/m128.
EVEX.256.66.0F3A.W0 39 /r ib VEXTRACTI32X4 xmm1/m128 {k1}{z}, ymm2, imm8	T4	V/V	AVX512VL AVX512F	Extract 128 bits of double-word integer values from ymm2 and store results in xmm1/m128 subject to writemask k1.
EVEX.512.66.0F3A.W0 39 /r ib VEXTRACTI32x4 xmm1/m128 {k1}{z}, zmm2, imm8	T4	V/V	AVX512F	Extract 128 bits of double-word integer values from zmm2 and store results in xmm1/m128 subject to writemask k1.
EVEX.256.66.0F3A.W1 39 /r ib VEXTRACTI64X2 xmm1/m128 {k1}{z}, ymm2, imm8	T2	V/V	AVX512VL AVX512DQ	Extract 128 bits of quad-word integer values from ymm2 and store results in xmm1/m128 subject to writemask k1.
EVEX.512.66.0F3A.W1 39 /r ib VEXTRACTI64X2 xmm1/m128 {k1}{z}, zmm2, imm8	T2	V/V	AVX512DQ	Extract 128 bits of quad-word integer values from zmm2 and store results in xmm1/m128 subject to writemask k1.
EVEX.512.66.0F3A.W0 3B /r ib VEXTRACTI32X8 ymm1/m256 {k1}{z}, zmm2, imm8	T8	V/V	AVX512DQ	Extract 256 bits of double-word integer values from zmm2 and store results in ymm1/m256 subject to writemask k1.
EVEX.512.66.0F3A.W1 3B /r ib VEXTRACTI64x4 ymm1/m256 {k1}{z}, zmm2, imm8	T4	V/V	AVX512F	Extract 256 bits of quad-word integer values from zmm2 and store results in ymm1/m256 subject to writemask k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:r/m (w)	ModRM:reg (r)	Imm8	NA
T2, T4, T8	ModRM:r/m (w)	ModRM:reg (r)	Imm8	NA

### Description

VEXTRACTI128/VEXTRACTI32x4 and VEXTRACTI64x2 extract 128-bits of doubleword integer values from the source operand (the second operand) and store to the low 128-bit of the destination operand (the first operand). The 128-bit data extraction occurs at an 128-bit granular offset specified by imm8[0] (256-bit) or imm8[1:0] as the multiply factor. The destination may be either a vector register or an 128-bit memory location.

VEXTRACTI32x4: The low 128-bit of the destination operand is updated at 32-bit granularity according to the writemask.

VEXTRACTI64x2: The low 128-bit of the destination operand is updated at 64-bit granularity according to the writemask.

VEXTRACTI32x8 and VEXTRACTI64x4 extract 256-bits of quadword integer values from the source operand (the second operand) and store to the low 256-bit of the destination operand (the first operand). The 256-bit data extraction occurs at an 256-bit granular offset specified by imm8[0] (256-bit) or imm8[0] as the multiply factor. The destination may be either a vector register or a 256-bit memory location.

**VEXTRACTI32x8:** The low 256-bit of the destination operand is updated at 32-bit granularity according to the writemask.

**VEXTRACTI64x4:** The low 256-bit of the destination operand is updated at 64-bit granularity according to the writemask.

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

The high 7 bits (6 bits in EVEX.512) of the immediate are ignored.

If VEXTRACTI128 is encoded with VEX.L= 0, an attempt to execute the instruction encoded with VEX.L= 0 will cause an #UD exception.

### Operation

#### **VEXTRACTI32x4 (EVEX encoded versions) when destination is a register**

VL = 256, 512

IF VL = 256

```
CASE (imm8[0]) OF
  0: TMP_DEST[127:0] ← SRC1[127:0]
  1: TMP_DEST[127:0] ← SRC1[255:128]
ESAC.
```

FI;

IF VL = 512

```
CASE (imm8[1:0]) OF
  00: TMP_DEST[127:0] ← SRC1[127:0]
  01: TMP_DEST[127:0] ← SRC1[255:128]
  10: TMP_DEST[127:0] ← SRC1[383:256]
  11: TMP_DEST[127:0] ← SRC1[511:384]
ESAC.
```

FI;

FOR j ← 0 TO 3

i ← j \* 32

IF k1[j] OR \*no writemask\*

THEN DEST[i+31:i] ← TMP\_DEST[i+31:i]

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+31:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

ENDFOR

DEST[MAX\_VL-1:128] ← 0

#### **VEXTRACTI32x4 (EVEX encoded versions) when destination is memory**

VL = 256, 512

IF VL = 256

```
CASE (imm8[0]) OF
  0: TMP_DEST[127:0] ← SRC1[127:0]
  1: TMP_DEST[127:0] ← SRC1[255:128]
ESAC.
```

FI;

IF VL = 512

```
CASE (imm8[1:0]) OF
  00: TMP_DEST[127:0] ← SRC1[127:0]
  01: TMP_DEST[127:0] ← SRC1[255:128]
  10: TMP_DEST[127:0] ← SRC1[383:256]
```

```

    11: TMP_DEST[127:0] ← SRC1[511:384]
  ESAC.
FI;

FOR j ← 0 TO 3
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
    ELSE *DEST[i+31:i] remains unchanged*      ; merging-masking
  FI;
ENDFOR

```

**VEEXTRACTI64x2 (EVEX encoded versions) when destination is a register**

VL = 256, 512

IF VL = 256

```

CASE (imm8[0]) OF
  0: TMP_DEST[127:0] ← SRC1[127:0]
  1: TMP_DEST[127:0] ← SRC1[255:128]
ESAC.

```

FI;

IF VL = 512

```

CASE (imm8[1:0]) OF
  00: TMP_DEST[127:0] ← SRC1[127:0]
  01: TMP_DEST[127:0] ← SRC1[255:128]
  10: TMP_DEST[127:0] ← SRC1[383:256]
  11: TMP_DEST[127:0] ← SRC1[511:384]
ESAC.

```

FI;

FOR j ← 0 TO 1

```

  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
    ELSE
      IF *merging-masking*      ; merging-masking
        THEN *DEST[i+63:i] remains unchanged*
      ELSE *zeroing-masking*    ; zeroing-masking
        DEST[i+63:i] ← 0
    FI
  FI;

```

```

  FI;
ENDFOR

```

DEST[MAX\_VL-1:128] ← 0

**VEEXTRACTI64x2 (EVEX encoded versions) when destination is memory**

VL = 256, 512

IF VL = 256

```

CASE (imm8[0]) OF
  0: TMP_DEST[127:0] ← SRC1[127:0]
  1: TMP_DEST[127:0] ← SRC1[255:128]
ESAC.

```

FI;

IF VL = 512

```

CASE (imm8[1:0]) OF
  00: TMP_DEST[127:0] ← SRC1[127:0]

```

```

    01: TMP_DEST[127:0] ← SRC1[255:128]
    10: TMP_DEST[127:0] ← SRC1[383:256]
    11: TMP_DEST[127:0] ← SRC1[511:384]
  ESAC.
  FI;

  FOR j ← 0 TO 1
    i ← j * 64
    IF k1[j] OR *no writemask*
      THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
      ELSE *DEST[i+63:i] remains unchanged*      ; merging-masking
    FI;
  ENDFOR

```

#### **VEXTRACTI32x8 (EVEX.U1.512 encoded version) when destination is a register**

```

  VL = 512
  CASE (imm8[0]) OF
    0: TMP_DEST[255:0] ← SRC1[255:0]
    1: TMP_DEST[255:0] ← SRC1[511:256]
  ESAC.

  FOR j ← 0 TO 7
    i ← j * 32
    IF k1[j] OR *no writemask*
      THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
      ELSE
        IF *merging-masking*      ; merging-masking
          THEN *DEST[i+31:i] remains unchanged*
          ELSE *zeroing-masking*  ; zeroing-masking
            DEST[i+31:i] ← 0
        FI
      FI;
  ENDFOR
  DEST[MAX_VL-1:256] ← 0

```

#### **VEXTRACTI32x8 (EVEX.U1.512 encoded version) when destination is memory**

```

  CASE (imm8[0]) OF
    0: TMP_DEST[255:0] ← SRC1[255:0]
    1: TMP_DEST[255:0] ← SRC1[511:256]
  ESAC.

  FOR j ← 0 TO 7
    i ← j * 32
    IF k1[j] OR *no writemask*
      THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
      ELSE *DEST[i+31:i] remains unchanged*      ; merging-masking
    FI;
  ENDFOR

```

#### **VEXTRACTI64x4 (EVEX.512 encoded version) when destination is a register**

```

  VL = 512
  CASE (imm8[0]) OF
    0: TMP_DEST[255:0] ← SRC1[255:0]
    1: TMP_DEST[255:0] ← SRC1[511:256]

```

ESAC.

```

FOR j ← 0 TO 3
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
  ELSE
    IF *merging-masking*           ; merging-masking
      THEN *DEST[i+63:i] remains unchanged*
    ELSE *zeroing-masking*         ; zeroing-masking
      DEST[i+63:i] ← 0
    FI
  FI;
ENDFOR
DEST[MAX_VL-1:256] ← 0

```

#### **VEEXTRACTI64x4 (EVEX.512 encoded version) when destination is memory**

```

CASE (imm8[0]) OF
  0: TMP_DEST[255:0] ← SRC1[255:0]
  1: TMP_DEST[255:0] ← SRC1[511:256]
ESAC.
FOR j ← 0 TO 3
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
  ELSE *DEST[i+63:i] remains unchanged* ; merging-masking
  FI;
ENDFOR

```

#### **VEEXTRACTI128 (memory destination form)**

```

CASE (imm8[0]) OF
  0: DEST[127:0] ← SRC1[127:0]
  1: DEST[127:0] ← SRC1[255:128]
ESAC.

```

#### **VEEXTRACTI128 (register destination form)**

```

CASE (imm8[0]) OF
  0: DEST[127:0] ← SRC1[127:0]
  1: DEST[127:0] ← SRC1[255:128]
ESAC.
DEST[MAX_VL-1:128] ← 0

```

#### **Intel C/C++ Compiler Intrinsic Equivalent**

```

VEEXTRACTI32x4 __m128i_mm512_extracti32x4_epi32(__m512i a, const int nid);
VEEXTRACTI32x4 __m128i_mm512_mask_extracti32x4_epi32(__m128i s, __mmask8 k, __m512i a, const int nid);
VEEXTRACTI32x4 __m128i_mm512_maskz_extracti32x4_epi32(__mmask8 k, __m512i a, const int nid);
VEEXTRACTI32x4 __m128i_mm256_extracti32x4_epi32(__m256i a, const int nid);
VEEXTRACTI32x4 __m128i_mm256_mask_extracti32x4_epi32(__m128i s, __mmask8 k, __m256i a, const int nid);
VEEXTRACTI32x4 __m128i_mm256_maskz_extracti32x4_epi32(__mmask8 k, __m256i a, const int nid);
VEEXTRACTI32x8 __m256i_mm512_extracti32x8_epi32(__m512i a, const int nid);
VEEXTRACTI32x8 __m256i_mm512_mask_extracti32x8_epi32(__m256i s, __mmask8 k, __m512i a, const int nid);
VEEXTRACTI32x8 __m256i_mm512_maskz_extracti32x8_epi32(__mmask8 k, __m512i a, const int nid);
VEEXTRACTI64x2 __m128i_mm512_extracti64x2_epi64(__m512i a, const int nid);
VEEXTRACTI64x2 __m128i_mm512_mask_extracti64x2_epi64(__m128i s, __mmask8 k, __m512i a, const int nid);

```

```

VEXTRACTI64x2 __m128i __mm512_maskz_extracti64x2_epi64( __mmask8 k, __m512i a, const int nidx);
VEXTRACTI64x2 __m128i __mm256_extracti64x2_epi64(__m256i a, const int nidx);
VEXTRACTI64x2 __m128i __mm256_mask_extracti64x2_epi64(__m128i s, __mmask8 k, __m256i a, const int nidx);
VEXTRACTI64x2 __m128i __mm256_maskz_extracti64x2_epi64( __mmask8 k, __m256i a, const int nidx);
VEXTRACTI64x4 __m256i __mm512_extracti64x4_epi64(__m512i a, const int nidx);
VEXTRACTI64x4 __m256i __mm512_mask_extracti64x4_epi64(__m256i s, __mmask8 k, __m512i a, const int nidx);
VEXTRACTI64x4 __m256i __mm512_maskz_extracti64x4_epi64( __mmask8 k, __m512i a, const int nidx);
VEXTRACTI128 __m128i __mm256_extracti128_si256(__m256i a, int offset);

```

### SIMD Floating-Point Exceptions

None

### Other Exceptions

VEX-encoded instructions, see Exceptions Type 6;

EVEX-encoded instructions, see Exceptions Type E6NF.

#UD IF VEX.L = 0.

#UD If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

## VFIXUPIMMPD—Fix Up Special Packed Float64 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.128.66.0F3A.W1 54 /r ib VFIXUPIMMPD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst, imm8	FV	V/V	AVX512VL AVX512F	Fix up special numbers in float64 vector xmm1, float64 vector xmm2 and int64 vector xmm3/m128/m64bcst and store the result in xmm1, under writemask.
EVEX.NDS.256.66.0F3A.W1 54 /r ib VFIXUPIMMPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst, imm8	FV	V/V	AVX512VL AVX512F	Fix up special numbers in float64 vector ymm1, float64 vector ymm2 and int64 vector ymm3/m256/m64bcst and store the result in ymm1, under writemask.
EVEX.NDS.512.66.0F3A.W1 54 /r ib VFIXUPIMMPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{sae}, imm8	FV	V/V	AVX512F	Fix up elements of float64 vector in zmm2 using int64 vector table in zmm3/m512/m64bcst, combine with preserved elements from zmm1, and store the result in zmm1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
FV	ModRM:reg (r, w)	EVEX.vvvv	ModRM:r/m (r)	Imm8

### Description

Perform fix-up of quad-word elements encoded in double-precision floating-point format in the first source operand (the second operand) using a 32-bit, two-level look-up table specified in the corresponding quadword element of the second source operand (the third operand) with exception reporting specifier imm8. The elements that are fixed-up are selected by mask bits of 1 specified in the opmask k1. Mask bits of 0 in the opmask k1 or table response action of 0000b preserves the corresponding element of the first operand. The fixed-up elements from the first source operand and the preserved element in the first operand are combined as the final results in the destination operand (the first operand).

The destination and the first source operands are ZMM/YMM/XMM registers. 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 64-bit memory location.

The two-level look-up table perform a fix-up of each DP FP input data in the first source operand by decoding the input data encoding into 8 token types. A response table is defined for each token type that converts the input encoding in the first source operand with one of 16 response actions.

This instruction is specifically intended for use in fixing up the results of arithmetic calculations involving one source so that they match the spec, although it is generally useful for fixing up the results of multiple-instruction sequences to reflect special-number inputs. For example, consider `rcp(0)`. Input 0 to `rcp`, and you should get INF according to the DX10 spec. However, evaluating `rcp` via Newton-Raphson, where  $x \approx \text{approx}(1/0)$ , yields an incorrect result. To deal with this, `VFIXUPIMMPD` can be used after the N-R reciprocal sequence to set the result to the correct value (i.e. INF when the input is 0).

If `MXCSR.DAZ` is not set, denormal input elements in the first source operand are considered as normal inputs and do not trigger any fixup nor fault reporting.

Imm8 is used to set the required flags reporting. It supports `#ZE` and `#IE` fault reporting (see details below).

MXCSR mask bits are ignored and are treated as if all mask bits are set to masked response). If any of the imm8 bits is set and the condition met for fault reporting, `MXCSR.IE` or `MXCSR.ZE` might be updated.

This instruction is writemasked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1. Elements in the destination with the corresponding bit clear in k1 retain their previous values or are set to 0.



**Operation**

```
enum TOKEN_TYPE
```

```
{
  QNAN_TOKEN ← 0,
  SNAN_TOKEN ← 1,
  ZERO_VALUE_TOKEN ← 2,
  POS_ONE_VALUE_TOKEN ← 3,
  NEG_INF_TOKEN ← 4,
  POS_INF_TOKEN ← 5,
  NEG_VALUE_TOKEN ← 6,
  POS_VALUE_TOKEN ← 7
}
```

```
FIXUPIMM_DP (dest[63:0], src1[63:0], tbl3[63:0], imm8 [7:0]){
  tsrc[63:0] ← ((src1[62:52] = 0) AND (MXCSR.DAZ = 1)) ? 0.0 : src1[63:0]
  CASE(tsrc[63:0] of TOKEN_TYPE) {
    QNAN_TOKEN: j ← 0;
    SNAN_TOKEN: j ← 1;
    ZERO_VALUE_TOKEN: j ← 2;
    POS_ONE_VALUE_TOKEN: j ← 3;
    NEG_INF_TOKEN: j ← 4;
    POS_INF_TOKEN: j ← 5;
    NEG_VALUE_TOKEN: j ← 6;
    POS_VALUE_TOKEN: j ← 7;
  } ; end source special CASE(tsrc...)
```

; The required response from src3 table is extracted

```
token_response[3:0] = tbl3[3+4*j:4*j];
```

```
CASE(token_response[3:0]) {
  0000: dest[63:0] ← dest[63:0]; ; preserve content of DEST
  0001: dest[63:0] ← tsrc[63:0]; ; pass through src1 normal input value, denormal as zero
  0010: dest[63:0] ← QNaN(tsrc[63:0]);
  0011: dest[63:0] ← QNAN_Indefinite;
  0100: dest[63:0] ← -INF;
  0101: dest[63:0] ← +INF;
  0110: dest[63:0] ← tsrc.sign? -INF : +INF;
  0111: dest[63:0] ← -0;
  1000: dest[63:0] ← +0;
  1001: dest[63:0] ← -1;
  1010: dest[63:0] ← +1;
  1011: dest[63:0] ← ½;
  1100: dest[63:0] ← 90.0;
  1101: dest[63:0] ← PI/2;
  1110: dest[63:0] ← MAX_FLOAT;
  1111: dest[63:0] ← -MAX_FLOAT;
} ; end of token_response CASE
```

; The required fault reporting from imm8 is extracted

; TOKENs are mutually exclusive and TOKENs priority defines the order.

; Multiple faults related to a single token can occur simultaneously.

```
IF (tsrc[63:0] of TOKEN_TYPE: ZERO_VALUE_TOKEN) AND imm8[0] then set #ZE;
```

```
IF (tsrc[63:0] of TOKEN_TYPE: ZERO_VALUE_TOKEN) AND imm8[1] then set #IE;
```

```
IF (tsrc[63:0] of TOKEN_TYPE: ONE_VALUE_TOKEN) AND imm8[2] then set #ZE;
```

```

IF (tsrc[63:0] of TOKEN_TYPE: ONE_VALUE_TOKEN) AND imm8[3] then set #IE;
IF (tsrc[63:0] of TOKEN_TYPE: SNAN_TOKEN) AND imm8[4] then set #IE;
IF (tsrc[63:0] of TOKEN_TYPE: NEG_INF_TOKEN) AND imm8[5] then set #IE;
IF (tsrc[63:0] of TOKEN_TYPE: NEG_VALUE_TOKEN) AND imm8[6] then set #IE;
IF (tsrc[63:0] of TOKEN_TYPE: POS_INF_TOKEN) AND imm8[7] then set #IE;
    ; end fault reporting
return dest[63:0];
} ; end of FIXUPIMM_DP()

```

**VFIXUPIMMPD**

(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) AND (SRC2 \*is memory\*)

        THEN

          DEST[i+63:i] ← FIXUPIMM\_DP(DEST[i+63:i], SRC1[i+63:i], SRC2[63:0], imm8 [7:0])

        ELSE

          DEST[i+63:i] ← FIXUPIMM\_DP(DEST[i+63:i], SRC1[i+63:i], SRC2[i+63:i], imm8 [7:0])

      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[MAX\_VL-1:VL] ← 0

Immediate Control Description:

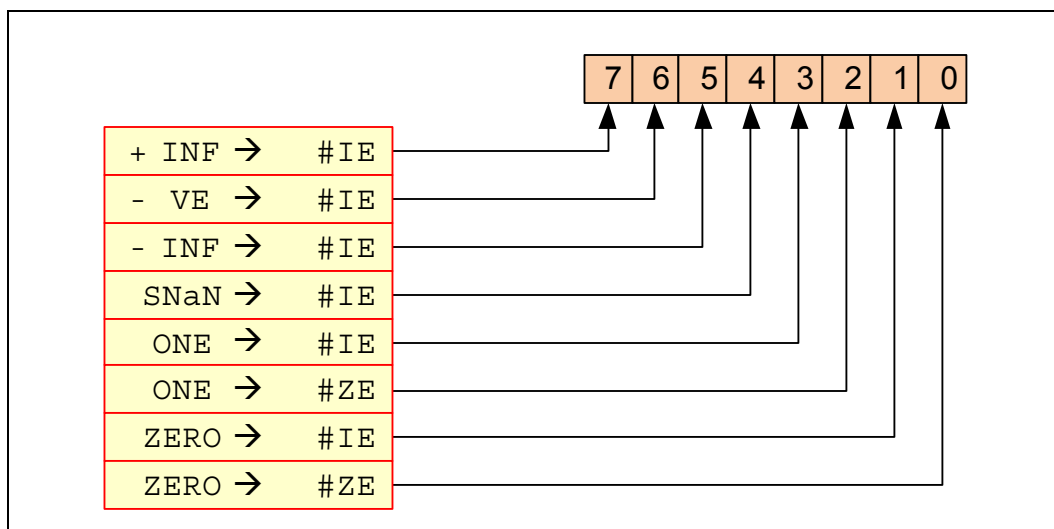


Figure 5-9. VFIXUPIMMPD Immediate Control Description

### Intel C/C++ Compiler Intrinsic Equivalent

```

VFIXUPIMMPD __m512d __mm512_fixupimm_pd( __m512d a, __m512i tbl, int imm);
VFIXUPIMMPD __m512d __mm512_mask_fixupimm_pd(__m512d s, __mmask8 k, __m512d a, __m512i tbl, int imm);
VFIXUPIMMPD __m512d __mm512_maskz_fixupimm_pd( __mmask8 k, __m512d a, __m512i tbl, int imm);
VFIXUPIMMPD __m512d __mm512_fixupimm_round_pd( __m512d a, __m512i tbl, int imm, int sae);
VFIXUPIMMPD __m512d __mm512_mask_fixupimm_round_pd(__m512d s, __mmask8 k, __m512d a, __m512i tbl, int imm, int sae);
VFIXUPIMMPD __m512d __mm512_maskz_fixupimm_round_pd( __mmask8 k, __m512d a, __m512i tbl, int imm, int sae);
VFIXUPIMMPD __m256d __mm256_fixupimm_pd( __m256d a, __m256i tbl, int imm);
VFIXUPIMMPD __m256d __mm256_mask_fixupimm_pd(__m256d s, __mmask8 k, __m256d a, __m256i tbl, int imm);
VFIXUPIMMPD __m256d __mm256_maskz_fixupimm_pd( __mmask8 k, __m256d a, __m256i tbl, int imm);
VFIXUPIMMPD __m128d __mm_fixupimm_pd( __m128d a, __m128i tbl, int imm);
VFIXUPIMMPD __m128d __mm_mask_fixupimm_pd(__m128d s, __mmask8 k, __m128d a, __m128i tbl, int imm);
VFIXUPIMMPD __m128d __mm_maskz_fixupimm_pd( __mmask8 k, __m128d a, __m128i tbl, int imm);

```

### SIMD Floating-Point Exceptions

Zero, Invalid

### Other Exceptions

See Exceptions Type E2.

## VFIXUPIMMPS—Fix Up Special Packed Float32 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.128.66.0F3A.W0 54 /r VFIXUPIMMPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst, imm8	FV	V/V	AVX512VL AVX512F	Fix up special numbers in float32 vector xmm1, float32 vector xmm2 and int32 vector xmm3/m128/m32bcst and store the result in xmm1, under writemask.
EVEX.NDS.256.66.0F3A.W0 54 /r VFIXUPIMMPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst, imm8	FV	V/V	AVX512VL AVX512F	Fix up special numbers in float32 vector ymm1, float32 vector ymm2 and int32 vector ymm3/m256/m32bcst and store the result in ymm1, under writemask.
EVEX.NDS.512.66.0F3A.W0 54 /r ib VFIXUPIMMPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{sae}, imm8	FV	V/V	AVX512F	Fix up elements of float32 vector in zmm2 using int32 vector table in zmm3/m512/m32bcst, combine with preserved elements from zmm1, and store the result in zmm1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
FV	ModRM:reg (r, w)	EVEX.vvvv	ModRM:r/m (r)	Imm8

### Description

Perform fix-up of doubleword elements encoded in single-precision floating-point format in the first source operand (the second operand) using a 32-bit, two-level look-up table specified in the corresponding doubleword element of the second source operand (the third operand) with exception reporting specifier imm8. The elements that are fixed-up are selected by mask bits of 1 specified in the opmask k1. Mask bits of 0 in the opmask k1 or table response action of 0000b preserves the corresponding element of the first operand. The fixed-up elements from the first source operand and the preserved element in the first operand are combined as the final results in the destination operand (the first operand).

The destination and the first source operands are ZMM/YMM/XMM registers. 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 64-bit memory location.

The two-level look-up table perform a fix-up of each SP FP input data in the first source operand by decoding the input data encoding into 8 token types. A response table is defined for each token type that converts the input encoding in the first source operand with one of 16 response actions.

This instruction is specifically intended for use in fixing up the results of arithmetic calculations involving one source so that they match the spec, although it is generally useful for fixing up the results of multiple-instruction sequences to reflect special-number inputs. For example, consider `rcp(0)`. Input 0 to `rcp`, and you should get INF according to the DX10 spec. However, evaluating `rcp` via Newton-Raphson, where  $x = \text{approx}(1/0)$ , yields an incorrect result. To deal with this, `VFIXUPIMMPS` can be used after the N-R reciprocal sequence to set the result to the correct value (i.e. INF when the input is 0).

If `MXCSR.DAZ` is not set, denormal input elements in the first source operand are considered as normal inputs and do not trigger any fixup nor fault reporting.

Imm8 is used to set the required flags reporting. It supports `#ZE` and `#IE` fault reporting (see details below).

`MXCSR.DAZ` is used and refer to zmm2 only (i.e. zmm1 is not considered as zero in case `MXCSR.DAZ` is set).

`MXCSR` mask bits are ignored and are treated as if all mask bits are set to masked response). If any of the imm8 bits is set and the condition met for fault reporting, `MXCSR.IE` or `MXCSR.ZE` might be updated.

**Operation**

```

enum TOKEN_TYPE
{
    QNAN_TOKEN ← 0,
    SNAN_TOKEN ← 1,
    ZERO_VALUE_TOKEN ← 2,
    POS_ONE_VALUE_TOKEN ← 3,
    NEG_INF_TOKEN ← 4,
    POS_INF_TOKEN ← 5,
    NEG_VALUE_TOKEN ← 6,
    POS_VALUE_TOKEN ← 7
}

FIXUPIMM_SP ( dest[31:0], src1[31:0], tbl3[31:0], imm8 [7:0]){
    tsrc[31:0] ← ((src1[30:23] = 0) AND (MXCSR.DAZ = 1)) ? 0.0 : src1[31:0]
    CASE(tsrc[63:0] of TOKEN_TYPE) {
        QNAN_TOKEN: j ← 0;
        SNAN_TOKEN: j ← 1;
        ZERO_VALUE_TOKEN: j ← 2;
        POS_ONE_VALUE_TOKEN: j ← 3;
        NEG_INF_TOKEN: j ← 4;
        POS_INF_TOKEN: j ← 5;
        NEG_VALUE_TOKEN: j ← 6;
        POS_VALUE_TOKEN: j ← 7;
    } ; end source special CASE(tsrc...)

; The required response from src3 table is extracted
token_response[3:0] = tbl3[3+4*j:4*j];

CASE(token_response[3:0]) {
    0000: dest[31:0] ← dest[31:0]; ; preserve content of DEST
    0001: dest[31:0] ← tsrc[31:0]; ; pass through src1 normal input value, denormal as zero
    0010: dest[31:0] ← QNaN(tsrc[31:0]);
    0011: dest[31:0] ← QNAN_Indefinite;
    0100: dest[31:0] ← -INF;
    0101: dest[31:0] ← +INF;
    0110: dest[31:0] ← tsrc.sign? -INF : +INF;
    0111: dest[31:0] ← -0;
    1000: dest[31:0] ← +0;
    1001: dest[31:0] ← -1;
    1010: dest[31:0] ← +1;
    1011: dest[31:0] ← ½;
    1100: dest[31:0] ← 90.0;
    1101: dest[31:0] ← PI/2;
    1110: dest[31:0] ← MAX_FLOAT;
    1111: dest[31:0] ← -MAX_FLOAT;
} ; end of token_response CASE

; The required fault reporting from imm8 is extracted
; TOKENs are mutually exclusive and TOKENs priority defines the order.
; Multiple faults related to a single token can occur simultaneously.
IF (tsrc[31:0] of TOKEN_TYPE: ZERO_VALUE_TOKEN) AND imm8[0] then set #ZE;
IF (tsrc[31:0] of TOKEN_TYPE: ZERO_VALUE_TOKEN) AND imm8[1] then set #IE;
IF (tsrc[31:0] of TOKEN_TYPE: ONE_VALUE_TOKEN) AND imm8[2] then set #ZE;

```

```

IF (tsrc[31:0] of TOKEN_TYPE: ONE_VALUE_TOKEN) AND imm8[3] then set #IE;
IF (tsrc[31:0] of TOKEN_TYPE: SNAN_TOKEN) AND imm8[4] then set #IE;
IF (tsrc[31:0] of TOKEN_TYPE: NEG_INF_TOKEN) AND imm8[5] then set #IE;
IF (tsrc[31:0] of TOKEN_TYPE: NEG_VALUE_TOKEN) AND imm8[6] then set #IE;
IF (tsrc[31:0] of TOKEN_TYPE: POS_INF_TOKEN) AND imm8[7] then set #IE;
    ; end fault reporting
return dest[31:0];
} ; end of FIXUPIMM_SP()

```

**VFIXUPIMMPS (EVEX)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1) AND (SRC2 *is memory*)
        THEN
          DEST[i+31:i] ← FIXUPIMM_SP(DEST[i+31:i], SRC1[i+31:i], SRC2[31:0], imm8 [7:0])
        ELSE
          DEST[i+31:i] ← FIXUPIMM_SP(DEST[i+31:i], SRC1[i+31:i], SRC2[i+31:i], imm8 [7:0])
        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[MAX_VL-1:VL] ← 0

```

Immediate Control Description:

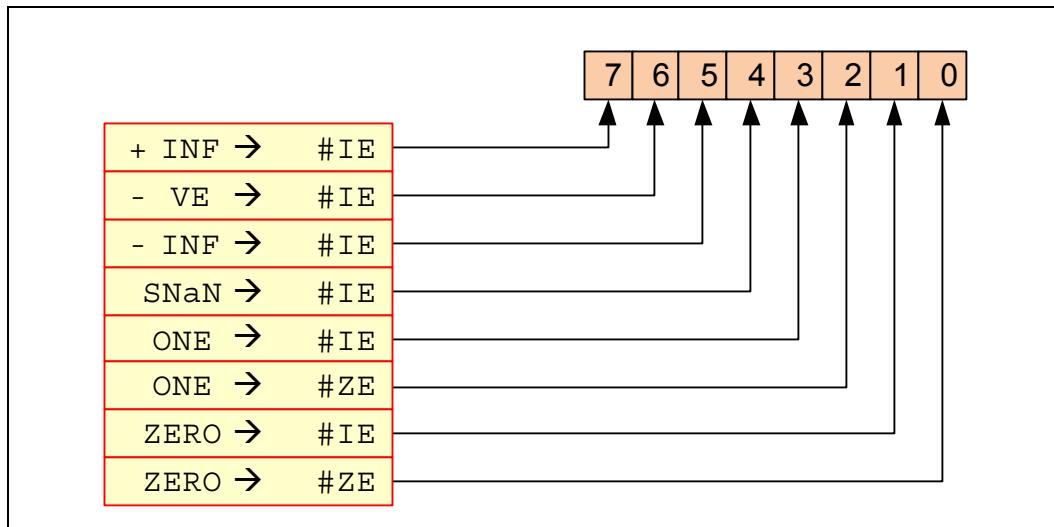


Figure 5-10. VFIXUPIMMPS Immediate Control Description

### Intel C/C++ Compiler Intrinsic Equivalent

```

VFIXUPIMMPS __m512 __mm512_fixupimm_ps( __m512 a, __m512i tbl, int imm);
VFIXUPIMMPS __m512 __mm512_mask_fixupimm_ps(__m512 s, __mmask16 k, __m512 a, __m512i tbl, int imm);
VFIXUPIMMPS __m512 __mm512_maskz_fixupimm_ps( __mmask16 k, __m512 a, __m512i tbl, int imm);
VFIXUPIMMPS __m512 __mm512_fixupimm_round_ps( __m512 a, __m512i tbl, int imm, int sae);
VFIXUPIMMPS __m512 __mm512_mask_fixupimm_round_ps(__m512 s, __mmask16 k, __m512 a, __m512i tbl, int imm, int sae);
VFIXUPIMMPS __m512 __mm512_maskz_fixupimm_round_ps( __mmask16 k, __m512 a, __m512i tbl, int imm, int sae);
VFIXUPIMMPS __m256 __mm256_fixupimm_ps( __m256 a, __m256i tbl, int imm);
VFIXUPIMMPS __m256 __mm256_mask_fixupimm_ps(__m256 s, __mmask8 k, __m256 a, __m256i tbl, int imm);
VFIXUPIMMPS __m256 __mm256_maskz_fixupimm_ps( __mmask8 k, __m256 a, __m256i tbl, int imm);
VFIXUPIMMPS __m128 __mm_fixupimm_ps( __m128 a, __m128i tbl, int imm);
VFIXUPIMMPS __m128 __mm_mask_fixupimm_ps(__m128 s, __mmask8 k, __m128 a, __m128i tbl, int imm);
VFIXUPIMMPS __m128 __mm_maskz_fixupimm_ps( __mmask8 k, __m128 a, __m128i tbl, int imm);

```

### SIMD Floating-Point Exceptions

Zero, Invalid

### Other Exceptions

See Exceptions Type E2.

## VFIXUPIMMSD—Fix Up Special Scalar Float64 Value

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.LIG.66.0F3A.W1 55 /r ib VFIXUPIMMSD xmm1 {k1}{z}, xmm2, xmm3/m64{sae}, imm8	T1S	V/V	AVX512F	Fix up a float64 number in the low quadword element of xmm2 using scalar int32 table in xmm3/m64 and store the result in xmm1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
T1S	ModRM:reg (r, w)	EVEX.vvvv	ModRM:r/m (r)	Imm8

### Description

Perform a fix-up of the low quadword element encoded in double-precision floating-point format in the first source operand (the second operand) using a 32-bit, two-level look-up table specified in the low quadword element of the second source operand (the third operand) with exception reporting specifier imm8. The element that is fixed-up is selected by mask bit of 1 specified in the opmask k1. Mask bit of 0 in the opmask k1 or table response action of 0000b preserves the corresponding element of the first operand. The fixed-up element from the first source operand or the preserved element in the first operand becomes the low quadword element of the destination operand (the first operand). Bits 127:64 of the destination operand is copied from the corresponding bits of the first source operand. The destination and first source operands are XMM registers. The second source operand can be a XMM register or a 64-bit memory location.

The two-level look-up table perform a fix-up of each DP FP input data in the first source operand by decoding the input data encoding into 8 token types. A response table is defined for each token type that converts the input encoding in the first source operand with one of 16 response actions.

This instruction is specifically intended for use in fixing up the results of arithmetic calculations involving one source so that they match the spec, although it is generally useful for fixing up the results of multiple-instruction sequences to reflect special-number inputs. For example, consider `rcp(0)`. Input 0 to `rcp`, and you should get INF according to the DX10 spec. However, evaluating `rcp` via Newton-Raphson, where  $x \approx 1/0$ , yields an incorrect result. To deal with this, `VFIXUPIMMSD` can be used after the N-R reciprocal sequence to set the result to the correct value (i.e. INF when the input is 0).

If `MXCSR.DAZ` is not set, denormal input elements in the first source operand are considered as normal inputs and do not trigger any fixup nor fault reporting.

Imm8 is used to set the required flags reporting. It supports #ZE and #IE fault reporting (see details below).

`MXCSR.DAZ` is used and refer to `zmm2` only (i.e. `zmm1` is not considered as zero in case `MXCSR.DAZ` is set).

`MXCSR` mask bits are ignored and are treated as if all mask bits are set to masked response). If any of the imm8 bits is set and the condition met for fault reporting, `MXCSR.IE` or `MXCSR.ZE` might be updated.

### Operation

```
enum TOKEN_TYPE
{
    QNAN_TOKEN ← 0,
    SNAN_TOKEN ← 1,
    ZERO_VALUE_TOKEN ← 2,
    POS_ONE_VALUE_TOKEN ← 3,
    NEG_INF_TOKEN ← 4,
    POS_INF_TOKEN ← 5,
    NEG_VALUE_TOKEN ← 6,
    POS_VALUE_TOKEN ← 7
}
```



```

FIXUPIMM_DP (dest[63:0], src1[63:0], tbl3[63:0], imm8 [7:0]){
  tsrc[63:0] ← ((src1[62:52] = 0) AND (MXCSR.DAZ = 1)) ? 0.0 : src1[63:0]
  CASE(tsrc[63:0] of TOKEN_TYPE) {
    QNAN_TOKEN: j ← 0;
    SNAN_TOKEN: j ← 1;
    ZERO_VALUE_TOKEN: j ← 2;
    POS_ONE_VALUE_TOKEN: j ← 3;
    NEG_INF_TOKEN: j ← 4;
    POS_INF_TOKEN: j ← 5;
    NEG_VALUE_TOKEN: j ← 6;
    POS_VALUE_TOKEN: j ← 7;
  } ; end source special CASE(tsrc...)

```

; The required response from src3 table is extracted

```
token_response[3:0] = tbl3[3+4*j:4*j];
```

```

CASE(token_response[3:0]) {
  0000: dest[63:0] ← dest[63:0] ; preserve content of DEST
  0001: dest[63:0] ← tsrc[63:0]; ; pass through src1 normal input value, denormal as zero
  0010: dest[63:0] ← QNaN(tsrc[63:0]);
  0011: dest[63:0] ← QNaN_Indefinite;
  0100: dest[63:0] ← -INF;
  0101: dest[63:0] ← +INF;
  0110: dest[63:0] ← tsrc.sign? -INF : +INF;
  0111: dest[63:0] ← -0;
  1000: dest[63:0] ← +0;
  1001: dest[63:0] ← -1;
  1010: dest[63:0] ← +1;
  1011: dest[63:0] ← ½;
  1100: dest[63:0] ← 90.0;
  1101: dest[63:0] ← PI/2;
  1110: dest[63:0] ← MAX_FLOAT;
  1111: dest[63:0] ← -MAX_FLOAT;
} ; end of token_response CASE

```

; The required fault reporting from imm8 is extracted

; TOKENs are mutually exclusive and TOKENs priority defines the order.

; Multiple faults related to a single token can occur simultaneously.

```
IF (tsrc[63:0] of TOKEN_TYPE: ZERO_VALUE_TOKEN) AND imm8[0] then set #ZE;
```

```
IF (tsrc[63:0] of TOKEN_TYPE: ZERO_VALUE_TOKEN) AND imm8[1] then set #IE;
```

```
IF (tsrc[63:0] of TOKEN_TYPE: ONE_VALUE_TOKEN) AND imm8[2] then set #ZE;
```

```
IF (tsrc[63:0] of TOKEN_TYPE: ONE_VALUE_TOKEN) AND imm8[3] then set #IE;
```

```
IF (tsrc[63:0] of TOKEN_TYPE: SNAN_TOKEN) AND imm8[4] then set #IE;
```

```
IF (tsrc[63:0] of TOKEN_TYPE: NEG_INF_TOKEN) AND imm8[5] then set #IE;
```

```
IF (tsrc[63:0] of TOKEN_TYPE: NEG_VALUE_TOKEN) AND imm8[6] then set #IE;
```

```
IF (tsrc[63:0] of TOKEN_TYPE: POS_INF_TOKEN) AND imm8[7] then set #IE;
```

```
 ; end fault reporting
```

```
return dest[63:0];
```

```
} ; end of FIXUPIMM_DP()
```

**VFIXUPIMMSD (EVEX encoded version)**

```

IF k1[0] OR *no writemask*
  THEN DEST[63:0] ← FIXUPIMM_DP(DEST[63:0], SRC1[63:0], SRC2[63:0], imm8 [7:0])
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[63:0] remains unchanged*
      ELSE DEST[63:0] ← 0 ; zeroing-masking
    FI
FI;
DEST[127:64] ← SRC1[127:64]
DEST[MAX_VL-1:128] ← 0

```

Immediate Control Description:

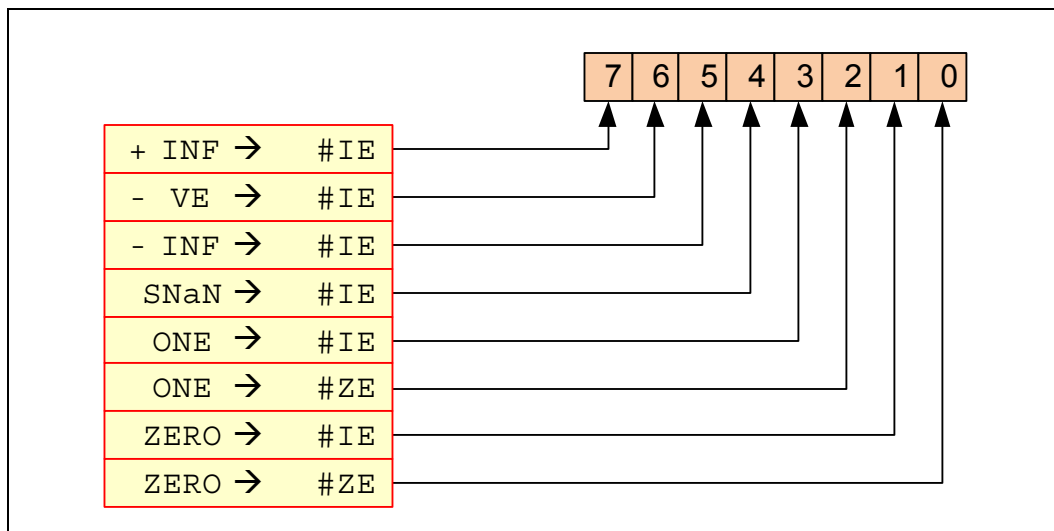


Figure 5-11. VFIXUPIMMSD Immediate Control Description

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VFIXUPIMMSD __m128d __mm_fixupimm_sd(__m128d a, __m128i tbl, int imm);
VFIXUPIMMSD __m128d __mm_mask_fixupimm_sd(__m128d s, __mmask8 k, __m128d a, __m128i tbl, int imm);
VFIXUPIMMSD __m128d __mm_maskz_fixupimm_sd(__mmask8 k, __m128d a, __m128i tbl, int imm);
VFIXUPIMMSD __m128d __mm_fixupimm_round_sd(__m128d a, __m128i tbl, int imm, int sae);
VFIXUPIMMSD __m128d __mm_mask_fixupimm_round_sd(__m128d s, __mmask8 k, __m128d a, __m128i tbl, int imm, int sae);
VFIXUPIMMSD __m128d __mm_maskz_fixupimm_round_sd(__mmask8 k, __m128d a, __m128i tbl, int imm, int sae);

```

**SIMD Floating-Point Exceptions**

Zero, Invalid

**Other Exceptions**

See Exceptions Type E3.

## VFIXUPIMMSS—Fix Up Special Scalar Float32 Value

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.LIG.66.0F3A.W0 55 /r ib VFIXUPIMMSS xmm1 {k1}{z}, xmm2, xmm3/m32{sae}, imm8	T1S	V/V	AVX512F	Fix up a float32 number in the low doubleword element in xmm2 using scalar int32 table in xmm3/m32 and store the result in xmm1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
T1S	ModRM:reg (r, w)	EVEX.vvvv	ModRM:r/m (r)	Imm8

### Description

Perform a fix-up of the low doubleword element encoded in single-precision floating-point format in the first source operand (the second operand) using a 32-bit, two-level look-up table specified in the low doubleword element of the second source operand (the third operand) with exception reporting specifier imm8. The element that is fixed-up is selected by mask bit of 1 specified in the opmask k1. Mask bit of 0 in the opmask k1 or table response action of 0000b preserves the corresponding element of the first operand. The fixed-up element from the first source operand or the preserved element in the first operand becomes the low doubleword element of the destination operand (the first operand) Bits 127:32 of the destination operand is copied from the corresponding bits of the first source operand. The destination and first source operands are XMM registers. The second source operand can be a XMM register or a 32-bit memory location.

The two-level look-up table perform a fix-up of each SP FP input data in the first source operand by decoding the input data encoding into 8 token types. A response table is defined for each token type that converts the input encoding in the first source operand with one of 16 response actions.

This instruction is specifically intended for use in fixing up the results of arithmetic calculations involving one source so that they match the spec, although it is generally useful for fixing up the results of multiple-instruction sequences to reflect special-number inputs. For example, consider `rcp(0)`. Input 0 to `rcp`, and you should get INF according to the DX10 spec. However, evaluating `rcp` via Newton-Raphson, where  $x \approx 1/0$ , yields an incorrect result. To deal with this, `VFIXUPIMMSS` can be used after the N-R reciprocal sequence to set the result to the correct value (i.e. INF when the input is 0).

If `MXCSR.DAZ` is not set, denormal input elements in the first source operand are considered as normal inputs and do not trigger any fixup nor fault reporting.

Imm8 is used to set the required flags reporting. It supports `#ZE` and `#IE` fault reporting (see details below).

`MXCSR.DAZ` is used and refer to `zmm2` only (i.e. `zmm1` is not considered as zero in case `MXCSR.DAZ` is set).

`MXCSR` mask bits are ignored and are treated as if all mask bits are set to masked response). If any of the imm8 bits is set and the condition met for fault reporting, `MXCSR.IE` or `MXCSR.ZE` might be updated.

### Operation

```
enum TOKEN_TYPE
{
    QNAN_TOKEN ← 0,
    SNAN_TOKEN ← 1,
    ZERO_VALUE_TOKEN ← 2,
    POS_ONE_VALUE_TOKEN ← 3,
    NEG_INF_TOKEN ← 4,
    POS_INF_TOKEN ← 5,
    NEG_VALUE_TOKEN ← 6,
    POS_VALUE_TOKEN ← 7
}
```

```

FIXUPIMM_SP (dest[31:0], src1[31:0],tbl3[31:0], imm8 [7:0]){
  tsrc[31:0] ← ((src1[30:23] = 0) AND (MXCSR.DAZ = 1)) ? 0.0 : src1[31:0]
  CASE(tsrc[63:0] of TOKEN_TYPE) {
    QNAN_TOKEN: j ← 0;
    SNAN_TOKEN: j ← 1;
    ZERO_VALUE_TOKEN: j ← 2;
    POS_ONE_VALUE_TOKEN: j ← 3;
    NEG_INF_TOKEN: j ← 4;
    POS_INF_TOKEN: j ← 5;
    NEG_VALUE_TOKEN: j ← 6;
    POS_VALUE_TOKEN: j = 7;
  } ; end source special CASE(tsrc...)

; The required response from src3 table is extracted
token_response[3:0] = tbl3[3+4*j:4*j];

CASE(token_response[3:0]) {
  0000: dest[31:0] ← dest[31:0]; ; preserve content of DEST
  0001: dest[31:0] ← tsrc[31:0]; ; pass through src1 normal input value, denormal as zero
  0010: dest[31:0] ← QNaN(tsrc[31:0]);
  0011: dest[31:0] ← QNaN_Indefinite;
  0100: dest[31:0] ← -INF;
  0101: dest[31:0] ← +INF;
  0110: dest[31:0] ← tsrc.sign? -INF : +INF;
  0111: dest[31:0] ← -0;
  1000: dest[31:0] ← +0;
  1001: dest[31:0] ← -1;
  1010: dest[31:0] ← +1;
  1011: dest[31:0] ← ½;
  1100: dest[31:0] ← 90.0;
  1101: dest[31:0] ← PI/2;
  1110: dest[31:0] ← MAX_FLOAT;
  1111: dest[31:0] ← -MAX_FLOAT;
} ; end of token_response CASE

; The required fault reporting from imm8 is extracted
; TOKENs are mutually exclusive and TOKENs priority defines the order.
; Multiple faults related to a single token can occur simultaneously.
IF (tsrc[31:0] of TOKEN_TYPE: ZERO_VALUE_TOKEN) AND imm8[0] then set #ZE;
IF (tsrc[31:0] of TOKEN_TYPE: ZERO_VALUE_TOKEN) AND imm8[1] then set #IE;
IF (tsrc[31:0] of TOKEN_TYPE: ONE_VALUE_TOKEN) AND imm8[2] then set #ZE;
IF (tsrc[31:0] of TOKEN_TYPE: ONE_VALUE_TOKEN) AND imm8[3] then set #IE;
IF (tsrc[31:0] of TOKEN_TYPE: SNAN_TOKEN) AND imm8[4] then set #IE;
IF (tsrc[31:0] of TOKEN_TYPE: NEG_INF_TOKEN) AND imm8[5] then set #IE;
IF (tsrc[31:0] of TOKEN_TYPE: NEG_VALUE_TOKEN) AND imm8[6] then set #IE;
IF (tsrc[31:0] of TOKEN_TYPE: POS_INF_TOKEN) AND imm8[7] then set #IE;
; end fault reporting
return dest[31:0];
} ; end of FIXUPIMM_SP()

```

**VFIXUPIMMSS (EVEX encoded version)**

```

IF k1[0] OR *no writemask*
  THEN DEST[31:0] ← FIXUPIMM_SP(DEST[31:0], SRC1[31:0], SRC2[31:0], imm8 [7:0])
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[31:0] remains unchanged*
      ELSE DEST[31:0] ← 0 ; zeroing-masking
    FI
  FI;
DEST[127:32] ← SRC1[127:32]
DEST[MAX_VL-1:128] ← 0

```

Immediate Control Description:

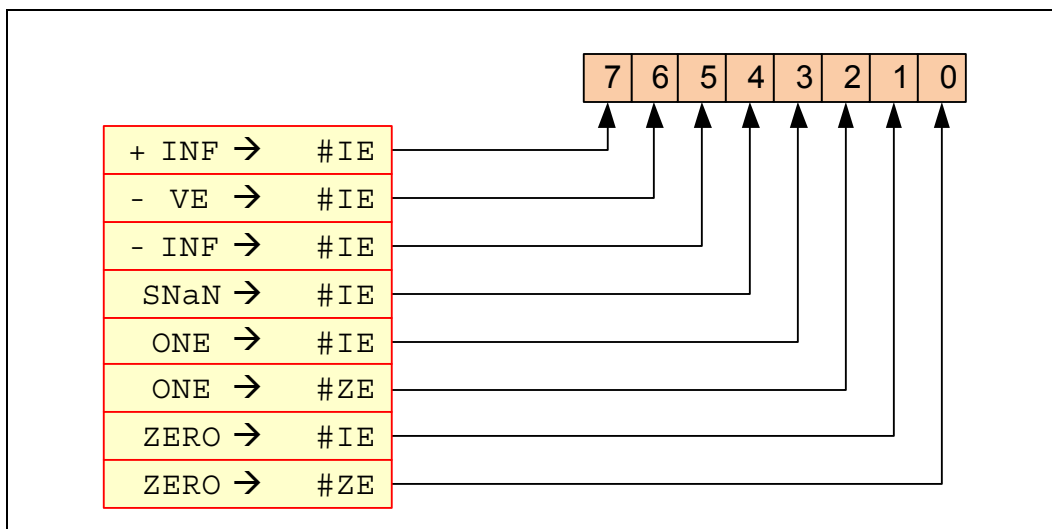


Figure 5-12. VFIXUPIMMSS Immediate Control Description

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VFIXUPIMMSS __m128 __mm_fixupimm_ss(__m128 a, __m128i tbl, int imm);
VFIXUPIMMSS __m128 __mm_mask_fixupimm_ss(__m128 s, __mmask8 k, __m128 a, __m128i tbl, int imm);
VFIXUPIMMSS __m128 __mm_maskz_fixupimm_ss(__mmask8 k, __m128 a, __m128i tbl, int imm);
VFIXUPIMMSS __m128 __mm_fixupimm_round_ss(__m128 a, __m128i tbl, int imm, int sae);
VFIXUPIMMSS __m128 __mm_mask_fixupimm_round_ss(__m128 s, __mmask8 k, __m128 a, __m128i tbl, int imm, int sae);
VFIXUPIMMSS __m128 __mm_maskz_fixupimm_round_ss(__mmask8 k, __m128 a, __m128i tbl, int imm, int sae);

```

**SIMD Floating-Point Exceptions**

Zero, Invalid

**Other Exceptions**

See Exceptions Type E3.

## VFMADD132PD/VFMADD213PD/VFMADD231PD—Fused Multiply-Add of Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.NDS.128.66.0F38.W1 98 /r VFMADD132PD xmm1, xmm2, xmm3/m128	RVM	V/V	FMA	Multiply packed double-precision floating-point values from xmm1 and xmm3/mem, add to xmm2 and put result in xmm1.
VEX.NDS.128.66.0F38.W1 A8 /r VFMADD213PD xmm1, xmm2, xmm3/m128	RVM	V/V	FMA	Multiply packed double-precision floating-point values from xmm1 and xmm2, add to xmm3/mem and put result in xmm1.
VEX.NDS.128.66.0F38.W1 B8 /r VFMADD231PD xmm1, xmm2, xmm3/m128	RVM	V/V	FMA	Multiply packed double-precision floating-point values from xmm2 and xmm3/mem, add to xmm1 and put result in xmm1.
VEX.NDS.256.66.0F38.W1 98 /r VFMADD132PD ymm1, ymm2, ymm3/m256	RVM	V/V	FMA	Multiply packed double-precision floating-point values from ymm1 and ymm3/mem, add to ymm2 and put result in ymm1.
VEX.NDS.256.66.0F38.W1 A8 /r VFMADD213PD ymm1, ymm2, ymm3/m256	RVM	V/V	FMA	Multiply packed double-precision floating-point values from ymm1 and ymm2, add to ymm3/mem and put result in ymm1.
VEX.NDS.256.66.0F38.W1 B8 /r VFMADD231PD ymm1, ymm2, ymm3/m256	RVM	V/V	FMA	Multiply packed double-precision floating-point values from ymm2 and ymm3/mem, add to ymm1 and put result in ymm1.
EVEX.NDS.128.66.0F38.W1 98 /r VFMADD132PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	RVM	V/V	AVX512VL AVX512F	Multiply packed double-precision floating-point values from xmm1 and xmm3/m128/m64bcst, add to xmm2 and put result in xmm1.
EVEX.NDS.128.66.0F38.W1 A8 /r VFMADD213PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	FV	V/V	AVX512VL AVX512F	Multiply packed double-precision floating-point values from xmm1 and xmm2, add to xmm3/m128/m64bcst and put result in xmm1.
EVEX.NDS.128.66.0F38.W1 B8 /r VFMADD231PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	FV	V/V	AVX512VL AVX512F	Multiply packed double-precision floating-point values from xmm2 and xmm3/m128/m64bcst, add to xmm1 and put result in xmm1.
EVEX.NDS.256.66.0F38.W1 98 /r VFMADD132PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	FV	V/V	AVX512VL AVX512F	Multiply packed double-precision floating-point values from ymm1 and ymm3/m256/m64bcst, add to ymm2 and put result in ymm1.
EVEX.NDS.256.66.0F38.W1 A8 /r VFMADD213PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	FV	V/V	AVX512VL AVX512F	Multiply packed double-precision floating-point values from ymm1 and ymm2, add to ymm3/m256/m64bcst and put result in ymm1.
EVEX.NDS.256.66.0F38.W1 B8 /r VFMADD231PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	FV	V/V	AVX512VL AVX512F	Multiply packed double-precision floating-point values from ymm2 and ymm3/m256/m64bcst, add to ymm1 and put result in ymm1.
EVEX.NDS.512.66.0F38.W1 98 /r VFMADD132PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	FV	V/V	AVX512F	Multiply packed double-precision floating-point values from zmm1 and zmm3/m512/m64bcst, add to zmm2 and put result in zmm1.
EVEX.NDS.512.66.0F38.W1 A8 /r VFMADD213PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	FV	V/V	AVX512F	Multiply packed double-precision floating-point values from zmm1 and zmm2, add to zmm3/m512/m64bcst and put result in zmm1.
EVEX.NDS.512.66.0F38.W1 B8 /r VFMADD231PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	FV	V/V	AVX512F	Multiply packed double-precision floating-point values from zmm2 and zmm3/m512/m64bcst, add to zmm1 and put result in zmm1.

## Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RVM	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FV	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

## Description

Performs a set of SIMD multiply-add computation on packed double-precision floating-point values using three source operands and writes the multiply-add results in the destination operand. The destination operand is also the first source operand. The second operand must be a SIMD register. The third source operand can be a SIMD register or a memory location.

**VFMADD132PD:** Multiplies the two, four or eight packed double-precision floating-point values from the first source operand to the two, four or eight packed double-precision floating-point values in the third source operand, adds the infinite precision intermediate result to the two, four or eight packed double-precision floating-point values in the second source operand, performs rounding and stores the resulting two, four or eight packed double-precision floating-point values to the destination operand (first source operand).

**VFMADD213PD:** Multiplies the two, four or eight packed double-precision floating-point values from the second source operand to the two, four or eight packed double-precision floating-point values in the first source operand, adds the infinite precision intermediate result to the two, four or eight packed double-precision floating-point values in the third source operand, performs rounding and stores the resulting two, four or eight packed double-precision floating-point values to the destination operand (first source operand).

**VFMADD231PD:** Multiplies the two, four or eight packed double-precision floating-point values from the second source to the two, four or eight packed double-precision floating-point values in the third source operand, adds the infinite precision intermediate result to the two, four or eight packed double-precision floating-point values in the first source operand, performs rounding and stores the resulting two, four or eight packed double-precision floating-point values to the destination operand (first source operand).

**EVEX encoded versions:** The destination operand (also first source operand) is a ZMM register and encoded in `reg_field`. The second source operand is a ZMM register and encoded in `EVEX.vvvv`. The third source operand is a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 64-bit memory location. The destination operand is conditionally updated with write mask `k1`.

**VEX.256 encoded version:** The destination operand (also first source operand) is a YMM register and encoded in `reg_field`. The second source operand is a YMM register and encoded in `VEX.vvvv`. The third source operand is a YMM register or a 256-bit memory location and encoded in `rm_field`.

**VEX.128 encoded version:** The destination operand (also first source operand) is a XMM register and encoded in `reg_field`. The second source operand is a XMM register and encoded in `VEX.vvvv`. The third source operand is a XMM register or a 128-bit memory location and encoded in `rm_field`. The upper 128 bits of the YMM destination register are zeroed.

## Operation

In the operations below, "\*" and "+" symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding).

**VFMADD132PD DEST, SRC2, SRC3 (VEX encoded version)**

IF (VEX.128) THEN

    MAXNUM ← 2

ELSEIF (VEX.256)

    MAXNUM ← 4

FI

For i = 0 to MAXNUM-1 {

    n ← 64\*i;

    DEST[n+63:n] ← RoundFPControl\_MXCSR(DEST[n+63:n]\*SRC3[n+63:n] + SRC2[n+63:n])

}

```

IF (VEX.128) THEN
    DEST[MAX_VL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAX_VL-1:256] ← 0
FI

```

**VFMADD213PD DEST, SRC2, SRC3 (VEX encoded version)**

```

IF (VEX.128) THEN
    MAXNUM ← 2
ELSEIF (VEX.256)
    MAXNUM ← 4
FI

For i = 0 to MAXNUM-1 {
    n ← 64*i;
    DEST[n+63:n] ← RoundFPControl_MXCSR(SRC2[n+63:n]*DEST[n+63:n] + SRC3[n+63:n])
}

IF (VEX.128) THEN
    DEST[MAX_VL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAX_VL-1:256] ← 0
FI

```

**VFMADD231PD DEST, SRC2, SRC3 (VEX encoded version)**

```

IF (VEX.128) THEN
    MAXNUM ← 2
ELSEIF (VEX.256)
    MAXNUM ← 4
FI

For i = 0 to MAXNUM-1 {
    n ← 64*i;
    DEST[n+63:n] ← RoundFPControl_MXCSR(SRC2[n+63:n]*SRC3[n+63:n] + DEST[n+63:n])
}

IF (VEX.128) THEN
    DEST[MAX_VL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAX_VL-1:256] ← 0
FI

```

**VFMADD132PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

```

IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;

FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ←
            RoundFPControl(DEST[i+63:i]*SRC3[i+63:i] + SRC2[i+63:i])
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*

```



```

        ELSE                                ; zeroing-masking
            DEST[i+63:i] ← 0
        FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VFMAADD132PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 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] ←
                    RoundFPControl_MXCSR(DEST[i+63:i]*SRC3[63:0] + SRC2[i+63:i])
                ELSE
                    DEST[i+63:i] ←
                    RoundFPControl_MXCSR(DEST[i+63:i]*SRC3[i+63:i] + SRC2[i+63:i])
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
            ELSE                                  ; zeroing-masking
                DEST[i+63:i] ← 0
            FI
        FI;
    ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VFMAADD213PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

IF (VL = 512) AND (EVEX.b = 1)

```

    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
    FI;
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ←
        RoundFPControl(SRC2[i+63:i]*DEST[i+63:i] + SRC3[i+63:i])
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[i+63:i] remains unchanged*
        ELSE                                  ; zeroing-masking
            DEST[i+63:i] ← 0
        FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VFMAADD213PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 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] ←
            RoundFPControl_MXCSR(SRC2[i+63:i]*DEST[i+63:i] + SRC3[63:0])
        ELSE
          DEST[i+63:i] ←
            RoundFPControl_MXCSR(SRC2[i+63:i]*DEST[i+63:i] + SRC3[i+63:i])
        FI;
      ELSE
        IF *merging-masking*           ; merging-masking
          THEN *DEST[i+63:i] remains unchanged*
        ELSE                             ; zeroing-masking
          DEST[i+63:i] ← 0
        FI
      FI;
    ENDFOR
  DEST[MAX_VL-1:VL] ← 0

```

**VFMAADD231PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

IF (VL = 512) AND (EVEX.b = 1)

```

  THEN
    SET_RM(EVEX.RC);
  ELSE
    SET_RM(MXCSR.RM);
  FI;
FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] ←
      RoundFPControl(SRC2[i+63:i]*SRC3[i+63:i] + DEST[i+63:i])
    ELSE
      IF *merging-masking*           ; merging-masking
        THEN *DEST[i+63:i] remains unchanged*
      ELSE                             ; zeroing-masking
        DEST[i+63:i] ← 0
      FI
    FI;
  ENDFOR
  DEST[MAX_VL-1:VL] ← 0

```

**VFMAADD231PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 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[j+63:i] ←
      RoundFPControl_MXCSR(SRC2[j+63:i]*SRC3[63:0] + DEST[j+63:i])
    ELSE
      DEST[j+63:i] ←
      RoundFPControl_MXCSR(SRC2[j+63:i]*SRC3[j+63:i] + DEST[j+63:i])
  FI;
ELSE
  IF *merging-masking*           ; merging-masking
    THEN *DEST[j+63:i] remains unchanged*
  ELSE                             ; zeroing-masking
    DEST[j+63:i] ← 0
  FI
FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

### Intel C/C++ Compiler Intrinsic Equivalent

```

VFMADDxxxPD __m512d _mm512_fmadd_pd(__m512d a, __m512d b, __m512d c);
VFMADDxxxPD __m512d _mm512_fmadd_round_pd(__m512d a, __m512d b, __m512d c, int r);
VFMADDxxxPD __m512d _mm512_mask_fmadd_pd(__m512d a, __mmask8 k, __m512d b, __m512d c);
VFMADDxxxPD __m512d _mm512_maskz_fmadd_pd(__mmask8 k, __m512d a, __m512d b, __m512d c);
VFMADDxxxPD __m512d _mm512_mask3_fmadd_pd(__m512d a, __m512d b, __m512d c, __mmask8 k);
VFMADDxxxPD __m512d _mm512_mask_fmadd_round_pd(__m512d a, __mmask8 k, __m512d b, __m512d c, int r);
VFMADDxxxPD __m512d _mm512_maskz_fmadd_round_pd(__mmask8 k, __m512d a, __m512d b, __m512d c, int r);
VFMADDxxxPD __m512d _mm512_mask3_fmadd_round_pd(__m512d a, __m512d b, __m512d c, __mmask8 k, int r);
VFMADDxxxPD __m256d _mm256_mask_fmadd_pd(__m256d a, __mmask8 k, __m256d b, __m256d c);
VFMADDxxxPD __m256d _mm256_maskz_fmadd_pd(__mmask8 k, __m256d a, __m256d b, __m256d c);
VFMADDxxxPD __m256d _mm256_mask3_fmadd_pd(__m256d a, __m256d b, __m256d c, __mmask8 k);
VFMADDxxxPD __m128d _mm_mask_fmadd_pd(__m128d a, __mmask8 k, __m128d b, __m128d c);
VFMADDxxxPD __m128d _mm_maskz_fmadd_pd(__mmask8 k, __m128d a, __m128d b, __m128d c);
VFMADDxxxPD __m128d _mm_mask3_fmadd_pd(__m128d a, __m128d b, __m128d c, __mmask8 k);
VFMADDxxxPD __m128d _mm_fmadd_pd (__m128d a, __m128d b, __m128d c);
VFMADDxxxPD __m256d _mm256_fmadd_pd (__m256d a, __m256d b, __m256d c);

```

### SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

### Other Exceptions

VEX-encoded instructions, see Exceptions Type 2.

EVEX-encoded instructions, see Exceptions Type E2.

## VFMADD132PS/VFMADD213PS/VFMADD231PS—Fused Multiply-Add of Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.NDS.128.66.0F38.W0 98 /r VFMADD132PS xmm1, xmm2, xmm3/m128	RVM	V/V	FMA	Multiply packed single-precision floating-point values from xmm1 and xmm3/mem, add to xmm2 and put result in xmm1.
VEX.NDS.128.66.0F38.W0 A8 /r VFMADD213PS xmm1, xmm2, xmm3/m128	RVM	V/V	FMA	Multiply packed single-precision floating-point values from xmm1 and xmm2, add to xmm3/mem and put result in xmm1.
VEX.NDS.128.66.0F38.W0 B8 /r VFMADD231PS xmm1, xmm2, xmm3/m128	RVM	V/V	FMA	Multiply packed single-precision floating-point values from xmm2 and xmm3/mem, add to xmm1 and put result in xmm1.
VEX.NDS.256.66.0F38.W0 98 /r VFMADD132PS ymm1, ymm2, ymm3/m256	RVM	V/V	FMA	Multiply packed single-precision floating-point values from ymm1 and ymm3/mem, add to ymm2 and put result in ymm1.
VEX.NDS.256.66.0F38.W0 A8 /r VFMADD213PS ymm1, ymm2, ymm3/m256	RVM	V/V	FMA	Multiply packed single-precision floating-point values from ymm1 and ymm2, add to ymm3/mem and put result in ymm1.
VEX.NDS.256.66.0F38.0 B8 /r VFMADD231PS ymm1, ymm2, ymm3/m256	RVM	V/V	FMA	Multiply packed single-precision floating-point values from ymm2 and ymm3/mem, add to ymm1 and put result in ymm1.
EVEX.NDS.128.66.0F38.W0 98 /r VFMADD132PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	FV	V/V	AVX512VL AVX512F	Multiply packed single-precision floating-point values from xmm1 and xmm3/m128/m32bcst, add to xmm2 and put result in xmm1.
EVEX.NDS.128.66.0F38.W0 A8 /r VFMADD213PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	FV	V/V	AVX512VL AVX512F	Multiply packed single-precision floating-point values from xmm1 and xmm2, add to xmm3/m128/m32bcst and put result in xmm1.
EVEX.NDS.128.66.0F38.W0 B8 /r VFMADD231PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	FV	V/V	AVX512VL AVX512F	Multiply packed single-precision floating-point values from xmm2 and xmm3/m128/m32bcst, add to xmm1 and put result in xmm1.
EVEX.NDS.256.66.0F38.W0 98 /r VFMADD132PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	FV	V/V	AVX512VL AVX512F	Multiply packed single-precision floating-point values from ymm1 and ymm3/m256/m32bcst, add to ymm2 and put result in ymm1.
EVEX.NDS.256.66.0F38.W0 A8 /r VFMADD213PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	FV	V/V	AVX512VL AVX512F	Multiply packed single-precision floating-point values from ymm1 and ymm2, add to ymm3/m256/m32bcst and put result in ymm1.
EVEX.NDS.256.66.0F38.W0 B8 /r VFMADD231PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	FV	V/V	AVX512VL AVX512F	Multiply packed single-precision floating-point values from ymm2 and ymm3/m256/m32bcst, add to ymm1 and put result in ymm1.
EVEX.NDS.512.66.0F38.W0 98 /r VFMADD132PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}	FV	V/V	AVX512F	Multiply packed single-precision floating-point values from zmm1 and zmm3/m512/m32bcst, add to zmm2 and put result in zmm1.
EVEX.NDS.512.66.0F38.W0 A8 /r VFMADD213PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}	FV	V/V	AVX512F	Multiply packed single-precision floating-point values from zmm1 and zmm2, add to zmm3/m512/m32bcst and put result in zmm1.
EVEX.NDS.512.66.0F38.W0 B8 /r VFMADD231PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}	FV	V/V	AVX512F	Multiply packed single-precision floating-point values from zmm2 and zmm3/m512/m32bcst, add to zmm1 and put result in zmm1.

## Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RVM	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FV	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

## Description

Performs a set of SIMD multiply-add computation on packed single-precision floating-point values using three source operands and writes the multiply-add results in the destination operand. The destination operand is also the first source operand. The second operand must be a SIMD register. The third source operand can be a SIMD register or a memory location.

**VFMADD132PS:** Multiplies the four, eight or sixteen packed single-precision floating-point values from the first source operand to the four, eight or sixteen packed single-precision floating-point values in the third source operand, adds the infinite precision intermediate result to the four, eight or sixteen packed single-precision floating-point values in the second source operand, performs rounding and stores the resulting four, eight or sixteen packed single-precision floating-point values to the destination operand (first source operand).

**VFMADD213PS:** Multiplies the four, eight or sixteen packed single-precision floating-point values from the second source operand to the four, eight or sixteen packed single-precision floating-point values in the first source operand, adds the infinite precision intermediate result to the four, eight or sixteen packed single-precision floating-point values in the third source operand, performs rounding and stores the resulting the four, eight or sixteen packed single-precision floating-point values to the destination operand (first source operand).

**VFMADD231PS:** Multiplies the four, eight or sixteen packed single-precision floating-point values from the second source operand to the four, eight or sixteen packed single-precision floating-point values in the third source operand, adds the infinite precision intermediate result to the four, eight or sixteen packed single-precision floating-point values in the first source operand, performs rounding and stores the resulting four, eight or sixteen packed single-precision floating-point values to the destination operand (first source operand).

**EVEX encoded versions:** The destination operand (also first source operand) is a ZMM register and encoded in `reg_field`. The second source operand is a ZMM register and encoded in `EVEX.vvvv`. The third source operand is a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 32-bit memory location. The destination operand is conditionally updated with write mask `k1`.

**VEX.256 encoded version:** The destination operand (also first source operand) is a YMM register and encoded in `reg_field`. The second source operand is a YMM register and encoded in `VEX.vvvv`. The third source operand is a YMM register or a 256-bit memory location and encoded in `rm_field`.

**VEX.128 encoded version:** The destination operand (also first source operand) is a XMM register and encoded in `reg_field`. The second source operand is a XMM register and encoded in `VEX.vvvv`. The third source operand is a XMM register or a 128-bit memory location and encoded in `rm_field`. The upper 128 bits of the YMM destination register are zeroed.

## Operation

In the operations below, "\*" and "+" symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding).

**VFMADD132PS DEST, SRC2, SRC3**

```

IF (VEX.128) THEN
    MAXNUM ← 4
ELSEIF (VEX.256)
    MAXNUM ← 8
FI
For i = 0 to MAXNUM-1 {
    n ← 32*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR(DEST[n+31:n]*SRC3[n+31:n] + SRC2[n+31:n])
}
IF (VEX.128) THEN
    DEST[MAX_VL-1:128] ← 0

```

```

ELSEIF (VEX.256)
    DEST[MAX_VL-1:256] ← 0
FI

```

**VFMADD213PS DEST, SRC2, SRC3**

```

IF (VEX.128) THEN
    MAXNUM ← 4
ELSEIF (VEX.256)
    MAXNUM ← 8
FI
For i = 0 to MAXNUM-1 {
    n ← 32*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR(SRC2[n+31:n]*DEST[n+31:n] + SRC3[n+31:n])
}
IF (VEX.128) THEN
    DEST[MAX_VL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAX_VL-1:256] ← 0
FI

```

**VFMADD231PS DEST, SRC2, SRC3**

```

IF (VEX.128) THEN
    MAXNUM ← 4
ELSEIF (VEX.256)
    MAXNUM ← 8
FI
For i = 0 to MAXNUM-1 {
    n ← 32*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR(SRC2[n+31:n]*SRC3[n+31:n] + DEST[n+31:n])
}
IF (VEX.128) THEN
    DEST[MAX_VL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAX_VL-1:256] ← 0
FI

```

**VFMADD132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

```

IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ←
            RoundFPControl(DEST[i+31:i]*SRC3[j+31:i] + SRC2[j+31:i])
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+31:i] ← 0

```

```

    FI
  FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VFMADD132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)**

(KL, VL) = (4, 128), (8, 256), (16, 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] ←
            RoundFPControl_MXCSR(DEST[i+31:i]*SRC3[31:0] + SRC2[i+31:i])
        ELSE
          DEST[i+31:i] ←
            RoundFPControl_MXCSR(DEST[i+31:i]*SRC3[i+31:i] + SRC2[i+31:i])
        FI;
      ELSE
        IF *merging-masking* ; merging-masking
          THEN *DEST[i+31:i] remains unchanged*
        ELSE ; zeroing-masking
          DEST[i+31:i] ← 0
        FI
      FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VFMADD213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

IF (VL = 512) AND (EVEX.b = 1)

```

  THEN
    SET_RM(EVEX.RC);
  ELSE
    SET_RM(MXCSR.RM);
  FI;
FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ←
      RoundFPControl(SRC2[i+31:i]*DEST[i+31:i] + SRC3[i+31:i])
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+31:i] remains unchanged*
      ELSE ; zeroing-masking
        DEST[i+31:i] ← 0
      FI
    FI
  FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VFMAADD213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)**  
 (KL, VL) = (4, 128), (8, 256), (16, 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] ←
            RoundFPControl_MXCSR(SRC2[i+31:i]*DEST[i+31:i] + SRC3[31:0])
        ELSE
          DEST[i+31:i] ←
            RoundFPControl_MXCSR(SRC2[i+31:i]*DEST[i+31:i] + SRC3[i+31:i])
        FI;
      ELSE
        IF *merging-masking*           ; merging-masking
          THEN *DEST[i+31:i] remains unchanged*
        ELSE                             ; zeroing-masking
          DEST[i+31:i] ← 0
        FI
      FI;
    ENDFOR
  DEST[MAX_VL-1:VL] ← 0

```

**VFMAADD231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

IF (VL = 512) AND (EVEX.b = 1)

```

  THEN
    SET_RM(EVEX.RC);
  ELSE
    SET_RM(MXCSR.RM);
  FI;
FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ←
      RoundFPControl(SRC2[i+31:i]*SRC3[i+31:i] + DEST[i+31:i])
    ELSE
      IF *merging-masking*           ; merging-masking
        THEN *DEST[i+31:i] remains unchanged*
      ELSE                             ; zeroing-masking
        DEST[i+31:i] ← 0
      FI
    FI;
  ENDFOR
  DEST[MAX_VL-1:VL] ← 0

```

**VFMAADD231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)**

(KL, VL) = (4, 128), (8, 256), (16, 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] ←
      RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[31:0] + DEST[i+31:i])
    ELSE
      DEST[i+31:i] ←
      RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[i+31:i] + DEST[i+31:i])
  FI;
ELSE
  IF *merging-masking*           ; merging-masking
    THEN *DEST[i+31:i] remains unchanged*
  ELSE                             ; zeroing-masking
    DEST[i+31:i] ← 0
  FI
FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

### Intel C/C++ Compiler Intrinsic Equivalent

```

VFMADDxxxPS __m512 __mm512_fmadd_ps(__m512 a, __m512 b, __m512 c);
VFMADDxxxPS __m512 __mm512_fmadd_round_ps(__m512 a, __m512 b, __m512 c, int r);
VFMADDxxxPS __m512 __mm512_mask_fmadd_ps(__m512 a, __mmask16 k, __m512 b, __m512 c);
VFMADDxxxPS __m512 __mm512_maskz_fmadd_ps(__mmask16 k, __m512 a, __m512 b, __m512 c);
VFMADDxxxPS __m512 __mm512_mask3_fmadd_ps(__m512 a, __m512 b, __m512 c, __mmask16 k);
VFMADDxxxPS __m512 __mm512_mask_fmadd_round_ps(__m512 a, __mmask16 k, __m512 b, __m512 c, int r);
VFMADDxxxPS __m512 __mm512_maskz_fmadd_round_ps(__mmask16 k, __m512 a, __m512 b, __m512 c, int r);
VFMADDxxxPS __m512 __mm512_mask3_fmadd_round_ps(__m512 a, __m512 b, __m512 c, __mmask16 k, int r);
VFMADDxxxPS __m256 __mm256_mask_fmadd_ps(__m256 a, __mmask8 k, __m256 b, __m256 c);
VFMADDxxxPS __m256 __mm256_maskz_fmadd_ps(__mmask8 k, __m256 a, __m256 b, __m256 c);
VFMADDxxxPS __m256 __mm256_mask3_fmadd_ps(__m256 a, __m256 b, __m256 c, __mmask8 k);
VFMADDxxxPS __m128 __mm_mask_fmadd_ps(__m128 a, __mmask8 k, __m128 b, __m128 c);
VFMADDxxxPS __m128 __mm_maskz_fmadd_ps(__mmask8 k, __m128 a, __m128 b, __m128 c);
VFMADDxxxPS __m128 __mm_mask3_fmadd_ps(__m128 a, __m128 b, __m128 c, __mmask8 k);
VFMADDxxxPS __m128 __mm_fmadd_ps (__m128 a, __m128 b, __m128 c);
VFMADDxxxPS __m256 __mm256_fmadd_ps (__m256 a, __m256 b, __m256 c);

```

### SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

### Other Exceptions

VEX-encoded instructions, see Exceptions Type 2.

EVEX-encoded instructions, see Exceptions Type E2.

## VFMADD132SD/VFMADD213SD/VFMADD231SD—Fused Multiply-Add of Scalar Double-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.DDS.LIG.66.0F38.W1 99 /r VFMADD132SD xmm1, xmm2, xmm3/m64	RVM	V/V	FMA	Multiply scalar double-precision floating-point value from xmm1 and xmm3/m64, add to xmm2 and put result in xmm1.
VEX.DDS.LIG.66.0F38.W1 A9 /r VFMADD213SD xmm1, xmm2, xmm3/m64	RVM	V/V	FMA	Multiply scalar double-precision floating-point value from xmm1 and xmm2, add to xmm3/m64 and put result in xmm1.
VEX.DDS.LIG.66.0F38.W1 B9 /r VFMADD231SD xmm1, xmm2, xmm3/m64	RVM	V/V	FMA	Multiply scalar double-precision floating-point value from xmm2 and xmm3/m64, add to xmm1 and put result in xmm1.
EVEX.DDS.LIG.66.0F38.W1 99 /r VFMADD132SD xmm1 {k1}{z}, xmm2, xmm3/m64{er}	T1S	V/V	AVX512F	Multiply scalar double-precision floating-point value from xmm1 and xmm3/m64, add to xmm2 and put result in xmm1.
EVEX.DDS.LIG.66.0F38.W1 A9 /r VFMADD213SD xmm1 {k1}{z}, xmm2, xmm3/m64{er}	T1S	V/V	AVX512F	Multiply scalar double-precision floating-point value from xmm1 and xmm2, add to xmm3/m64 and put result in xmm1.
EVEX.DDS.LIG.66.0F38.W1 B9 /r VFMADD231SD xmm1 {k1}{z}, xmm2, xmm3/m64{er}	T1S	V/V	AVX512F	Multiply scalar double-precision floating-point value from xmm2 and xmm3/m64, add to xmm1 and put result in xmm1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RVM	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
T1S	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Performs a SIMD multiply-add computation on the low double-precision floating-point values using three source operands and writes the multiply-add result in the destination operand. The destination operand is also the first source operand. The first and second operand are XMM registers. The third source operand can be an XMM register or a 64-bit memory location.

**VFMADD132SD:** Multiplies the low double-precision floating-point value from the first source operand to the low double-precision floating-point value in the third source operand, adds the infinite precision intermediate result to the low double-precision floating-point values in the second source operand, performs rounding and stores the resulting double-precision floating-point value to the destination operand (first source operand).

**VFMADD213SD:** Multiplies the low double-precision floating-point value from the second source operand to the low double-precision floating-point value in the first source operand, adds the infinite precision intermediate result to the low double-precision floating-point value in the third source operand, performs rounding and stores the resulting double-precision floating-point value to the destination operand (first source operand).

**VFMADD231SD:** Multiplies the low double-precision floating-point value from the second source to the low double-precision floating-point value in the third source operand, adds the infinite precision intermediate result to the low double-precision floating-point value in the first source operand, performs rounding and stores the resulting double-precision floating-point value to the destination operand (first source operand).

VEX.128 and EVEX encoded version: The destination operand (also first source operand) is encoded in `reg_field`. The second source operand is encoded in VEX.vvvv/EVEX.vvvv. The third source operand is encoded in `rm_field`. Bits 127:64 of the destination are unchanged. Bits MAXVL-1:128 of the destination register are zeroed.

EVEX encoded version: The low quadword element of the destination is updated according to the writemask.

### Operation

In the operations below, “\*” and “+” symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding).

#### VFMADD132SD DEST, SRC2, SRC3 (EVEX encoded version)

```
IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
    THEN DEST[63:0] ← RoundFPControl(DEST[63:0]*SRC3[63:0] + SRC2[63:0])
    ELSE
        IF *merging-masking*           ; merging-masking
            THEN *DEST[63:0] remains unchanged*
        ELSE                             ; zeroing-masking
            THEN DEST[63:0] ← 0
        FI;
FI;
DEST[127:64] ← DEST[127:64]
DEST[MAX_VL-1:128] ← 0
```

#### VFMADD213SD DEST, SRC2, SRC3 (EVEX encoded version)

```
IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
    THEN DEST[63:0] ← RoundFPControl(SRC2[63:0]*DEST[63:0] + SRC3[63:0])
    ELSE
        IF *merging-masking*           ; merging-masking
            THEN *DEST[63:0] remains unchanged*
        ELSE                             ; zeroing-masking
            THEN DEST[63:0] ← 0
        FI;
FI;
DEST[127:64] ← DEST[127:64]
DEST[MAX_VL-1:128] ← 0
```

#### VFMADD231SD DEST, SRC2, SRC3 (EVEX encoded version)

```
IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
```

```

IF k1[0] or *no writemask*
  THEN DEST[63:0] ← RoundFPControl(SRC2[63:0]*SRC3[63:0] + DEST[63:0])
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[63:0] remains unchanged*
    ELSE ; zeroing-masking
      THEN DEST[63:0] ← 0
    FI;
  FI;
DEST[127:64] ← DEST[127:64]
DEST[MAX_VL-1:128] ← 0

```

**VFMADD132SD DEST, SRC2, SRC3 (VEX encoded version)**

```

DEST[63:0] ← MAX_VL-1:128RoundFPControl_MXCSR(DEST[63:0]*SRC3[63:0] + SRC2[63:0])
DEST[127:63] ← DEST[127:63]
DEST[MAX_VL-1:128] ← 0

```

**VFMADD213SD DEST, SRC2, SRC3 (VEX encoded version)**

```

DEST[63:0] ← RoundFPControl_MXCSR(SRC2[63:0]*DEST[63:0] + SRC3[63:0])
DEST[127:63] ← DEST[127:63]
DEST[MAX_VL-1:128] ← 0

```

**VFMADD231SD DEST, SRC2, SRC3 (VEX encoded version)**

```

DEST[63:0] ← RoundFPControl_MXCSR(SRC2[63:0]*SRC3[63:0] + DEST[63:0])
DEST[127:63] ← DEST[127:63]
DEST[MAX_VL-1:128] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VFMADDxxxSD __m128d __mm_fmadd_round_sd(__m128d a, __m128d b, __m128d c, int r);
VFMADDxxxSD __m128d __mm_mask_fmadd_sd(__m128d a, __mmask8 k, __m128d b, __m128d c);
VFMADDxxxSD __m128d __mm_maskz_fmadd_sd(__mmask8 k, __m128d a, __m128d b, __m128d c);
VFMADDxxxSD __m128d __mm_mask3_fmadd_sd(__m128d a, __m128d b, __m128d c, __mmask8 k);
VFMADDxxxSD __m128d __mm_mask_fmadd_round_sd(__m128d a, __mmask8 k, __m128d b, __m128d c, int r);
VFMADDxxxSD __m128d __mm_maskz_fmadd_round_sd(__mmask8 k, __m128d a, __m128d b, __m128d c, int r);
VFMADDxxxSD __m128d __mm_mask3_fmadd_round_sd(__m128d a, __m128d b, __m128d c, __mmask8 k, int r);
VFMADDxxxSD __m128d __mm_fmadd_sd (__m128d a, __m128d b, __m128d c);

```

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal

**Other Exceptions**

VEX-encoded instructions, see Exceptions Type 3.

EVEX-encoded instructions, see Exceptions Type E3.

## VFMADD132SS/VFMADD213SS/VFMADD231SS—Fused Multiply-Add of Scalar Single-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.DDS.LIG.66.0F38.W0 99 /r VFMADD132SS xmm1, xmm2, xmm3/m32	RVM	V/V	FMA	Multiply scalar single-precision floating-point value from xmm1 and xmm3/m32, add to xmm2 and put result in xmm1.
VEX.DDS.LIG.66.0F38.W0 A9 /r VFMADD213SS xmm1, xmm2, xmm3/m32	RVM	V/V	FMA	Multiply scalar single-precision floating-point value from xmm1 and xmm2, add to xmm3/m32 and put result in xmm1.
VEX.DDS.LIG.66.0F38.W0 B9 /r VFMADD231SS xmm1, xmm2, xmm3/m32	RVM	V/V	FMA	Multiply scalar single-precision floating-point value from xmm2 and xmm3/m32, add to xmm1 and put result in xmm1.
EVEX.DDS.LIG.66.0F38.W0 99 /r VFMADD132SS xmm1 {k1}{z}, xmm2, xmm3/m32{er}	T1S	V/V	AVX512F	Multiply scalar single-precision floating-point value from xmm1 and xmm3/m32, add to xmm2 and put result in xmm1.
EVEX.DDS.LIG.66.0F38.W0 A9 /r VFMADD213SS xmm1 {k1}{z}, xmm2, xmm3/m32{er}	T1S	V/V	AVX512F	Multiply scalar single-precision floating-point value from xmm1 and xmm2, add to xmm3/m32 and put result in xmm1.
EVEX.DDS.LIG.66.0F38.W0 B9 /r VFMADD231SS xmm1 {k1}{z}, xmm2, xmm3/m32{er}	T1S	V/V	AVX512F	Multiply scalar single-precision floating-point value from xmm2 and xmm3/m32, add to xmm1 and put result in xmm1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RVM	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
T1S	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Performs a SIMD multiply-add computation on single-precision floating-point values using three source operands and writes the multiply-add results in the destination operand. The destination operand is also the first source operand. The first and second operands are XMM registers. The third source operand can be a XMM register or a 32-bit memory location.

**VFMADD132SS:** Multiplies the low single-precision floating-point value from the first source operand to the low single-precision floating-point value in the third source operand, adds the infinite precision intermediate result to the low single-precision floating-point value in the second source operand, performs rounding and stores the resulting single-precision floating-point value to the destination operand (first source operand).

**VFMADD213SS:** Multiplies the low single-precision floating-point value from the second source operand to the low single-precision floating-point value in the first source operand, adds the infinite precision intermediate result to the low single-precision floating-point value in the third source operand, performs rounding and stores the resulting single-precision floating-point value to the destination operand (first source operand).

**VFMADD231SS:** Multiplies the low single-precision floating-point value from the second source operand to the low single-precision floating-point value in the third source operand, adds the infinite precision intermediate result to the low single-precision floating-point value in the first source operand, performs rounding and stores the resulting single-precision floating-point value to the destination operand (first source operand).

VEX.128 and EVEX encoded version: The destination operand (also first source operand) is encoded in `reg_field`. The second source operand is encoded in VEX.vvvv/EVEX.vvvv. The third source operand is encoded in `rm_field`. Bits 127:32 of the destination are unchanged. Bits MAXVL-1:128 of the destination register are zeroed.

EVEX encoded version: The low doubleword element of the destination is updated according to the writemask.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NaNs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column.

### Operation

In the operations below, "\*" and "+" symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding).

#### VFMADD132SS DEST, SRC2, SRC3 (EVEX encoded version)

IF (EVEX.b = 1) and SRC3 \*is a register\*

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

IF k1[0] or \*no writemask\*

THEN DEST[31:0] ← RoundFPControl(DEST[31:0]\*SRC3[31:0] + SRC2[31:0])

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[31:0] remains unchanged\*

ELSE ; zeroing-masking

THEN DEST[31:0] ← 0

FI;

FI;

DEST[127:32] ← DEST[127:32]

DEST[MAX\_VL-1:128] ← 0

#### VFMADD213SS DEST, SRC2, SRC3 (EVEX encoded version)

IF (EVEX.b = 1) and SRC3 \*is a register\*

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

IF k1[0] or \*no writemask\*

THEN DEST[31:0] ← RoundFPControl(SRC2[31:0]\*DEST[31:0] + SRC3[31:0])

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[31:0] remains unchanged\*

ELSE ; zeroing-masking

THEN DEST[31:0] ← 0

FI;

FI;

DEST[127:32] ← DEST[127:32]

DEST[MAX\_VL-1:128] ← 0

**VFMAADD231SS DEST, SRC2, SRC3 (EVEX encoded version)**

```

IF (EVEX.b = 1) and SRC3 *is a register*
  THEN
    SET_RM(EVEX.RC);
  ELSE
    SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
  THEN DEST[31:0] ← RoundFPControl(SRC2[31:0]*SRC3[31:0] + DEST[31:0])
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[31:0] remains unchanged*
    ELSE ; zeroing-masking
      THEN DEST[31:0] ← 0
    FI;
  FI;
DEST[127:32] ← DEST[127:32]
DEST[MAX_VL-1:128] ← 0

```

**VFMAADD132SS DEST, SRC2, SRC3 (VEX encoded version)**

```

DEST[31:0] ← RoundFPControl_MXCSR(DEST[31:0]*SRC3[31:0] + SRC2[31:0])
DEST[127:32] ← DEST[127:32]
DEST[MAX_VL-1:128] ← 0

```

**VFMAADD213SS DEST, SRC2, SRC3 (VEX encoded version)**

```

DEST[31:0] ← RoundFPControl_MXCSR(SRC2[31:0]*DEST[31:0] + SRC3[31:0])
DEST[127:32] ← DEST[127:32]
DEST[MAX_VL-1:128] ← 0

```

**VFMAADD231SS DEST, SRC2, SRC3 (VEX encoded version)**

```

DEST[31:0] ← RoundFPControl_MXCSR(SRC2[31:0]*SRC3[31:0] + DEST[31:0])
DEST[127:32] ← DEST[127:32]
DEST[MAX_VL-1:128] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VFMAADDxxxSS __m128 __mm_fmadd_round_ss(__m128 a, __m128 b, __m128 c, int r);
VFMAADDxxxSS __m128 __mm_mask_fmadd_ss(__m128 a, __mmask8 k, __m128 b, __m128 c);
VFMAADDxxxSS __m128 __mm_maskz_fmadd_ss(__mmask8 k, __m128 a, __m128 b, __m128 c);
VFMAADDxxxSS __m128 __mm_mask3_fmadd_ss(__m128 a, __m128 b, __m128 c, __mmask8 k);
VFMAADDxxxSS __m128 __mm_mask_fmadd_round_ss(__m128 a, __mmask8 k, __m128 b, __m128 c, int r);
VFMAADDxxxSS __m128 __mm_maskz_fmadd_round_ss(__mmask8 k, __m128 a, __m128 b, __m128 c, int r);
VFMAADDxxxSS __m128 __mm_mask3_fmadd_round_ss(__m128 a, __m128 b, __m128 c, __mmask8 k, int r);
VFMAADDxxxSS __m128 __mm_fmadd_ss(__m128 a, __m128 b, __m128 c);

```

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal

**Other Exceptions**

VEX-encoded instructions, see Exceptions Type 3.

EVEX-encoded instructions, see Exceptions Type E3.

## VFMADDSUB132PD/VFMADDSUB213PD/VFMADDSUB231PD—Fused Multiply-Alternating Add/Subtract of Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.DDS.128.66.0F38.W1 96 /r VFMADDSUB132PD xmm1, xmm2, xmm3/m128	RVM	V/V	FMA	Multiply packed double-precision floating-point values from xmm1 and xmm3/mem, add/subtract elements in xmm2 and put result in xmm1.
VEX.DDS.128.66.0F38.W1 A6 /r VFMADDSUB213PD xmm1, xmm2, xmm3/m128	RVM	V/V	FMA	Multiply packed double-precision floating-point values from xmm1 and xmm2, add/subtract elements in xmm3/mem and put result in xmm1.
VEX.DDS.128.66.0F38.W1 B6 /r VFMADDSUB231PD xmm1, xmm2, xmm3/m128	RVM	V/V	FMA	Multiply packed double-precision floating-point values from xmm2 and xmm3/mem, add/subtract elements in xmm1 and put result in xmm1.
VEX.DDS.256.66.0F38.W1 96 /r VFMADDSUB132PD ymm1, ymm2, ymm3/m256	RVM	V/V	FMA	Multiply packed double-precision floating-point values from ymm1 and ymm3/mem, add/subtract elements in ymm2 and put result in ymm1.
VEX.DDS.256.66.0F38.W1 A6 /r VFMADDSUB213PD ymm1, ymm2, ymm3/m256	RVM	V/V	FMA	Multiply packed double-precision floating-point values from ymm1 and ymm2, add/subtract elements in ymm3/mem and put result in ymm1.
VEX.DDS.256.66.0F38.W1 B6 /r VFMADDSUB231PD ymm1, ymm2, ymm3/m256	RVM	V/V	FMA	Multiply packed double-precision floating-point values from ymm2 and ymm3/mem, add/subtract elements in ymm1 and put result in ymm1.
EVEX.DDS.128.66.0F38.W1 A6 /r VFMADDSUB213PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	FV	V/V	AVX512VL AVX512F	Multiply packed double-precision floating-point values from xmm1 and xmm2, add/subtract elements in xmm3/m128/m64bcst and put result in xmm1 subject to writemask k1.
EVEX.DDS.128.66.0F38.W1 B6 /r VFMADDSUB231PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	FV	V/V	AVX512VL AVX512F	Multiply packed double-precision floating-point values from xmm2 and xmm3/m128/m64bcst, add/subtract elements in xmm1 and put result in xmm1 subject to writemask k1.
EVEX.DDS.128.66.0F38.W1 96 /r VFMADDSUB132PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	FV	V/V	AVX512VL AVX512F	Multiply packed double-precision floating-point values from xmm1 and xmm3/m128/m64bcst, add/subtract elements in xmm2 and put result in xmm1 subject to writemask k1.
EVEX.DDS.256.66.0F38.W1 A6 /r VFMADDSUB213PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	FV	V/V	AVX512VL AVX512F	Multiply packed double-precision floating-point values from ymm1 and ymm2, add/subtract elements in ymm3/m256/m64bcst and put result in ymm1 subject to writemask k1.
EVEX.DDS.256.66.0F38.W1 B6 /r VFMADDSUB231PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	FV	V/V	AVX512VL AVX512F	Multiply packed double-precision floating-point values from ymm2 and ymm3/m256/m64bcst, add/subtract elements in ymm1 and put result in ymm1 subject to writemask k1.
EVEX.DDS.256.66.0F38.W1 96 /r VFMADDSUB132PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	FV	V/V	AVX512VL AVX512F	Multiply packed double-precision floating-point values from ymm1 and ymm3/m256/m64bcst, add/subtract elements in ymm2 and put result in ymm1 subject to writemask k1.
EVEX.DDS.512.66.0F38.W1 A6 /r VFMADDSUB213PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	FV	V/V	AVX512F	Multiply packed double-precision floating-point values from zmm1 and zmm2, add/subtract elements in zmm3/m512/m64bcst and put result in zmm1 subject to writemask k1.



Opcode/ Instruction	Op / En	64/32 bitMode Support	CPUID Feature Flag	Description
EVEX.DDS.512.66.0F38.W1 B6 /r VFMADDSUB231PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	FV	V/V	AVX512F	Multiply packed double-precision floating-point values from zmm2 and zmm3/m512/m64bcst, add/subtract elements in zmm1 and put result in zmm1 subject to writemask k1.
EVEX.DDS.512.66.0F38.W1 96 /r VFMADDSUB132PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	FV	V/V	AVX512F	Multiply packed double-precision floating-point values from zmm1 and zmm3/m512/m64bcst, add/subtract elements in zmm2 and put result in zmm1 subject to writemask k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RVM	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FV	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

#### Description

VFMADDSUB132PD: Multiplies the two, four, or eight packed double-precision floating-point values from the first source operand to the two or four packed double-precision floating-point values in the third source operand. From the infinite precision intermediate result, adds the odd double-precision floating-point elements and subtracts the even double-precision floating-point values in the second source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand (first source operand).

VFMADDSUB213PD: Multiplies the two, four, or eight packed double-precision floating-point values from the second source operand to the two or four packed double-precision floating-point values in the first source operand. From the infinite precision intermediate result, adds the odd double-precision floating-point elements and subtracts the even double-precision floating-point values in the third source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand (first source operand).

VFMADDSUB231PD: Multiplies the two, four, or eight packed double-precision floating-point values from the second source operand to the two or four packed double-precision floating-point values in the third source operand. From the infinite precision intermediate result, adds the odd double-precision floating-point elements and subtracts the even double-precision floating-point values in the first source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand (first source operand).

EVEX encoded versions: The destination operand (also first source operand) and the second source operand are ZMM/YMM/XMM register. The third 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 64-bit memory location. The destination operand is conditionally updated with write mask k1.

VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in `reg_field`. The second source operand is a YMM register and encoded in `VEX.vvvv`. The third source operand is a YMM register or a 256-bit memory location and encoded in `rm_field`.

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in `reg_field`. The second source operand is a XMM register and encoded in `VEX.vvvv`. The third source operand is a XMM register or a 128-bit memory location and encoded in `rm_field`. The upper 128 bits of the YMM destination register are zeroed.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NaNs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column.

**Operation**

In the operations below, “\*” and “-” symbols represent multiplication and subtraction with infinite precision inputs and outputs (no rounding).

**VFMADDSUB132PD DEST, SRC2, SRC3**

```
IF (VEX.128) THEN
    DEST[63:0] ← RoundFPControl_MXCSR(DEST[63:0]*SRC3[63:0] - SRC2[63:0])
    DEST[127:64] ← RoundFPControl_MXCSR(DEST[127:64]*SRC3[127:64] + SRC2[127:64])
    DEST[MAX_VL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[63:0] ← RoundFPControl_MXCSR(DEST[63:0]*SRC3[63:0] - SRC2[63:0])
    DEST[127:64] ← RoundFPControl_MXCSR(DEST[127:64]*SRC3[127:64] + SRC2[127:64])
    DEST[191:128] ← RoundFPControl_MXCSR(DEST[191:128]*SRC3[191:128] - SRC2[191:128])
    DEST[255:192] ← RoundFPControl_MXCSR(DEST[255:192]*SRC3[255:192] + SRC2[255:192])
FI
```

**VFMADDSUB213PD DEST, SRC2, SRC3**

```
IF (VEX.128) THEN
    DEST[63:0] ← RoundFPControl_MXCSR(SRC2[63:0]*DEST[63:0] - SRC3[63:0])
    DEST[127:64] ← RoundFPControl_MXCSR(SRC2[127:64]*DEST[127:64] + SRC3[127:64])
    DEST[MAX_VL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[63:0] ← RoundFPControl_MXCSR(SRC2[63:0]*DEST[63:0] - SRC3[63:0])
    DEST[127:64] ← RoundFPControl_MXCSR(SRC2[127:64]*DEST[127:64] + SRC3[127:64])
    DEST[191:128] ← RoundFPControl_MXCSR(SRC2[191:128]*DEST[191:128] - SRC3[191:128])
    DEST[255:192] ← RoundFPControl_MXCSR(SRC2[255:192]*DEST[255:192] + SRC3[255:192])
FI
```

**VFMADDSUB231PD DEST, SRC2, SRC3**

```
IF (VEX.128) THEN
    DEST[63:0] ← RoundFPControl_MXCSR(SRC2[63:0]*SRC3[63:0] - DEST[63:0])
    DEST[127:64] ← RoundFPControl_MXCSR(SRC2[127:64]*SRC3[127:64] + DEST[127:64])
    DEST[MAX_VL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[63:0] ← RoundFPControl_MXCSR(SRC2[63:0]*SRC3[63:0] - DEST[63:0])
    DEST[127:64] ← RoundFPControl_MXCSR(SRC2[127:64]*SRC3[127:64] + DEST[127:64])
    DEST[191:128] ← RoundFPControl_MXCSR(SRC2[191:128]*SRC3[191:128] - DEST[191:128])
    DEST[255:192] ← RoundFPControl_MXCSR(SRC2[255:192]*SRC3[255:192] + DEST[255:192])
FI
```

**VFMADDSUB132PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

IF (VL = 512) AND (EVEX.b = 1)

```
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
```

FI;

FOR j ← 0 TO KL-1

```
    i ← j * 64
```

```
    IF k1[j] OR *no writemask*
```

```
        THEN
```

```
            IF j *is even*
```

```
                THEN DEST[i+63:i] ←
```

```

        RoundFPControl(DEST[i+63:i]*SRC3[i+63:i] - SRC2[i+63:i])
    ELSE DEST[i+63:i] ←
        RoundFPControl(DEST[i+63:i]*SRC3[i+63:i] + SRC2[i+63:i])
    FI
ELSE
    IF *merging-masking*           ; merging-masking
    THEN *DEST[i+63:i] remains unchanged*
    ELSE                             ; zeroing-masking
        DEST[i+63:i] ← 0
    FI
FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VFMADDSUB132PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 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 j *is even*
        THEN
            IF (EVEX.b = 1)
            THEN
                DEST[i+63:i] ←
                RoundFPControl_MXCSR(DEST[i+63:i]*SRC3[63:0] - SRC2[i+63:i])
            ELSE
                DEST[i+63:i] ←
                RoundFPControl_MXCSR(DEST[i+63:i]*SRC3[i+63:i] - SRC2[i+63:i])
            FI;
        ELSE
            IF (EVEX.b = 1)
            THEN
                DEST[i+63:i] ←
                RoundFPControl_MXCSR(DEST[i+63:i]*SRC3[63:0] + SRC2[i+63:i])
            ELSE
                DEST[i+63:i] ←
                RoundFPControl_MXCSR(DEST[i+63:i]*SRC3[i+63:i] + SRC2[i+63:i])
            FI;
        FI
    ELSE
        IF *merging-masking*           ; merging-masking
        THEN *DEST[i+63:i] remains unchanged*
        ELSE                             ; zeroing-masking
            DEST[i+63:i] ← 0
        FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VFMADDSUB213PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

IF (VL = 512) AND (EVEX.b = 1)

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN

IF j \*is even\*

THEN DEST[i+63:i] ←

RoundFPControl(SRC2[i+63:i]\*DEST[i+63:i] - SRC3[i+63:i])

ELSE DEST[i+63:i] ←

RoundFPControl(SRC2[i+63:i]\*DEST[i+63:i] + SRC3[i+63:i])

FI

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VFMADDSUB213PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 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 j \*is even\*

THEN

IF (EVEX.b = 1)

THEN

DEST[i+63:i] ←

RoundFPControl\_MXCSR(SRC2[i+63:i]\*DEST[i+63:i] - SRC3[63:0])

ELSE

DEST[i+63:i] ←

RoundFPControl\_MXCSR(SRC2[i+63:i]\*DEST[i+63:i] - SRC3[i+63:i])

FI;

ELSE

IF (EVEX.b = 1)

THEN

DEST[i+63:i] ←

RoundFPControl\_MXCSR(SRC2[i+63:i]\*DEST[i+63:i] + SRC3[63:0])

ELSE

DEST[i+63:i] ←

RoundFPControl\_MXCSR(SRC2[i+63:i]\*DEST[i+63:i] + SRC3[i+63:i])

FI;

```

        FI
    ELSE
        IF *merging-masking*           ; merging-masking
            THEN *DEST[i+63:i] remains unchanged*
        ELSE                             ; zeroing-masking
            DEST[i+63:i] ← 0
        FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VFMADDSUB231PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

IF (VL = 512) AND (EVEX.b = 1)

```

    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
    FI;
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN DEST[i+63:i] ←
                    RoundFPControl(SRC2[i+63:i]*SRC3[i+63:i] - DEST[i+63:i])
            ELSE DEST[i+63:i] ←
                RoundFPControl(SRC2[i+63:i]*SRC3[i+63:i] + DEST[i+63:i])
            FI
        ELSE
            IF *merging-masking*           ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
            ELSE                             ; zeroing-masking
                DEST[i+63:i] ← 0
            FI
        FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VFMADDSUB231PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 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 j *is even*
                THEN
                    IF (EVEX.b = 1)
                        THEN
                            DEST[i+63:i] ←
                                RoundFPControl_MXCSR(SRC2[i+63:i]*SRC3[63:0] - DEST[i+63:i])
                    ELSE
                        DEST[i+63:i] ←

```

```

        RoundFPControl_MXCSR(SRC2[i+63:i]*SRC3[i+63:i] - DEST[i+63:i])
    FI;
ELSE
    IF (EVEX.b = 1)
        THEN
            DEST[i+63:i] ←
            RoundFPControl_MXCSR(SRC2[i+63:i]*SRC3[63:0] + DEST[i+63:i])
        ELSE
            DEST[i+63:i] ←
            RoundFPControl_MXCSR(SRC2[i+63:i]*SRC3[i+63:i] + DEST[i+63:i])
    FI;
FI
ELSE
    IF *merging-masking*                ; merging-masking
    THEN *DEST[i+63:i] remains unchanged*
    ELSE                                  ; zeroing-masking
        DEST[i+63:i] ← 0
    FI
FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

### Intel C/C++ Compiler Intrinsic Equivalent

```

VFMADDSUBxxxPD __m512d __mm512_fmaddsub_pd(__m512d a, __m512d b, __m512d c);
VFMADDSUBxxxPD __m512d __mm512_fmaddsub_round_pd(__m512d a, __m512d b, __m512d c, int r);
VFMADDSUBxxxPD __m512d __mm512_mask_fmaddsub_pd(__m512d a, __mmask8 k, __m512d b, __m512d c);
VFMADDSUBxxxPD __m512d __mm512_maskz_fmaddsub_pd(__mmask8 k, __m512d a, __m512d b, __m512d c);
VFMADDSUBxxxPD __m512d __mm512_mask3_fmaddsub_pd(__m512d a, __m512d b, __m512d c, __mmask8 k);
VFMADDSUBxxxPD __m512d __mm512_mask_fmaddsub_round_pd(__m512d a, __mmask8 k, __m512d b, __m512d c, int r);
VFMADDSUBxxxPD __m512d __mm512_maskz_fmaddsub_round_pd(__mmask8 k, __m512d a, __m512d b, __m512d c, int r);
VFMADDSUBxxxPD __m512d __mm512_mask3_fmaddsub_round_pd(__m512d a, __m512d b, __m512d c, __mmask8 k, int r);
VFMADDSUBxxxPD __m256d __mm256_mask_fmaddsub_pd(__m256d a, __mmask8 k, __m256d b, __m256d c);
VFMADDSUBxxxPD __m256d __mm256_maskz_fmaddsub_pd(__mmask8 k, __m256d a, __m256d b, __m256d c);
VFMADDSUBxxxPD __m256d __mm256_mask3_fmaddsub_pd(__m256d a, __m256d b, __m256d c, __mmask8 k);
VFMADDSUBxxxPD __m128d __mm_mask_fmaddsub_pd(__m128d a, __mmask8 k, __m128d b, __m128d c);
VFMADDSUBxxxPD __m128d __mm_maskz_fmaddsub_pd(__mmask8 k, __m128d a, __m128d b, __m128d c);
VFMADDSUBxxxPD __m128d __mm_mask3_fmaddsub_pd(__m128d a, __m128d b, __m128d c, __mmask8 k);
VFMADDSUBxxxPD __m128d __mm_fmaddsub_pd (__m128d a, __m128d b, __m128d c);
VFMADDSUBxxxPD __m256d __mm256_fmaddsub_pd (__m256d a, __m256d b, __m256d c);

```

### SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

### Other Exceptions

VEX-encoded instructions, see Exceptions Type 2.

EVEX-encoded instructions, see Exceptions Type E2.

## VFMADDSUB132PS/VFMADDSUB213PS/VFMADDSUB231PS—Fused Multiply-Alternating Add/Subtract of Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.DDS.128.66.0F38.W0 96 /r VFMADDSUB132PS xmm1, xmm2, xmm3/m128	RVM	V/V	FMA	Multiply packed single-precision floating-point values from xmm1 and xmm3/mem, add/subtract elements in xmm2 and put result in xmm1.
VEX.DDS.128.66.0F38.W0 A6 /r VFMADDSUB213PS xmm1, xmm2, xmm3/m128	RVM	V/V	FMA	Multiply packed single-precision floating-point values from xmm1 and xmm2, add/subtract elements in xmm3/mem and put result in xmm1.
VEX.DDS.128.66.0F38.W0 B6 /r VFMADDSUB231PS xmm1, xmm2, xmm3/m128	RVM	V/V	FMA	Multiply packed single-precision floating-point values from xmm2 and xmm3/mem, add/subtract elements in xmm1 and put result in xmm1.
VEX.DDS.256.66.0F38.W0 96 /r VFMADDSUB132PS ymm1, ymm2, ymm3/m256	RVM	V/V	FMA	Multiply packed single-precision floating-point values from ymm1 and ymm3/mem, add/subtract elements in ymm2 and put result in ymm1.
VEX.DDS.256.66.0F38.W0 A6 /r VFMADDSUB213PS ymm1, ymm2, ymm3/m256	RVM	V/V	FMA	Multiply packed single-precision floating-point values from ymm1 and ymm2, add/subtract elements in ymm3/mem and put result in ymm1.
VEX.DDS.256.66.0F38.W0 B6 /r VFMADDSUB231PS ymm1, ymm2, ymm3/m256	RVM	V/V	FMA	Multiply packed single-precision floating-point values from ymm2 and ymm3/mem, add/subtract elements in ymm1 and put result in ymm1.
EVEX.DDS.128.66.0F38.W0 A6 /r VFMADDSUB213PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	FV	V/V	AVX512VL AVX512F	Multiply packed single-precision floating-point values from xmm1 and xmm2, add/subtract elements in xmm3/m128/m32bcst and put result in xmm1 subject to writemask k1.
EVEX.DDS.128.66.0F38.W0 B6 /r VFMADDSUB231PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	FV	V/V	AVX512VL AVX512F	Multiply packed single-precision floating-point values from xmm2 and xmm3/m128/m32bcst, add/subtract elements in xmm1 and put result in xmm1 subject to writemask k1.
EVEX.DDS.128.66.0F38.W0 96 /r VFMADDSUB132PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	FV	V/V	AVX512VL AVX512F	Multiply packed single-precision floating-point values from xmm1 and xmm3/m128/m32bcst, add/subtract elements in zmm2 and put result in xmm1 subject to writemask k1.
EVEX.DDS.256.66.0F38.W0 A6 /r VFMADDSUB213PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	FV	V/V	AVX512VL AVX512F	Multiply packed single-precision floating-point values from ymm1 and ymm2, add/subtract elements in ymm3/m256/m32bcst and put result in ymm1 subject to writemask k1.
EVEX.DDS.256.66.0F38.W0 B6 /r VFMADDSUB231PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	FV	V/V	AVX512VL AVX512F	Multiply packed single-precision floating-point values from ymm2 and ymm3/m256/m32bcst, add/subtract elements in ymm1 and put result in ymm1 subject to writemask k1.
EVEX.DDS.256.66.0F38.W0 96 /r VFMADDSUB132PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	FV	V/V	AVX512VL AVX512F	Multiply packed single-precision floating-point values from ymm1 and ymm3/m256/m32bcst, add/subtract elements in ymm2 and put result in ymm1 subject to writemask k1.
EVEX.DDS.512.66.0F38.W0 A6 /r VFMADDSUB213PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}	FV	V/V	AVX512F	Multiply packed single-precision floating-point values from zmm1 and zmm2, add/subtract elements in zmm3/m512/m32bcst and put result in zmm1 subject to writemask k1.
EVEX.DDS.512.66.0F38.W0 B6 /r VFMADDSUB231PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}	FV	V/V	AVX512F	Multiply packed single-precision floating-point values from zmm2 and zmm3/m512/m32bcst, add/subtract elements in zmm1 and put result in zmm1 subject to writemask k1.
EVEX.DDS.512.66.0F38.W0 96 /r VFMADDSUB132PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}	FV	V/V	AVX512F	Multiply packed single-precision floating-point values from zmm1 and zmm3/m512/m32bcst, add/subtract elements in zmm2 and put result in zmm1 subject to writemask k1.

## Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RVM	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FV	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

## Description

**VFMADDSUB132PS:** Multiplies the four, eight or sixteen packed single-precision floating-point values from the first source operand to the corresponding packed single-precision floating-point values in the third source operand. From the infinite precision intermediate result, adds the odd single-precision floating-point elements and subtracts the even single-precision floating-point values in the second source operand, performs rounding and stores the resulting packed single-precision floating-point values to the destination operand (first source operand).

**VFMADDSUB213PS:** Multiplies the four, eight or sixteen packed single-precision floating-point values from the second source operand to the corresponding packed single-precision floating-point values in the first source operand. From the infinite precision intermediate result, adds the odd single-precision floating-point elements and subtracts the even single-precision floating-point values in the third source operand, performs rounding and stores the resulting packed single-precision floating-point values to the destination operand (first source operand).

**VFMADDSUB231PS:** Multiplies the four, eight or sixteen packed single-precision floating-point values from the second source operand to the corresponding packed single-precision floating-point values in the third source operand. From the infinite precision intermediate result, adds the odd single-precision floating-point elements and subtracts the even single-precision floating-point values in the first source operand, performs rounding and stores the resulting packed single-precision floating-point values to the destination operand (first source operand).

**EVEX encoded versions:** The destination operand (also first source operand) and the second source operand are ZMM/YMM/XMM register. The third 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 32-bit memory location. The destination operand is conditionally updated with write mask k1.

**VEX.256 encoded version:** The destination operand (also first source operand) is a YMM register and encoded in `reg_field`. The second source operand is a YMM register and encoded in `VEX.vvvv`. The third source operand is a YMM register or a 256-bit memory location and encoded in `rm_field`.

**VEX.128 encoded version:** The destination operand (also first source operand) is a XMM register and encoded in `reg_field`. The second source operand is a XMM register and encoded in `VEX.vvvv`. The third source operand is a XMM register or a 128-bit memory location and encoded in `rm_field`. The upper 128 bits of the YMM destination register are zeroed.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NaNs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column.

## Operation

In the operations below, "\*" and "+" symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding).

**VFMADDSUB132PS DEST, SRC2, SRC3**

```

IF (VEX.128) THEN
    MAXNUM ← 2
ELSEIF (VEX.256)
    MAXNUM ← 4
FI
For i = 0 to MAXNUM - 1{
    n ← 64*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR(DEST[n+31:n]*SRC3[n+31:n] - SRC2[n+31:n])
    DEST[n+63:n+32] ← RoundFPControl_MXCSR(DEST[n+63:n+32]*SRC3[n+63:n+32] + SRC2[n+63:n+32])
}
IF (VEX.128) THEN

```



```

    DEST[MAX_VL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAX_VL-1:256] ← 0
FI

```

#### **VFMADDSUB213PS DEST, SRC2, SRC3**

```

IF (VEX.128) THEN
    MAXNUM ← 2
ELSEIF (VEX.256)
    MAXNUM ← 4
FI
For i = 0 to MAXNUM - 1{
    n ← 64*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR(SRC2[n+31:n]*DEST[n+31:n] - SRC3[n+31:n])
    DEST[n+63:n+32] ← RoundFPControl_MXCSR(SRC2[n+63:n+32]*DEST[n+63:n+32] + SRC3[n+63:n+32])
}
IF (VEX.128) THEN
    DEST[MAX_VL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAX_VL-1:256] ← 0
FI

```

#### **VFMADDSUB231PS DEST, SRC2, SRC3**

```

IF (VEX.128) THEN
    MAXNUM ← 2
ELSEIF (VEX.256)
    MAXNUM ← 4
FI
For i = 0 to MAXNUM - 1{
    n ← 64*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR(SRC2[n+31:n]*SRC3[n+31:n] - DEST[n+31:n])
    DEST[n+63:n+32] ← RoundFPControl_MXCSR(SRC2[n+63:n+32]*SRC3[n+63:n+32] + DEST[n+63:n+32])
}
IF (VEX.128) THEN
    DEST[MAX_VL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAX_VL-1:256] ← 0
FI

```

#### **VFMADDSUB132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)**

```

(KL, VL) (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN DEST[i+31:i] ←
                    RoundFPControl(DEST[i+31:i]*SRC3[j+31:i] - SRC2[j+31:i])

```

```

        ELSE DEST[j+31:i] ←
            RoundFPControl(DEST[j+31:i]*SRC3[j+31:i] + SRC2[j+31:i])
    FI
ELSE
    IF *merging-masking*           ; merging-masking
        THEN *DEST[j+31:i] remains unchanged*
    ELSE                             ; zeroing-masking
        DEST[j+31:i] ← 0
    FI
FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VFMADDSUB132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)**  
(KL, VL) = (4, 128), (8, 256), (16, 512)

```

FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN
                    IF (EVEX.b = 1)
                        THEN
                            DEST[j+31:i] ←
                                RoundFPControl_MXCSR(DEST[j+31:i]*SRC3[31:0] - SRC2[j+31:i])
                        ELSE
                            DEST[j+31:i] ←
                                RoundFPControl_MXCSR(DEST[j+31:i]*SRC3[j+31:i] - SRC2[j+31:i])
                        FI;
                    ELSE
                        IF (EVEX.b = 1)
                            THEN
                                DEST[j+31:i] ←
                                    RoundFPControl_MXCSR(DEST[j+31:i]*SRC3[31:0] + SRC2[j+31:i])
                            ELSE
                                DEST[j+31:i] ←
                                    RoundFPControl_MXCSR(DEST[j+31:i]*SRC3[j+31:i] + SRC2[j+31:i])
                            FI;
                        FI;
                    ELSE
                        IF *merging-masking*           ; merging-masking
                            THEN *DEST[j+31:i] remains unchanged*
                        ELSE                             ; zeroing-masking
                            DEST[j+31:i] ← 0
                        FI
                    FI;
                ELSE
                    IF *merging-masking*           ; merging-masking
                        THEN *DEST[j+31:i] remains unchanged*
                    ELSE                             ; zeroing-masking
                        DEST[j+31:i] ← 0
                    FI
                FI;
            ELSE
                DEST[MAX_VL-1:VL] ← 0
            ENDFOR

```

**VFMADDSUB213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

IF (VL = 512) AND (EVEX.b = 1)

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\*

THEN

IF j \*is even\*

THEN DEST[i+31:i] ←

RoundFPControl(SRC2[i+31:i]\*DEST[i+31:i] - SRC3[i+31:i])

ELSE DEST[i+31:i] ←

RoundFPControl(SRC2[i+31:i]\*DEST[i+31:i] + SRC3[i+31:i])

FI

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+31:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VFMADDSUB213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\*

THEN

IF j \*is even\*

THEN

IF (EVEX.b = 1)

THEN

DEST[i+31:i] ←

RoundFPControl\_MXCSR(SRC2[i+31:i]\*DEST[i+31:i] - SRC3[31:0])

ELSE

DEST[i+31:i] ←

RoundFPControl\_MXCSR(SRC2[i+31:i]\*DEST[i+31:i] - SRC3[i+31:i])

FI;

ELSE

IF (EVEX.b = 1)

THEN

DEST[i+31:i] ←

RoundFPControl\_MXCSR(SRC2[i+31:i]\*DEST[i+31:i] + SRC3[31:0])

ELSE

DEST[i+31:i] ←

RoundFPControl\_MXCSR(SRC2[i+31:i]\*DEST[i+31:i] + SRC3[i+31:i])

FI;

```

        FI
    ELSE
        IF *merging-masking*           ; merging-masking
            THEN *DEST[j+31:i] remains unchanged*
        ELSE                             ; zeroing-masking
            DEST[j+31:i] ← 0
        FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VFMADDSUB231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

IF (VL = 512) AND (EVEX.b = 1)

```

    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
    FI;
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN DEST[j+31:i] ←
                    RoundFPControl(SRC2[j+31:i]*SRC3[j+31:i] - DEST[j+31:i])
                ELSE DEST[j+31:i] ←
                    RoundFPControl(SRC2[j+31:i]*SRC3[j+31:i] + DEST[j+31:i])
            FI
        ELSE
            IF *merging-masking*           ; merging-masking
                THEN *DEST[j+31:i] remains unchanged*
            ELSE                             ; zeroing-masking
                DEST[j+31:i] ← 0
            FI
        FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VFMADDSUB231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

```

FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN
                    IF (EVEX.b = 1)
                        THEN
                            DEST[j+31:i] ←
                                RoundFPControl_MXCSR(SRC2[j+31:i]*SRC3[31:0] - DEST[j+31:i])
                        ELSE
                            DEST[j+31:i] ←

```

```

        RoundFPControl_MXCSR(SRC2[j+31:i]*SRC3[j+31:i] - DEST[j+31:i])
    FI;
ELSE
    IF (EVEX.b = 1)
        THEN
            DEST[j+31:i] ←
            RoundFPControl_MXCSR(SRC2[j+31:i]*SRC3[31:0] + DEST[j+31:i])
        ELSE
            DEST[j+31:i] ←
            RoundFPControl_MXCSR(SRC2[j+31:i]*SRC3[j+31:i] + DEST[j+31:i])
    FI;
FI
ELSE
    IF *merging-masking*                ; merging-masking
    THEN *DEST[j+31:i] remains unchanged*
    ELSE                                  ; zeroing-masking
        DEST[j+31:i] ← 0
    FI
FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

### Intel C/C++ Compiler Intrinsic Equivalent

```

VFMADDSUBxxxPS __m512_mm512_fmaddsub_ps(__m512 a, __m512 b, __m512 c);
VFMADDSUBxxxPS __m512_mm512_fmaddsub_round_ps(__m512 a, __m512 b, __m512 c, int r);
VFMADDSUBxxxPS __m512_mm512_mask_fmaddsub_ps(__m512 a, __mmask16 k, __m512 b, __m512 c);
VFMADDSUBxxxPS __m512_mm512_maskz_fmaddsub_ps(__mmask16 k, __m512 a, __m512 b, __m512 c);
VFMADDSUBxxxPS __m512_mm512_mask3_fmaddsub_ps(__m512 a, __m512 b, __m512 c, __mmask16 k);
VFMADDSUBxxxPS __m512_mm512_mask_fmaddsub_round_ps(__m512 a, __mmask16 k, __m512 b, __m512 c, int r);
VFMADDSUBxxxPS __m512_mm512_maskz_fmaddsub_round_ps(__mmask16 k, __m512 a, __m512 b, __m512 c, int r);
VFMADDSUBxxxPS __m512_mm512_mask3_fmaddsub_round_ps(__m512 a, __m512 b, __m512 c, __mmask16 k, int r);
VFMADDSUBxxxPS __m256_mm256_mask_fmaddsub_ps(__m256 a, __mmask8 k, __m256 b, __m256 c);
VFMADDSUBxxxPS __m256_mm256_maskz_fmaddsub_ps(__mmask8 k, __m256 a, __m256 b, __m256 c);
VFMADDSUBxxxPS __m256_mm256_mask3_fmaddsub_ps(__m256 a, __m256 b, __m256 c, __mmask8 k);
VFMADDSUBxxxPS __m128_mm_mask_fmaddsub_ps(__m128 a, __mmask8 k, __m128 b, __m128 c);
VFMADDSUBxxxPS __m128_mm_maskz_fmaddsub_ps(__mmask8 k, __m128 a, __m128 b, __m128 c);
VFMADDSUBxxxPS __m128_mm_mask3_fmaddsub_ps(__m128 a, __m128 b, __m128 c, __mmask8 k);
VFMADDSUBxxxPS __m128_mm_fmaddsub_ps (__m128 a, __m128 b, __m128 c);
VFMADDSUBxxxPS __m256_mm256_fmaddsub_ps (__m256 a, __m256 b, __m256 c);

```

### SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

### Other Exceptions

VEX-encoded instructions, see Exceptions Type 2.

EVEX-encoded instructions, see Exceptions Type E2.

## VFMSUBADD132PD/VFMSUBADD213PD/VFMSUBADD231PD—Fused Multiply-Alternating Subtract/Add of Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.DDS.128.66.0F38.W1 97 /r VFMSUBADD132PD xmm1, xmm2, xmm3/m128	RVM	V/V	FMA	Multiply packed double-precision floating-point values from xmm1 and xmm3/mem, subtract/add elements in xmm2 and put result in xmm1.
VEX.DDS.128.66.0F38.W1 A7 /r VFMSUBADD213PD xmm1, xmm2, xmm3/m128	RVM	V/V	FMA	Multiply packed double-precision floating-point values from xmm1 and xmm2, subtract/add elements in xmm3/mem and put result in xmm1.
VEX.DDS.128.66.0F38.W1 B7 /r VFMSUBADD231PD xmm1, xmm2, xmm3/m128	RVM	V/V	FMA	Multiply packed double-precision floating-point values from xmm2 and xmm3/mem, subtract/add elements in xmm1 and put result in xmm1.
VEX.DDS.256.66.0F38.W1 97 /r VFMSUBADD132PD ymm1, ymm2, ymm3/m256	RVM	V/V	FMA	Multiply packed double-precision floating-point values from ymm1 and ymm3/mem, subtract/add elements in ymm2 and put result in ymm1.
VEX.DDS.256.66.0F38.W1 A7 /r VFMSUBADD213PD ymm1, ymm2, ymm3/m256	RVM	V/V	FMA	Multiply packed double-precision floating-point values from ymm1 and ymm2, subtract/add elements in ymm3/mem and put result in ymm1.
VEX.DDS.256.66.0F38.W1 B7 /r VFMSUBADD231PD ymm1, ymm2, ymm3/m256	RVM	V/V	FMA	Multiply packed double-precision floating-point values from ymm2 and ymm3/mem, subtract/add elements in ymm1 and put result in ymm1.
EVEX.DDS.128.66.0F38.W1 97 /r VFMSUBADD132PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	FV	V/V	AVX512VL AVX512F	Multiply packed double-precision floating-point values from xmm1 and xmm3/m128/m64bcst, subtract/add elements in xmm2 and put result in xmm1 subject to writemask k1.
EVEX.DDS.128.66.0F38.W1 A7 /r VFMSUBADD213PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	FV	V/V	AVX512VL AVX512F	Multiply packed double-precision floating-point values from xmm1 and xmm2, subtract/add elements in xmm3/m128/m64bcst and put result in xmm1 subject to writemask k1.
EVEX.DDS.128.66.0F38.W1 B7 /r VFMSUBADD231PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	FV	V/V	AVX512VL AVX512F	Multiply packed double-precision floating-point values from xmm2 and xmm3/m128/m64bcst, subtract/add elements in xmm1 and put result in xmm1 subject to writemask k1.
EVEX.DDS.256.66.0F38.W1 97 /r VFMSUBADD132PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	FV	V/V	AVX512VL AVX512F	Multiply packed double-precision floating-point values from ymm1 and ymm3/m256/m64bcst, subtract/add elements in ymm2 and put result in ymm1 subject to writemask k1.
EVEX.DDS.256.66.0F38.W1 A7 /r VFMSUBADD213PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	FV	V/V	AVX512VL AVX512F	Multiply packed double-precision floating-point values from ymm1 and ymm2, subtract/add elements in ymm3/m256/m64bcst and put result in ymm1 subject to writemask k1.
EVEX.DDS.256.66.0F38.W1 B7 /r VFMSUBADD231PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	FV	V/V	AVX512VL AVX512F	Multiply packed double-precision floating-point values from ymm2 and ymm3/m256/m64bcst, subtract/add elements in ymm1 and put result in ymm1 subject to writemask k1.
EVEX.DDS.512.66.0F38.W1 97 /r VFMSUBADD132PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	FV	V/V	AVX512F	Multiply packed double-precision floating-point values from zmm1 and zmm3/m512/m64bcst, subtract/add elements in zmm2 and put result in zmm1 subject to writemask k1.

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.DDS.512.66.0F38.W1 A7 /r VFMSUBADD213PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	FV	V/V	AVX512F	Multiply packed double-precision floating-point values from zmm1 and zmm2, subtract/add elements in zmm3/m512/m64bcst and put result in zmm1 subject to writemask k1.
EVEX.DDS.512.66.0F38.W1 B7 /r VFMSUBADD231PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	FV	V/V	AVX512F	Multiply packed double-precision floating-point values from zmm2 and zmm3/m512/m64bcst, subtract/add elements in zmm1 and put result in zmm1 subject to writemask k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RVM	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FV	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

**VFMSUBADD132PD:** Multiplies the two, four, or eight packed double-precision floating-point values from the first source operand to the two or four packed double-precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the odd double-precision floating-point elements and adds the even double-precision floating-point values in the second source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand (first source operand).

**VFMSUBADD213PD:** Multiplies the two, four, or eight packed double-precision floating-point values from the second source operand to the two or four packed double-precision floating-point values in the first source operand. From the infinite precision intermediate result, subtracts the odd double-precision floating-point elements and adds the even double-precision floating-point values in the third source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand (first source operand).

**VFMSUBADD231PD:** Multiplies the two, four, or eight packed double-precision floating-point values from the second source operand to the two or four packed double-precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the odd double-precision floating-point elements and adds the even double-precision floating-point values in the first source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand (first source operand).

**EVEX encoded versions:** The destination operand (also first source operand) and the second source operand are ZMM/YMM/XMM register. The third 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 64-bit memory location. The destination operand is conditionally updated with write mask k1.

**VEX.256 encoded version:** The destination operand (also first source operand) is a YMM register and encoded in `reg_field`. The second source operand is a YMM register and encoded in `VEX.vvvv`. The third source operand is a YMM register or a 256-bit memory location and encoded in `rm_field`.

**VEX.128 encoded version:** The destination operand (also first source operand) is a XMM register and encoded in `reg_field`. The second source operand is a XMM register and encoded in `VEX.vvvv`. The third source operand is a XMM register or a 128-bit memory location and encoded in `rm_field`. The upper 128 bits of the YMM destination register are zeroed.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NaNs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column.

**Operation**

In the operations below, “\*” and “+” symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding).

**VFMSUBADD132PD DEST, SRC2, SRC3**

```
IF (VEX.128) THEN
    DEST[63:0] ← RoundFPControl_MXCSR(DEST[63:0]*SRC3[63:0] + SRC2[63:0])
    DEST[127:64] ← RoundFPControl_MXCSR(DEST[127:64]*SRC3[127:64] - SRC2[127:64])
    DEST[MAX_VL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[63:0] ← RoundFPControl_MXCSR(DEST[63:0]*SRC3[63:0] + SRC2[63:0])
    DEST[127:64] ← RoundFPControl_MXCSR(DEST[127:64]*SRC3[127:64] - SRC2[127:64])
    DEST[191:128] ← RoundFPControl_MXCSR(DEST[191:128]*SRC3[191:128] + SRC2[191:128])
    DEST[255:192] ← RoundFPControl_MXCSR(DEST[255:192]*SRC3[255:192] - SRC2[255:192])
FI
```

**VFMSUBADD213PD DEST, SRC2, SRC3**

```
IF (VEX.128) THEN
    DEST[63:0] ← RoundFPControl_MXCSR(SRC2[63:0]*DEST[63:0] + SRC3[63:0])
    DEST[127:64] ← RoundFPControl_MXCSR(SRC2[127:64]*DEST[127:64] - SRC3[127:64])
    DEST[MAX_VL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[63:0] ← RoundFPControl_MXCSR(SRC2[63:0]*DEST[63:0] + SRC3[63:0])
    DEST[127:64] ← RoundFPControl_MXCSR(SRC2[127:64]*DEST[127:64] - SRC3[127:64])
    DEST[191:128] ← RoundFPControl_MXCSR(SRC2[191:128]*DEST[191:128] + SRC3[191:128])
    DEST[255:192] ← RoundFPControl_MXCSR(SRC2[255:192]*DEST[255:192] - SRC3[255:192])
FI
```

**VFMSUBADD231PD DEST, SRC2, SRC3**

```
IF (VEX.128) THEN
    DEST[63:0] ← RoundFPControl_MXCSR(SRC2[63:0]*SRC3[63:0] + DEST[63:0])
    DEST[127:64] ← RoundFPControl_MXCSR(SRC2[127:64]*SRC3[127:64] - DEST[127:64])
    DEST[MAX_VL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[63:0] ← RoundFPControl_MXCSR(SRC2[63:0]*SRC3[63:0] + DEST[63:0])
    DEST[127:64] ← RoundFPControl_MXCSR(SRC2[127:64]*SRC3[127:64] - DEST[127:64])
    DEST[191:128] ← RoundFPControl_MXCSR(SRC2[191:128]*SRC3[191:128] + DEST[191:128])
    DEST[255:192] ← RoundFPControl_MXCSR(SRC2[255:192]*SRC3[255:192] - DEST[255:192])
FI
```

**VFMSUBADD132PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

IF (VL = 512) AND (EVEX.b = 1)

```
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
```

FI;

FOR j ← 0 TO KL-1

```
    i ← j * 64
```

```
    IF k1[j] OR *no writemask*
```

```
        THEN
```

```
            IF j *is even*
```

```
                THEN DEST[i+63:i] ←
```



```

        RoundFPControl(DEST[i+63:i]*SRC3[i+63:i] + SRC2[i+63:i])
    ELSE DEST[i+63:i] ←
        RoundFPControl(DEST[i+63:i]*SRC3[i+63:i] - SRC2[i+63:i])
    FI
ELSE
    IF *merging-masking*           ; merging-masking
    THEN *DEST[i+63:i] remains unchanged*
    ELSE                             ; zeroing-masking
        DEST[i+63:i] ← 0
    FI
FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VFMSUBADD132PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 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 j *is even*
        THEN
            IF (EVEX.b = 1)
            THEN
                DEST[i+63:i] ←
                RoundFPControl_MXCSR(DEST[i+63:i]*SRC3[63:0] + SRC2[i+63:i])
            ELSE
                DEST[i+63:i] ←
                RoundFPControl_MXCSR(DEST[i+63:i]*SRC3[i+63:i] + SRC2[i+63:i])
            FI;
        ELSE
            IF (EVEX.b = 1)
            THEN
                DEST[i+63:i] ←
                RoundFPControl_MXCSR(DEST[i+63:i]*SRC3[63:0] - SRC2[i+63:i])
            ELSE
                DEST[i+63:i] ←
                RoundFPControl_MXCSR(DEST[i+63:i]*SRC3[i+63:i] - SRC2[i+63:i])
            FI;
        FI
    ELSE
        IF *merging-masking*           ; merging-masking
        THEN *DEST[i+63:i] remains unchanged*
        ELSE                             ; zeroing-masking
            DEST[i+63:i] ← 0
        FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VFMSUBADD213PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

IF (VL = 512) AND (EVEX.b = 1)

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN

IF j \*is even\*

THEN DEST[i+63:i] ←

RoundFPControl(SRC2[i+63:i]\*DEST[i+63:i] + SRC3[i+63:i])

ELSE DEST[i+63:i] ←

RoundFPControl(SRC2[i+63:i]\*DEST[i+63:i] - SRC3[i+63:i])

FI

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VFMSUBADD213PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 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 j \*is even\*

THEN

IF (EVEX.b = 1)

THEN

DEST[i+63:i] ←

RoundFPControl\_MXCSR(SRC2[i+63:i]\*DEST[i+63:i] + SRC3[63:0])

ELSE

DEST[i+63:i] ←

RoundFPControl\_MXCSR(SRC2[i+63:i]\*DEST[i+63:i] + SRC3[i+63:i])

FI;

ELSE

IF (EVEX.b = 1)

THEN

DEST[i+63:i] ←

RoundFPControl\_MXCSR(SRC2[i+63:i]\*DEST[i+63:i] - SRC3[63:0])

ELSE

DEST[i+63:i] ←

RoundFPControl\_MXCSR(SRC2[i+63:i]\*DEST[i+63:i] - SRC3[i+63:i])

FI;

```

        FI
    ELSE
        IF *merging-masking*           ; merging-masking
            THEN *DEST[i+63:i] remains unchanged*
        ELSE                             ; zeroing-masking
            DEST[i+63:i] ← 0
        FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VFMSUBADD231PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

IF (VL = 512) AND (EVEX.b = 1)

```

    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
    FI;
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN DEST[i+63:i] ←
                    RoundFPControl(SRC2[i+63:i]*SRC3[i+63:i] + DEST[i+63:i])
            ELSE DEST[i+63:i] ←
                RoundFPControl(SRC2[i+63:i]*SRC3[i+63:i] - DEST[i+63:i])
            FI
        ELSE
            IF *merging-masking*           ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
            ELSE                             ; zeroing-masking
                DEST[i+63:i] ← 0
            FI
        FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VFMSUBADD231PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 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 j *is even*
                THEN
                    IF (EVEX.b = 1)
                        THEN
                            DEST[i+63:i] ←
                                RoundFPControl_MXCSR(SRC2[i+63:i]*SRC3[63:0] + DEST[i+63:i])
                        ELSE
                            DEST[i+63:i] ←

```

```

        RoundFPControl_MXCSR(SRC2[i+63:i]*SRC3[i+63:i] + DEST[i+63:i])
        FI;
    ELSE
        IF (EVEX.b = 1)
            THEN
                DEST[i+63:i] ←
                RoundFPControl_MXCSR(SRC2[i+63:i]*SRC3[63:0] - DEST[i+63:i])

            ELSE
                DEST[i+63:i] ←
                RoundFPControl_MXCSR(SRC2[i+63:i]*SRC3[i+63:i] - DEST[i+63:i])
        FI;
    FI
ELSE
    IF *merging-masking*                ; merging-masking
    THEN *DEST[i+63:i] remains unchanged*
    ELSE                                  ; zeroing-masking
        DEST[i+63:i] ← 0
    FI
FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

### Intel C/C++ Compiler Intrinsic Equivalent

```

VFMSUBADDxxxPD __m512d __mm512_fmsubadd_pd(__m512d a, __m512d b, __m512d c);
VFMSUBADDxxxPD __m512d __mm512_fmsubadd_round_pd(__m512d a, __m512d b, __m512d c, int r);
VFMSUBADDxxxPD __m512d __mm512_mask_fmsubadd_pd(__m512d a, __mmask8 k, __m512d b, __m512d c);
VFMSUBADDxxxPD __m512d __mm512_maskz_fmsubadd_pd(__mmask8 k, __m512d a, __m512d b, __m512d c);
VFMSUBADDxxxPD __m512d __mm512_mask3_fmsubadd_pd(__m512d a, __m512d b, __m512d c, __mmask8 k);
VFMSUBADDxxxPD __m512d __mm512_mask_fmsubadd_round_pd(__m512d a, __mmask8 k, __m512d b, __m512d c, int r);
VFMSUBADDxxxPD __m512d __mm512_maskz_fmsubadd_round_pd(__mmask8 k, __m512d a, __m512d b, __m512d c, int r);
VFMSUBADDxxxPD __m512d __mm512_mask3_fmsubadd_round_pd(__m512d a, __m512d b, __m512d c, __mmask8 k, int r);
VFMSUBADDxxxPD __m256d __mm256_mask_fmsubadd_pd(__m256d a, __mmask8 k, __m256d b, __m256d c);
VFMSUBADDxxxPD __m256d __mm256_maskz_fmsubadd_pd(__mmask8 k, __m256d a, __m256d b, __m256d c);
VFMSUBADDxxxPD __m256d __mm256_mask3_fmsubadd_pd(__m256d a, __m256d b, __m256d c, __mmask8 k);
VFMSUBADDxxxPD __m128d __mm_mask_fmsubadd_pd(__m128d a, __mmask8 k, __m128d b, __m128d c);
VFMSUBADDxxxPD __m128d __mm_maskz_fmsubadd_pd(__mmask8 k, __m128d a, __m128d b, __m128d c);
VFMSUBADDxxxPD __m128d __mm_mask3_fmsubadd_pd(__m128d a, __m128d b, __m128d c, __mmask8 k);
VFMSUBADDxxxPD __m128d __mm_fmsubadd_pd(__m128d a, __m128d b, __m128d c);
VFMSUBADDxxxPD __m256d __mm256_fmsubadd_pd(__m256d a, __m256d b, __m256d c);

```

### SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

### Other Exceptions

VEX-encoded instructions, see Exceptions Type 2.

EVEX-encoded instructions, see Exceptions Type E2.

## VFMSUBADD132PS/VFMSUBADD213PS/VFMSUBADD231PS—Fused Multiply-Alternating Subtract/Add of Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.DDS.128.66.0F38.W0 97 /r VFMSUBADD132PS xmm1, xmm2, xmm3/m128	RVM	V/V	FMA	Multiply packed single-precision floating-point values from xmm1 and xmm3/mem, subtract/add elements in xmm2 and put result in xmm1.
VEX.DDS.128.66.0F38.W0 A7 /r VFMSUBADD213PS xmm1, xmm2, xmm3/m128	RVM	V/V	FMA	Multiply packed single-precision floating-point values from xmm1 and xmm2, subtract/add elements in xmm3/mem and put result in xmm1.
VEX.DDS.128.66.0F38.W0 B7 /r VFMSUBADD231PS xmm1, xmm2, xmm3/m128	RVM	V/V	FMA	Multiply packed single-precision floating-point values from xmm2 and xmm3/mem, subtract/add elements in xmm1 and put result in xmm1.
VEX.DDS.256.66.0F38.W0 97 /r VFMSUBADD132PS ymm1, ymm2, ymm3/m256	RVM	V/V	FMA	Multiply packed single-precision floating-point values from ymm1 and ymm3/mem, subtract/add elements in ymm2 and put result in ymm1.
VEX.DDS.256.66.0F38.W0 A7 /r VFMSUBADD213PS ymm1, ymm2, ymm3/m256	RVM	V/V	FMA	Multiply packed single-precision floating-point values from ymm1 and ymm2, subtract/add elements in ymm3/mem and put result in ymm1.
VEX.DDS.256.66.0F38.W0 B7 /r VFMSUBADD231PS ymm1, ymm2, ymm3/m256	RVM	V/V	FMA	Multiply packed single-precision floating-point values from ymm2 and ymm3/mem, subtract/add elements in ymm1 and put result in ymm1.
EVEX.DDS.128.66.0F38.W0 97 /r VFMSUBADD132PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	FV	V/V	AVX512VL AVX512F	Multiply packed single-precision floating-point values from xmm1 and xmm3/m128/m32bcst, subtract/add elements in xmm2 and put result in xmm1 subject to writemask k1.
EVEX.DDS.128.66.0F38.W0 A7 /r VFMSUBADD213PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	FV	V/V	AVX512VL AVX512F	Multiply packed single-precision floating-point values from xmm1 and xmm2, subtract/add elements in xmm3/m128/m32bcst and put result in xmm1 subject to writemask k1.
EVEX.DDS.128.66.0F38.W0 B7 /r VFMSUBADD231PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	FV	V/V	AVX512VL AVX512F	Multiply packed single-precision floating-point values from xmm2 and xmm3/m128/m32bcst, subtract/add elements in xmm1 and put result in xmm1 subject to writemask k1.
EVEX.DDS.256.66.0F38.W0 97 /r VFMSUBADD132PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	FV	V/V	AVX512VL AVX512F	Multiply packed single-precision floating-point values from ymm1 and ymm3/m256/m32bcst, subtract/add elements in ymm2 and put result in ymm1 subject to writemask k1.
EVEX.DDS.256.66.0F38.W0 A7 /r VFMSUBADD213PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	FV	V/V	AVX512VL AVX512F	Multiply packed single-precision floating-point values from ymm1 and ymm2, subtract/add elements in ymm3/m256/m32bcst and put result in ymm1 subject to writemask k1.
EVEX.DDS.256.66.0F38.W0 B7 /r VFMSUBADD231PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	FV	V/V	AVX512VL AVX512F	Multiply packed single-precision floating-point values from ymm2 and ymm3/m256/m32bcst, subtract/add elements in ymm1 and put result in ymm1 subject to writemask k1.
EVEX.DDS.512.66.0F38.W0 97 /r VFMSUBADD132PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}	FV	V/V	AVX512F	Multiply packed single-precision floating-point values from zmm1 and zmm3/m512/m32bcst, subtract/add elements in zmm2 and put result in zmm1 subject to writemask k1.
EVEX.DDS.512.66.0F38.W0 A7 /r VFMSUBADD213PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}	FV	V/V	AVX512F	Multiply packed single-precision floating-point values from zmm1 and zmm2, subtract/add elements in zmm3/m512/m32bcst and put result in zmm1 subject to writemask k1.
EVEX.DDS.512.66.0F38.W0 B7 /r VFMSUBADD231PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}	FV	V/V	AVX512F	Multiply packed single-precision floating-point values from zmm2 and zmm3/m512/m32bcst, subtract/add elements in zmm1 and put result in zmm1 subject to writemask k1.

## Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RVM	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FV	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

## Description

**VFMSUBADD132PS:** Multiplies the four, eight or sixteen packed single-precision floating-point values from the first source operand to the corresponding packed single-precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the odd single-precision floating-point elements and adds the even single-precision floating-point values in the second source operand, performs rounding and stores the resulting packed single-precision floating-point values to the destination operand (first source operand).

**VFMSUBADD213PS:** Multiplies the four, eight or sixteen packed single-precision floating-point values from the second source operand to the corresponding packed single-precision floating-point values in the first source operand. From the infinite precision intermediate result, subtracts the odd single-precision floating-point elements and adds the even single-precision floating-point values in the third source operand, performs rounding and stores the resulting packed single-precision floating-point values to the destination operand (first source operand).

**VFMSUBADD231PS:** Multiplies the four, eight or sixteen packed single-precision floating-point values from the second source operand to the corresponding packed single-precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the odd single-precision floating-point elements and adds the even single-precision floating-point values in the first source operand, performs rounding and stores the resulting packed single-precision floating-point values to the destination operand (first source operand).

**EVEX encoded versions:** The destination operand (also first source operand) and the second source operand are ZMM/YMM/XMM register. The third 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 32-bit memory location. The destination operand is conditionally updated with write mask k1.

**VEX.256 encoded version:** The destination operand (also first source operand) is a YMM register and encoded in `reg_field`. The second source operand is a YMM register and encoded in `VEX.vvvv`. The third source operand is a YMM register or a 256-bit memory location and encoded in `rm_field`.

**VEX.128 encoded version:** The destination operand (also first source operand) is a XMM register and encoded in `reg_field`. The second source operand is a XMM register and encoded in `VEX.vvvv`. The third source operand is a XMM register or a 128-bit memory location and encoded in `rm_field`. The upper 128 bits of the YMM destination register are zeroed.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NaNs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column.

## Operation

In the operations below, "\*" and "+" symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding).

**VFMSUBADD132PS DEST, SRC2, SRC3**

```

IF (VEX.128) THEN
    MAXNUM ← 2
ELSEIF (VEX.256)
    MAXNUM ← 4
FI
For i = 0 to MAXNUM - 1{
    n ← 64*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR(DEST[n+31:n]*SRC3[n+31:n] + SRC2[n+31:n])
    DEST[n+63:n+32] ← RoundFPControl_MXCSR(DEST[n+63:n+32]*SRC3[n+63:n+32] - SRC2[n+63:n+32])
}
IF (VEX.128) THEN

```

```

    DEST[MAX_VL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAX_VL-1:256] ← 0
FI

```

#### VFMSUBADD213PS DEST, SRC2, SRC3

```

IF (VEX.128) THEN
    MAXNUM ← 2
ELSEIF (VEX.256)
    MAXNUM ← 4
FI
For i = 0 to MAXNUM - 1{
    n ← 64*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR(SRC2[n+31:n]*DEST[n+31:n] + SRC3[n+31:n])
    DEST[n+63:n+32] ← RoundFPControl_MXCSR(SRC2[n+63:n+32]*DEST[n+63:n+32] - SRC3[n+63:n+32])
}
IF (VEX.128) THEN
    DEST[MAX_VL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAX_VL-1:256] ← 0
FI

```

#### VFMSUBADD231PS DEST, SRC2, SRC3

```

IF (VEX.128) THEN
    MAXNUM ← 2
ELSEIF (VEX.256)
    MAXNUM ← 4
FI
For i = 0 to MAXNUM - 1{
    n ← 64*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR(SRC2[n+31:n]*SRC3[n+31:n] + DEST[n+31:n])
    DEST[n+63:n+32] ← RoundFPControl_MXCSR(SRC2[n+63:n+32]*SRC3[n+63:n+32] - DEST[n+63:n+32])
}
IF (VEX.128) THEN
    DEST[MAX_VL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAX_VL-1:256] ← 0
FI

```

#### VFMSUBADD132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```

(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN DEST[i+31:i] ←
                    RoundFPControl(DEST[i+31:i]*SRC3[j+31:i] + SRC2[i+31:i])

```

```

        ELSE DEST[j+31:i] ←
            RoundFPControl(DEST[j+31:i]*SRC3[j+31:i] - SRC2[j+31:i])
    FI
ELSE
    IF *merging-masking*                ; merging-masking
        THEN *DEST[j+31:i] remains unchanged*
    ELSE                                ; zeroing-masking
        DEST[j+31:i] ← 0
    FI
FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VFMSUBADD132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)**  
 (KL, VL) = (4, 128), (8, 256), (16, 512)

```

FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN
                    IF (EVEX.b = 1)
                        THEN
                            DEST[j+31:i] ←
                                RoundFPControl_MXCSR(DEST[j+31:i]*SRC3[31:0] + SRC2[j+31:i])
                        ELSE
                            DEST[j+31:i] ←
                                RoundFPControl_MXCSR(DEST[j+31:i]*SRC3[j+31:i] + SRC2[j+31:i])
                    FI;
                ELSE
                    IF (EVEX.b = 1)
                        THEN
                            DEST[j+31:i] ←
                                RoundFPControl_MXCSR(DEST[j+31:i]*SRC3[31:0] - SRC2[j+31:i])
                        ELSE
                            DEST[j+31:i] ←
                                RoundFPControl_MXCSR(DEST[j+31:i]*SRC3[j+31:i] - SRC2[j+31:i])
                    FI;
                FI
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[j+31:i] remains unchanged*
            ELSE                                ; zeroing-masking
                DEST[j+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```



**VFMSUBADD213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

IF (VL = 512) AND (EVEX.b = 1)

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\*

THEN

IF j \*is even\*

THEN DEST[i+31:i] ←

RoundFPControl(SRC2[i+31:i]\*DEST[i+31:i] + SRC3[i+31:i])

ELSE DEST[i+31:i] ←

RoundFPControl(SRC2[i+31:i]\*DEST[i+31:i] - SRC3[i+31:i])

FI

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+31:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VFMSUBADD213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\*

THEN

IF j \*is even\*

THEN

IF (EVEX.b = 1)

THEN

DEST[i+31:i] ←

RoundFPControl\_MXCSR(SRC2[i+31:i]\*DEST[i+31:i] + SRC3[31:0])

ELSE

DEST[i+31:i] ←

RoundFPControl\_MXCSR(SRC2[i+31:i]\*DEST[i+31:i] + SRC3[i+31:i])

FI;

ELSE

IF (EVEX.b = 1)

THEN

DEST[i+31:i] ←

RoundFPControl\_MXCSR(SRC2[i+31:i]\*DEST[i+31:i] - SRC3[i+31:i])

ELSE

DEST[i+31:i] ←

RoundFPControl\_MXCSR(SRC2[i+31:i]\*DEST[i+31:i] - SRC3[31:0])

FI;

```

        FI
    ELSE
        IF *merging-masking*           ; merging-masking
            THEN *DEST[j+31:i] remains unchanged*
        ELSE                             ; zeroing-masking
            DEST[j+31:i] ← 0
        FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VFMSUBADD231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

IF (VL = 512) AND (EVEX.b = 1)

```

    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
    FI;
    FOR j ← 0 TO KL-1
        i ← j * 32
        IF k1[j] OR *no writemask*
            THEN
                IF j *is even*
                    THEN DEST[j+31:i] ←
                        RoundFPControl(SRC2[j+31:i]*SRC3[j+31:i] + DEST[j+31:i])
                    ELSE DEST[j+31:i] ←
                        RoundFPControl(SRC2[j+31:i]*SRC3[j+31:i] - DEST[j+31:i])
                FI
            ELSE
                IF *merging-masking*           ; merging-masking
                    THEN *DEST[j+31:i] remains unchanged*
                ELSE                             ; zeroing-masking
                    DEST[j+31:i] ← 0
                FI
            FI;
        ENDFOR
        DEST[MAX_VL-1:VL] ← 0

```

**VFMSUBADD231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

```

    FOR j ← 0 TO KL-1
        i ← j * 32
        IF k1[j] OR *no writemask*
            THEN
                IF j *is even*
                    THEN
                        IF (EVEX.b = 1)
                            THEN
                                DEST[j+31:i] ←
                                    RoundFPControl_MXCSR(SRC2[j+31:i]*SRC3[31:0] + DEST[j+31:i])
                            ELSE
                                DEST[j+31:i] ←

```

```

        RoundFPControl_MXCSR(SRC2[j+31:i]*SRC3[j+31:i] + DEST[j+31:i])
    FI;
ELSE
    IF (EVEX.b = 1)
        THEN
            DEST[j+31:i] ←
            RoundFPControl_MXCSR(SRC2[j+31:i]*SRC3[31:0] - DEST[j+31:i])
        ELSE
            DEST[j+31:i] ←
            RoundFPControl_MXCSR(SRC2[j+31:i]*SRC3[j+31:i] - DEST[j+31:i])
    FI;
FI
ELSE
    IF *merging-masking*                ; merging-masking
    THEN *DEST[j+31:i] remains unchanged*
    ELSE                                  ; zeroing-masking
        DEST[j+31:i] ← 0
    FI
FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

### Intel C/C++ Compiler Intrinsic Equivalent

```

VFMSUBADDxxxPS __m512_mm512_fmsubadd_ps(__m512 a, __m512 b, __m512 c);
VFMSUBADDxxxPS __m512_mm512_fmsubadd_round_ps(__m512 a, __m512 b, __m512 c, int r);
VFMSUBADDxxxPS __m512_mm512_mask_fmsubadd_ps(__m512 a, __mmask16 k, __m512 b, __m512 c);
VFMSUBADDxxxPS __m512_mm512_maskz_fmsubadd_ps(__mmask16 k, __m512 a, __m512 b, __m512 c);
VFMSUBADDxxxPS __m512_mm512_mask3_fmsubadd_ps(__m512 a, __m512 b, __m512 c, __mmask16 k);
VFMSUBADDxxxPS __m512_mm512_mask_fmsubadd_round_ps(__m512 a, __mmask16 k, __m512 b, __m512 c, int r);
VFMSUBADDxxxPS __m512_mm512_maskz_fmsubadd_round_ps(__mmask16 k, __m512 a, __m512 b, __m512 c, int r);
VFMSUBADDxxxPS __m512_mm512_mask3_fmsubadd_round_ps(__m512 a, __m512 b, __m512 c, __mmask16 k, int r);
VFMSUBADDxxxPS __m256_mm256_mask_fmsubadd_ps(__m256 a, __mmask8 k, __m256 b, __m256 c);
VFMSUBADDxxxPS __m256_mm256_maskz_fmsubadd_ps(__mmask8 k, __m256 a, __m256 b, __m256 c);
VFMSUBADDxxxPS __m256_mm256_mask3_fmsubadd_ps(__m256 a, __m256 b, __m256 c, __mmask8 k);
VFMSUBADDxxxPS __m128_mm_mask_fmsubadd_ps(__m128 a, __mmask8 k, __m128 b, __m128 c);
VFMSUBADDxxxPS __m128_mm_maskz_fmsubadd_ps(__mmask8 k, __m128 a, __m128 b, __m128 c);
VFMSUBADDxxxPS __m128_mm_mask3_fmsubadd_ps(__m128 a, __m128 b, __m128 c, __mmask8 k);
VFMSUBADDxxxPS __m128_mm_fmsubadd_ps (__m128 a, __m128 b, __m128 c);
VFMSUBADDxxxPS __m256_mm256_fmsubadd_ps (__m256 a, __m256 b, __m256 c);

```

### SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

### Other Exceptions

VEX-encoded instructions, see Exceptions Type 2.

EVEX-encoded instructions, see Exceptions Type E2.

## VFMSUB132PD/VFMSUB213PD/VFMSUB231PD—Fused Multiply-Subtract of Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.NDS.128.66.0F38.W1 9A /r VFMSUB132PD xmm1, xmm2, xmm3/m128	RVM	V/V	FMA	Multiply packed double-precision floating-point values from xmm1 and xmm3/mem, subtract xmm2 and put result in xmm1.
VEX.NDS.128.66.0F38.W1 AA /r VFMSUB213PD xmm1, xmm2, xmm3/m128	RVM	V/V	FMA	Multiply packed double-precision floating-point values from xmm1 and xmm2, subtract xmm3/mem and put result in xmm1.
VEX.NDS.128.66.0F38.W1 BA /r VFMSUB231PD xmm1, xmm2, xmm3/m128	RVM	V/V	FMA	Multiply packed double-precision floating-point values from xmm2 and xmm3/mem, subtract xmm1 and put result in xmm1.
VEX.NDS.256.66.0F38.W1 9A /r VFMSUB132PD ymm1, ymm2, ymm3/m256	RVM	V/V	FMA	Multiply packed double-precision floating-point values from ymm1 and ymm3/mem, subtract ymm2 and put result in ymm1.
VEX.NDS.256.66.0F38.W1 AA /r VFMSUB213PD ymm1, ymm2, ymm3/m256	RVM	V/V	FMA	Multiply packed double-precision floating-point values from ymm1 and ymm2, subtract ymm3/mem and put result in ymm1.
VEX.NDS.256.66.0F38.W1 BA /r VFMSUB231PD ymm1, ymm2, ymm3/m256	RVM	V/V	FMA	Multiply packed double-precision floating-point values from ymm2 and ymm3/mem, subtract ymm1 and put result in ymm1.S
EVEX.NDS.128.66.0F38.W1 9A /r VFMSUB132PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	FV	V/V	AVX512VL AVX512F	Multiply packed double-precision floating-point values from xmm1 and xmm3/m128/m64bcst, subtract xmm2 and put result in xmm1 subject to writemask k1.
EVEX.NDS.128.66.0F38.W1 AA /r VFMSUB213PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	FV	V/V	AVX512VL AVX512F	Multiply packed double-precision floating-point values from xmm1 and xmm2, subtract xmm3/m128/m64bcst and put result in xmm1 subject to writemask k1.
EVEX.NDS.128.66.0F38.W1 BA /r VFMSUB231PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	FV	V/V	AVX512VL AVX512F	Multiply packed double-precision floating-point values from xmm2 and xmm3/m128/m64bcst, subtract xmm1 and put result in xmm1 subject to writemask k1.
EVEX.NDS.256.66.0F38.W1 9A /r VFMSUB132PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	FV	V/V	AVX512VL AVX512F	Multiply packed double-precision floating-point values from ymm1 and ymm3/m256/m64bcst, subtract ymm2 and put result in ymm1 subject to writemask k1.
EVEX.NDS.256.66.0F38.W1 AA /r VFMSUB213PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	FV	V/V	AVX512VL AVX512F	Multiply packed double-precision floating-point values from ymm1 and ymm2, subtract ymm3/m256/m64bcst and put result in ymm1 subject to writemask k1.
EVEX.NDS.256.66.0F38.W1 BA /r VFMSUB231PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	FV	V/V	AVX512VL AVX512F	Multiply packed double-precision floating-point values from ymm2 and ymm3/m256/m64bcst, subtract ymm1 and put result in ymm1 subject to writemask k1.
EVEX.NDS.512.66.0F38.W1 9A /r VFMSUB132PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	FV	V/V	AVX512F	Multiply packed double-precision floating-point values from zmm1 and zmm3/m512/m64bcst, subtract zmm2 and put result in zmm1 subject to writemask k1.
EVEX.NDS.512.66.0F38.W1 AA /r VFMSUB213PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	FV	V/V	AVX512F	Multiply packed double-precision floating-point values from zmm1 and zmm2, subtract zmm3/m512/m64bcst and put result in zmm1 subject to writemask k1.
EVEX.NDS.512.66.0F38.W1 BA /r VFMSUB231PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	FV	V/V	AVX512F	Multiply packed double-precision floating-point values from zmm2 and zmm3/m512/m64bcst, subtract zmm1 and put result in zmm1 subject to writemask k1.

## Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RVM	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FV	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

## Description

Performs a set of SIMD multiply-subtract computation on packed double-precision floating-point values using three source operands and writes the multiply-subtract results in the destination operand. The destination operand is also the first source operand. The second operand must be a SIMD register. The third source operand can be a SIMD register or a memory location.

**VFMSUB132PD:** Multiplies the two, four or eight packed double-precision floating-point values from the first source operand to the two, four or eight packed double-precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the two, four or eight packed double-precision floating-point values in the second source operand, performs rounding and stores the resulting two, four or eight packed double-precision floating-point values to the destination operand (first source operand).

**VFMSUB213PD:** Multiplies the two, four or eight packed double-precision floating-point values from the second source operand to the two, four or eight packed double-precision floating-point values in the first source operand. From the infinite precision intermediate result, subtracts the two, four or eight packed double-precision floating-point values in the third source operand, performs rounding and stores the resulting two, four or eight packed double-precision floating-point values to the destination operand (first source operand).

**VFMSUB231PD:** Multiplies the two, four or eight packed double-precision floating-point values from the second source to the two, four or eight packed double-precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the two, four or eight packed double-precision floating-point values in the first source operand, performs rounding and stores the resulting two, four or eight packed double-precision floating-point values to the destination operand (first source operand).

**EVEX encoded versions:** The destination operand (also first source operand) and the second source operand are ZMM/YMM/XMM register. The third 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 64-bit memory location. The destination operand is conditionally updated with write mask k1.

**VEX.256 encoded version:** The destination operand (also first source operand) is a YMM register and encoded in `reg_field`. The second source operand is a YMM register and encoded in `VEX.vvvv`. The third source operand is a YMM register or a 256-bit memory location and encoded in `rm_field`.

**VEX.128 encoded version:** The destination operand (also first source operand) is a XMM register and encoded in `reg_field`. The second source operand is a XMM register and encoded in `VEX.vvvv`. The third source operand is a XMM register or a 128-bit memory location and encoded in `rm_field`. The upper 128 bits of the YMM destination register are zeroed.

## Operation

In the operations below, “\*” and “-” symbols represent multiplication and subtraction with infinite precision inputs and outputs (no rounding).

**VFMSUB132PD DEST, SRC2, SRC3 (VEX encoded versions)**

```

IF (VEX.128) THEN
    MAXNUM ← 2
ELSEIF (VEX.256)
    MAXNUM ← 4
FI
For i = 0 to MAXNUM-1 {
    n ← 64*i;
    DEST[n+63:n] ← RoundFPControl_MXCSR(DEST[n+63:n]*SRC3[n+63:n] - SRC2[n+63:n])
}
IF (VEX.128) THEN
    DEST[MAX_VL-1:128] ← 0

```

```

ELSEIF (VEX.256)
    DEST[MAX_VL-1:256] ← 0
FI

```

**VFMSUB213PD DEST, SRC2, SRC3 (VEX encoded versions)**

```

IF (VEX.128) THEN
    MAXNUM ← 2
ELSEIF (VEX.256)
    MAXNUM ← 4
FI
For i = 0 to MAXNUM-1 {
    n ← 64*i;
    DEST[n+63:n] ← RoundFPControl_MXCSR(SRC2[n+63:n]*DEST[n+63:n] - SRC3[n+63:n])
}
IF (VEX.128) THEN
    DEST[MAX_VL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAX_VL-1:256] ← 0
FI

```

**VFMSUB231PD DEST, SRC2, SRC3 (VEX encoded versions)**

```

IF (VEX.128) THEN
    MAXNUM ← 2
ELSEIF (VEX.256)
    MAXNUM ← 4
FI
For i = 0 to MAXNUM-1 {
    n ← 64*i;
    DEST[n+63:n] ← RoundFPControl_MXCSR(SRC2[n+63:n]*SRC3[n+63:n] - DEST[n+63:n])
}
IF (VEX.128) THEN
    DEST[MAX_VL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAX_VL-1:256] ← 0
FI

```

**VFMSUB132PD DEST, SRC2, SRC3 (EVEX encoded versions, when src3 operand is a register)**

```

(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ←
            RoundFPControl(DEST[i+63:i]*SRC3[i+63:i] - SRC2[i+63:i])
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+63:i] ← 0

```

```

        FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VFMSUB132PD DEST, SRC2, SRC3 (EVEX encoded versions, when src3 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] ←
                    RoundFPControl_MXCSR(DEST[i+63:i]*SRC3[63:0] - SRC2[i+63:i])
                ELSE
                    DEST[i+63:i] ←
                    RoundFPControl_MXCSR(DEST[i+63:i]*SRC3[i+63:i] - SRC2[i+63:i])
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
            ELSE                                  ; zeroing-masking
                DEST[i+63:i] ← 0
            FI
        FI;
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VFMSUB213PD DEST, SRC2, SRC3 (EVEX encoded versions, when src3 operand is a register)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

IF (VL = 512) AND (EVEX.b = 1)

```

    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
    FI;
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ←
        RoundFPControl(SRC2[i+63:i]*DEST[i+63:i] - SRC3[i+63:i])
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[i+63:i] remains unchanged*
        ELSE                                  ; zeroing-masking
            DEST[i+63:i] ← 0
        FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VFMSUB213PD DEST, SRC2, SRC3 (EVEX encoded versions, when src3 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] ←
            RoundFPControl_MXCSR(SRC2[i+63:i]*DEST[i+63:i] - SRC3[63:0]
+31:i])
          ELSE
            DEST[i+63:i] ←
              RoundFPControl_MXCSR(SRC2[i+63:i]*DEST[i+63:i] - SRC3[i+63:i])
            FI;
          ELSE
            IF *merging-masking*           ; merging-masking
              THEN *DEST[i+63:i] remains unchanged*
            ELSE                             ; zeroing-masking
              DEST[i+63:i] ← 0
            FI
          FI;
        ENDFOR
      DEST[MAX_VL-1:VL] ← 0
  
```

**VFMSUB231PD DEST, SRC2, SRC3 (EVEX encoded versions, when src3 operand is a register)**  
 (KL, VL) = (2, 128), (4, 256), (8, 512)

```

IF (VL = 512) AND (EVEX.b = 1)
  THEN
    SET_RM(EVEX.RC);
  ELSE
    SET_RM(MXCSR.RM);
  FI;
FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] ←
      RoundFPControl(SRC2[i+63:i]*SRC3[i+63:i] - DEST[i+63:i])
    ELSE
      IF *merging-masking*           ; merging-masking
        THEN *DEST[i+63:i] remains unchanged*
      ELSE                             ; zeroing-masking
        DEST[i+63:i] ← 0
      FI
    FI;
  ENDFOR
  DEST[MAX_VL-1:VL] ← 0
  
```



**VFMSUB231PD DEST, SRC2, SRC3 (EVEX encoded versions, when src3 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] ←
            RoundFPControl_MXCSR(SRC2[i+63:i]*SRC3[63:0] - DEST[i+63:i])
        ELSE
          DEST[i+63:i] ←
            RoundFPControl_MXCSR(SRC2[i+63:i]*SRC3[i+63:i] - DEST[i+63:i])
        FI;
      ELSE
        IF *merging-masking*           ; merging-masking
          THEN *DEST[i+63:i] remains unchanged*
        ELSE                             ; zeroing-masking
          DEST[i+63:i] ← 0
        FI
      FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VFMSUBxxxPD __m512d __mm512_fmsub_pd(__m512d a, __m512d b, __m512d c);
VFMSUBxxxPD __m512d __mm512_fmsub_round_pd(__m512d a, __m512d b, __m512d c, int r);
VFMSUBxxxPD __m512d __mm512_mask_fmsub_pd(__m512d a, __mmask8 k, __m512d b, __m512d c);
VFMSUBxxxPD __m512d __mm512_maskz_fmsub_pd(__mmask8 k, __m512d a, __m512d b, __m512d c);
VFMSUBxxxPD __m512d __mm512_mask3_fmsub_pd(__m512d a, __m512d b, __m512d c, __mmask8 k);
VFMSUBxxxPD __m512d __mm512_mask_fmsub_round_pd(__m512d a, __mmask8 k, __m512d b, __m512d c, int r);
VFMSUBxxxPD __m512d __mm512_maskz_fmsub_round_pd(__mmask8 k, __m512d a, __m512d b, __m512d c, int r);
VFMSUBxxxPD __m512d __mm512_mask3_fmsub_round_pd(__m512d a, __m512d b, __m512d c, __mmask8 k, int r);
VFMSUBxxxPD __m256d __mm256_mask_fmsub_pd(__m256d a, __mmask8 k, __m256d b, __m256d c);
VFMSUBxxxPD __m256d __mm256_maskz_fmsub_pd(__mmask8 k, __m256d a, __m256d b, __m256d c);
VFMSUBxxxPD __m256d __mm256_mask3_fmsub_pd(__m256d a, __m256d b, __m256d c, __mmask8 k);
VFMSUBxxxPD __m128d __mm_mask_fmsub_pd(__m128d a, __mmask8 k, __m128d b, __m128d c);
VFMSUBxxxPD __m128d __mm_maskz_fmsub_pd(__mmask8 k, __m128d a, __m128d b, __m128d c);
VFMSUBxxxPD __m128d __mm_mask3_fmsub_pd(__m128d a, __m128d b, __m128d c, __mmask8 k);
VFMSUBxxxPD __m128d __mm_fmsub_pd (__m128d a, __m128d b, __m128d c);
VFMSUBxxxPD __m256d __mm256_fmsub_pd (__m256d a, __m256d b, __m256d c);

```

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal

**Other Exceptions**

VEX-encoded instructions, see Exceptions Type 2.

EVEX-encoded instructions, see Exceptions Type E2.

## VFMSUB132PS/VFMSUB213PS/VFMSUB231PS—Fused Multiply-Subtract of Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op/E n	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.NDS.128.66.0F38.W0 9A /r VFMSUB132PS xmm1, xmm2, xmm3/m128	RVM	V/V	FMA	Multiply packed single-precision floating-point values from xmm1 and xmm3/mem, subtract xmm2 and put result in xmm1.
VEX.NDS.128.66.0F38.W0 AA /r VFMSUB213PS xmm1, xmm2, xmm3/m128	RVM	V/V	FMA	Multiply packed single-precision floating-point values from xmm1 and xmm2, subtract xmm3/mem and put result in xmm1.
VEX.NDS.128.66.0F38.W0 BA /r VFMSUB231PS xmm1, xmm2, xmm3/m128	RVM	V/V	FMA	Multiply packed single-precision floating-point values from xmm2 and xmm3/mem, subtract xmm1 and put result in xmm1.
VEX.NDS.256.66.0F38.W0 9A /r VFMSUB132PS ymm1, ymm2, ymm3/m256	RVM	V/V	FMA	Multiply packed single-precision floating-point values from ymm1 and ymm3/mem, subtract ymm2 and put result in ymm1.
VEX.NDS.256.66.0F38.W0 AA /r VFMSUB213PS ymm1, ymm2, ymm3/m256	RVM	V/V	FMA	Multiply packed single-precision floating-point values from ymm1 and ymm2, subtract ymm3/mem and put result in ymm1.
VEX.NDS.256.66.0F38.0 BA /r VFMSUB231PS ymm1, ymm2, ymm3/m256	RVM	V/V	FMA	Multiply packed single-precision floating-point values from ymm2 and ymm3/mem, subtract ymm1 and put result in ymm1.
EVEX.NDS.128.66.0F38.W0 9A /r VFMSUB132PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	FV	V/V	AVX512VL AVX512F	Multiply packed single-precision floating-point values from xmm1 and xmm3/m128/m32bcst, subtract xmm2 and put result in xmm1.
EVEX.NDS.128.66.0F38.W0 AA /r VFMSUB213PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	FV	V/V	AVX512VL AVX512F	Multiply packed single-precision floating-point values from xmm1 and xmm2, subtract xmm3/m128/m32bcst and put result in xmm1.
EVEX.NDS.128.66.0F38.W0 BA /r VFMSUB231PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	FV	V/V	AVX512VL AVX512F	Multiply packed single-precision floating-point values from xmm2 and xmm3/m128/m32bcst, subtract xmm1 and put result in xmm1.
EVEX.NDS.256.66.0F38.W0 9A /r VFMSUB132PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	FV	V/V	AVX512VL AVX512F	Multiply packed single-precision floating-point values from ymm1 and ymm3/m256/m32bcst, subtract ymm2 and put result in ymm1.
EVEX.NDS.256.66.0F38.W0 AA /r VFMSUB213PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	FV	V/V	AVX512VL AVX512F	Multiply packed single-precision floating-point values from ymm1 and ymm2, subtract ymm3/m256/m32bcst and put result in ymm1.
EVEX.NDS.256.66.0F38.W0 BA /r VFMSUB231PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	FV	V/V	AVX512VL AVX512F	Multiply packed single-precision floating-point values from ymm2 and ymm3/m256/m32bcst, subtract ymm1 and put result in ymm1.
EVEX.NDS.512.66.0F38.W0 9A /r VFMSUB132PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}	FV	V/V	AVX512F	Multiply packed single-precision floating-point values from zmm1 and zmm3/m512/m32bcst, subtract zmm2 and put result in zmm1.
EVEX.NDS.512.66.0F38.W0 AA /r VFMSUB213PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}	FV	V/V	AVX512F	Multiply packed single-precision floating-point values from zmm1 and zmm2, subtract zmm3/m512/m32bcst and put result in zmm1.
EVEX.NDS.512.66.0F38.W0 BA /r VFMSUB231PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}	FV	V/V	AVX512F	Multiply packed single-precision floating-point values from zmm2 and zmm3/m512/m32bcst, subtract zmm1 and put result in zmm1.

## Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RVM	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FV	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

## Description

Performs a set of SIMD multiply-subtract computation on packed single-precision floating-point values using three source operands and writes the multiply-subtract results in the destination operand. The destination operand is also the first source operand. The second operand must be a SIMD register. The third source operand can be a SIMD register or a memory location.

**VFMSUB132PS:** Multiplies the four, eight or sixteen packed single-precision floating-point values from the first source operand to the four, eight or sixteen packed single-precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the four, eight or sixteen packed single-precision floating-point values in the second source operand, performs rounding and stores the resulting four, eight or sixteen packed single-precision floating-point values to the destination operand (first source operand).

**VFMSUB213PS:** Multiplies the four, eight or sixteen packed single-precision floating-point values from the second source operand to the four, eight or sixteen packed single-precision floating-point values in the first source operand. From the infinite precision intermediate result, subtracts the four, eight or sixteen packed single-precision floating-point values in the third source operand, performs rounding and stores the resulting four, eight or sixteen packed single-precision floating-point values to the destination operand (first source operand).

**VFMSUB231PS:** Multiplies the four, eight or sixteen packed single-precision floating-point values from the second source to the four, eight or sixteen packed single-precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the four, eight or sixteen packed single-precision floating-point values in the first source operand, performs rounding and stores the resulting four, eight or sixteen packed single-precision floating-point values to the destination operand (first source operand).

**EVEX encoded versions:** The destination operand (also first source operand) and the second source operand are ZMM/YMM/XMM register. The third 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 32-bit memory location. The destination operand is conditionally updated with write mask k1.

**VEX.256 encoded version:** The destination operand (also first source operand) is a YMM register and encoded in `reg_field`. The second source operand is a YMM register and encoded in `VEX.vvvv`. The third source operand is a YMM register or a 256-bit memory location and encoded in `rm_field`.

**VEX.128 encoded version:** The destination operand (also first source operand) is a XMM register and encoded in `reg_field`. The second source operand is a XMM register and encoded in `VEX.vvvv`. The third source operand is a XMM register or a 128-bit memory location and encoded in `rm_field`. The upper 128 bits of the YMM destination register are zeroed.

## Operation

In the operations below, “\*” and “-” symbols represent multiplication and subtraction with infinite precision inputs and outputs (no rounding).

**VFMSUB132PS DEST, SRC2, SRC3 (VEX encoded version)**

```

IF (VEX.128) THEN
    MAXNUM ← 2
ELSEIF (VEX.256)
    MAXNUM ← 4
FI
For i = 0 to MAXNUM-1 {
    n ← 32*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR(DEST[n+31:n]*SRC3[n+31:n] - SRC2[n+31:n])
}
IF (VEX.128) THEN
    DEST[MAX_VL-1:128] ← 0

```

```

ELSEIF (VEX.256)
    DEST[MAX_VL-1:256] ← 0
FI

```

**VFMSUB213PS DEST, SRC2, SRC3 (VEX encoded version)**

```

IF (VEX.128) THEN
    MAXNUM ← 2
ELSEIF (VEX.256)
    MAXNUM ← 4
FI
For i = 0 to MAXNUM-1 {
    n ← 32*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR(SRC2[n+31:n]*DEST[n+31:n] - SRC3[n+31:n])
}
IF (VEX.128) THEN
    DEST[MAX_VL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAX_VL-1:256] ← 0
FI

```

**VFMSUB231PS DEST, SRC2, SRC3 (VEX encoded version)**

```

IF (VEX.128) THEN
    MAXNUM ← 2
ELSEIF (VEX.256)
    MAXNUM ← 4
FI
For i = 0 to MAXNUM-1 {
    n ← 32*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR(SRC2[n+31:n]*SRC3[n+31:n] - DEST[n+31:n])
}
IF (VEX.128) THEN
    DEST[MAX_VL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAX_VL-1:256] ← 0
FI

```

**VFMSUB132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)**

```

(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ←
            RoundFPControl(DEST[i+31:i]*SRC3[j+31:i] - SRC2[j+31:i])
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+31:i] ← 0

```

```

    FI
  FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VFMSUB132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)**

(KL, VL) = (4, 128), (8, 256), (16, 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] ←
            RoundFPControl_MXCSR(DEST[i+31:i]*SRC3[31:0] - SRC2[i+31:i])
        ELSE
          DEST[i+31:i] ←
            RoundFPControl_MXCSR(DEST[i+31:i]*SRC3[i+31:i] - SRC2[i+31:i])
        FI;
      ELSE
        IF *merging-masking* ; merging-masking
          THEN *DEST[i+31:i] remains unchanged*
        ELSE ; zeroing-masking
          DEST[i+31:i] ← 0
        FI
      FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VFMSUB213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

IF (VL = 512) AND (EVEX.b = 1)

```

  THEN
    SET_RM(EVEX.RC);
  ELSE
    SET_RM(MXCSR.RM);
  FI;
FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ←
      RoundFPControl_MXCSR(SRC2[i+31:i]*DEST[i+31:i] - SRC3[i+31:i])
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+31:i] remains unchanged*
      ELSE ; zeroing-masking
        DEST[i+31:i] ← 0
      FI
    FI
  FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VFMSUB213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)**  
 (KL, VL) = (4, 128), (8, 256), (16, 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] ←
            RoundFPControl_MXCSR(SRC2[i+31:i]*DEST[i+31:i] - SRC3[31:0])
        ELSE
          DEST[i+31:i] ←
            RoundFPControl_MXCSR(SRC2[i+31:i]*DEST[i+31:i] - SRC3[i+31:i])
        FI;
      ELSE
        IF *merging-masking*           ; merging-masking
          THEN *DEST[i+31:i] remains unchanged*
        ELSE                             ; zeroing-masking
          DEST[i+31:i] ← 0
        FI
      FI;
    ENDFOR
  DEST[MAX_VL-1:VL] ← 0

```

**VFMSUB231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

```

IF (VL = 512) AND (EVEX.b = 1)
  THEN
    SET_RM(EVEX.RC);
  ELSE
    SET_RM(MXCSR.RM);
  FI;
FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ←
      RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[i+31:i] - DEST[i+31:i])
    ELSE
      IF *merging-masking*           ; merging-masking
        THEN *DEST[i+31:i] remains unchanged*
      ELSE                             ; zeroing-masking
        DEST[i+31:i] ← 0
      FI
    FI;
  ENDFOR
  DEST[MAX_VL-1:VL] ← 0

```

**VFMSUB231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*

```

```

THEN
  IF (EVEX.b = 1)
    THEN
      DEST[j+31:i] ←
      RoundFPControl_MXCSR(SRC2[j+31:i]*SRC3[31:0] - DEST[j+31:i])
    ELSE
      DEST[j+31:i] ←
      RoundFPControl_MXCSR(SRC2[j+31:i]*SRC3[j+31:i] - DEST[j+31:i])
  FI;
ELSE
  IF *merging-masking*           ; merging-masking
    THEN *DEST[j+31:i] remains unchanged*
  ELSE                             ; zeroing-masking
    DEST[j+31:i] ← 0
  FI
FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

### Intel C/C++ Compiler Intrinsic Equivalent

```

VFMSUBxxxPS __m512 __mm512_fmsub_ps(__m512 a, __m512 b, __m512 c);
VFMSUBxxxPS __m512 __mm512_fmsub_round_ps(__m512 a, __m512 b, __m512 c, int r);
VFMSUBxxxPS __m512 __mm512_mask_fmsub_ps(__m512 a, __mmask16 k, __m512 b, __m512 c);
VFMSUBxxxPS __m512 __mm512_maskz_fmsub_ps(__mmask16 k, __m512 a, __m512 b, __m512 c);
VFMSUBxxxPS __m512 __mm512_mask3_fmsub_ps(__m512 a, __m512 b, __m512 c, __mmask16 k);
VFMSUBxxxPS __m512 __mm512_mask_fmsub_round_ps(__m512 a, __mmask16 k, __m512 b, __m512 c, int r);
VFMSUBxxxPS __m512 __mm512_maskz_fmsub_round_ps(__mmask16 k, __m512 a, __m512 b, __m512 c, int r);
VFMSUBxxxPS __m512 __mm512_mask3_fmsub_round_ps(__m512 a, __m512 b, __m512 c, __mmask16 k, int r);
VFMSUBxxxPS __m256 __mm256_mask_fmsub_ps(__m256 a, __mmask8 k, __m256 b, __m256 c);
VFMSUBxxxPS __m256 __mm256_maskz_fmsub_ps(__mmask8 k, __m256 a, __m256 b, __m256 c);
VFMSUBxxxPS __m256 __mm256_mask3_fmsub_ps(__m256 a, __m256 b, __m256 c, __mmask8 k);
VFMSUBxxxPS __m128 __mm_mask_fmsub_ps(__m128 a, __mmask8 k, __m128 b, __m128 c);
VFMSUBxxxPS __m128 __mm_maskz_fmsub_ps(__mmask8 k, __m128 a, __m128 b, __m128 c);
VFMSUBxxxPS __m128 __mm_mask3_fmsub_ps(__m128 a, __m128 b, __m128 c, __mmask8 k);
VFMSUBxxxPS __m128 __mm_fmsub_ps (__m128 a, __m128 b, __m128 c);
VFMSUBxxxPS __m256 __mm256_fmsub_ps (__m256 a, __m256 b, __m256 c);

```

### SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

### Other Exceptions

VEX-encoded instructions, see Exceptions Type 2.

EVEX-encoded instructions, see Exceptions Type E2.

## VFMSUB132SD/VFMSUB213SD/VFMSUB231SD—Fused Multiply-Subtract of Scalar Double-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.DDS.LIG.66.0F38.W1 9B /r VFMSUB132SD xmm1, xmm2, xmm3/m64	RVM	V/V	FMA	Multiply scalar double-precision floating-point value from xmm1 and xmm3/m64, subtract xmm2 and put result in xmm1.
VEX.DDS.LIG.66.0F38.W1 AB /r VFMSUB213SD xmm1, xmm2, xmm3/m64	RVM	V/V	FMA	Multiply scalar double-precision floating-point value from xmm1 and xmm2, subtract xmm3/m64 and put result in xmm1.
VEX.DDS.LIG.66.0F38.W1 BB /r VFMSUB231SD xmm1, xmm2, xmm3/m64	RVM	V/V	FMA	Multiply scalar double-precision floating-point value from xmm2 and xmm3/m64, subtract xmm1 and put result in xmm1.
EVEX.DDS.LIG.66.0F38.W1 9B /r VFMSUB132SD xmm1 {k1}{z}, xmm2, xmm3/m64{er}	T1S	V/V	AVX512F	Multiply scalar double-precision floating-point value from xmm1 and xmm3/m64, subtract xmm2 and put result in xmm1.
EVEX.DDS.LIG.66.0F38.W1 AB /r VFMSUB213SD xmm1 {k1}{z}, xmm2, xmm3/m64{er}	T1S	V/V	AVX512F	Multiply scalar double-precision floating-point value from xmm1 and xmm2, subtract xmm3/m64 and put result in xmm1.
EVEX.DDS.LIG.66.0F38.W1 BB /r VFMSUB231SD xmm1 {k1}{z}, xmm2, xmm3/m64{er}	T1S	V/V	AVX512F	Multiply scalar double-precision floating-point value from xmm2 and xmm3/m64, subtract xmm1 and put result in xmm1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RVM	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
T1S	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Performs a SIMD multiply-subtract computation on the low packed double-precision floating-point values using three source operands and writes the multiply-subtract result in the destination operand. The destination operand is also the first source operand. The second operand must be a XMM register. The third source operand can be a XMM register or a 64-bit memory location.

**VFMSUB132SD:** Multiplies the low packed double-precision floating-point value from the first source operand to the low packed double-precision floating-point value in the third source operand. From the infinite precision intermediate result, subtracts the low packed double-precision floating-point values in the second source operand, performs rounding and stores the resulting packed double-precision floating-point value to the destination operand (first source operand).

**VFMSUB213SD:** Multiplies the low packed double-precision floating-point value from the second source operand to the low packed double-precision floating-point value in the first source operand. From the infinite precision intermediate result, subtracts the low packed double-precision floating-point value in the third source operand, performs rounding and stores the resulting packed double-precision floating-point value to the destination operand (first source operand).

**VFMSUB231SD:** Multiplies the low packed double-precision floating-point value from the second source to the low packed double-precision floating-point value in the third source operand. From the infinite precision intermediate result, subtracts the low packed double-precision floating-point value in the first source operand, performs



rounding and stores the resulting packed double-precision floating-point value to the destination operand (first source operand).

VEX.128 and EVEX encoded version: The destination operand (also first source operand) is encoded in `reg_field`. The second source operand is encoded in VEX.vvvv/EVEX.vvvv. The third source operand is encoded in `rm_field`. Bits 127:64 of the destination are unchanged. Bits MAXVL-1:128 of the destination register are zeroed.

EVEX encoded version: The low quadword element of the destination is updated according to the writemask.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NaNs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column.

### Operation

In the operations below, "\*" and "-" symbols represent multiplication and subtraction with infinite precision inputs and outputs (no rounding).

#### VFMSUB132SD DEST, SRC2, SRC3 (EVEX encoded version)

```
IF (EVEX.b = 1) and SRC3 *is a register*
  THEN
    SET_RM(EVEX.RC);
  ELSE
    SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
  THEN DEST[63:0] ← RoundFPControl(DEST[63:0]*SRC3[63:0] - SRC2[63:0])
  ELSE
    IF *merging-masking*           ; merging-masking
      THEN *DEST[63:0] remains unchanged*
    ELSE                             ; zeroing-masking
      THEN DEST[63:0] ← 0
    FI;
FI;
DEST[127:64] ← DEST[127:64]
DEST[MAX_VL-1:128] ← 0
```

#### VFMSUB213SD DEST, SRC2, SRC3 (EVEX encoded version)

```
IF (EVEX.b = 1) and SRC3 *is a register*
  THEN
    SET_RM(EVEX.RC);
  ELSE
    SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
  THEN DEST[63:0] ← RoundFPControl(SRC2[63:0]*DEST[63:0] - SRC3[63:0])
  ELSE
    IF *merging-masking*           ; merging-masking
      THEN *DEST[63:0] remains unchanged*
    ELSE                             ; zeroing-masking
      THEN DEST[63:0] ← 0
    FI;
FI;
DEST[127:64] ← DEST[127:64]
DEST[MAX_VL-1:128] ← 0
```

**VFMSUB231SD DEST, SRC2, SRC3 (EVEX encoded version)**

```

IF (EVEX.b = 1) and SRC3 *is a register*
  THEN
    SET_RM(EVEX.RC);
  ELSE
    SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
  THEN DEST[63:0] ← RoundFPControl(SRC2[63:0]*SRC3[63:0] - DEST[63:0])
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[63:0] remains unchanged*
    ELSE ; zeroing-masking
      THEN DEST[63:0] ← 0
    FI;
FI;
DEST[127:64] ← DEST[127:64]
DEST[MAX_VL-1:128] ← 0

```

**VFMSUB132SD DEST, SRC2, SRC3 (VEX encoded version)**

```

DEST[63:0] ← RoundFPControl_MXCSR(DEST[63:0]*SRC3[63:0] - SRC2[63:0])
DEST[127:64] ← DEST[127:64]
DEST[MAX_VL-1:128] ← 0

```

**VFMSUB213SD DEST, SRC2, SRC3 (VEX encoded version)**

```

DEST[63:0] ← RoundFPControl_MXCSR(SRC2[63:0]*DEST[63:0] - SRC3[63:0])
DEST[127:64] ← DEST[127:64]
DEST[MAX_VL-1:128] ← 0

```

**VFMSUB231SD DEST, SRC2, SRC3 (VEX encoded version)**

```

DEST[63:0] ← RoundFPControl_MXCSR(SRC2[63:0]*SRC3[63:0] - DEST[63:0])
DEST[127:64] ← DEST[127:64]
DEST[MAX_VL-1:128] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VFMSUBxxxSD __m128d __mm_fmsub_round_sd(__m128d a, __m128d b, __m128d c, int r);
VFMSUBxxxSD __m128d __mm_mask_fmsub_sd(__m128d a, __mmask8 k, __m128d b, __m128d c);
VFMSUBxxxSD __m128d __mm_maskz_fmsub_sd(__mmask8 k, __m128d a, __m128d b, __m128d c);
VFMSUBxxxSD __m128d __mm_mask3_fmsub_sd(__m128d a, __m128d b, __m128d c, __mmask8 k);
VFMSUBxxxSD __m128d __mm_mask_fmsub_round_sd(__m128d a, __mmask8 k, __m128d b, __m128d c, int r);
VFMSUBxxxSD __m128d __mm_maskz_fmsub_round_sd(__mmask8 k, __m128d a, __m128d b, __m128d c, int r);
VFMSUBxxxSD __m128d __mm_mask3_fmsub_round_sd(__m128d a, __m128d b, __m128d c, __mmask8 k, int r);
VFMSUBxxxSD __m128d __mm_fmsub_sd (__m128d a, __m128d b, __m128d c);

```

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal

**Other Exceptions**

VEX-encoded instructions, see Exceptions Type 3.

EVEX-encoded instructions, see Exceptions Type E3.

## VFMSUB132SS/VFMSUB213SS/VFMSUB231SS—Fused Multiply-Subtract of Scalar Single-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.DDS.LIG.66.0F38.W0 9B /r VFMSUB132SS xmm1, xmm2, xmm3/m32	RVM	V/V	FMA	Multiply scalar single-precision floating-point value from xmm1 and xmm3/m32, subtract xmm2 and put result in xmm1.
VEX.DDS.LIG.66.0F38.W0 AB /r VFMSUB213SS xmm1, xmm2, xmm3/m32	RVM	V/V	FMA	Multiply scalar single-precision floating-point value from xmm1 and xmm2, subtract xmm3/m32 and put result in xmm1.
VEX.DDS.LIG.66.0F38.W0 BB /r VFMSUB231SS xmm1, xmm2, xmm3/m32	RVM	V/V	FMA	Multiply scalar single-precision floating-point value from xmm2 and xmm3/m32, subtract xmm1 and put result in xmm1.
EVEX.DDS.LIG.66.0F38.W0 9B /r VFMSUB132SS xmm1 {k1}{z}, xmm2, xmm3/m32{er}	T1S	V/V	AVX512F	Multiply scalar single-precision floating-point value from xmm1 and xmm3/m32, subtract xmm2 and put result in xmm1.
EVEX.DDS.LIG.66.0F38.W0 AB /r VFMSUB213SS xmm1 {k1}{z}, xmm2, xmm3/m32{er}	T1S	V/V	AVX512F	Multiply scalar single-precision floating-point value from xmm1 and xmm2, subtract xmm3/m32 and put result in xmm1.
EVEX.DDS.LIG.66.0F38.W0 BB /r VFMSUB231SS xmm1 {k1}{z}, xmm2, xmm3/m32{er}	T1S	V/V	AVX512F	Multiply scalar single-precision floating-point value from xmm2 and xmm3/m32, subtract xmm1 and put result in xmm1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RVM	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
T1S	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Performs a SIMD multiply-subtract computation on the low packed single-precision floating-point values using three source operands and writes the multiply-subtract result in the destination operand. The destination operand is also the first source operand. The second operand must be a XMM register. The third source operand can be a XMM register or a 32-bit memory location.

**VFMSUB132SS:** Multiplies the low packed single-precision floating-point value from the first source operand to the low packed single-precision floating-point value in the third source operand. From the infinite precision intermediate result, subtracts the low packed single-precision floating-point values in the second source operand, performs rounding and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

**VFMSUB213SS:** Multiplies the low packed single-precision floating-point value from the second source operand to the low packed single-precision floating-point value in the first source operand. From the infinite precision intermediate result, subtracts the low packed single-precision floating-point value in the third source operand, performs rounding and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

**VFMSUB231SS:** Multiplies the low packed single-precision floating-point value from the second source to the low packed single-precision floating-point value in the third source operand. From the infinite precision intermediate result, subtracts the low packed single-precision floating-point value in the first source operand, performs rounding

and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

VEX.128 and EVEX encoded version: The destination operand (also first source operand) is encoded in `reg_field`. The second source operand is encoded in VEX.vvvv/EVEX.vvvv. The third source operand is encoded in `rm_field`. Bits 127:32 of the destination are unchanged. Bits MAXVL-1:128 of the destination register are zeroed.

EVEX encoded version: The low doubleword element of the destination is updated according to the writemask.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NaNs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column.

### Operation

In the operations below, "\*" and "-" symbols represent multiplication and subtraction with infinite precision inputs and outputs (no rounding).

#### VFMSUB132SS DEST, SRC2, SRC3 (EVEX encoded version)

IF (EVEX.b = 1) and SRC3 \*is a register\*

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

IF k1[0] or \*no writemask\*

THEN DEST[31:0] ← RoundFPControl(DEST[31:0]\*SRC3[31:0] - SRC2[31:0])

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[31:0] remains unchanged\*

ELSE ; zeroing-masking

THEN DEST[31:0] ← 0

FI;

FI;

DEST[127:32] ← DEST[127:32]

DEST[MAX\_VL-1:128] ← 0

#### VFMSUB213SS DEST, SRC2, SRC3 (EVEX encoded version)

IF (EVEX.b = 1) and SRC3 \*is a register\*

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

IF k1[0] or \*no writemask\*

THEN DEST[31:0] ← RoundFPControl(SRC2[31:0]\*DEST[31:0] - SRC3[31:0])

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[31:0] remains unchanged\*

ELSE ; zeroing-masking

THEN DEST[31:0] ← 0

FI;

FI;

DEST[127:32] ← DEST[127:32]

DEST[MAX\_VL-1:128] ← 0

**VFMSUB231SS DEST, SRC2, SRC3 (EVEX encoded version)**

```

IF (EVEX.b = 1) and SRC3 *is a register*
  THEN
    SET_RM(EVEX.RC);
  ELSE
    SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
  THEN DEST[31:0] ← RoundFPControl(SRC2[31:0]*SRC3[63:0] - DEST[31:0])
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[31:0] remains unchanged*
    ELSE ; zeroing-masking
      THEN DEST[31:0] ← 0
    FI;
FI;
DEST[127:32] ← DEST[127:32]
DEST[MAX_VL-1:128] ← 0

```

**VFMSUB132SS DEST, SRC2, SRC3 (VEX encoded version)**

```

DEST[31:0] ← RoundFPControl_MXCSR(DEST[31:0]*SRC3[31:0] - SRC2[31:0])
DEST[127:32] ← DEST[127:32]
DEST[MAX_VL-1:128] ← 0

```

**VFMSUB213SS DEST, SRC2, SRC3 (VEX encoded version)**

```

DEST[31:0] ← RoundFPControl_MXCSR(SRC2[31:0]*DEST[31:0] - SRC3[31:0])
DEST[127:32] ← DEST[127:32]
DEST[MAX_VL-1:128] ← 0

```

**VFMSUB231SS DEST, SRC2, SRC3 (VEX encoded version)**

```

DEST[31:0] ← RoundFPControl_MXCSR(SRC2[31:0]*SRC3[31:0] - DEST[31:0])
DEST[127:32] ← DEST[127:32]
DEST[MAX_VL-1:128] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VFMSUBxxxSS __m128 _mm_fmsub_round_ss(__m128 a, __m128 b, __m128 c, int r);
VFMSUBxxxSS __m128 _mm_mask_fmsub_ss(__m128 a, __mmask8 k, __m128 b, __m128 c);
VFMSUBxxxSS __m128 _mm_maskz_fmsub_ss(__mmask8 k, __m128 a, __m128 b, __m128 c);
VFMSUBxxxSS __m128 _mm_mask3_fmsub_ss(__m128 a, __m128 b, __m128 c, __mmask8 k);
VFMSUBxxxSS __m128 _mm_mask_fmsub_round_ss(__m128 a, __mmask8 k, __m128 b, __m128 c, int r);
VFMSUBxxxSS __m128 _mm_maskz_fmsub_round_ss(__mmask8 k, __m128 a, __m128 b, __m128 c, int r);
VFMSUBxxxSS __m128 _mm_mask3_fmsub_round_ss(__m128 a, __m128 b, __m128 c, __mmask8 k, int r);
VFMSUBxxxSS __m128 _mm_fmsub_ss (__m128 a, __m128 b, __m128 c);

```

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal

**Other Exceptions**

VEX-encoded instructions, see Exceptions Type 3.

EVEX-encoded instructions, see Exceptions Type E3.

## VFNMADD132PD/VFNMADD213PD/VFNMADD231PD—Fused Negative Multiply-Add of Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.NDS.128.66.0F38.W1 9C /r VFNMADD132PD xmm1, xmm2, xmm3/m128	RVM	V/V	FMA	Multiply packed double-precision floating-point values from xmm1 and xmm3/mem, negate the multiplication result and add to xmm2 and put result in xmm1.
VEX.NDS.128.66.0F38.W1 AC /r VFNMADD213PD xmm1, xmm2, xmm3/m128	RVM	V/V	FMA	Multiply packed double-precision floating-point values from xmm1 and xmm2, negate the multiplication result and add to xmm3/mem and put result in xmm1.
VEX.NDS.128.66.0F38.W1 BC /r VFNMADD231PD xmm1, xmm2, xmm3/m128	RVM	V/V	FMA	Multiply packed double-precision floating-point values from xmm2 and xmm3/mem, negate the multiplication result and add to xmm1 and put result in xmm1.
VEX.NDS.256.66.0F38.W1 9C /r VFNMADD132PD ymm1, ymm2, ymm3/m256	RVM	V/V	FMA	Multiply packed double-precision floating-point values from ymm1 and ymm3/mem, negate the multiplication result and add to ymm2 and put result in ymm1.
VEX.NDS.256.66.0F38.W1 AC /r VFNMADD213PD ymm1, ymm2, ymm3/m256	RVM	V/V	FMA	Multiply packed double-precision floating-point values from ymm1 and ymm2, negate the multiplication result and add to ymm3/mem and put result in ymm1.
VEX.NDS.256.66.0F38.W1 BC /r VFNMADD231PD ymm1, ymm2, ymm3/m256	RVM	V/V	FMA	Multiply packed double-precision floating-point values from ymm2 and ymm3/mem, negate the multiplication result and add to ymm1 and put result in ymm1.
EVEX.NDS.128.66.0F38.W1 9C /r VFNMADD132PD xmm0 {k1}{z}, xmm1, xmm2/m128/m64bcst	FV	V/V	AVX512VL AVX512F	Multiply packed double-precision floating-point values from xmm1 and xmm3/m128/m64bcst, negate the multiplication result and add to xmm2 and put result in xmm1.
EVEX.NDS.128.66.0F38.W1 AC /r VFNMADD213PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	FV	V/V	AVX512VL AVX512F	Multiply packed double-precision floating-point values from xmm1 and xmm2, negate the multiplication result and add to xmm3/m128/m64bcst and put result in xmm1.
EVEX.NDS.128.66.0F38.W1 BC /r VFNMADD231PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	FV	V/V	AVX512VL AVX512F	Multiply packed double-precision floating-point values from xmm2 and xmm3/m128/m64bcst, negate the multiplication result and add to xmm1 and put result in xmm1.
EVEX.NDS.256.66.0F38.W1 9C /r VFNMADD132PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	FV	V/V	AVX512VL AVX512F	Multiply packed double-precision floating-point values from ymm1 and ymm3/m256/m64bcst, negate the multiplication result and add to ymm2 and put result in ymm1.
EVEX.NDS.256.66.0F38.W1 AC /r VFNMADD213PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	FV	V/V	AVX512VL AVX512F	Multiply packed double-precision floating-point values from ymm1 and ymm2, negate the multiplication result and add to ymm3/m256/m64bcst and put result in ymm1.
EVEX.NDS.256.66.0F38.W1 BC /r VFNMADD231PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	FV	V/V	AVX512VL AVX512F	Multiply packed double-precision floating-point values from ymm2 and ymm3/m256/m64bcst, negate the multiplication result and add to ymm1 and put result in ymm1.
EVEX.NDS.512.66.0F38.W1 9C /r VFNMADD132PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	FV	V/V	AVX512F	Multiply packed double-precision floating-point values from zmm1 and zmm3/m512/m64bcst, negate the multiplication result and add to zmm2 and put result in zmm1.

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.512.66.0F38.W1 AC /r VFNMADD213PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	FV	V/V	AVX512F	Multiply packed double-precision floating-point values from zmm1 and zmm2, negate the multiplication result and add to zmm3/m512/m64bcst and put result in zmm1.
EVEX.NDS.512.66.0F38.W1 BC /r VFNMADD231PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	FV	V/V	AVX512F	Multiply packed double-precision floating-point values from zmm2 and zmm3/m512/m64bcst, negate the multiplication result and add to zmm1 and put result in zmm1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RVM	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FV	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

VFNMADD132PD: Multiplies the two, four or eight packed double-precision floating-point values from the first source operand to the two, four or eight packed double-precision floating-point values in the third source operand, adds the negated infinite precision intermediate result to the two, four or eight packed double-precision floating-point values in the second source operand, performs rounding and stores the resulting two, four or eight packed double-precision floating-point values to the destination operand (first source operand).

VFNMADD213PD: Multiplies the two, four or eight packed double-precision floating-point values from the second source operand to the two, four or eight packed double-precision floating-point values in the first source operand, adds the negated infinite precision intermediate result to the two, four or eight packed double-precision floating-point values in the third source operand, performs rounding and stores the resulting two, four or eight packed double-precision floating-point values to the destination operand (first source operand).

VFNMADD231PD: Multiplies the two, four or eight packed double-precision floating-point values from the second source to the two, four or eight packed double-precision floating-point values in the third source operand, the negated infinite precision intermediate result to the two, four or eight packed double-precision floating-point values in the first source operand, performs rounding and stores the resulting two, four or eight packed double-precision floating-point values to the destination operand (first source operand).

EVEX encoded versions: The destination operand (also first source operand) and the second source operand are ZMM/YMM/XMM register. The third 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 64-bit memory location. The destination operand is conditionally updated with write mask k1.

VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in `reg_field`. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in `rm_field`.

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in `reg_field`. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 128-bit memory location and encoded in `rm_field`. The upper 128 bits of the YMM destination register are zeroed.

### Operation

In the operations below, "\*" and "-" symbols represent multiplication and subtraction with infinite precision inputs and outputs (no rounding).

**VFMADD132PD DEST, SRC2, SRC3 (VEX encoded version)**

```

IF (VEX.128) THEN
    MAXNUM ← 2
ELSEIF (VEX.256)
    MAXNUM ← 4
FI

For i = 0 to MAXNUM-1 {
    n ← 64*i;
    DEST[n+63:n] ← RoundFPControl_MXCSR(-(DEST[n+63:n]*SRC3[n+63:n]) + SRC2[n+63:n])
}

IF (VEX.128) THEN
    DEST[MAX_VL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAX_VL-1:256] ← 0
FI

```

**VFMADD213PD DEST, SRC2, SRC3 (VEX encoded version)**

```

IF (VEX.128) THEN
    MAXNUM ← 2
ELSEIF (VEX.256)
    MAXNUM ← 4
FI

For i = 0 to MAXNUM-1 {
    n ← 64*i;
    DEST[n+63:n] ← RoundFPControl_MXCSR(-(SRC2[n+63:n]*DEST[n+63:n]) + SRC3[n+63:n])
}

IF (VEX.128) THEN
    DEST[MAX_VL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAX_VL-1:256] ← 0
FI

```

**VFMADD231PD DEST, SRC2, SRC3 (VEX encoded version)**

```

IF (VEX.128) THEN
    MAXNUM ← 2
ELSEIF (VEX.256)
    MAXNUM ← 4
FI

For i = 0 to MAXNUM-1 {
    n ← 64*i;
    DEST[n+63:n] ← RoundFPControl_MXCSR(-(SRC2[n+63:n]*SRC3[n+63:n]) + DEST[n+63:n])
}

IF (VEX.128) THEN
    DEST[MAX_VL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAX_VL-1:256] ← 0
FI

```



**VFMADD132PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

IF (VL = 512) AND (EVEX.b = 1)

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN DEST[i+63:i] ←

RoundFPControl(-(DEST[i+63:i]\*SRC3[i+63:i]) + SRC2[i+63:i])

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VFMADD132PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 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] ←

RoundFPControl\_MXCSR(-(DEST[i+63:i]\*SRC3[63:0]) + SRC2[i+63:i])

ELSE

DEST[i+63:i] ←

RoundFPControl\_MXCSR(-(DEST[i+63:i]\*SRC3[i+63:i]) + SRC2[i+63:i])

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VFMADD213PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

IF (VL = 512) AND (EVEX.b = 1)

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN DEST[i+63:i] ←

RoundFPControl(-(SRC2[i+63:i]\*DEST[i+63:i]) + SRC3[i+63:i])

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VFMADD213PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 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] ←

RoundFPControl\_MXCSR(-(SRC2[i+63:i]\*DEST[i+63:i]) + SRC3[63:0])

ELSE

DEST[i+63:i] ←

RoundFPControl\_MXCSR(-(SRC2[i+63:i]\*DEST[i+63:i]) + SRC3[i+63:i])

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VFMADD231PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

IF (VL = 512) AND (EVEX.b = 1)

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN DEST[i+63:i] ←

RoundFPControl(-(SRC2[i+63:i]\*SRC3[i+63:i]) + DEST[i+63:i])

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VFMADD231PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 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] ←

RoundFPControl\_MXCSR(-(SRC2[i+63:i]\*SRC3[63:0]) + DEST[i+63:i])

ELSE

DEST[i+63:i] ←

RoundFPControl\_MXCSR(-(SRC2[i+63:i]\*SRC3[i+63:i]) + DEST[i+63:i])

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

VFMADDxxxPD \_\_m512d \_\_mm512\_fnmadd\_pd(\_\_m512d a, \_\_m512d b, \_\_m512d c);

VFMADDxxxPD \_\_m512d \_\_mm512\_fnmadd\_round\_pd(\_\_m512d a, \_\_m512d b, \_\_m512d c, int r);

VFMADDxxxPD \_\_m512d \_\_mm512\_mask\_fnmadd\_pd(\_\_m512d a, \_\_mmask8 k, \_\_m512d b, \_\_m512d c);

VFMADDxxxPD \_\_m512d \_\_mm512\_maskz\_fnmadd\_pd(\_\_mmask8 k, \_\_m512d a, \_\_m512d b, \_\_m512d c);

VFMADDxxxPD \_\_m512d \_\_mm512\_mask3\_fnmadd\_pd(\_\_m512d a, \_\_m512d b, \_\_m512d c, \_\_mmask8 k);

VFMADDxxxPD \_\_m512d \_\_mm512\_mask\_fnmadd\_round\_pd(\_\_m512d a, \_\_mmask8 k, \_\_m512d b, \_\_m512d c, int r);  
 VFMADDxxxPD \_\_m512d \_\_mm512\_maskz\_fnmadd\_round\_pd(\_\_mmask8 k, \_\_m512d a, \_\_m512d b, \_\_m512d c, int r);  
 VFMADDxxxPD \_\_m512d \_\_mm512\_mask3\_fnmadd\_round\_pd(\_\_m512d a, \_\_m512d b, \_\_m512d c, \_\_mmask8 k, int r);  
 VFMADDxxxPD \_\_m256d \_\_mm256\_mask\_fnmadd\_pd(\_\_m256d a, \_\_mmask8 k, \_\_m256d b, \_\_m256d c);  
 VFMADDxxxPD \_\_m256d \_\_mm256\_maskz\_fnmadd\_pd(\_\_mmask8 k, \_\_m256d a, \_\_m256d b, \_\_m256d c);  
 VFMADDxxxPD \_\_m256d \_\_mm256\_mask3\_fnmadd\_pd(\_\_m256d a, \_\_m256d b, \_\_m256d c, \_\_mmask8 k);  
 VFMADDxxxPD \_\_m128d \_\_mm\_mask\_fnmadd\_pd(\_\_m128d a, \_\_mmask8 k, \_\_m128d b, \_\_m128d c);  
 VFMADDxxxPD \_\_m128d \_\_mm\_maskz\_fnmadd\_pd(\_\_mmask8 k, \_\_m128d a, \_\_m128d b, \_\_m128d c);  
 VFMADDxxxPD \_\_m128d \_\_mm\_mask3\_fnmadd\_pd(\_\_m128d a, \_\_m128d b, \_\_m128d c, \_\_mmask8 k);  
 VFMADDxxxPD \_\_m128d \_\_mm\_fnmadd\_pd (\_\_m128d a, \_\_m128d b, \_\_m128d c);  
 VFMADDxxxPD \_\_m256d \_\_mm256\_fnmadd\_pd (\_\_m256d a, \_\_m256d b, \_\_m256d c);

### SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

### Other Exceptions

VEX-encoded instructions, see Exceptions Type 2.

EVEX-encoded instructions, see Exceptions Type E2.

## VFMADD132PS/VFMADD213PS/VFMADD231PS—Fused Negative Multiply-Add of Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.NDS.128.66.0F38.W0 9C /r VFMADD132PS xmm1, xmm2, xmm3/m128	RVM	V/V	FMA	Multiply packed single-precision floating-point values from xmm1 and xmm3/mem, negate the multiplication result and add to xmm2 and put result in xmm1.
VEX.NDS.128.66.0F38.W0 AC /r VFMADD213PS xmm1, xmm2, xmm3/m128	RVM	V/V	FMA	Multiply packed single-precision floating-point values from xmm1 and xmm2, negate the multiplication result and add to xmm3/mem and put result in xmm1.
VEX.NDS.128.66.0F38.W0 BC /r VFMADD231PS xmm1, xmm2, xmm3/m128	RVM	V/V	FMA	Multiply packed single-precision floating-point values from xmm2 and xmm3/mem, negate the multiplication result and add to xmm1 and put result in xmm1.
VEX.NDS.256.66.0F38.W0 9C /r VFMADD132PS ymm1, ymm2, ymm3/m256	RVM	V/V	FMA	Multiply packed single-precision floating-point values from ymm1 and ymm3/mem, negate the multiplication result and add to ymm2 and put result in ymm1.
VEX.NDS.256.66.0F38.W0 AC /r VFMADD213PS ymm1, ymm2, ymm3/m256	RVM	V/V	FMA	Multiply packed single-precision floating-point values from ymm1 and ymm2, negate the multiplication result and add to ymm3/mem and put result in ymm1.
VEX.NDS.256.66.0F38.0 BC /r VFMADD231PS ymm1, ymm2, ymm3/m256	RVM	V/V	FMA	Multiply packed single-precision floating-point values from ymm2 and ymm3/mem, negate the multiplication result and add to ymm1 and put result in ymm1.
EVEX.NDS.128.66.0F38.W0 9C /r VFMADD132PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	FV	V/V	AVX512VL AVX512F	Multiply packed single-precision floating-point values from xmm1 and xmm3/m128/m32bcst, negate the multiplication result and add to xmm2 and put result in xmm1.
EVEX.NDS.128.66.0F38.W0 AC /r VFMADD213PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	FV	V/V	AVX512VL AVX512F	Multiply packed single-precision floating-point values from xmm1 and xmm2, negate the multiplication result and add to xmm3/m128/m32bcst and put result in xmm1.
EVEX.NDS.128.66.0F38.W0 BC /r VFMADD231PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	FV	V/V	AVX512VL AVX512F	Multiply packed single-precision floating-point values from xmm2 and xmm3/m128/m32bcst, negate the multiplication result and add to xmm1 and put result in xmm1.
EVEX.NDS.256.66.0F38.W0 9C /r VFMADD132PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	FV	V/V	AVX512VL AVX512F	Multiply packed single-precision floating-point values from ymm1 and ymm3/m256/m32bcst, negate the multiplication result and add to ymm2 and put result in ymm1.
EVEX.NDS.256.66.0F38.W0 AC /r VFMADD213PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	FV	V/V	AVX512VL AVX512F	Multiply packed single-precision floating-point values from ymm1 and ymm2, negate the multiplication result and add to ymm3/m256/m32bcst and put result in ymm1.
EVEX.NDS.256.66.0F38.W0 BC /r VFMADD231PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	FV	V/V	AVX512VL AVX512F	Multiply packed single-precision floating-point values from ymm2 and ymm3/m256/m32bcst, negate the multiplication result and add to ymm1 and put result in ymm1.
EVEX.NDS.512.66.0F38.W0 9C /r VFMADD132PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}	FV	V/V	AVX512VL AVX512F	Multiply packed single-precision floating-point values from zmm1 and zmm3/m512/m32bcst, negate the multiplication result and add to zmm2 and put result in zmm1.
EVEX.NDS.512.66.0F38.W0 AC /r VFMADD213PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}	FV	V/V	AVX512F	Multiply packed single-precision floating-point values from zmm1 and zmm2, negate the multiplication result and add to zmm3/m512/m32bcst and put result in zmm1.
EVEX.NDS.512.66.0F38.W0 BC /r VFMADD231PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}	FV	V/V	AVX512F	Multiply packed single-precision floating-point values from zmm2 and zmm3/m512/m32bcst, negate the multiplication result and add to zmm1 and put result in zmm1.

## Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RVM	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FV	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

## Description

**VFMADD132PS:** Multiplies the four, eight or sixteen packed single-precision floating-point values from the first source operand to the four, eight or sixteen packed single-precision floating-point values in the third source operand, adds the negated infinite precision intermediate result to the four, eight or sixteen packed single-precision floating-point values in the second source operand, performs rounding and stores the resulting four, eight or sixteen packed single-precision floating-point values to the destination operand (first source operand).

**VFMADD213PS:** Multiplies the four, eight or sixteen packed single-precision floating-point values from the second source operand to the four, eight or sixteen packed single-precision floating-point values in the first source operand, adds the negated infinite precision intermediate result to the four, eight or sixteen packed single-precision floating-point values in the third source operand, performs rounding and stores the resulting the four, eight or sixteen packed single-precision floating-point values to the destination operand (first source operand).

**VFMADD231PS:** Multiplies the four, eight or sixteen packed single-precision floating-point values from the second source operand to the four, eight or sixteen packed single-precision floating-point values in the third source operand, adds the negated infinite precision intermediate result to the four, eight or sixteen packed single-precision floating-point values in the first source operand, performs rounding and stores the resulting four, eight or sixteen packed single-precision floating-point values to the destination operand (first source operand).

**EVEX encoded versions:** The destination operand (also first source operand) and the second source operand are ZMM/YMM/XMM register. The third 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 32-bit memory location. The destination operand is conditionally updated with write mask k1.

**VEX.256 encoded version:** The destination operand (also first source operand) is a YMM register and encoded in `reg_field`. The second source operand is a YMM register and encoded in `VEX.vvvv`. The third source operand is a YMM register or a 256-bit memory location and encoded in `rm_field`.

**VEX.128 encoded version:** The destination operand (also first source operand) is a XMM register and encoded in `reg_field`. The second source operand is a XMM register and encoded in `VEX.vvvv`. The third source operand is a XMM register or a 128-bit memory location and encoded in `rm_field`. The upper 128 bits of the YMM destination register are zeroed.

## Operation

In the operations below, "\*" and "+" symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding).

**VFMADD132PS DEST, SRC2, SRC3 (VEX encoded version)**

IF (VEX.128) THEN

    MAXNUM ← 2

ELSEIF (VEX.256)

    MAXNUM ← 4

FI

For i = 0 to MAXNUM-1 {

    n ← 32\*i;

    DEST[n+31:n] ← RoundFPControl\_MXCSR(- (DEST[n+31:n]\*SRC3[n+31:n]) + SRC2[n+31:n])

}

IF (VEX.128) THEN

    DEST[MAX\_VL-1:128] ← 0

ELSEIF (VEX.256)

    DEST[MAX\_VL-1:256] ← 0

FI

**VFMADD213PS DEST, SRC2, SRC3 (VEX encoded version)**

```

IF (VEX.128) THEN
    MAXNUM ← 2
ELSEIF (VEX.256)
    MAXNUM ← 4
FI
For i = 0 to MAXNUM-1 {
    n ← 32*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR(- (SRC2[n+31:n]*DEST[n+31:n]) + SRC3[n+31:n])
}
IF (VEX.128) THEN
    DEST[MAX_VL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAX_VL-1:256] ← 0
FI

```

**VFMADD231PS DEST, SRC2, SRC3 (VEX encoded version)**

```

IF (VEX.128) THEN
    MAXNUM ← 2
ELSEIF (VEX.256)
    MAXNUM ← 4
FI
For i = 0 to MAXNUM-1 {
    n ← 32*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR(- (SRC2[n+31:n]*SRC3[n+31:n]) + DEST[n+31:n])
}
IF (VEX.128) THEN
    DEST[MAX_VL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAX_VL-1:256] ← 0
FI

```

**VFMADD132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)**

```

(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ←
            RoundFPControl(-(DEST[i+31:i]*SRC3[j+31:i]) + SRC2[j+31:i])
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VFMADD132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)**  
 (KL, VL) = (4, 128), (8, 256), (16, 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] ←
            RoundFPControl_MXCSR(-(DEST[i+31:i]*SRC3[31:0]) + SRC2[i+31:i])
        ELSE
          DEST[i+31:i] ←
            RoundFPControl_MXCSR(-(DEST[i+31:i]*SRC3[j+31:i]) + SRC2[i+31:i])
        FI;

    ELSE
      IF *merging-masking*           ; merging-masking
        THEN *DEST[i+31:i] remains unchanged*
      ELSE                             ; zeroing-masking
        DEST[i+31:i] ← 0
      FI
    FI;
  FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VFMADD213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)**  
 (KL, VL) = (4, 128), (8, 256), (16, 512)

```

IF (VL = 512) AND (EVEX.b = 1)
  THEN
    SET_RM(EVEX.RC);
  ELSE
    SET_RM(MXCSR.RM);
  FI;
FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ←
      RoundFPControl(-(SRC2[j+31:i]*DEST[i+31:i]) + SRC3[i+31:i])
    ELSE
      IF *merging-masking*           ; merging-masking
        THEN *DEST[i+31:i] remains unchanged*
      ELSE                             ; zeroing-masking
        DEST[i+31:i] ← 0
      FI
    FI;
  FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```



**VFMADD213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)**  
 (KL, VL) = (4, 128), (8, 256), (16, 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] ←
          RoundFPControl_MXCSR(-(SRC2[i+31:i]*DEST[i+31:i]) + SRC3[31:0])

          ELSE
            DEST[i+31:i] ←
            RoundFPControl_MXCSR(-(SRC2[i+31:i]*DEST[i+31:i]) + SRC3[j+31:i])
          FI;
        ELSE
          IF *merging-masking*           ; merging-masking
            THEN *DEST[i+31:i] remains unchanged*
          ELSE                             ; zeroing-masking
            DEST[i+31:i] ← 0
          FI
        FI;
      ENDFOR
    DEST[MAX_VL-1:VL] ← 0
  
```

**VFMADD231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)**  
 (KL, VL) = (4, 128), (8, 256), (16, 512)

```

IF (VL = 512) AND (EVEX.b = 1)
  THEN
    SET_RM(EVEX.RC);
  ELSE
    SET_RM(MXCSR.RM);
  FI;
FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ←
    RoundFPControl(-(SRC2[i+31:i]*SRC3[j+31:i]) + DEST[i+31:i])
  ELSE
    IF *merging-masking*           ; merging-masking
      THEN *DEST[i+31:i] remains unchanged*
    ELSE                             ; zeroing-masking
      DEST[i+31:i] ← 0
    FI
  FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0
  
```

**VFMADD231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)**  
 (KL, VL) = (4, 128), (8, 256), (16, 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] ←
            RoundFPControl_MXCSR(-(SRC2[i+31:i]*SRC3[31:0]) + DEST[i+31:i])
        ELSE
          DEST[i+31:i] ←
            RoundFPControl_MXCSR(-(SRC2[i+31:i]*SRC3[j+31:i]) + DEST[i+31:i])
        FI;
      ELSE
        IF *merging-masking*           ; merging-masking
          THEN *DEST[i+31:i] remains unchanged*
        ELSE                             ; zeroing-masking
          DEST[i+31:i] ← 0
        FI
      FI;
    ENDFOR
  DEST[MAX_VL-1:VL] ← 0

```

#### Intel C/C++ Compiler Intrinsic Equivalent

```

VFMADDxxxPS __m512 __mm512_fmadd_ps(__m512 a, __m512 b, __m512 c);
VFMADDxxxPS __m512 __mm512_fmadd_round_ps(__m512 a, __m512 b, __m512 c, int r);
VFMADDxxxPS __m512 __mm512_mask_fmadd_ps(__m512 a, __mmask16 k, __m512 b, __m512 c);
VFMADDxxxPS __m512 __mm512_maskz_fmadd_ps(__mmask16 k, __m512 a, __m512 b, __m512 c);
VFMADDxxxPS __m512 __mm512_mask3_fmadd_ps(__m512 a, __m512 b, __m512 c, __mmask16 k);
VFMADDxxxPS __m512 __mm512_mask_fmadd_round_ps(__m512 a, __mmask16 k, __m512 b, __m512 c, int r);
VFMADDxxxPS __m512 __mm512_maskz_fmadd_round_ps(__mmask16 k, __m512 a, __m512 b, __m512 c, int r);
VFMADDxxxPS __m512 __mm512_mask3_fmadd_round_ps(__m512 a, __m512 b, __m512 c, __mmask16 k, int r);
VFMADDxxxPS __m256 __mm256_mask_fmadd_ps(__m256 a, __mmask8 k, __m256 b, __m256 c);
VFMADDxxxPS __m256 __mm256_maskz_fmadd_ps(__mmask8 k, __m256 a, __m256 b, __m256 c);
VFMADDxxxPS __m256 __mm256_mask3_fmadd_ps(__m256 a, __m256 b, __m256 c, __mmask8 k);
VFMADDxxxPS __m128 __mm_mask_fmadd_ps(__m128 a, __mmask8 k, __m128 b, __m128 c);
VFMADDxxxPS __m128 __mm_maskz_fmadd_ps(__mmask8 k, __m128 a, __m128 b, __m128 c);
VFMADDxxxPS __m128 __mm_mask3_fmadd_ps(__m128 a, __m128 b, __m128 c, __mmask8 k);
VFMADDxxxPS __m128 __mm_fmadd_ps (__m128 a, __m128 b, __m128 c);
VFMADDxxxPS __m256 __mm256_fmadd_ps (__m256 a, __m256 b, __m256 c);

```

#### SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

#### Other Exceptions

VEX-encoded instructions, see Exceptions Type 2.

EVEX-encoded instructions, see Exceptions Type E2.

## VFMADD132SD/VFMADD213SD/VFMADD231SD—Fused Negative Multiply-Add of Scalar Double-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.DDS.LIG.66.0F38.W1 9D /r VFMADD132SD xmm1, xmm2, xmm3/m64	RVM	V/V	FMA	Multiply scalar double-precision floating-point value from xmm1 and xmm3/mem, negate the multiplication result and add to xmm2 and put result in xmm1.
VEX.DDS.LIG.66.0F38.W1 AD /r VFMADD213SD xmm1, xmm2, xmm3/m64	RVM	V/V	FMA	Multiply scalar double-precision floating-point value from xmm1 and xmm2, negate the multiplication result and add to xmm3/mem and put result in xmm1.
VEX.DDS.LIG.66.0F38.W1 BD /r VFMADD231SD xmm1, xmm2, xmm3/m64	RVM	V/V	FMA	Multiply scalar double-precision floating-point value from xmm2 and xmm3/mem, negate the multiplication result and add to xmm1 and put result in xmm1.
EVEX.DDS.LIG.66.0F38.W1 9D /r VFMADD132SD xmm1 {k1}{z}, xmm2, xmm3/m64{er}	T1S	V/V	AVX512F	Multiply scalar double-precision floating-point value from xmm1 and xmm3/m64, negate the multiplication result and add to xmm2 and put result in xmm1.
EVEX.DDS.LIG.66.0F38.W1 AD /r VFMADD213SD xmm1 {k1}{z}, xmm2, xmm3/m64{er}	T1S	V/V	AVX512F	Multiply scalar double-precision floating-point value from xmm1 and xmm2, negate the multiplication result and add to xmm3/m64 and put result in xmm1.
EVEX.DDS.LIG.66.0F38.W1 BD /r VFMADD231SD xmm1 {k1}{z}, xmm2, xmm3/m64{er}	T1S	V/V	AVX512F	Multiply scalar double-precision floating-point value from xmm2 and xmm3/m64, negate the multiplication result and add to xmm1 and put result in xmm1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RVM	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
T1S	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

**VFMADD132SD:** Multiplies the low packed double-precision floating-point value from the first source operand to the low packed double-precision floating-point value in the third source operand, adds the negated infinite precision intermediate result to the low packed double-precision floating-point values in the second source operand, performs rounding and stores the resulting packed double-precision floating-point value to the destination operand (first source operand).

**VFMADD213SD:** Multiplies the low packed double-precision floating-point value from the second source operand to the low packed double-precision floating-point value in the first source operand, adds the negated infinite precision intermediate result to the low packed double-precision floating-point value in the third source operand, performs rounding and stores the resulting packed double-precision floating-point value to the destination operand (first source operand).

**VFMADD231SD:** Multiplies the low packed double-precision floating-point value from the second source to the low packed double-precision floating-point value in the third source operand, adds the negated infinite precision intermediate result to the low packed double-precision floating-point value in the first source operand, performs rounding and stores the resulting packed double-precision floating-point value to the destination operand (first source operand).

VEX.128 and EVEX encoded version: The destination operand (also first source operand) is encoded in `reg_field`. The second source operand is encoded in VEX.vvvv/EVEX.vvvv. The third source operand is encoded in `rm_field`. Bits 127:64 of the destination are unchanged. Bits MAXVL-1:128 of the destination register are zeroed.

EVEX encoded version: The low quadword element of the destination is updated according to the writemask.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NaNs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column.

### Operation

In the operations below, “\*” and “+” symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding).

#### VFMADD132SD DEST, SRC2, SRC3 (EVEX encoded version)

IF (EVEX.b = 1) and SRC3 \*is a register\*

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

IF k1[0] or \*no writemask\*

THEN DEST[63:0] ← RoundFPControl(-(DEST[63:0]\*SRC3[63:0]) + SRC2[63:0])

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[63:0] remains unchanged\*

ELSE ; zeroing-masking

THEN DEST[63:0] ← 0

FI;

FI;

DEST[127:64] ← DEST[127:64]

DEST[MAX\_VL-1:128] ← 0

#### VFMADD213SD DEST, SRC2, SRC3 (EVEX encoded version)

IF (EVEX.b = 1) and SRC3 \*is a register\*

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

IF k1[0] or \*no writemask\*

THEN DEST[63:0] ← RoundFPControl(-(SRC2[63:0]\*DEST[63:0]) + SRC3[63:0])

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[63:0] remains unchanged\*

ELSE ; zeroing-masking

THEN DEST[63:0] ← 0

FI;

FI;

DEST[127:64] ← DEST[127:64]

DEST[MAX\_VL-1:128] ← 0

**VFMADD231SD DEST, SRC2, SRC3 (EVEX encoded version)**

```

IF (EVEX.b = 1) and SRC3 *is a register*
  THEN
    SET_RM(EVEX.RC);
  ELSE
    SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
  THEN DEST[63:0] ← RoundFPControl(-(SRC2[63:0]*SRC3[63:0]) + DEST[63:0])
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[63:0] remains unchanged*
    ELSE ; zeroing-masking
      THEN DEST[63:0] ← 0
    FI;
FI;
DEST[127:64] ← DEST[127:64]
DEST[MAX_VL-1:128] ← 0

```

**VFMADD132SD DEST, SRC2, SRC3 (VEX encoded version)**

```

DEST[63:0] ← RoundFPControl_MXCSR(- (DEST[63:0]*SRC3[63:0]) + SRC2[63:0])
DEST[127:64] ← DEST[127:64]
DEST[MAX_VL-1:128] ← 0

```

**VFMADD213SD DEST, SRC2, SRC3 (VEX encoded version)**

```

DEST[63:0] ← RoundFPControl_MXCSR(- (SRC2[63:0]*DEST[63:0]) + SRC3[63:0])
DEST[127:64] ← DEST[127:64]
DEST[MAX_VL-1:128] ← 0

```

**VFMADD231SD DEST, SRC2, SRC3 (VEX encoded version)**

```

DEST[63:0] ← RoundFPControl_MXCSR(- (SRC2[63:0]*SRC3[63:0]) + DEST[63:0])
DEST[127:64] ← DEST[127:64]
DEST[MAX_VL-1:128] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VFMADDxxxSD __m128d __mm_fmadd_round_sd(__m128d a, __m128d b, __m128d c, int r);
VFMADDxxxSD __m128d __mm_mask_fmadd_sd(__m128d a, __mmask8 k, __m128d b, __m128d c);
VFMADDxxxSD __m128d __mm_maskz_fmadd_sd(__mmask8 k, __m128d a, __m128d b, __m128d c);
VFMADDxxxSD __m128d __mm_mask3_fmadd_sd(__m128d a, __m128d b, __m128d c, __mmask8 k);
VFMADDxxxSD __m128d __mm_mask_fmadd_round_sd(__m128d a, __mmask8 k, __m128d b, __m128d c, int r);
VFMADDxxxSD __m128d __mm_maskz_fmadd_round_sd(__mmask8 k, __m128d a, __m128d b, __m128d c, int r);
VFMADDxxxSD __m128d __mm_mask3_fmadd_round_sd(__m128d a, __m128d b, __m128d c, __mmask8 k, int r);
VFMADDxxxSD __m128d __mm_fmadd_sd (__m128d a, __m128d b, __m128d c);

```

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal

**Other Exceptions**

VEX-encoded instructions, see Exceptions Type 3.

EVEX-encoded instructions, see Exceptions Type E3.

## VFMADD132SS/VFMADD213SS/VFMADD231SS—Fused Negative Multiply-Add of Scalar Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.DDS.LIG.66.0F38.W0 9D /r VFMADD132SS xmm1, xmm2, xmm3/m32	RVM	V/V	FMA	Multiply scalar single-precision floating-point value from xmm1 and xmm3/m32, negate the multiplication result and add to xmm2 and put result in xmm1.
VEX.DDS.LIG.66.0F38.W0 AD /r VFMADD213SS xmm1, xmm2, xmm3/m32	RVM	V/V	FMA	Multiply scalar single-precision floating-point value from xmm1 and xmm2, negate the multiplication result and add to xmm3/m32 and put result in xmm1.
VEX.DDS.LIG.66.0F38.W0 BD /r VFMADD231SS xmm1, xmm2, xmm3/m32	RVM	V/V	FMA	Multiply scalar single-precision floating-point value from xmm2 and xmm3/m32, negate the multiplication result and add to xmm1 and put result in xmm1.
EVEX.DDS.LIG.66.0F38.W0 9D /r VFMADD132SS xmm1 {k1}{z}, xmm2, xmm3/m32{er}	T1S	V/V	AVX512F	Multiply scalar single-precision floating-point value from xmm1 and xmm3/m32, negate the multiplication result and add to xmm2 and put result in xmm1.
EVEX.DDS.LIG.66.0F38.W0 AD /r VFMADD213SS xmm1 {k1}{z}, xmm2, xmm3/m32{er}	T1S	V/V	AVX512F	Multiply scalar single-precision floating-point value from xmm1 and xmm2, negate the multiplication result and add to xmm3/m32 and put result in xmm1.
EVEX.DDS.LIG.66.0F38.W0 BD /r VFMADD231SS xmm1 {k1}{z}, xmm2, xmm3/m32{er}	T1S	V/V	AVX512F	Multiply scalar single-precision floating-point value from xmm2 and xmm3/m32, negate the multiplication result and add to xmm1 and put result in xmm1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RVM	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
T1S	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

VFMADD132SS: Multiplies the low packed single-precision floating-point value from the first source operand to the low packed single-precision floating-point value in the third source operand, adds the negated infinite precision intermediate result to the low packed single-precision floating-point value in the second source operand, performs rounding and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

VFMADD213SS: Multiplies the low packed single-precision floating-point value from the second source operand to the low packed single-precision floating-point value in the first source operand, adds the negated infinite precision intermediate result to the low packed single-precision floating-point value in the third source operand, performs rounding and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

VFMADD231SS: Multiplies the low packed single-precision floating-point value from the second source operand to the low packed single-precision floating-point value in the third source operand, adds the negated infinite precision intermediate result to the low packed single-precision floating-point value in the first source operand, performs rounding and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

VEX.128 and EVEX encoded version: The destination operand (also first source operand) is encoded in `reg_field`. The second source operand is encoded in VEX.vvvv/EVEX.vvvv. The third source operand is encoded in `rm_field`. Bits 127:32 of the destination are unchanged. Bits MAXVL-1:128 of the destination register are zeroed.

EVEX encoded version: The low doubleword element of the destination is updated according to the writemask.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NaNs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column.

### Operation

In the operations below, “\*” and “+” symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding).

#### VFMADD132SS DEST, SRC2, SRC3 (EVEX encoded version)

IF (EVEX.b = 1) and SRC3 \*is a register\*

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

IF k1[0] or \*no writemask\*

THEN DEST[31:0] ← RoundFPControl(-(DEST[31:0]\*SRC3[31:0]) + SRC2[31:0])

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[31:0] remains unchanged\*

ELSE ; zeroing-masking

THEN DEST[31:0] ← 0

FI;

FI;

DEST[127:32] ← DEST[127:32]

DEST[MAX\_VL-1:128] ← 0

#### VFMADD213SS DEST, SRC2, SRC3 (EVEX encoded version)

IF (EVEX.b = 1) and SRC3 \*is a register\*

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

IF k1[0] or \*no writemask\*

THEN DEST[31:0] ← RoundFPControl(-(SRC2[31:0]\*DEST[31:0]) + SRC3[31:0])

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[31:0] remains unchanged\*

ELSE ; zeroing-masking

THEN DEST[31:0] ← 0

FI;

FI;

DEST[127:32] ← DEST[127:32]

DEST[MAX\_VL-1:128] ← 0

**VFMADD231SS DEST, SRC2, SRC3 (EVEX encoded version)**

```

IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
    THEN DEST[31:0] ← RoundFPControl(-(SRC2[31:0]*SRC3[63:0]) + DEST[31:0])
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE ; zeroing-masking
                THEN DEST[31:0] ← 0
        FI;
FI;
DEST[127:32] ← DEST[127:32]
DEST[MAX_VL-1:128] ← 0

```

**VFMADD132SS DEST, SRC2, SRC3 (VEX encoded version)**

```

DEST[31:0] ← RoundFPControl_MXCSR(- (DEST[31:0]*SRC3[31:0]) + SRC2[31:0])
DEST[127:32] ← DEST[127:32]
DEST[MAX_VL-1:128] ← 0

```

**VFMADD213SS DEST, SRC2, SRC3 (VEX encoded version)**

```

DEST[31:0] ← RoundFPControl_MXCSR(- (SRC2[31:0]*DEST[31:0]) + SRC3[31:0])
DEST[127:32] ← DEST[127:32]
DEST[MAX_VL-1:128] ← 0

```

**VFMADD231SS DEST, SRC2, SRC3 (VEX encoded version)**

```

DEST[31:0] ← RoundFPControl_MXCSR(- (SRC2[31:0]*SRC3[31:0]) + DEST[31:0])
DEST[127:32] ← DEST[127:32]
DEST[MAX_VL-1:128] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VFMADDxxxSS __m128 __mm_fmadd_round_ss(__m128 a, __m128 b, __m128 c, int r);
VFMADDxxxSS __m128 __mm_mask_fmadd_ss(__m128 a, __mmask8 k, __m128 b, __m128 c);
VFMADDxxxSS __m128 __mm_maskz_fmadd_ss(__mmask8 k, __m128 a, __m128 b, __m128 c);
VFMADDxxxSS __m128 __mm_mask3_fmadd_ss(__m128 a, __m128 b, __m128 c, __mmask8 k);
VFMADDxxxSS __m128 __mm_mask_fmadd_round_ss(__m128 a, __mmask8 k, __m128 b, __m128 c, int r);
VFMADDxxxSS __m128 __mm_maskz_fmadd_round_ss(__mmask8 k, __m128 a, __m128 b, __m128 c, int r);
VFMADDxxxSS __m128 __mm_mask3_fmadd_round_ss(__m128 a, __m128 b, __m128 c, __mmask8 k, int r);
VFMADDxxxSS __m128 __mm_fmadd_ss (__m128 a, __m128 b, __m128 c);

```

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal

**Other Exceptions**

VEX-encoded instructions, see Exceptions Type 3.

EVEX-encoded instructions, see Exceptions Type E3.



## VFNMSUB132PD/VFNMSUB213PD/VFNMSUB231PD—Fused Negative Multiply-Subtract of Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.NDS.128.66.0F38.W1 9E /r VFNMSUB132PD xmm1, xmm2, xmm3/m128	RVM	V/V	FMA	Multiply packed double-precision floating-point values from xmm1 and xmm3/mem, negate the multiplication result and subtract xmm2 and put result in xmm1.
VEX.NDS.128.66.0F38.W1 AE /r VFNMSUB213PD xmm1, xmm2, xmm3/m128	RVM	V/V	FMA	Multiply packed double-precision floating-point values from xmm1 and xmm2, negate the multiplication result and subtract xmm3/mem and put result in xmm1.
VEX.NDS.128.66.0F38.W1 BE /r VFNMSUB231PD xmm1, xmm2, xmm3/m128	RVM	V/V	FMA	Multiply packed double-precision floating-point values from xmm2 and xmm3/mem, negate the multiplication result and subtract xmm1 and put result in xmm1.
VEX.NDS.256.66.0F38.W1 9E /r VFNMSUB132PD ymm1, ymm2, ymm3/m256	RVM	V/V	FMA	Multiply packed double-precision floating-point values from ymm1 and ymm3/mem, negate the multiplication result and subtract ymm2 and put result in ymm1.
VEX.NDS.256.66.0F38.W1 AE /r VFNMSUB213PD ymm1, ymm2, ymm3/m256	RVM	V/V	FMA	Multiply packed double-precision floating-point values from ymm1 and ymm2, negate the multiplication result and subtract ymm3/mem and put result in ymm1.
VEX.NDS.256.66.0F38.W1 BE /r VFNMSUB231PD ymm1, ymm2, ymm3/m256	RVM	V/V	FMA	Multiply packed double-precision floating-point values from ymm2 and ymm3/mem, negate the multiplication result and subtract ymm1 and put result in ymm1.
EVEX.NDS.128.66.0F38.W1 9E /r VFNMSUB132PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	FV	V/V	AVX512VL AVX512F	Multiply packed double-precision floating-point values from xmm1 and xmm3/m128/m64bcst, negate the multiplication result and subtract xmm2 and put result in xmm1.
EVEX.NDS.128.66.0F38.W1 AE /r VFNMSUB213PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	FV	V/V	AVX512VL AVX512F	Multiply packed double-precision floating-point values from xmm1 and xmm2, negate the multiplication result and subtract xmm3/m128/m64bcst and put result in xmm1.
EVEX.NDS.128.66.0F38.W1 BE /r VFNMSUB231PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	FV	V/V	AVX512VL AVX512F	Multiply packed double-precision floating-point values from xmm2 and xmm3/m128/m64bcst, negate the multiplication result and subtract xmm1 and put result in xmm1.
EVEX.NDS.256.66.0F38.W1 9E /r VFNMSUB132PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	FV	V/V	AVX512VL AVX512F	Multiply packed double-precision floating-point values from ymm1 and ymm3/m256/m64bcst, negate the multiplication result and subtract ymm2 and put result in ymm1.
EVEX.NDS.256.66.0F38.W1 AE /r VFNMSUB213PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	FV	V/V	AVX512VL AVX512F	Multiply packed double-precision floating-point values from ymm1 and ymm2, negate the multiplication result and subtract ymm3/m256/m64bcst and put result in ymm1.
EVEX.NDS.256.66.0F38.W1 BE /r VFNMSUB231PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	FV	V/V	AVX512VL AVX512F	Multiply packed double-precision floating-point values from ymm2 and ymm3/m256/m64bcst, negate the multiplication result and subtract ymm1 and put result in ymm1.
EVEX.NDS.512.66.0F38.W1 9E /r VFNMSUB132PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	FV	V/V	AVX512F	Multiply packed double-precision floating-point values from zmm1 and zmm3/m512/m64bcst, negate the multiplication result and subtract zmm2 and put result in zmm1.
EVEX.NDS.512.66.0F38.W1 AE /r VFNMSUB213PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	FV	V/V	AVX512F	Multiply packed double-precision floating-point values from zmm1 and zmm2, negate the multiplication result and subtract zmm3/m512/m64bcst and put result in zmm1.
EVEX.NDS.512.66.0F38.W1 BE /r VFNMSUB231PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	FV	V/V	AVX512F	Multiply packed double-precision floating-point values from zmm2 and zmm3/m512/m64bcst, negate the multiplication result and subtract zmm1 and put result in zmm1.

## Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RVM	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FV	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

## Description

**VFNSUB132PD:** Multiplies the two, four or eight packed double-precision floating-point values from the first source operand to the two, four or eight packed double-precision floating-point values in the third source operand. From negated infinite precision intermediate results, subtracts the two, four or eight packed double-precision floating-point values in the second source operand, performs rounding and stores the resulting two, four or eight packed double-precision floating-point values to the destination operand (first source operand).

**VFNSUB213PD:** Multiplies the two, four or eight packed double-precision floating-point values from the second source operand to the two, four or eight packed double-precision floating-point values in the first source operand. From negated infinite precision intermediate results, subtracts the two, four or eight packed double-precision floating-point values in the third source operand, performs rounding and stores the resulting two, four or eight packed double-precision floating-point values to the destination operand (first source operand).

**VFNSUB231PD:** Multiplies the two, four or eight packed double-precision floating-point values from the second source to the two, four or eight packed double-precision floating-point values in the third source operand. From negated infinite precision intermediate results, subtracts the two, four or eight packed double-precision floating-point values in the first source operand, performs rounding and stores the resulting two, four or eight packed double-precision floating-point values to the destination operand (first source operand).

**EVEX encoded versions:** The destination operand (also first source operand) and the second source operand are ZMM/YMM/XMM register. The third 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 64-bit memory location. The destination operand is conditionally updated with write mask k1.

**VEX.256 encoded version:** The destination operand (also first source operand) is a YMM register and encoded in `reg_field`. The second source operand is a YMM register and encoded in `VEX.vvvv`. The third source operand is a YMM register or a 256-bit memory location and encoded in `rm_field`.

**VEX.128 encoded version:** The destination operand (also first source operand) is a XMM register and encoded in `reg_field`. The second source operand is a XMM register and encoded in `VEX.vvvv`. The third source operand is a XMM register or a 128-bit memory location and encoded in `rm_field`. The upper 128 bits of the YMM destination register are zeroed.

## Operation

In the operations below, "\*" and "-" symbols represent multiplication and subtraction with infinite precision inputs and outputs (no rounding).

**VFNSUB132PD DEST, SRC2, SRC3 (VEX encoded version)**

IF (VEX.128) THEN

    MAXNUM ← 2

ELSEIF (VEX.256)

    MAXNUM ← 4

FI

For i = 0 to MAXNUM-1 {

    n ← 64\*i;

    DEST[n+63:n] ← RoundFPControl\_MXCSR( - (DEST[n+63:n]\*SRC3[n+63:n]) - SRC2[n+63:n])

}

IF (VEX.128) THEN

    DEST[MAX\_VL-1:128] ← 0

ELSEIF (VEX.256)

    DEST[MAX\_VL-1:256] ← 0

FI

**VFNSUB213PD DEST, SRC2, SRC3 (VEX encoded version)**

```

IF (VEX.128) THEN
    MAXNUM ← 2
ELSEIF (VEX.256)
    MAXNUM ← 4
FI

For i = 0 to MAXNUM-1 {
    n ← 64*i;
    DEST[n+63:n] ← RoundFPControl_MXCSR( - (SRC2[n+63:n]*DEST[n+63:n]) - SRC3[n+63:n])
}

IF (VEX.128) THEN
    DEST[MAX_VL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAX_VL-1:256] ← 0
FI

```

**VFNSUB231PD DEST, SRC2, SRC3 (VEX encoded version)**

```

IF (VEX.128) THEN
    MAXNUM ← 2
ELSEIF (VEX.256)
    MAXNUM ← 4
FI

For i = 0 to MAXNUM-1 {
    n ← 64*i;
    DEST[n+63:n] ← RoundFPControl_MXCSR( - (SRC2[n+63:n]*SRC3[n+63:n]) - DEST[n+63:n])
}

IF (VEX.128) THEN
    DEST[MAX_VL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAX_VL-1:256] ← 0
FI

```

**VFNSUB132PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)**

```

(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;

FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ←
            RoundFPControl(-(DEST[i+63:i]*SRC3[i+63:i]) - SRC2[i+63:i])
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VFNMSUB132PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 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] ←
            RoundFPControl_MXCSR(-(DEST[i+63:i]*SRC3[63:0]) - SRC2[i+63:i])
        ELSE
          DEST[i+63:i] ←
            RoundFPControl_MXCSR(-(DEST[i+63:i]*SRC3[j+63:i]) - SRC2[i+63:i])
        FI;
      ELSE
        IF *merging-masking*           ; merging-masking
          THEN *DEST[i+63:i] remains unchanged*
        ELSE                             ; zeroing-masking
          DEST[i+63:i] ← 0
        FI
      FI;
    ENDFOR
  DEST[MAX_VL-1:VL] ← 0

```

**VFNMSUB213PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

IF (VL = 512) AND (EVEX.b = 1)

```

  THEN
    SET_RM(EVEX.RC);
  ELSE
    SET_RM(MXCSR.RM);
  FI;
FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] ←
      RoundFPControl(-(SRC2[j+63:i]*DEST[i+63:i]) - SRC3[j+63:i])
    ELSE
      IF *merging-masking*           ; merging-masking
        THEN *DEST[i+63:i] remains unchanged*
      ELSE                             ; zeroing-masking
        DEST[i+63:i] ← 0
      FI
    FI;
  ENDFOR
  DEST[MAX_VL-1:VL] ← 0

```

**VFNMSUB213PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 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] ←
          RoundFPControl_MXCSR(-(SRC2[i+63:i]*DEST[i+63:i]) - SRC3[63:0])
        ELSE
          DEST[i+63:i] ←
          RoundFPControl_MXCSR(-(SRC2[i+63:i]*DEST[i+63:i]) - SRC3[i+63:i])
        FI;
      ELSE
        IF *merging-masking*           ; merging-masking
          THEN *DEST[i+63:i] remains unchanged*
        ELSE                             ; zeroing-masking
          DEST[i+63:i] ← 0
        FI
      FI;
    ENDFOR
  DEST[MAX_VL-1:VL] ← 0

```

**VFNMSUB231PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

IF (VL = 512) AND (EVEX.b = 1)

```

  THEN
    SET_RM(EVEX.RC);
  ELSE
    SET_RM(MXCSR.RM);
  FI;
FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] ←
    RoundFPControl(-(SRC2[i+63:i]*SRC3[i+63:i]) - DEST[i+63:i])
  ELSE
    IF *merging-masking*           ; merging-masking
      THEN *DEST[i+63:i] remains unchanged*
    ELSE                             ; zeroing-masking
      DEST[i+63:i] ← 0
    FI
  FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VFNMSUB231PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 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] ←
            RoundFPControl_MXCSR(-(SRC2[i+63:i]*SRC3[63:0]) - DEST[i+63:i])
        ELSE
          DEST[i+63:i] ←
            RoundFPControl_MXCSR(-(SRC2[i+63:i]*SRC3[j+63:i]) - DEST[i+63:i])
        FI;
      ELSE
        IF *merging-masking*           ; merging-masking
          THEN *DEST[i+63:i] remains unchanged*
        ELSE                             ; zeroing-masking
          DEST[i+63:i] ← 0
        FI
      FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0
  
```

#### Intel C/C++ Compiler Intrinsic Equivalent

```

VFNMSUBxxxPD __m512d _mm512_fnmsub_pd(__m512d a, __m512d b, __m512d c);
VFNMSUBxxxPD __m512d _mm512_fnmsub_round_pd(__m512d a, __m512d b, __m512d c, int r);
VFNMSUBxxxPD __m512d _mm512_mask_fnmsub_pd(__m512d a, __mmask8 k, __m512d b, __m512d c);
VFNMSUBxxxPD __m512d _mm512_maskz_fnmsub_pd(__mmask8 k, __m512d a, __m512d b, __m512d c);
VFNMSUBxxxPD __m512d _mm512_mask3_fnmsub_pd(__m512d a, __m512d b, __m512d c, __mmask8 k);
VFNMSUBxxxPD __m512d _mm512_mask_fnmsub_round_pd(__m512d a, __mmask8 k, __m512d b, __m512d c, int r);
VFNMSUBxxxPD __m512d _mm512_maskz_fnmsub_round_pd(__mmask8 k, __m512d a, __m512d b, __m512d c, int r);
VFNMSUBxxxPD __m512d _mm512_mask3_fnmsub_round_pd(__m512d a, __m512d b, __m512d c, __mmask8 k, int r);
VFNMSUBxxxPD __m256d _mm256_mask_fnmsub_pd(__m256d a, __mmask8 k, __m256d b, __m256d c);
VFNMSUBxxxPD __m256d _mm256_maskz_fnmsub_pd(__mmask8 k, __m256d a, __m256d b, __m256d c);
VFNMSUBxxxPD __m256d _mm256_mask3_fnmsub_pd(__m256d a, __m256d b, __m256d c, __mmask8 k);
VFNMSUBxxxPD __m128d _mm_mask_fnmsub_pd(__m128d a, __mmask8 k, __m128d b, __m128d c);
VFNMSUBxxxPD __m128d _mm_maskz_fnmsub_pd(__mmask8 k, __m128d a, __m128d b, __m128d c);
VFNMSUBxxxPD __m128d _mm_mask3_fnmsub_pd(__m128d a, __m128d b, __m128d c, __mmask8 k);
VFNMSUBxxxPD __m128d _mm_fnmsub_pd (__m128d a, __m128d b, __m128d c);
VFNMSUBxxxPD __m256d _mm256_fnmsub_pd (__m256d a, __m256d b, __m256d c);
  
```

#### SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

#### Other Exceptions

VEX-encoded instructions, see Exceptions Type 2.

EVEX-encoded instructions, see Exceptions Type E2.

## VFNSUB132PS/VFNSUB213PS/VFNSUB231PS—Fused Negative Multiply-Subtract of Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.NDS.128.66.0F38.W0 9E /r VFNSUB132PS xmm1, xmm2, xmm3/m128	RVM	V/V	FMA	Multiply packed single-precision floating-point values from xmm1 and xmm3/mem, negate the multiplication result and subtract xmm2 and put result in xmm1.
VEX.NDS.128.66.0F38.W0 AE /r VFNSUB213PS xmm1, xmm2, xmm3/m128	RVM	V/V	FMA	Multiply packed single-precision floating-point values from xmm1 and xmm2, negate the multiplication result and subtract xmm3/mem and put result in xmm1.
VEX.NDS.128.66.0F38.W0 BE /r VFNSUB231PS xmm1, xmm2, xmm3/m128	RVM	V/V	FMA	Multiply packed single-precision floating-point values from xmm2 and xmm3/mem, negate the multiplication result and subtract xmm1 and put result in xmm1.
VEX.NDS.256.66.0F38.W0 9E /r VFNSUB132PS ymm1, ymm2, ymm3/m256	RVM	V/V	FMA	Multiply packed single-precision floating-point values from ymm1 and ymm3/mem, negate the multiplication result and subtract ymm2 and put result in ymm1.
VEX.NDS.256.66.0F38.W0 AE /r VFNSUB213PS ymm1, ymm2, ymm3/m256	RVM	V/V	FMA	Multiply packed single-precision floating-point values from ymm1 and ymm2, negate the multiplication result and subtract ymm3/mem and put result in ymm1.
VEX.NDS.256.66.0F38.0 BE /r VFNSUB231PS ymm1, ymm2, ymm3/m256	RVM	V/V	FMA	Multiply packed single-precision floating-point values from ymm2 and ymm3/mem, negate the multiplication result and subtract ymm1 and put result in ymm1.
EVEX.NDS.128.66.0F38.W0 9E /r VFNSUB132PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	FV	V/V	AVX512VL AVX512F	Multiply packed single-precision floating-point values from xmm1 and xmm3/m128/m32bcst, negate the multiplication result and subtract xmm2 and put result in xmm1.
EVEX.NDS.128.66.0F38.W0 AE /r VFNSUB213PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	FV	V/V	AVX512VL AVX512F	Multiply packed single-precision floating-point values from xmm1 and xmm2, negate the multiplication result and subtract xmm3/m128/m32bcst and put result in xmm1.
EVEX.NDS.128.66.0F38.W0 BE /r VFNSUB231PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	FV	V/V	AVX512VL AVX512F	Multiply packed single-precision floating-point values from xmm2 and xmm3/m128/m32bcst, negate the multiplication result subtract add to xmm1 and put result in xmm1.
EVEX.NDS.256.66.0F38.W0 9E /r VFNSUB132PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	FV	V/V	AVX512VL AVX512F	Multiply packed single-precision floating-point values from ymm1 and ymm3/m256/m32bcst, negate the multiplication result and subtract ymm2 and put result in ymm1.
EVEX.NDS.256.66.0F38.W0 AE /r VFNSUB213PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	FV	V/V	AVX512VL AVX512F	Multiply packed single-precision floating-point values from ymm1 and ymm2, negate the multiplication result and subtract ymm3/m256/m32bcst and put result in ymm1.
EVEX.NDS.256.66.0F38.W0 BE /r VFNSUB231PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	FV	V/V	AVX512VL AVX512F	Multiply packed single-precision floating-point values from ymm2 and ymm3/m256/m32bcst, negate the multiplication result subtract add to ymm1 and put result in ymm1.
EVEX.NDS.512.66.0F38.W0 9E /r VFNSUB132PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}	FV	V/V	AVX512F	Multiply packed single-precision floating-point values from zmm1 and zmm3/m512/m32bcst, negate the multiplication result and subtract zmm2 and put result in zmm1.
EVEX.NDS.512.66.0F38.W0 AE /r VFNSUB213PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}	FV	V/V	AVX512F	Multiply packed single-precision floating-point values from zmm1 and zmm2, negate the multiplication result and subtract zmm3/m512/m32bcst and put result in zmm1.
EVEX.NDS.512.66.0F38.W0 BE /r VFNSUB231PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}	FV	V/V	AVX512F	Multiply packed single-precision floating-point values from zmm2 and zmm3/m512/m32bcst, negate the multiplication result subtract add to zmm1 and put result in zmm1.

## Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RVM	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FV	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

## Description

**VFNMSUB132PS:** Multiplies the four, eight or sixteen packed single-precision floating-point values from the first source operand to the four, eight or sixteen packed single-precision floating-point values in the third source operand. From negated infinite precision intermediate results, subtracts the four, eight or sixteen packed single-precision floating-point values in the second source operand, performs rounding and stores the resulting four, eight or sixteen packed single-precision floating-point values to the destination operand (first source operand).

**VFNMSUB213PS:** Multiplies the four, eight or sixteen packed single-precision floating-point values from the second source operand to the four, eight or sixteen packed single-precision floating-point values in the first source operand. From negated infinite precision intermediate results, subtracts the four, eight or sixteen packed single-precision floating-point values in the third source operand, performs rounding and stores the resulting four, eight or sixteen packed single-precision floating-point values to the destination operand (first source operand).

**VFNMSUB231PS:** Multiplies the four, eight or sixteen packed single-precision floating-point values from the second source to the four, eight or sixteen packed single-precision floating-point values in the third source operand. From negated infinite precision intermediate results, subtracts the four, eight or sixteen packed single-precision floating-point values in the first source operand, performs rounding and stores the resulting four, eight or sixteen packed single-precision floating-point values to the destination operand (first source operand).

**EVEX encoded versions:** The destination operand (also first source operand) and the second source operand are ZMM/YMM/XMM register. The third 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 32-bit memory location. The destination operand is conditionally updated with write mask k1.

**VEX.256 encoded version:** The destination operand (also first source operand) is a YMM register and encoded in `reg_field`. The second source operand is a YMM register and encoded in `VEX.vvvv`. The third source operand is a YMM register or a 256-bit memory location and encoded in `rm_field`.

**VEX.128 encoded version:** The destination operand (also first source operand) is a XMM register and encoded in `reg_field`. The second source operand is a XMM register and encoded in `VEX.vvvv`. The third source operand is a XMM register or a 128-bit memory location and encoded in `rm_field`. The upper 128 bits of the YMM destination register are zeroed.

## Operation

In the operations below, "\*" and "-" symbols represent multiplication and subtraction with infinite precision inputs and outputs (no rounding).

**VFNMSUB132PS DEST, SRC2, SRC3 (VEX encoded version)**

```

IF (VEX.128) THEN
    MAXNUM ← 2
ELSEIF (VEX.256)
    MAXNUM ← 4
FI
For i = 0 to MAXNUM-1 {
    n ← 32*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR( - (DEST[n+31:n]*SRC3[n+31:n]) - SRC2[n+31:n])
}
IF (VEX.128) THEN
    DEST[MAX_VL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAX_VL-1:256] ← 0
FI

```



**VFNMSUB213PS DEST, SRC2, SRC3 (VEX encoded version)**

```

IF (VEX.128) THEN
    MAXNUM ← 2
ELSEIF (VEX.256)
    MAXNUM ← 4
FI
For i = 0 to MAXNUM-1 {
    n ← 32*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR( - (SRC2[n+31:n]*DEST[n+31:n]) - SRC3[n+31:n])
}
IF (VEX.128) THEN
    DEST[MAX_VL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAX_VL-1:256] ← 0
FI

```

**VFNMSUB231PS DEST, SRC2, SRC3 (VEX encoded version)**

```

IF (VEX.128) THEN
    MAXNUM ← 2
ELSEIF (VEX.256)
    MAXNUM ← 4
FI
For i = 0 to MAXNUM-1 {
    n ← 32*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR( - (SRC2[n+31:n]*SRC3[n+31:n]) - DEST[n+31:n])
}
IF (VEX.128) THEN
    DEST[MAX_VL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAX_VL-1:256] ← 0
FI

```

**VFNMSUB132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)**

```

(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ←
            RoundFPControl(-(DEST[i+31:i]*SRC3[j+31:i]) - SRC2[i+31:i])
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VFNMSUB132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)**  
 (KL, VL) = (4, 128), (8, 256), (16, 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] ←
            RoundFPControl_MXCSR(-(DEST[i+31:i]*SRC3[31:0]) - SRC2[i+31:i])
        ELSE
          DEST[i+31:i] ←
            RoundFPControl_MXCSR(-(DEST[i+31:i]*SRC3[j+31:i]) - SRC2[i+31:i])
        FI;
      ELSE
        IF *merging-masking*           ; merging-masking
          THEN *DEST[i+31:i] remains unchanged*
        ELSE                             ; zeroing-masking
          DEST[i+31:i] ← 0
        FI
      FI;
    ENDFOR
  DEST[MAX_VL-1:VL] ← 0

```

**VFNMSUB213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

IF (VL = 512) AND (EVEX.b = 1)

```

  THEN
    SET_RM(EVEX.RC);
  ELSE
    SET_RM(MXCSR.RM);
  FI;
FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ←
      RoundFPControl_MXCSR(-(SRC2[i+31:i]*DEST[i+31:i]) - SRC3[i+31:i])
    ELSE
      IF *merging-masking*           ; merging-masking
        THEN *DEST[i+31:i] remains unchanged*
      ELSE                             ; zeroing-masking
        DEST[i+31:i] ← 0
      FI
    FI;
  ENDFOR
  DEST[MAX_VL-1:VL] ← 0

```

**VFNMSUB213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)**

(KL, VL) = (4, 128), (8, 256), (16, 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] ←
            RoundFPControl_MXCSR(-(SRC2[i+31:i]*DEST[i+31:i]) - SRC3[31:0])
        ELSE
          DEST[i+31:i] ←
            RoundFPControl_MXCSR(-(SRC2[i+31:i]*DEST[i+31:i]) - SRC3[i+31:i])
        FI;
      ELSE
        IF *merging-masking*           ; merging-masking
          THEN *DEST[i+31:i] remains unchanged*
        ELSE                             ; zeroing-masking
          DEST[i+31:i] ← 0
        FI
      FI;
    ENDFOR
  DEST[MAX_VL-1:VL] ← 0

```

**VFNMSUB231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

IF (VL = 512) AND (EVEX.b = 1)

```

  THEN
    SET_RM(EVEX.RC);
  ELSE
    SET_RM(MXCSR.RM);
  FI;
FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ←
      RoundFPControl_MXCSR(-(SRC2[i+31:i]*SRC3[i+31:i]) - DEST[i+31:i])
    ELSE
      IF *merging-masking*           ; merging-masking
        THEN *DEST[i+31:i] remains unchanged*
      ELSE                             ; zeroing-masking
        DEST[i+31:i] ← 0
      FI
    FI;
  ENDFOR
  DEST[MAX_VL-1:VL] ← 0

```

**VFNMSUB231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)**

(KL, VL) = (4, 128), (8, 256), (16, 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] ←
      RoundFPControl_MXCSR(-(SRC2[i+31:i]*SRC3[31:0]) - DEST[i+31:i])
    ELSE
      DEST[i+31:i] ←
      RoundFPControl_MXCSR(-(SRC2[i+31:i]*SRC3[j+31:i]) - DEST[i+31:i])
  FI;
ELSE
  IF *merging-masking*           ; merging-masking
    THEN *DEST[i+31:i] remains unchanged*
  ELSE                             ; zeroing-masking
    DEST[i+31:i] ← 0
  FI
FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

### Intel C/C++ Compiler Intrinsic Equivalent

```

VFNMSUBxxxPS __m512 __mm512_fnmsub_ps(__m512 a, __m512 b, __m512 c);
VFNMSUBxxxPS __m512 __mm512_fnmsub_round_ps(__m512 a, __m512 b, __m512 c, int r);
VFNMSUBxxxPS __m512 __mm512_mask_fnmsub_ps(__m512 a, __mmask16 k, __m512 b, __m512 c);
VFNMSUBxxxPS __m512 __mm512_maskz_fnmsub_ps(__mmask16 k, __m512 a, __m512 b, __m512 c);
VFNMSUBxxxPS __m512 __mm512_mask3_fnmsub_ps(__m512 a, __m512 b, __m512 c, __mmask16 k);
VFNMSUBxxxPS __m512 __mm512_mask_fnmsub_round_ps(__m512 a, __mmask16 k, __m512 b, __m512 c, int r);
VFNMSUBxxxPS __m512 __mm512_maskz_fnmsub_round_ps(__mmask16 k, __m512 a, __m512 b, __m512 c, int r);
VFNMSUBxxxPS __m512 __mm512_mask3_fnmsub_round_ps(__m512 a, __m512 b, __m512 c, __mmask16 k, int r);
VFNMSUBxxxPS __m256 __mm256_mask_fnmsub_ps(__m256 a, __mmask8 k, __m256 b, __m256 c);
VFNMSUBxxxPS __m256 __mm256_maskz_fnmsub_ps(__mmask8 k, __m256 a, __m256 b, __m256 c);
VFNMSUBxxxPS __m256 __mm256_mask3_fnmsub_ps(__m256 a, __m256 b, __m256 c, __mmask8 k);
VFNMSUBxxxPS __m128 __mm_mask_fnmsub_ps(__m128 a, __mmask8 k, __m128 b, __m128 c);
VFNMSUBxxxPS __m128 __mm_maskz_fnmsub_ps(__mmask8 k, __m128 a, __m128 b, __m128 c);
VFNMSUBxxxPS __m128 __mm_mask3_fnmsub_ps(__m128 a, __m128 b, __m128 c, __mmask8 k);
VFNMSUBxxxPS __m128 __mm_fnmsub_ps (__m128 a, __m128 b, __m128 c);
VFNMSUBxxxPS __m256 __mm256_fnmsub_ps (__m256 a, __m256 b, __m256 c);

```

### SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

### Other Exceptions

VEX-encoded instructions, see Exceptions Type 2.

EVEX-encoded instructions, see Exceptions Type E2.

## VFNMSUB132SD/VFNMSUB213SD/VFNMSUB231SD—Fused Negative Multiply-Subtract of Scalar Double-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.DDS.LIG.66.0F38.W1 9F /r VFNMSUB132SD xmm1, xmm2, xmm3/m64	RVM	V/V	FMA	Multiply scalar double-precision floating-point value from xmm1 and xmm3/mem, negate the multiplication result and subtract xmm2 and put result in xmm1.
VEX.DDS.LIG.66.0F38.W1 AF /r VFNMSUB213SD xmm1, xmm2, xmm3/m64	RVM	V/V	FMA	Multiply scalar double-precision floating-point value from xmm1 and xmm2, negate the multiplication result and subtract xmm3/mem and put result in xmm1.
VEX.DDS.LIG.66.0F38.W1 BF /r VFNMSUB231SD xmm1, xmm2, xmm3/m64	RVM	V/V	FMA	Multiply scalar double-precision floating-point value from xmm2 and xmm3/mem, negate the multiplication result and subtract xmm1 and put result in xmm1.
EVEX.DDS.LIG.66.0F38.W1 9F /r VFNMSUB132SD xmm1 {k1}{z}, xmm2, xmm3/m64{er}	T1S	V/V	AVX512F	Multiply scalar double-precision floating-point value from xmm1 and xmm3/m64, negate the multiplication result and subtract xmm2 and put result in xmm1.
EVEX.DDS.LIG.66.0F38.W1 AF /r VFNMSUB213SD xmm1 {k1}{z}, xmm2, xmm3/m64{er}	T1S	V/V	AVX512F	Multiply scalar double-precision floating-point value from xmm1 and xmm2, negate the multiplication result and subtract xmm3/m64 and put result in xmm1.
EVEX.DDS.LIG.66.0F38.W1 BF /r VFNMSUB231SD xmm1 {k1}{z}, xmm2, xmm3/m64{er}	T1S	V/V	AVX512F	Multiply scalar double-precision floating-point value from xmm2 and xmm3/m64, negate the multiplication result and subtract xmm1 and put result in xmm1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RVM	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
T1S	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

**VFNMSUB132SD:** Multiplies the low packed double-precision floating-point value from the first source operand to the low packed double-precision floating-point value in the third source operand. From negated infinite precision intermediate result, subtracts the low double-precision floating-point value in the second source operand, performs rounding and stores the resulting packed double-precision floating-point value to the destination operand (first source operand).

**VFNMSUB213SD:** Multiplies the low packed double-precision floating-point value from the second source operand to the low packed double-precision floating-point value in the first source operand. From negated infinite precision intermediate result, subtracts the low double-precision floating-point value in the third source operand, performs rounding and stores the resulting packed double-precision floating-point value to the destination operand (first source operand).

**VFNMSUB231SD:** Multiplies the low packed double-precision floating-point value from the second source to the low packed double-precision floating-point value in the third source operand. From negated infinite precision intermediate result, subtracts the low double-precision floating-point value in the first source operand, performs rounding and stores the resulting packed double-precision floating-point value to the destination operand (first source operand).

VEX.128 and EVEX encoded version: The destination operand (also first source operand) is encoded in `reg_field`. The second source operand is encoded in VEX.vvvv/EVEX.vvvv. The third source operand is encoded in `rm_field`. Bits 127:64 of the destination are unchanged. Bits MAXVL-1:128 of the destination register are zeroed.

EVEX encoded version: The low quadword element of the destination is updated according to the writemask.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NaNs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column.

### Operation

In the operations below, "\*" and "-" symbols represent multiplication and subtraction with infinite precision inputs and outputs (no rounding).

#### VFNMSUB132SD DEST, SRC2, SRC3 (EVEX encoded version)

IF (EVEX.b = 1) and SRC3 \*is a register\*

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

IF k1[0] or \*no writemask\*

THEN DEST[63:0] ← RoundFPControl(-(DEST[63:0]\*SRC3[63:0]) - SRC2[63:0])

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[63:0] remains unchanged\*

ELSE ; zeroing-masking

THEN DEST[63:0] ← 0

FI;

FI;

DEST[127:64] ← DEST[127:64]

DEST[MAX\_VL-1:128] ← 0

#### VFNMSUB213SD DEST, SRC2, SRC3 (EVEX encoded version)

IF (EVEX.b = 1) and SRC3 \*is a register\*

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

IF k1[0] or \*no writemask\*

THEN DEST[63:0] ← RoundFPControl(-(SRC2[63:0]\*DEST[63:0]) - SRC3[63:0])

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[63:0] remains unchanged\*

ELSE ; zeroing-masking

THEN DEST[63:0] ← 0

FI;

FI;

DEST[127:64] ← DEST[127:64]

DEST[MAX\_VL-1:128] ← 0

**VFNMSUB231SD DEST, SRC2, SRC3 (EVEX encoded version)**

```

IF (EVEX.b = 1) and SRC3 *is a register*
  THEN
    SET_RM(EVEX.RC);
  ELSE
    SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
  THEN DEST[63:0] ← RoundFPControl(-(SRC2[63:0]*SRC3[63:0]) - DEST[63:0])
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[63:0] remains unchanged*
    ELSE ; zeroing-masking
      THEN DEST[63:0] ← 0
    FI;
FI;
DEST[127:64] ← DEST[127:64]
DEST[MAX_VL-1:128] ← 0

```

**VFNMSUB132SD DEST, SRC2, SRC3 (VEX encoded version)**

```

DEST[63:0] ← RoundFPControl_MXCSR(- (DEST[63:0]*SRC3[63:0]) - SRC2[63:0])
DEST[127:64] ← DEST[127:64]
DEST[MAX_VL-1:128] ← 0

```

**VFNMSUB213SD DEST, SRC2, SRC3 (VEX encoded version)**

```

DEST[63:0] ← RoundFPControl_MXCSR(- (SRC2[63:0]*DEST[63:0]) - SRC3[63:0])
DEST[127:64] ← DEST[127:64]
DEST[MAX_VL-1:128] ← 0

```

**VFNMSUB231SD DEST, SRC2, SRC3 (VEX encoded version)**

```

DEST[63:0] ← RoundFPControl_MXCSR(- (SRC2[63:0]*SRC3[63:0]) - DEST[63:0])
DEST[127:64] ← DEST[127:64]
DEST[MAX_VL-1:128] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VFNMSubxxxSD __m128d __mm_fnmsub_round_sd(__m128d a, __m128d b, __m128d c, int r);
VFNMSubxxxSD __m128d __mm_mask_fnmsub_sd(__m128d a, __mmask8 k, __m128d b, __m128d c);
VFNMSubxxxSD __m128d __mm_maskz_fnmsub_sd(__mmask8 k, __m128d a, __m128d b, __m128d c);
VFNMSubxxxSD __m128d __mm_mask3_fnmsub_sd(__m128d a, __m128d b, __m128d c, __mmask8 k);
VFNMSubxxxSD __m128d __mm_mask_fnmsub_round_sd(__m128d a, __mmask8 k, __m128d b, __m128d c, int r);
VFNMSubxxxSD __m128d __mm_maskz_fnmsub_round_sd(__mmask8 k, __m128d a, __m128d b, __m128d c, int r);
VFNMSubxxxSD __m128d __mm_mask3_fnmsub_round_sd(__m128d a, __m128d b, __m128d c, __mmask8 k, int r);
VFNMSubxxxSD __m128d __mm_fnmsub_sd (__m128d a, __m128d b, __m128d c);

```

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal

**Other Exceptions**

VEX-encoded instructions, see Exceptions Type 3.

EVEX-encoded instructions, see Exceptions Type E3.

## VFNSUB132SS/VFNSUB213SS/VFNSUB231SS—Fused Negative Multiply-Subtract of Scalar Single-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.DDS.LIG.66.0F38.W0 9F /r VFNSUB132SS xmm1, xmm2, xmm3/m32	RVM	V/V	FMA	Multiply scalar single-precision floating-point value from xmm1 and xmm3/m32, negate the multiplication result and subtract xmm2 and put result in xmm1.
VEX.DDS.LIG.66.0F38.W0 AF /r VFNSUB213SS xmm1, xmm2, xmm3/m32	RVM	V/V	FMA	Multiply scalar single-precision floating-point value from xmm1 and xmm2, negate the multiplication result and subtract xmm3/m32 and put result in xmm1.
VEX.DDS.LIG.66.0F38.W0 BF /r VFNSUB231SS xmm1, xmm2, xmm3/m32	RVM	V/V	FMA	Multiply scalar single-precision floating-point value from xmm2 and xmm3/m32, negate the multiplication result and subtract xmm1 and put result in xmm1.
EVEX.DDS.LIG.66.0F38.W0 9F /r VFNSUB132SS xmm1 {k1}{z}, xmm2, xmm3/m32{er}	T1S	V/V	AVX512F	Multiply scalar single-precision floating-point value from xmm1 and xmm3/m32, negate the multiplication result and subtract xmm2 and put result in xmm1.
EVEX.DDS.LIG.66.0F38.W0 AF /r VFNSUB213SS xmm1 {k1}{z}, xmm2, xmm3/m32{er}	T1S	V/V	AVX512F	Multiply scalar single-precision floating-point value from xmm1 and xmm2, negate the multiplication result and subtract xmm3/m32 and put result in xmm1.
EVEX.DDS.LIG.66.0F38.W0 BF /r VFNSUB231SS xmm1 {k1}{z}, xmm2, xmm3/m32{er}	T1S	V/V	AVX512F	Multiply scalar single-precision floating-point value from xmm2 and xmm3/m32, negate the multiplication result and subtract xmm1 and put result in xmm1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RVM	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
T1S	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

**VFNSUB132SS:** Multiplies the low packed single-precision floating-point value from the first source operand to the low packed single-precision floating-point value in the third source operand. From negated infinite precision intermediate result, the low single-precision floating-point value in the second source operand, performs rounding and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

**VFNSUB213SS:** Multiplies the low packed single-precision floating-point value from the second source operand to the low packed single-precision floating-point value in the first source operand. From negated infinite precision intermediate result, the low single-precision floating-point value in the third source operand, performs rounding and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

**VFNSUB231SS:** Multiplies the low packed single-precision floating-point value from the second source to the low packed single-precision floating-point value in the third source operand. From negated infinite precision intermediate result, the low single-precision floating-point value in the first source operand, performs rounding and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

**VEX.128 and EVEX encoded version:** The destination operand (also first source operand) is encoded in `reg_field`. The second source operand is encoded in `VEX.vvvv/EVEX.vvvv`. The third source operand is encoded in `rm_field`. Bits 127:32 of the destination are unchanged. Bits `MAXVL-1:128` of the destination register are zeroed.



EVEX encoded version: The low doubleword element of the destination is updated according to the writemask. Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NaNs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column.

### Operation

In the operations below, “\*” and “-” symbols represent multiplication and subtraction with infinite precision inputs and outputs (no rounding).

#### VFNMSUB132SS DEST, SRC2, SRC3 (EVEX encoded version)

```
IF (EVEX.b = 1) and SRC3 *is a register*
  THEN
    SET_RM(EVEX.RC);
  ELSE
    SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
  THEN DEST[31:0] ← RoundFPControl(-(DEST[31:0]*SRC3[31:0]) - SRC2[31:0])
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[31:0] remains unchanged*
    ELSE ; zeroing-masking
      THEN DEST[31:0] ← 0
    FI;
  FI;
DEST[127:32] ← DEST[127:32]
DEST[MAX_VL-1:128] ← 0
```

#### VFNMSUB213SS DEST, SRC2, SRC3 (EVEX encoded version)

```
IF (EVEX.b = 1) and SRC3 *is a register*
  THEN
    SET_RM(EVEX.RC);
  ELSE
    SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
  THEN DEST[31:0] ← RoundFPControl(-(SRC2[31:0]*DEST[31:0]) - SRC3[31:0])
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[31:0] remains unchanged*
    ELSE ; zeroing-masking
      THEN DEST[31:0] ← 0
    FI;
  FI;
DEST[127:32] ← DEST[127:32]
DEST[MAX_VL-1:128] ← 0
```

#### VFNMSUB231SS DEST, SRC2, SRC3 (EVEX encoded version)

```
IF (EVEX.b = 1) and SRC3 *is a register*
  THEN
    SET_RM(EVEX.RC);
  ELSE
    SET_RM(MXCSR.RM);
```

```

FI;
IF k1[0] or *no writemask*
  THEN  DEST[31:0] ← RoundFPControl(-(SRC2[31:0]*SRC3[63:0]) - DEST[31:0])
  ELSE
    IF *merging-masking*           ; merging-masking
      THEN *DEST[31:0] remains unchanged*
    ELSE                             ; zeroing-masking
      THEN DEST[31:0] ← 0
FI;
FI;
DEST[127:32] ← DEST[127:32]
DEST[MAX_VL-1:128] ← 0

```

**VFNSUB132SS DEST, SRC2, SRC3 (VEX encoded version)**

```

DEST[31:0] ← RoundFPControl_MXCSR(- (DEST[31:0]*SRC3[31:0]) - SRC2[31:0])
DEST[127:32] ← DEST[127:32]
DEST[MAX_VL-1:128] ← 0

```

**VFNSUB213SS DEST, SRC2, SRC3 (VEX encoded version)**

```

DEST[31:0] ← RoundFPControl_MXCSR(- (SRC2[31:0]*DEST[31:0]) - SRC3[31:0])
DEST[127:32] ← DEST[127:32]
DEST[MAX_VL-1:128] ← 0

```

**VFNSUB231SS DEST, SRC2, SRC3 (VEX encoded version)**

```

DEST[31:0] ← RoundFPControl_MXCSR(- (SRC2[31:0]*SRC3[31:0]) - DEST[31:0])
DEST[127:32] ← DEST[127:32]
DEST[MAX_VL-1:128] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VFNSUBxxxSS __m128 _mm_fnmsub_round_ss(__m128 a, __m128 b, __m128 c, int r);
VFNSUBxxxSS __m128 _mm_mask_fnmsub_ss(__m128 a, __mmask8 k, __m128 b, __m128 c);
VFNSUBxxxSS __m128 _mm_maskz_fnmsub_ss(__mmask8 k, __m128 a, __m128 b, __m128 c);
VFNSUBxxxSS __m128 _mm_mask3_fnmsub_ss(__m128 a, __m128 b, __m128 c, __mmask8 k);
VFNSUBxxxSS __m128 _mm_mask_fnmsub_round_ss(__m128 a, __mmask8 k, __m128 b, __m128 c, int r);
VFNSUBxxxSS __m128 _mm_maskz_fnmsub_round_ss(__mmask8 k, __m128 a, __m128 b, __m128 c, int r);
VFNSUBxxxSS __m128 _mm_mask3_fnmsub_round_ss(__m128 a, __m128 b, __m128 c, __mmask8 k, int r);
VFNSUBxxxSS __m128 _mm_fnmsub_ss (__m128 a, __m128 b, __m128 c);

```

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal

**Other Exceptions**

VEX-encoded instructions, see Exceptions Type 3.

EVEX-encoded instructions, see Exceptions Type E3.

## VFPCLASSPD—Tests Types Of a Packed Float64 Values

Opcode/ Instruction	Op/ En	64/32 bitMode Support	CPUID Feature Flag	Description
EVEX.128.66.0F3A.W1 66 /r ib VFPCLASSPD k2 {k1}, xmm2/m128/m64bcst, imm8	FV	V/V	AVX512VL AVX512DQ	Tests the input for the following categories: NaN, +0, -0, +Infinity, -Infinity, denormal, finite negative. The immediate field provides a mask bit for each of these category tests. The masked test results are OR-ed together to form a mask result.
EVEX.256.66.0F3A.W1 66 /r ib VFPCLASSPD k2 {k1}, ymm2/m256/m64bcst, imm8	FV	V/V	AVX512VL AVX512DQ	Tests the input for the following categories: NaN, +0, -0, +Infinity, -Infinity, denormal, finite negative. The immediate field provides a mask bit for each of these category tests. The masked test results are OR-ed together to form a mask result.
EVEX.512.66.0F3A.W1 66 /r ib VFPCLASSPD k2 {k1}, zmm2/m512/m64bcst, imm8	FV	V/V	AVX512DQ	Tests the input for the following categories: NaN, +0, -0, +Infinity, -Infinity, denormal, finite negative. The immediate field provides a mask bit for each of these category tests. The masked test results are OR-ed together to form a mask result.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
FV	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

The FPCLASSPD instruction checks the packed double precision 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-1:8/4/2] of the destination are cleared.

The classification categories specified by imm8 are shown in Figure 5-13. The classification test for each category is listed in Table 5-6.

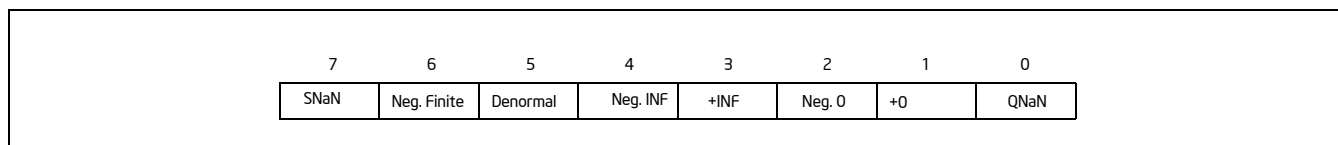


Figure 5-13. Imm8 Byte Specifier of Special Case FP Values for VFPCLASSPD/SD/PS/SS

Table 5-6. Classifier Operations for VFPCLASSPD/SD/PS/SS

Bits	Imm8[0]	Imm8[1]	Imm8[2]	Imm8[3]	Imm8[4]	Imm8[5]	Imm8[6]	Imm8[7]
Category	QNaN	PosZero	NegZero	PosINF	NegINF	Denormal	Negative	SNaN
Classifier	Checks for QNaN	Checks for +0	Checks for -0	Checks for +INF	Checks for -INF	Checks for Denormal	Checks for Negative finite	Checks for SNaN

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 64-bit memory location.

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

**Operation**

```
CheckFPClassDP (tsrc[63:0], imm8[7:0]){
```

```
    /* Start checking the source operand for special type */
    NegNum ← tsrc[63];
    IF (tsrc[62:52]=07FFh) Then ExpAllOnes ← 1; FI;
    IF (tsrc[62:52]=0h) Then ExpAllZeros ← 1;
    IF (ExpAllZeros AND MXCSR.DAZ) Then
        MantAllZeros ← 1;
    ELSIF (tsrc[51:0]=0h) Then
        MantAllZeros ← 1;
    FI;
    ZeroNumber ← ExpAllZeros AND MantAllZeros
    SignalingBit ← tsrc[51];

    sNaN_res ← ExpAllOnes AND NOT(MantAllZeros) AND NOT(SignalingBit); // sNaN
    qNaN_res ← ExpAllOnes AND NOT(MantAllZeros) AND SignalingBit; // qNaN
    Pzero_res ← NOT(NegNum) AND ExpAllZeros AND MantAllZeros;; // +0
    Nzero_res ← NegNum AND ExpAllZeros AND MantAllZeros;; // -0
    Plnf_res ← NOT(NegNum) AND ExpAllOnes AND MantAllZeros;; // +Inf
    Nlnf_res ← NegNum AND ExpAllOnes AND MantAllZeros;; // -Inf
    Denorm_res ← ExpAllZeros AND NOT(MantAllZeros); // denorm
    FinNeg_res ← NegNum AND NOT(ExpAllOnes) AND NOT(ZeroNumber); // -finite

    bResult = ( imm8[0] AND qNaN_res ) OR ( imm8[1] AND Pzero_res ) OR
              ( imm8[2] AND Nzero_res ) OR ( imm8[3] AND Plnf_res ) OR
              ( imm8[4] AND Nlnf_res ) OR ( imm8[5] AND Denorm_res ) OR
              ( imm8[6] AND FinNeg_res ) OR ( imm8[7] AND sNaN_res );
    Return bResult;
} /* end of CheckFPClassDP() */
```

**VFPCLASSPD (EVEX Encoded versions)**

(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) AND (SRC *is memory*)
                THEN
                    DEST[j] ← CheckFPClassDP(SRC1[63:0], imm8[7:0]);
                ELSE
                    DEST[j] ← CheckFPClassDP(SRC1[i+63:i], imm8[7:0]);
            FI;
        ELSE DEST[j] ← 0 ; zeroing-masking only
    FI;
ENDFOR
DEST[MAX_KL-1:KL] ← 0
```

**Intel C/C++ Compiler Intrinsic Equivalent**

```
VFPCLASSPD __mmask8 __mm512_fpclass_pd_mask( __m512d a, int c);
VFPCLASSPD __mmask8 __mm512_mask_fpclass_pd_mask( __mmask8 m, __m512d a, int c)
VFPCLASSPD __mmask8 __mm256_fpclass_pd_mask( __m256d a, int c)
VFPCLASSPD __mmask8 __mm256_mask_fpclass_pd_mask( __mmask8 m, __m256d a, int c)
VFPCLASSPD __mmask8 __mm_fpclass_pd_mask( __m128d a, int c)
```

VFPCASSPD \_\_mmask8 \_\_mm\_mask\_fpclass\_pd\_mask( \_\_mmask8 m, \_\_m128d a, int c)

### SIMD Floating-Point Exceptions

None

### Other Exceptions

See Exceptions Type E4

#UD If EVEX.vvvv != 1111B.

## VFPCCLASSPS—Tests Types Of a Packed Float32 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F3A.W0 66 /r ib VFPCCLASSPS k2 {k1}, xmm2/m128/m32bcst, imm8	FV	V/V	AVX512VL AVX512DQ	Tests the input for the following categories: NaN, +0, -0, +Infinity, -Infinity, denormal, finite negative. The immediate field provides a mask bit for each of these category tests. The masked test results are OR-ed together to form a mask result.
EVEX.256.66.0F3A.W0 66 /r ib VFPCCLASSPS k2 {k1}, ymm2/m256/m32bcst, imm8	FV	V/V	AVX512VL AVX512DQ	Tests the input for the following categories: NaN, +0, -0, +Infinity, -Infinity, denormal, finite negative. The immediate field provides a mask bit for each of these category tests. The masked test results are OR-ed together to form a mask result.
EVEX.512.66.0F3A.W0 66 /r ib VFPCCLASSPS k2 {k1}, zmm2/m512/m32bcst, imm8	FV	V/V	AVX512DQ	Tests the input for the following categories: NaN, +0, -0, +Infinity, -Infinity, denormal, finite negative. The immediate field provides a mask bit for each of these category tests. The masked test results are OR-ed together to form a mask result.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
FV	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

The FPCLASSPS instruction checks the packed single-precision 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-1:16/8/4] of the destination are cleared.

The classification categories specified by imm8 are shown in Figure 5-13. The classification test for each category is listed in Table 5-6.

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 32-bit memory location.

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

### Operation

CheckFPClassSP (tsrc[31:0], imm8[7:0]){

```
    /* Start checking the source operand for special type */
```

```
    NegNum ← tsrc[31];
```

```
    IF (tsrc[30:23]=0FFh) Then ExpAllOnes ← 1; FI;
```

```
    IF (tsrc[30:23]=0h) Then ExpAllZeros ← 1;
```

```
    IF (ExpAllZeros AND MXCSR.DAZ) Then
```

```
        MantAllZeros ← 1;
```

```
    ELSIF (tsrc[22:0]=0h) Then
```

```
        MantAllZeros ← 1;
```

```
    FI;
```

```
    ZeroNumber= ExpAllZeros AND MantAllZeros
```

```
    SignalingBit= tsrc[22];
```

```
    sNaN_res ← ExpAllOnes AND NOT(MantAllZeros) AND NOT(SignalingBit); // sNaN
```

```
    qNaN_res ← ExpAllOnes AND NOT(MantAllZeros) AND SignalingBit; // qNaN
```

```
    Pzero_res ← NOT(NegNum) AND ExpAllZeros AND MantAllZeros; // +0
```

```

Nzero_res ← NegNum AND ExpAllZeros AND MantAllZeros;; // -0
Plnf_res ← NOT(NegNum) AND ExpAllOnes AND MantAllZeros;; // +Inf
Nlnf_res ← NegNum AND ExpAllOnes AND MantAllZeros;; // -Inf
Denorm_res ← ExpAllZeros AND NOT(MantAllZeros);; // denorm
FinNeg_res ← NegNum AND NOT(ExpAllOnes) AND NOT(ZeroNumber);; // -finite

bResult = ( imm8[0] AND qNaN_res ) OR (imm8[1] AND Pzero_res ) OR
           ( imm8[2] AND Nzero_res ) OR ( imm8[3] AND Plnf_res ) OR
           ( imm8[4] AND Nlnf_res ) OR ( imm8[5] AND Denorm_res ) OR
           ( imm8[6] AND FinNeg_res ) OR ( imm8[7] AND sNaN_res );
Return bResult;
} /* end of CheckSPClassSP() */

```

### VFPCCLASSPS (EVEX encoded versions)

(KL, VL) = (4, 128), (8, 256), (16, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b == 1) AND (SRC *is memory*)
        THEN
          DEST[j] ← CheckFPClassDP(SRC1[31:0], imm8[7:0]);
        ELSE
          DEST[j] ← CheckFPClassDP(SRC1[i+31:i], imm8[7:0]);
        FI;
      ELSE DEST[j] ← 0 ; zeroing-masking only
    FI;
ENDFOR
DEST[MAX_KL-1:KL] ← 0

```

### Intel C/C++ Compiler Intrinsic Equivalent

```

VFPCCLASSPS __mmask16 __mm512_fpclass_ps_mask( __m512 a, int c);
VFPCCLASSPS __mmask16 __mm512_mask_fpclass_ps_mask( __mmask16 m, __m512 a, int c)
VFPCCLASSPS __mmask8 __mm256_fpclass_ps_mask( __m256 a, int c)
VFPCCLASSPS __mmask8 __mm256_mask_fpclass_ps_mask( __mmask8 m, __m256 a, int c)
VFPCCLASSPS __mmask8 __mm_fpclass_ps_mask( __m128 a, int c)
VFPCCLASSPS __mmask8 __mm_mask_fpclass_ps_mask( __mmask8 m, __m128 a, int c)

```

### SIMD Floating-Point Exceptions

None

### Other Exceptions

See Exceptions Type E4

#UD If EVEX.vvvv != 1111B.

## VFPCLASSSD—Tests Types Of a Scalar Float64 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LIG.66.0F3A.W1 67 /r ib VFPCLASSSD k2 {k1}, xmm2/m64, imm8	T1S	V/V	AVX512DQ	Tests the input for the following categories: NaN, +0, -0, +Infinity, -Infinity, denormal, finite negative. The immediate field provides a mask bit for each of these category tests. The masked test results are OR-ed together to form a mask result.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
T1S	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

The FPCCLASSSD instruction checks the low double precision floating point value in the source operand 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 is written to the low bit in a mask register k2 according to the writemask k1. Bits MAX\_KL-1: 1 of the destination are cleared.

The classification categories specified by imm8 are shown in Figure 5-13. The classification test for each category is listed in Table 5-6.

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

### Operation

CheckFPClassDP (tsrc[63:0], imm8[7:0]){

```

NegNum ← tsrc[63];
IF (tsrc[62:52]=07FFh) Then ExpAllOnes ← 1; FI;
IF (tsrc[62:52]=0h) Then ExpAllZeros ← 1;
IF (ExpAllZeros AND MXCSR.DAZ) Then
    MantAllZeros ← 1;
ELSIF (tsrc[51:0]=0h) Then
    MantAllZeros ← 1;
FI;
ZeroNumber ← ExpAllZeros AND MantAllZeros
SignalingBit ← tsrc[51];

sNaN_res ← ExpAllOnes AND NOT(MantAllZeros) AND NOT(SignalingBit); // sNaN
qNaN_res ← ExpAllOnes AND NOT(MantAllZeros) AND SignalingBit; // qNaN
Pzero_res ← NOT(NegNum) AND ExpAllZeros AND MantAllZeros; // +0
Nzero_res ← NegNum AND ExpAllZeros AND MantAllZeros; // -0
PInf_res ← NOT(NegNum) AND ExpAllOnes AND MantAllZeros; // +Inf
NInf_res ← NegNum AND ExpAllOnes AND MantAllZeros; // -Inf
Denorm_res ← ExpAllZeros AND NOT(MantAllZeros); // denorm
FinNeg_res ← NegNum AND NOT(ExpAllOnes) AND NOT(ZeroNumber); // -finite

bResult = ( imm8[0] AND qNaN_res ) OR ( imm8[1] AND Pzero_res ) OR
           ( imm8[2] AND Nzero_res ) OR ( imm8[3] AND PInf_res ) OR
           ( imm8[4] AND NInf_res ) OR ( imm8[5] AND Denorm_res ) OR
           ( imm8[6] AND FinNeg_res ) OR ( imm8[7] AND sNaN_res );
Return bResult;
} // * end of CheckFPClassDP() * //
```



**VFPCLASSSD (EVEX encoded version)**

```

IF k1[0] OR *no writemask*
  THEN DEST[0] ←
    CheckFPClassDP(SRC1[63:0], imm8[7:0])
  ELSE DEST[0] ← 0 ; zeroing-masking only
FI;
DEST[MAX_KL-1:1] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

VFPCLASSSD \_\_mmask8 \_mm\_fpclass\_sd\_mask( \_\_m128d a, int c)

VFPCLASSSD \_\_mmask8 \_mm\_mask\_fpclass\_sd\_mask( \_\_mmask8 m, \_\_m128d a, int c)

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type E6

#UD If EVEX.vvvv != 1111B.

## VFPCLASSSS—Tests Types Of a Scalar Float32 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LIG.66.0F3A.W0 67 /r VFPCLASSSS k2 {k1}, xmm2/m32, imm8	T1S	V/V	AVX512DQ	Tests the input for the following categories: NaN, +0, -0, +Infinity, -Infinity, denormal, finite negative. The immediate field provides a mask bit for each of these category tests. The masked test results are OR-ed together to form a mask result.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
T1S	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

The FPCCLASSSS instruction checks the low single-precision floating point value in the source operand 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 is written to the low bit in a mask register k2 according to the writemask k1. Bits MAX\_KL-1: 1 of the destination are cleared.

The classification categories specified by imm8 are shown in Figure 5-13. The classification test for each category is listed in Table 5-6.

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

### Operation

CheckFPClassSP (tsrc[31:0], imm8[7:0]){

```

/* Start checking the source operand for special type */
NegNum ← tsrc[31];
IF (tsrc[30:23]=0FFh) Then ExpAllOnes ← 1; FI;
IF (tsrc[30:23]=0h) Then ExpAllZeros ← 1;
IF (ExpAllZeros AND MXCSR.DAZ) Then
    MantAllZeros ← 1;
ELSIF (tsrc[22:0]=0h) Then
    MantAllZeros ← 1;
FI;
ZeroNumber= ExpAllZeros AND MantAllZeros
SignalingBit= tsrc[22];

sNaN_res ← ExpAllOnes AND NOT(MantAllZeros) AND NOT(SignalingBit); // sNaN
qNaN_res ← ExpAllOnes AND NOT(MantAllZeros) AND SignalingBit; // qNaN
Pzero_res ← NOT(NegNum) AND ExpAllZeros AND MantAllZeros; // +0
Nzero_res ← NegNum AND ExpAllZeros AND MantAllZeros; // -0
PInf_res ← NOT(NegNum) AND ExpAllOnes AND MantAllZeros; // +Inf
NInf_res ← NegNum AND ExpAllOnes AND MantAllZeros; // -Inf
Denorm_res ← ExpAllZeros AND NOT(MantAllZeros); // denorm
FinNeg_res ← NegNum AND NOT(ExpAllOnes) AND NOT(ZeroNumber); // -finite

bResult = ( imm8[0] AND qNaN_res ) OR ( imm8[1] AND Pzero_res ) OR
           ( imm8[2] AND Nzero_res ) OR ( imm8[3] AND PInf_res ) OR
           ( imm8[4] AND NInf_res ) OR ( imm8[5] AND Denorm_res ) OR
           ( imm8[6] AND FinNeg_res ) OR ( imm8[7] AND sNaN_res );
Return bResult;

```

```
} /* end of CheckSPClassSP() */
```

#### VFPCLASSSS (EVEX encoded version)

```
IF k1[0] OR *no writemask*
  THEN DEST[0] ←
    CheckFPClassSP(SRC1[31:0], imm8[7:0])
  ELSE DEST[0] ← 0 ; zeroing-masking only
FI;
DEST[MAX_KL-1:1] ← 0
```

#### Intel C/C++ Compiler Intrinsic Equivalent

```
VFPCLASSSS __mmask8 _mm_fpclass_ss_mask( __m128 a, int c)
VFPCLASSSS __mmask8 _mm_mask_fpclass_ss_mask( __mmask8 m, __m128 a, int c)
```

#### SIMD Floating-Point Exceptions

None

#### Other Exceptions

See Exceptions Type E6

#UD If EVEX.vvvv != 1111B.

## VGATHERDPS/VGATHERDPD—Gather Packed Single, Packed Double with Signed Dword

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 92 /vsib VGATHERDPS xmm1 {k1}, vm32x	T1S	V/V	AVX512VL AVX512F	Using signed dword indices, gather single-precision floating-point values from memory using k1 as completion mask.
EVEX.256.66.0F38.W0 92 /vsib VGATHERDPS ymm1 {k1}, vm32y	T1S	V/V	AVX512VL AVX512F	Using signed dword indices, gather single-precision floating-point values from memory using k1 as completion mask.
EVEX.512.66.0F38.W0 92 /vsib VGATHERDPS zmm1 {k1}, vm32z	T1S	V/V	AVX512F	Using signed dword indices, gather single-precision floating-point values from memory using k1 as completion mask.
EVEX.128.66.0F38.W1 92 /vsib VGATHERDPD xmm1 {k1}, vm32x	T1S	V/V	AVX512VL AVX512F	Using signed dword indices, gather float64 vector into float64 vector xmm1 using k1 as completion mask.
EVEX.256.66.0F38.W1 92 /vsib VGATHERDPD ymm1 {k1}, vm32x	T1S	V/V	AVX512VL AVX512F	Using signed dword indices, gather float64 vector into float64 vector ymm1 using k1 as completion mask.
EVEX.512.66.0F38.W1 92 /vsib VGATHERDPD zmm1 {k1}, vm32y	T1S	V/V	AVX512F	Using signed dword indices, gather float64 vector into float64 vector zmm1 using k1 as completion mask.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
T1S	ModRM:reg (w)	BaseReg (R): VSIB:base, VectorReg(R): VSIB:index	NA	NA

### Description

A set of single-precision/double-precision floating-point memory locations pointed by base address `BASE_ADDR` and index vector `V_INDEX` with scale `SCALE` are gathered. The result is written into a vector register. The elements are specified via the VSIB (i.e., the index register is a vector register, holding packed indices). Elements will only be loaded if their corresponding mask bit is one. If an element's mask bit is not set, the corresponding element of the destination register is left unchanged. The entire mask register will be set to zero by this instruction unless it triggers an exception.

This instruction can be suspended by an exception if at least one element is already gathered (i.e., if the exception is triggered by an element other than the right most one with its mask bit set). When this happens, the destination register and the mask register (k1) are partially updated; those elements that have been gathered are placed into the destination register and have their mask bits set to zero. If any traps or interrupts are pending from already gathered elements, they will be delivered in lieu of the exception; in this case, `EFLAG.RF` is set to one so an instruction breakpoint is not re-triggered when the instruction is continued.

If the data element size is less than the index element size, the higher part of the destination register and the mask register do not correspond to any elements being gathered. This instruction sets those higher parts to zero. It may update these unused elements to one or both of those registers even if the instruction triggers an exception, and even if the instruction triggers the exception before gathering any elements.

Note that:

- The values may be read from memory in any order. Memory ordering with other instructions follows the Intel-64 memory-ordering model.
- Faults are delivered in a right-to-left manner. That is, if a fault is triggered by an element and delivered, all elements closer to the LSB of the destination zmm will be completed (and non-faulting). Individual elements

closer to the MSB may or may not be completed. If a given element triggers multiple faults, they are delivered in the conventional order.

- Elements may be gathered in any order, but faults must be delivered in a right-to-left order; thus, elements to the left of a faulting one may be gathered before the fault is delivered. A given implementation of this instruction is repeatable - given the same input values and architectural state, the same set of elements to the left of the faulting one will be gathered.
- This instruction does not perform AC checks, and so will never deliver an AC fault.
- Not valid with 16-bit effective addresses. Will deliver a #UD fault.

Note that the presence of VSIB byte is enforced in this instruction. Hence, the instruction will #UD fault if ModRM.rm is different than 100b.

This instruction has special  $\text{disp8} * N$  and alignment rules. N is considered to be the size of a single vector element.

The scaled index may require more bits to represent than the address bits used by the processor (e.g., in 32-bit mode, if the scale is greater than one). In this case, the most significant bits beyond the number of address bits are ignored.

The instruction will #UD fault if the destination vector zmm1 is the same as index vector VINDEX. The instruction will #UD fault if the k0 mask register is specified.

### Operation

BASE\_ADDR stands for the memory operand base address (a GPR); may not exist

VINDEX stands for the memory operand vector of indices (a vector register)

SCALE stands for the memory operand scalar (1, 2, 4 or 8)

DISP is the optional 1, 2 or 4 byte displacement

#### VGATHERDPS (EVEX encoded version)

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

  i ← j \* 32

  IF k1[j]

    THEN DEST[i+31:i] ←

      MEM[BASE\_ADDR +

        SignExtend(VINDEX[i+31:i]) \* SCALE + DISP]

      k1[j] ← 0

    ELSE \*DEST[i+31:i] ← remains unchanged\*

  FI;

ENDFOR

k1[MAX\_KL-1:KL] ← 0

DEST[MAX\_VL-1:VL] ← 0

#### VGATHERDPD (EVEX encoded version)

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

  i ← j \* 64

  k ← j \* 32

  IF k1[j]

    THEN DEST[i+63:i] ← MEM[BASE\_ADDR +

      SignExtend(VINDEX[k+31:k]) \* SCALE + DISP]

    k1[j] ← 0

    ELSE \*DEST[i+63:i] ← remains unchanged\*

  FI;

ENDFOR

k1[MAX\_KL-1:KL] ← 0

DEST[MAX\_VL-1:VL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VGATHERDPD __m512d __mm512_i32gather_pd(__m256i vdx, void * base, int scale);
VGATHERDPD __m512d __mm512_mask_i32gather_pd(__m512d s, __mmask8 k, __m256i vdx, void * base, int scale);
VGATHERDPD __m256d __mm256_mask_i32gather_pd(__m256d s, __mmask8 k, __m128i vdx, void * base, int scale);
VGATHERDPD __m128d __mm_mask_i32gather_pd(__m128d s, __mmask8 k, __m128i vdx, void * base, int scale);
VGATHERDPS __m512 __mm512_i32gather_ps(__m512i vdx, void * base, int scale);
VGATHERDPS __m512 __mm512_mask_i32gather_ps(__m512 s, __mmask16 k, __m512i vdx, void * base, int scale);
VGATHERDPS __m256 __mm256_mask_i32gather_ps(__m256 s, __mmask8 k, __m256i vdx, void * base, int scale);
GATHERDPS __m128 __mm_mask_i32gather_ps(__m128 s, __mmask8 k, __m128i vdx, void * base, int scale);

```

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type E12.

## VGATHERPFODPS/VGATHERPFOQPS/VGATHERPFODPD/VGATHERPFOQPD—Sparse Prefetch Packed SP/DP Data Values with Signed Dword, Signed Qword Indices Using T0 Hint

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.512.66.0F38.W0 C6 /1 /vsib VGATHERPFODPS vm32z {k1}	T1S	V/V	AVX512PF	Using signed dword indices, prefetch sparse byte memory locations containing single-precision data using opmask k1 and T0 hint.
EVEX.512.66.0F38.W0 C7 /1 /vsib VGATHERPFOQPS vm64z {k1}	T1S	V/V	AVX512PF	Using signed qword indices, prefetch sparse byte memory locations containing single-precision data using opmask k1 and T0 hint.
EVEX.512.66.0F38.W1 C6 /1 /vsib VGATHERPFODPD vm32y {k1}	T1S	V/V	AVX512PF	Using signed dword indices, prefetch sparse byte memory locations containing double-precision data using opmask k1 and T0 hint.
EVEX.512.66.0F38.W1 C7 /1 /vsib VGATHERPFOQPD vm64z {k1}	T1S	V/V	AVX512PF	Using signed qword indices, prefetch sparse byte memory locations containing double-precision data using opmask k1 and T0 hint.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
T1S	BaseReg (R): VSIB:base, VectorReg(R): VSIB:index	NA	NA	NA

### Description

The instruction conditionally prefetches up to sixteen 32-bit or eight 64-bit integer byte data elements. The elements are specified via the VSIB (i.e., the index register is an zmm, holding packed indices). Elements will only be prefetched if their corresponding mask bit is one.

Lines prefetched are loaded into to a location in the cache hierarchy specified by a locality hint (T0):

- T0 (temporal data)—prefetch data into the first level cache.

[PS data] For dword indices, the instruction will prefetch sixteen memory locations. For qword indices, the instruction will prefetch eight values.

[PD data] For dword and qword indices, the instruction will prefetch eight memory locations.

Note that:

- (1) The prefetches may happen in any order (or not at all). The instruction is a hint.
- (2) The mask is left unchanged.
- (3) Not valid with 16-bit effective addresses. Will deliver a #UD fault.
- (4) No FP nor memory faults may be produced by this instruction.
- (5) Prefetches do not handle cache line splits
- (6) A #UD is signaled if the memory operand is encoded without the SIB byte.

### Operation

BASE\_ADDR stands for the memory operand base address (a GPR); may not exist

VINDEX stands for the memory operand vector of indices (a vector register)

SCALE stands for the memory operand scalar (1, 2, 4 or 8)

DISP is the optional 1, 2 or 4 byte displacement

PREFETCH(mem, Level, State) Prefetches a byte memory location pointed by 'mem' into the cache level specified by 'Level'; a request for exclusive/ownership is done if 'State' is 1. Note that the memory location ignore cache line splits. This operation is considered a hint for the processor and may be skipped depending on implementation.

**VGATHERPFODPS (EVEX encoded version)**

```
(KL, VL) = (16, 512)
FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j]
    Prefetch( [BASE_ADDR + SignExtend(VINDEX[j+31:i]) * SCALE + DISP], Level=0, RFO = 0)
  FI;
ENDFOR
```

**VGATHERPFODPD (EVEX encoded version)**

```
(KL, VL) = (8, 512)
FOR j ← 0 TO KL-1
  i ← j * 64
  k ← j * 32
  IF k1[j]
    Prefetch( [BASE_ADDR + SignExtend(VINDEX[k+31:k]) * SCALE + DISP], Level=0, RFO = 0)
  FI;
ENDFOR
```

**VGATHERPFOQPS (EVEX encoded version)**

```
(KL, VL) = (8, 256)
FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j]
    Prefetch( [BASE_ADDR + SignExtend(VINDEX[j+63:i]) * SCALE + DISP], Level=0, RFO = 0)
  FI;
ENDFOR
```

**VGATHERPFOQPD (EVEX encoded version)**

```
(KL, VL) = (8, 512)
FOR j ← 0 TO KL-1
  i ← j * 64
  k ← j * 64
  IF k1[j]
    Prefetch( [BASE_ADDR + SignExtend(VINDEX[k+63:k]) * SCALE + DISP], Level=0, RFO = 0)
  FI;
ENDFOR
```

**Intel C/C++ Compiler Intrinsic Equivalent**

```
VGATHERPFOQPD void __mm512_mask_prefetch_j32gather_pd(__m256i vdx, __mmask8 m, void * base, int scale, int hint);
VGATHERPFOQPS void __mm512_mask_prefetch_j32gather_ps(__m512i vdx, __mmask16 m, void * base, int scale, int hint);
VGATHERPFOQPD void __mm512_mask_prefetch_j64gather_pd(__m512i vdx, __mmask8 m, void * base, int scale, int hint);
VGATHERPFOQPS void __mm512_mask_prefetch_j64gather_ps(__m512i vdx, __mmask8 m, void * base, int scale, int hint);
```

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type E12NP.



## VGATHERPF1DPS/VGATHERPF1QPS/VGATHERPF1DPD/VGATHERPF1QPD—Sparse Prefetch Packed SP/DP Data Values with Signed Dword, Signed Qword Indices Using T1 Hint

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.512.66.0F38.W0 C6 /2 /vsib VGATHERPF1DPS vm32z {k1}	T1S	V/V	AVX512PF	Using signed dword indices, prefetch sparse byte memory locations containing single-precision data using opmask k1 and T1 hint.
EVEX.512.66.0F38.W0 C7 /2 /vsib VGATHERPF1QPS vm64z {k1}	T1S	V/V	AVX512PF	Using signed qword indices, prefetch sparse byte memory locations containing single-precision data using opmask k1 and T1 hint.
EVEX.512.66.0F38.W1 C6 /2 /vsib VGATHERPF1DPD vm32y {k1}	T1S	V/V	AVX512PF	Using signed dword indices, prefetch sparse byte memory locations containing double-precision data using opmask k1 and T1 hint.
EVEX.512.66.0F38.W1 C7 /2 /vsib VGATHERPF1QPD vm64z {k1}	T1S	V/V	AVX512PF	Using signed qword indices, prefetch sparse byte memory locations containing double-precision data using opmask k1 and T1 hint.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
T1S	BaseReg (R): VSIB:base, VectorReg(R): VSIB:index	NA	NA	NA

### Description

The instruction conditionally prefetches up to sixteen 32-bit or eight 64-bit integer byte data elements. The elements are specified via the VSIB (i.e., the index register is an zmm, holding packed indices). Elements will only be prefetched if their corresponding mask bit is one.

Lines prefetched are loaded into to a location in the cache hierarchy specified by a locality hint (T1):

- T1 (temporal data)—prefetch data into the second level cache.

[PS data] For dword indices, the instruction will prefetch sixteen memory locations. For qword indices, the instruction will prefetch eight values.

[PD data] For dword and qword indices, the instruction will prefetch eight memory locations.

Note that:

- (1) The prefetches may happen in any order (or not at all). The instruction is a hint.
- (2) The mask is left unchanged.
- (3) Not valid with 16-bit effective addresses. Will deliver a #UD fault.
- (4) No FP nor memory faults may be produced by this instruction.
- (5) Prefetches do not handle cache line splits
- (6) A #UD is signaled if the memory operand is encoded without the SIB byte.

### Operation

BASE\_ADDR stands for the memory operand base address (a GPR); may not exist

VINDEX stands for the memory operand vector of indices (a vector register)

SCALE stands for the memory operand scalar (1, 2, 4 or 8)

DISP is the optional 1, 2 or 4 byte displacement

PREFETCH(mem, Level, State) Prefetches a byte memory location pointed by 'mem' into the cache level specified by 'Level'; a request for exclusive/ownership is done if 'State' is 1. Note that the memory location ignore cache line splits. This operation is considered a hint for the processor and may be skipped depending on implementation.

**VGATHERPF1DPS (EVEX encoded version)**

```
(KL, VL) = (16, 512)
FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j]
    Prefetch( [BASE_ADDR + SignExtend(VINDEX[j+31:i]) * SCALE + DISP], Level=1, RFO = 0)
  FI;
ENDFOR
```

**VGATHERPF1DPD (EVEX encoded version)**

```
(KL, VL) = (8, 512)
FOR j ← 0 TO KL-1
  i ← j * 64
  k ← j * 32
  IF k1[j]
    Prefetch( [BASE_ADDR + SignExtend(VINDEX[k+31:k]) * SCALE + DISP], Level=1, RFO = 0)
  FI;
ENDFOR
```

**VGATHERPF1QPS (EVEX encoded version)**

```
(KL, VL) = (8, 256)
FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j]
    Prefetch( [BASE_ADDR + SignExtend(VINDEX[j+63:i]) * SCALE + DISP], Level=1, RFO = 0)
  FI;
ENDFOR
```

**VGATHERPF1QPD (EVEX encoded version)**

```
(KL, VL) = (8, 512)
FOR j ← 0 TO KL-1
  i ← j * 64
  k ← j * 64
  IF k1[j]
    Prefetch( [BASE_ADDR + SignExtend(VINDEX[k+63:k]) * SCALE + DISP], Level=1, RFO = 0)
  FI;
ENDFOR
```

**Intel C/C++ Compiler Intrinsic Equivalent**

```
VGATHERPF1DPD void __mm512_mask_prefetch_j32gather_pd(__m256i vdx, __mmask8 m, void * base, int scale, int hint);
VGATHERPF1DPS void __mm512_mask_prefetch_j32gather_ps(__m512i vdx, __mmask16 m, void * base, int scale, int hint);
VGATHERPF1QPD void __mm512_mask_prefetch_j64gather_pd(__m512i vdx, __mmask8 m, void * base, int scale, int hint);
VGATHERPF1QPS void __mm512_mask_prefetch_j64gather_ps(__m512i vdx, __mmask8 m, void * base, int scale, int hint);
```

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type E12NP.

## VGATHERQPS/VGATHERQPD—Gather Packed Single, Packed Double with Signed Qword Indices

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 93 /vsib VGATHERQPS xmm1 {k1}, vm64x	T1S	V/V	AVX512VL AVX512F	Using signed qword indices, gather single-precision floating-point values from memory using k1 as completion mask.
EVEX.256.66.0F38.W0 93 /vsib VGATHERQPS xmm1 {k1}, vm64y	T1S	V/V	AVX512VL AVX512F	Using signed qword indices, gather single-precision floating-point values from memory using k1 as completion mask.
EVEX.512.66.0F38.W0 93 /vsib VGATHERQPS ymm1 {k1}, vm64z	T1S	V/V	AVX512F	Using signed qword indices, gather single-precision floating-point values from memory using k1 as completion mask.
EVEX.128.66.0F38.W1 93 /vsib VGATHERQPD xmm1 {k1}, vm64x	T1S	V/V	AVX512VL AVX512F	Using signed qword indices, gather float64 vector into float64 vector xmm1 using k1 as completion mask.
EVEX.256.66.0F38.W1 93 /vsib VGATHERQPD ymm1 {k1}, vm64y	T1S	V/V	AVX512VL AVX512F	Using signed qword indices, gather float64 vector into float64 vector ymm1 using k1 as completion mask.
EVEX.512.66.0F38.W1 93 /vsib VGATHERQPD zmm1 {k1}, vm64z	T1S	V/V	AVX512F	Using signed qword indices, gather float64 vector into float64 vector zmm1 using k1 as completion mask.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
T1S	ModRM:reg (w)	BaseReg (R): VSIB:base, VectorReg(R): VSIB:index	NA	NA

### Description

A set of 8 single-precision/double-precision floating-point memory locations pointed by base address `BASE_ADDR` and index vector `V_INDEX` with scale `SCALE` are gathered. The result is written into vector a register. The elements are specified via the VSIB (i.e., the index register is a vector register, holding packed indices). Elements will only be loaded if their corresponding mask bit is one. If an element's mask bit is not set, the corresponding element of the destination register is left unchanged. The entire mask register will be set to zero by this instruction unless it triggers an exception.

This instruction can be suspended by an exception if at least one element is already gathered (i.e., if the exception is triggered by an element other than the rightmost one with its mask bit set). When this happens, the destination register and the mask register (k1) are partially updated; those elements that have been gathered are placed into the destination register and have their mask bits set to zero. If any traps or interrupts are pending from already gathered elements, they will be delivered in lieu of the exception; in this case, `EFLAG.RF` is set to one so an instruction breakpoint is not re-triggered when the instruction is continued.

If the data element size is less than the index element size, the higher part of the destination register and the mask register do not correspond to any elements being gathered. This instruction sets those higher parts to zero. It may update these unused elements to one or both of those registers even if the instruction triggers an exception, and even if the instruction triggers the exception before gathering any elements.

Note that:

- The values may be read from memory in any order. Memory ordering with other instructions follows the Intel-64 memory-ordering model.
- Faults are delivered in a right-to-left manner. That is, if a fault is triggered by an element and delivered, all elements closer to the LSB of the destination zmm will be completed (and non-faulting). Individual elements closer to the MSB may or may not be completed. If a given element triggers multiple faults, they are delivered in the conventional order.

- Elements may be gathered in any order, but faults must be delivered in a right-to-left order; thus, elements to the left of a faulting one may be gathered before the fault is delivered. A given implementation of this instruction is repeatable - given the same input values and architectural state, the same set of elements to the left of the faulting one will be gathered.
- This instruction does not perform AC checks, and so will never deliver an AC fault.
- Not valid with 16-bit effective addresses. Will deliver a #UD fault.

Note that the presence of VSIB byte is enforced in this instruction. Hence, the instruction will #UD fault if ModRM.rm is different than 100b.

This instruction has special  $\text{disp8} * N$  and alignment rules.  $N$  is considered to be the size of a single vector element. The scaled index may require more bits to represent than the address bits used by the processor (e.g., in 32-bit mode, if the scale is greater than one). In this case, the most significant bits beyond the number of address bits are ignored.

The instruction will #UD fault if the destination vector `zmm1` is the same as index vector `VINDEX`. The instruction will #UD fault if the `k0` mask register is specified.

### Operation

BASE\_ADDR stands for the memory operand base address (a GPR); may not exist

VINDEX stands for the memory operand vector of indices (a ZMM register)

SCALE stands for the memory operand scalar (1, 2, 4 or 8)

DISP is the optional 1, 2 or 4 byte displacement

#### VGATHERQPS (EVEX encoded version)

(KL, VL) = (2, 64), (4, 128), (8, 256)

FOR  $j \leftarrow 0$  TO  $KL-1$

```

    i ← j * 32
    k ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ←
            MEM[BASE_ADDR + (VINDEX[k+63:k]) * SCALE + DISP]
            k1[j] ← 0
        ELSE *DEST[i+31:i] ← remains unchanged*
    FI;
ENDFOR
k1[MAX_KL-1:KL] ← 0
DEST[MAX_VL-1:VL/2] ← 0

```

#### VGATHERQPD (EVEX encoded version)

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR  $j \leftarrow 0$  TO  $KL-1$

```

    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← MEM[BASE_ADDR + (VINDEX[i+63:i]) * SCALE + DISP]
            k1[j] ← 0
        ELSE *DEST[i+63:i] ← remains unchanged*
    FI;
ENDFOR
k1[MAX_KL-1:KL] ← 0
DEST[MAX_VL-1:VL] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VGATHERQPD __m512d __mm512_i64gather_pd(__m512i vdx, void * base, int scale);
VGATHERQPD __m512d __mm512_mask_i64gather_pd(__m512d s, __mmask8 k, __m512i vdx, void * base, int scale);
VGATHERQPD __m256d __mm256_mask_i64gather_pd(__m256d s, __mmask8 k, __m256i vdx, void * base, int scale);
VGATHERQPD __m128d __mm_mask_i64gather_pd(__m128d s, __mmask8 k, __m128i vdx, void * base, int scale);
VGATHERQPS __m256 __mm512_i64gather_ps(__m512i vdx, void * base, int scale);
VGATHERQPS __m256 __mm512_mask_i64gather_ps(__m256 s, __mmask16 k, __m512i vdx, void * base, int scale);
VGATHERQPS __m128 __mm256_mask_i64gather_ps(__m128 s, __mmask8 k, __m256i vdx, void * base, int scale);
VGATHERQPS __m128 __mm_mask_i64gather_ps(__m128 s, __mmask8 k, __m128i vdx, void * base, int scale);

```

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type E12.

## VPGATHERDD/VPGATHERDQ—Gather Packed Dword, Packed Qword with Signed Dword Indices

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 90 /vsib VPGATHERDD xmm1 {k1}, vm32x	T1S	V/V	AVX512VL AVX512F	Using signed dword indices, gather dword values from memory using writemask k1 for merging-masking.
EVEX.256.66.0F38.W0 90 /vsib VPGATHERDD ymm1 {k1}, vm32y	T1S	V/V	AVX512VL AVX512F	Using signed dword indices, gather dword values from memory using writemask k1 for merging-masking.
EVEX.512.66.0F38.W0 90 /vsib VPGATHERDD zmm1 {k1}, vm32z	T1S	V/V	AVX512F	Using signed dword indices, gather dword values from memory using writemask k1 for merging-masking.
EVEX.128.66.0F38.W1 90 /vsib VPGATHERDQ xmm1 {k1}, vm32x	T1S	V/V	AVX512VL AVX512F	Using signed dword indices, gather quadword values from memory using writemask k1 for merging-masking.
EVEX.256.66.0F38.W1 90 /vsib VPGATHERDQ ymm1 {k1}, vm32x	T1S	V/V	AVX512VL AVX512F	Using signed dword indices, gather quadword values from memory using writemask k1 for merging-masking.
EVEX.512.66.0F38.W1 90 /vsib VPGATHERDQ zmm1 {k1}, vm32y	T1S	V/V	AVX512F	Using signed dword indices, gather quadword values from memory using writemask k1 for merging-masking.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
T1S	ModRM:reg (w)	BaseReg (R): VSIB:base, VectorReg(R): VSIB:index	NA	NA

### Description

A set of 16 or 8 doubleword/quadword memory locations pointed to by base address `BASE_ADDR` and index vector `VINDEX` with scale `SCALE` are gathered. The result is written into vector `zmm1`. The elements are specified via the `VSIB` (i.e., the index register is a `zmm`, holding packed indices). Elements will only be loaded if their corresponding mask bit is one. If an element's mask bit is not set, the corresponding element of the destination register (`zmm1`) is left unchanged. The entire mask register will be set to zero by this instruction unless it triggers an exception.

This instruction can be suspended by an exception if at least one element is already gathered (i.e., if the exception is triggered by an element other than the rightmost one with its mask bit set). When this happens, the destination register and the mask register (`k1`) are partially updated; those elements that have been gathered are placed into the destination register and have their mask bits set to zero. If any traps or interrupts are pending from already gathered elements, they will be delivered in lieu of the exception; in this case, `EFLAG.RF` is set to one so an instruction breakpoint is not re-triggered when the instruction is continued.

If the data element size is less than the index element size, the higher part of the destination register and the mask register do not correspond to any elements being gathered. This instruction sets those higher parts to zero. It may update these unused elements to one or both of those registers even if the instruction triggers an exception, and even if the instruction triggers the exception before gathering any elements.

Note that:

- The values may be read from memory in any order. Memory ordering with other instructions follows the Intel-64 memory-ordering model.
- Faults are delivered in a right-to-left manner. That is, if a fault is triggered by an element and delivered, all elements closer to the LSB of the destination `zmm` will be completed (and non-faulting). Individual elements closer to the MSB may or may not be completed. If a given element triggers multiple faults, they are delivered in the conventional order.
- Elements may be gathered in any order, but faults must be delivered in a right-to-left order; thus, elements to the left of a faulting one may be gathered before the fault is delivered. A given implementation of this

instruction is repeatable - given the same input values and architectural state, the same set of elements to the left of the faulting one will be gathered.

- This instruction does not perform AC checks, and so will never deliver an AC fault.
- Not valid with 16-bit effective addresses. Will deliver a #UD fault.
- These instructions do not accept zeroing-masking since the 0 values in k1 are used to determine completion.

Note that the presence of VSIB byte is enforced in this instruction. Hence, the instruction will #UD fault if ModRM.rm is different than 100b.

This instruction has the same  $\text{disp8} * N$  and alignment rules as for scalar instructions (Tuple 1).

The instruction will #UD fault if the destination vector zmm1 is the same as index vector VINDEX. The instruction will #UD fault if the k0 mask register is specified.

The scaled index may require more bits to represent than the address bits used by the processor (e.g., in 32-bit mode, if the scale is greater than one). In this case, the most significant bits beyond the number of address bits are ignored.

### Operation

BASE\_ADDR stands for the memory operand base address (a GPR); may not exist

VINDEX stands for the memory operand vector of indices (a ZMM register)

SCALE stands for the memory operand scalar (1, 2, 4 or 8)

DISP is the optional 1, 2 or 4 byte displacement

#### VPGATHERDD (EVEX encoded version)

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

    i ← j \* 32

    IF k1[j]

        THEN DEST[i+31:i] ← MEM[BASE\_ADDR +  
                                  SignExtend(VINDEX[i+31:i]) \* SCALE + DISP], 1)

        k1[j] ← 0

    ELSE \*DEST[i+31:i] ← remains unchanged\*           ; Only merging masking is allowed

    FI;

ENDFOR

k1[MAX\_KL-1:KL] ← 0

DEST[MAX\_VL-1:VL] ← 0

#### VPGATHERDQ (EVEX encoded version)

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

    i ← j \* 64

    k ← j \* 32

    IF k1[j]

        THEN DEST[i+63:i] ←  
                                  MEM[BASE\_ADDR + SignExtend(VINDEX[k+31:k]) \* SCALE + DISP]

        k1[j] ← 0

    ELSE \*DEST[i+63:i] ← remains unchanged\*           ; Only merging masking is allowed

    FI;

ENDFOR

k1[MAX\_KL-1:KL] ← 0

DEST[MAX\_VL-1:VL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

```
VPGATHERDD __m512i __mm512_i32gather_epi32( __m512i vdx, void * base, int scale);  
VPGATHERDD __m512i __mm512_mask_i32gather_epi32(__m512i s, __mmask16 k, __m512i vdx, void * base, int scale);  
VPGATHERDD __m256i __mm256_mask_i32gather_epi32(__m256i s, __mmask8 k, __m256i vdx, void * base, int scale);  
VPGATHERDD __m128i __mm_mask_i32gather_epi32(__m128i s, __mmask8 k, __m128i vdx, void * base, int scale);  
VPGATHERDQ __m512i __mm512_i32logather_epi64( __m256i vdx, void * base, int scale);  
VPGATHERDQ __m512i __mm512_mask_i32logather_epi64(__m512i s, __mmask8 k, __m256i vdx, void * base, int scale);  
VPGATHERDQ __m256i __mm256_mask_i32logather_epi64(__m256i s, __mmask8 k, __m128i vdx, void * base, int scale);  
VPGATHERDQ __m128i __mm_mask_i32gather_epi64(__m128i s, __mmask8 k, __m128i vdx, void * base, int scale);
```

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type E12.



## VPGATHERQD/VPGATHERQQ—Gather Packed Dword, Packed Qword with Signed Qword Indices

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 91 /vsib VPGATHERQD xmm1 {k1}, vm64x	T1S	V/V	AVX512VL AVX512F	Using signed qword indices, gather dword values from memory using writemask k1 for merging-masking.
EVEX.256.66.0F38.W0 91 /vsib VPGATHERQD xmm1 {k1}, vm64y	T1S	V/V	AVX512VL AVX512F	Using signed qword indices, gather dword values from memory using writemask k1 for merging-masking.
EVEX.512.66.0F38.W0 91 /vsib VPGATHERQD ymm1 {k1}, vm64z	T1S	V/V	AVX512F	Using signed qword indices, gather dword values from memory using writemask k1 for merging-masking.
EVEX.128.66.0F38.W1 91 /vsib VPGATHERQQ xmm1 {k1}, vm64x	T1S	V/V	AVX512VL AVX512F	Using signed qword indices, gather quadword values from memory using writemask k1 for merging-masking.
EVEX.256.66.0F38.W1 91 /vsib VPGATHERQQ ymm1 {k1}, vm64y	T1S	V/V	AVX512VL AVX512F	Using signed qword indices, gather quadword values from memory using writemask k1 for merging-masking.
EVEX.512.66.0F38.W1 91 /vsib VPGATHERQQ zmm1 {k1}, vm64z	T1S	V/V	AVX512F	Using signed qword indices, gather quadword values from memory using writemask k1 for merging-masking.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
T1S	ModRM:reg (w)	BaseReg (R): VSIB:base, VectorReg(R): VSIB:index	NA	NA

### Description

A set of 8 doubleword/quadword memory locations pointed to by base address `BASE_ADDR` and index vector `VINDEX` with scale `SCALE` are gathered. The result is written into a vector register. The elements are specified via the `VSIB` (i.e., the index register is a vector register, holding packed indices). Elements will only be loaded if their corresponding mask bit is one. If an element's mask bit is not set, the corresponding element of the destination register is left unchanged. The entire mask register will be set to zero by this instruction unless it triggers an exception.

This instruction can be suspended by an exception if at least one element is already gathered (i.e., if the exception is triggered by an element other than the rightmost one with its mask bit set). When this happens, the destination register and the mask register (`k1`) are partially updated; those elements that have been gathered are placed into the destination register and have their mask bits set to zero. If any traps or interrupts are pending from already gathered elements, they will be delivered in lieu of the exception; in this case, `EFLAG.RF` is set to one so an instruction breakpoint is not re-triggered when the instruction is continued.

If the data element size is less than the index element size, the higher part of the destination register and the mask register do not correspond to any elements being gathered. This instruction sets those higher parts to zero. It may update these unused elements to one or both of those registers even if the instruction triggers an exception, and even if the instruction triggers the exception before gathering any elements.

Note that:

- The values may be read from memory in any order. Memory ordering with other instructions follows the Intel-64 memory-ordering model.
- Faults are delivered in a right-to-left manner. That is, if a fault is triggered by an element and delivered, all elements closer to the LSB of the destination `zmm` will be completed (and non-faulting). Individual elements closer to the MSB may or may not be completed. If a given element triggers multiple faults, they are delivered in the conventional order.

- Elements may be gathered in any order, but faults must be delivered in a right-to-left order; thus, elements to the left of a faulting one may be gathered before the fault is delivered. A given implementation of this instruction is repeatable - given the same input values and architectural state, the same set of elements to the left of the faulting one will be gathered.
- This instruction does not perform AC checks, and so will never deliver an AC fault.
- Not valid with 16-bit effective addresses. Will deliver a #UD fault.
- These instructions do not accept zeroing-masking since the 0 values in k1 are used to determine completion.

Note that the presence of VSIB byte is enforced in this instruction. Hence, the instruction will #UD fault if ModRM.rm is different than 100b.

This instruction has the same  $\text{disp}8*N$  and alignment rules as for scalar instructions (Tuple 1).

The instruction will #UD fault if the destination vector zmm1 is the same as index vector VINDEX. The instruction will #UD fault if the k0 mask register is specified.

The scaled index may require more bits to represent than the address bits used by the processor (e.g., in 32-bit mode, if the scale is greater than one). In this case, the most significant bits beyond the number of address bits are ignored.

### Operation

BASE\_ADDR stands for the memory operand base address (a GPR); may not exist

VINDEX stands for the memory operand vector of indices (a ZMM register)

SCALE stands for the memory operand scalar (1, 2, 4 or 8)

DISP is the optional 1, 2 or 4 byte displacement

#### VPGATHERQD (EVEX encoded version)

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

  i ← j \* 32

  k ← j \* 64

  IF k1[jj]

    THEN DEST[i+31:i] ← MEM[BASE\_ADDR + (VINDEX[k+63:k]) \* SCALE + DISP], 1)

    k1[jj] ← 0

    ELSE \*DEST[i+31:i] ← remains unchanged\* ; Only merging masking is allowed

  FI;

ENDFOR

k1[MAX\_KL-1:KL] ← 0

DEST[MAX\_VL-1:VL/2] ← 0

#### VPGATHERQQ (EVEX encoded version)

(KL, VL) = (2, 64), (4, 128), (8, 256)

FOR j ← 0 TO KL-1

  i ← j \* 64

  IF k1[jj]

    THEN DEST[i+63:i] ←

      MEM[BASE\_ADDR + (VINDEX[i+63:i]) \* SCALE + DISP]

    k1[jj] ← 0

    ELSE \*DEST[i+63:i] ← remains unchanged\* ; Only merging masking is allowed

  FI;

ENDFOR

k1[MAX\_KL-1:KL] ← 0

DEST[MAX\_VL-1:VL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VPGATHERQD __m256i _mm512_i64gather_epi32(__m512i vdx, void * base, int scale);
VPGATHERQD __m256i _mm512_mask_i64gather_epi32lo(__m256i s, __mmask8 k, __m512i vdx, void * base, int scale);
VPGATHERQD __m128i _mm256_mask_i64gather_epi32lo(__m128i s, __mmask8 k, __m256i vdx, void * base, int scale);
VPGATHERQD __m128i _mm_mask_i64gather_epi32(__m128i s, __mmask8 k, __m128i vdx, void * base, int scale);
VPGATHERQQ __m512i _mm512_i64gather_epi64(__m512i vdx, void * base, int scale);
VPGATHERQQ __m512i _mm512_mask_i64gather_epi64(__m512i s, __mmask8 k, __m512i vdx, void * base, int scale);
VPGATHERQQ __m256i _mm256_mask_i64gather_epi64(__m256i s, __mmask8 k, __m256i vdx, void * base, int scale);
VPGATHERQQ __m128i _mm_mask_i64gather_epi64(__m128i s, __mmask8 k, __m128i vdx, void * base, int scale);

```

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type E12.

## VGEXPPD—Convert Exponents of Packed DP FP Values to DP FP Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W1 42 /r VGEXPPD xmm1 {k1}{z}, xmm2/m128/m64bcst	FV	V/V	AVX512VL AVX512F	Convert the exponent of packed double-precision floating-point values in the source operand to DP FP results representing unbiased integer exponents and stores the results in the destination register.
EVEX.256.66.0F38.W1 42 /r VGEXPPD ymm1 {k1}{z}, ymm2/m256/m64bcst	FV	V/V	AVX512VL AVX512F	Convert the exponent of packed double-precision floating-point values in the source operand to DP FP results representing unbiased integer exponents and stores the results in the destination register.
EVEX.512.66.0F38.W1 42 /r VGEXPPD zmm1 {k1}{z}, zmm2/m512/m64bcst{sae}	FV	V/V	AVX512F	Convert the exponent of packed double-precision floating-point values in the source operand to DP FP results representing unbiased integer exponents and stores the results in the destination under writemask k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
FV	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Extracts the biased exponents from the normalized DP FP representation of each qword data element of the source operand (the second operand) as unbiased signed integer value, or convert the denormal representation of input data to unbiased negative integer values. Each integer value of the unbiased exponent is converted to double-precision FP value and written to the corresponding qword elements of the destination operand (the first operand) as DP 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 64-bit memory location.

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

Each GETEXP operation converts the exponent value into a FP number (permitting input value in denormal representation). Special cases of input values are listed in Table 5-7.

The formula is:

$$\text{GETEXP}(x) = \text{floor}(\log_2(|x|))$$

Notation **floor(x)** stands for the greatest integer not exceeding real number x.

Table 5-7. VGEXPPD/SD Special Cases

Input Operand	Result	Comments
src1 = NaN	QNaN(src1)	No Exceptions
$0 <  src1  < INF$	$\text{floor}(\log_2( src1 ))$	
$ src1  = +INF$	+INF	
$ src1  = 0$	-INF	

### Operation

NormalizeExpTinyDPFP(SRC[63:0])

```
{
    // jbit is the hidden integral bit of a FP number. In case of denormal number it has the value of ZERO.
    Src.jbit ← 0;
```

```

Dst.exp ← 1;
Dst.fraction ← SRC[51:0];
WHILE(Src.Jbit = 0)
{
    Src.Jbit ← Dst.fraction[51];    // Get the fraction MSB
    Dst.fraction ← Dst.fraction << 1;    // One bit shift left
    Dst.exp--;    // Decrement the exponent
}
Dst.fraction ← 0;    // zero out fraction bits
Dst.sign ← 1;    // Return negative sign
TMP[63:0] ← MXCSR.DAZ? 0 : (Dst.sign << 63) OR (Dst.exp << 52) OR (Dst.fraction);
Return (TMP[63:0]);
}

ConvertExpDPFP(SRC[63:0])
{
    Src.sign ← 0;    // Zero out sign bit
    Src.exp ← SRC[62:52];
    Src.fraction ← SRC[51:0];
    // Check for NaN
    IF (SRC = NaN)
    {
        IF (SRC = SNAN) SET IE;
        Return QNAN(SRC);
    }
    // Check for +INF
    IF (SRC = +INF) Return (SRC);

    // check if zero operand
    IF ((Src.exp = 0) AND ((Src.fraction = 0) OR (MXCSR.DAZ = 1))) Return (-INF);
}
ELSE    // check if denormal operand (notice that MXCSR.DAZ = 0)
{
    IF ((Src.exp = 0) AND (Src.fraction != 0))
    {
        TMP[63:0] ← NormalizeExpTinyDPFP(SRC[63:0]);    // Get Normalized Exponent
        Set #DE
    }
    ELSE    // exponent value is correct
    {
        Dst.fraction ← 0;    // zero out fraction bits
        TMP[63:0] ← (Src.sign << 63) OR (Src.exp << 52) OR (Src.fraction);
    }
    TMP ← SAR(TMP, 52);    // Shift Arithmetic Right
    TMP ← TMP - 1023;    // Subtract Bias
    Return CvtI2D(TMP);    // Convert INT to Double-Precision FP number
}
}

```

**VGETEXPPD (EVEX encoded versions)**

(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) AND (SRC *is memory*)
    THEN
      DEST[i+63:i] ←
      ConvertExpDPFP(SRC[63:0])
    ELSE
      DEST[i+63:i] ←
      ConvertExpDPFP(SRC[i+63:i])
    FI;
  ELSE
    IF *merging-masking*           ; merging-masking
      THEN *DEST[i+63:i] remains unchanged*
    ELSE                             ; zeroing-masking
      DEST[i+63:i] ← 0
    FI
  FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

### Intel C/C++ Compiler Intrinsic Equivalent

```

VGETEXPPD __m512d _mm512_getexp_pd(__m512d a);
VGETEXPPD __m512d _mm512_mask_getexp_pd(__m512d s, __mmask8 k, __m512d a);
VGETEXPPD __m512d _mm512_maskz_getexp_pd( __mmask8 k, __m512d a);
VGETEXPPD __m512d _mm512_getexp_round_pd(__m512d a, int sae);
VGETEXPPD __m512d _mm512_mask_getexp_round_pd(__m512d s, __mmask8 k, __m512d a, int sae);
VGETEXPPD __m512d _mm512_maskz_getexp_round_pd( __mmask8 k, __m512d a, int sae);
VGETEXPPD __m256d _mm256_getexp_pd(__m256d a);
VGETEXPPD __m256d _mm256_mask_getexp_pd(__m256d s, __mmask8 k, __m256d a);
VGETEXPPD __m256d _mm256_maskz_getexp_pd( __mmask8 k, __m256d a);
VGETEXPPD __m128d _mm_getexp_pd(__m128d a);
VGETEXPPD __m128d _mm_mask_getexp_pd(__m128d s, __mmask8 k, __m128d a);
VGETEXPPD __m128d _mm_maskz_getexp_pd( __mmask8 k, __m128d a);

```

### SIMD Floating-Point Exceptions

Invalid, Denormal

### Other Exceptions

See Exceptions Type E2.

#UD If EVEX.vvvv != 1111B.

## VGETEXPPS—Convert Exponents of Packed SP FP Values to SP FP Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 42 /r VGETEXPPS xmm1 {k1}{z}, xmm2/m128/m32bcst	FV	V/V	AVX512VL AVX512F	Convert the exponent of packed single-precision floating-point values in the source operand to SP FP results representing unbiased integer exponents and stores the results in the destination register.
EVEX.256.66.0F38.W0 42 /r VGETEXPPS ymm1 {k1}{z}, ymm2/m256/m32bcst	FV	V/V	AVX512VL AVX512F	Convert the exponent of packed single-precision floating-point values in the source operand to SP FP results representing unbiased integer exponents and stores the results in the destination register.
EVEX.512.66.0F38.W0 42 /r VGETEXPPS zmm1 {k1}{z}, zmm2/m512/m32bcst{sae}	FV	V/V	AVX512F	Convert the exponent of packed single-precision floating-point values in the source operand to SP FP results representing unbiased integer exponents and stores the results in the destination register.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
FV	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Extracts the biased exponents from the normalized SP FP representation of each dword element of the source operand (the second operand) as unbiased signed integer value, or convert the denormal representation of input data to unbiased negative integer values. Each integer value of the unbiased exponent is converted to single-precision FP value and written to the corresponding dword elements of the destination operand (the first operand) as SP 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 32-bit memory location.

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

Each GETEXP operation converts the exponent value into a FP number (permitting input value in denormal representation). Special cases of input values are listed in Table 5-8.

The formula is:

$$\text{GETEXP}(x) = \text{floor}(\log_2(|x|))$$

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

Software usage of VGETEXPxx and VGETMANTxx instructions generally involve a combination of GETEXP operation and GETMANT operation (see VGETMANTPD). Thus VGETEXPxx instruction do not require software to handle SIMD FP exceptions.

Table 5-8. VGETEXPPS/SS Special Cases

Input Operand	Result	Comments
src1 = NaN	QNaN(src1)	No Exceptions
$0 <  src1  < INF$	$\text{floor}(\log_2( src1 ))$	
$ src1  = +INF$	+INF	
$ src1  = 0$	-INF	

Figure 5-14 illustrates the VGETEXPPS functionality on input values with normalized representation.

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
	s	exp								Fraction																							
Src = 2 <sup>M</sup>	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
SAR Src, 23 = 080h	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
-Bias	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	1	
Tmp - Bias = 1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
Cvt_PI2PS(01h) = 2 <sup>0</sup>	0	0	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 5-14. VGETEXPPS Functionality On Normal Input values

**Operation**

```
NormalizeExpTinySPFP(SRC[31:0])
```

```
{
    // Jbit is the hidden integral bit of a FP number. In case of denormal number it has the value of ZERO.
    Src.Jbit ← 0;
    Dst.exp ← 1;
    Dst.fraction ← SRC[22:0];
    WHILE(Src.Jbit = 0)
    {
        Src.Jbit ← Dst.fraction[22]; // Get the fraction MSB
        Dst.fraction ← Dst.fraction << 1; // One bit shift left
        Dst.exp--; // Decrement the exponent
    }
    Dst.fraction ← 0; // zero out fraction bits
    Dst.sign ← 1; // Return negative sign
    TMP[31:0] ← MXCSR.DAZ? 0 : (Dst.sign << 31) OR (Dst.exp << 23) OR (Dst.fraction);
    Return (TMP[31:0]);
}
```

```
ConvertExpSPFP(SRC[31:0])
```

```
{
    Src.sign ← 0; // Zero out sign bit
    Src.exp ← SRC[30:23];
    Src.fraction ← SRC[22:0];
    // Check for NaN
    IF (SRC = NaN)
    {
        IF (SRC = SNAN) SET IE;
        Return QNAN(SRC);
    }
    // Check for +INF
    IF (SRC = +INF) Return (SRC);

    // check if zero operand
    IF ((Src.exp = 0) AND ((Src.fraction = 0) OR (MXCSR.DAZ = 1))) Return (-INF);
}
ELSE // check if denormal operand (notice that MXCSR.DAZ = 0)
{
    IF ((Src.exp = 0) AND (Src.fraction != 0))
    {
```



```

    TMP[31:0] ← NormalizeExpTinySPFP(SRC[31:0]);    // Get Normalized Exponent
    Set #DE
  }
  ELSE      // exponent value is correct
  {
    Dst.fraction ← 0;          // zero out fraction bits
    TMP[31:0] ← (Src.sign << 31) OR (Src.exp << 23) OR (Src.fraction);
  }
  TMP ← SAR(TMP, 23);        // Shift Arithmetic Right
  TMP ← TMP - 127;          // Subtract Bias
  Return CvtI2D(TMP);        // Convert INT to Single-Precision FP number
}
}

```

**VGETEXPPS (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\*

THEN

IF (EVEX.b = 1) AND (SRC \*is memory\*)

THEN

DEST[i+31:i] ←

ConvertExpSPFP(SRC[31:0])

ELSE

DEST[i+31:i] ←

ConvertExpSPFP(SRC[i+31:i])

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+31:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

VGETEXPPS \_\_m512 \_\_mm512\_getexp\_ps(\_\_m512 a);

VGETEXPPS \_\_m512 \_\_mm512\_mask\_getexp\_ps(\_\_m512 s, \_\_mmask16 k, \_\_m512 a);

VGETEXPPS \_\_m512 \_\_mm512\_maskz\_getexp\_ps(\_\_mmask16 k, \_\_m512 a);

VGETEXPPS \_\_m512 \_\_mm512\_getexp\_round\_ps(\_\_m512 a, int sae);

VGETEXPPS \_\_m512 \_\_mm512\_mask\_getexp\_round\_ps(\_\_m512 s, \_\_mmask16 k, \_\_m512 a, int sae);

VGETEXPPS \_\_m512 \_\_mm512\_maskz\_getexp\_round\_ps(\_\_mmask16 k, \_\_m512 a, int sae);

VGETEXPPS \_\_m256 \_\_mm256\_getexp\_ps(\_\_m256 a);

VGETEXPPS \_\_m256 \_\_mm256\_mask\_getexp\_ps(\_\_m256 s, \_\_mmask8 k, \_\_m256 a);

VGETEXPPS \_\_m256 \_\_mm256\_maskz\_getexp\_ps(\_\_mmask8 k, \_\_m256 a);

VGETEXPPS \_\_m128 \_\_mm\_getexp\_ps(\_\_m128 a);

VGETEXPPS \_\_m128 \_\_mm\_mask\_getexp\_ps(\_\_m128 s, \_\_mmask8 k, \_\_m128 a);

VGETEXPPS \_\_m128 \_\_mm\_maskz\_getexp\_ps(\_\_mmask8 k, \_\_m128 a);

**SIMD Floating-Point Exceptions**

Invalid, Denormal

**Other Exceptions**

See Exceptions Type E2.

#UD                      If EVEX.vvvv != 1111B.

## VGETEXPSD—Convert Exponents of Scalar DP FP Values to DP FP Value

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.LIG.66.0F38.W1 43 /r VGETEXPSD xmm1 {k1}{z}, xmm2, xmm3/m64{sae}	T1S	V/V	AVX512F	Convert the biased exponent (bits 62:52) of the low double-precision floating-point value in xmm3/m64 to a DP FP value representing unbiased integer exponent. Stores the result to the low 64-bit of xmm1 under the writemask k1 and merge with the other elements of xmm2.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
T1S	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Extracts the biased exponent from the normalized DP FP representation of the low qword data element of the source operand (the third operand) as unbiased signed integer value, or convert the denormal representation of input data to unbiased negative integer values. The integer value of the unbiased exponent is converted to double-precision FP value and written to the destination operand (the first operand) as DP FP numbers. Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand.

The destination must be a XMM register, the source operand can be a XMM register or a float64 memory location. The low quadword element of the destination operand is conditionally updated with writemask k1.

Each GETEXP operation converts the exponent value into a FP number (permitting input value in denormal representation). Special cases of input values are listed in Table 5-7.

The formula is:

$$\text{GETEXP}(x) = \text{floor}(\log_2(|x|))$$

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

### Operation

// NormalizeExpTinyDPFP(SRC[63:0]) is defined in the Operation section of VGETEXPPD

// ConvertExpDPFP(SRC[63:0]) is defined in the Operation section of VGETEXPPD

#### VGETEXPSD (EVEX encoded version)

```

IF k1[0] OR *no writemask*
  THEN DEST[63:0] ←
    ConvertExpDPFP(SRC2[63:0])
  ELSE
    IF *merging-masking*           ; merging-masking
      THEN *DEST[63:0] remains unchanged*
    ELSE                             ; zeroing-masking
      DEST[63:0] ← 0
  FI
FI;
DEST[127:64] ← SRC1[127:64]
DEST[MAX_VL-1:128] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

VGETEXPSD \_\_m128d\_mm\_getexp\_sd( \_\_m128d a, \_\_m128d b);

VGETEXPSD \_\_m128d\_mm\_mask\_getexp\_sd(\_\_m128d s, \_\_mmask8 k, \_\_m128d a, \_\_m128d b);

VGETEXPSD \_\_m128d\_mm\_maskz\_getexp\_sd( \_\_mmask8 k, \_\_m128d a, \_\_m128d b);

VGETEXPSD \_\_m128d\_mm\_getexp\_round\_sd( \_\_m128d a, \_\_m128d b, int sae);

VGETEXPSD \_\_m128d\_mm\_mask\_getexp\_round\_sd(\_\_m128d s, \_\_mmask8 k, \_\_m128d a, \_\_m128d b, int sae);

VGETEXPSD \_\_m128d\_mm\_maskz\_getexp\_round\_sd( \_\_mmask8 k, \_\_m128d a, \_\_m128d b, int sae);

**SIMD Floating-Point Exceptions**

Invalid, Denormal

**Other Exceptions**

See Exceptions Type E3.

## VGETEXPSS—Convert Exponents of Scalar SP FP Values to SP FP Value

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.LIG.66.0F38.W0 43 /r VGETEXPSS xmm1 {k1}{z}, xmm2, xmm3/m32{sae}	T1S	V/V	AVX512F	Convert the biased exponent (bits 30:23) of the low single-precision floating-point value in xmm3/m32 to a SP FP value representing unbiased integer exponent. Stores the result to xmm1 under the writemask k1 and merge with the other elements of xmm2.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
T1S	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Extracts the biased exponent from the normalized SP FP representation of the low doubleword data element of the source operand (the third operand) as unbiased signed integer value, or convert the denormal representation of input data to unbiased negative integer values. The integer value of the unbiased exponent is converted to single-precision FP value and written to the destination operand (the first operand) as SP FP numbers. Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand.

The destination must be a XMM register, the source operand can be a XMM register or a float32 memory location. The the low doubleword element of the destination operand is conditionally updated with writemask k1.

Each GETEXP operation converts the exponent value into a FP number (permitting input value in denormal representation). Special cases of input values are listed in Table 5-8.

The formula is:

$$\text{GETEXP}(x) = \text{floor}(\log_2(|x|))$$

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

Software usage of VGETEXPxx and VGETMANTxx instructions generally involve a combination of GETEXP operation and GETMANT operation (see VGETMANTPD). Thus VGETEXPxx instruction do not require software to handle SIMD FP exceptions.

### Operation

// NormalizeExpTinySPFP(SRC[31:0]) is defined in the Operation section of VGETEXPPS

// ConvertExpSPFP(SRC[31:0]) is defined in the Operation section of VGETEXPPS

#### VGETEXPSS (EVEX encoded version)

```

IF k1[0] OR *no writemask*
  THEN DEST[31:0] ←
    ConvertExpDPFP(SRC2[31:0])
  ELSE
    IF *merging-masking*           ; merging-masking
      THEN *DEST[31:0] remains unchanged*
    ELSE                             ; zeroing-masking
      DEST[31:0] ← 0
    FI
  FI;
ENDFOR
DEST[127:32] ← SRC1[127:32]
DEST[MAX_VL-1:128] ← 0

```

### Intel C/C++ Compiler Intrinsic Equivalent

VGETEXPSS \_\_m128 \_mm\_getexp\_ss( \_\_m128 a, \_\_m128 b);  
VGETEXPSS \_\_m128 \_mm\_mask\_getexp\_ss(\_\_m128 s, \_\_mmask8 k, \_\_m128 a, \_\_m128 b);  
VGETEXPSS \_\_m128 \_mm\_maskz\_getexp\_ss( \_\_mmask8 k, \_\_m128 a, \_\_m128 b);  
VGETEXPSS \_\_m128 \_mm\_getexp\_round\_ss( \_\_m128 a, \_\_m128 b, int sae);  
VGETEXPSS \_\_m128 \_mm\_mask\_getexp\_round\_ss(\_\_m128 s, \_\_mmask8 k, \_\_m128 a, \_\_m128 b, int sae);  
VGETEXPSS \_\_m128 \_mm\_maskz\_getexp\_round\_ss( \_\_mmask8 k, \_\_m128 a, \_\_m128 b, int sae);

### SIMD Floating-Point Exceptions

Invalid, Denormal

### Other Exceptions

See Exceptions Type E3.

## VGETMANTPD—Extract Float64 Vector of Normalized Mantissas from Float64 Vector

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F3A.W1 26 /r ib VGETMANTPD xmm1 {k1}{z}, xmm2/m128/m64bcst, imm8	FV	V/V	AVX512VL AVX512F	Get Normalized Mantissa from float64 vector xmm2/m128/m64bcst and store the result in xmm1, using imm8 for sign control and mantissa interval normalization, under writemask.
EVEX.256.66.0F3A.W1 26 /r ib VGETMANTPD ymm1 {k1}{z}, ymm2/m256/m64bcst, imm8	FV	V/V	AVX512VL AVX512F	Get Normalized Mantissa from float64 vector ymm2/m256/m64bcst and store the result in ymm1, using imm8 for sign control and mantissa interval normalization, under writemask.
EVEX.512.66.0F3A.W1 26 /r ib VGETMANTPD zmm1 {k1}{z}, zmm2/m512/m64bcst{sae}, imm8	FV	V/V	AVX512F	Get Normalized Mantissa from float64 vector zmm2/m512/m64bcst and store the result in zmm1, using imm8 for sign control and mantissa interval normalization, under writemask.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
FVI	ModRM:reg (w)	ModRM:r/m (r)	Imm8	NA

### Description

Convert double-precision floating values in the source operand (the second operand) to DP FP values with the mantissa normalization and sign control specified by the imm8 byte, see Figure 5-15. 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 64-bit memory location.

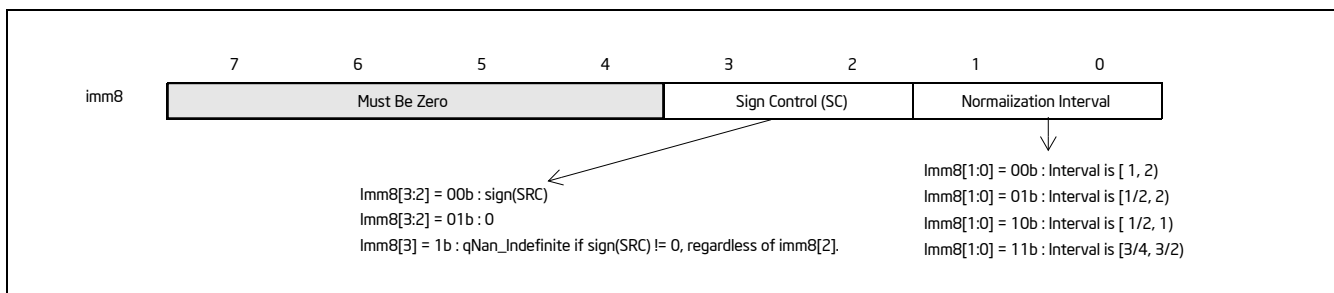


Figure 5-15. Imm8 Controls for VGETMANTPD/SD/PS/SS

For each input DP FP value  $x$ , The conversion operation is:

$$\text{GetMant}(x) = \pm 2^k |x.\text{significand}|$$

where:

$$1 \leq |x.\text{significand}| < 2$$

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  $interv \neq 0$  then  $k = -1$ , otherwise  $k = 0$ . The encoded value of  $imm8[1:0]$  and sign control are shown in Figure 5-15.

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

The `GetMant()` function follows Table 5-9 when dealing with floating-point special numbers.

This instruction is writemasked, so only those elements with the corresponding bit set in vector mask register  $k1$  are computed and stored into the destination. Elements in  $zmm1$  with the corresponding bit clear in  $k1$  retain their previous values.

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

**Table 5-9. GetMant() Special Float Values Behavior**

Input	Result	Exceptions / Comments
NaN	QNaN(SRC)	Ignore <i>interv</i> If (SRC = SNaN) then #IE
$+\infty$	1.0	Ignore <i>interv</i>
+0	1.0	Ignore <i>interv</i>
-0	IF (SC[0]) THEN +1.0 ELSE -1.0	Ignore <i>interv</i>
$-\infty$	IF (SC[1]) THEN {QNaN_Indefinite} ELSE { IF (SC[0]) THEN +1.0 ELSE -1.0	Ignore <i>interv</i> If (SC[1]) then #IE
negative	SC[1] ? QNaN_Indefinite : Getmant(SRC)	If (SC[1]) then #IE

### Operation

`GetNormalizeMantissaDP(SRC[63:0], SignCtrl[1:0], Interv[1:0])`

```
{
  // Extracting the SRC sign, exponent and mantissa fields
  Dst.sign ← SignCtrl[0] ? 0 : Src[63];           // Get sign bit
  Dst.exp ← SRC[62:52]; // Get original exponent value
  Dst.fraction ← SRC[51:0]; // Get original fraction value
  ZeroOperand ← (Dst.exp = 0) AND (Dst.fraction = 0);
  DenormOperand ← (Dst.exp = 0h) AND (Dst.fraction != 0);
  InfiniteOperand ← (Dst.exp = 07FFh) AND (Dst.fraction = 0);
  NaNOperand ← (Dst.exp = 07FFh) AND (Dst.fraction != 0);
  // Check for NAN operand
  IF (NaNOperand)
  {
    IF (SRC = SNaN) {Set #IE;}
    Return QNaN(SRC);
  }
  // Check for Zero and Infinite operands
  IF ((ZeroOperand) OR (InfiniteOperand))
  {
    Dst.exp ← 03FFh; // Override exponent with BIAS
    Return ((Dst.sign << 63) | (Dst.exp << 52) | (Dst.fraction));
  }
  // Check for negative operand (including -0.0)
  IF ((Src[63] = 1) AND SignCtrl[1])
  {
    Set #IE;
    Return QNaN_Indefinite;
  }
}
```



```

// Checking for denormal operands
IF (DenormOperand)
{
  IF (MXCSR.DAZ=1) Dst.fraction ← 0; // Zero out fraction
  ELSE
  {
    // Jbit is the hidden integral bit. Zero in case of denormal operand.
    Src.Jbit ← 0; // Zero Src Jbit
    Dst.exp ← 03FFh; // Override exponent with BIAS
    WHILE (Src.Jbit = 0) { // normalize mantissa
      Src.Jbit ← Dst.fraction[51]; // Get the fraction MSB
      Dst.fraction ← (Dst.fraction << 1); // Start normalizing the mantissa
      Dst.exp--; // Adjust the exponent
    }
    SET #DE; // Set DE bit
  }
}
// At this point, Dst.fraction is normalized.
// Checking for exponent response
Unbiased.exp ← Dst.exp - 03FFh; // subtract the bias from exponent
IsOddExp ← Unbiased.exp[0]; // recognized unbiased ODD exponent
SignalingBit ← Dst.fraction[51];
CASE (interv[1:0])
  00: Dst.exp ← 03FFh; // This is the bias
  01: Dst.exp ← (IsOddExp) ? 03FEh : 03FFh; // either bias-1, or bias
  10: Dst.exp ← 03FEh; // bias-1
  11: Dst.exp ← (SignalingBit) ? 03FEh : 03FFh; // either bias-1, or bias
ESCA
// At this point Dst.exp has the correct result. Form the final destination
DEST[63:0] ← (Dst.sign << 63) OR (Dst.exp << 52) OR (Dst.fraction);
Return (DEST);
}

```

```

SignCtrl[1:0] ← IMM8[3:2];
Interv[1:0] ← IMM8[1:0];

```

#### VGETMANTPD (EVEX encoded versions)

(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) AND (SRC \*is memory\*)

THEN

DEST[i+63:i] ← GetNormalizedMantissaDP(SRC[63:0], sc, interv)

ELSE

DEST[i+63:i] ← GetNormalizedMantissaDP(SRC[i+63:i], sc, interv)

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VGETMANTPD __m512d __mm512_getmant_pd( __m512d a, enum intv, enum sgn);
VGETMANTPD __m512d __mm512_mask_getmant_pd(__m512d s, __mmask8 k, __m512d a, enum intv, enum sgn);
VGETMANTPD __m512d __mm512_maskz_getmant_pd( __mmask8 k, __m512d a, enum intv, enum sgn);
VGETMANTPD __m512d __mm512_getmant_round_pd( __m512d a, enum intv, enum sgn, int r);
VGETMANTPD __m512d __mm512_mask_getmant_round_pd(__m512d s, __mmask8 k, __m512d a, enum intv, enum sgn, int r);
VGETMANTPD __m512d __mm512_maskz_getmant_round_pd( __mmask8 k, __m512d a, enum intv, enum sgn, int r);
VGETMANTPD __m256d __mm256_getmant_pd( __m256d a, enum intv, enum sgn);
VGETMANTPD __m256d __mm256_mask_getmant_pd(__m256d s, __mmask8 k, __m256d a, enum intv, enum sgn);
VGETMANTPD __m256d __mm256_maskz_getmant_pd( __mmask8 k, __m256d a, enum intv, enum sgn);
VGETMANTPD __m128d __mm_getmant_pd( __m128d a, enum intv, enum sgn);
VGETMANTPD __m128d __mm_mask_getmant_pd(__m128d s, __mmask8 k, __m128d a, enum intv, enum sgn);
VGETMANTPD __m128d __mm_maskz_getmant_pd( __mmask8 k, __m128d a, enum intv, enum sgn);

```

**SIMD Floating-Point Exceptions**

Denormal, Invalid

**Other Exceptions**

See Exceptions Type E2.

#UD                    If EVEX.vvvv != 1111B.

## VGETMANTPS—Extract Float32 Vector of Normalized Mantissas from Float32 Vector

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F3A.W0 26 /r ib VGETMANTPS xmm1 {k1}{z}, xmm2/m128/m32bcst, imm8	FV	V/V	AVX512VL AVX512F	Get normalized mantissa from float32 vector xmm2/m128/m32bcst and store the result in xmm1, using imm8 for sign control and mantissa interval normalization, under writemask.
EVEX.256.66.0F3A.W0 26 /r ib VGETMANTPS ymm1 {k1}{z}, ymm2/m256/m32bcst, imm8	FV	V/V	AVX512VL AVX512F	Get normalized mantissa from float32 vector ymm2/m256/m32bcst and store the result in ymm1, using imm8 for sign control and mantissa interval normalization, under writemask.
EVEX.512.66.0F3A.W0 26 /r ib VGETMANTPS zmm1 {k1}{z}, zmm2/m512/m32bcst{sae}, imm8	FV	V/V	AVX512F	Get normalized mantissa from float32 vector zmm2/m512/m32bcst and store the result in zmm1, using imm8 for sign control and mantissa interval normalization, under writemask.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
FVI	ModRM:reg (w)	ModRM:r/m (r)	Imm8	NA

### Description

Convert single-precision floating values in the source operand (the second operand) to SP FP values with the mantissa normalization and sign control specified by the imm8 byte, see Figure 5-15. 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 32-bit memory location.

For each input SP FP value  $x$ , The conversion operation is:

$$\text{GetMant}(x) = \pm 2^k |x.\text{significand}|$$

where:

$$1 \leq |x.\text{significand}| < 2$$

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  $K = 0$ . The encoded value of imm8[1:0] and sign control are shown in Figure 5-15.

Each converted SP 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.

The GetMant() function follows Table 5-9 when dealing with floating-point special numbers.

This instruction is writemasked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into the destination. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values.

Note: EVEX.vvvv is reserved and must be 1111b, VEX.L must be 0; otherwise instructions will #UD.

**Operation**

```

GetNormalizeMantissaSP(SRC[31:0], SignCtrl[1:0], Interv[1:0])
{
    // Extracting the SRC sign, exponent and mantissa fields
    Dst.sign ← SignCtrl[0] ? 0 : Src[31];           // Get sign bit
    Dst.exp ← SRC[30:23]; // Get original exponent value
    Dst.fraction ← SRC[22:0]; // Get original fraction value
    ZeroOperand ← (Dst.exp = 0) AND (Dst.fraction = 0);
    DenormOperand ← (Dst.exp = 0h) AND (Dst.fraction != 0);
    InfiniteOperand ← (Dst.exp = 0FFh) AND (Dst.fraction = 0);
    NaNOperand ← (Dst.exp = 0FFh) AND (Dst.fraction != 0);
    // Check for NAN operand
    IF (NaNOperand)
    {
        IF (SRC = SNaN) {Set #IE;}
        Return QNaN(SRC);
    }
    // Check for Zero and Infinite operands
    IF ((ZeroOperand) OR (InfiniteOperand))
    {
        Dst.exp ← 07Fh; // Override exponent with BIAS
        Return ((Dst.sign << 31) | (Dst.exp << 23) | (Dst.fraction));
    }
    // Check for negative operand (including -0.0)
    IF ((Src[31] = 1) AND SignCtrl[1])
    {
        Set #IE;
        Return QNaN_Indefinite;
    }
    // Checking for denormal operands
    IF (DenormOperand)
    {
        IF (MXCSR.DAZ=1) Dst.fraction ← 0; // Zero out fraction
        ELSE
        {
            // Jbit is the hidden integral bit. Zero in case of denormal operand.
            Src.Jbit ← 0; // Zero Src Jbit
            Dst.exp ← 07Fh; // Override exponent with BIAS
            WHILE (Src.Jbit = 0) { // normalize mantissa
                Src.Jbit ← Dst.fraction[22]; // Get the fraction MSB
                Dst.fraction ← (Dst.fraction << 1); // Start normalizing the mantissa
                Dst.exp--; // Adjust the exponent
            }
            SET #DE; // Set DE bit
        }
    }
    // At this point, Dst.fraction is normalized.
    // Checking for exponent response
    Unbiased.exp ← Dst.exp - 07Fh; // subtract the bias from exponent
    IsOddExp ← Unbiased.exp[0]; // recognized unbiased ODD exponent
    SignalingBit ← Dst.fraction[22];
    CASE (interv[1:0])
        00: Dst.exp ← 07Fh; // This is the bias
        01: Dst.exp ← (IsOddExp) ? 07Eh : 07Fh; // either bias-1, or bias
        10: Dst.exp ← 07Eh; // bias-1
        11: Dst.exp ← (SignalingBit) ? 07Eh : 07Fh; // either bias-1, or bias
    ESCA

    // Form the final destination
    DEST[31:0] ← (Dst.sign << 31) OR (Dst.exp << 23) OR (Dst.fraction);
}

```

```

    Return (DEST);
}

```

```

SignCtrl[1:0] ← IMM8[3:2];
Interv[1:0] ← IMM8[1:0];

```

#### VGETMANTPS (EVEX encoded versions)

(KL, VL) = (4, 128), (8, 256), (16, 512)

```

FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC *is memory*)
                THEN
                    DEST[i+31:i] ← GetNormalizedMantissaSP(SRC[31:0], sc, interv)
                ELSE
                    DEST[i+31:i] ← GetNormalizedMantissaSP(SRC[i+31:i], sc, interv)
            FI;
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+31:i] ← 0
            FI
        FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

#### Intel C/C++ Compiler Intrinsic Equivalent

```

VGETMANTPS __m512 __mm512_getmant_ps( __m512 a, enum intv, enum sgn);
VGETMANTPS __m512 __mm512_mask_getmant_ps(__m512 s, __mmask16 k, __m512 a, enum intv, enum sgn);
VGETMANTPS __m512 __mm512_maskz_getmant_ps(__mmask16 k, __m512 a, enum intv, enum sgn);
VGETMANTPS __m512 __mm512_getmant_round_ps( __m512 a, enum intv, enum sgn, int r);
VGETMANTPS __m512 __mm512_mask_getmant_round_ps(__m512 s, __mmask16 k, __m512 a, enum intv, enum sgn, int r);
VGETMANTPS __m512 __mm512_maskz_getmant_round_ps(__mmask16 k, __m512 a, enum intv, enum sgn, int r);
VGETMANTPS __m256 __mm256_getmant_ps( __m256 a, enum intv, enum sgn);
VGETMANTPS __m256 __mm256_mask_getmant_ps(__m256 s, __mmask8 k, __m256 a, enum intv, enum sgn);
VGETMANTPS __m256 __mm256_maskz_getmant_ps( __mmask8 k, __m256 a, enum intv, enum sgn);
VGETMANTPS __m128 __mm_getmant_ps( __m128 a, enum intv, enum sgn);
VGETMANTPS __m128 __mm_mask_getmant_ps(__m128 s, __mmask8 k, __m128 a, enum intv, enum sgn);
VGETMANTPS __m128 __mm_maskz_getmant_ps( __mmask8 k, __m128 a, enum intv, enum sgn);

```

#### SIMD Floating-Point Exceptions

Denormal, Invalid

#### Other Exceptions

See Exceptions Type E2.

#UD If EVEX.vvvv != 1111B.

## VGETMANTSD—Extract Float64 of Normalized Mantissas from Float64 Scalar

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.LIG.66.0F3A.W1 27 /r ib VGETMANTSD xmm1 {k1}{z}, xmm2, xmm3/m64{sae}, imm8	T1S	V/V	AVX512F	Extract the normalized mantissa of the low float64 element in xmm3/m64 using <i>imm8</i> for sign control and mantissa interval normalization. Store the mantissa to xmm1 under the writemask k1 and merge with the other elements of xmm2.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
T1S	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Convert the double-precision floating values in the low quadword element of the second source operand (the third operand) to DP FP value with the mantissa normalization and sign control specified by the *imm8* byte, see Figure 5-15. The converted result is written to the low quadword element of the destination operand (the first operand) using writemask k1. Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand. 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 conversion operation is:

$$\text{GetMant}(x) = \pm 2^k |x.\text{significand}|$$

where:

$$1 \leq |x.\text{significand}| < 2$$

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. The encoded value of *imm8*[1:0] and sign control are shown in Figure 5-15.

The converted DP 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*.

The *GetMant()* function follows Table 5-9 when dealing with floating-point special numbers.

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

### Operation

// *GetNormalizeMantissaDP*(SRC[63:0], SignCtrl[1:0], Interv[1:0]) is defined in the operation section of *VGETMANTPD*

```
SignCtrl[1:0] ← IMM8[3:2];
Interv[1:0] ← IMM8[1:0];
```

### VGETMANTSD (EVEX encoded version)

```
IF k1[0] OR *no writemask*
  THEN DEST[63:0] ←
    GetNormalizedMantissaDP(SRC2[63:0], sc, interv)
ELSE
  IF *merging-masking* ; merging-masking
```

```

        THEN *DEST[63:0] remains unchanged*
        ELSE                                ; zeroing-masking
            DEST[63:0] ← 0
    FI
FI;
DEST[127:64] ← SRC1[127:64]
DEST[MAX_VL-1:128] ← 0

```

### Intel C/C++ Compiler Intrinsic Equivalent

```

VGETMANTSD __m128d _mm_getmant_sd(__m128d a, __m128 b, enum intv, enum sgn);
VGETMANTSD __m128d _mm_mask_getmant_sd(__m128d s, __mmask8 k, __m128d a, __m128d b, enum intv, enum sgn);
VGETMANTSD __m128d _mm_maskz_getmant_sd(__mmask8 k, __m128 a, __m128d b, enum intv, enum sgn);
VGETMANTSD __m128d _mm_getmant_round_sd(__m128d a, __m128 b, enum intv, enum sgn, int r);
VGETMANTSD __m128d _mm_mask_getmant_round_sd(__m128d s, __mmask8 k, __m128d a, __m128d b, enum intv, enum sgn, int r);
VGETMANTSD __m128d _mm_maskz_getmant_round_sd(__mmask8 k, __m128d a, __m128d b, enum intv, enum sgn, int r);

```

### SIMD Floating-Point Exceptions

Denormal, Invalid

### Other Exceptions

See Exceptions Type E3.

**VGETMANTSS—Extract Float32 Vector of Normalized Mantissa from Float32 Vector**

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.LIG.66.0F3A.W0 27 /r ib VGETMANTSS xmm1 {k1}{z}, xmm2, xmm3/m32{sae}, imm8	T1S	V/V	AVX512F	Extract the normalized mantissa from the low float32 element of xmm3/m32 using imm8 for sign control and mantissa interval normalization, store the mantissa to xmm1 under the writemask k1 and merge with the other elements of xmm2.

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
T1S	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

**Description**

Convert the single-precision floating values in the low doubleword element of the second source operand (the third operand) to SP FP value with the mantissa normalization and sign control specified by the imm8 byte, see Figure 5-15. The converted result is written to the low doubleword element of the destination operand (the first operand) using writemask k1. Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand. 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 conversion operation is:

$$GetMant(x) = \pm 2^k |x.significand|$$

where:

$$1 \leq |x.significand| < 2$$

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. The encoded value of imm8[1:0] and sign control are shown in Figure 5-15.

The converted SP 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.

The GetMant() function follows Table 5-9 when dealing with floating-point special numbers.

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

**Operation**

// GetNormalizeMantissaSP(SRC[31:0], SignCtrl[1:0], Interv[1:0]) is defined in the operation section of VGETMANTPD

SignCtrl[1:0] ← IMM8[3:2];

Interv[1:0] ← IMM8[1:0];

**VGETMANTSS (EVEX encoded version)**

IF k1[0] OR \*no writemask\*

THEN DEST[31:0] ←

GetNormalizedMantissaSP(SRC2[31:0], sc, interv)

ELSE

IF \*merging-masking\* ; merging-masking



```

        THEN *DEST[31:0] remains unchanged*
        ELSE                                ; zeroing-masking
            DEST[31:0] ← 0
    FI
FI;
DEST[127:32] ← SRC1[127:64]
DEST[MAX_VL-1:128] ← 0

```

### Intel C/C++ Compiler Intrinsic Equivalent

```

VGETMANTSS __m128 _mm_getmant_ss( __m128 a, __m128 b, enum intv, enum sgn);
VGETMANTSS __m128 _mm_mask_getmant_ss(__m128 s, __mmask8 k, __m128 a, __m128 b, enum intv, enum sgn);
VGETMANTSS __m128 _mm_maskz_getmant_ss( __mmask8 k, __m128 a, __m128 b, enum intv, enum sgn);
VGETMANTSS __m128 _mm_getmant_round_ss(__m128 a, __m128 b, enum intv, enum sgn, int r);
VGETMANTSS __m128 _mm_mask_getmant_round_ss(__m128 s, __mmask8 k, __m128 a, __m128 b, enum intv, enum sgn, int r);
VGETMANTSS __m128 _mm_maskz_getmant_round_ss( __mmask8 k, __m128 a, __m128 b, enum intv, enum sgn, int r);

```

### SIMD Floating-Point Exceptions

Denormal, Invalid

### Other Exceptions

See Exceptions Type E3.

## VINSERTF128/VINSERTF32x4/VINSERTF64x2/VINSERTF32x8/VINSERTF64x4—Insert Packed Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.NDS.256.66.0F3A.W0 18 /r ib VINSERTF128 ymm1, ymm2, xmm3/m128, imm8	RVM1	V/V	AVX	Insert 128 bits of packed floating-point values from xmm3/m128 and the remaining values from ymm2 into ymm1.
EVEX.NDS.256.66.0F3A.W0 18 /r ib VINSERTF32X4 ymm1 {k1}{z}, ymm2, xmm3/m128, imm8	T4	V/V	AVX512VL AVX512F	Insert 128 bits of packed single-precision floating-point values from xmm3/m128 and the remaining values from ymm2 into ymm1 under writemask k1.
EVEX.NDS.512.66.0F3A.W0 18 /r ib VINSERTF32X4 zmm1 {k1}{z}, zmm2, xmm3/m128, imm8	T4	V/V	AVX512F	Insert 128 bits of packed single-precision floating-point values from xmm3/m128 and the remaining values from zmm2 into zmm1 under writemask k1.
EVEX.NDS.256.66.0F3A.W1 18 /r ib VINSERTF64X2 ymm1 {k1}{z}, ymm2, xmm3/m128, imm8	T2	V/V	AVX512VL AVX512DQ	Insert 128 bits of packed double-precision floating-point values from xmm3/m128 and the remaining values from ymm2 into ymm1 under writemask k1.
EVEX.NDS.512.66.0F3A.W1 18 /r ib VINSERTF64X2 zmm1 {k1}{z}, zmm2, xmm3/m128, imm8	T2	V/V	AVX512DQ	Insert 128 bits of packed double-precision floating-point values from xmm3/m128 and the remaining values from zmm2 into zmm1 under writemask k1.
EVEX.NDS.512.66.0F3A.W0 1A /r ib VINSERTF32X8 zmm1 {k1}{z}, zmm2, ymm3/m256, imm8	T8	V/V	AVX512DQ	Insert 256 bits of packed single-precision floating-point values from ymm3/m256 and the remaining values from zmm2 into zmm1 under writemask k1.
EVEX.NDS.512.66.0F3A.W1 1A /r ib VINSERTF64X4 zmm1 {k1}{z}, zmm2, ymm3/m256, imm8	T4	V/V	AVX512F	Insert 256 bits of packed double-precision floating-point values from ymm3/m256 and the remaining values from zmm2 into zmm1 under writemask k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RVM1	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	Imm8
T2, T4, T8	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	Imm8

### Description

VINSERTF128/VINSERTF32x4 and VINSERTF64x2 insert 128-bits of packed floating-point values from the second source operand (the third operand) into the destination operand (the first operand) at a 128-bit granularity offset multiplied by imm8[0] (256-bit) or imm8[1:0]. The remaining portions of the destination operand are copied from the corresponding fields of the first source operand (the second operand). The second source operand can be either an XMM register or a 128-bit memory location. The destination and first source operands are vector registers.

VINSERTF32x4: The destination operand is a ZMM/YMM register and updated at 32-bit granularity according to the writemask. The high 6/7 bits of the immediate are ignored.

VINSERTF64x2: The destination operand is a ZMM/YMM register and updated at 64-bit granularity according to the writemask. The high 6/7 bits of the immediate are ignored.

VINSERTF32x8 and VINSERTF64x4 inserts 256-bits of packed floating-point values from the second source operand (the third operand) into the destination operand (the first operand) at a 256-bit granular offset multiplied by imm8[0]. The remaining portions of the destination are copied from the corresponding fields of the first source operand (the second operand). The second source operand can be either an YMM register or a 256-bit memory

location. The high 7 bits of the immediate are ignored. The destination operand is a ZMM register and updated at 32/64-bit granularity according to the writemask.

### Operation

#### VINSERTF32x4 (EVEX encoded versions)

(KL, VL) = (8, 256), (16, 512)

TEMP\_DEST[VL-1:0] ← SRC1[VL-1:0]

IF VL = 256

CASE (imm8[0]) OF

0: TMP\_DEST[127:0] ← SRC2[127:0]

1: TMP\_DEST[255:128] ← SRC2[127:0]

ESAC.

FI;

IF VL = 512

CASE (imm8[1:0]) OF

00: TMP\_DEST[127:0] ← SRC2[127:0]

01: TMP\_DEST[255:128] ← SRC2[127:0]

10: TMP\_DEST[383:256] ← SRC2[127:0]

11: TMP\_DEST[511:384] ← SRC2[127:0]

ESAC.

FI;

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\*

THEN DEST[i+31:i] ← TMP\_DEST[i+31:i]

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+31:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

#### VINSERTF64x2 (EVEX encoded versions)

(KL, VL) = (4, 256), (8, 512)

TEMP\_DEST[VL-1:0] ← SRC1[VL-1:0]

IF VL = 256

CASE (imm8[0]) OF

0: TMP\_DEST[127:0] ← SRC2[127:0]

1: TMP\_DEST[255:128] ← SRC2[127:0]

ESAC.

FI;

IF VL = 512

CASE (imm8[1:0]) OF

00: TMP\_DEST[127:0] ← SRC2[127:0]

01: TMP\_DEST[255:128] ← SRC2[127:0]

10: TMP\_DEST[383:256] ← SRC2[127:0]

11: TMP\_DEST[511:384] ← SRC2[127:0]

ESAC.

FI;

FOR j ← 0 TO KL-1

i ← j \* 64

```

    IF k1[j] OR *no writemask*
      THEN DEST[j+63:i] ← TMP_DEST[j+63:i]
    ELSE
      IF *merging-masking*           ; merging-masking
        THEN *DEST[j+63:i] remains unchanged*
      ELSE                             ; zeroing-masking
        DEST[j+63:i] ← 0
      FI
    FI;
  ENDFOR
  DEST[MAX_VL-1:VL] ← 0

```

**VINSERTF32x8 (EVEX.U1.512 encoded version)**

```

  TEMP_DEST[VL-1:0] ← SRC1[VL-1:0]
  CASE (imm8[0]) OF
    0: TMP_DEST[255:0] ← SRC2[255:0]
    1: TMP_DEST[511:256] ← SRC2[255:0]
  ESAC.

```

```

  FOR j ← 0 TO 15
    i ← j * 32
    IF k1[j] OR *no writemask*
      THEN DEST[j+31:i] ← TMP_DEST[j+31:i]
    ELSE
      IF *merging-masking*           ; merging-masking
        THEN *DEST[j+31:i] remains unchanged*
      ELSE                             ; zeroing-masking
        DEST[j+31:i] ← 0
      FI
    FI;
  ENDFOR
  DEST[MAX_VL-1:VL] ← 0

```

**VINSERTF64x4 (EVEX.512 encoded version)**

```

  VL = 512
  TEMP_DEST[VL-1:0] ← SRC1[VL-1:0]
  CASE (imm8[0]) OF
    0: TMP_DEST[255:0] ← SRC2[255:0]
    1: TMP_DEST[511:256] ← SRC2[255:0]
  ESAC.

```

```

  FOR j ← 0 TO 7
    i ← j * 64
    IF k1[j] OR *no writemask*
      THEN DEST[j+63:i] ← TMP_DEST[j+63:i]
    ELSE
      IF *merging-masking*           ; merging-masking
        THEN *DEST[j+63:i] remains unchanged*
      ELSE                             ; zeroing-masking
        DEST[j+63:i] ← 0
      FI
    FI;
  ENDFOR
  DEST[MAX_VL-1:VL] ← 0

```

**VINSERTF128 (VEX encoded version)**

```

TEMP[255:0] ← SRC1[255:0]
CASE (imm8[0]) OF
  0: TEMP[127:0] ← SRC2[127:0]
  1: TEMP[255:128] ← SRC2[127:0]
ESAC
DEST ← TEMP

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VINSERTF32x4 __m512 __mm512_insertf32x4(__m512 a, __m128 b, int imm);
VINSERTF32x4 __m512 __mm512_mask_insertf32x4(__m512 s, __mmask16 k, __m512 a, __m128 b, int imm);
VINSERTF32x4 __m512 __mm512_maskz_insertf32x4(__mmask16 k, __m512 a, __m128 b, int imm);
VINSERTF32x4 __m256 __mm256_insertf32x4(__m256 a, __m128 b, int imm);
VINSERTF32x4 __m256 __mm256_mask_insertf32x4(__m256 s, __mmask8 k, __m256 a, __m128 b, int imm);
VINSERTF32x4 __m256 __mm256_maskz_insertf32x4(__mmask8 k, __m256 a, __m128 b, int imm);
VINSERTF32x8 __m512 __mm512_insertf32x8(__m512 a, __m256 b, int imm);
VINSERTF32x8 __m512 __mm512_mask_insertf32x8(__m512 s, __mmask16 k, __m512 a, __m256 b, int imm);
VINSERTF32x8 __m512 __mm512_maskz_insertf32x8(__mmask16 k, __m512 a, __m256 b, int imm);
VINSERTF64x2 __m512d __mm512_insertf64x2(__m512d a, __m128d b, int imm);
VINSERTF64x2 __m512d __mm512_mask_insertf64x2(__m512d s, __mmask8 k, __m512d a, __m128d b, int imm);
VINSERTF64x2 __m512d __mm512_maskz_insertf64x2(__mmask8 k, __m512d a, __m128d b, int imm);
VINSERTF64x2 __m256d __mm256_insertf64x2(__m256d a, __m128d b, int imm);
VINSERTF64x2 __m256d __mm256_mask_insertf64x2(__m256d s, __mmask8 k, __m256d a, __m128d b, int imm);
VINSERTF64x2 __m256d __mm256_maskz_insertf64x2(__mmask8 k, __m256d a, __m128d b, int imm);
VINSERTF64x4 __m512d __mm512_insertf64x4(__m512d a, __m256d b, int imm);
VINSERTF64x4 __m512d __mm512_mask_insertf64x4(__m512d s, __mmask8 k, __m512d a, __m256d b, int imm);
VINSERTF64x4 __m512d __mm512_maskz_insertf64x4(__mmask8 k, __m512d a, __m256d b, int imm);
VINSERTF128 __m256 __mm256_insertf128_ps(__m256 a, __m128 b, int offset);
VINSERTF128 __m256d __mm256_insertf128_pd(__m256d a, __m128d b, int offset);
VINSERTF128 __m256i __mm256_insertf128_si256(__m256i a, __m128i b, int offset);

```

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

VEX-encoded instruction, see Exceptions Type 6; additionally

#UD If VEX.L = 0.

EVEX-encoded instruction, see Exceptions Type E6NF.

## VINSERTI128/VINSERTI32x4/VINSERTI64x2/VINSERTI32x8/VINSERTI64x4—Insert Packed Integer Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.NDS.256.66.0F3A.W0 38 /r ib VINSERTI128 ymm1, ymm2, xmm3/m128, imm8	RVMI	V/V	AVX2	Insert 128 bits of integer data from xmm3/m128 and the remaining values from ymm2 into ymm1.
EVEX.NDS.256.66.0F3A.W0 38 /r ib VINSERTI32X4 ymm1 {k1}{z}, ymm2, xmm3/m128, imm8	T4	V/V	AVX512VL AVX512F	Insert 128 bits of packed doubleword integer values from xmm3/m128 and the remaining values from ymm2 into ymm1 under writemask k1.
EVEX.NDS.512.66.0F3A.W0 38 /r ib VINSERTI32X4 zmm1 {k1}{z}, zmm2, xmm3/m128, imm8	T4	V/V	AVX512F	Insert 128 bits of packed doubleword integer values from xmm3/m128 and the remaining values from zmm2 into zmm1 under writemask k1.
EVEX.NDS.256.66.0F3A.W1 38 /r ib VINSERTI64X2 ymm1 {k1}{z}, ymm2, xmm3/m128, imm8	T2	V/V	AVX512VL AVX512DQ	Insert 128 bits of packed quadword integer values from xmm3/m128 and the remaining values from ymm2 into ymm1 under writemask k1.
EVEX.NDS.512.66.0F3A.W1 38 /r ib VINSERTI64X2 zmm1 {k1}{z}, zmm2, xmm3/m128, imm8	T2	V/V	AVX512DQ	Insert 128 bits of packed quadword integer values from xmm3/m128 and the remaining values from zmm2 into zmm1 under writemask k1.
EVEX.NDS.512.66.0F3A.W0 3A /r ib VINSERTI32X8 zmm1 {k1}{z}, zmm2, ymm3/m256, imm8	T8	V/V	AVX512DQ	Insert 256 bits of packed doubleword integer values from ymm3/m256 and the remaining values from zmm2 into zmm1 under writemask k1.
EVEX.NDS.512.66.0F3A.W1 3A /r ib VINSERTI64X4 zmm1 {k1}{z}, zmm2, ymm3/m256, imm8	T4	V/V	AVX512F	Insert 256 bits of packed quadword integer values from ymm3/m256 and the remaining values from zmm2 into zmm1 under writemask k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RVMI	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	Imm8
T2, T4, T8	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	Imm8

### Description

VINSERTI32x4 and VINSERTI64x2 inserts 128-bits of packed integer values from the second source operand (the third operand) into the destination operand (the first operand) at an 128-bit granular offset multiplied by imm8[0] (256-bit) or imm8[1:0]. The remaining portions of the destination are copied from the corresponding fields of the first source operand (the second operand). The second source operand can be either an XMM register or a 128-bit memory location. The high 6/7bits of the immediate are ignored. The destination operand is a ZMM/YMM register and updated at 32 and 64-bit granularity according to the writemask.

VINSERTI32x8 and VINSERTI64x4 inserts 256-bits of packed integer values from the second source operand (the third operand) into the destination operand (the first operand) at a 256-bit granular offset multiplied by imm8[0]. The remaining portions of the destination are copied from the corresponding fields of the first source operand (the second operand). The second source operand can be either an YMM register or a 256-bit memory location. The upper bits of the immediate are ignored. The destination operand is a ZMM register and updated at 32 and 64-bit granularity according to the writemask.

VINSERTI128 inserts 128-bits of packed integer data from the second source operand (the third operand) into the destination operand (the first operand) at a 128-bit granular offset multiplied by imm8[0]. The remaining portions of the destination are copied from the corresponding fields of the first source operand (the second operand). The second source operand can be either an XMM register or a 128-bit memory location. The high 7 bits of the immediate are ignored. VEX.L must be 1, otherwise attempt to execute this instruction with VEX.L=0 will cause #UD.

### Operation

#### VINSERTI32x4 (EVEX encoded versions)

(KL, VL) = (8, 256), (16, 512)

TEMP\_DEST[VL-1:0] ← SRC1[VL-1:0]

IF VL = 256

CASE (imm8[0]) OF

0: TMP\_DEST[127:0] ← SRC2[127:0]

1: TMP\_DEST[255:128] ← SRC2[127:0]

ESAC.

FI;

IF VL = 512

CASE (imm8[1:0]) OF

00: TMP\_DEST[127:0] ← SRC2[127:0]

01: TMP\_DEST[255:128] ← SRC2[127:0]

10: TMP\_DEST[383:256] ← SRC2[127:0]

11: TMP\_DEST[511:384] ← SRC2[127:0]

ESAC.

FI;

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\*

THEN DEST[i+31:i] ← TMP\_DEST[i+31:i]

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+31:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

#### VINSERTI64x2 (EVEX encoded versions)

(KL, VL) = (4, 256), (8, 512)

TEMP\_DEST[VL-1:0] ← SRC1[VL-1:0]

IF VL = 256

CASE (imm8[0]) OF

0: TMP\_DEST[127:0] ← SRC2[127:0]

1: TMP\_DEST[255:128] ← SRC2[127:0]

ESAC.

FI;

IF VL = 512

CASE (imm8[1:0]) OF

00: TMP\_DEST[127:0] ← SRC2[127:0]

01: TMP\_DEST[255:128] ← SRC2[127:0]

10: TMP\_DEST[383:256] ← SRC2[127:0]

11: TMP\_DEST[511:384] ← SRC2[127:0]

ESAC.

```

FI;
FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+63:i] remains unchanged*
    ELSE ; zeroing-masking
      DEST[i+63:i] ← 0
    FI
  FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VINSERTI32x8 (EVEX.U1.512 encoded version)**

```

TEMP_DEST[VL-1:0] ← SRC1[VL-1:0]
CASE (imm8[0]) OF
  0: TMP_DEST[255:0] ← SRC2[255:0]
  1: TMP_DEST[511:256] ← SRC2[255:0]
ESAC.

```

```

FOR j ← 0 TO 15
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+31:i] remains unchanged*
    ELSE ; zeroing-masking
      DEST[i+31:i] ← 0
    FI
  FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VINSERTI64x4 (EVEX.512 encoded version)**

```

VL = 512
TEMP_DEST[VL-1:0] ← SRC1[VL-1:0]
CASE (imm8[0]) OF
  0: TMP_DEST[255:0] ← SRC2[255:0]
  1: TMP_DEST[511:256] ← SRC2[255:0]
ESAC.

```

```

FOR j ← 0 TO 7
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+63:i] remains unchanged*
    ELSE ; zeroing-masking
      DEST[i+63:i] ← 0
    FI
  FI;
ENDFOR

```



```

FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VINSERTI128**

```

TEMP[255:0] ← SRC1[255:0]
CASE (imm8[0]) OF
  0: TEMP[127:0] ← SRC2[127:0]
  1: TEMP[255:128] ← SRC2[127:0]
ESAC
DEST ← TEMP

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VINSERTI32x4 __m512i _inserti32x4( __m512i a, __m128i b, int imm);
VINSERTI32x4 __m512i _mask_inserti32x4( __m512i s, __mmask16 k, __m512i a, __m128i b, int imm);
VINSERTI32x4 __m512i _maskz_inserti32x4( __mmask16 k, __m512i a, __m128i b, int imm);
VINSERTI32x4 __m256i _mm256_inserti32x4( __m256i a, __m128i b, int imm);
VINSERTI32x4 __m256i _mm256_mask_inserti32x4( __m256i s, __mmask8 k, __m256i a, __m128i b, int imm);
VINSERTI32x4 __m256i _mm256_maskz_inserti32x4( __mmask8 k, __m256i a, __m128i b, int imm);
VINSERTI32x8 __m512i _mm512_inserti32x8( __m512i a, __m256i b, int imm);
VINSERTI32x8 __m512i _mm512_mask_inserti32x8( __m512i s, __mmask16 k, __m512i a, __m256i b, int imm);
VINSERTI32x8 __m512i _mm512_maskz_inserti32x8( __mmask16 k, __m512i a, __m256i b, int imm);
VINSERTI64x2 __m512i _mm512_inserti64x2( __m512i a, __m128i b, int imm);
VINSERTI64x2 __m512i _mm512_mask_inserti64x2( __m512i s, __mmask8 k, __m512i a, __m128i b, int imm);
VINSERTI64x2 __m512i _mm512_maskz_inserti64x2( __mmask8 k, __m512i a, __m128i b, int imm);
VINSERTI64x2 __m256i _mm256_inserti64x2( __m256i a, __m128i b, int imm);
VINSERTI64x2 __m256i _mm256_mask_inserti64x2( __m256i s, __mmask8 k, __m256i a, __m128i b, int imm);
VINSERTI64x2 __m256i _mm256_maskz_inserti64x2( __mmask8 k, __m256i a, __m128i b, int imm);
VINSERTI64x4 __m512i _mm512_inserti64x4( __m512i a, __m256i b, int imm);
VINSERTI64x4 __m512i _mm512_mask_inserti64x4( __m512i s, __mmask8 k, __m512i a, __m256i b, int imm);
VINSERTI64x4 __m512i _mm512_maskz_inserti64x4( __mmask8 k, __m512i a, __m256i b, int imm);
VINSERTI128 __m256i _mm256_insertf128_si256( __m256i a, __m128i b, int offset);

```

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

VEX-encoded instruction, see Exceptions Type 6; additionally

#UD If VEX.L = 0.

EVEX-encoded instruction, see Exceptions Type E6NF.

## VPBLENDMB/VPBLENDMW—Blend Byte/Word Vectors Using an Opmask Control

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.128.66.0F38.W0 66 /r VPBLENDMB xmm1 {k1}{z}, xmm2, xmm3/m128	FVM	V/V	AVX512VL AVX512BW	Blend byte integer vector xmm2 and byte vector xmm3/m128 and store the result in xmm1, under control mask.
EVEX.NDS.256.66.0F38.W0 66 /r VPBLENDMB ymm1 {k1}{z}, ymm2, ymm3/m256	FVM	V/V	AVX512VL AVX512BW	Blend byte integer vector ymm2 and byte vector ymm3/m256 and store the result in ymm1, under control mask.
EVEX.NDS.512.66.0F38.W0 66 /r VPBLENDMB zmm1 {k1}{z}, zmm2, zmm3/m512	FVM	V/V	AVX512BW	Blend byte integer vector zmm2 and byte vector zmm3/m512 and store the result in zmm1, under control mask.
EVEX.NDS.128.66.0F38.W1 66 /r VPBLENDMW xmm1 {k1}{z}, xmm2, xmm3/m128	FVM	V/V	AVX512VL AVX512BW	Blend word integer vector xmm2 and word vector xmm3/m128 and store the result in xmm1, under control mask.
EVEX.NDS.256.66.0F38.W1 66 /r VPBLENDMW ymm1 {k1}{z}, ymm2, ymm3/m256	FVM	V/V	AVX512VL AVX512BW	Blend word integer vector ymm2 and word vector ymm3/m256 and store the result in ymm1, under control mask.
EVEX.NDS.512.66.0F38.W1 66 /r VPBLENDMW zmm1 {k1}{z}, zmm2, zmm3/m512	FVM	V/V	AVX512BW	Blend word integer vector zmm2 and word vector zmm3/m512 and store the result in zmm1, under control mask.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
FVM	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

### Description

Performs an element-by-element blending of byte/word elements between the first source operand byte vector register and the second source operand byte vector from memory or register, using the instruction mask as selector. The result is written into the destination byte vector register.

The destination and first source operands are ZMM/YMM/XMM registers. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit memory location.

The mask is not used as a writemask for this instruction. Instead, the mask is used as an element selector: every element of the destination is conditionally selected between first source or second source using the value of the related mask bit (0 for first source, 1 for second source).

### Operation

#### VPBLENDMB (EVEX encoded versions)

(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] ← SRC2[i+7:i]

  ELSE

    IF \*merging-masking\* ; merging-masking

      THEN DEST[i+7:i] ← SRC1[i+7:i]

```

        ELSE                                ; zeroing-masking
            DEST[i+7:i] ← 0
        FI;
    ENDFOR
DEST[MAX_VL-1:VL] ← 0;

```

**VPBLENDMW (EVEX encoded versions)**

(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] ← SRC2[i+15:i]
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN DEST[i+15:i] ← SRC1[i+15:i]
        ELSE                                  ; zeroing-masking
            DEST[i+15:i] ← 0
        FI;
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VPBLENDMB __m512i __mm512_mask_blend_epi8(__mmask64 m, __m512i a, __m512i b);
VPBLENDMB __m256i __mm256_mask_blend_epi8(__mmask32 m, __m256i a, __m256i b);
VPBLENDMB __m128i __mm_mask_blend_epi8(__mmask16 m, __m128i a, __m128i b);
VPBLENDMW __m512i __mm512_mask_blend_epi16(__mmask32 m, __m512i a, __m512i b);
VPBLENDMW __m256i __mm256_mask_blend_epi16(__mmask16 m, __m256i a, __m256i b);
VPBLENDMW __m128i __mm_mask_blend_epi16(__mmask8 m, __m128i a, __m128i b);

```

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type E4.

## VPBLENDMD/VPBLENDMQ—Blend Int32/Int64 Vectors Using an OpMask Control

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.128.66.0F38.W0 64 /r VPBLENDMD xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	FV	V/V	AVX512VL AVX512F	Blend doubleword integer vector xmm2 and doubleword vector xmm3/m128/m32bcst and store the result in xmm1, under control mask.
EVEX.NDS.256.66.0F38.W0 64 /r VPBLENDMD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	FV	V/V	AVX512VL AVX512F	Blend doubleword integer vector ymm2 and doubleword vector ymm3/m256/m32bcst and store the result in ymm1, under control mask.
EVEX.NDS.512.66.0F38.W0 64 /r VPBLENDMD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	FV	V/V	AVX512F	Blend doubleword integer vector zmm2 and doubleword vector zmm3/m512/m32bcst and store the result in zmm1, under control mask.
EVEX.NDS.128.66.0F38.W1 64 /r VPBLENDMQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	FV	V/V	AVX512VL AVX512F	Blend quadword integer vector xmm2 and quadword vector xmm3/m128/m64bcst and store the result in xmm1, under control mask.
EVEX.NDS.256.66.0F38.W1 64 /r VPBLENDMQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	FV	V/V	AVX512VL AVX512F	Blend quadword integer vector ymm2 and quadword vector ymm3/m256/m64bcst and store the result in ymm1, under control mask.
EVEX.NDS.512.66.0F38.W1 64 /r VPBLENDMQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	FV	V/V	AVX512F	Blend quadword integer vector zmm2 and quadword vector zmm3/m512/m64bcst and store the result in zmm1, under control mask.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
FV	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

### Description

Performs an element-by-element blending of dword/qword elements between the first source operand (the second operand) and the elements of the second source operand (the third operand) using an opmask register as select control. The blended result is written into the destination.

The destination and first source operands are ZMM registers. The second source operand can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32-bit memory location.

The opmask register is not used as a writemask for this instruction. Instead, the mask is used as an element selector: every element of the destination is conditionally selected between first source or second source using the value of the related mask bit (0 for the first source operand, 1 for the second source operand).

If EVEX.z is set, the elements with corresponding mask bit value of 0 in the destination operand are zeroed.

### Operation

#### VPBLENDMD (EVEX encoded versions)

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

  i ← j \* 32

  IF k1[j] OR \*no controlmask\*

    THEN

      IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

        THEN

```

        DEST[i+31:i] ← SRC2[31:0]
    ELSE
        DEST[i+31:i] ← SRC2[i+31:i]
FI;
ELSE
    IF *merging-masking*           ; merging-masking
    THEN DEST[i+31:i] ← SRC1[i+31:i]
    ELSE                           ; zeroing-masking
        DEST[i+31:i] ← 0
    FI;
FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0;

```

**VPBLENDMD (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no controlmask\*

THEN

IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

THEN

DEST[i+31:i] ← SRC2[31:0]

ELSE

DEST[i+31:i] ← SRC2[i+31:i]

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN DEST[i+31:i] ← SRC1[i+31:i]

ELSE

DEST[i+31:i] ← 0

FI;

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

VPBLENDMD \_\_m512i \_mm512\_mask\_blend\_epi32(\_\_mmask16 k, \_\_m512i a, \_\_m512i b);

VPBLENDMD \_\_m256i \_mm256\_mask\_blend\_epi32(\_\_mmask8 m, \_\_m256i a, \_\_m256i b);

VPBLENDMD \_\_m128i \_mm\_mask\_blend\_epi32(\_\_mmask8 m, \_\_m128i a, \_\_m128i b);

VPBLENDMQ \_\_m512i \_mm512\_mask\_blend\_epi64(\_\_mmask8 k, \_\_m512i a, \_\_m512i b);

VPBLENDMQ \_\_m256i \_mm256\_mask\_blend\_epi64(\_\_mmask8 m, \_\_m256i a, \_\_m256i b);

VPBLENDMQ \_\_m128i \_mm\_mask\_blend\_epi64(\_\_mmask8 m, \_\_m128i a, \_\_m128i b);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type E4.

## VPBROADCASTB/W/D/Q—Load with Broadcast Integer Data from General Purpose Register

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 7A /r VPBROADCASTB xmm1 {k1}{z}, reg	T1S	V/V	AVX512VL AVX512BW	Broadcast an 8-bit value from a GPR to all bytes in the 128-bit destination subject to writemask k1.
EVEX.256.66.0F38.W0 7A /r VPBROADCASTB ymm1 {k1}{z}, reg	T1S	V/V	AVX512VL AVX512BW	Broadcast an 8-bit value from a GPR to all bytes in the 256-bit destination subject to writemask k1.
EVEX.512.66.0F38.W0 7A /r VPBROADCASTB zmm1 {k1}{z}, reg	T1S	V/V	AVX512BW	Broadcast an 8-bit value from a GPR to all bytes in the 512-bit destination subject to writemask k1.
EVEX.128.66.0F38.W0 7B /r VPBROADCASTW xmm1 {k1}{z}, reg	T1S	V/V	AVX512VL AVX512BW	Broadcast a 16-bit value from a GPR to all words in the 128-bit destination subject to writemask k1.
EVEX.256.66.0F38.W0 7B /r VPBROADCASTW ymm1 {k1}{z}, reg	T1S	V/V	AVX512VL AVX512BW	Broadcast a 16-bit value from a GPR to all words in the 256-bit destination subject to writemask k1.
EVEX.512.66.0F38.W0 7B /r VPBROADCASTW zmm1 {k1}{z}, reg	T1S	V/V	AVX512BW	Broadcast a 16-bit value from a GPR to all words in the 512-bit destination subject to writemask k1.
EVEX.128.66.0F38.W0 7C /r VPBROADCASTD xmm1 {k1}{z}, r32	T1S	V/V	AVX512VL AVX512F	Broadcast a 32-bit value from a GPR to all double-words in the 128-bit destination subject to writemask k1.
EVEX.256.66.0F38.W0 7C /r VPBROADCASTD ymm1 {k1}{z}, r32	T1S	V/V	AVX512VL AVX512F	Broadcast a 32-bit value from a GPR to all double-words in the 256-bit destination subject to writemask k1.
EVEX.512.66.0F38.W0 7C /r VPBROADCASTD zmm1 {k1}{z}, r32	T1S	V/V	AVX512F	Broadcast a 32-bit value from a GPR to all double-words in the 512-bit destination subject to writemask k1.
EVEX.128.66.0F38.W1 7C /r VPBROADCASTQ xmm1 {k1}{z}, r64	T1S	V/N.E. <sup>1</sup>	AVX512VL AVX512F	Broadcast a 64-bit value from a GPR to all quad-words in the 128-bit destination subject to writemask k1.
EVEX.256.66.0F38.W1 7C /r VPBROADCASTQ ymm1 {k1}{z}, r64	T1S	V/N.E. <sup>1</sup>	AVX512VL AVX512F	Broadcast a 64-bit value from a GPR to all quad-words in the 256-bit destination subject to writemask k1.
EVEX.512.66.0F38.W1 7C /r VPBROADCASTQ zmm1 {k1}{z}, r64	T1S	V/N.E. <sup>1</sup>	AVX512F	Broadcast a 64-bit value from a GPR to all quad-words in the 512-bit destination subject to writemask k1.

### NOTES:

- EVEX.W in non-64 bit is ignored; the instructions behaves as if the W0 version is used.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
T1S	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Broadcasts a 8-bit, 16-bit, 32-bit or 64-bit value from a general-purpose register (the second operand) to all the locations in the destination vector register (the first operand) using the writemask k1.

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

### Operation

#### VPBROADCASTB (EVEX encoded versions)

(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[7:0]
  ELSE
    IF *merging-masking*           ; merging-masking
      THEN *DEST[i+7:i] remains unchanged*
    ELSE                             ; zeroing-masking
      DEST[i+7:i] ← 0
    FI
  FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0
```

#### VPBROADCASTW (EVEX encoded versions)

(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[15:0]
  ELSE
    IF *merging-masking*           ; merging-masking
      THEN *DEST[i+15:i] remains unchanged*
    ELSE                             ; zeroing-masking
      DEST[i+15:i] ← 0
    FI
  FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0
```

#### VPBROADCASTD (EVEX encoded versions)

(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[31:0]
  ELSE
    IF *merging-masking*           ; merging-masking
      THEN *DEST[i+31:i] remains unchanged*
    ELSE                             ; zeroing-masking
      DEST[i+31:i] ← 0
    FI
  FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0
```

**VPBROADCASTQ (EVEX encoded versions)**

(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[63:0]

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

VPBROADCASTB \_\_m512i \_\_mm512\_mask\_set1\_epi8(\_\_m512i s, \_\_mmask64 k, int a);

VPBROADCASTB \_\_m512i \_\_mm512\_maskz\_set1\_epi8( \_\_mmask64 k, int a);

VPBROADCASTB \_\_m256i \_\_mm256\_mask\_set1\_epi8(\_\_m256i s, \_\_mmask32 k, int a);

VPBROADCASTB \_\_m256i \_\_mm256\_maskz\_set1\_epi8( \_\_mmask32 k, int a);

VPBROADCASTB \_\_m128i \_\_mm\_mask\_set1\_epi8(\_\_m128i s, \_\_mmask16 k, int a);

VPBROADCASTB \_\_m128i \_\_mm\_maskz\_set1\_epi8( \_\_mmask16 k, int a);

VPBROADCASTD \_\_m512i \_\_mm512\_mask\_set1\_epi32(\_\_m512i s, \_\_mmask16 k, int a);

VPBROADCASTD \_\_m512i \_\_mm512\_maskz\_set1\_epi32( \_\_mmask16 k, int a);

VPBROADCASTD \_\_m256i \_\_mm256\_mask\_set1\_epi32(\_\_m256i s, \_\_mmask8 k, int a);

VPBROADCASTD \_\_m256i \_\_mm256\_maskz\_set1\_epi32( \_\_mmask8 k, int a);

VPBROADCASTD \_\_m128i \_\_mm\_mask\_set1\_epi32(\_\_m128i s, \_\_mmask8 k, int a);

VPBROADCASTD \_\_m128i \_\_mm\_maskz\_set1\_epi32( \_\_mmask8 k, int a);

VPBROADCASTQ \_\_m512i \_\_mm512\_mask\_set1\_epi64(\_\_m512i s, \_\_mmask8 k, \_\_int64 a);

VPBROADCASTQ \_\_m512i \_\_mm512\_maskz\_set1\_epi64( \_\_mmask8 k, \_\_int64 a);

VPBROADCASTQ \_\_m256i \_\_mm256\_mask\_set1\_epi64(\_\_m256i s, \_\_mmask8 k, \_\_int64 a);

VPBROADCASTQ \_\_m256i \_\_mm256\_maskz\_set1\_epi64( \_\_mmask8 k, \_\_int64 a);

VPBROADCASTQ \_\_m128i \_\_mm\_mask\_set1\_epi64(\_\_m128i s, \_\_mmask8 k, \_\_int64 a);

VPBROADCASTQ \_\_m128i \_\_mm\_maskz\_set1\_epi64( \_\_mmask8 k, \_\_int64 a);

VPBROADCASTW \_\_m512i \_\_mm512\_mask\_set1\_epi16(\_\_m512i s, \_\_mmask32 k, int a);

VPBROADCASTW \_\_m512i \_\_mm512\_maskz\_set1\_epi16( \_\_mmask32 k, int a);

VPBROADCASTW \_\_m256i \_\_mm256\_mask\_set1\_epi16(\_\_m256i s, \_\_mmask16 k, int a);

VPBROADCASTW \_\_m256i \_\_mm256\_maskz\_set1\_epi16( \_\_mmask16 k, int a);

VPBROADCASTW \_\_m128i \_\_mm\_mask\_set1\_epi16(\_\_m128i s, \_\_mmask8 k, int a);

VPBROADCASTW \_\_m128i \_\_mm\_maskz\_set1\_epi16( \_\_mmask8 k, int a);

**Exceptions**

EVEX-encoded instructions, see Exceptions Type E7NM.

#UD If EVEX.vvvv != 1111B.



## VPBROADCAST—Load Integer and Broadcast

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F38.W0 78 /r VPBROADCASTB xmm1, xmm2/m8	RM	V/V	AVX2	Broadcast a byte integer in the source operand to sixteen locations in xmm1.
VEX.256.66.0F38.W0 78 /r VPBROADCASTB ymm1, xmm2/m8	RM	V/V	AVX2	Broadcast a byte integer in the source operand to thirty-two locations in ymm1.
EVEX.128.66.0F38.W0 78 /r VPBROADCASTB xmm1{k1}{z}, xmm2/m8	T1S	V/V	AVX512VL AVX512BW	Broadcast a byte integer in the source operand to locations in xmm1 subject to writemask k1.
EVEX.256.66.0F38.W0 78 /r VPBROADCASTB ymm1{k1}{z}, xmm2/m8	T1S	V/V	AVX512VL AVX512BW	Broadcast a byte integer in the source operand to locations in ymm1 subject to writemask k1.
EVEX.512.66.0F38.W0 78 /r VPBROADCASTB zmm1{k1}{z}, xmm2/m8	T1S	V/V	AVX512BW	Broadcast a byte integer in the source operand to 64 locations in zmm1 subject to writemask k1.
VEX.128.66.0F38.W0 79 /r VPBROADCASTW xmm1, xmm2/m16	RM	V/V	AVX2	Broadcast a word integer in the source operand to eight locations in xmm1.
VEX.256.66.0F38.W0 79 /r VPBROADCASTW ymm1, xmm2/m16	RM	V/V	AVX2	Broadcast a word integer in the source operand to sixteen locations in ymm1.
EVEX.128.66.0F38.W0 79 /r VPBROADCASTW xmm1{k1}{z}, xmm2/m16	T1S	V/V	AVX512VL AVX512BW	Broadcast a word integer in the source operand to locations in xmm1 subject to writemask k1.
EVEX.256.66.0F38.W0 79 /r VPBROADCASTW ymm1{k1}{z}, xmm2/m16	T1S	V/V	AVX512VL AVX512BW	Broadcast a word integer in the source operand to locations in ymm1 subject to writemask k1.
EVEX.512.66.0F38.W0 79 /r VPBROADCASTW zmm1{k1}{z}, xmm2/m16	T1S	V/V	AVX512BW	Broadcast a word integer in the source operand to 32 locations in zmm1 subject to writemask k1.
VEX.128.66.0F38.W0 58 /r VPBROADCASTD xmm1, xmm2/m32	RM	V/V	AVX2	Broadcast a dword integer in the source operand to four locations in xmm1.
VEX.256.66.0F38.W0 58 /r VPBROADCASTD ymm1, xmm2/m32	RM	V/V	AVX2	Broadcast a dword integer in the source operand to eight locations in ymm1.
EVEX.128.66.0F38.W0 58 /r VPBROADCASTD xmm1 {k1}{z}, xmm2/m32	T1S	V/V	AVX512VL AVX512F	Broadcast a dword integer in the source operand to locations in xmm1 subject to writemask k1.
EVEX.256.66.0F38.W0 58 /r VPBROADCASTD ymm1 {k1}{z}, xmm2/m32	T1S	V/V	AVX512VL AVX512F	Broadcast a dword integer in the source operand to locations in ymm1 subject to writemask k1.
EVEX.512.66.0F38.W0 58 /r VPBROADCASTD zmm1 {k1}{z}, xmm2/m32	T1S	V/V	AVX512F	Broadcast a dword integer in the source operand to locations in zmm1 subject to writemask k1.
VEX.128.66.0F38.W0 59 /r VPBROADCASTQ xmm1, xmm2/m64	RM	V/V	AVX2	Broadcast a qword element in source operand to two locations in xmm1.

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.256.66.0F38.W0 59 /r VPBROADCASTQ ymm1, xmm2/m64	RM	V/V	AVX2	Broadcast a qword element in source operand to four locations in ymm1.
EVEX.128.66.0F38.W1 59 /r VPBROADCASTQ xmm1 {k1}{z}, xmm2/m64	T1S	V/V	AVX512VL AVX512F	Broadcast a qword element in source operand to locations in xmm1 subject to writemask k1.
EVEX.256.66.0F38.W1 59 /r VPBROADCASTQ ymm1 {k1}{z}, xmm2/m64	T1S	V/V	AVX512VL AVX512F	Broadcast a qword element in source operand to locations in ymm1 subject to writemask k1.
EVEX.512.66.0F38.W1 59 /r VPBROADCASTQ zmm1 {k1}{z}, xmm2/m64	T1S	V/V	AVX512F	Broadcast a qword element in source operand to locations in zmm1 subject to writemask k1.
EVEX.128.66.0F38.W0 59 /r VBROADCASTI32x2 xmm1 {k1}{z}, xmm2/m64	T2	V/V	AVX512VL AVX512DQ	Broadcast two dword elements in source operand to locations in xmm1 subject to writemask k1.
EVEX.256.66.0F38.W0 59 /r VBROADCASTI32x2 ymm1 {k1}{z}, xmm2/m64	T2	V/V	AVX512VL AVX512DQ	Broadcast two dword elements in source operand to locations in ymm1 subject to writemask k1.
EVEX.512.66.0F38.W0 59 /r VBROADCASTI32x2 zmm1 {k1}{z}, xmm2/m64	T2	V/V	AVX512DQ	Broadcast two dword elements in source operand to locations in zmm1 subject to writemask k1.
VEX.256.66.0F38.W0 5A /r VBROADCASTI128 ymm1, m128	RM	V/V	AVX2	Broadcast 128 bits of integer data in mem to low and high 128-bits in ymm1.
EVEX.256.66.0F38.W0 5A /r VBROADCASTI32X4 ymm1 {k1}{z}, m128	T4	V/V	AVX512VL AVX512F	Broadcast 128 bits of 4 doubleword integer data in mem to locations in ymm1 using writemask k1.
EVEX.512.66.0F38.W0 5A /r VBROADCASTI32X4 zmm1 {k1}{z}, m128	T4	V/V	AVX512F	Broadcast 128 bits of 4 doubleword integer data in mem to locations in zmm1 using writemask k1.
EVEX.256.66.0F38.W1 5A /r VBROADCASTI64X2 ymm1 {k1}{z}, m128	T2	V/V	AVX512VL AVX512DQ	Broadcast 128 bits of 2 quadword integer data in mem to locations in ymm1 using writemask k1.
EVEX.512.66.0F38.W1 5A /r VBROADCASTI64X2 zmm1 {k1}{z}, m128	T2	V/V	AVX512DQ	Broadcast 128 bits of 2 quadword integer data in mem to locations in zmm1 using writemask k1.
EVEX.512.66.0F38.W0 5B /r VBROADCASTI32X8 zmm1 {k1}{z}, m256	T8	V/V	AVX512DQ	Broadcast 256 bits of 8 doubleword integer data in mem to locations in zmm1 using writemask k1.
EVEX.512.66.0F38.W1 5B /r VBROADCASTI64X4 zmm1 {k1}{z}, m256	T4	V/V	AVX512F	Broadcast 256 bits of 4 quadword integer data in mem to locations in zmm1 using writemask k1.

## Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
T1S, T2, T4, T8	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

## Description

Load integer data from the source operand (the second operand) and broadcast to all elements of the destination operand (the first operand).

VEX256-encoded VPBROADCASTB/W/D/Q: The source operand is 8-bit, 16-bit, 32-bit, 64-bit memory location or the low 8-bit, 16-bit 32-bit, 64-bit data in an XMM register. The destination operand is a YMM register.

VPBROADCASTI128 support the source operand of 128-bit memory location. Register source encodings for VPBROADCASTI128 is reserved and will #UD. Bits (MAX\_VL-1:256) of the destination register are zeroed.

EVEX-encoded VPBROADCASTD/Q: The source operand is a 32-bit, 64-bit memory location or the low 32-bit, 64-bit data in an XMM register. The destination operand is a ZMM/YMM/XMM register and updated according to the writemask k1.

VPBROADCASTI32X4 and VPBROADCASTI64X4: The destination operand is a ZMM register and updated according to the writemask k1. The source operand is 128-bit or 256-bit memory location. Register source encodings for VPBROADCASTI32X4 and VPBROADCASTI64X4 are reserved and will #UD.

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

If VPBROADCASTI128 is encoded with VEX.L= 0, an attempt to execute the instruction encoded with VEX.L= 0 will cause an #UD exception.

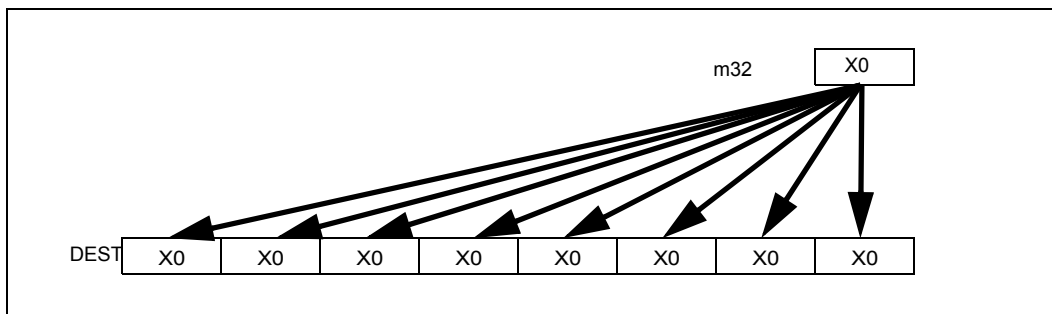


Figure 5-16. VPBROADCASTD Operation (VEX.256 encoded version)

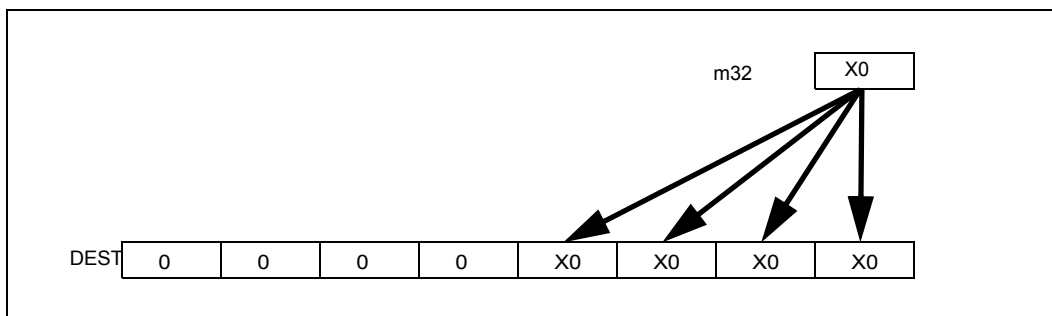


Figure 5-17. VPBROADCASTD Operation (128-bit version)

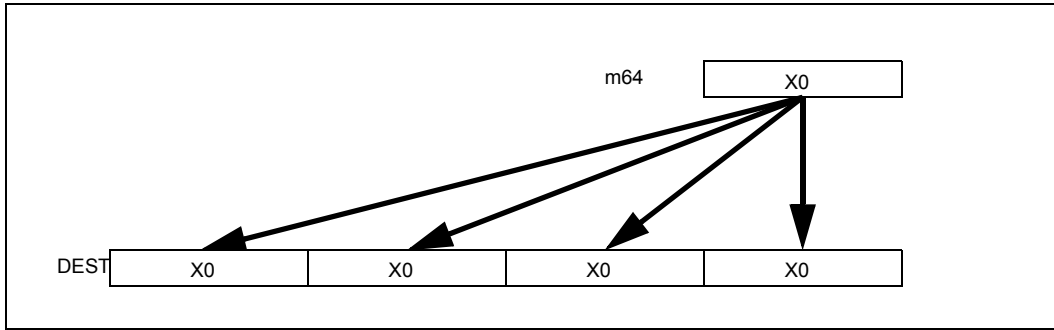


Figure 5-18. VPBROADCASTQ Operation (256-bit version)

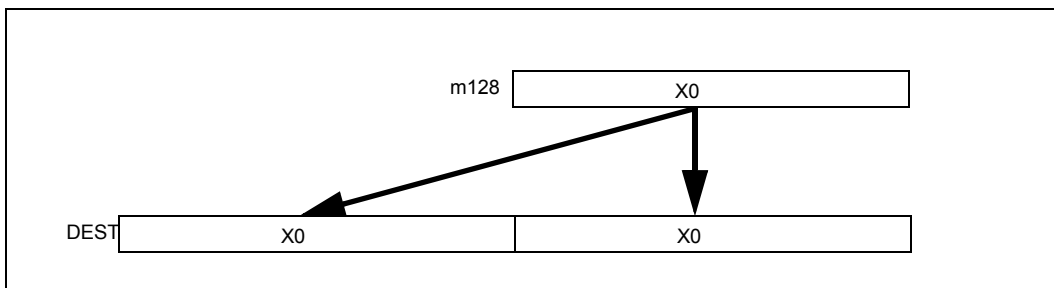


Figure 5-19. VBROADCASTI128 Operation (256-bit version)

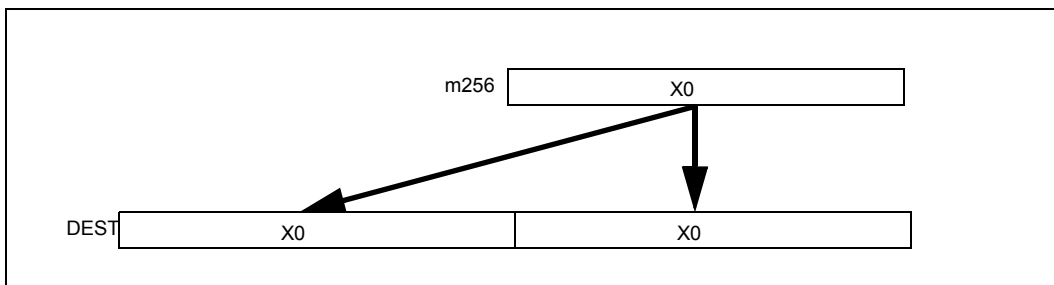


Figure 5-20. VBROADCASTI256 Operation (512-bit version)

**Operation**

**VPBROADCASTB (EVEX encoded versions)**

(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[7:0]

  ELSE

    IF \*merging-masking\* ; merging-masking

      THEN \*DEST[i+7:i] remains unchanged\*

    ELSE ; zeroing-masking

      DEST[i+7:i] ← 0

  FI

```

FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VPBROADCASTW (EVEX encoded versions)**

(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[15:0]
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+15:i] remains unchanged*
    ELSE ; zeroing-masking
      DEST[i+15:i] ← 0
    FI
  FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VPBROADCASTD (128 bit version)**

```

temp ← SRC[31:0]
DEST[31:0] ← temp
DEST[63:32] ← temp
DEST[95:64] ← temp
DEST[127:96] ← temp
DEST[MAX_VL-1:128] ← 0

```

**VPBROADCASTD (VEX.256 encoded version)**

```

temp ← SRC[31:0]
DEST[31:0] ← temp
DEST[63:32] ← temp
DEST[95:64] ← temp
DEST[127:96] ← temp
DEST[159:128] ← temp
DEST[191:160] ← temp
DEST[223:192] ← temp
DEST[255:224] ← temp
DEST[MAX_VL-1:256] ← 0

```

**VPBROADCASTD (EVEX encoded versions)**

(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[31:0]
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+31:i] remains unchanged*
    ELSE ; zeroing-masking
      DEST[i+31:i] ← 0
    FI
  FI;
ENDFOR

```

DEST[MAX\_VL-1:VL] ← 0

**VPBROADCASTQ (VEX.256 encoded version)**

temp ← SRC[63:0]  
 DEST[63:0] ← temp  
 DEST[127:64] ← temp  
 DEST[191:128] ← temp  
 DEST[255:192] ← temp  
 DEST[MAX\_VL-1:256] ← 0

**VPBROADCASTQ (EVEX encoded versions)**

(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[63:0]

  ELSE

    IF \*merging-masking\* ; merging-masking

      THEN \*DEST[i+63:i] remains unchanged\*

    ELSE ; zeroing-masking

      DEST[i+63:i] ← 0

  FI

  FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VBROADCASTI32x2 (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

  i ← j \* 32

  n ← (j mod 2) \* 32

  IF k1[j] OR \*no writemask\*

    THEN DEST[i+31:i] ← SRC[n+31:n]

  ELSE

    IF \*merging-masking\* ; merging-masking

      THEN \*DEST[i+31:i] remains unchanged\*

    ELSE ; zeroing-masking

      DEST[i+31:i] ← 0

  FI

  FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VBROADCASTI128 (VEX.256 encoded version)**

temp ← SRC[127:0]  
 DEST[127:0] ← temp  
 DEST[255:128] ← temp  
 DEST[MAX\_VL-1:256] ← 0

**VBROADCASTI32x4 (EVEX encoded versions)**

(KL, VL) = (8, 256), (16, 512)

FOR j ← 0 TO KL-1

  i ← j \* 32

  n ← (j modulo 4) \* 32

  IF k1[j] OR \*no writemask\*

```

    THEN DEST[i+31:i] ← SRC[n+31:n]
  ELSE
    IF *merging-masking*           ; merging-masking
      THEN *DEST[i+31:i] remains unchanged*
    ELSE                             ; zeroing-masking
      DEST[i+31:i] ← 0
    FI
  FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VBROADCASTI64X2 (EVEX encoded versions)**

(KL, VL) = (8, 256), (16, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 64
  n ← (j modulo 2) * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] ← SRC[n+63:n]
  ELSE
    IF *merging-masking*           ; merging-masking
      THEN *DEST[i+63:i] remains unchanged*
    ELSE                             ; zeroing-masking
      DEST[i+63:i] = 0
    FI
  FI;
ENDFOR;

```

**VBROADCASTI32X8 (EVEX.U1.512 encoded version)**

```

FOR j ← 0 TO 15
  i ← j * 32
  n ← (j modulo 8) * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ← SRC[n+31:n]
  ELSE
    IF *merging-masking*           ; merging-masking
      THEN *DEST[i+31:i] remains unchanged*
    ELSE                             ; zeroing-masking
      DEST[i+31:i] ← 0
    FI
  FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VBROADCASTI64X4 (EVEX.512 encoded version)**

```

FOR j ← 0 TO 7
  i ← j * 64
  n ← (j modulo 4) * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] ← SRC[n+63:n]
  ELSE
    IF *merging-masking*           ; merging-masking
      THEN *DEST[i+63:i] remains unchanged*
    ELSE                             ; zeroing-masking
      DEST[i+63:i] ← 0
    FI
  FI;

```

FI

```

FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VPBROADCASTB __m512i __mm512_broadcastb_epi8( __m128i a);
VPBROADCASTB __m512i __mm512_mask_broadcastb_epi8(__m512i s, __mmask64 k, __m128i a);
VPBROADCASTB __m512i __mm512_maskz_broadcastb_epi8( __mmask64 k, __m128i a);
VPBROADCASTB __m256i __mm256_broadcastb_epi8(__m128i a);
VPBROADCASTB __m256i __mm256_mask_broadcastb_epi8(__m256i s, __mmask32 k, __m128i a);
VPBROADCASTB __m256i __mm256_maskz_broadcastb_epi8( __mmask32 k, __m128i a);
VPBROADCASTB __m128i __mm_mask_broadcastb_epi8(__m128i s, __mmask16 k, __m128i a);
VPBROADCASTB __m128i __mm_maskz_broadcastb_epi8( __mmask16 k, __m128i a);
VPBROADCASTB __m128i __mm_broadcastb_epi8(__m128i a);
VPBROADCASTD __m512i __mm512_broadcastd_epi32( __m128i a);
VPBROADCASTD __m512i __mm512_mask_broadcastd_epi32(__m512i s, __mmask16 k, __m128i a);
VPBROADCASTD __m512i __mm512_maskz_broadcastd_epi32( __mmask16 k, __m128i a);
VPBROADCASTD __m256i __mm256_broadcastd_epi32( __m128i a);
VPBROADCASTD __m256i __mm256_mask_broadcastd_epi32(__m256i s, __mmask8 k, __m128i a);
VPBROADCASTD __m256i __mm256_maskz_broadcastd_epi32( __mmask8 k, __m128i a);
VPBROADCASTD __m128i __mm_broadcastd_epi32(__m128i a);
VPBROADCASTD __m128i __mm_mask_broadcastd_epi32(__m128i s, __mmask8 k, __m128i a);
VPBROADCASTD __m128i __mm_maskz_broadcastd_epi32( __mmask8 k, __m128i a);
VPBROADCASTQ __m512i __mm512_broadcastq_epi64( __m128i a);
VPBROADCASTQ __m512i __mm512_mask_broadcastq_epi64(__m512i s, __mmask8 k, __m128i a);
VPBROADCASTQ __m512i __mm512_maskz_broadcastq_epi64( __mmask8 k, __m128i a);
VPBROADCASTQ __m256i __mm256_broadcastq_epi64(__m128i a);
VPBROADCASTQ __m256i __mm256_mask_broadcastq_epi64(__m256i s, __mmask8 k, __m128i a);
VPBROADCASTQ __m256i __mm256_maskz_broadcastq_epi64( __mmask8 k, __m128i a);
VPBROADCASTQ __m128i __mm_broadcastq_epi64(__m128i a);
VPBROADCASTQ __m128i __mm_mask_broadcastq_epi64(__m128i s, __mmask8 k, __m128i a);
VPBROADCASTQ __m128i __mm_maskz_broadcastq_epi64( __mmask8 k, __m128i a);
VPBROADCASTW __m512i __mm512_broadcastw_epi16(__m128i a);
VPBROADCASTW __m512i __mm512_mask_broadcastw_epi16(__m512i s, __mmask32 k, __m128i a);
VPBROADCASTW __m512i __mm512_maskz_broadcastw_epi16( __mmask32 k, __m128i a);
VPBROADCASTW __m256i __mm256_broadcastw_epi16(__m128i a);
VPBROADCASTW __m256i __mm256_mask_broadcastw_epi16(__m256i s, __mmask16 k, __m128i a);
VPBROADCASTW __m256i __mm256_maskz_broadcastw_epi16( __mmask16 k, __m128i a);
VPBROADCASTW __m128i __mm_broadcastw_epi16(__m128i a);
VPBROADCASTW __m128i __mm_mask_broadcastw_epi16(__m128i s, __mmask8 k, __m128i a);
VPBROADCASTW __m128i __mm_maskz_broadcastw_epi16( __mmask8 k, __m128i a);
VBROADCASTI32x2 __m512i __mm512_broadcast_i32x2( __m128i a);
VBROADCASTI32x2 __m512i __mm512_mask_broadcast_i32x2(__m512i s, __mmask16 k, __m128i a);
VBROADCASTI32x2 __m512i __mm512_maskz_broadcast_i32x2( __mmask16 k, __m128i a);
VBROADCASTI32x2 __m256i __mm256_broadcast_i32x2( __m128i a);
VBROADCASTI32x2 __m256i __mm256_mask_broadcast_i32x2(__m256i s, __mmask8 k, __m128i a);
VBROADCASTI32x2 __m256i __mm256_maskz_broadcast_i32x2( __mmask8 k, __m128i a);
VBROADCASTI32x2 __m128i __mm_broadcastq_i32x2(__m128i a);
VBROADCASTI32x2 __m128i __mm_mask_broadcastq_i32x2(__m128i s, __mmask8 k, __m128i a);
VBROADCASTI32x2 __m128i __mm_maskz_broadcastq_i32x2( __mmask8 k, __m128i a);
VBROADCASTI32x4 __m512i __mm512_broadcast_i32x4( __m128i a);
VBROADCASTI32x4 __m512i __mm512_mask_broadcast_i32x4(__m512i s, __mmask16 k, __m128i a);
VBROADCASTI32x4 __m512i __mm512_maskz_broadcast_i32x4( __mmask16 k, __m128i a);

```



VBROADCASTI32x4 \_\_m256i \_\_mm256\_broadcast\_i32x4( \_\_m128i a);  
 VBROADCASTI32x4 \_\_m256i \_\_mm256\_mask\_broadcast\_i32x4(\_\_m256i s, \_\_mmask8 k, \_\_m128i a);  
 VBROADCASTI32x4 \_\_m256i \_\_mm256\_maskz\_broadcast\_i32x4( \_\_mmask8 k, \_\_m128i a);  
 VBROADCASTI32x8 \_\_m512i \_\_mm512\_broadcast\_i32x8( \_\_m256i a);  
 VBROADCASTI32x8 \_\_m512i \_\_mm512\_mask\_broadcast\_i32x8(\_\_m512i s, \_\_mmask16 k, \_\_m256i a);  
 VBROADCASTI32x8 \_\_m512i \_\_mm512\_maskz\_broadcast\_i32x8( \_\_mmask16 k, \_\_m256i a);  
 VBROADCASTI64x2 \_\_m512i \_\_mm512\_broadcast\_i64x2( \_\_m128i a);  
 VBROADCASTI64x2 \_\_m512i \_\_mm512\_mask\_broadcast\_i64x2(\_\_m512i s, \_\_mmask8 k, \_\_m128i a);  
 VBROADCASTI64x2 \_\_m512i \_\_mm512\_maskz\_broadcast\_i64x2( \_\_mmask8 k, \_\_m128i a);  
 VBROADCASTI64x2 \_\_m256i \_\_mm256\_broadcast\_i64x2( \_\_m128i a);  
 VBROADCASTI64x2 \_\_m256i \_\_mm256\_mask\_broadcast\_i64x2(\_\_m256i s, \_\_mmask8 k, \_\_m128i a);  
 VBROADCASTI64x2 \_\_m256i \_\_mm256\_maskz\_broadcast\_i64x2( \_\_mmask8 k, \_\_m128i a);  
 VBROADCASTI64x4 \_\_m512i \_\_mm512\_broadcast\_i64x4( \_\_m256i a);  
 VBROADCASTI64x4 \_\_m512i \_\_mm512\_mask\_broadcast\_i64x4(\_\_m512i s, \_\_mmask8 k, \_\_m256i a);  
 VBROADCASTI64x4 \_\_m512i \_\_mm512\_maskz\_broadcast\_i64x4( \_\_mmask8 k, \_\_m256i a);

### SIMD Floating-Point Exceptions

None

### Other Exceptions

EVEX-encoded instructions, see Exceptions Type 6;

EVEX-encoded instructions, syntax with reg/mem operand, see Exceptions Type E6.

#UD                    If VEX.L = 0 for VPBROADCASTQ, VPBROADCASTI128.  
                           If EVEX.L'L = 0 for VBROADCASTI32X4/VBROADCASTI64X2.  
                           If EVEX.L'L < 10b for VBROADCASTI32X8/VBROADCASTI64X4.

## VPCMPB/VPCMPUB—Compare Packed Byte Values Into Mask

Opcode/ Instruction	Op/ En	64/32 bitMode Support	CPUID Feature Flag	Description
EVEX.NDS.128.66.0F3A.W0 3F /r ib VPCMPB k1 {k2}, xmm2, xmm3/m128, imm8	FVM	V/V	AVX512VL AVX512BW	Compare packed signed byte values in xmm3/m128 and xmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.
EVEX.NDS.256.66.0F3A.W0 3F /r ib VPCMPB k1 {k2}, ymm2, ymm3/m256, imm8	FVM	V/V	AVX512VL AVX512BW	Compare packed signed byte values in ymm3/m256 and ymm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.
EVEX.NDS.512.66.0F3A.W0 3F /r ib VPCMPB k1 {k2}, zmm2, zmm3/m512, imm8	FVM	V/V	AVX512BW	Compare packed signed byte values in zmm3/m512 and zmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.
EVEX.NDS.128.66.0F3A.W0 3E /r ib VPCMPUB k1 {k2}, xmm2, xmm3/m128, imm8	FVM	V/V	AVX512VL AVX512BW	Compare packed unsigned byte values in xmm3/m128 and xmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.
EVEX.NDS.256.66.0F3A.W0 3E /r ib VPCMPUB k1 {k2}, ymm2, ymm3/m256, imm8	FVM	V/V	AVX512VL AVX512BW	Compare packed unsigned byte values in ymm3/m256 and ymm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.
EVEX.NDS.512.66.0F3A.W0 3E /r ib VPCMPUB k1 {k2}, zmm2, zmm3/m512, imm8	FVM	V/V	AVX512BW	Compare packed unsigned byte values in zmm3/m512 and zmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
FVM	ModRM:reg (w)	vvvv (r)	ModRM:r/m (r)	NA

### Description

Performs a SIMD compare of the packed byte values in the second source operand and the first source operand and returns the results of the comparison to the mask destination operand. The comparison predicate operand (immediate byte) specifies the type of comparison performed on each pair of packed values in the two source operands. The result of each comparison is a single mask bit result of 1 (comparison true) or 0 (comparison false).

VPCMPB performs a comparison between pairs of signed byte values.

VPCMPUB performs a comparison between pairs of unsigned byte values.

The first source operand (second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operand (first operand) is a mask register k1. Up to 64/32/16 comparisons are performed with results written to the destination operand under the writemask k2.

The comparison predicate operand is an 8-bit immediate: bits 2:0 define the type of comparison to be performed. Bits 3 through 7 of the immediate are reserved. Compiler can implement the pseudo-op mnemonic listed in Table 5-10.

Table 5-10. Pseudo-Op and VPCMP\* Implementation

Pseudo-Op	PCMPM Implementation
VPCMPEQ* <i>reg1, reg2, reg3</i>	VPCMP* <i>reg1, reg2, reg3, 0</i>
VPCMPLT* <i>reg1, reg2, reg3</i>	VPCMP* <i>reg1, reg2, reg3, 1</i>
VPCMPLE* <i>reg1, reg2, reg3</i>	VPCMP* <i>reg1, reg2, reg3, 2</i>
VPCMPNEQ* <i>reg1, reg2, reg3</i>	VPCMP* <i>reg1, reg2, reg3, 4</i>
VPPCMPNLT* <i>reg1, reg2, reg3</i>	VPCMP* <i>reg1, reg2, reg3, 5</i>
VPCMPNLE* <i>reg1, reg2, reg3</i>	VPCMP* <i>reg1, reg2, reg3, 6</i>

### Operation

CASE (COMPARISON PREDICATE) OF

- 0: OP ← EQ;
- 1: OP ← LT;
- 2: OP ← LE;
- 3: OP ← FALSE;
- 4: OP ← NEQ;
- 5: OP ← NLT;
- 6: OP ← NLE;
- 7: OP ← TRUE;

ESAC;

#### VPCMPB (EVEX encoded versions)

(KL, VL) = (16, 128), (32, 256), (64, 512)

FOR  $j \leftarrow 0$  TO KL-1

$i \leftarrow j * 8$

IF  $k2[j]$  OR \*no writemask\*

THEN

CMP ← SRC1[ $i+7:i$ ] OP SRC2[ $i+7:i$ ];

IF CMP = TRUE

THEN DEST[ $j$ ] ← 1;

ELSE DEST[ $j$ ] ← 0; FI;

ELSE DEST[ $j$ ] = 0 ; zeroing-masking onlyFI;

FI;

ENDFOR

DEST[MAX\_KL-1:KL] ← 0

#### VPCMPUB (EVEX encoded versions)

(KL, VL) = (16, 128), (32, 256), (64, 512)

FOR  $j \leftarrow 0$  TO KL-1

$i \leftarrow j * 8$

IF  $k2[j]$  OR \*no writemask\*

THEN

CMP ← SRC1[ $i+7:i$ ] OP SRC2[ $i+7:i$ ];

IF CMP = TRUE

THEN DEST[ $j$ ] ← 1;

ELSE DEST[ $j$ ] ← 0; FI;

ELSE DEST[ $j$ ] = 0 ; zeroing-masking onlyFI;

FI;

ENDFOR

DEST[MAX\_KL-1:KL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VPCMPB __mmask64_mm512_cmp_epi8_mask( __m512i a, __m512i b, int cmp);
VPCMPB __mmask64_mm512_mask_cmp_epi8_mask( __mmask64 m, __m512i a, __m512i b, int cmp);
VPCMPB __mmask32_mm256_cmp_epi8_mask( __m256i a, __m256i b, int cmp);
VPCMPB __mmask32_mm256_mask_cmp_epi8_mask( __mmask32 m, __m256i a, __m256i b, int cmp);
VPCMPB __mmask16_mm_cmp_epi8_mask( __m128i a, __m128i b, int cmp);
VPCMPB __mmask16_mm_mask_cmp_epi8_mask( __mmask16 m, __m128i a, __m128i b, int cmp);
VPCMPB __mmask64_mm512_cmp[eq|ge|gt|le|lt|neq]_epi8_mask( __m512i a, __m512i b);
VPCMPB __mmask64_mm512_mask_cmp[eq|ge|gt|le|lt|neq]_epi8_mask( __mmask64 m, __m512i a, __m512i b);
VPCMPB __mmask32_mm256_cmp[eq|ge|gt|le|lt|neq]_epi8_mask( __m256i a, __m256i b);
VPCMPB __mmask32_mm256_mask_cmp[eq|ge|gt|le|lt|neq]_epi8_mask( __mmask32 m, __m256i a, __m256i b);
VPCMPB __mmask16_mm_cmp[eq|ge|gt|le|lt|neq]_epi8_mask( __m128i a, __m128i b);
VPCMPB __mmask16_mm_mask_cmp[eq|ge|gt|le|lt|neq]_epi8_mask( __mmask16 m, __m128i a, __m128i b);
VPCMPUB __mmask64_mm512_cmp_epu8_mask( __m512i a, __m512i b, int cmp);
VPCMPUB __mmask64_mm512_mask_cmp_epu8_mask( __mmask64 m, __m512i a, __m512i b, int cmp);
VPCMPUB __mmask32_mm256_cmp_epu8_mask( __m256i a, __m256i b, int cmp);
VPCMPUB __mmask32_mm256_mask_cmp_epu8_mask( __mmask32 m, __m256i a, __m256i b, int cmp);
VPCMPUB __mmask16_mm_cmp_epu8_mask( __m128i a, __m128i b, int cmp);
VPCMPUB __mmask16_mm_mask_cmp_epu8_mask( __mmask16 m, __m128i a, __m128i b, int cmp);
VPCMPUB __mmask64_mm512_cmp[eq|ge|gt|le|lt|neq]_epu8_mask( __m512i a, __m512i b, int cmp);
VPCMPUB __mmask64_mm512_mask_cmp[eq|ge|gt|le|lt|neq]_epu8_mask( __mmask64 m, __m512i a, __m512i b, int cmp);
VPCMPUB __mmask32_mm256_cmp[eq|ge|gt|le|lt|neq]_epu8_mask( __m256i a, __m256i b, int cmp);
VPCMPUB __mmask32_mm256_mask_cmp[eq|ge|gt|le|lt|neq]_epu8_mask( __mmask32 m, __m256i a, __m256i b, int cmp);
VPCMPUB __mmask16_mm_cmp[eq|ge|gt|le|lt|neq]_epu8_mask( __m128i a, __m128i b, int cmp);
VPCMPUB __mmask16_mm_mask_cmp[eq|ge|gt|le|lt|neq]_epu8_mask( __mmask16 m, __m128i a, __m128i b, int cmp);

```

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

EVEX-encoded instruction, see Exceptions Type E4.nb.

## VPCMPD/VPCMPUD—Compare Packed Integer Values into Mask

Opcode/ Instruction	Op/ En	64/32 bitMode Support	CPUID Feature Flag	Description
EVEX.NDS.128.66.0F3A.W0 1F /r ib VPCMPD k1 {k2}, xmm2, xmm3/m128/m32bcst, imm8	FV	V/V	AVX512VL AVX512F	Compare packed signed doubleword integer values in xmm3/m128/m32bcst and xmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.
EVEX.NDS.256.66.0F3A.W0 1F /r ib VPCMPD k1 {k2}, ymm2, ymm3/m256/m32bcst, imm8	FV	V/V	AVX512VL AVX512F	Compare packed signed doubleword integer values in ymm3/m256/m32bcst and ymm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.
EVEX.NDS.512.66.0F3A.W0 1F /r ib VPCMPD k1 {k2}, zmm2, zmm3/m512/m32bcst, imm8	FV	V/V	AVX512F	Compare packed signed doubleword integer values in zmm2 and zmm3/m512/m32bcst using bits 2:0 of imm8 as a comparison predicate. The comparison results are written to the destination k1 under writemask k2.
EVEX.NDS.128.66.0F3A.W0 1E /r ib VPCMPUD k1 {k2}, xmm2, xmm3/m128/m32bcst, imm8	FV	V/V	AVX512VL AVX512F	Compare packed unsigned doubleword integer values in xmm3/m128/m32bcst and xmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.
EVEX.NDS.256.66.0F3A.W0 1E /r ib VPCMPUD k1 {k2}, ymm2, ymm3/m256/m32bcst, imm8	FV	V/V	AVX512VL AVX512F	Compare packed unsigned doubleword integer values in ymm3/m256/m32bcst and ymm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.
EVEX.NDS.512.66.0F3A.W0 1E /r ib VPCMPUD k1 {k2}, zmm2, zmm3/m512/m32bcst, imm8	FV	V/V	AVX512F	Compare packed unsigned doubleword integer values in zmm2 and zmm3/m512/m32bcst using bits 2:0 of imm8 as a comparison predicate. The comparison results are written to the destination k1 under writemask k2.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
FV	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	Imm8

### Description

Performs a SIMD compare of the packed integer values in the second source operand and the first source operand and returns the results of the comparison to the mask destination operand. The comparison predicate operand (immediate byte) specifies the type of comparison performed on each pair of packed values in the two source operands. The result of each comparison is a single mask bit result of 1 (comparison true) or 0 (comparison false).

VPCMPD/VPCMPUD performs a comparison between pairs of signed/unsigned doubleword integer values.

The first source operand (second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register or a 512/256/128-bit memory location or a 512-bit vector broadcasted from a 32-bit memory location. The destination operand (first operand) is a mask register k1. Up to 16/8/4 comparisons are performed with results written to the destination operand under the writemask k2.

The comparison predicate operand is an 8-bit immediate: bits 2:0 define the type of comparison to be performed. Bits 3 through 7 of the immediate are reserved. Compiler can implement the pseudo-op mnemonic listed in Table 5-10.

**Operation**

CASE (COMPARISON PREDICATE) OF

0: OP ← EQ;  
 1: OP ← LT;  
 2: OP ← LE;  
 3: OP ← FALSE;  
 4: OP ← NEQ;  
 5: OP ← NLT;  
 6: OP ← NLE;  
 7: OP ← TRUE;

ESAC;

**VPCMPD (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

i ← j \* 32

IF k2[j] OR \*no writemask\*

THEN

IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

THEN CMP ← SRC1[i+31:i] OP SRC2[31:0];

ELSE CMP ← SRC1[i+31:i] OP SRC2[i+31:i];

FI;

IF CMP = TRUE

THEN DEST[j] ← 1;

ELSE DEST[j] ← 0; FI;

ELSE DEST[j] ← 0 ; zeroing-masking onlyFI;

FI;

ENDFOR

DEST[MAX\_KL-1:KL] ← 0

**VPCMPUD (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

i ← j \* 32

IF k2[j] OR \*no writemask\*

THEN

IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

THEN CMP ← SRC1[i+31:i] OP SRC2[31:0];

ELSE CMP ← SRC1[i+31:i] OP SRC2[i+31:i];

FI;

IF CMP = TRUE

THEN DEST[j] ← 1;

ELSE DEST[j] ← 0; FI;

ELSE DEST[j] ← 0 ; zeroing-masking onlyFI;

FI;

ENDFOR

DEST[MAX\_KL-1:KL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

VPCMPD \_\_mmask16 \_\_mm512\_cmp\_epi32\_mask( \_\_m512i a, \_\_m512i b, int imm);

VPCMPD \_\_mmask16 \_\_mm512\_mask\_cmp\_epi32\_mask(\_\_mmask16 k, \_\_m512i a, \_\_m512i b, int imm);

VPCMPD \_\_mmask16 \_\_mm512\_cmp[eq|ge|gt|le|lt|neq]\_epi32\_mask( \_\_m512i a, \_\_m512i b);

VPCMPD \_\_mmask16 \_\_mm512\_mask\_cmp[eq|ge|gt|le|lt|neq]\_epi32\_mask(\_\_mmask16 k, \_\_m512i a, \_\_m512i b);

VPCMPUD \_\_mmask16 \_\_mm512\_cmp\_epu32\_mask( \_\_m512i a, \_\_m512i b, int imm);

VPCMPUD \_\_mmask16 \_\_mm512\_mask\_cmp\_epu32\_mask(\_\_mmask16 k, \_\_m512i a, \_\_m512i b, int imm);  
 VPCMPUD \_\_mmask16 \_\_mm512\_cmp[eq|ge|gt|le|lt|neq]\_epu32\_mask( \_\_m512i a, \_\_m512i b);  
 VPCMPUD \_\_mmask16 \_\_mm512\_mask\_cmp[eq|ge|gt|le|lt|neq]\_epu32\_mask(\_\_mmask16 k, \_\_m512i a, \_\_m512i b);  
 VPCMPD \_\_mmask8 \_\_mm256\_cmp\_epi32\_mask( \_\_m256i a, \_\_m256i b, int imm);  
 VPCMPD \_\_mmask8 \_\_mm256\_mask\_cmp\_epi32\_mask(\_\_mmask8 k, \_\_m256i a, \_\_m256i b, int imm);  
 VPCMPD \_\_mmask8 \_\_mm256\_cmp[eq|ge|gt|le|lt|neq]\_epi32\_mask( \_\_m256i a, \_\_m256i b);  
 VPCMPD \_\_mmask8 \_\_mm256\_mask\_cmp[eq|ge|gt|le|lt|neq]\_epi32\_mask(\_\_mmask8 k, \_\_m256i a, \_\_m256i b);  
 VPCMPUD \_\_mmask8 \_\_mm256\_cmp\_epu32\_mask( \_\_m256i a, \_\_m256i b, int imm);  
 VPCMPUD \_\_mmask8 \_\_mm256\_mask\_cmp\_epu32\_mask(\_\_mmask8 k, \_\_m256i a, \_\_m256i b, int imm);  
 VPCMPUD \_\_mmask8 \_\_mm256\_cmp[eq|ge|gt|le|lt|neq]\_epu32\_mask( \_\_m256i a, \_\_m256i b);  
 VPCMPUD \_\_mmask8 \_\_mm256\_mask\_cmp[eq|ge|gt|le|lt|neq]\_epu32\_mask(\_\_mmask8 k, \_\_m256i a, \_\_m256i b);  
 VPCMPD \_\_mmask8 \_\_mm\_cmp\_epi32\_mask( \_\_m128i a, \_\_m128i b, int imm);  
 VPCMPD \_\_mmask8 \_\_mm\_mask\_cmp\_epi32\_mask(\_\_mmask8 k, \_\_m128i a, \_\_m128i b, int imm);  
 VPCMPD \_\_mmask8 \_\_mm\_cmp[eq|ge|gt|le|lt|neq]\_epi32\_mask( \_\_m128i a, \_\_m128i b);  
 VPCMPD \_\_mmask8 \_\_mm\_mask\_cmp[eq|ge|gt|le|lt|neq]\_epi32\_mask(\_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
 VPCMPUD \_\_mmask8 \_\_mm\_cmp\_epu32\_mask( \_\_m128i a, \_\_m128i b, int imm);  
 VPCMPUD \_\_mmask8 \_\_mm\_mask\_cmp\_epu32\_mask(\_\_mmask8 k, \_\_m128i a, \_\_m128i b, int imm);  
 VPCMPUD \_\_mmask8 \_\_mm\_cmp[eq|ge|gt|le|lt|neq]\_epu32\_mask( \_\_m128i a, \_\_m128i b);  
 VPCMPUD \_\_mmask8 \_\_mm\_mask\_cmp[eq|ge|gt|le|lt|neq]\_epu32\_mask(\_\_mmask8 k, \_\_m128i a, \_\_m128i b);

### SIMD Floating-Point Exceptions

None

### Other Exceptions

EVEX-encoded instruction, see Exceptions Type E4.

## VPCMPQ/VPCMPUQ—Compare Packed Integer Values into Mask

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.128.66.0F3A.W1 1F /r ib VPCMPQ k1 {k2}, xmm2, xmm3/m128/m64bcst, imm8	FV	V/V	AVX512VL AVX512F	Compare packed signed quadword integer values in xmm3/m128/m64bcst and xmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.
EVEX.NDS.256.66.0F3A.W1 1F /r ib VPCMPQ k1 {k2}, ymm2, ymm3/m256/m64bcst, imm8	FV	V/V	AVX512VL AVX512F	Compare packed signed quadword integer values in ymm3/m256/m64bcst and ymm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.
EVEX.NDS.512.66.0F3A.W1 1F /r ib VPCMPQ k1 {k2}, zmm2, zmm3/m512/m64bcst, imm8	FV	V/V	AVX512F	Compare packed signed quadword integer values in zmm3/m512/m64bcst and zmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.
EVEX.NDS.128.66.0F3A.W1 1E /r ib VPCMPUQ k1 {k2}, xmm2, xmm3/m128/m64bcst, imm8	FV	V/V	AVX512VL AVX512F	Compare packed unsigned quadword integer values in xmm3/m128/m64bcst and xmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.
EVEX.NDS.256.66.0F3A.W1 1E /r ib VPCMPUQ k1 {k2}, ymm2, ymm3/m256/m64bcst, imm8	FV	V/V	AVX512VL AVX512F	Compare packed unsigned quadword integer values in ymm3/m256/m64bcst and ymm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.
EVEX.NDS.512.66.0F3A.W1 1E /r ib VPCMPUQ k1 {k2}, zmm2, zmm3/m512/m64bcst, imm8	FV	V/V	AVX512F	Compare packed unsigned quadword integer values in zmm3/m512/m64bcst and zmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
FV	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	Imm8

### Description

Performs a SIMD compare of the packed integer values in the second source operand and the first source operand and returns the results of the comparison to the mask destination operand. The comparison predicate operand (immediate byte) specifies the type of comparison performed on each pair of packed values in the two source operands. The result of each comparison is a single mask bit result of 1 (comparison true) or 0 (comparison false).

VPCMPQ/VPCMPUQ performs a comparison between pairs of signed/unsigned quadword integer values.

The first source operand (second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register or a 512/256/128-bit memory location or a 512-bit vector broadcasted from a 64-bit memory location. The destination operand (first operand) is a mask register k1. Up to 8/4/2 comparisons are performed with results written to the destination operand under the writemask k2.

The comparison predicate operand is an 8-bit immediate: bits 2:0 define the type of comparison to be performed. Bits 3 through 7 of the immediate are reserved. Compiler can implement the pseudo-op mnemonic listed in Table 5-10.



**Operation**

CASE (COMPARISON PREDICATE) OF

0: OP ← EQ;  
 1: OP ← LT;  
 2: OP ← LE;  
 3: OP ← FALSE;  
 4: OP ← NEQ;  
 5: OP ← NLT;  
 6: OP ← NLE;  
 7: OP ← TRUE;

ESAC;

**VPCMPQ (EVEX encoded versions)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

i ← j \* 64

IF k2[j] OR \*no writemask\*

THEN

IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

THEN CMP ← SRC1[i+63:i] OP SRC2[63:0];

ELSE CMP ← SRC1[i+63:i] OP SRC2[i+63:i];

FI;

IF CMP = TRUE

THEN DEST[j] ← 1;

ELSE DEST[j] ← 0; FI;

ELSE DEST[j] ← 0 ; zeroing-masking only

FI;

ENDFOR

DEST[MAX\_KL-1:KL] ← 0

**VPCMPUQ (EVEX encoded versions)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

i ← j \* 64

IF k2[j] OR \*no writemask\*

THEN

IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

THEN CMP ← SRC1[i+63:i] OP SRC2[63:0];

ELSE CMP ← SRC1[i+63:i] OP SRC2[i+63:i];

FI;

IF CMP = TRUE

THEN DEST[j] ← 1;

ELSE DEST[j] ← 0; FI;

ELSE DEST[j] ← 0 ; zeroing-masking only

FI;

ENDFOR

DEST[MAX\_KL-1:KL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

VPCMPQ \_\_mmask8 \_mm512\_cmp\_epi64\_mask( \_\_m512i a, \_\_m512i b, int imm);

VPCMPQ \_\_mmask8 \_mm512\_mask\_cmp\_epi64\_mask(\_\_mmask8 k, \_\_m512i a, \_\_m512i b, int imm);

VPCMPQ \_\_mmask8 \_mm512\_cmp[eq|ge|gt|le|lt|neq]\_epi64\_mask( \_\_m512i a, \_\_m512i b);

VPCMPQ \_\_mmask8 \_mm512\_mask\_cmp[eq|ge|gt|le|lt|neq]\_epi64\_mask(\_\_mmask8 k, \_\_m512i a, \_\_m512i b);

VPCMPUQ \_\_mmask8 \_mm512\_cmp\_epu64\_mask( \_\_m512i a, \_\_m512i b, int imm);

VPCMPUQ \_\_mmask8\_mm512\_mask\_cmp\_epu64\_mask(\_\_mmask8 k, \_\_m512i a, \_\_m512i b, int imm);  
 VPCMPUQ \_\_mmask8\_mm512\_cmp[eq|ge|gt|le|lt|neq]\_epu64\_mask( \_\_m512i a, \_\_m512i b);  
 VPCMPUQ \_\_mmask8\_mm512\_mask\_cmp[eq|ge|gt|le|lt|neq]\_epu64\_mask(\_\_mmask8 k, \_\_m512i a, \_\_m512i b);  
 VPCMPQ \_\_mmask8\_mm256\_mask\_cmp\_epi64\_mask( \_\_m256i a, \_\_m256i b, int imm);  
 VPCMPQ \_\_mmask8\_mm256\_mask\_cmp\_epi64\_mask(\_\_mmask8 k, \_\_m256i a, \_\_m256i b, int imm);  
 VPCMPQ \_\_mmask8\_mm256\_cmp[eq|ge|gt|le|lt|neq]\_epi64\_mask( \_\_m256i a, \_\_m256i b);  
 VPCMPQ \_\_mmask8\_mm256\_mask\_cmp[eq|ge|gt|le|lt|neq]\_epi64\_mask(\_\_mmask8 k, \_\_m256i a, \_\_m256i b);  
 VPCMPUQ \_\_mmask8\_mm256\_cmp\_epu64\_mask( \_\_m256i a, \_\_m256i b, int imm);  
 VPCMPUQ \_\_mmask8\_mm256\_mask\_cmp\_epu64\_mask(\_\_mmask8 k, \_\_m256i a, \_\_m256i b, int imm);  
 VPCMPUQ \_\_mmask8\_mm256\_cmp[eq|ge|gt|le|lt|neq]\_epu64\_mask( \_\_m256i a, \_\_m256i b);  
 VPCMPUQ \_\_mmask8\_mm256\_mask\_cmp[eq|ge|gt|le|lt|neq]\_epu64\_mask(\_\_mmask8 k, \_\_m256i a, \_\_m256i b);  
 VPCMPQ \_\_mmask8\_mm\_cmp\_epi64\_mask( \_\_m128i a, \_\_m128i b, int imm);  
 VPCMPQ \_\_mmask8\_mm\_mask\_cmp\_epi64\_mask(\_\_mmask8 k, \_\_m128i a, \_\_m128i b, int imm);  
 VPCMPQ \_\_mmask8\_mm\_cmp[eq|ge|gt|le|lt|neq]\_epi64\_mask( \_\_m128i a, \_\_m128i b);  
 VPCMPQ \_\_mmask8\_mm\_mask\_cmp[eq|ge|gt|le|lt|neq]\_epi64\_mask(\_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
 VPCMPUQ \_\_mmask8\_mm\_cmp\_epu64\_mask( \_\_m128i a, \_\_m128i b, int imm);  
 VPCMPUQ \_\_mmask8\_mm\_mask\_cmp\_epu64\_mask(\_\_mmask8 k, \_\_m128i a, \_\_m128i b, int imm);  
 VPCMPUQ \_\_mmask8\_mm\_cmp[eq|ge|gt|le|lt|neq]\_epu64\_mask( \_\_m128i a, \_\_m128i b);  
 VPCMPUQ \_\_mmask8\_mm\_mask\_cmp[eq|ge|gt|le|lt|neq]\_epu64\_mask(\_\_mmask8 k, \_\_m128i a, \_\_m128i b);

### SIMD Floating-Point Exceptions

None

### Other Exceptions

EVEX-encoded instruction, see Exceptions Type E4.

## VPCMPW/VPCMPUW—Compare Packed Word Values Into Mask

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.128.66.0F3A.W1 3F /r ib VPCMPW k1 {k2}, xmm2, xmm3/m128, imm8	FVM	V/V	AVX512VL AVX512BW	Compare packed signed word integers in xmm3/m128 and xmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.
EVEX.NDS.256.66.0F3A.W1 3F /r ib VPCMPW k1 {k2}, ymm2, ymm3/m256, imm8	FVM	V/V	AVX512VL AVX512BW	Compare packed signed word integers in ymm3/m256 and ymm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.
EVEX.NDS.512.66.0F3A.W1 3F /r ib VPCMPW k1 {k2}, zmm2, zmm3/m512, imm8	FVM	V/V	AVX512BW	Compare packed signed word integers in zmm3/m512 and zmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.
EVEX.NDS.128.66.0F3A.W1 3E /r ib VPCMPUW k1 {k2}, xmm2, xmm3/m128, imm8	FVM	V/V	AVX512VL AVX512BW	Compare packed unsigned word integers in xmm3/m128 and xmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.
EVEX.NDS.256.66.0F3A.W1 3E /r ib VPCMPUW k1 {k2}, ymm2, ymm3/m256, imm8	FVM	V/V	AVX512VL AVX512BW	Compare packed unsigned word integers in ymm3/m256 and ymm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.
EVEX.NDS.512.66.0F3A.W1 3E /r ib VPCMPUW k1 {k2}, zmm2, zmm3/m512, imm8	FVM	V/V	AVX512BW	Compare packed unsigned word integers in zmm3/m512 and zmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
FVM	ModRM:reg (w)	vvvv (r)	ModRM:r/m (r)	NA

### Description

Performs a SIMD compare of the packed integer word in the second source operand and the first source operand and returns the results of the comparison to the mask destination operand. The comparison predicate operand (immediate byte) specifies the type of comparison performed on each pair of packed values in the two source operands. The result of each comparison is a single mask bit result of 1 (comparison true) or 0 (comparison false).

VPCMPW performs a comparison between pairs of signed word values.

VPCMPUW performs a comparison between pairs of unsigned word values.

The first source operand (second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operand (first operand) is a mask register k1. Up to 32/16/8 comparisons are performed with results written to the destination operand under the writemask k2.

The comparison predicate operand is an 8-bit immediate: bits 2:0 define the type of comparison to be performed. Bits 3 through 7 of the immediate are reserved. Compiler can implement the pseudo-op mnemonic listed in Table 5-10.

**Operation**

CASE (COMPARISON PREDICATE) OF

0: OP ← EQ;  
 1: OP ← LT;  
 2: OP ← LE;  
 3: OP ← FALSE;  
 4: OP ← NEQ;  
 5: OP ← NLT;  
 6: OP ← NLE;  
 7: OP ← TRUE;

ESAC;

**VPCMPW (EVEX encoded versions)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j ← 0 TO KL-1

i ← j \* 16

IF k2[j] OR \*no writemask\*

THEN

ICMP ← SRC1[i+15:i] OP SRC2[i+15:i];

IF CMP = TRUE

THEN DEST[j] ← 1;

ELSE DEST[j] ← 0; FI;

ELSE DEST[j] = 0 ; zeroing-masking only

FI;

ENDFOR

DEST[MAX\_KL-1:KL] ← 0

**VPCMPUW (EVEX encoded versions)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j ← 0 TO KL-1

i ← j \* 16

IF k2[j] OR \*no writemask\*

THEN

CMP ← SRC1[i+15:i] OP SRC2[i+15:i];

IF CMP = TRUE

THEN DEST[j] ← 1;

ELSE DEST[j] ← 0; FI;

ELSE DEST[j] = 0 ; zeroing-masking only

FI;

ENDFOR

DEST[MAX\_KL-1:KL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

VPCMPW \_\_mmask32 \_\_mm512\_cmp\_epi16\_mask( \_\_m512i a, \_\_m512i b, int cmp);

VPCMPW \_\_mmask32 \_\_mm512\_mask\_cmp\_epi16\_mask( \_\_mmask32 m, \_\_m512i a, \_\_m512i b, int cmp);

VPCMPW \_\_mmask16 \_\_mm256\_cmp\_epi16\_mask( \_\_m256i a, \_\_m256i b, int cmp);

VPCMPW \_\_mmask16 \_\_mm256\_mask\_cmp\_epi16\_mask( \_\_mmask16 m, \_\_m256i a, \_\_m256i b, int cmp);

VPCMPW \_\_mmask8 \_\_mm\_cmp\_epi16\_mask( \_\_m128i a, \_\_m128i b, int cmp);

VPCMPW \_\_mmask8 \_\_mm\_mask\_cmp\_epi16\_mask( \_\_mmask8 m, \_\_m128i a, \_\_m128i b, int cmp);

VPCMPW \_\_mmask32 \_\_mm512\_cmp[eq|ge|gt|le|lt|neq]\_epi16\_mask( \_\_m512i a, \_\_m512i b);

VPCMPW \_\_mmask32 \_\_mm512\_mask\_cmp[eq|ge|gt|le|lt|neq]\_epi16\_mask( \_\_mmask32 m, \_\_m512i a, \_\_m512i b);

VPCMPW \_\_mmask16 \_\_mm256\_cmp[eq|ge|gt|le|lt|neq]\_epi16\_mask( \_\_m256i a, \_\_m256i b);

VPCMPW \_\_mmask16 \_\_mm256\_mask\_cmp[eq|ge|gt|le|lt|neq]\_epi16\_mask( \_\_mmask16 m, \_\_m256i a, \_\_m256i b);

VPCMPW \_\_mmask8 \_\_mm\_cmp[eq|ge|gt|le|lt|neq]\_epi16\_mask( \_\_m128i a, \_\_m128i b);

VPCMPW \_\_mmask8 \_mm\_mask\_cmp[eq|ge|gt|le|lt|neq]\_epi16\_mask( \_\_mmask8 m, \_\_m128i a, \_\_m128i b);  
 VPCMPUW \_\_mmask32 \_mm512\_cmp\_epu16\_mask( \_\_m512i a, \_\_m512i b, int cmp);  
 VPCMPUW \_\_mmask32 \_mm512\_mask\_cmp\_epu16\_mask( \_\_mmask32 m, \_\_m512i a, \_\_m512i b, int cmp);  
 VPCMPUW \_\_mmask16 \_mm256\_cmp\_epu16\_mask( \_\_m256i a, \_\_m256i b, int cmp);  
 VPCMPUW \_\_mmask16 \_mm256\_mask\_cmp\_epu16\_mask( \_\_mmask16 m, \_\_m256i a, \_\_m256i b, int cmp);  
 VPCMPUW \_\_mmask8 \_mm\_cmp\_epu16\_mask( \_\_m128i a, \_\_m128i b, int cmp);  
 VPCMPUW \_\_mmask8 \_mm\_mask\_cmp\_epu16\_mask( \_\_mmask8 m, \_\_m128i a, \_\_m128i b, int cmp);  
 VPCMPUW \_\_mmask32 \_mm512\_cmp[eq|ge|gt|le|lt|neq]\_epu16\_mask( \_\_m512i a, \_\_m512i b, int cmp);  
 VPCMPUW \_\_mmask32 \_mm512\_mask\_cmp[eq|ge|gt|le|lt|neq]\_epu16\_mask( \_\_mmask32 m, \_\_m512i a, \_\_m512i b, int cmp);  
 VPCMPUW \_\_mmask16 \_mm256\_cmp[eq|ge|gt|le|lt|neq]\_epu16\_mask( \_\_m256i a, \_\_m256i b, int cmp);  
 VPCMPUW \_\_mmask16 \_mm256\_mask\_cmp[eq|ge|gt|le|lt|neq]\_epu16\_mask( \_\_mmask16 m, \_\_m256i a, \_\_m256i b, int cmp);  
 VPCMPUW \_\_mmask8 \_mm\_cmp[eq|ge|gt|le|lt|neq]\_epu16\_mask( \_\_m128i a, \_\_m128i b, int cmp);  
 VPCMPUW \_\_mmask8 \_mm\_mask\_cmp[eq|ge|gt|le|lt|neq]\_epu16\_mask( \_\_mmask8 m, \_\_m128i a, \_\_m128i b, int cmp);

### SIMD Floating-Point Exceptions

None

### Other Exceptions

EVEX-encoded instruction, see Exceptions Type E4.nb.

## VPCOMPRESSD—Store Sparse Packed Doubleword Integer Values into Dense Memory/Register

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 8B /r VPCOMPRESSD xmm1/m128 {k1}{z}, xmm2	T1S	V/V	AVX512VL AVX512F	Compress packed doubleword integer values from xmm2 to xmm1/m128 using controlmask k1.
EVEX.256.66.0F38.W0 8B /r VPCOMPRESSD ymm1/m256 {k1}{z}, ymm2	T1S	V/V	AVX512VL AVX512F	Compress packed doubleword integer values from ymm2 to ymm1/m256 using controlmask k1.
EVEX.512.66.0F38.W0 8B /r VPCOMPRESSD zmm1/m512 {k1}{z}, zmm2	T1S	V/V	AVX512F	Compress packed doubleword integer values from zmm2 to zmm1/m512 using controlmask k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
T1S	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

### Description

Compress (store) up to 16/8/4 doubleword integer values from the source operand (second operand) to the destination operand (first operand). The source operand is a ZMM/YMM/XMM register, the destination operand can be a ZMM/YMM/XMM register or a 512/256/128-bit memory location.

The opmask register k1 selects the active elements (partial vector or possibly non-contiguous if less than 16 active elements) from the source operand to compress into a contiguous vector. The contiguous vector is written to the destination starting from the low element of the destination operand.

Memory destination version: Only the contiguous vector is written to the destination memory location. EVEX.z must be zero.

Register destination version: If the vector length of the contiguous vector is less than that of the input vector in the source operand, the upper bits of the destination register are unmodified if EVEX.z is not set, otherwise the upper bits are zeroed.

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

Note that the compressed displacement assumes a pre-scaling (N) corresponding to the size of one single element instead of the size of the full vector.

### Operation

**VPCOMPRESSD (EVEX encoded versions) store form**

(KL, VL) = (4, 128), (8, 256), (16, 512)

SIZE ← 32

k ← 0

FOR j ← 0 TO KL-1

  i ← j \* 32

  IF k1[j] OR \*no controlmask\*

    THEN

      DEST[k+SIZE-1:k] ← SRC[i+31:i]

      k ← k + SIZE

  FI;

ENDFOR;

**VPCOMPRESSD (EVEX encoded versions) reg-reg form**

(KL, VL) = (4, 128), (8, 256), (16, 512)

SIZE ← 32

k ← 0

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no controlmask\*

THEN

DEST[k+SIZE-1:k] ← SRC[i+31:i]

k ← k + SIZE

FI;

ENDFOR

IF \*merging-masking\*

THEN \*DEST[VL-1:k] remains unchanged\*

ELSE DEST[VL-1:k] ← 0

FI

DEST[MAX\_VL-1:VL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

VPCOMPRESSD \_\_m512i \_\_mm512\_mask\_compress\_epi32(\_\_m512i s, \_\_mmask16 c, \_\_m512i a);

VPCOMPRESSD \_\_m512i \_\_mm512\_maskz\_compress\_epi32(\_\_mmask16 c, \_\_m512i a);

VPCOMPRESSD void \_\_mm512\_mask\_compressstoreu\_epi32(void \* a, \_\_mmask16 c, \_\_m512i s);

VPCOMPRESSD \_\_m256i \_\_mm256\_mask\_compress\_epi32(\_\_m256i s, \_\_mmask8 c, \_\_m256i a);

VPCOMPRESSD \_\_m256i \_\_mm256\_maskz\_compress\_epi32(\_\_mmask8 c, \_\_m256i a);

VPCOMPRESSD void \_\_mm256\_mask\_compressstoreu\_epi32(void \* a, \_\_mmask8 c, \_\_m256i s);

VPCOMPRESSD \_\_m128i \_\_mm\_mask\_compress\_epi32(\_\_m128i s, \_\_mmask8 c, \_\_m128i a);

VPCOMPRESSD \_\_m128i \_\_mm\_maskz\_compress\_epi32(\_\_mmask8 c, \_\_m128i a);

VPCOMPRESSD void \_\_mm\_mask\_compressstoreu\_epi32(void \* a, \_\_mmask8 c, \_\_m128i s);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

EVEX-encoded instruction, see Exceptions Type E4.nb.

## VPCOMPRESSQ—Store Sparse Packed Quadword Integer Values into Dense Memory/Register

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W1 8B /r VPCOMPRESSQ xmm1/m128 {k1}{z}, xmm2	T1S	V/V	AVX512VL AVX512F	Compress packed quadword integer values from xmm2 to xmm1/m128 using controlmask k1.
EVEX.256.66.0F38.W1 8B /r VPCOMPRESSQ ymm1/m256 {k1}{z}, ymm2	T1S	V/V	AVX512VL AVX512F	Compress packed quadword integer values from ymm2 to ymm1/m256 using controlmask k1.
EVEX.512.66.0F38.W1 8B /r VPCOMPRESSQ zmm1/m512 {k1}{z}, zmm2	T1S	V/V	AVX512F	Compress packed quadword integer values from zmm2 to zmm1/m512 using controlmask k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
T1S	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

### Description

Compress (stores) up to 8/4/2 quadword integer values from the source operand (second operand) to the destination operand (first operand). The source operand is a ZMM/YMM/XMM register, the destination operand can be a ZMM/YMM/XMM register or a 512/256/128-bit memory location.

The opmask register k1 selects the active elements (partial vector or possibly non-contiguous if less than 8 active elements) from the source operand to compress into a contiguous vector. The contiguous vector is written to the destination starting from the low element of the destination operand.

Memory destination version: Only the contiguous vector is written to the destination memory location. EVEX.z must be zero.

Register destination version: If the vector length of the contiguous vector is less than that of the input vector in the source operand, the upper bits of the destination register are unmodified if EVEX.z is not set, otherwise the upper bits are zeroed.

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

Note that the compressed displacement assumes a pre-scaling (N) corresponding to the size of one single element instead of the size of the full vector.

### Operation

#### VPCOMPRESSQ (EVEX encoded versions) store form

(KL, VL) = (2, 128), (4, 256), (8, 512)

SIZE ← 64

k ← 0

FOR j ← 0 TO KL-1

  i ← j \* 64

  IF k1[j] OR \*no controlmask\*

    THEN

      DEST[k+SIZE-1:k] ← SRC[i+63:i]

      k ← k + SIZE

  FI;

ENFOR



**VPCOMPRESSQ (EVEX encoded versions) reg-reg form**

(KL, VL) = (2, 128), (4, 256), (8, 512)

SIZE ← 64

k ← 0

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no controlmask\*

THEN

DEST[k+SIZE-1:k] ← SRC[i+63:i]

k ← k + SIZE

FI;

ENDFOR

IF \*merging-masking\*

THEN \*DEST[VL-1:k] remains unchanged\*

ELSE DEST[VL-1:k] ← 0

FI

DEST[MAX\_VL-1:VL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

VPCOMPRESSQ \_\_m512i \_\_mm512\_mask\_compress\_epi64(\_\_m512i s, \_\_mmask8 c, \_\_m512i a);

VPCOMPRESSQ \_\_m512i \_\_mm512\_maskz\_compress\_epi64(\_\_mmask8 c, \_\_m512i a);

VPCOMPRESSQ void \_\_mm512\_mask\_compressstoreu\_epi64(void \* a, \_\_mmask8 c, \_\_m512i s);

VPCOMPRESSQ \_\_m256i \_\_mm256\_mask\_compress\_epi64(\_\_m256i s, \_\_mmask8 c, \_\_m256i a);

VPCOMPRESSQ \_\_m256i \_\_mm256\_maskz\_compress\_epi64(\_\_mmask8 c, \_\_m256i a);

VPCOMPRESSQ void \_\_mm256\_mask\_compressstoreu\_epi64(void \* a, \_\_mmask8 c, \_\_m256i s);

VPCOMPRESSQ \_\_m128i \_\_mm\_mask\_compress\_epi64(\_\_m128i s, \_\_mmask8 c, \_\_m128i a);

VPCOMPRESSQ \_\_m128i \_\_mm\_maskz\_compress\_epi64(\_\_mmask8 c, \_\_m128i a);

VPCOMPRESSQ void \_\_mm\_mask\_compressstoreu\_epi64(void \* a, \_\_mmask8 c, \_\_m128i s);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

EVEX-encoded instruction, see Exceptions Type E4.nb.

## VPCONFLICTD/Q—Detect Conflicts Within a Vector of Packed Dword/Qword Values into Dense Memory/ Register

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 C4 /r VPCONFLICTD xmm1 {k1}{z}, xmm2/m128/m32bcst	FV	V/V	AVX512VL AVX512CD	Detect duplicate double-word values in xmm2/m128/m32bcst using writemask k1.
EVEX.256.66.0F38.W0 C4 /r VPCONFLICTD ymm1 {k1}{z}, ymm2/m256/m32bcst	FV	V/V	AVX512VL AVX512CD	Detect duplicate double-word values in ymm2/m256/m32bcst using writemask k1.
EVEX.512.66.0F38.W0 C4 /r VPCONFLICTD zmm1 {k1}{z}, zmm2/m512/m32bcst	FV	V/V	AVX512CD	Detect duplicate double-word values in zmm2/m512/m32bcst using writemask k1.
EVEX.128.66.0F38.W1 C4 /r VPCONFLICTQ xmm1 {k1}{z}, xmm2/m128/m64bcst	FV	V/V	AVX512VL AVX512CD	Detect duplicate quad-word values in xmm2/m128/m64bcst using writemask k1.
EVEX.256.66.0F38.W1 C4 /r VPCONFLICTQ ymm1 {k1}{z}, ymm2/m256/m64bcst	FV	V/V	AVX512VL AVX512CD	Detect duplicate quad-word values in ymm2/m256/m64bcst using writemask k1.
EVEX.512.66.0F38.W1 C4 /r VPCONFLICTQ zmm1 {k1}{z}, zmm2/m512/m64bcst	FV	V/V	AVX512CD	Detect duplicate quad-word values in zmm2/m512/m64bcst using writemask k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
FV	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Test each dword/qword element of the source operand (the second operand) for equality with all other elements in the source operand closer to the least significant element. Each element's comparison results form a bit vector, which is then zero extended and written to the destination according to the writemask.

EVEX.512 encoded version: The source operand is a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

EVEX.256 encoded version: The source operand is a YMM register, a 256-bit memory location, or a 256-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a YMM register, conditionally updated using writemask k1.

EVEX.128 encoded version: The source operand is a XMM register, a 128-bit memory location, or a 128-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a XMM register, conditionally updated using writemask k1.

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

**Operation****VPCONFLICTD**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

i ← j\*32

IF MaskBit(j) OR \*no writemask\* THEN

FOR k ← 0 TO j-1

m ← k\*32

IF ((SRC[i+31:i] = SRC[m+31:m])) THEN

DEST[i+k] ← 1

ELSE

DEST[i+k] ← 0

FI

ENDFOR

DEST[i+31:i+j] ← 0

ELSE

IF \*merging-masking\* THEN

\*DEST[i+31:i] remains unchanged\*

ELSE

DEST[i+31:i] ← 0

FI

FI

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VPCONFLICTQ**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

i ← j\*64

IF MaskBit(j) OR \*no writemask\* THEN

FOR k ← 0 TO j-1

m ← k\*64

IF ((SRC[i+63:i] = SRC[m+63:m])) THEN

DEST[i+k] ← 1

ELSE

DEST[i+k] ← 0

FI

ENDFOR

DEST[i+63:i+j] ← 0

ELSE

IF \*merging-masking\* THEN

\*DEST[i+63:i] remains unchanged\*

ELSE

DEST[i+63:i] ← 0

FI

FI

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

VPCONFLICTD \_\_m512i \_mm512\_conflict\_epi32(\_\_m512i a);  
 VPCONFLICTD \_\_m512i \_mm512\_mask\_conflict\_epi32(\_\_m512i s, \_\_mmask16 m, \_\_m512i a);  
 VPCONFLICTD \_\_m512i \_mm512\_maskz\_conflict\_epi32(\_\_mmask16 m, \_\_m512i a);  
 VPCONFLICTQ \_\_m512i \_mm512\_conflict\_epi64(\_\_m512i a);  
 VPCONFLICTQ \_\_m512i \_mm512\_mask\_conflict\_epi64(\_\_m512i s, \_\_mmask8 m, \_\_m512i a);  
 VPCONFLICTQ \_\_m512i \_mm512\_maskz\_conflict\_epi64(\_\_mmask8 m, \_\_m512i a);  
 VPCONFLICTD \_\_m256i \_mm256\_conflict\_epi32(\_\_m256i a);  
 VPCONFLICTD \_\_m256i \_mm256\_mask\_conflict\_epi32(\_\_m256i s, \_\_mmask8 m, \_\_m256i a);  
 VPCONFLICTD \_\_m256i \_mm256\_maskz\_conflict\_epi32(\_\_mmask8 m, \_\_m256i a);  
 VPCONFLICTQ \_\_m256i \_mm256\_conflict\_epi64(\_\_m256i a);  
 VPCONFLICTQ \_\_m256i \_mm256\_mask\_conflict\_epi64(\_\_m256i s, \_\_mmask8 m, \_\_m256i a);  
 VPCONFLICTQ \_\_m256i \_mm256\_maskz\_conflict\_epi64(\_\_mmask8 m, \_\_m256i a);  
 VPCONFLICTD \_\_m128i \_mm\_conflict\_epi32(\_\_m128i a);  
 VPCONFLICTD \_\_m128i \_mm\_mask\_conflict\_epi32(\_\_m128i s, \_\_mmask8 m, \_\_m128i a);  
 VPCONFLICTD \_\_m128i \_mm\_maskz\_conflict\_epi32(\_\_mmask8 m, \_\_m128i a);  
 VPCONFLICTQ \_\_m128i \_mm\_conflict\_epi64(\_\_m128i a);  
 VPCONFLICTQ \_\_m128i \_mm\_mask\_conflict\_epi64(\_\_m128i s, \_\_mmask8 m, \_\_m128i a);  
 VPCONFLICTQ \_\_m128i \_mm\_maskz\_conflict\_epi64(\_\_mmask8 m, \_\_m128i a);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

EVEX-encoded instruction, see Exceptions Type E4.

## VPERMD/VPERMW—Permute Packed Doublewords/Words Elements

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.NDS.256.66.0F38.W0 36 /r VPERMD ymm1, ymm2, ymm3/m256	RVM	V/V	AVX2	Permute doublewords in ymm3/m256 using indices in ymm2 and store the result in ymm1.
EVEX.NDS.256.66.0F38.W0 36 /r VPERMD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	FV	V/V	AVX512VL AVX512F	Permute doublewords in ymm3/m256/m32bcst using indexes in ymm2 and store the result in ymm1 using writemask k1.
EVEX.NDS.512.66.0F38.W0 36 /r VPERMD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	FV	V/V	AVX512F	Permute doublewords in zmm3/m512/m32bcst using indices in zmm2 and store the result in zmm1 using writemask k1.
EVEX.NDS.128.66.0F38.W1 8D /r VPERMW xmm1 {k1}{z}, xmm2, xmm3/m128	FVM	V/V	AVX512VL AVX512BW	Permute word integers in xmm3/m128 using indexes in xmm2 and store the result in xmm1 using writemask k1.
EVEX.NDS.256.66.0F38.W1 8D /r VPERMW ymm1 {k1}{z}, ymm2, ymm3/m256	FVM	V/V	AVX512VL AVX512BW	Permute word integers in ymm3/m256 using indexes in ymm2 and store the result in ymm1 using writemask k1.
EVEX.NDS.512.66.0F38.W1 8D /r VPERMW zmm1 {k1}{z}, zmm2, zmm3/m512	FVM	V/V	AVX512BW	Permute word integers in zmm3/m512 using indexes in zmm2 and store the result in zmm1 using writemask k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RVM	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
FV	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA
FVM	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA

### Description

Copies doublewords (or words) from the second source operand (the third operand) to the destination operand (the first operand) according to the indices in the first source operand (the second operand). Note that this instruction permits a doubleword (word) in the source operand to be copied to more than one location in the destination operand.

VEX.256 encoded VPERMD: The first and second operands are YMM registers, the third operand can be a YMM register or memory location. Bits (MAX\_VL-1:256) of the corresponding destination register are zeroed.

EVEX encoded VPERMD: The first and second operands are ZMM/YMM registers, the third operand can be a ZMM/YMM register, a 512/256-bit memory location or a 512/256-bit vector broadcasted from a 32-bit memory location. The elements in the destination are updated using the writemask k1.

VPERMW: first and second operands are ZMM/YMM/XMM registers, the third operand can be a ZMM/YMM/XMM register, or a 512/256/128-bit memory location. The destination is updated using the writemask k1.

EVEX.128 encoded versions: Bits (MAX\_VL-1:128) of the corresponding ZMM register are zeroed.

**Operation****VPERMD (EVEX encoded versions)**

```

(KL, VL) = (8, 256), (16, 512)
IF VL = 256 THEN n ← 2; FI;
IF VL = 512 THEN n ← 3; FI;
FOR j ← 0 TO KL-1
  i ← j * 32
  id ← 32*SRC1[i+n:i]
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1) AND (SRC2 *is memory*)
        THEN DEST[i+31:i] ← SRC2[31:0];
        ELSE DEST[i+31:i] ← SRC2[id+31:id];
      FI;
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+31:i] remains unchanged*
        ELSE ; zeroing-masking
          DEST[i+31:i] ← 0
        FI
      FI;
    ENDFOR
  DEST[MAX_VL-1:VL] ← 0

```

**VPERMD (VEX.256 encoded version)**

```

DEST[31:0] ← (SRC2[255:0] >> (SRC1[2:0] * 32))[31:0];
DEST[63:32] ← (SRC2[255:0] >> (SRC1[34:32] * 32))[31:0];
DEST[95:64] ← (SRC2[255:0] >> (SRC1[66:64] * 32))[31:0];
DEST[127:96] ← (SRC2[255:0] >> (SRC1[98:96] * 32))[31:0];
DEST[159:128] ← (SRC2[255:0] >> (SRC1[130:128] * 32))[31:0];
DEST[191:160] ← (SRC2[255:0] >> (SRC1[162:160] * 32))[31:0];
DEST[223:192] ← (SRC2[255:0] >> (SRC1[194:192] * 32))[31:0];
DEST[255:224] ← (SRC2[255:0] >> (SRC1[226:224] * 32))[31:0];
DEST[MAX_VL-1:256] ← 0

```

**VPERMW (EVEX encoded versions)**

```

(KL, VL) = (8, 128), (16, 256), (32, 512)
IF VL = 128 THEN n ← 2; FI;
IF VL = 256 THEN n ← 3; FI;
IF VL = 512 THEN n ← 4; FI;
FOR j ← 0 TO KL-1
  i ← j * 16
  id ← 16*SRC1[i+n:i]
  IF k1[j] OR *no writemask*
    THEN DEST[i+15:i] ← SRC2[id+15:id]
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+15:i] remains unchanged*
        ELSE ; zeroing-masking
          DEST[i+15:i] ← 0
        FI
      FI;
    ENDFOR
  FI;
  ENDFOR
  DEST[MAX_VL-1:VL] ← 0

```

### Intel C/C++ Compiler Intrinsic Equivalent

```

VPERMD __m512i __mm512_permutexvar_epi32( __m512i idx, __m512i a);
VPERMD __m512i __mm512_mask_permutexvar_epi32(__m512i s, __mmask16 k, __m512i idx, __m512i a);
VPERMD __m512i __mm512_maskz_permutexvar_epi32(__mmask16 k, __m512i idx, __m512i a);
VPERMD __m256i __mm256_permutexvar_epi32( __m256i idx, __m256i a);
VPERMD __m256i __mm256_mask_permutexvar_epi32(__m256i s, __mmask8 k, __m256i idx, __m256i a);
VPERMD __m256i __mm256_maskz_permutexvar_epi32(__mmask8 k, __m256i idx, __m256i a);
VPERMW __m512i __mm512_permutexvar_epi16( __m512i idx, __m512i a);
VPERMW __m512i __mm512_mask_permutexvar_epi16(__m512i s, __mmask32 k, __m512i idx, __m512i a);
VPERMW __m512i __mm512_maskz_permutexvar_epi16(__mmask32 k, __m512i idx, __m512i a);
VPERMW __m256i __mm256_permutexvar_epi16( __m256i idx, __m256i a);
VPERMW __m256i __mm256_mask_permutexvar_epi16(__m256i s, __mmask16 k, __m256i idx, __m256i a);
VPERMW __m256i __mm256_maskz_permutexvar_epi16(__mmask16 k, __m256i idx, __m256i a);
VPERMW __m128i __mm_permutexvar_epi16( __m128i idx, __m128i a);
VPERMW __m128i __mm_mask_permutexvar_epi16(__m128i s, __mmask8 k, __m128i idx, __m128i a);
VPERMW __m128i __mm_maskz_permutexvar_epi16( __mmask8 k, __m128i idx, __m128i a);

```

### SIMD Floating-Point Exceptions

None

### Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded VPERMD, see Exceptions Type E4NF.

EVEX-encoded VPERMW, see Exceptions Type E4NF.nb.

#UD                    If VEX.L = 0.  
                           If EVEX.L'L = 0 for VPERMD.

## VPERMI2W/D/Q/PS/PD—Full Permute From Two Tables Overwriting the Index

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.DDS.128.66.0F38.W1 75 /r VPERMI2W xmm1 {k1}{z}, xmm2, xmm3/m128	FVM	V/V	AVX512VL AVX512BW	Permute word integers from two tables in xmm3/m128 and xmm2 using indexes in xmm1 and store the result in xmm1 using writemask k1.
EVEX.DDS.256.66.0F38.W1 75 /r VPERMI2W ymm1 {k1}{z}, ymm2, ymm3/m256	FVM	V/V	AVX512VL AVX512BW	Permute word integers from two tables in ymm3/m256 and ymm2 using indexes in ymm1 and store the result in ymm1 using writemask k1.
EVEX.DDS.512.66.0F38.W1 75 /r VPERMI2W zmm1 {k1}{z}, zmm2, zmm3/m512	FVM	V/V	AVX512BW	Permute word integers from two tables in zmm3/m512 and zmm2 using indexes in zmm1 and store the result in zmm1 using writemask k1.
EVEX.DDS.128.66.0F38.W0 76 /r VPERMI2D xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	FV	V/V	AVX512VL AVX512F	Permute double-words from two tables in xmm3/m128/m32bcst and xmm2 using indexes in xmm1 and store the result in xmm1 using writemask k1.
EVEX.DDS.256.66.0F38.W0 76 /r VPERMI2D ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	FV	V/V	AVX512VL AVX512F	Permute double-words from two tables in ymm3/m256/m32bcst and ymm2 using indexes in ymm1 and store the result in ymm1 using writemask k1.
EVEX.DDS.512.66.0F38.W0 76 /r VPERMI2D zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	FV	V/V	AVX512F	Permute double-words from two tables in zmm3/m512/m32bcst and zmm2 using indices in zmm1 and store the result in zmm1 using writemask k1.
EVEX.DDS.128.66.0F38.W1 76 /r VPERMI2Q xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	FV	V/V	AVX512VL AVX512F	Permute quad-words from two tables in xmm3/m128/m64bcst and xmm2 using indexes in xmm1 and store the result in xmm1 using writemask k1.
EVEX.DDS.256.66.0F38.W1 76 /r VPERMI2Q ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	FV	V/V	AVX512VL AVX512F	Permute quad-words from two tables in ymm3/m256/m64bcst and ymm2 using indexes in ymm1 and store the result in ymm1 using writemask k1.
EVEX.DDS.512.66.0F38.W1 76 /r VPERMI2Q zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	FV	V/V	AVX512F	Permute quad-words from two tables in zmm3/m512/m64bcst and zmm2 using indices in zmm1 and store the result in zmm1 using writemask k1.
EVEX.DDS.128.66.0F38.W0 77 /r VPERMI2PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	FV	V/V	AVX512VL AVX512F	Permute single-precision FP values from two tables in xmm3/m128/m32bcst and xmm2 using indexes in xmm1 and store the result in xmm1 using writemask k1.
EVEX.DDS.256.66.0F38.W0 77 /r VPERMI2PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	FV	V/V	AVX512VL AVX512F	Permute single-precision FP values from two tables in ymm3/m256/m32bcst and ymm2 using indexes in ymm1 and store the result in ymm1 using writemask k1.
EVEX.DDS.512.66.0F38.W0 77 /r VPERMI2PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	FV	V/V	AVX512F	Permute single-precision FP values from two tables in zmm3/m512/m32bcst and zmm2 using indices in zmm1 and store the result in zmm1 using writemask k1.



Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.DDS.128.66.0F38.W1 77 /r VPERMI2PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	FV	V/V	AVX512VL AVX512F	Permute double-precision FP values from two tables in xmm3/m128/m64bcst and xmm2 using indexes in xmm1 and store the result in xmm1 using writemask k1.
EVEX.DDS.256.66.0F38.W1 77 /r VPERMI2PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	FV	V/V	AVX512VL AVX512F	Permute double-precision FP values from two tables in ymm3/m256/m64bcst and ymm2 using indexes in ymm1 and store the result in ymm1 using writemask k1.
EVEX.DDS.512.66.0F38.W1 77 /r VPERMI2PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	FV	V/V	AVX512F	Permute double-precision FP values from two tables in zmm3/m512/m64bcst and zmm2 using indices in zmm1 and store the result in zmm1 using writemask k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
FVM	ModRM:reg (r,w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA
FV	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

#### Description

Permutates 16-bit/32-bit/64-bit values in the second operand (the first source operand) and the third operand (the second source operand) using indices in the first operand to select elements from the second and third operands. The selected elements are written to the destination operand (the first operand) according to the writemask k1.

The first and second operands are ZMM/YMM/XMM registers. The first operand contains input indices to select elements from the two input tables in the 2nd and 3rd operands. The first operand is also the destination of the result.

D/Q/PS/PD element versions: 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 32/64-bit memory location. Broadcast from the low 32/64-bit memory location is performed if EVEX.b and the id bit for table selection are set (selecting table\_2).

Dword/PS versions: The id bit for table selection is bit 4/3/2, depending on VL=512, 256, 128. Bits [3:0]/[2:0]/[1:0] of each element in the input index vector select an element within the two source operands, If the id bit is 0, table\_1 (the first source) is selected; otherwise the second source operand is selected.

Qword/PD versions: The id bit for table selection is bit 3/2/1, and bits [2:0]/[1:0] /bit 0 selects element within each input table.

Word element versions: The second source operand can be a ZMM/YMM/XMM register, or a 512/256/128-bit memory location. The id bit for table selection is bit 5/4/3, and bits [4:0]/[3:0]/[2:0] selects element within each input table.

Note that these instructions permit a 16-bit/32-bit/64-bit value in the source operands to be copied to more than one location in the destination operand. Note also that in this case, the same table can be reused for example for a second iteration, while the index elements are overwritten.

Bits (MAX\_VL-1:256/128) of the destination are zeroed for VL=256,128.

**Operation****VPERMI2W (EVEX encoded versions)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

IF VL = 128

id ← 2

FI;

IF VL = 256

id ← 3

FI;

IF VL = 512

id ← 4

FI;

TMP\_DEST ← DEST

FOR j ← 0 TO KL-1

i ← j \* 16

off ← 16 \* TMP\_DEST[i+id:i]

IF k1[j] OR \*no writemask\*

THEN

DEST[i+15:i] = TMP\_DEST[i+id+1] ? SRC2[off+15:off]  
: SRC1[off+15:off]

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+15:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+15:i] ← 0

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VPERMI2D/VPERMI2PS (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

IF VL = 128

id ← 1

FI;

IF VL = 256

id ← 2

FI;

IF VL = 512

id ← 3

FI;

TMP\_DEST ← DEST

FOR j ← 0 TO KL-1

i ← j \* 32

off ← 32 \* TMP\_DEST[i+id:i]

IF k1[j] OR \*no writemask\*

THEN

IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

THEN

DEST[i+31:i] ← TMP\_DEST[i+id+1] ? SRC2[31:0]  
: SRC1[off+31:off]

ELSE

DEST[i+31:i] ← TMP\_DEST[i+id+1] ? SRC2[off+31:off]  
: SRC1[off+31:off]

```

        FI
    ELSE
        IF *merging-masking*           ; merging-masking
            THEN *DEST[i+31:i] remains unchanged*
        ELSE                             ; zeroing-masking
            DEST[i+31:i] ← 0
        FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

#### VPERMI2Q/VPERMI2PD (EVEX encoded versions)

(KL, VL) = (2, 128), (4, 256), (8, 512)

IF VL = 128

id ← 0

FI;

IF VL = 256

id ← 1

FI;

IF VL = 512

id ← 2

FI;

TMP\_DEST ← DEST

FOR j ← 0 TO KL-1

i ← j \* 64

off ← 64 \* TMP\_DEST[i+id:i]

IF k1[j] OR \*no writemask\*

THEN

IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

THEN

DEST[i+63:i] ← TMP\_DEST[i+id+1] ? SRC2[63:0]

: SRC1[off+63:off]

ELSE

DEST[i+63:i] ← TMP\_DEST[i+id+1] ? SRC2[off+63:off]

: SRC1[off+63:off]

FI

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

#### Intel C/C++ Compiler Intrinsic Equivalent

VPERMI2D \_\_m512i \_mm512\_permutex2var\_epi32(\_\_m512i a, \_\_m512i idx, \_\_m512i b);

VPERMI2D \_\_m512i \_mm512\_mask\_permutex2var\_epi32(\_\_m512i a, \_\_mmask16 k, \_\_m512i idx, \_\_m512i b);

VPERMI2D \_\_m512i \_mm512\_mask2\_permutex2var\_epi32(\_\_m512i a, \_\_m512i idx, \_\_mmask16 k, \_\_m512i b);

VPERMI2D \_\_m512i \_mm512\_maskz\_permutex2var\_epi32(\_\_mmask16 k, \_\_m512i a, \_\_m512i idx, \_\_m512i b);

VPERMI \_\_m256i \_mm256\_permutex2var\_epi32(\_\_m256i a, \_\_m256i idx, \_\_m256i b);

VPERMI2D \_\_m256i \_mm256\_mask\_permutex2var\_epi32(\_\_m256i a, \_\_mmask8 k, \_\_m256i idx, \_\_m256i b);

VPERMI2D \_\_m256i \_\_mm256\_mask2\_permutex2var\_epi32(\_\_m256i a, \_\_m256i idx, \_\_mmask8 k, \_\_m256i b);  
 VPERMI2D \_\_m256i \_\_mm256\_maskz\_permutex2var\_epi32(\_\_mmask8 k, \_\_m256i a, \_\_m256i idx, \_\_m256i b);  
 VPERMI2D \_\_m128i \_\_mm\_permutex2var\_epi32(\_\_m128i a, \_\_m128i idx, \_\_m128i b);  
 VPERMI2D \_\_m128i \_\_mm\_mask\_permutex2var\_epi32(\_\_m128i a, \_\_mmask8 k, \_\_m128i idx, \_\_m128i b);  
 VPERMI2D \_\_m128i \_\_mm\_mask2\_permutex2var\_epi32(\_\_m128i a, \_\_m128i idx, \_\_mmask8 k, \_\_m128i b);  
 VPERMI2D \_\_m128i \_\_mm\_maskz\_permutex2var\_epi32(\_\_mmask8 k, \_\_m128i a, \_\_m128i idx, \_\_m128i b);  
 VPERMI2PD \_\_m512d \_\_mm512\_permutex2var\_pd(\_\_m512d a, \_\_m512i idx, \_\_m512d b);  
 VPERMI2PD \_\_m512d \_\_mm512\_mask\_permutex2var\_pd(\_\_m512d a, \_\_mmask8 k, \_\_m512i idx, \_\_m512d b);  
 VPERMI2PD \_\_m512d \_\_mm512\_mask2\_permutex2var\_pd(\_\_m512d a, \_\_m512i idx, \_\_mmask8 k, \_\_m512d b);  
 VPERMI2PD \_\_m512d \_\_mm512\_maskz\_permutex2var\_pd(\_\_mmask8 k, \_\_m512d a, \_\_m512i idx, \_\_m512d b);  
 VPERMI2PD \_\_m256d \_\_mm256\_permutex2var\_pd(\_\_m256d a, \_\_m256i idx, \_\_m256d b);  
 VPERMI2PD \_\_m256d \_\_mm256\_mask\_permutex2var\_pd(\_\_m256d a, \_\_mmask8 k, \_\_m256i idx, \_\_m256d b);  
 VPERMI2PD \_\_m256d \_\_mm256\_mask2\_permutex2var\_pd(\_\_m256d a, \_\_m256i idx, \_\_mmask8 k, \_\_m256d b);  
 VPERMI2PD \_\_m256d \_\_mm256\_maskz\_permutex2var\_pd(\_\_mmask8 k, \_\_m256d a, \_\_m256i idx, \_\_m256d b);  
 VPERMI2PD \_\_m128d \_\_mm\_permutex2var\_pd(\_\_m128d a, \_\_m128i idx, \_\_m128d b);  
 VPERMI2PD \_\_m128d \_\_mm\_mask\_permutex2var\_pd(\_\_m128d a, \_\_mmask8 k, \_\_m128i idx, \_\_m128d b);  
 VPERMI2PD \_\_m128d \_\_mm\_mask2\_permutex2var\_pd(\_\_m128d a, \_\_m128i idx, \_\_mmask8 k, \_\_m128d b);  
 VPERMI2PD \_\_m128d \_\_mm\_maskz\_permutex2var\_pd(\_\_mmask8 k, \_\_m128d a, \_\_m128i idx, \_\_m128d b);  
 VPERMI2PS \_\_m512 \_\_mm512\_permutex2var\_ps(\_\_m512 a, \_\_m512i idx, \_\_m512 b);  
 VPERMI2PS \_\_m512 \_\_mm512\_mask\_permutex2var\_ps(\_\_m512 a, \_\_mmask16 k, \_\_m512i idx, \_\_m512 b);  
 VPERMI2PS \_\_m512 \_\_mm512\_mask2\_permutex2var\_ps(\_\_m512 a, \_\_m512i idx, \_\_mmask16 k, \_\_m512 b);  
 VPERMI2PS \_\_m512 \_\_mm512\_maskz\_permutex2var\_ps(\_\_mmask16 k, \_\_m512 a, \_\_m512i idx, \_\_m512 b);  
 VPERMI2PS \_\_m256 \_\_mm256\_permutex2var\_ps(\_\_m256 a, \_\_m256i idx, \_\_m256 b);  
 VPERMI2PS \_\_m256 \_\_mm256\_mask\_permutex2var\_ps(\_\_m256 a, \_\_mmask8 k, \_\_m256i idx, \_\_m256 b);  
 VPERMI2PS \_\_m256 \_\_mm256\_mask2\_permutex2var\_ps(\_\_m256 a, \_\_m256i idx, \_\_mmask8 k, \_\_m256 b);  
 VPERMI2PS \_\_m256 \_\_mm256\_maskz\_permutex2var\_ps(\_\_mmask8 k, \_\_m256 a, \_\_m256i idx, \_\_m256 b);  
 VPERMI2PS \_\_m128 \_\_mm\_permutex2var\_ps(\_\_m128 a, \_\_m128i idx, \_\_m128 b);  
 VPERMI2PS \_\_m128 \_\_mm\_mask\_permutex2var\_ps(\_\_m128 a, \_\_mmask8 k, \_\_m128i idx, \_\_m128 b);  
 VPERMI2PS \_\_m128 \_\_mm\_mask2\_permutex2var\_ps(\_\_m128 a, \_\_m128i idx, \_\_mmask8 k, \_\_m128 b);  
 VPERMI2PS \_\_m128 \_\_mm\_maskz\_permutex2var\_ps(\_\_mmask8 k, \_\_m128 a, \_\_m128i idx, \_\_m128 b);  
 VPERMI2Q \_\_m512i \_\_mm512\_permutex2var\_epi64(\_\_m512i a, \_\_m512i idx, \_\_m512i b);  
 VPERMI2Q \_\_m512i \_\_mm512\_mask\_permutex2var\_epi64(\_\_m512i a, \_\_mmask8 k, \_\_m512i idx, \_\_m512i b);  
 VPERMI2Q \_\_m512i \_\_mm512\_mask2\_permutex2var\_epi64(\_\_m512i a, \_\_m512i idx, \_\_mmask8 k, \_\_m512i b);  
 VPERMI2Q \_\_m512i \_\_mm512\_maskz\_permutex2var\_epi64(\_\_mmask8 k, \_\_m512i a, \_\_m512i idx, \_\_m512i b);  
 VPERMI2Q \_\_m256i \_\_mm256\_permutex2var\_epi64(\_\_m256i a, \_\_m256i idx, \_\_m256i b);  
 VPERMI2Q \_\_m256i \_\_mm256\_mask\_permutex2var\_epi64(\_\_m256i a, \_\_mmask8 k, \_\_m256i idx, \_\_m256i b);  
 VPERMI2Q \_\_m256i \_\_mm256\_mask2\_permutex2var\_epi64(\_\_m256i a, \_\_m256i idx, \_\_mmask8 k, \_\_m256i b);  
 VPERMI2Q \_\_m256i \_\_mm256\_maskz\_permutex2var\_epi64(\_\_mmask8 k, \_\_m256i a, \_\_m256i idx, \_\_m256i b);  
 VPERMI2Q \_\_m128i \_\_mm\_permutex2var\_epi64(\_\_m128i a, \_\_m128i idx, \_\_m128i b);  
 VPERMI2Q \_\_m128i \_\_mm\_mask\_permutex2var\_epi64(\_\_m128i a, \_\_mmask8 k, \_\_m128i idx, \_\_m128i b);  
 VPERMI2Q \_\_m128i \_\_mm\_mask2\_permutex2var\_epi64(\_\_m128i a, \_\_m128i idx, \_\_mmask8 k, \_\_m128i b);  
 VPERMI2Q \_\_m128i \_\_mm\_maskz\_permutex2var\_epi64(\_\_mmask8 k, \_\_m128i a, \_\_m128i idx, \_\_m128i b);  
 VPERMI2W \_\_m512i \_\_mm512\_permutex2var\_epi16(\_\_m512i a, \_\_m512i idx, \_\_m512i b);  
 VPERMI2W \_\_m512i \_\_mm512\_mask\_permutex2var\_epi16(\_\_m512i a, \_\_mmask32 k, \_\_m512i idx, \_\_m512i b);  
 VPERMI2W \_\_m512i \_\_mm512\_mask2\_permutex2var\_epi16(\_\_m512i a, \_\_m512i idx, \_\_mmask32 k, \_\_m512i b);  
 VPERMI2W \_\_m512i \_\_mm512\_maskz\_permutex2var\_epi16(\_\_mmask32 k, \_\_m512i a, \_\_m512i idx, \_\_m512i b);  
 VPERMI2W \_\_m256i \_\_mm256\_permutex2var\_epi16(\_\_m256i a, \_\_m256i idx, \_\_m256i b);  
 VPERMI2W \_\_m256i \_\_mm256\_mask\_permutex2var\_epi16(\_\_m256i a, \_\_mmask16 k, \_\_m256i idx, \_\_m256i b);  
 VPERMI2W \_\_m256i \_\_mm256\_mask2\_permutex2var\_epi16(\_\_m256i a, \_\_m256i idx, \_\_mmask16 k, \_\_m256i b);

VPERMI2W \_\_m256i \_mm256\_maskz\_permutex2var\_epi16(\_\_mmask16 k, \_\_m256i a, \_\_m256i idx, \_\_m256i b);

VPERMI2W \_\_m128i \_mm\_permutex2var\_epi16(\_\_m128i a, \_\_m128i idx, \_\_m128i b);

VPERMI2W \_\_m128i \_mm\_mask\_permutex2var\_epi16(\_\_m128i a, \_\_mmask8 k, \_\_m128i idx, \_\_m128i b);

VPERMI2W \_\_m128i \_mm\_mask2\_permutex2var\_epi16(\_\_m128i a, \_\_m128i idx, \_\_mmask8 k, \_\_m128i b);

VPERMI2W \_\_m128i \_mm\_maskz\_permutex2var\_epi16(\_\_mmask8 k, \_\_m128i a, \_\_m128i idx, \_\_m128i b);

### SIMD Floating-Point Exceptions

None

### Other Exceptions

VPERMI2D/Q/PS/PD: See Exceptions Type E4NF.

VPERMI2W: See Exceptions Type E4NF.nb.

## VPERMILPD—Permute In-Lane of Pairs of Double-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.NDS.128.66.0F38.W0 0D /r VPERMILPD xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Permute double-precision floating-point values in xmm2 using controls from xmm3/m128 and store result in xmm1.
VEX.NDS.256.66.0F38.W0 0D /r VPERMILPD ymm1, ymm2, ymm3/m256	RVM	V/V	AVX	Permute double-precision floating-point values in ymm2 using controls from ymm3/m256 and store result in ymm1.
EVEX.NDS.128.66.0F38.W1 0D /r VPERMILPD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	FV-RVM	V/V	AVX512VL AVX512F	Permute double-precision floating-point values in xmm2 using control from xmm3/m128/m64bcst and store the result in xmm1 using writemask k1.
EVEX.NDS.256.66.0F38.W1 0D /r VPERMILPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	FV-RVM	V/V	AVX512VL AVX512F	Permute double-precision floating-point values in ymm2 using control from ymm3/m256/m64bcst and store the result in ymm1 using writemask k1.
EVEX.NDS.512.66.0F38.W1 0D /r VPERMILPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	FV-RVM	V/V	AVX512F	Permute double-precision floating-point values in zmm2 using control from zmm3/m512/m64bcst and store the result in zmm1 using writemask k1.
VEX.128.66.0F3A.W0 05 /r ib VPERMILPD xmm1, xmm2/m128, imm8	RM	V/V	AVX	Permute double-precision floating-point values in xmm2/m128 using controls from imm8.
VEX.256.66.0F3A.W0 05 /r ib VPERMILPD ymm1, ymm2/m256, imm8	RM	V/V	AVX	Permute double-precision floating-point values in ymm2/m256 using controls from imm8.
EVEX.128.66.0F3A.W1 05 /r ib VPERMILPD xmm1 {k1}{z}, xmm2/m128/m64bcst, imm8	FV-RM	V/V	AVX512VL AVX512F	Permute double-precision floating-point values in xmm2/m128/m64bcst using controls from imm8 and store the result in xmm1 using writemask k1.
EVEX.256.66.0F3A.W1 05 /r ib VPERMILPD ymm1 {k1}{z}, ymm2/m256/m64bcst, imm8	FV-RM	V/V	AVX512VL AVX512F	Permute double-precision floating-point values in ymm2/m256/m64bcst using controls from imm8 and store the result in ymm1 using writemask k1.
EVEX.512.66.0F3A.W1 05 /r ib VPERMILPD zmm1 {k1}{z}, zmm2/m512/m64bcst, imm8	FV-RM	V/V	AVX512F	Permute double-precision floating-point values in zmm2/m512/m64bcst using controls from imm8 and store the result in zmm1 using writemask k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
FV-RVM	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA
FV-RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

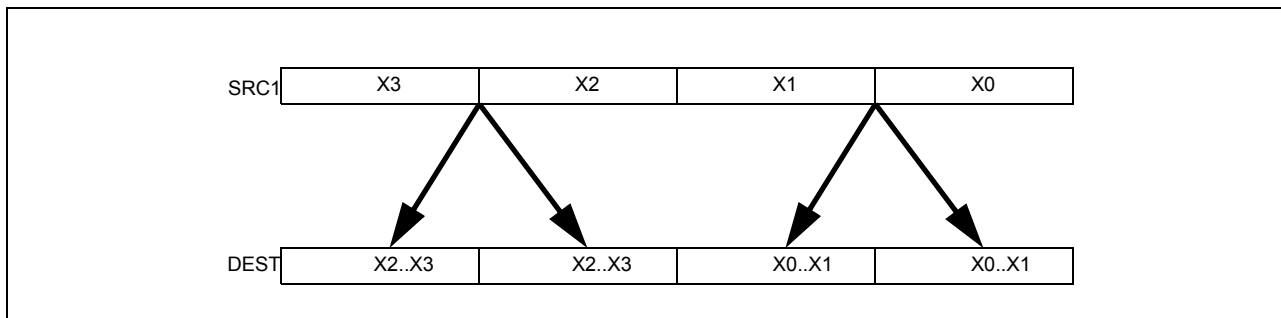
**Description**

(variable control version)

Permute pairs of double-precision floating-point values in the first source operand (second operand), each using a 1-bit control field residing in the corresponding quadword element of the second source operand (third operand). Permuted results are stored in the destination operand (first operand).

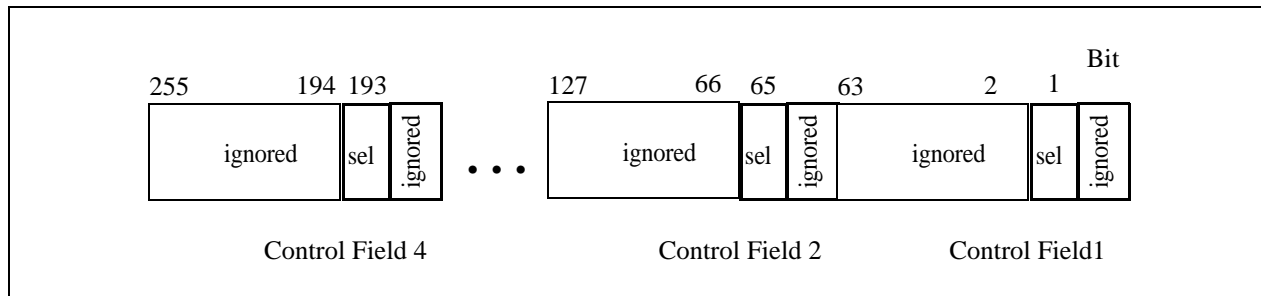
The control bits are located at bit 0 of each quadword element (see Figure 5-22). Each control determines which of the source element in an input pair is selected for the destination element. Each pair of source elements must lie in the same 128-bit region as the destination.

EVEX version: The second source operand (third operand) is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. Permuted results are written to the destination under the writemask.



**Figure 5-21. VPERMILPD Operation**

VEX.256 encoded version: Bits (MAX\_VL-1:256) of the corresponding ZMM register are zeroed.



**Figure 5-22. VPERMILPD Shuffle Control**

(immediate control version)

Permute pairs of double-precision floating-point values in the first source operand (second operand), each pair using a 1-bit control field in the imm8 byte. Each element in the destination operand (first operand) use a separate control bit of the imm8 byte.

VEX version: The source operand is a YMM/XMM register or a 256/128-bit memory location and the destination operand is a YMM/XMM register. Imm8 byte provides the lower 4/2 bit as permute control fields.

EVEX version: The source operand (second operand) is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. Permuted results are written to the destination under the writemask. Imm8 byte provides the lower 8/4/2 bit as permute control fields.

Note: For the imm8 versions, VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instruction will #UD.

**Operation****VPERMILPD (EVEX immediate versions)**

(KL, VL) = (8, 512)

FOR j ← 0 TO KL-1

i ← j \* 64

IF (EVEX.b = 1) AND (SRC1 \*is memory\*)

THEN TMP\_SRC1[i+63:i] ← SRC1[63:0];

ELSE TMP\_SRC1[i+63:i] ← SRC1[i+63:i];

FI;

ENDFOR;

IF (imm8[0] = 0) THEN TMP\_DEST[63:0] ← SRC1[63:0]; FI;

IF (imm8[0] = 1) THEN TMP\_DEST[63:0] ← TMP\_SRC1[127:64]; FI;

IF (imm8[1] = 0) THEN TMP\_DEST[127:64] ← TMP\_SRC1[63:0]; FI;

IF (imm8[1] = 1) THEN TMP\_DEST[127:64] ← TMP\_SRC1[127:64]; FI;

IF VL ≥ 256

IF (imm8[2] = 0) THEN TMP\_DEST[191:128] ← TMP\_SRC1[191:128]; FI;

IF (imm8[2] = 1) THEN TMP\_DEST[191:128] ← TMP\_SRC1[255:192]; FI;

IF (imm8[3] = 0) THEN TMP\_DEST[255:192] ← TMP\_SRC1[191:128]; FI;

IF (imm8[3] = 1) THEN TMP\_DEST[255:192] ← TMP\_SRC1[255:192]; FI;

FI;

IF VL ≥ 512

IF (imm8[4] = 0) THEN TMP\_DEST[319:256] ← TMP\_SRC1[319:256]; FI;

IF (imm8[4] = 1) THEN TMP\_DEST[319:256] ← TMP\_SRC1[383:320]; FI;

IF (imm8[5] = 0) THEN TMP\_DEST[383:320] ← TMP\_SRC1[319:256]; FI;

IF (imm8[5] = 1) THEN TMP\_DEST[383:320] ← TMP\_SRC1[383:320]; FI;

IF (imm8[6] = 0) THEN TMP\_DEST[447:384] ← TMP\_SRC1[447:384]; FI;

IF (imm8[6] = 1) THEN TMP\_DEST[447:384] ← TMP\_SRC1[511:448]; FI;

IF (imm8[7] = 0) THEN TMP\_DEST[511:448] ← TMP\_SRC1[447:384]; FI;

IF (imm8[7] = 1) THEN TMP\_DEST[511:448] ← TMP\_SRC1[511:448]; FI;

FI;

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN DEST[i+63:i] ← TMP\_DEST[i+63:i]

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VPERMILPD (256-bit immediate version)**

IF (imm8[0] = 0) THEN DEST[63:0] ← SRC1[63:0]

IF (imm8[0] = 1) THEN DEST[63:0] ← SRC1[127:64]

IF (imm8[1] = 0) THEN DEST[127:64] ← SRC1[63:0]

IF (imm8[1] = 1) THEN DEST[127:64] ← SRC1[127:64]

IF (imm8[2] = 0) THEN DEST[191:128] ← SRC1[191:128]

IF (imm8[2] = 1) THEN DEST[191:128] ← SRC1[255:192]

IF (imm8[3] = 0) THEN DEST[255:192] ← SRC1[191:128]

IF (imm8[3] = 1) THEN DEST[255:192] ← SRC1[255:192]

DEST[MAX\_VL-1:256] ← 0



**VPERMILPD (128-bit immediate version)**

```

IF (imm8[0] = 0) THEN DEST[63:0] ← SRC1[63:0]
IF (imm8[0] = 1) THEN DEST[63:0] ← SRC1[127:64]
IF (imm8[1] = 0) THEN DEST[127:64] ← SRC1[63:0]
IF (imm8[1] = 1) THEN DEST[127:64] ← SRC1[127:64]
DEST[MAX_VL-1:128] ← 0

```

**VPERMILPD (EVEX variable versions)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

```
FOR j ← 0 TO KL-1
```

```
  i ← j * 64
```

```
  IF (EVEX.b = 1) AND (SRC2 *is memory*)
    THEN TMP_SRC2[i+63:i] ← SRC2[63:0];
    ELSE TMP_SRC2[i+63:i] ← SRC2[i+63:i];

```

```
  FI;
```

```
ENDFOR;
```

```

IF (TMP_SRC2[1] = 0) THEN TMP_DEST[63:0] ← SRC1[63:0]; FI;
IF (TMP_SRC2[1] = 1) THEN TMP_DEST[63:0] ← SRC1[127:64]; FI;
IF (TMP_SRC2[65] = 0) THEN TMP_DEST[127:64] ← SRC1[63:0]; FI;
IF (TMP_SRC2[65] = 1) THEN TMP_DEST[127:64] ← SRC1[127:64]; FI;
IF VL >= 256
  IF (TMP_SRC2[129] = 0) THEN TMP_DEST[191:128] ← SRC1[191:128]; FI;
  IF (TMP_SRC2[129] = 1) THEN TMP_DEST[191:128] ← SRC1[255:192]; FI;
  IF (TMP_SRC2[193] = 0) THEN TMP_DEST[255:192] ← SRC1[191:128]; FI;
  IF (TMP_SRC2[193] = 1) THEN TMP_DEST[255:192] ← SRC1[255:192]; FI;
FI;
IF VL >= 512
  IF (TMP_SRC2[257] = 0) THEN TMP_DEST[319:256] ← SRC1[319:256]; FI;
  IF (TMP_SRC2[257] = 1) THEN TMP_DEST[319:256] ← SRC1[383:320]; FI;
  IF (TMP_SRC2[321] = 0) THEN TMP_DEST[383:320] ← SRC1[319:256]; FI;
  IF (TMP_SRC2[321] = 1) THEN TMP_DEST[383:320] ← SRC1[383:320]; FI;
  IF (TMP_SRC2[385] = 0) THEN TMP_DEST[447:384] ← SRC1[447:384]; FI;
  IF (TMP_SRC2[385] = 1) THEN TMP_DEST[447:384] ← SRC1[511:448]; FI;
  IF (TMP_SRC2[449] = 0) THEN TMP_DEST[511:448] ← SRC1[447:384]; FI;
  IF (TMP_SRC2[449] = 1) THEN TMP_DEST[511:448] ← SRC1[511:448]; FI;
FI;

```

```
FOR j ← 0 TO KL-1
```

```
  i ← j * 64
```

```
  IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+63:i] remains unchanged*
      ELSE ; zeroing-masking
        DEST[i+63:i] ← 0
      FI
    FI

```

```
  FI;
```

```
ENDFOR
```

```
DEST[MAX_VL-1:VL] ← 0
```

**VPERMILPD (256-bit variable version)**

```

IF (SRC2[1] = 0) THEN DEST[63:0] ← SRC1[63:0]
IF (SRC2[1] = 1) THEN DEST[63:0] ← SRC1[127:64]
IF (SRC2[65] = 0) THEN DEST[127:64] ← SRC1[63:0]
IF (SRC2[65] = 1) THEN DEST[127:64] ← SRC1[127:64]
IF (SRC2[129] = 0) THEN DEST[191:128] ← SRC1[191:128]
IF (SRC2[129] = 1) THEN DEST[191:128] ← SRC1[255:192]
IF (SRC2[193] = 0) THEN DEST[255:192] ← SRC1[191:128]
IF (SRC2[193] = 1) THEN DEST[255:192] ← SRC1[255:192]
DEST[MAX_VL-1:256] ← 0

```

**VPERMILPD (128-bit variable version)**

```

IF (SRC2[1] = 0) THEN DEST[63:0] ← SRC1[63:0]
IF (SRC2[1] = 1) THEN DEST[63:0] ← SRC1[127:64]
IF (SRC2[65] = 0) THEN DEST[127:64] ← SRC1[63:0]
IF (SRC2[65] = 1) THEN DEST[127:64] ← SRC1[127:64]
DEST[MAX_VL-1:128] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VPERMILPD __m512d __mm512_permute_pd( __m512d a, int imm);
VPERMILPD __m512d __mm512_mask_permute_pd( __m512d s, __mmask8 k, __m512d a, int imm);
VPERMILPD __m512d __mm512_maskz_permute_pd( __mmask8 k, __m512d a, int imm);
VPERMILPD __m256d __mm256_mask_permute_pd( __m256d s, __mmask8 k, __m256d a, int imm);
VPERMILPD __m256d __mm256_maskz_permute_pd( __mmask8 k, __m256d a, int imm);
VPERMILPD __m128d __mm_mask_permute_pd( __m128d s, __mmask8 k, __m128d a, int imm);
VPERMILPD __m128d __mm_maskz_permute_pd( __mmask8 k, __m128d a, int imm);
VPERMILPD __m512d __mm512_permutevar_pd( __m512i i, __m512d a);
VPERMILPD __m512d __mm512_mask_permutevar_pd( __m512d s, __mmask8 k, __m512i i, __m512d a);
VPERMILPD __m512d __mm512_maskz_permutevar_pd( __mmask8 k, __m512i i, __m512d a);
VPERMILPD __m256d __mm256_mask_permutevar_pd( __m256d s, __mmask8 k, __m256d i, __m256d a);
VPERMILPD __m256d __mm256_maskz_permutevar_pd( __mmask8 k, __m256d i, __m256d a);
VPERMILPD __m128d __mm_mask_permutevar_pd( __m128d s, __mmask8 k, __m128d i, __m128d a);
VPERMILPD __m128d __mm_maskz_permutevar_pd( __mmask8 k, __m128d i, __m128d a);
VPERMILPD __m128d __mm_permute_pd( __m128d a, int control)
VPERMILPD __m256d __mm256_permute_pd( __m256d a, int control)
VPERMILPD __m128d __mm_permutevar_pd( __m128d a, __m128i control);
VPERMILPD __m256d __mm256_permutevar_pd( __m256d a, __m256i control);

```

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 4; additionally

#UD If VEX.W = 1.

EVEX-encoded instruction, see Exceptions Type E4NF.

#UD If either (E)VEX.vvvv != 1111B and with imm8.

## VPERMILPS—Permute In-Lane of Quadruples of Single-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.NDS.128.66.0F3A.W0 0C /r VPERMILPS xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Permute single-precision floating-point values in xmm2 using controls from xmm3/m128 and store result in xmm1.
VEX.128.66.0F3A.W0 04 /r ib VPERMILPS xmm1, xmm2/m128, imm8	RM	V/V	AVX	Permute single-precision floating-point values in xmm2/m128 using controls from imm8 and store result in xmm1.
VEX.NDS.256.66.0F3A.W0 0C /r VPERMILPS ymm1, ymm2, ymm3/m256	RVM	V/V	AVX	Permute single-precision floating-point values in ymm2 using controls from ymm3/m256 and store result in ymm1.
VEX.256.66.0F3A.W0 04 /r ib VPERMILPS ymm1, ymm2/m256, imm8	RM	V/V	AVX	Permute single-precision floating-point values in ymm2/m256 using controls from imm8 and store result in ymm1.
EVEX.NDS.128.66.0F3A.W0 0C /r VPERMILPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	FV-RVM	V/V	AVX512VL AVX512F	Permute single-precision floating-point values xmm2 using control from xmm3/m128/m32bcst and store the result in xmm1 using writemask k1.
EVEX.NDS.256.66.0F3A.W0 0C /r VPERMILPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	FV-RVM	V/V	AVX512VL AVX512F	Permute single-precision floating-point values ymm2 using control from ymm3/m256/m32bcst and store the result in ymm1 using writemask k1.
EVEX.NDS.512.66.0F3A.W0 0C /r VPERMILPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	FV-RVM	V/V	AVX512F	Permute single-precision floating-point values zmm2 using control from zmm3/m512/m32bcst and store the result in zmm1 using writemask k1.
EVEX.128.66.0F3A.W0 04 /r ib VPERMILPS xmm1 {k1}{z}, xmm2/m128/m32bcst, imm8	FV-RM	V/V	AVX512VL AVX512F	Permute single-precision floating-point values xmm2/m128/m32bcst using controls from imm8 and store the result in xmm1 using writemask k1.
EVEX.256.66.0F3A.W0 04 /r ib VPERMILPS ymm1 {k1}{z}, ymm2/m256/m32bcst, imm8	FV-RM	V/V	AVX512VL AVX512F	Permute single-precision floating-point values ymm2/m256/m32bcst using controls from imm8 and store the result in ymm1 using writemask k1.
EVEX.512.66.0F3A.W0 04 /r ib VPERMILPS zmm1 {k1}{z}, zmm2/m512/m32bcst, imm8	FV-RM	V/V	AVX512F	Permute single-precision floating-point values zmm2/m512/m32bcst using controls from imm8 and store the result in zmm1 using writemask k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
FV-RVM	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA
FV-RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

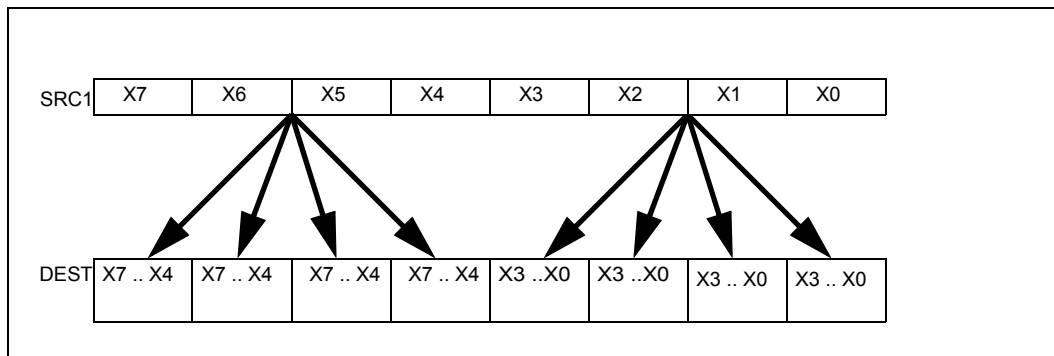
**Description**

(variable control version)

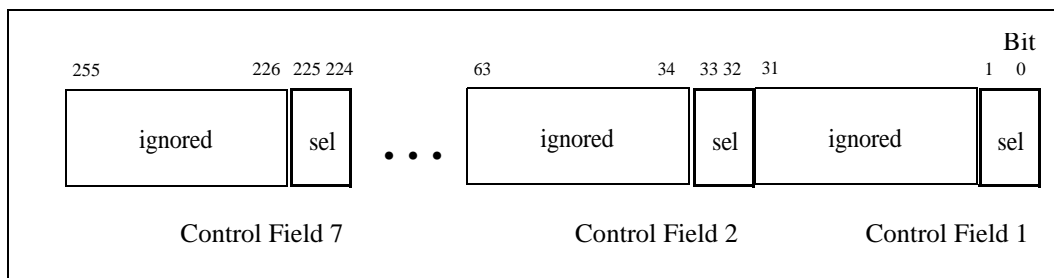
Permute quadruples of single-precision floating-point values in the first source operand (second operand), each quadruplet using a 2-bit control field in the corresponding dword element of the second source operand. Permuted results are stored in the destination operand (first operand).

The 2-bit control fields are located at the low two bits of each dword element (see Figure 5-24). Each control determines which of the source element in an input quadruple is selected for the destination element. Each quadruple of source elements must lie in the same 128-bit region as the destination.

EVEX version: The second source operand (third operand) is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. Permuted results are written to the destination under the writemask.



**Figure 5-23. VPERMILPS Operation**



**Figure 5-24. VPERMILPS Shuffle Control**

(immediate control version)

Permute quadruples of single-precision floating-point values in the first source operand (second operand), each quadruplet using a 2-bit control field in the imm8 byte. Each 128-bit lane in the destination operand (first operand) use the four control fields of the same imm8 byte.

VEX version: The source operand is a YMM/XMM register or a 256/128-bit memory location and the destination operand is a YMM/XMM register.

EVEX version: The source operand (second operand) is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. Permuted results are written to the destination under the writemask.

Note: For the imm8 version, VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instruction will #UD.

**Operation**

```

Select4(SRC, control) {
CASE (control[1:0]) OF
  0:  TMP ← SRC[31:0];
  1:  TMP ← SRC[63:32];
  2:  TMP ← SRC[95:64];
  3:  TMP ← SRC[127:96];
ESAC;
RETURN TMP
}

```

**VPERMILPS (EVEX immediate versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

  i ← j \* 32

  IF (EVEX.b = 1) AND (SRC1 \*is memory\*)

    THEN TMP\_SRC1[i+31:i] ← SRC1[31:0];

    ELSE TMP\_SRC1[i+31:i] ← SRC1[i+31:i];

  FI;

ENDFOR;

TMP\_DEST[31:0] ← Select4(TMP\_SRC1[127:0], imm8[1:0]);

TMP\_DEST[63:32] ← Select4(TMP\_SRC1[127:0], imm8[3:2]);

TMP\_DEST[95:64] ← Select4(TMP\_SRC1[127:0], imm8[5:4]);

TMP\_DEST[127:96] ← Select4(TMP\_SRC1[127:0], imm8[7:6]); FI;

IF VL >= 256

  TMP\_DEST[159:128] ← Select4(TMP\_SRC1[255:128], imm8[1:0]); FI;

  TMP\_DEST[191:160] ← Select4(TMP\_SRC1[255:128], imm8[3:2]); FI;

  TMP\_DEST[223:192] ← Select4(TMP\_SRC1[255:128], imm8[5:4]); FI;

  TMP\_DEST[255:224] ← Select4(TMP\_SRC1[255:128], imm8[7:6]); FI;

FI;

IF VL >= 512

  TMP\_DEST[287:256] ← Select4(TMP\_SRC1[383:256], imm8[1:0]); FI;

  TMP\_DEST[319:288] ← Select4(TMP\_SRC1[383:256], imm8[3:2]); FI;

  TMP\_DEST[351:320] ← Select4(TMP\_SRC1[383:256], imm8[5:4]); FI;

  TMP\_DEST[383:352] ← Select4(TMP\_SRC1[383:256], imm8[7:6]); FI;

  TMP\_DEST[415:384] ← Select4(TMP\_SRC1[511:384], imm8[1:0]); FI;

  TMP\_DEST[447:416] ← Select4(TMP\_SRC1[511:384], imm8[3:2]); FI;

  TMP\_DEST[479:448] ← Select4(TMP\_SRC1[511:384], imm8[5:4]); FI;

  TMP\_DEST[511:480] ← Select4(TMP\_SRC1[511:384], imm8[7:6]); FI;

FI;

FOR j ← 0 TO KL-1

  i ← j \* 32

  IF k1[j] OR \*no writemask\*

    THEN DEST[i+31:i] ← TMP\_DEST[i+31:i]

    ELSE

      IF \*merging-masking\*

        THEN \*DEST[i+31:i] remains unchanged\*

        ELSE DEST[i+31:i] ← 0 ;zeroing-masking

    FI;

  FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VPERMILPS (256-bit immediate version)**

```

DEST[31:0] ← Select4(SRC1[127:0], imm8[1:0]);
DEST[63:32] ← Select4(SRC1[127:0], imm8[3:2]);
DEST[95:64] ← Select4(SRC1[127:0], imm8[5:4]);
DEST[127:96] ← Select4(SRC1[127:0], imm8[7:6]);
DEST[159:128] ← Select4(SRC1[255:128], imm8[1:0]);
DEST[191:160] ← Select4(SRC1[255:128], imm8[3:2]);
DEST[223:192] ← Select4(SRC1[255:128], imm8[5:4]);
DEST[255:224] ← Select4(SRC1[255:128], imm8[7:6]);

```

**VPERMILPS (128-bit immediate version)**

```

DEST[31:0] ← Select4(SRC1[127:0], imm8[1:0]);
DEST[63:32] ← Select4(SRC1[127:0], imm8[3:2]);
DEST[95:64] ← Select4(SRC1[127:0], imm8[5:4]);
DEST[127:96] ← Select4(SRC1[127:0], imm8[7:6]);
DEST[MAX_VL-1:128] ← 0

```

**VPERMILPS (EVEX variable versions)**

(KL, VL) = (16, 512)

FOR j ← 0 TO KL-1

  i ← j \* 32

  IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

    THEN TMP\_SRC2[i+31:i] ← SRC2[31:0];

    ELSE TMP\_SRC2[i+31:i] ← SRC2[i+31:i];

  FI;

ENDFOR;

TMP\_DEST[31:0] ← Select4(SRC1[127:0], TMP\_SRC2[1:0]);

TMP\_DEST[63:32] ← Select4(SRC1[127:0], TMP\_SRC2[33:32]);

TMP\_DEST[95:64] ← Select4(SRC1[127:0], TMP\_SRC2[65:64]);

TMP\_DEST[127:96] ← Select4(SRC1[127:0], TMP\_SRC2[97:96]);

IF VL >= 256

  TMP\_DEST[159:128] ← Select4(SRC1[255:128], TMP\_SRC2[129:128]);

  TMP\_DEST[191:160] ← Select4(SRC1[255:128], TMP\_SRC2[161:160]);

  TMP\_DEST[223:192] ← Select4(SRC1[255:128], TMP\_SRC2[193:192]);

  TMP\_DEST[255:224] ← Select4(SRC1[255:128], TMP\_SRC2[225:224]);

FI;

IF VL >= 512

  TMP\_DEST[287:256] ← Select4(SRC1[383:256], TMP\_SRC2[257:256]);

  TMP\_DEST[319:288] ← Select4(SRC1[383:256], TMP\_SRC2[289:288]);

  TMP\_DEST[351:320] ← Select4(SRC1[383:256], TMP\_SRC2[321:320]);

  TMP\_DEST[383:352] ← Select4(SRC1[383:256], TMP\_SRC2[353:352]);

  TMP\_DEST[415:384] ← Select4(SRC1[511:384], TMP\_SRC2[385:384]);

  TMP\_DEST[447:416] ← Select4(SRC1[511:384], TMP\_SRC2[417:416]);

  TMP\_DEST[479:448] ← Select4(SRC1[511:384], TMP\_SRC2[449:448]);

  TMP\_DEST[511:480] ← Select4(SRC1[511:384], TMP\_SRC2[481:480]);

FI;

FOR j ← 0 TO KL-1

  i ← j \* 32

  IF k1[j] OR \*no writemask\*

    THEN DEST[i+31:i] ← TMP\_DEST[i+31:i]

    ELSE

      IF \*merging-masking\*

        THEN \*DEST[i+31:i] remains unchanged\*

        ELSE DEST[i+31:i] ← 0 ;zeroing-masking

```

    FI;
  ENDFOR
  DEST[MAX_VL-1:VL] ← 0

```

**VPERMILPS (256-bit variable version)**

```

DEST[31:0] ← Select4(SRC1[127:0], SRC2[1:0]);
DEST[63:32] ← Select4(SRC1[127:0], SRC2[33:32]);
DEST[95:64] ← Select4(SRC1[127:0], SRC2[65:64]);
DEST[127:96] ← Select4(SRC1[127:0], SRC2[97:96]);
DEST[159:128] ← Select4(SRC1[255:128], SRC2[129:128]);
DEST[191:160] ← Select4(SRC1[255:128], SRC2[161:160]);
DEST[223:192] ← Select4(SRC1[255:128], SRC2[193:192]);
DEST[255:224] ← Select4(SRC1[255:128], SRC2[225:224]);
DEST[MAX_VL-1:256] ← 0

```

**VPERMILPS (128-bit variable version)**

```

DEST[31:0] ← Select4(SRC1[127:0], SRC2[1:0]);
DEST[63:32] ← Select4(SRC1[127:0], SRC2[33:32]);
DEST[95:64] ← Select4(SRC1[127:0], SRC2[65:64]);
DEST[127:96] ← Select4(SRC1[127:0], SRC2[97:96]);
DEST[MAX_VL-1:128] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VPERMILPS __m512 __mm512_permute_ps( __m512 a, int imm);
VPERMILPS __m512 __mm512_mask_permute_ps( __m512 s, __mmask16 k, __m512 a, int imm);
VPERMILPS __m512 __mm512_maskz_permute_ps( __mmask16 k, __m512 a, int imm);
VPERMILPS __m256 __mm256_mask_permute_ps( __m256 s, __mmask8 k, __m256 a, int imm);
VPERMILPS __m256 __mm256_maskz_permute_ps( __mmask8 k, __m256 a, int imm);
VPERMILPS __m128 __mm_mask_permute_ps( __m128 s, __mmask8 k, __m128 a, int imm);
VPERMILPS __m128 __mm_maskz_permute_ps( __mmask8 k, __m128 a, int imm);
VPERMILPS __m512 __mm512_permutevar_ps( __m512i i, __m512 a);
VPERMILPS __m512 __mm512_mask_permutevar_ps( __m512 s, __mmask16 k, __m512i i, __m512 a);
VPERMILPS __m512 __mm512_maskz_permutevar_ps( __mmask16 k, __m512i i, __m512 a);
VPERMILPS __m256 __mm256_mask_permutevar_ps( __m256 s, __mmask8 k, __m256 i, __m256 a);
VPERMILPS __m256 __mm256_maskz_permutevar_ps( __mmask8 k, __m256 i, __m256 a);
VPERMILPS __m128 __mm_mask_permutevar_ps( __m128 s, __mmask8 k, __m128 i, __m128 a);
VPERMILPS __m128 __mm_maskz_permutevar_ps( __mmask8 k, __m128 i, __m128 a);
VPERMILPS __m128 __mm_permute_ps( __m128 a, int control);
VPERMILPS __m256 __mm256_permute_ps( __m256 a, int control);
VPERMILPS __m128 __mm_permutevar_ps( __m128 a, __m128i control);
VPERMILPS __m256 __mm256_permutevar_ps( __m256 a, __m256i control);

```

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 4;

#UD If VEX.W = 1.

EVEX-encoded instruction, see Exceptions Type E4NF.

#UD If either (E)VEX.vvvv != 1111B and with imm8.

## VPERMPD—Permute Double-Precision Floating-Point Elements

Opcode/ Instruction	Op / En	64/32 bitMode Support	CPUID Feature Flag	Description
VEX.256.66.0F3A.W1 01 /r ib VPERMPD ymm1, ymm2/m256, imm8	RMI	V/V	AVX2	Permute double-precision floating-point elements in ymm2/m256 using indices in imm8 and store the result in ymm1.
EVEX.256.66.0F3A.W1 01 /r ib VPERMPD ymm1 {k1}{z}, ymm2/m256/m64bcst, imm8	FV-RM	V/V	AVX512VL AVX512F	Permute double-precision floating-point elements in ymm2/m256/m64bcst using indexes in imm8 and store the result in ymm1 subject to writemask k1.
EVEX.512.66.0F3A.W1 01 /r ib VPERMPD zmm1 {k1}{z}, zmm2/m512/m64bcst, imm8	FV-RMI	V/V	AVX512F	Permute double-precision floating-point elements in zmm2/m512/m64bcst using indices in imm8 and store the result in zmm1 subject to writemask k1.
EVEX.NDS.256.66.0F38.W1 16 /r VPERMPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	FV-RVM	V/V	AVX512VL AVX512F	Permute double-precision floating-point elements in ymm3/m256/m64bcst using indexes in ymm2 and store the result in ymm1 subject to writemask k1.
EVEX.NDS.512.66.0F38.W1 16 /r VPERMPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	FV-RVM	V/V	AVX512F	Permute double-precision floating-point elements in zmm3/m512/m64bcst using indices in zmm2 and store the result in zmm1 subject to writemask k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (w)	ModRM:r/m (r)	Imm8	NA
FV-RMI	ModRM:reg (w)	ModRM:r/m (r)	Imm8	NA
FV-RVM	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

The imm8 version: Copies quadword elements of double-precision floating-point values from the source operand (the second operand) to the destination operand (the first operand) according to the indices specified by the immediate operand (the third operand). Each two-bit value in the immediate byte selects a qword element in the source operand.

VEX version: The source operand can be a YMM register or a memory location. Bits (MAX\_VL-1:256) of the corresponding destination register are zeroed.

In EVEX.512 encoded version, The elements in the destination are updated using the writemask k1 and the imm8 bits are reused as control bits for the upper 256-bit half when the control bits are coming from immediate. The source operand can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 64-bit memory location.

The imm8 versions: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

The vector control version: Copies quadword elements of double-precision floating-point values from the second source operand (the third operand) to the destination operand (the first operand) according to the indices in the first source operand (the second operand). The first 3 bits of each 64 bit element in the index operand selects which quadword in the second source operand to copy. The first and second operands are ZMM registers, the third operand can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 64-bit memory location. The elements in the destination are updated using the writemask k1.

Note that this instruction permits a qword in the source operand to be copied to multiple locations in the destination operand.

If VPERMPD is encoded with VEX.L= 0, an attempt to execute the instruction encoded with VEX.L= 0 will cause an #UD exception.



**Operation****VPERMPD (EVEX - imm8 control forms)**

(KL, VL) = (4, 256), (8, 512)

FOR j ← 0 TO KL-1

i ← j \* 64

IF (EVEX.b = 1) AND (SRC \*is memory\*)

THEN TMP\_SRC[i+63:i] ← SRC[63:0];

ELSE TMP\_SRC[i+63:i] ← SRC[i+63:i];

FI;

ENDFOR;

TMP\_DEST[63:0] ← (TMP\_SRC[256:0] &gt;&gt; (IMM8[1:0] \* 64))[63:0];

TMP\_DEST[127:64] ← (TMP\_SRC[256:0] &gt;&gt; (IMM8[3:2] \* 64))[63:0];

TMP\_DEST[191:128] ← (TMP\_SRC[256:0] &gt;&gt; (IMM8[5:4] \* 64))[63:0];

TMP\_DEST[255:192] ← (TMP\_SRC[256:0] &gt;&gt; (IMM8[7:6] \* 64))[63:0];

IF VL &gt;= 512

TMP\_DEST[319:256] ← (TMP\_SRC[511:256] &gt;&gt; (IMM8[1:0] \* 64))[63:0];

TMP\_DEST[383:320] ← (TMP\_SRC[511:256] &gt;&gt; (IMM8[3:2] \* 64))[63:0];

TMP\_DEST[447:384] ← (TMP\_SRC[511:256] &gt;&gt; (IMM8[5:4] \* 64))[63:0];

TMP\_DEST[511:448] ← (TMP\_SRC[511:256] &gt;&gt; (IMM8[7:6] \* 64))[63:0];

FI;

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN DEST[i+63:i] ← TMP\_DEST[i+63:i]

ELSE

IF \*merging-masking\* ;merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE ;zeroing-masking

DEST[i+63:i] ← 0 ;zeroing-masking

FI;

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VPERMPD (EVEX - vector control forms)**

(KL, VL) = (4, 256), (8, 512)

FOR j ← 0 TO KL-1

i ← j \* 64

IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

THEN TMP\_SRC2[i+63:i] ← SRC2[63:0];

ELSE TMP\_SRC2[i+63:i] ← SRC2[i+63:i];

FI;

ENDFOR;

IF VL = 256

TMP\_DEST[63:0] ← (TMP\_SRC2[255:0] &gt;&gt; (SRC1[1:0] \* 64))[63:0];

TMP\_DEST[127:64] ← (TMP\_SRC2[255:0] &gt;&gt; (SRC1[65:64] \* 64))[63:0];

TMP\_DEST[191:128] ← (TMP\_SRC2[255:0] &gt;&gt; (SRC1[129:128] \* 64))[63:0];

TMP\_DEST[255:192] ← (TMP\_SRC2[255:0] &gt;&gt; (SRC1[193:192] \* 64))[63:0];

FI;

IF VL = 512

TMP\_DEST[63:0] ← (TMP\_SRC2[511:0] &gt;&gt; (SRC1[2:0] \* 64))[63:0];

```

TMP_DEST[127:64] ← (TMP_SRC2[511:0] >> (SRC1[66:64] * 64))[63:0];
TMP_DEST[191:128] ← (TMP_SRC2[511:0] >> (SRC1[130:128] * 64))[63:0];
TMP_DEST[255:192] ← (TMP_SRC2[511:0] >> (SRC1[194:192] * 64))[63:0];
TMP_DEST[319:256] ← (TMP_SRC2[511:0] >> (SRC1[258:256] * 64))[63:0];
TMP_DEST[383:320] ← (TMP_SRC2[511:0] >> (SRC1[322:320] * 64))[63:0];
TMP_DEST[447:384] ← (TMP_SRC2[511:0] >> (SRC1[386:384] * 64))[63:0];
TMP_DEST[511:448] ← (TMP_SRC2[511:0] >> (SRC1[450:448] * 64))[63:0];
FI;
FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] ← TMP_DEST[j+63:i]
  ELSE
    IF *merging-masking* ;merging-masking
      THEN *DEST[i+63:i] remains unchanged*
    ELSE ;zeroing-masking
      DEST[i+63:i] ← 0 ;zeroing-masking
  FI;
FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VPERMPD (VEX.256 encoded version)**

```

DEST[63:0] ←(SRC[255:0] >> (IMM8[1:0] * 64))[63:0];
DEST[127:64] ←(SRC[255:0] >> (IMM8[3:2] * 64))[63:0];
DEST[191:128] ←(SRC[255:0] >> (IMM8[5:4] * 64))[63:0];
DEST[255:192] ←(SRC[255:0] >> (IMM8[7:6] * 64))[63:0];
DEST[MAX_VL-1:256] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VPERMPD __m512d __mm512_permutex_pd( __m512d a, int imm);
VPERMPD __m512d __mm512_mask_permutex_pd(__m512d s, __mmask16 k, __m512d a, int imm);
VPERMPD __m512d __mm512_maskz_permutex_pd( __mmask16 k, __m512d a, int imm);
VPERMPD __m512d __mm512_permutexvar_pd( __m512i i, __m512d a);
VPERMPD __m512d __mm512_mask_permutexvar_pd(__m512d s, __mmask16 k, __m512i i, __m512d a);
VPERMPD __m512d __mm512_maskz_permutexvar_pd( __mmask16 k, __m512i i, __m512d a);
VPERMPD __m256d __mm256_permutex_epi64( __m256d a, int imm);
VPERMPD __m256d __mm256_mask_permutex_epi64(__m256i s, __mmask8 k, __m256d a, int imm);
VPERMPD __m256d __mm256_maskz_permutex_epi64( __mmask8 k, __m256d a, int imm);
VPERMPD __m256d __mm256_permutexvar_epi64( __m256i i, __m256d a);
VPERMPD __m256d __mm256_mask_permutexvar_epi64(__m256i s, __mmask8 k, __m256i i, __m256d a);
VPERMPD __m256d __mm256_maskz_permutexvar_epi64( __mmask8 k, __m256i i, __m256d a);

```

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 4; additionally

```

#UD          If VEX.L = 0.
             If VEX.vvvv != 1111B.

```

EVEX-encoded instruction, see Exceptions Type E4NF.

```

#UD          If encoded with EVEX.128.
             If EVEX.vvvv != 1111B and with imm8.

```

## VPERMPS—Permute Single-Precision Floating-Point Elements

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.256.66.0F38.W0 16 /r VPERMPS ymm1, ymm2, ymm3/m256	RVM	V/V	AVX2	Permute single-precision floating-point elements in ymm3/m256 using indices in ymm2 and store the result in ymm1.
EVEX.NDS.256.66.0F38.W0 16 /r VPERMPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	FV	V/V	AVX512VL AVX512F	Permute single-precision floating-point elements in ymm3/m256/m32bcst using indexes in ymm2 and store the result in ymm1 subject to write mask k1.
EVEX.NDS.512.66.0F38.W0 16 /r VPERMPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	FV	V/V	AVX512F	Permute single-precision floating-point values in zmm3/m512/m32bcst using indices in zmm2 and store the result in zmm1 subject to write mask k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FV	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Copies doubleword elements of single-precision floating-point values from the second source operand (the third operand) to the destination operand (the first operand) according to the indices in the first source operand (the second operand). Note that this instruction permits a doubleword in the source operand to be copied to more than one location in the destination operand.

VEX.256 versions: The first and second operands are YMM registers, the third operand can be a YMM register or memory location. Bits (MAX\_VL-1:256) of the corresponding destination register are zeroed.

EVEX encoded version: The first and second operands are ZMM registers, the third operand can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32-bit memory location. The elements in the destination are updated using the writemask k1.

If VPERMPS is encoded with VEX.L= 0, an attempt to execute the instruction encoded with VEX.L= 0 will cause an #UD exception.

### Operation

#### VPERMPS (EVEX forms)

(KL, VL) (8, 256), (16, 512)

FOR j ← 0 TO KL-1

  i ← j \* 64

  IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

    THEN TMP\_SRC2[i+31:i] ← SRC2[31:0];

    ELSE TMP\_SRC2[i+31:i] ← SRC2[i+31:i];

  FI;

ENDFOR;

IF VL = 256

  TMP\_DEST[31:0] ← (TMP\_SRC2[255:0] >> (SRC1[2:0] \* 32))[31:0];

  TMP\_DEST[63:32] ← (TMP\_SRC2[255:0] >> (SRC1[34:32] \* 32))[31:0];

  TMP\_DEST[95:64] ← (TMP\_SRC2[255:0] >> (SRC1[66:64] \* 32))[31:0];

  TMP\_DEST[127:96] ← (TMP\_SRC2[255:0] >> (SRC1[98:96] \* 32))[31:0];

  TMP\_DEST[159:128] ← (TMP\_SRC2[255:0] >> (SRC1[130:128] \* 32))[31:0];

```

TMP_DEST[191:160] ← (TMP_SRC2[255:0] >> (SRC1[162:160] * 32))[31:0];
TMP_DEST[223:192] ← (TMP_SRC2[255:0] >> (SRC1[193:192] * 32))[31:0];
TMP_DEST[255:224] ← (TMP_SRC2[255:0] >> (SRC1[226:224] * 32))[31:0];
FI;
IF VL = 512
  TMP_DEST[31:0] ← (TMP_SRC2[511:0] >> (SRC1[3:0] * 32))[31:0];
  TMP_DEST[63:32] ← (TMP_SRC2[511:0] >> (SRC1[35:32] * 32))[31:0];
  TMP_DEST[95:64] ← (TMP_SRC2[511:0] >> (SRC1[67:64] * 32))[31:0];
  TMP_DEST[127:96] ← (TMP_SRC2[511:0] >> (SRC1[99:96] * 32))[31:0];
  TMP_DEST[159:128] ← (TMP_SRC2[511:0] >> (SRC1[131:128] * 32))[31:0];
  TMP_DEST[191:160] ← (TMP_SRC2[511:0] >> (SRC1[163:160] * 32))[31:0];
  TMP_DEST[223:192] ← (TMP_SRC2[511:0] >> (SRC1[195:192] * 32))[31:0];
  TMP_DEST[255:224] ← (TMP_SRC2[511:0] >> (SRC1[227:224] * 32))[31:0];
  TMP_DEST[287:256] ← (TMP_SRC2[511:0] >> (SRC1[259:256] * 32))[31:0];
  TMP_DEST[319:288] ← (TMP_SRC2[511:0] >> (SRC1[291:288] * 32))[31:0];
  TMP_DEST[351:320] ← (TMP_SRC2[511:0] >> (SRC1[323:320] * 32))[31:0];
  TMP_DEST[383:352] ← (TMP_SRC2[511:0] >> (SRC1[355:352] * 32))[31:0];
  TMP_DEST[415:384] ← (TMP_SRC2[511:0] >> (SRC1[387:384] * 32))[31:0];
  TMP_DEST[447:416] ← (TMP_SRC2[511:0] >> (SRC1[419:416] * 32))[31:0];
  TMP_DEST[479:448] ← (TMP_SRC2[511:0] >> (SRC1[451:448] * 32))[31:0];
  TMP_DEST[511:480] ← (TMP_SRC2[511:0] >> (SRC1[483:480] * 32))[31:0];
FI;
FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ← TMP_DEST[j+31:j]
  ELSE
    IF *merging-masking* ;merging-masking
      THEN *DEST[i+31:i] remains unchanged*
    ELSE ;zeroing-masking
      DEST[i+31:i] ← 0 ;zeroing-masking
  FI;
FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VPERMPS (VEX.256 encoded version)**

```

DEST[31:0] ←(SRC2[255:0] >> (SRC1[2:0] * 32))[31:0];
DEST[63:32] ←(SRC2[255:0] >> (SRC1[34:32] * 32))[31:0];
DEST[95:64] ←(SRC2[255:0] >> (SRC1[66:64] * 32))[31:0];
DEST[127:96] ←(SRC2[255:0] >> (SRC1[98:96] * 32))[31:0];
DEST[159:128] ←(SRC2[255:0] >> (SRC1[130:128] * 32))[31:0];
DEST[191:160] ←(SRC2[255:0] >> (SRC1[162:160] * 32))[31:0];
DEST[223:192] ←(SRC2[255:0] >> (SRC1[194:192] * 32))[31:0];
DEST[255:224] ←(SRC2[255:0] >> (SRC1[226:224] * 32))[31:0];
DEST[MAX_VL-1:256] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VPERMPS __m512 __mm512_permutexvar_ps(__m512i i, __m512 a);
VPERMPS __m512 __mm512_mask_permutexvar_ps(__m512 s, __mmask16 k, __m512i i, __m512 a);
VPERMPS __m512 __mm512_maskz_permutexvar_ps(__mmask16 k, __m512i i, __m512 a);
VPERMPS __m256 __mm256_permutexvar_ps(__m256 i, __m256 a);
VPERMPS __m256 __mm256_mask_permutexvar_ps(__m256 s, __mmask8 k, __m256 i, __m256 a);
VPERMPS __m256 __mm256_maskz_permutexvar_ps(__mmask8 k, __m256 i, __m256 a);

```

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 4; additionally

#UD                      If VEX.L = 0.

EVEX-encoded instruction, see Exceptions Type E4NF.

## VPERMQ—Qwords Element Permutation

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.256.66.0F3A.W1 00 /r ib VPERMQ ymm1, ymm2/m256, imm8	RMI	V/V	AVX2	Permute qwords in ymm2/m256 using indices in imm8 and store the result in ymm1.
EVEX.256.66.0F3A.W1 00 /r ib VPERMQ ymm1 {k1}{z}, ymm2/m256/m64bcst, imm8	FV-RM	V/V	AVX512VL AVX512F	Permute qwords in ymm2/m256/m64bcst using indexes in imm8 and store the result in ymm1.
EVEX.512.66.0F3A.W1 00 /r ib VPERMQ zmm1 {k1}{z}, zmm2/m512/m64bcst, imm8	FV-RMI	V/V	AVX512F	Permute qwords in zmm2/m512/m64bcst using indices in imm8 and store the result in zmm1.
EVEX.NDS.256.66.0F38.W1 36 /r VPERMQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	FV-RVM	V/V	AVX512VL AVX512F	Permute qwords in ymm3/m256/m64bcst using indexes in ymm2 and store the result in ymm1.
EVEX.NDS.512.66.0F38.W1 36 /r VPERMQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	FV-RVM	V/V	AVX512F	Permute qwords in zmm3/m512/m64bcst using indices in zmm2 and store the result in zmm1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (w)	ModRM:r/m (r)	Imm8	NA
FV-RMI	ModRM:reg (w)	ModRM:r/m (r)	Imm8	NA
FV-RVM	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

The imm8 version: Copies quadwords from the source operand (the second operand) to the destination operand (the first operand) according to the indices specified by the immediate operand (the third operand). Each two-bit value in the immediate byte selects a qword element in the source operand.

VEX version: The source operand can be a YMM register or a memory location. Bits (MAX\_VL-1:256) of the corresponding destination register are zeroed.

In EVEX.512 encoded version, The elements in the destination are updated using the writemask k1 and the imm8 bits are reused as control bits for the upper 256-bit half when the control bits are coming from immediate. The source operand can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 64-bit memory location.

Immediate control versions: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

The vector control version: Copies quadwords from the second source operand (the third operand) to the destination operand (the first operand) according to the indices in the first source operand (the second operand). The first 3 bits of each 64 bit element in the index operand selects which quadword in the second source operand to copy. The first and second operands are ZMM registers, the third operand can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 64-bit memory location. The elements in the destination are updated using the writemask k1.

Note that this instruction permits a qword in the source operand to be copied to multiple locations in the destination operand.

If VPERMPQ is encoded with VEX.L= 0 or EVEX.128, an attempt to execute the instruction will cause an #UD exception.

**Operation****VPERMQ (EVEX - imm8 control forms)**

(KL, VL) = (4, 256), (8, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 64
  IF (EVEX.b = 1) AND (SRC *is memory*)
    THEN TMP_SRC[i+63:i] ← SRC[63:0];
    ELSE TMP_SRC[i+63:i] ← SRC[i+63:i];
  FI;
ENDFOR;
TMP_DEST[63:0] ← (TMP_SRC[255:0] >> (IMM8[1:0] * 64))[63:0];
TMP_DEST[127:64] ← (TMP_SRC[255:0] >> (IMM8[3:2] * 64))[63:0];
TMP_DEST[191:128] ← (TMP_SRC[255:0] >> (IMM8[5:4] * 64))[63:0];
TMP_DEST[255:192] ← (TMP_SRC[255:0] >> (IMM8[7:6] * 64))[63:0];
IF VL >= 512
  TMP_DEST[319:256] ← (TMP_SRC[511:256] >> (IMM8[1:0] * 64))[63:0];
  TMP_DEST[383:320] ← (TMP_SRC[511:256] >> (IMM8[3:2] * 64))[63:0];
  TMP_DEST[447:384] ← (TMP_SRC[511:256] >> (IMM8[5:4] * 64))[63:0];
  TMP_DEST[511:448] ← (TMP_SRC[511:256] >> (IMM8[7:6] * 64))[63:0];
FI;
FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
    ELSE
      IF *merging-masking* ;merging-masking
        THEN *DEST[i+63:i] remains unchanged*
      ELSE ;zeroing-masking
        DEST[i+63:i] ← 0 ;zeroing-masking
      FI;
    FI;
  FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VPERMQ (EVEX - vector control forms)**

(KL, VL) = (4, 256), (8, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 64
  IF (EVEX.b = 1) AND (SRC2 *is memory*)
    THEN TMP_SRC2[i+63:i] ← SRC2[63:0];
    ELSE TMP_SRC2[i+63:i] ← SRC2[i+63:i];
  FI;
ENDFOR;
IF VL = 256
  TMP_DEST[63:0] ← (TMP_SRC2[255:0] >> (SRC1[1:0] * 64))[63:0];
  TMP_DEST[127:64] ← (TMP_SRC2[255:0] >> (SRC1[65:64] * 64))[63:0];
  TMP_DEST[191:128] ← (TMP_SRC2[255:0] >> (SRC1[129:128] * 64))[63:0];
  TMP_DEST[255:192] ← (TMP_SRC2[255:0] >> (SRC1[193:192] * 64))[63:0];
FI;
IF VL = 512
  TMP_DEST[63:0] ← (TMP_SRC2[511:0] >> (SRC1[2:0] * 64))[63:0];
  TMP_DEST[127:64] ← (TMP_SRC2[511:0] >> (SRC1[66:64] * 64))[63:0];
  TMP_DEST[191:128] ← (TMP_SRC2[511:0] >> (SRC1[130:128] * 64))[63:0];
  TMP_DEST[255:192] ← (TMP_SRC2[511:0] >> (SRC1[194:192] * 64))[63:0];

```

```

TMP_DEST[319:256] ← (TMP_SRC2[511:0] >> (SRC1[258:256] * 64))[63:0];
TMP_DEST[383:320] ← (TMP_SRC2[511:0] >> (SRC1[322:320] * 64))[63:0];
TMP_DEST[447:384] ← (TMP_SRC2[511:0] >> (SRC1[386:384] * 64))[63:0];
TMP_DEST[511:448] ← (TMP_SRC2[511:0] >> (SRC1[450:448] * 64))[63:0];
FI;
FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
  ELSE
    IF *merging-masking* ;merging-masking
      THEN *DEST[i+63:i] remains unchanged*
    ELSE ;zeroing-masking
      DEST[i+63:i] ← 0 ;zeroing-masking
  FI;
FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VPERMQ (VEX.256 encoded version)**

```

DEST[63:0] ← (SRC[255:0] >> (IMM8[1:0] * 64))[63:0];
DEST[127:64] ← (SRC[255:0] >> (IMM8[3:2] * 64))[63:0];
DEST[191:128] ← (SRC[255:0] >> (IMM8[5:4] * 64))[63:0];
DEST[255:192] ← (SRC[255:0] >> (IMM8[7:6] * 64))[63:0];
DEST[MAX_VL-1:256] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VPERMQ __m512i __mm512_permutex_epi64( __m512i a, int imm);
VPERMQ __m512i __mm512_mask_permutex_epi64( __m512i s, __mmask8 k, __m512i a, int imm);
VPERMQ __m512i __mm512_maskz_permutex_epi64( __mmask8 k, __m512i a, int imm);
VPERMQ __m512i __mm512_permutexvar_epi64( __m512i a, __m512i b);
VPERMQ __m512i __mm512_mask_permutexvar_epi64( __m512i s, __mmask8 k, __m512i a, __m512i b);
VPERMQ __m512i __mm512_maskz_permutexvar_epi64( __mmask8 k, __m512i a, __m512i b);
VPERMQ __m256i __mm256_permutex_epi64( __m256i a, int imm);
VPERMQ __m256i __mm256_mask_permutex_epi64( __m256i s, __mmask8 k, __m256i a, int imm);
VPERMQ __m256i __mm256_maskz_permutex_epi64( __mmask8 k, __m256i a, int imm);
VPERMQ __m256i __mm256_permutexvar_epi64( __m256i a, __m256i b);
VPERMQ __m256i __mm256_mask_permutexvar_epi64( __m256i s, __mmask8 k, __m256i a, __m256i b);
VPERMQ __m256i __mm256_maskz_permutexvar_epi64( __mmask8 k, __m256i a, __m256i b);

```

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 4; additionally

```

#UD          If VEX.L = 0.
             If VEX.vvvv != 1111B.

```

EVEX-encoded instruction, see Exceptions Type E4NF.

```

#UD          If encoded with EVEX.128.
             If EVEX.vvvv != 1111B and with imm8.

```



## VPEXPANDD—Load Sparse Packed Doubleword Integer Values from Dense Memory / Register

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 89 /r VPEXPANDD xmm1 {k1}{z}, xmm2/m128	T1S	V/V	AVX512VL AVX512F	Expand packed double-word integer values from xmm2/m128 to xmm1 using writemask k1.
EVEX.256.66.0F38.W0 89 /r VPEXPANDD ymm1 {k1}{z}, ymm2/m256	T1S	V/V	AVX512VL AVX512F	Expand packed double-word integer values from ymm2/m256 to ymm1 using writemask k1.
EVEX.512.66.0F38.W0 89 /r VPEXPANDD zmm1 {k1}{z}, zmm2/m512	T1S	V/V	AVX512F	Expand packed double-word integer values from zmm2/m512 to zmm1 using writemask k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
T1S	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Expand (load) up to 16 contiguous doubleword integer values of the input vector in the source operand (the second operand) to sparse elements in the destination operand (the first operand), selected by the writemask k1. The destination operand is a ZMM register, the source operand can be a ZMM register or memory location.

The input vector starts from the lowest element in the source operand. The opmask register k1 selects the destination elements (a partial vector or sparse elements if less than 8 elements) to be replaced by the ascending elements in the input vector. Destination elements not selected by the writemask k1 are either unmodified or zeroed, depending on EVEX.z.

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

Note that the compressed displacement assumes a pre-scaling (N) corresponding to the size of one single element instead of the size of the full vector.

### Operation

#### VPEXPANDD (EVEX encoded versions)

(KL, VL) = (4, 128), (8, 256), (16, 512)

$k \leftarrow 0$

FOR  $j \leftarrow 0$  TO KL-1

$i \leftarrow j * 32$

  IF  $k1[j]$  OR \*no writemask\*

    THEN

$DEST[i+31:i] \leftarrow SRC[k+31:k];$

$k \leftarrow k + 32$

    ELSE

      IF \*merging-masking\* ; merging-masking

        THEN \*DEST[i+31:i] remains unchanged\*

        ELSE ; zeroing-masking

$DEST[i+31:i] \leftarrow 0$

    FI

  FI;

ENDFOR

$DEST[MAX\_VL-1:VL] \leftarrow 0$

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VEXPANDD __m512i __mm512_mask_expandloadu_epi32(__m512i s, __mmask16 k, void * a);
VEXPANDD __m512i __mm512_maskz_expandloadu_epi32(__mmask16 k, void * a);
VEXPANDD __m512i __mm512_mask_expand_epi32(__m512i s, __mmask16 k, __m512i a);
VEXPANDD __m512i __mm512_maskz_expand_epi32(__mmask16 k, __m512i a);
VEXPANDD __m256i __mm256_mask_expandloadu_epi32(__m256i s, __mmask8 k, void * a);
VEXPANDD __m256i __mm256_maskz_expandloadu_epi32(__mmask8 k, void * a);
VEXPANDD __m256i __mm256_mask_expand_epi32(__m256i s, __mmask8 k, __m256i a);
VEXPANDD __m256i __mm256_maskz_expand_epi32(__mmask8 k, __m256i a);
VEXPANDD __m128i __mm_mask_expandloadu_epi32(__m128i s, __mmask8 k, void * a);
VEXPANDD __m128i __mm_maskz_expandloadu_epi32(__mmask8 k, void * a);
VEXPANDD __m128i __mm_mask_expand_epi32(__m128i s, __mmask8 k, __m128i a);
VEXPANDD __m128i __mm_maskz_expand_epi32(__mmask8 k, __m128i a);

```

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

EVEX-encoded instruction, see Exceptions Type E4.nb.

#UD                    If EVEX.vvvv != 1111B.

## VPEXPANDQ—Load Sparse Packed Quadword Integer Values from Dense Memory / Register

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W1 89 /r VPEXPANDQ xmm1 {k1}{z}, xmm2/m128	T1S	V/V	AVX512VL AVX512F	Expand packed quad-word integer values from xmm2/m128 to xmm1 using writemask k1.
EVEX.256.66.0F38.W1 89 /r VPEXPANDQ ymm1 {k1}{z}, ymm2/m256	T1S	V/V	AVX512VL AVX512F	Expand packed quad-word integer values from ymm2/m256 to ymm1 using writemask k1.
EVEX.512.66.0F38.W1 89 /r VPEXPANDQ zmm1 {k1}{z}, zmm2/m512	T1S	V/V	AVX512F	Expand packed quad-word integer values from zmm2/m512 to zmm1 using writemask k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
T1S	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Expand (load) up to 8 quadword integer values from the source operand (the second operand) to sparse elements in the destination operand (the first operand), selected by the writemask k1. The destination operand is a ZMM register, the source operand can be a ZMM register or memory location.

The input vector starts from the lowest element in the source operand. The opmask register k1 selects the destination elements (a partial vector or sparse elements if less than 8 elements) to be replaced by the ascending elements in the input vector. Destination elements not selected by the writemask k1 are either unmodified or zeroed, depending on EVEX.z.

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

Note that the compressed displacement assumes a pre-scaling (N) corresponding to the size of one single element instead of the size of the full vector.

### Operation

#### VPEXPANDQ (EVEX encoded versions)

(KL, VL) = (2, 128), (4, 256), (8, 512)

k ← 0

FOR j ← 0 TO KL-1

  i ← j \* 64

  IF k1[j] OR \*no writemask\*

    THEN

      DEST[i+63:i] ← SRC[k+63:k];

      k ← k + 64

    ELSE

      IF \*merging-masking\* ; merging-masking

        THEN \*DEST[i+63:i] remains unchanged\*

        ELSE ; zeroing-masking

          THEN DEST[i+63:i] ← 0

    FI

  FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VEXPANDQ __m512i __mm512_mask_expandloadu_epi64(__m512i s, __mmask8 k, void * a);
VEXPANDQ __m512i __mm512_maskz_expandloadu_epi64(__mmask8 k, void * a);
VEXPANDQ __m512i __mm512_mask_expand_epi64(__m512i s, __mmask8 k, __m512i a);
VEXPANDQ __m512i __mm512_maskz_expand_epi64(__mmask8 k, __m512i a);
VEXPANDQ __m256i __mm256_mask_expandloadu_epi64(__m256i s, __mmask8 k, void * a);
VEXPANDQ __m256i __mm256_maskz_expandloadu_epi64(__mmask8 k, void * a);
VEXPANDQ __m256i __mm256_mask_expand_epi64(__m256i s, __mmask8 k, __m256i a);
VEXPANDQ __m256i __mm256_maskz_expand_epi64(__mmask8 k, __m256i a);
VEXPANDQ __m128i __mm_mask_expandloadu_epi64(__m128i s, __mmask8 k, void * a);
VEXPANDQ __m128i __mm_maskz_expandloadu_epi64(__mmask8 k, void * a);
VEXPANDQ __m128i __mm_mask_expand_epi64(__m128i s, __mmask8 k, __m128i a);
VEXPANDQ __m128i __mm_maskz_expand_epi64(__mmask8 k, __m128i a);

```

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

EVEX-encoded instruction, see Exceptions Type E4.nb.

#UD If EVEX.vvvv != 1111B.

## VPLZCNTD/Q—Count the Number of Leading Zero Bits for Packed Dword, Packed Qword Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 44 /r VPLZCNTD xmm1 {k1}{z}, xmm2/m128/m32bcst	FV	V/V	AVX512VL AVX512CD	Count the number of leading zero bits in each dword element of xmm2/m128/m32bcst using writemask k1.
EVEX.256.66.0F38.W0 44 /r VPLZCNTD ymm1 {k1}{z}, ymm2/m256/m32bcst	FV	V/V	AVX512VL AVX512CD	Count the number of leading zero bits in each dword element of ymm2/m256/m32bcst using writemask k1.
EVEX.512.66.0F38.W0 44 /r VPLZCNTD zmm1 {k1}{z}, zmm2/m512/m32bcst	FV	V/V	AVX512CD	Count the number of leading zero bits in each dword element of zmm2/m512/m32bcst using writemask k1.
EVEX.128.66.0F38.W1 44 /r VPLZCNTQ xmm1 {k1}{z}, xmm2/m128/m64bcst	FV	V/V	AVX512VL AVX512CD	Count the number of leading zero bits in each qword element of xmm2/m128/m64bcst using writemask k1.
EVEX.256.66.0F38.W1 44 /r VPLZCNTQ ymm1 {k1}{z}, ymm2/m256/m64bcst	FV	V/V	AVX512VL AVX512CD	Count the number of leading zero bits in each qword element of ymm2/m256/m64bcst using writemask k1.
EVEX.512.66.0F38.W1 44 /r VPLZCNTQ zmm1 {k1}{z}, zmm2/m512/m64bcst	FV	V/V	AVX512CD	Count the number of leading zero bits in each qword element of zmm2/m512/m64bcst using writemask k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
FV	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Counts the number of leading most significant zero bits in each dword or qword element of the source operand (the second operand) and stores the results in the destination register (the first operand) according to the writemask. If an element is zero, the result for that element is the operand size of the element.

EVEX.512 encoded version: The source operand is a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

EVEX.256 encoded version: The source operand is a YMM register, a 256-bit memory location, or a 256-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a YMM register, conditionally updated using writemask k1.

EVEX.128 encoded version: The source operand is a XMM register, a 128-bit memory location, or a 128-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a XMM register, conditionally updated using writemask k1.

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

**Operation****VPLZCNTD**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

i ← j\*32

IF MaskBit(j) OR \*no writemask\*

THEN

temp ← 32

DEST[i+31:i] ← 0

WHILE (temp &gt; 0) AND (SRC[i+temp-1] = 0)

DO

temp ← temp - 1

DEST[i+31:i] ← DEST[i+31:i] + 1

OD

ELSE

IF \*merging-masking\*

THEN \*DEST[i+31:i] remains unchanged\*

ELSE DEST[i+31:i] ← 0

FI

FI

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VPLZCNTQ**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

i ← j\*64

IF MaskBit(j) OR \*no writemask\*

THEN

temp ← 64

DEST[i+63:i] ← 0

WHILE (temp &gt; 0) AND (SRC[i+temp-1] = 0)

DO

temp ← temp - 1

DEST[i+63:i] ← DEST[i+63:i] + 1

OD

ELSE

IF \*merging-masking\*

THEN \*DEST[i+63:i] remains unchanged\*

ELSE DEST[i+63:i] ← 0

FI

FI

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

VPLZCNTD \_\_m512i \_\_mm512\_lzcnt\_epi32(\_\_m512i a);

VPLZCNTD \_\_m512i \_\_mm512\_mask\_lzcnt\_epi32(\_\_m512i s, \_\_mmask16 m, \_\_m512i a);

VPLZCNTD \_\_m512i \_\_mm512\_maskz\_lzcnt\_epi32(\_\_mmask16 m, \_\_m512i a);

VPLZCNTQ \_\_m512i \_\_mm512\_lzcnt\_epi64(\_\_m512i a);

VPLZCNTQ \_\_m512i \_\_mm512\_mask\_lzcnt\_epi64(\_\_m512i s, \_\_mmask8 m, \_\_m512i a);

VPLZCNTQ \_\_m512i \_\_mm512\_maskz\_lzcnt\_epi64(\_\_mmask8 m, \_\_m512i a);

VPLZCNTD \_\_m256i \_\_mm256\_lzcnt\_epi32(\_\_m256i a);

VPLZCNTD \_\_m256i \_mm256\_mask\_lzcnt\_epi32(\_\_m256i s, \_\_mmask8 m, \_\_m256i a);  
 VPLZCNTD \_\_m256i \_mm256\_maskz\_lzcnt\_epi32(\_\_mmask8 m, \_\_m256i a);  
 VPLZCNTQ \_\_m256i \_mm256\_lzcnt\_epi64(\_\_m256i a);  
 VPLZCNTQ \_\_m256i \_mm256\_mask\_lzcnt\_epi64(\_\_m256i s, \_\_mmask8 m, \_\_m256i a);  
 VPLZCNTQ \_\_m256i \_mm256\_maskz\_lzcnt\_epi64(\_\_mmask8 m, \_\_m256i a);  
 VPLZCNTD \_\_m128i \_mm\_lzcnt\_epi32(\_\_m128i a);  
 VPLZCNTD \_\_m128i \_mm\_mask\_lzcnt\_epi32(\_\_m128i s, \_\_mmask8 m, \_\_m128i a);  
 VPLZCNTD \_\_m128i \_mm\_maskz\_lzcnt\_epi32(\_\_mmask8 m, \_\_m128i a);  
 VPLZCNTQ \_\_m128i \_mm\_lzcnt\_epi64(\_\_m128i a);  
 VPLZCNTQ \_\_m128i \_mm\_mask\_lzcnt\_epi64(\_\_m128i s, \_\_mmask8 m, \_\_m128i a);  
 VPLZCNTQ \_\_m128i \_mm\_maskz\_lzcnt\_epi64(\_\_mmask8 m, \_\_m128i a);

### SIMD Floating-Point Exceptions

None

### Other Exceptions

EVEX-encoded instruction, see Exceptions Type E4.

## VPMOVM2B/VPMOVM2W/VPMOVM2D/VPMOVM2Q—Convert a Mask Register to a Vector Register

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.F3.0F38.W0 28 /r VPMOVM2B xmm1, k1	RM	V/V	AVX512VL AVX512BW	Sets each byte in XMM1 to all 1's or all 0's based on the value of the corresponding bit in k1.
EVEX.256.F3.0F38.W0 28 /r VPMOVM2B ymm1, k1	RM	V/V	AVX512VL AVX512BW	Sets each byte in YMM1 to all 1's or all 0's based on the value of the corresponding bit in k1.
EVEX.512.F3.0F38.W0 28 /r VPMOVM2B zmm1, k1	RM	V/V	AVX512BW	Sets each byte in ZMM1 to all 1's or all 0's based on the value of the corresponding bit in k1.
EVEX.128.F3.0F38.W1 28 /r VPMOVM2W xmm1, k1	RM	V/V	AVX512VL AVX512BW	Sets each word in XMM1 to all 1's or all 0's based on the value of the corresponding bit in k1.
EVEX.256.F3.0F38.W1 28 /r VPMOVM2W ymm1, k1	RM	V/V	AVX512VL AVX512BW	Sets each word in YMM1 to all 1's or all 0's based on the value of the corresponding bit in k1.
EVEX.512.F3.0F38.W1 28 /r VPMOVM2W zmm1, k1	RM	V/V	AVX512BW	Sets each word in ZMM1 to all 1's or all 0's based on the value of the corresponding bit in k1.
EVEX.128.F3.0F38.W0 38 /r VPMOVM2D xmm1, k1	RM	V/V	AVX512VL AVX512DQ	Sets each doubleword in XMM1 to all 1's or all 0's based on the value of the corresponding bit in k1.
EVEX.256.F3.0F38.W0 38 /r VPMOVM2D ymm1, k1	RM	V/V	AVX512VL AVX512DQ	Sets each doubleword in YMM1 to all 1's or all 0's based on the value of the corresponding bit in k1.
EVEX.512.F3.0F38.W0 38 /r VPMOVM2D zmm1, k1	RM	V/V	AVX512DQ	Sets each doubleword in ZMM1 to all 1's or all 0's based on the value of the corresponding bit in k1.
EVEX.128.F3.0F38.W1 38 /r VPMOVM2Q xmm1, k1	RM	V/V	AVX512VL AVX512DQ	Sets each quadword in XMM1 to all 1's or all 0's based on the value of the corresponding bit in k1.
EVEX.256.F3.0F38.W1 38 /r VPMOVM2Q ymm1, k1	RM	V/V	AVX512VL AVX512DQ	Sets each quadword in YMM1 to all 1's or all 0's based on the value of the corresponding bit in k1.
EVEX.512.F3.0F38.W1 38 /r VPMOVM2Q zmm1, k1	RM	V/V	AVX512DQ	Sets each quadword in ZMM1 to all 1's or all 0's based on the value of the corresponding bit in k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Converts a mask register to a vector register. Each element in the destination register is set to all 1's or all 0's depending on the value of the corresponding bit in the source mask register.

The source operand is a mask register. The destination operand is a ZMM/YMM/XMM register.

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



**Operation****VPMOVM2B (EVEX encoded versions)**

(KL, VL) = (16, 128), (32, 256), (64, 512)

FOR j ← 0 TO KL-1

i ← j \* 8

IF SRC[j]

THEN DEST[i+7:i] ← -1

ELSE DEST[i+7:i] ← 0

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VPMOVM2W (EVEX encoded versions)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j ← 0 TO KL-1

i ← j \* 16

IF SRC[j]

THEN DEST[i+15:i] ← -1

ELSE DEST[i+15:i] ← 0

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VPMOVM2D (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

i ← j \* 32

IF SRC[j]

THEN DEST[i+31:i] ← -1

ELSE DEST[i+31:i] ← 0

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VPMOVM2Q (EVEX encoded versions)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

i ← j \* 64

IF SRC[j]

THEN DEST[i+63:i] ← -1

ELSE DEST[i+63:i] ← 0

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalents**

```

VPMOVM2B __m512i __mm512_movm_epi8(__mmask64);
VPMOVM2D __m512i __mm512_movm_epi32(__mmask8);
VPMOVM2Q __m512i __mm512_movm_epi64(__mmask16);
VPMOVM2W __m512i __mm512_movm_epi16(__mmask32);
VPMOVM2B __m256i __mm256_movm_epi8(__mmask32);
VPMOVM2D __m256i __mm256_movm_epi32(__mmask8);
VPMOVM2Q __m256i __mm256_movm_epi64(__mmask8);
VPMOVM2W __m256i __mm256_movm_epi16(__mmask16);
VPMOVM2B __m128i __mm_movm_epi8(__mmask16);
VPMOVM2D __m128i __mm_movm_epi32(__mmask8);
VPMOVM2Q __m128i __mm_movm_epi64(__mmask8);
VPMOVM2W __m128i __mm_movm_epi16(__mmask8);

```

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

EVEX-encoded instruction, see Exceptions Type E7NM

#UD                    If EVEX.vvvv != 1111B.

## VPMOVB2M/VPMOVW2M/VPMOVD2M/VPMOVQ2M—Convert a Vector Register to a Mask

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.F3.0F38.W0 29 /r VPMOVB2M k1, xmm1	RM	V/V	AVX512VL AVX512BW	Sets each bit in k1 to 1 or 0 based on the value of the most significant bit of the corresponding byte in XMM1.
EVEX.256.F3.0F38.W0 29 /r VPMOVB2M k1, ymm1	RM	V/V	AVX512VL AVX512BW	Sets each bit in k1 to 1 or 0 based on the value of the most significant bit of the corresponding byte in YMM1.
EVEX.512.F3.0F38.W0 29 /r VPMOVB2M k1, zmm1	RM	V/V	AVX512BW	Sets each bit in k1 to 1 or 0 based on the value of the most significant bit of the corresponding byte in ZMM1.
EVEX.128.F3.0F38.W1 29 /r VPMOVW2M k1, xmm1	RM	V/V	AVX512VL AVX512BW	Sets each bit in k1 to 1 or 0 based on the value of the most significant bit of the corresponding word in XMM1.
EVEX.256.F3.0F38.W1 29 /r VPMOVW2M k1, ymm1	RM	V/V	AVX512VL AVX512BW	Sets each bit in k1 to 1 or 0 based on the value of the most significant bit of the corresponding word in YMM1.
EVEX.512.F3.0F38.W1 29 /r VPMOVW2M k1, zmm1	RM	V/V	AVX512BW	Sets each bit in k1 to 1 or 0 based on the value of the most significant bit of the corresponding word in ZMM1.
EVEX.128.F3.0F38.W0 39 /r VPMOVD2M k1, xmm1	RM	V/V	AVX512VL AVX512DQ	Sets each bit in k1 to 1 or 0 based on the value of the most significant bit of the corresponding doubleword in XMM1.
EVEX.256.F3.0F38.W0 39 /r VPMOVD2M k1, ymm1	RM	V/V	AVX512VL AVX512DQ	Sets each bit in k1 to 1 or 0 based on the value of the most significant bit of the corresponding doubleword in YMM1.
EVEX.512.F3.0F38.W0 39 /r VPMOVD2M k1, zmm1	RM	V/V	AVX512DQ	Sets each bit in k1 to 1 or 0 based on the value of the most significant bit of the corresponding doubleword in ZMM1.
EVEX.128.F3.0F38.W1 39 /r VPMOVQ2M k1, xmm1	RM	V/V	AVX512VL AVX512DQ	Sets each bit in k1 to 1 or 0 based on the value of the most significant bit of the corresponding quadword in XMM1.
EVEX.256.F3.0F38.W1 39 /r VPMOVQ2M k1, ymm1	RM	V/V	AVX512VL AVX512DQ	Sets each bit in k1 to 1 or 0 based on the value of the most significant bit of the corresponding quadword in YMM1.
EVEX.512.F3.0F38.W1 39 /r VPMOVQ2M k1, zmm1	RM	V/V	AVX512DQ	Sets each bit in k1 to 1 or 0 based on the value of the most significant bit of the corresponding quadword in ZMM1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

#### Description

Converts a vector register to a mask register. Each element in the destination register is set to 1 or 0 depending on the value of most significant bit of the corresponding element in the source register.

The source operand is a ZMM/YMM/XMM register. The destination operand is a mask register.

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

**Operation****VPMOVB2M (EVEX encoded versions)**

(KL, VL) = (16, 128), (32, 256), (64, 512)

FOR j ← 0 TO KL-1

i ← j \* 8

IF SRC[i+7]

THEN DEST[j] ← 1

ELSE DEST[j] ← 0

FI;

ENDFOR

DEST[MAX\_KL-1:KL] ← 0

**VPMOVW2M (EVEX encoded versions)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j ← 0 TO KL-1

i ← j \* 16

IF SRC[i+15]

THEN DEST[j] ← 1

ELSE DEST[j] ← 0

FI;

ENDFOR

DEST[MAX\_KL-1:KL] ← 0

**VPMOVD2M (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

i ← j \* 32

IF SRC[i+31]

THEN DEST[j] ← 1

ELSE DEST[j] ← 0

FI;

ENDFOR

DEST[MAX\_KL-1:KL] ← 0

**VPMOVQ2M (EVEX encoded versions)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

i ← j \* 64

IF SRC[i+63]

THEN DEST[j] ← 1

ELSE DEST[j] ← 0

FI;

ENDFOR

DEST[MAX\_KL-1:KL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalents**

```

VPMPOVB2M __mmask64 _mm512_movepi8_mask( __m512i );
VPMPOVD2M __mmask16 _mm512_movepi32_mask( __m512i );
VPMPOVQ2M __mmask8 _mm512_movepi64_mask( __m512i );
VPMPOVW2M __mmask32 _mm512_movepi16_mask( __m512i );
VPMPOVB2M __mmask32 _mm256_movepi8_mask( __m256i );
VPMPOVD2M __mmask8 _mm256_movepi32_mask( __m256i );
VPMPOVQ2M __mmask8 _mm256_movepi64_mask( __m256i );
VPMPOVW2M __mmask16 _mm256_movepi16_mask( __m256i );
VPMPOVB2M __mmask16 _mm_movepi8_mask( __m128i );
VPMPOVD2M __mmask8 _mm_movepi32_mask( __m128i );
VPMPOVQ2M __mmask8 _mm_movepi64_mask( __m128i );
VPMPOVW2M __mmask8 _mm_movepi16_mask( __m128i );

```

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

EVEX-encoded instruction, see Exceptions Type E7NM

#UD                      If EVEX.vvvv != 1111B.

## VPMOVQB/VPMOVSQB/VPMOVUSQB—Down Convert QWord to Byte

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.F3.0F38.W0 32 /r VPMOVQB <i>xmm1/m16</i> { <i>k1</i> }{ <i>z</i> }, <i>xmm2</i>	OVM	V/V	AVX512VL AVX512F	Converts 2 packed quad-word integers from <i>xmm2</i> into 2 packed byte integers in <i>xmm1/m16</i> with truncation under writemask <i>k1</i> .
EVEX.128.F3.0F38.W0 22 /r VPMOVSQB <i>xmm1/m16</i> { <i>k1</i> }{ <i>z</i> }, <i>xmm2</i>	OVM	V/V	AVX512VL AVX512F	Converts 2 packed signed quad-word integers from <i>xmm2</i> into 2 packed signed byte integers in <i>xmm1/m16</i> using signed saturation under writemask <i>k1</i> .
EVEX.128.F3.0F38.W0 12 /r VPMOVUSQB <i>xmm1/m16</i> { <i>k1</i> }{ <i>z</i> }, <i>xmm2</i>	OVM	V/V	AVX512VL AVX512F	Converts 2 packed unsigned quad-word integers from <i>xmm2</i> into 2 packed unsigned byte integers in <i>xmm1/m16</i> using unsigned saturation under writemask <i>k1</i> .
EVEX.256.F3.0F38.W0 32 /r VPMOVQB <i>xmm1/m32</i> { <i>k1</i> }{ <i>z</i> }, <i>ymm2</i>	OVM	V/V	AVX512VL AVX512F	Converts 4 packed quad-word integers from <i>ymm2</i> into 4 packed byte integers in <i>xmm1/m32</i> with truncation under writemask <i>k1</i> .
EVEX.256.F3.0F38.W0 22 /r VPMOVSQB <i>xmm1/m32</i> { <i>k1</i> }{ <i>z</i> }, <i>ymm2</i>	OVM	V/V	AVX512VL AVX512F	Converts 4 packed signed quad-word integers from <i>ymm2</i> into 4 packed signed byte integers in <i>xmm1/m32</i> using signed saturation under writemask <i>k1</i> .
EVEX.256.F3.0F38.W0 12 /r VPMOVUSQB <i>xmm1/m32</i> { <i>k1</i> }{ <i>z</i> }, <i>ymm2</i>	OVM	V/V	AVX512VL AVX512F	Converts 4 packed unsigned quad-word integers from <i>ymm2</i> into 4 packed unsigned byte integers in <i>xmm1/m32</i> using unsigned saturation under writemask <i>k1</i> .
EVEX.512.F3.0F38.W0 32 /r VPMOVQB <i>xmm1/m64</i> { <i>k1</i> }{ <i>z</i> }, <i>zmm2</i>	OVM	V/V	AVX512F	Converts 8 packed quad-word integers from <i>zmm2</i> into 8 packed byte integers in <i>xmm1/m64</i> with truncation under writemask <i>k1</i> .
EVEX.512.F3.0F38.W0 22 /r VPMOVSQB <i>xmm1/m64</i> { <i>k1</i> }{ <i>z</i> }, <i>zmm2</i>	OVM	V/V	AVX512F	Converts 8 packed signed quad-word integers from <i>zmm2</i> into 8 packed signed byte integers in <i>xmm1/m64</i> using signed saturation under writemask <i>k1</i> .
EVEX.512.F3.0F38.W0 12 /r VPMOVUSQB <i>xmm1/m64</i> { <i>k1</i> }{ <i>z</i> }, <i>zmm2</i>	OVM	V/V	AVX512F	Converts 8 packed unsigned quad-word integers from <i>zmm2</i> into 8 packed unsigned byte integers in <i>xmm1/m64</i> using unsigned saturation under writemask <i>k1</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
OVM	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

### Description

VPMOVQB down converts 64-bit integer elements in the source operand (the second operand) into packed byte elements using truncation. VPMOVSQB converts signed 64-bit integers into packed signed bytes using signed saturation. VPMOVUSQB convert unsigned quad-word values into unsigned byte values using unsigned saturation. The source operand is a vector register. The destination operand is an XMM register or a memory location.

Down-converted byte elements are written to the destination operand (the first operand) from the least-significant byte. Byte elements of the destination operand are updated according to the writemask. Bits (MAX\_VL-1:64) of the destination are zeroed.

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

**Operation****VPMOVB instruction (EVEX encoded versions) when dest is a register**

```

(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
  i ← j * 8
  m ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+7:i] ← TruncateQuadWordToByte (SRC[m+63:m])
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+7:i] remains unchanged*
        ELSE *zeroing-masking* ; zeroing-masking
          DEST[i+7:i] ← 0
      FI
  FI;
ENDFOR
DEST[MAX_VL-1:VL/8] ← 0;

```

**VPMOVB instruction (EVEX encoded versions) when dest is memory**

```

(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
  i ← j * 8
  m ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+7:i] ← TruncateQuadWordToByte (SRC[m+63:m])
    ELSE
      *DEST[i+7:i] remains unchanged* ; merging-masking
  FI;
ENDFOR

```

**VPMOVS instruction (EVEX encoded versions) when dest is a register**

```

(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
  i ← j * 8
  m ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+7:i] ← SaturateSignedQuadWordToByte (SRC[m+63:m])
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+7:i] remains unchanged*
        ELSE *zeroing-masking* ; zeroing-masking
          DEST[i+7:i] ← 0
      FI
  FI;
ENDFOR
DEST[MAX_VL-1:VL/8] ← 0;

```

**VPMOVSQB instruction (EVEX encoded versions) when dest is memory**

```

(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
  i ← j * 8
  m ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+7:i] ← SaturateSignedQuadWordToByte (SRC[m+63:m])
  ELSE
    *DEST[i+7:i] remains unchanged*      ; merging-masking
  FI;
ENDFOR

```

**VPMOVUSQB instruction (EVEX encoded versions) when dest is a register**

```

(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
  i ← j * 8
  m ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+7:i] ← SaturateUnsignedQuadWordToByte (SRC[m+63:m])
  ELSE
    IF *merging-masking*      ; merging-masking
      THEN *DEST[i+7:i] remains unchanged*
    ELSE *zeroing-masking*    ; zeroing-masking
      DEST[i+7:i] ← 0
    FI
  FI;
ENDFOR
DEST[MAX_VL-1:VL/8] ← 0;

```

**VPMOVUSQB instruction (EVEX encoded versions) when dest is memory**

```

(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
  i ← j * 8
  m ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+7:i] ← SaturateUnsignedQuadWordToByte (SRC[m+63:m])
  ELSE
    *DEST[i+7:i] remains unchanged*      ; merging-masking
  FI;
ENDFOR

```

**Intel C/C++ Compiler Intrinsic Equivalents**

```

VPMOVQB __m128i __mm512_cvtepi64_epi8( __m512i a);
VPMOVQB __m128i __mm512_mask_cvtepi64_epi8(__m128i s, __mmask8 k, __m512i a);
VPMOVQB __m128i __mm512_maskz_cvtepi64_epi8( __mmask8 k, __m512i a);
VPMOVQB void __mm512_mask_cvtepi64_storeu_epi8(void * d, __mmask8 k, __m512i a);
VPMOVSQB __m128i __mm512_cvtsepi64_epi8( __m512i a);
VPMOVSQB __m128i __mm512_mask_cvtsepi64_epi8(__m128i s, __mmask8 k, __m512i a);
VPMOVSQB __m128i __mm512_maskz_cvtsepi64_epi8( __mmask8 k, __m512i a);
VPMOVSQB void __mm512_mask_cvtsepi64_storeu_epi8(void * d, __mmask8 k, __m512i a);
VPMOVUSQB __m128i __mm512_cvtusepi64_epi8( __m512i a);
VPMOVUSQB __m128i __mm512_mask_cvtusepi64_epi8(__m128i s, __mmask8 k, __m512i a);
VPMOVUSQB __m128i __mm512_maskz_cvtusepi64_epi8( __mmask8 k, __m512i a);
VPMOVUSQB void __mm512_mask_cvtusepi64_storeu_epi8(void * d, __mmask8 k, __m512i a);

```



```

VPMOVUSQB __m128i _mm256_cvtusepi64_epi8(__m256i a);
VPMOVUSQB __m128i _mm256_mask_cvtusepi64_epi8(__m128i a, __mmask8 k, __m256i b);
VPMOVUSQB __m128i _mm256_maskz_cvtusepi64_epi8(__mmask8 k, __m256i b);
VPMOVUSQB void _mm256_mask_cvtusepi64_storeu_epi8(void *, __mmask8 k, __m256i b);
VPMOVUSQB __m128i _mm_cvtusepi64_epi8(__m128i a);
VPMOVUSQB __m128i _mm_mask_cvtusepi64_epi8(__m128i a, __mmask8 k, __m128i b);
VPMOVUSQB __m128i _mm_maskz_cvtusepi64_epi8(__mmask8 k, __m128i b);
VPMOVUSQB void _mm_mask_cvtusepi64_storeu_epi8(void *, __mmask8 k, __m128i b);
VPMOVSQB __m128i _mm256_cvtsepi64_epi8(__m256i a);
VPMOVSQB __m128i _mm256_mask_cvtsepi64_epi8(__m128i a, __mmask8 k, __m256i b);
VPMOVSQB __m128i _mm256_maskz_cvtsepi64_epi8(__mmask8 k, __m256i b);
VPMOVSQB void _mm256_mask_cvtsepi64_storeu_epi8(void *, __mmask8 k, __m256i b);
VPMOVSQB __m128i _mm_cvtsepi64_epi8(__m128i a);
VPMOVSQB __m128i _mm_mask_cvtsepi64_epi8(__m128i a, __mmask8 k, __m128i b);
VPMOVSQB __m128i _mm_maskz_cvtsepi64_epi8(__mmask8 k, __m128i b);
VPMOVSQB void _mm_mask_cvtsepi64_storeu_epi8(void *, __mmask8 k, __m128i b);
VPMOVQB __m128i _mm256_cvtepi64_epi8(__m256i a);
VPMOVQB __m128i _mm256_mask_cvtepi64_epi8(__m128i a, __mmask8 k, __m256i b);
VPMOVQB __m128i _mm256_maskz_cvtepi64_epi8(__mmask8 k, __m256i b);
VPMOVQB void _mm256_mask_cvtepi64_storeu_epi8(void *, __mmask8 k, __m256i b);
VPMOVQB __m128i _mm_cvtepi64_epi8(__m128i a);
VPMOVQB __m128i _mm_mask_cvtepi64_epi8(__m128i a, __mmask8 k, __m128i b);
VPMOVQB __m128i _mm_maskz_cvtepi64_epi8(__mmask8 k, __m128i b);
VPMOVQB void _mm_mask_cvtepi64_storeu_epi8(void *, __mmask8 k, __m128i b);

```

### SIMD Floating-Point Exceptions

None

### Other Exceptions

EVEX-encoded instruction, see Exceptions Type E6.

#UD If EVEX.vvvv != 1111B.

## VPMOVQW/VPMOVSQW/VPMOVUSQW—Down Convert QWord to Word

Opcode/ Instruction	Op / En	64/32 bitMode Support	CPUID Feature Flag	Description
EEX.128.F3.0F38.W0 34 /r VPMOVQW <i>xmm1/m32</i> {k1}{z}, <i>xmm2</i>	QVM	V/V	AVX512VL AVX512F	Converts 2 packed quad-word integers from <i>xmm2</i> into 2 packed word integers in <i>xmm1/m32</i> with truncation under writemask k1.
EEX.128.F3.0F38.W0 24 /r VPMOVSQW <i>xmm1/m32</i> {k1}{z}, <i>xmm2</i>	QVM	V/V	AVX512VL AVX512F	Converts 8 packed signed quad-word integers from <i>zmm2</i> into 8 packed signed word integers in <i>xmm1/m32</i> using signed saturation under writemask k1.
EEX.128.F3.0F38.W0 14 /r VPMOVUSQW <i>xmm1/m32</i> {k1}{z}, <i>xmm2</i>	QVM	V/V	AVX512VL AVX512F	Converts 2 packed unsigned quad-word integers from <i>xmm2</i> into 2 packed unsigned word integers in <i>xmm1/m32</i> using unsigned saturation under writemask k1.
EEX.256.F3.0F38.W0 34 /r VPMOVQW <i>xmm1/m64</i> {k1}{z}, <i>ymm2</i>	QVM	V/V	AVX512VL AVX512F	Converts 4 packed quad-word integers from <i>ymm2</i> into 4 packed word integers in <i>xmm1/m64</i> with truncation under writemask k1.
EEX.256.F3.0F38.W0 24 /r VPMOVSQW <i>xmm1/m64</i> {k1}{z}, <i>ymm2</i>	QVM	V/V	AVX512VL AVX512F	Converts 4 packed signed quad-word integers from <i>ymm2</i> into 4 packed signed word integers in <i>xmm1/m64</i> using signed saturation under writemask k1.
EEX.256.F3.0F38.W0 14 /r VPMOVUSQW <i>xmm1/m64</i> {k1}{z}, <i>ymm2</i>	QVM	V/V	AVX512VL AVX512F	Converts 4 packed unsigned quad-word integers from <i>ymm2</i> into 4 packed unsigned word integers in <i>xmm1/m64</i> using unsigned saturation under writemask k1.
EEX.512.F3.0F38.W0 34 /r VPMOVQW <i>xmm1/m128</i> {k1}{z}, <i>zmm2</i>	QVM	V/V	AVX512F	Converts 8 packed quad-word integers from <i>zmm2</i> into 8 packed word integers in <i>xmm1/m128</i> with truncation under writemask k1.
EEX.512.F3.0F38.W0 24 /r VPMOVSQW <i>xmm1/m128</i> {k1}{z}, <i>zmm2</i>	QVM	V/V	AVX512F	Converts 8 packed signed quad-word integers from <i>zmm2</i> into 8 packed signed word integers in <i>xmm1/m128</i> using signed saturation under writemask k1.
EEX.512.F3.0F38.W0 14 /r VPMOVUSQW <i>xmm1/m128</i> {k1}{z}, <i>zmm2</i>	QVM	V/V	AVX512F	Converts 8 packed unsigned quad-word integers from <i>zmm2</i> into 8 packed unsigned word integers in <i>xmm1/m128</i> using unsigned saturation under writemask k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
QVM	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

### Description

VPMOVQW down converts 64-bit integer elements in the source operand (the second operand) into packed words using truncation. VPMOVSQW converts signed 64-bit integers into packed signed words using signed saturation. VPMOVUSQW convert unsigned quad-word values into unsigned word values using unsigned saturation.

The source operand is a ZMM/YMM/XMM register. The destination operand is a XMM register or a 128/64/32-bit memory location.

Down-converted word elements are written to the destination operand (the first operand) from the least-significant word. Word elements of the destination operand are updated according to the writemask. Bits (MAX\_VL-1:128/64/32) of the register destination are zeroed.

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

**Operation****VPMOVQW instruction (EVEX encoded versions) when dest is a register**

```

(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
  i ← j * 16
  m ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+15:i] ← TruncateQuadWordToWord (SRC[m+63:m])
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+15:i] remains unchanged*
        ELSE *zeroing-masking* ; zeroing-masking
          DEST[i+15:i] ← 0
      FI
  FI;
ENDFOR
DEST[MAX_VL-1:VL/4] ← 0;

```

**VPMOVQW instruction (EVEX encoded versions) when dest is memory**

```

(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
  i ← j * 16
  m ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+15:i] ← TruncateQuadWordToWord (SRC[m+63:m])
    ELSE
      *DEST[i+15:i] remains unchanged* ; merging-masking
  FI;
ENDFOR

```

**VPMOVSQW instruction (EVEX encoded versions) when dest is a register**

```

(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
  i ← j * 16
  m ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+15:i] ← SaturateSignedQuadWordToWord (SRC[m+63:m])
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+15:i] remains unchanged*
        ELSE *zeroing-masking* ; zeroing-masking
          DEST[i+15:i] ← 0
      FI
  FI;
ENDFOR
DEST[MAX_VL-1:VL/4] ← 0;

```

**VPMOVSQW instruction (EVEX encoded versions) when dest is memory**

```

(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
  i ← j * 16
  m ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+15:i] ← SaturateSignedQuadWordToWord (SRC[m+63:m])
  ELSE
    *DEST[i+15:i] remains unchanged* ; merging-masking
  FI;
ENDFOR

```

**VPMOVUSQW instruction (EVEX encoded versions) when dest is a register**

```

(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
  i ← j * 16
  m ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+15:i] ← SaturateUnsignedQuadWordToWord (SRC[m+63:m])
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+15:i] remains unchanged*
    ELSE *zeroing-masking* ; zeroing-masking
      DEST[i+15:i] ← 0
    FI
  FI;
ENDFOR
DEST[MAX_VL-1:VL/4] ← 0;

```

**VPMOVUSQW instruction (EVEX encoded versions) when dest is memory**

```

(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
  i ← j * 16
  m ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+15:i] ← SaturateUnsignedQuadWordToWord (SRC[m+63:m])
  ELSE
    *DEST[i+15:i] remains unchanged* ; merging-masking
  FI;
ENDFOR

```

**Intel C/C++ Compiler Intrinsic Equivalents**

```

VPMOVQW __m128i __mm512_cvtepi64_epi16(__m512i a);
VPMOVQW __m128i __mm512_mask_cvtepi64_epi16(__m128i s, __mmask8 k, __m512i a);
VPMOVQW __m128i __mm512_maskz_cvtepi64_epi16(__mmask8 k, __m512i a);
VPMOVQW void __mm512_mask_cvtepi64_storeu_epi16(void * d, __mmask8 k, __m512i a);
VPMOVSQW __m128i __mm512_cvtsepi64_epi16(__m512i a);
VPMOVSQW __m128i __mm512_mask_cvtsepi64_epi16(__m128i s, __mmask8 k, __m512i a);
VPMOVSQW __m128i __mm512_maskz_cvtsepi64_epi16(__mmask8 k, __m512i a);
VPMOVSQW void __mm512_mask_cvtsepi64_storeu_epi16(void * d, __mmask8 k, __m512i a);
VPMOVUSQW __m128i __mm512_cvtusepi64_epi16(__m512i a);
VPMOVUSQW __m128i __mm512_mask_cvtusepi64_epi16(__m128i s, __mmask8 k, __m512i a);
VPMOVUSQW __m128i __mm512_maskz_cvtusepi64_epi16(__mmask8 k, __m512i a);
VPMOVUSQW void __mm512_mask_cvtusepi64_storeu_epi16(void * d, __mmask8 k, __m512i a);

```

VPMOVUSQD \_\_m128i \_\_mm256\_cvtusepi64\_epi32(\_\_m256i a);  
 VPMOVUSQD \_\_m128i \_\_mm256\_mask\_cvtusepi64\_epi32(\_\_m128i a, \_\_mmask8 k, \_\_m256i b);  
 VPMOVUSQD \_\_m128i \_\_mm256\_maskz\_cvtusepi64\_epi32(\_\_mmask8 k, \_\_m256i b);  
 VPMOVUSQD void \_\_mm256\_mask\_cvtusepi64\_storeu\_epi32(void \*, \_\_mmask8 k, \_\_m256i b);  
 VPMOVUSQD \_\_m128i \_\_mm\_cvtusepi64\_epi32(\_\_m128i a);  
 VPMOVUSQD \_\_m128i \_\_mm\_mask\_cvtusepi64\_epi32(\_\_m128i a, \_\_mmask8 k, \_\_m128i b);  
 VPMOVUSQD \_\_m128i \_\_mm\_maskz\_cvtusepi64\_epi32(\_\_mmask8 k, \_\_m128i b);  
 VPMOVUSQD void \_\_mm\_mask\_cvtusepi64\_storeu\_epi32(void \*, \_\_mmask8 k, \_\_m128i b);  
 VPMOVSQD \_\_m128i \_\_mm256\_cvtsepi64\_epi32(\_\_m256i a);  
 VPMOVSQD \_\_m128i \_\_mm256\_mask\_cvtsepi64\_epi32(\_\_m128i a, \_\_mmask8 k, \_\_m256i b);  
 VPMOVSQD \_\_m128i \_\_mm256\_maskz\_cvtsepi64\_epi32(\_\_mmask8 k, \_\_m256i b);  
 VPMOVSQD void \_\_mm256\_mask\_cvtsepi64\_storeu\_epi32(void \*, \_\_mmask8 k, \_\_m256i b);  
 VPMOVSQD \_\_m128i \_\_mm\_cvtsepi64\_epi32(\_\_m128i a);  
 VPMOVSQD \_\_m128i \_\_mm\_mask\_cvtsepi64\_epi32(\_\_m128i a, \_\_mmask8 k, \_\_m128i b);  
 VPMOVSQD \_\_m128i \_\_mm\_maskz\_cvtsepi64\_epi32(\_\_mmask8 k, \_\_m128i b);  
 VPMOVSQD void \_\_mm\_mask\_cvtsepi64\_storeu\_epi32(void \*, \_\_mmask8 k, \_\_m128i b);  
 VPMOVQD \_\_m128i \_\_mm256\_cvtepi64\_epi32(\_\_m256i a);  
 VPMOVQD \_\_m128i \_\_mm256\_mask\_cvtepi64\_epi32(\_\_m128i a, \_\_mmask8 k, \_\_m256i b);  
 VPMOVQD \_\_m128i \_\_mm256\_maskz\_cvtepi64\_epi32(\_\_mmask8 k, \_\_m256i b);  
 VPMOVQD void \_\_mm256\_mask\_cvtepi64\_storeu\_epi32(void \*, \_\_mmask8 k, \_\_m256i b);  
 VPMOVQD \_\_m128i \_\_mm\_cvtepi64\_epi32(\_\_m128i a);  
 VPMOVQD \_\_m128i \_\_mm\_mask\_cvtepi64\_epi32(\_\_m128i a, \_\_mmask8 k, \_\_m128i b);  
 VPMOVQD \_\_m128i \_\_mm\_maskz\_cvtepi64\_epi32(\_\_mmask8 k, \_\_m128i b);  
 VPMOVQD void \_\_mm\_mask\_cvtepi64\_storeu\_epi32(void \*, \_\_mmask8 k, \_\_m128i b);

### SIMD Floating-Point Exceptions

None

### Other Exceptions

EVEX-encoded instruction, see Exceptions Type E6.

#UD If EVEX.vvvv != 1111B.

## VPMOVQD/VPMOVSQD/VPMOVUSQD—Down Convert QWord to DWord

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EEX.128.F3.0F38.W0 35 /r VPMOVQD <i>xmm1/m128 {k1}{z}, xmm2</i>	A	V/V	AVX512VL AVX512F	Converts 2 packed quad-word integers from <i>xmm2</i> into 2 packed double-word integers in <i>xmm1/m128</i> with truncation subject to writemask <i>k1</i> .
EEX.128.F3.0F38.W0 25 /r VPMOVSQD <i>xmm1/m64 {k1}{z}, xmm2</i>	A	V/V	AVX512VL AVX512F	Converts 2 packed signed quad-word integers from <i>xmm2</i> into 2 packed signed double-word integers in <i>xmm1/m64</i> using signed saturation subject to writemask <i>k1</i> .
EEX.128.F3.0F38.W0 15 /r VPMOVUSQD <i>xmm1/m64 {k1}{z}, xmm2</i>	A	V/V	AVX512VL AVX512F	Converts 2 packed unsigned quad-word integers from <i>xmm2</i> into 2 packed unsigned double-word integers in <i>xmm1/m64</i> using unsigned saturation subject to writemask <i>k1</i> .
EEX.256.F3.0F38.W0 35 /r VPMOVQD <i>xmm1/m128 {k1}{z}, ymm2</i>	A	V/V	AVX512VL AVX512F	Converts 4 packed quad-word integers from <i>ymm2</i> into 4 packed double-word integers in <i>xmm1/m128</i> with truncation subject to writemask <i>k1</i> .
EEX.256.F3.0F38.W0 25 /r VPMOVSQD <i>xmm1/m128 {k1}{z}, ymm2</i>	A	V/V	AVX512VL AVX512F	Converts 4 packed signed quad-word integers from <i>ymm2</i> into 4 packed signed double-word integers in <i>xmm1/m128</i> using signed saturation subject to writemask <i>k1</i> .
EEX.256.F3.0F38.W0 15 /r VPMOVUSQD <i>xmm1/m128 {k1}{z}, ymm2</i>	A	V/V	AVX512VL AVX512F	Converts 4 packed unsigned quad-word integers from <i>ymm2</i> into 4 packed unsigned double-word integers in <i>xmm1/m128</i> using unsigned saturation subject to writemask <i>k1</i> .
EEX.512.F3.0F38.W0 35 /r VPMOVQD <i>ymm1/m256 {k1}{z}, zmm2</i>	HVM	V/V	AVX512F	Converts 8 packed quad-word integers from <i>zmm2</i> into 8 packed double-word integers in <i>ymm1/m256</i> with truncation subject to writemask <i>k1</i> .
EEX.512.F3.0F38.W0 25 /r VPMOVSQD <i>ymm1/m256 {k1}{z}, zmm2</i>	HVM	V/V	AVX512F	Converts 8 packed signed quad-word integers from <i>zmm2</i> into 8 packed signed double-word integers in <i>ymm1/m256</i> using signed saturation subject to writemask <i>k1</i> .
EEX.512.F3.0F38.W0 15 /r VPMOVUSQD <i>ymm1/m256 {k1}{z}, zmm2</i>	HVM	V/V	AVX512F	Converts 8 packed unsigned quad-word integers from <i>zmm2</i> into 8 packed unsigned double-word integers in <i>ymm1/m256</i> using unsigned saturation subject to writemask <i>k1</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
HVM	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

### Description

VPMOVQW down converts 64-bit integer elements in the source operand (the second operand) into packed double-words using truncation. VPMOVSQW converts signed 64-bit integers into packed signed doublewords using signed saturation. VPMOVUSQW convert unsigned quad-word values into unsigned double-word values using unsigned saturation.

The source operand is a ZMM/YMM/XMM register. The destination operand is a YMM/XMM/XMM register or a 256/128/64-bit memory location.

Down-converted doubleword elements are written to the destination operand (the first operand) from the least-significant doubleword. Doubleword elements of the destination operand are updated according to the writemask. Bits (MAX\_VL-1:256/128/64) of the register destination are zeroed.

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

### Operation

#### VPMOVBQD instruction (EVEX encoded version) reg-reg form

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
  i ← j * 32
  m ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ← TruncateQuadWordToDWord (SRC[m+63:m])
    ELSE *zeroing-masking*           ; zeroing-masking
      DEST[i+31:i] ← 0
  FI
FI;
ENDFOR
DEST[MAX_VL-1:VL/2] ← 0;
```

#### VPMOVBQD instruction (EVEX encoded version) memory form

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
  i ← j * 32
  m ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ← TruncateQuadWordToDWord (SRC[m+63:m])
    ELSE *DEST[i+31:i] remains unchanged* ; merging-masking
  FI;
ENDFOR
```

#### VPMOVSQD instruction (EVEX encoded version) reg-reg form

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
  i ← j * 32
  m ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ← SaturateSignedQuadWordToDWord (SRC[m+63:m])
    ELSE
      IF *merging-masking*           ; merging-masking
        THEN *DEST[i+31:i] remains unchanged*
        ELSE *zeroing-masking*       ; zeroing-masking
          DEST[i+31:i] ← 0
      FI
  FI;
ENDFOR
DEST[MAX_VL-1:VL/2] ← 0;
```

**VPMOVSQD instruction (EVEX encoded version) memory form**

```

(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
  i ← j * 32
  m ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ← SaturateSignedQuadWordToDWord (SRC[m+63:m])
    ELSE *DEST[i+31:i] remains unchanged* ;merging-masking
  FI;
ENDFOR

```

**VPMOVUSQD instruction (EVEX encoded version) reg-reg form**

```

(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
  i ← j * 32
  m ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ← SaturateUnsignedQuadWordToDWord (SRC[m+63:m])
    ELSE
      IF *merging-masking* ;merging-masking
        THEN *DEST[i+31:i] remains unchanged*
      ELSE *zeroing-masking* ;zeroing-masking
        DEST[i+31:i] ← 0
      FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL/2] ← 0;

```

**VPMOVUSQD instruction (EVEX encoded version) memory form**

```

(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
  i ← j * 32
  m ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ← SaturateUnsignedQuadWordToDWord (SRC[m+63:m])
    ELSE *DEST[i+31:i] remains unchanged* ;merging-masking
  FI;
ENDFOR

```

**Intel C/C++ Compiler Intrinsic Equivalents**

```

VPMOVQD __m256i __mm512_cvtepi64_epi32(__m512i a);
VPMOVQD __m256i __mm512_mask_cvtepi64_epi32(__m256i s, __mmask8 k, __m512i a);
VPMOVQD __m256i __mm512_maskz_cvtepi64_epi32(__mmask8 k, __m512i a);
VPMOVQD void __mm512_mask_cvtepi64_storeu_epi32(void * d, __mmask8 k, __m512i a);
VPMOVSQD __m256i __mm512_cvtsepi64_epi32(__m512i a);
VPMOVSQD __m256i __mm512_mask_cvtsepi64_epi32(__m256i s, __mmask8 k, __m512i a);
VPMOVSQD __m256i __mm512_maskz_cvtsepi64_epi32(__mmask8 k, __m512i a);
VPMOVSQD void __mm512_mask_cvtsepi64_storeu_epi32(void * d, __mmask8 k, __m512i a);
VPMOVUSQD __m256i __mm512_cvtusepi64_epi32(__m512i a);
VPMOVUSQD __m256i __mm512_mask_cvtusepi64_epi32(__m256i s, __mmask8 k, __m512i a);
VPMOVUSQD __m256i __mm512_maskz_cvtusepi64_epi32(__mmask8 k, __m512i a);
VPMOVUSQD void __mm512_mask_cvtusepi64_storeu_epi32(void * d, __mmask8 k, __m512i a);
VPMOVUSQD __m128i __mm256_cvtusepi64_epi32(__m256i a);
VPMOVUSQD __m128i __mm256_mask_cvtusepi64_epi32(__m128i a, __mmask8 k, __m256i b);

```



```

VPMOVUSQD __m128i _mm256_maskz_cvtusepi64_epi32( __mmask8 k, __m256i b);
VPMOVUSQD void _mm256_mask_cvtusepi64_storeu_epi32(void * , __mmask8 k, __m256i b);
VPMOVUSQD __m128i _mm_cvtusepi64_epi32(__m128i a);
VPMOVUSQD __m128i _mm_mask_cvtusepi64_epi32(__m128i a, __mmask8 k, __m128i b);
VPMOVUSQD __m128i _mm_maskz_cvtusepi64_epi32( __mmask8 k, __m128i b);
VPMOVUSQD void _mm_mask_cvtusepi64_storeu_epi32(void * , __mmask8 k, __m128i b);
VPMOVSQD __m128i _mm256_cvtsepi64_epi32(__m256i a);
VPMOVSQD __m128i _mm256_mask_cvtsepi64_epi32(__m128i a, __mmask8 k, __m256i b);
VPMOVSQD __m128i _mm256_maskz_cvtsepi64_epi32( __mmask8 k, __m256i b);
VPMOVSQD void _mm256_mask_cvtsepi64_storeu_epi32(void * , __mmask8 k, __m256i b);
VPMOVSQD __m128i _mm_cvtsepi64_epi32(__m128i a);
VPMOVSQD __m128i _mm_mask_cvtsepi64_epi32(__m128i a, __mmask8 k, __m128i b);
VPMOVSQD __m128i _mm_maskz_cvtsepi64_epi32( __mmask8 k, __m128i b);
VPMOVSQD void _mm_mask_cvtsepi64_storeu_epi32(void * , __mmask8 k, __m128i b);
VPMOVQD __m128i _mm256_cvtepi64_epi32(__m256i a);
VPMOVQD __m128i _mm256_mask_cvtepi64_epi32(__m128i a, __mmask8 k, __m256i b);
VPMOVQD __m128i _mm256_maskz_cvtepi64_epi32( __mmask8 k, __m256i b);
VPMOVQD void _mm256_mask_cvtepi64_storeu_epi32(void * , __mmask8 k, __m256i b);
VPMOVQD __m128i _mm_cvtepi64_epi32(__m128i a);
VPMOVQD __m128i _mm_mask_cvtepi64_epi32(__m128i a, __mmask8 k, __m128i b);
VPMOVQD __m128i _mm_maskz_cvtepi64_epi32( __mmask8 k, __m128i b);
VPMOVQD void _mm_mask_cvtepi64_storeu_epi32(void * , __mmask8 k, __m128i b);

```

### SIMD Floating-Point Exceptions

None

### Other Exceptions

EVEX-encoded instruction, see Exceptions Type E6.

#UD                      If EVEX.vvvv != 1111B.

## VPMOVD/VPMSDB/VPMSDB—Down Convert DWord to Byte

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.F3.0F38.W0 31 /r VPMOVD <i>xmm1/m32</i> { <i>k1</i> }{ <i>z</i> }, <i>xmm2</i>	QVM	V/V	AVX512VL AVX512F	Converts 4 packed double-word integers from <i>xmm2</i> into 4 packed byte integers in <i>xmm1/m32</i> with truncation under writemask <i>k1</i> .
EVEX.128.F3.0F38.W0 21 /r VPMSDB <i>xmm1/m32</i> { <i>k1</i> }{ <i>z</i> }, <i>xmm2</i>	QVM	V/V	AVX512VL AVX512F	Converts 4 packed signed double-word integers from <i>xmm2</i> into 4 packed signed byte integers in <i>xmm1/m32</i> using signed saturation under writemask <i>k1</i> .
EVEX.128.F3.0F38.W0 11 /r VPMSDB <i>xmm1/m32</i> { <i>k1</i> }{ <i>z</i> }, <i>xmm2</i>	QVM	V/V	AVX512VL AVX512F	Converts 4 packed unsigned double-word integers from <i>xmm2</i> into 4 packed unsigned byte integers in <i>xmm1/m32</i> using unsigned saturation under writemask <i>k1</i> .
EVEX.256.F3.0F38.W0 31 /r VPMOVD <i>xmm1/m64</i> { <i>k1</i> }{ <i>z</i> }, <i>ymm2</i>	QVM	V/V	AVX512VL AVX512F	Converts 8 packed double-word integers from <i>ymm2</i> into 8 packed byte integers in <i>xmm1/m64</i> with truncation under writemask <i>k1</i> .
EVEX.256.F3.0F38.W0 21 /r VPMSDB <i>xmm1/m64</i> { <i>k1</i> }{ <i>z</i> }, <i>ymm2</i>	QVM	V/V	AVX512VL AVX512F	Converts 8 packed signed double-word integers from <i>ymm2</i> into 8 packed signed byte integers in <i>xmm1/m64</i> using signed saturation under writemask <i>k1</i> .
EVEX.256.F3.0F38.W0 11 /r VPMSDB <i>xmm1/m64</i> { <i>k1</i> }{ <i>z</i> }, <i>ymm2</i>	QVM	V/V	AVX512VL AVX512F	Converts 8 packed unsigned double-word integers from <i>ymm2</i> into 8 packed unsigned byte integers in <i>xmm1/m64</i> using unsigned saturation under writemask <i>k1</i> .
EVEX.512.F3.0F38.W0 31 /r VPMOVD <i>xmm1/m128</i> { <i>k1</i> }{ <i>z</i> }, <i>zmm2</i>	QVM	V/V	AVX512F	Converts 16 packed double-word integers from <i>zmm2</i> into 16 packed byte integers in <i>xmm1/m128</i> with truncation under writemask <i>k1</i> .
EVEX.512.F3.0F38.W0 21 /r VPMSDB <i>xmm1/m128</i> { <i>k1</i> }{ <i>z</i> }, <i>zmm2</i>	QVM	V/V	AVX512F	Converts 16 packed signed double-word integers from <i>zmm2</i> into 16 packed signed byte integers in <i>xmm1/m128</i> using signed saturation under writemask <i>k1</i> .
EVEX.512.F3.0F38.W0 11 /r VPMSDB <i>xmm1/m128</i> { <i>k1</i> }{ <i>z</i> }, <i>zmm2</i>	QVM	V/V	AVX512F	Converts 16 packed unsigned double-word integers from <i>zmm2</i> into 16 packed unsigned byte integers in <i>xmm1/m128</i> using unsigned saturation under writemask <i>k1</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
QVM	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

### Description

VPMOVD down converts 32-bit integer elements in the source operand (the second operand) into packed bytes using truncation. VPMSDB converts signed 32-bit integers into packed signed bytes using signed saturation. VPMSDB convert unsigned double-word values into unsigned byte values using unsigned saturation.

The source operand is a ZMM/YMM/XMM register. The destination operand is a XMM register or a 128/64/32-bit memory location.

Down-converted byte elements are written to the destination operand (the first operand) from the least-significant byte. Byte elements of the destination operand are updated according to the writemask. Bits (MAX\_VL-1:128/64/32) of the register destination are zeroed.

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

**Operation****VPMOVD B instruction (EVEX encoded versions) when dest is a register**

```

(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
  i ← j * 8
  m ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+7:i] ← TruncateDoubleWordToByte (SRC[m+31:m])
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+7:i] remains unchanged*
        ELSE *zeroing-masking* ; zeroing-masking
          DEST[i+7:i] ← 0
      FI
  FI;
ENDFOR
DEST[MAX_VL-1:VL/4] ← 0;

```

**VPMOVD B instruction (EVEX encoded versions) when dest is memory**

```

(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
  i ← j * 8
  m ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+7:i] ← TruncateDoubleWordToByte (SRC[m+31:m])
    ELSE *DEST[i+7:i] remains unchanged* ; merging-masking
  FI;
ENDFOR

```

**VPMOVSDB instruction (EVEX encoded versions) when dest is a register**

```

(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
  i ← j * 8
  m ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+7:i] ← SaturateSignedDoubleWordToByte (SRC[m+31:m])
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+7:i] remains unchanged*
        ELSE *zeroing-masking* ; zeroing-masking
          DEST[i+7:i] ← 0
      FI
  FI;
ENDFOR
DEST[MAX_VL-1:VL/4] ← 0;

```

**VPMOVSDB instruction (EVEX encoded versions) when dest is memory**

```

(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
  i ← j * 8
  m ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+7:i] ← SaturateSignedDoubleWordToByte (SRC[m+31:m])
    ELSE *DEST[i+7:i] remains unchanged* ; merging-masking
  FI;
ENDFOR

```

**VPMOVUSDB instruction (EVEX encoded versions) when dest is a register**

```

(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
  i ← j * 8
  m ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+7:i] ← SaturateUnsignedDoubleWordToByte (SRC[m+31:m])
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+7:i] remains unchanged*
      ELSE *zeroing-masking* ; zeroing-masking
        DEST[i+7:i] ← 0
      FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL/4] ← 0;

```

**VPMOVUSDB instruction (EVEX encoded versions) when dest is memory**

```

(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
  i ← j * 8
  m ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+7:i] ← SaturateUnsignedDoubleWordToByte (SRC[m+31:m])
    ELSE *DEST[i+7:i] remains unchanged* ; merging-masking
  FI;
ENDFOR

```

**Intel C/C++ Compiler Intrinsic Equivalents**

```

VPMOVDDB __m128i __mm512_cvtepi32_epi8( __m512i a);
VPMOVDDB __m128i __mm512_mask_cvtepi32_epi8(__m128i s, __mmask16 k, __m512i a);
VPMOVDDB __m128i __mm512_maskz_cvtepi32_epi8( __mmask16 k, __m512i a);
VPMOVDDB void __mm512_mask_cvtepi32_storeu_epi8(void * d, __mmask16 k, __m512i a);
VPMOVSDB __m128i __mm512_cvtsepi32_epi8( __m512i a);
VPMOVSDB __m128i __mm512_mask_cvtsepi32_epi8(__m128i s, __mmask16 k, __m512i a);
VPMOVSDB __m128i __mm512_maskz_cvtsepi32_epi8( __mmask16 k, __m512i a);
VPMOVSDB void __mm512_mask_cvtsepi32_storeu_epi8(void * d, __mmask16 k, __m512i a);
VPMOVUSDB __m128i __mm512_cvtusepi32_epi8( __m512i a);
VPMOVUSDB __m128i __mm512_mask_cvtusepi32_epi8(__m128i s, __mmask16 k, __m512i a);
VPMOVUSDB __m128i __mm512_maskz_cvtusepi32_epi8( __mmask16 k, __m512i a);
VPMOVUSDB void __mm512_mask_cvtusepi32_storeu_epi8(void * d, __mmask16 k, __m512i a);
VPMOVUSDB __m128i __mm256_cvtusepi32_epi8(__m256i a);
VPMOVUSDB __m128i __mm256_mask_cvtusepi32_epi8(__m128i a, __mmask8 k, __m256i b);

```

```

VPMOVUSDB __m128i _mm256_maskz_cvtusepi32_epi8( __mmask8 k, __m256i b);
VPMOVUSDB void _mm256_mask_cvtusepi32_storeu_epi8(void *, __mmask8 k, __m256i b);
VPMOVUSDB __m128i _mm_cvtusepi32_epi8(__m128i a);
VPMOVUSDB __m128i _mm_mask_cvtusepi32_epi8(__m128i a, __mmask8 k, __m128i b);
VPMOVUSDB __m128i _mm_maskz_cvtusepi32_epi8( __mmask8 k, __m128i b);
VPMOVUSDB void _mm_mask_cvtusepi32_storeu_epi8(void *, __mmask8 k, __m128i b);
VPMOVUSDB __m128i _mm256_cvtsepi32_epi8(__m256i a);
VPMOVUSDB __m128i _mm256_mask_cvtsepi32_epi8(__m128i a, __mmask8 k, __m256i b);
VPMOVUSDB __m128i _mm256_maskz_cvtsepi32_epi8( __mmask8 k, __m256i b);
VPMOVUSDB void _mm256_mask_cvtsepi32_storeu_epi8(void *, __mmask8 k, __m256i b);
VPMOVUSDB __m128i _mm_cvtsepi32_epi8(__m128i a);
VPMOVUSDB __m128i _mm_mask_cvtsepi32_epi8(__m128i a, __mmask8 k, __m128i b);
VPMOVUSDB __m128i _mm_maskz_cvtsepi32_epi8( __mmask8 k, __m128i b);
VPMOVUSDB void _mm_mask_cvtsepi32_storeu_epi8(void *, __mmask8 k, __m128i b);
VPMOVDB __m128i _mm256_cvtepi32_epi8(__m256i a);
VPMOVDB __m128i _mm256_mask_cvtepi32_epi8(__m128i a, __mmask8 k, __m256i b);
VPMOVDB __m128i _mm256_maskz_cvtepi32_epi8( __mmask8 k, __m256i b);
VPMOVDB void _mm256_mask_cvtepi32_storeu_epi8(void *, __mmask8 k, __m256i b);
VPMOVDB __m128i _mm_cvtepi32_epi8(__m128i a);
VPMOVDB __m128i _mm_mask_cvtepi32_epi8(__m128i a, __mmask8 k, __m128i b);
VPMOVDB __m128i _mm_maskz_cvtepi32_epi8( __mmask8 k, __m128i b);
VPMOVDB void _mm_mask_cvtepi32_storeu_epi8(void *, __mmask8 k, __m128i b);

```

### SIMD Floating-Point Exceptions

None

### Other Exceptions

EVEX-encoded instruction, see Exceptions Type E6.

#UD If EVEX.vvvv != 1111B.

## VPMOVDW/VPMOVSDW/VPMOVUSDW—Down Convert DWord to Word

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.F3.0F38.W0 33 /r VPMOVDW <i>xmm1/m64</i> {k1}{z}, <i>xmm2</i>	HVM	V/V	AVX512VL AVX512F	Converts 4 packed double-word integers from <i>xmm2</i> into 4 packed word integers in <i>xmm1/m64</i> with truncation under writemask k1.
EVEX.128.F3.0F38.W0 23 /r VPMOVSDW <i>xmm1/m64</i> {k1}{z}, <i>xmm2</i>	HVM	V/V	AVX512VL AVX512F	Converts 4 packed signed double-word integers from <i>xmm2</i> into 4 packed signed word integers in <i>xmm1/m64</i> using signed saturation under writemask k1.
EVEX.128.F3.0F38.W0 13 /r VPMOVUSDW <i>xmm1/m64</i> {k1}{z}, <i>xmm2</i>	HVM	V/V	AVX512VL AVX512F	Converts 4 packed unsigned double-word integers from <i>xmm2</i> into 4 packed unsigned word integers in <i>xmm1/m64</i> using unsigned saturation under writemask k1.
EVEX.256.F3.0F38.W0 33 /r VPMOVDW <i>xmm1/m128</i> {k1}{z}, <i>yymm2</i>	HVM	V/V	AVX512VL AVX512F	Converts 8 packed double-word integers from <i>yymm2</i> into 8 packed word integers in <i>xmm1/m128</i> with truncation under writemask k1.
EVEX.256.F3.0F38.W0 23 /r VPMOVSDW <i>xmm1/m128</i> {k1}{z}, <i>yymm2</i>	HVM	V/V	AVX512VL AVX512F	Converts 8 packed signed double-word integers from <i>yymm2</i> into 8 packed signed word integers in <i>xmm1/m128</i> using signed saturation under writemask k1.
EVEX.256.F3.0F38.W0 13 /r VPMOVUSDW <i>xmm1/m128</i> {k1}{z}, <i>yymm2</i>	HVM	V/V	AVX512VL AVX512F	Converts 8 packed unsigned double-word integers from <i>yymm2</i> into 8 packed unsigned word integers in <i>xmm1/m128</i> using unsigned saturation under writemask k1.
EVEX.512.F3.0F38.W0 33 /r VPMOVDW <i>yymm1/m256</i> {k1}{z}, <i>zmmm2</i>	HVM	V/V	AVX512F	Converts 16 packed double-word integers from <i>zmmm2</i> into 16 packed word integers in <i>yymm1/m256</i> with truncation under writemask k1.
EVEX.512.F3.0F38.W0 23 /r VPMOVSDW <i>yymm1/m256</i> {k1}{z}, <i>zmmm2</i>	HVM	V/V	AVX512F	Converts 16 packed signed double-word integers from <i>zmmm2</i> into 16 packed signed word integers in <i>yymm1/m256</i> using signed saturation under writemask k1.
EVEX.512.F3.0F38.W0 13 /r VPMOVUSDW <i>yymm1/m256</i> {k1}{z}, <i>zmmm2</i>	HVM	V/V	AVX512F	Converts 16 packed unsigned double-word integers from <i>zmmm2</i> into 16 packed unsigned word integers in <i>yymm1/m256</i> using unsigned saturation under writemask k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
HVM	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

### Description

VPMOVDW down converts 32-bit integer elements in the source operand (the second operand) into packed words using truncation. VPMOVSDW converts signed 32-bit integers into packed signed words using signed saturation. VPMOVUSDW convert unsigned double-word values into unsigned word values using unsigned saturation.

The source operand is a ZMM/YMM/XMM register. The destination operand is a YMM/XMM/XMM register or a 256/128/64-bit memory location.

Down-converted word elements are written to the destination operand (the first operand) from the least-significant word. Word elements of the destination operand are updated according to the writemask. Bits (MAX\_VL-1:256/128/64) of the register destination are zeroed.

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

**Operation****VPMOVDW instruction (EVEX encoded versions) when dest is a register**

```

(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
  i ← j * 16
  m ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+15:i] ← TruncateDoubleWordToWord (SRC[m+31:m])
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+15:i] remains unchanged*
        ELSE *zeroing-masking* ; zeroing-masking
          DEST[i+15:i] ← 0
      FI
  FI;
ENDFOR
DEST[MAX_VL-1:VL/2] ← 0;

```

**VPMOVDW instruction (EVEX encoded versions) when dest is memory**

```

(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
  i ← j * 16
  m ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+15:i] ← TruncateDoubleWordToWord (SRC[m+31:m])
    ELSE
      *DEST[i+15:i] remains unchanged* ; merging-masking
  FI;
ENDFOR

```

**VPMOVSDW instruction (EVEX encoded versions) when dest is a register**

```

(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
  i ← j * 16
  m ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+15:i] ← SaturateSignedDoubleWordToWord (SRC[m+31:m])
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+15:i] remains unchanged*
        ELSE *zeroing-masking* ; zeroing-masking
          DEST[i+15:i] ← 0
      FI
  FI;
ENDFOR
DEST[MAX_VL-1:VL/2] ← 0;

```

**VPMOVS<sub>DW</sub> instruction (EVEX encoded versions) when dest is memory**

```

(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
  i ← j * 16
  m ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+15:i] ← SaturateSignedDoubleWordToWord (SRC[m+31:m])
  ELSE
    *DEST[i+15:i] remains unchanged* ; merging-masking
  FI;
ENDFOR

```

**VPMOV<sub>USDW</sub> instruction (EVEX encoded versions) when dest is a register**

```

(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
  i ← j * 16
  m ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+15:i] ← SaturateUnsignedDoubleWordToWord (SRC[m+31:m])
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+15:i] remains unchanged*
    ELSE *zeroing-masking* ; zeroing-masking
      DEST[i+15:i] ← 0
    FI
  FI;
ENDFOR
DEST[MAX_VL-1:VL/2] ← 0;

```

**VPMOV<sub>USDW</sub> instruction (EVEX encoded versions) when dest is memory**

```

(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
  i ← j * 16
  m ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+15:i] ← SaturateUnsignedDoubleWordToWord (SRC[m+31:m])
  ELSE
    *DEST[i+15:i] remains unchanged* ; merging-masking
  FI;
ENDFOR

```

**Intel C/C++ Compiler Intrinsic Equivalents**

```

VPMOVDW __m256i __mm512_cvtepi32_epi16(__m512i a);
VPMOVDW __m256i __mm512_mask_cvtepi32_epi16(__m256i s, __mmask16 k, __m512i a);
VPMOVDW __m256i __mm512_maskz_cvtepi32_epi16(__mmask16 k, __m512i a);
VPMOVDW void __mm512_mask_cvtepi32_storeu_epi16(void * d, __mmask16 k, __m512i a);
VPMOVSDW __m256i __mm512_cvtsepi32_epi16(__m512i a);
VPMOVSDW __m256i __mm512_mask_cvtsepi32_epi16(__m256i s, __mmask16 k, __m512i a);
VPMOVSDW __m256i __mm512_maskz_cvtsepi32_epi16(__mmask16 k, __m512i a);
VPMOVSDW void __mm512_mask_cvtsepi32_storeu_epi16(void * d, __mmask16 k, __m512i a);
VPMOVUSDW __m256i __mm512_cvtusepi32_epi16(__m512i a);
VPMOVUSDW __m256i __mm512_mask_cvtusepi32_epi16(__m256i s, __mmask16 k, __m512i a);
VPMOVUSDW __m256i __mm512_maskz_cvtusepi32_epi16(__mmask16 k, __m512i a);
VPMOVUSDW void __mm512_mask_cvtusepi32_storeu_epi16(void * d, __mmask16 k, __m512i a);

```



VPMOVUSDW \_\_m128i \_mm256\_cvtusepi32\_epi16(\_\_m256i a);  
 VPMOVUSDW \_\_m128i \_mm256\_mask\_cvtusepi32\_epi16(\_\_m128i a, \_\_mmask8 k, \_\_m256i b);  
 VPMOVUSDW \_\_m128i \_mm256\_maskz\_cvtusepi32\_epi16(\_\_mmask8 k, \_\_m256i b);  
 VPMOVUSDW void \_mm256\_mask\_cvtusepi32\_storeu\_epi16(void \*, \_\_mmask8 k, \_\_m256i b);  
 VPMOVUSDW \_\_m128i \_mm\_cvtusepi32\_epi16(\_\_m128i a);  
 VPMOVUSDW \_\_m128i \_mm\_mask\_cvtusepi32\_epi16(\_\_m128i a, \_\_mmask8 k, \_\_m128i b);  
 VPMOVUSDW \_\_m128i \_mm\_maskz\_cvtusepi32\_epi16(\_\_mmask8 k, \_\_m128i b);  
 VPMOVUSDW void \_mm\_mask\_cvtusepi32\_storeu\_epi16(void \*, \_\_mmask8 k, \_\_m128i b);  
 VPMOVSDW \_\_m128i \_mm256\_cvtsepi32\_epi16(\_\_m256i a);  
 VPMOVSDW \_\_m128i \_mm256\_mask\_cvtsepi32\_epi16(\_\_m128i a, \_\_mmask8 k, \_\_m256i b);  
 VPMOVSDW \_\_m128i \_mm256\_maskz\_cvtsepi32\_epi16(\_\_mmask8 k, \_\_m256i b);  
 VPMOVSDW void \_mm256\_mask\_cvtsepi32\_storeu\_epi16(void \*, \_\_mmask8 k, \_\_m256i b);  
 VPMOVSDW \_\_m128i \_mm\_cvtsepi32\_epi16(\_\_m128i a);  
 VPMOVSDW \_\_m128i \_mm\_mask\_cvtsepi32\_epi16(\_\_m128i a, \_\_mmask8 k, \_\_m128i b);  
 VPMOVSDW \_\_m128i \_mm\_maskz\_cvtsepi32\_epi16(\_\_mmask8 k, \_\_m128i b);  
 VPMOVSDW void \_mm\_mask\_cvtsepi32\_storeu\_epi16(void \*, \_\_mmask8 k, \_\_m128i b);  
 VPMOVDW \_\_m128i \_mm256\_cvtepi32\_epi16(\_\_m256i a);  
 VPMOVDW \_\_m128i \_mm256\_mask\_cvtepi32\_epi16(\_\_m128i a, \_\_mmask8 k, \_\_m256i b);  
 VPMOVDW \_\_m128i \_mm256\_maskz\_cvtepi32\_epi16(\_\_mmask8 k, \_\_m256i b);  
 VPMOVDW void \_mm256\_mask\_cvtepi32\_storeu\_epi16(void \*, \_\_mmask8 k, \_\_m256i b);  
 VPMOVDW \_\_m128i \_mm\_cvtepi32\_epi16(\_\_m128i a);  
 VPMOVDW \_\_m128i \_mm\_mask\_cvtepi32\_epi16(\_\_m128i a, \_\_mmask8 k, \_\_m128i b);  
 VPMOVDW \_\_m128i \_mm\_maskz\_cvtepi32\_epi16(\_\_mmask8 k, \_\_m128i b);  
 VPMOVDW void \_mm\_mask\_cvtepi32\_storeu\_epi16(void \*, \_\_mmask8 k, \_\_m128i b);

### SIMD Floating-Point Exceptions

None

### Other Exceptions

EVEX-encoded instruction, see Exceptions Type E6.

#UD If EVEX.vvvv != 1111B.

## VPMOVWB/VPMOVSWB/VPMOVUSWB—Down Convert Word to Byte

Opcode/ Instruction	Op/ En	64/32 bitMode Support	CPUID Feature Flag	Description
EVEX.128.F3.0F38.W0 30 /r VPMOVWB <i>xmm1/m64</i> { <i>k1</i> }{ <i>z</i> }, <i>xmm2</i>	HVM	V/V	AVX512VL AVX512BW	Converts 8 packed word integers from <i>xmm2</i> into 8 packed bytes in <i>xmm1/m64</i> with truncation under writemask <i>k1</i> .
EVEX.128.F3.0F38.W0 20 /r VPMOVSWB <i>xmm1/m64</i> { <i>k1</i> }{ <i>z</i> }, <i>xmm2</i>	HVM	V/V	AVX512VL AVX512BW	Converts 8 packed signed word integers from <i>xmm2</i> into 8 packed signed bytes in <i>xmm1/m64</i> using signed saturation under writemask <i>k1</i> .
EVEX.128.F3.0F38.W0 10 /r VPMOVUSWB <i>xmm1/m64</i> { <i>k1</i> }{ <i>z</i> }, <i>xmm2</i>	HVM	V/V	AVX512VL AVX512BW	Converts 8 packed unsigned word integers from <i>xmm2</i> into 8 packed unsigned bytes in <i>xmm1/m64</i> using unsigned saturation under writemask <i>k1</i> .
EVEX.256.F3.0F38.W0 30 /r VPMOVWB <i>xmm1/m128</i> { <i>k1</i> }{ <i>z</i> }, <i>ymm2</i>	HVM	V/V	AVX512VL AVX512BW	Converts 16 packed word integers from <i>ymm2</i> into 16 packed bytes in <i>xmm1/m128</i> with truncation under writemask <i>k1</i> .
EVEX.256.F3.0F38.W0 20 /r VPMOVSWB <i>xmm1/m128</i> { <i>k1</i> }{ <i>z</i> }, <i>ymm2</i>	HVM	V/V	AVX512VL AVX512BW	Converts 16 packed signed word integers from <i>ymm2</i> into 16 packed signed bytes in <i>xmm1/m128</i> using signed saturation under writemask <i>k1</i> .
EVEX.256.F3.0F38.W0 10 /r VPMOVUSWB <i>xmm1/m128</i> { <i>k1</i> }{ <i>z</i> }, <i>ymm2</i>	HVM	V/V	AVX512VL AVX512BW	Converts 16 packed unsigned word integers from <i>ymm2</i> into 16 packed unsigned bytes in <i>xmm1/m128</i> using unsigned saturation under writemask <i>k1</i> .
EVEX.512.F3.0F38.W0 30 /r VPMOVWB <i>ymm1/m256</i> { <i>k1</i> }{ <i>z</i> }, <i>zmm2</i>	HVM	V/V	AVX512BW	Converts 32 packed word integers from <i>zmm2</i> into 32 packed bytes in <i>ymm1/m256</i> with truncation under writemask <i>k1</i> .
EVEX.512.F3.0F38.W0 20 /r VPMOVSWB <i>ymm1/m256</i> { <i>k1</i> }{ <i>z</i> }, <i>zmm2</i>	HVM	V/V	AVX512BW	Converts 32 packed signed word integers from <i>zmm2</i> into 32 packed signed bytes in <i>ymm1/m256</i> using signed saturation under writemask <i>k1</i> .
EVEX.512.F3.0F38.W0 10 /r VPMOVUSWB <i>ymm1/m256</i> { <i>k1</i> }{ <i>z</i> }, <i>zmm2</i>	HVM	V/V	AVX512BW	Converts 32 packed unsigned word integers from <i>zmm2</i> into 32 packed unsigned bytes in <i>ymm1/m256</i> using unsigned saturation under writemask <i>k1</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
HVM	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

#### Description

VPMOVWB down converts 16-bit integers into packed bytes using truncation. VPMOVSWB converts signed 16-bit integers into packed signed bytes using signed saturation. VPMOVUSWB convert unsigned word values into unsigned byte values using unsigned saturation.

The source operand is a ZMM/YMM/XMM register. The destination operand is a YMM/XMM/XMM register or a 256/128/64-bit memory location.

Down-converted byte elements are written to the destination operand (the first operand) from the least-significant byte. Byte elements of the destination operand are updated according to the writemask. Bits (MAX\_VL-1:256/128/64) of the register destination are zeroed.

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

### Operation

#### VPMOVB instruction (EVEX encoded versions) when dest is a register

(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j ← 0 TO KI-1

  i ← j \* 8

  m ← j \* 16

  IF k1[j] OR \*no writemask\*

    THEN DEST[i+7:i] ← TruncateWordToByte (SRC[m+15:m])

  ELSE

    IF \*merging-masking\* ; merging-masking

      THEN \*DEST[i+7:i] remains unchanged\*

    ELSE \*zeroing-masking\* ; zeroing-masking

      DEST[i+7:i] = 0

  FI

  FI;

ENDFOR

DEST[MAX\_VL-1:VL/2] ← 0;

#### VPMOVB instruction (EVEX encoded versions) when dest is memory

(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j ← 0 TO KI-1

  i ← j \* 8

  m ← j \* 16

  IF k1[j] OR \*no writemask\*

    THEN DEST[i+7:i] ← TruncateWordToByte (SRC[m+15:m])

  ELSE

    \*DEST[i+7:i] remains unchanged\* ; merging-masking

  FI;

ENDFOR

#### VPMOVS WB instruction (EVEX encoded versions) when dest is a register

(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j ← 0 TO KI-1

  i ← j \* 8

  m ← j \* 16

  IF k1[j] OR \*no writemask\*

    THEN DEST[i+7:i] ← SaturateSignedWordToByte (SRC[m+15:m])

  ELSE

    IF \*merging-masking\* ; merging-masking

      THEN \*DEST[i+7:i] remains unchanged\*

    ELSE \*zeroing-masking\* ; zeroing-masking

      DEST[i+7:i] = 0

  FI

  FI;

ENDFOR

DEST[MAX\_VL-1:VL/2] ← 0;

**VPMOVS WB instruction (EVEX encoded versions) when dest is memory**

```

(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j ← 0 TO KI-1
  i ← j * 8
  m ← j * 16
  IF k1[j] OR *no writemask*
    THEN DEST[i+7:i] ← SaturateSignedWordToByte (SRC[m+15:m])
  ELSE
    *DEST[i+7:i] remains unchanged* ; merging-masking
  FI;
ENDFOR

```

**VPMOVUS WB instruction (EVEX encoded versions) when dest is a register**

```

(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j ← 0 TO KI-1
  i ← j * 8
  m ← j * 16
  IF k1[j] OR *no writemask*
    THEN DEST[i+7:i] ← SaturateUnsignedWordToByte (SRC[m+15:m])
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+7:i] remains unchanged*
    ELSE *zeroing-masking* ; zeroing-masking
      DEST[i+7:i] = 0
    FI
  FI;
ENDFOR
DEST[MAX_VL-1:VL/2] ← 0;

```

**VPMOVUS WB instruction (EVEX encoded versions) when dest is memory**

```

(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j ← 0 TO KI-1
  i ← j * 8
  m ← j * 16
  IF k1[j] OR *no writemask*
    THEN DEST[i+7:i] ← SaturateUnsignedWordToByte (SRC[m+15:m])
  ELSE
    *DEST[i+7:i] remains unchanged* ; merging-masking
  FI;
ENDFOR

```

**Intel C/C++ Compiler Intrinsic Equivalents**

```

VPMOVUSWB __m256i __mm512_cvtusepi16_epi8(__m512i a);
VPMOVUSWB __m256i __mm512_mask_cvtusepi16_epi8(__m256i a, __mmask32 k, __m512i b);
VPMOVUSWB __m256i __mm512_maskz_cvtusepi16_epi8(__mmask32 k, __m512i b);
VPMOVUSWB void __mm512_mask_cvtusepi16_storeu_epi8(void *, __mmask32 k, __m512i b);
VPMOVSWB __m256i __mm512_cvtsepi16_epi8(__m512i a);
VPMOVSWB __m256i __mm512_mask_cvtsepi16_epi8(__m256i a, __mmask32 k, __m512i b);
VPMOVSWB __m256i __mm512_maskz_cvtsepi16_epi8(__mmask32 k, __m512i b);
VPMOVSWB void __mm512_mask_cvtsepi16_storeu_epi8(void *, __mmask32 k, __m512i b);
VPMOVWB __m256i __mm512_cvtepi16_epi8(__m512i a);
VPMOVWB __m256i __mm512_mask_cvtepi16_epi8(__m256i a, __mmask32 k, __m512i b);
VPMOVWB __m256i __mm512_maskz_cvtepi16_epi8(__mmask32 k, __m512i b);
VPMOVWB void __mm512_mask_cvtepi16_storeu_epi8(void *, __mmask32 k, __m512i b);

```

```

VPMOVUSWB __m128i __mm256_cvtusepi16_epi8(__m256i a);
VPMOVUSWB __m128i __mm256_mask_cvtusepi16_epi8(__m128i a, __mmask16 k, __m256i b);
VPMOVUSWB __m128i __mm256_maskz_cvtusepi16_epi8(__mmask16 k, __m256i b);
VPMOVUSWB void __mm256_mask_cvtusepi16_storeu_epi8(void *, __mmask16 k, __m256i b);
VPMOVUSWB __m128i __mm_cvtusepi16_epi8(__m128i a);
VPMOVUSWB __m128i __mm_mask_cvtusepi16_epi8(__m128i a, __mmask8 k, __m128i b);
VPMOVUSWB __m128i __mm_maskz_cvtusepi16_epi8(__mmask8 k, __m128i b);
VPMOVUSWB void __mm_mask_cvtusepi16_storeu_epi8(void *, __mmask8 k, __m128i b);
VPMOVSWB __m128i __mm256_cvtsepi16_epi8(__m256i a);
VPMOVSWB __m128i __mm256_mask_cvtsepi16_epi8(__m128i a, __mmask16 k, __m256i b);
VPMOVSWB __m128i __mm256_maskz_cvtsepi16_epi8(__mmask16 k, __m256i b);
VPMOVSWB void __mm256_mask_cvtsepi16_storeu_epi8(void *, __mmask16 k, __m256i b);
VPMOVSWB __m128i __mm_cvtsepi16_epi8(__m128i a);
VPMOVSWB __m128i __mm_mask_cvtsepi16_epi8(__m128i a, __mmask8 k, __m128i b);
VPMOVSWB __m128i __mm_maskz_cvtsepi16_epi8(__mmask8 k, __m128i b);
VPMOVSWB void __mm_mask_cvtsepi16_storeu_epi8(void *, __mmask8 k, __m128i b);
VPMOVWB __m128i __mm256_cvtepi16_epi8(__m256i a);
VPMOVWB __m128i __mm256_mask_cvtepi16_epi8(__m128i a, __mmask16 k, __m256i b);
VPMOVWB __m128i __mm256_maskz_cvtepi16_epi8(__mmask16 k, __m256i b);
VPMOVWB void __mm256_mask_cvtepi16_storeu_epi8(void *, __mmask16 k, __m256i b);
VPMOVWB __m128i __mm_cvtepi16_epi8(__m128i a);
VPMOVWB __m128i __mm_mask_cvtepi16_epi8(__m128i a, __mmask8 k, __m128i b);
VPMOVWB __m128i __mm_maskz_cvtepi16_epi8(__mmask8 k, __m128i b);
VPMOVWB void __mm_mask_cvtepi16_storeu_epi8(void *, __mmask8 k, __m128i b);

```

### SIMD Floating-Point Exceptions

None

### Other Exceptions

EVEX-encoded instruction, see Exceptions Type E6NF

#UD If EVEX.vvvv != 1111B.

## PROLD/PROLVD/PROLQ/PROLVQ—Bit Rotate Left

Opcode/ Instruction	Op / En	64/32 bitMode Support	CPUID Feature Flag	Description
EVEX.NDS.128.66.0F38.W0 15 /r VPROLVD xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	FV-RVM	V/V	AVX512VL AVX512F	Rotate doublewords in xmm2 left by count in the corresponding element of xmm3/m128/m32bcst. Result written to xmm1 under writemask k1.
EVEX.NDD.128.66.0F.W0 72 /1 ib VPROLD xmm1 {k1}{z}, xmm2/m128/m32bcst, imm8	FV-VMI	V/V	AVX512VL AVX512F	Rotate doublewords in xmm2/m128/m32bcst left by imm8. Result written to xmm1 using writemask k1.
EVEX.NDS.128.66.0F38.W1 15 /r VPROLVQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	FV-RVM	V/V	AVX512VL AVX512F	Rotate quadwords in xmm2 left by count in the corresponding element of xmm3/m128/m64bcst. Result written to xmm1 under writemask k1.
EVEX.NDD.128.66.0F.W1 72 /1 ib VPROLQ xmm1 {k1}{z}, xmm2/m128/m64bcst, imm8	FV-VMI	V/V	AVX512VL AVX512F	Rotate quadwords in xmm2/m128/m64bcst left by imm8. Result written to xmm1 using writemask k1.
EVEX.NDS.256.66.0F38.W0 15 /r VPROLVD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	FV-RVM	V/V	AVX512VL AVX512F	Rotate doublewords in ymm2 left by count in the corresponding element of ymm3/m256/m32bcst. Result written to ymm1 under writemask k1.
EVEX.NDD.256.66.0F.W0 72 /1 ib VPROLD ymm1 {k1}{z}, ymm2/m256/m32bcst, imm8	FV-VMI	V/V	AVX512VL AVX512F	Rotate doublewords in ymm2/m256/m32bcst left by imm8. Result written to ymm1 using writemask k1.
EVEX.NDS.256.66.0F38.W1 15 /r VPROLVQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	FV-RVM	V/V	AVX512VL AVX512F	Rotate quadwords in ymm2 left by count in the corresponding element of ymm3/m256/m64bcst. Result written to ymm1 under writemask k1.
EVEX.NDD.256.66.0F.W1 72 /1 ib VPROLQ ymm1 {k1}{z}, ymm2/m256/m64bcst, imm8	FV-VMI	V/V	AVX512VL AVX512F	Rotate quadwords in ymm2/m256/m64bcst left by imm8. Result written to ymm1 using writemask k1.
EVEX.NDS.512.66.0F38.W0 15 /r VPROLVD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	FV-RVM	V/V	AVX512F	Rotate left of doublewords in zmm2 by count in the corresponding element of zmm3/m512/m32bcst. Result written to zmm1 using writemask k1.
EVEX.NDD.512.66.0F.W0 72 /1 ib VPROLD zmm1 {k1}{z}, zmm2/m512/m32bcst, imm8	FV-VMI	V/V	AVX512F	Rotate left of doublewords in zmm2/m512/m32bcst by imm8. Result written to zmm1 using writemask k1.
EVEX.NDS.512.66.0F38.W1 15 /r VPROLVQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	FV-RVM	V/V	AVX512F	Rotate quadwords in zmm2 left by count in the corresponding element of zmm3/m512/m64bcst. Result written to zmm1 under writemask k1.
EVEX.NDD.512.66.0F.W1 72 /1 ib VPROLQ zmm1 {k1}{z}, zmm2/m512/m64bcst, imm8	FV-VMI	V/V	AVX512F	Rotate quadwords in zmm2/m512/m64bcst left by imm8. Result written to zmm1 using writemask k1.

## Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
FV-VMI	VEX.vvvv (w)	ModRM:r/m (R)	Imm8	NA
FV-RVM	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

## Description

Rotates the bits in the individual data elements (doublewords, or quadword) in the first source operand to the left by the number of bits specified in the count operand. If the value specified by the count operand is greater than 31 (for doublewords), or 63 (for a quadword), then the count operand modulo the data size (32 or 64) is used.

EVEX.128 encoded version: The destination operand is a XMM register. The source operand is a XMM register or a memory location (for immediate form). The count operand can come either from an XMM register or a memory location or an 8-bit immediate. Bits (MAX\_VL-1:128) of the corresponding ZMM register are zeroed.

EVEX.256 encoded version: The destination operand is a YMM register. The source operand is a YMM register or a memory location (for immediate form). The count operand can come either from an XMM register or a memory location or an 8-bit immediate. Bits (MAX\_VL-1:256) of the corresponding ZMM register are zeroed.

EVEX.512 encoded version: The destination operand is a ZMM register updated according to the writemask. For the count operand in immediate form, the source operand can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32/64-bit memory location, the count operand is an 8-bit immediate. For the count operand in variable form, the first source operand (the second operand) is a ZMM register and the counter operand (the third operand) is a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32/64-bit memory location.

## Operation

```
LEFT_ROTATE_DWORDS(SRC, COUNT_SRC)
COUNT ← COUNT_SRC modulo 32;
DEST[31:0] ← (SRC << COUNT) | (SRC >> (32 - COUNT));
```

```
LEFT_ROTATE_QWORDS(SRC, COUNT_SRC)
COUNT ← COUNT_SRC modulo 64;
DEST[63:0] ← (SRC << COUNT) | (SRC >> (64 - COUNT));
```

**VPROLD (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

  i ← j \* 32

  IF k1[j] OR \*no writemask\* THEN

    IF (EVEX.b = 1) AND (SRC1 \*is memory\*)

      THEN DEST[i+31:i] ← LEFT\_ROTATE\_DWORDS(SRC1[31:0], imm8)

      ELSE DEST[i+31:i] ← LEFT\_ROTATE\_DWORDS(SRC1[i+31:i], imm8)

    FI;

  ELSE

    IF \*merging-masking\* ; merging-masking

      THEN \*DEST[i+31:i] remains unchanged\*

    ELSE \*zeroing-masking\* ; zeroing-masking

      DEST[i+31:i] ← 0

    FI

  FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VPROLVD (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\* THEN

IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

THEN DEST[i+31:i] ← LEFT\_ROTATE\_DWORDS(SRC1[i+31:i], SRC2[31:0])

ELSE DEST[i+31:i] ← LEFT\_ROTATE\_DWORDS(SRC1[i+31:i], SRC2[i+31:i])

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+31:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VPROLQ (EVEX encoded versions)**

(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) AND (SRC1 \*is memory\*)

THEN DEST[i+63:i] ← LEFT\_ROTATE\_QWORDS(SRC1[63:0], imm8)

ELSE DEST[i+63:i] ← LEFT\_ROTATE\_QWORDS(SRC1[i+63:i], imm8)

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VPROLVQ (EVEX encoded versions)**

(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) AND (SRC2 \*is memory\*)

THEN DEST[i+63:i] ← LEFT\_ROTATE\_QWORDS(SRC1[i+63:i], SRC2[63:0])

ELSE DEST[i+63:i] ← LEFT\_ROTATE\_QWORDS(SRC1[i+63:i], SRC2[i+63:i])

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;



ENDFOR  
 DEST[MAX\_VL-1:VL] ← 0

### Intel C/C++ Compiler Intrinsic Equivalent

VPROLD \_\_m512i \_\_mm512\_rol\_epi32(\_\_m512i a, int imm);  
 VPROLD \_\_m512i \_\_mm512\_mask\_rol\_epi32(\_\_m512i a, \_\_mmask16 k, \_\_m512i b, int imm);  
 VPROLD \_\_m512i \_\_mm512\_maskz\_rol\_epi32(\_\_mmask16 k, \_\_m512i a, int imm);  
 VPROLD \_\_m256i \_\_mm256\_rol\_epi32(\_\_m256i a, int imm);  
 VPROLD \_\_m256i \_\_mm256\_mask\_rol\_epi32(\_\_m256i a, \_\_mmask8 k, \_\_m256i b, int imm);  
 VPROLD \_\_m256i \_\_mm256\_maskz\_rol\_epi32(\_\_mmask8 k, \_\_m256i a, int imm);  
 VPROLD \_\_m128i \_\_mm\_rol\_epi32(\_\_m128i a, int imm);  
 VPROLD \_\_m128i \_\_mm\_mask\_rol\_epi32(\_\_m128i a, \_\_mmask8 k, \_\_m128i b, int imm);  
 VPROLD \_\_m128i \_\_mm\_maskz\_rol\_epi32(\_\_mmask8 k, \_\_m128i a, int imm);  
 VPROLQ \_\_m512i \_\_mm512\_rol\_epi64(\_\_m512i a, int imm);  
 VPROLQ \_\_m512i \_\_mm512\_mask\_rol\_epi64(\_\_m512i a, \_\_mmask8 k, \_\_m512i b, int imm);  
 VPROLQ \_\_m512i \_\_mm512\_maskz\_rol\_epi64(\_\_mmask8 k, \_\_m512i a, int imm);  
 VPROLQ \_\_m256i \_\_mm256\_rol\_epi64(\_\_m256i a, int imm);  
 VPROLQ \_\_m256i \_\_mm256\_mask\_rol\_epi64(\_\_m256i a, \_\_mmask8 k, \_\_m256i b, int imm);  
 VPROLQ \_\_m256i \_\_mm256\_maskz\_rol\_epi64(\_\_mmask8 k, \_\_m256i a, int imm);  
 VPROLQ \_\_m128i \_\_mm\_rol\_epi64(\_\_m128i a, int imm);  
 VPROLQ \_\_m128i \_\_mm\_mask\_rol\_epi64(\_\_m128i a, \_\_mmask8 k, \_\_m128i b, int imm);  
 VPROLQ \_\_m128i \_\_mm\_maskz\_rol\_epi64(\_\_mmask8 k, \_\_m128i a, int imm);  
 VPROLVD \_\_m512i \_\_mm512\_rolv\_epi32(\_\_m512i a, \_\_m512i cnt);  
 VPROLVD \_\_m512i \_\_mm512\_mask\_rolv\_epi32(\_\_m512i a, \_\_mmask16 k, \_\_m512i b, \_\_m512i cnt);  
 VPROLVD \_\_m512i \_\_mm512\_maskz\_rolv\_epi32(\_\_mmask16 k, \_\_m512i a, \_\_m512i cnt);  
 VPROLVD \_\_m256i \_\_mm256\_rolv\_epi32(\_\_m256i a, \_\_m256i cnt);  
 VPROLVD \_\_m256i \_\_mm256\_mask\_rolv\_epi32(\_\_m256i a, \_\_mmask8 k, \_\_m256i b, \_\_m256i cnt);  
 VPROLVD \_\_m256i \_\_mm256\_maskz\_rolv\_epi32(\_\_mmask8 k, \_\_m256i a, \_\_m256i cnt);  
 VPROLVD \_\_m128i \_\_mm\_rolv\_epi32(\_\_m128i a, \_\_m128i cnt);  
 VPROLVD \_\_m128i \_\_mm\_mask\_rolv\_epi32(\_\_m128i a, \_\_mmask8 k, \_\_m128i b, \_\_m128i cnt);  
 VPROLVD \_\_m128i \_\_mm\_maskz\_rolv\_epi32(\_\_mmask8 k, \_\_m128i a, \_\_m128i cnt);  
 VPROLVQ \_\_m512i \_\_mm512\_rolv\_epi64(\_\_m512i a, \_\_m512i cnt);  
 VPROLVQ \_\_m512i \_\_mm512\_mask\_rolv\_epi64(\_\_m512i a, \_\_mmask8 k, \_\_m512i b, \_\_m512i cnt);  
 VPROLVQ \_\_m512i \_\_mm512\_maskz\_rolv\_epi64(\_\_mmask8 k, \_\_m512i a, \_\_m512i cnt);  
 VPROLVQ \_\_m256i \_\_mm256\_rolv\_epi64(\_\_m256i a, \_\_m256i cnt);  
 VPROLVQ \_\_m256i \_\_mm256\_mask\_rolv\_epi64(\_\_m256i a, \_\_mmask8 k, \_\_m256i b, \_\_m256i cnt);  
 VPROLVQ \_\_m256i \_\_mm256\_maskz\_rolv\_epi64(\_\_mmask8 k, \_\_m256i a, \_\_m256i cnt);  
 VPROLVQ \_\_m128i \_\_mm\_rolv\_epi64(\_\_m128i a, \_\_m128i cnt);  
 VPROLVQ \_\_m128i \_\_mm\_mask\_rolv\_epi64(\_\_m128i a, \_\_mmask8 k, \_\_m128i b, \_\_m128i cnt);  
 VPROLVQ \_\_m128i \_\_mm\_maskz\_rolv\_epi64(\_\_mmask8 k, \_\_m128i a, \_\_m128i cnt);

### SIMD Floating-Point Exceptions

None

### Other Exceptions

EVEX-encoded instruction, see Exceptions Type E4.

**PRORD/PRORVD/PRORQ/PRORVQ—Bit Rotate Right**

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.128.66.0F38.W0 14 /r VPRORVD xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	FV-RVM	V/V	AVX512VL AVX512F	Rotate doublewords in xmm2 right by count in the corresponding element of xmm3/m128/m32bcst, store result using writemask k1.
EVEX.NDD.128.66.0F.W0 72 /0 ib VPRORD xmm1 {k1}{z}, xmm2/m128/m32bcst, imm8	FV-VMI	V/V	AVX512VL AVX512F	Rotate doublewords in xmm2/m128/m32bcst right by imm8, store result using writemask k1.
EVEX.NDS.128.66.0F38.W1 14 /r VPRORVQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	FV-RVM	V/V	AVX512VL AVX512F	Rotate quadwords in xmm2 right by count in the corresponding element of xmm3/m128/m64bcst, store result using writemask k1.
EVEX.NDD.128.66.0F.W1 72 /0 ib VPRORQ xmm1 {k1}{z}, xmm2/m128/m64bcst, imm8	FV-VMI	V/V	AVX512VL AVX512F	Rotate quadwords in xmm2/m128/m64bcst right by imm8, store result using writemask k1.
EVEX.NDS.256.66.0F38.W0 14 /r VPRORVD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	FV-RVM	V/V	AVX512VL AVX512F	Rotate doublewords in ymm2 right by count in the corresponding element of ymm3/m256/m32bcst, store using result writemask k1.
EVEX.NDD.256.66.0F.W0 72 /0 ib VPRORD ymm1 {k1}{z}, ymm2/m256/m32bcst, imm8	FV-VMI	V/V	AVX512VL AVX512F	Rotate doublewords in ymm2/m256/m32bcst right by imm8, store result using writemask k1.
EVEX.NDS.256.66.0F38.W1 14 /r VPRORVQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	FV-RVM	V/V	AVX512VL AVX512F	Rotate quadwords in ymm2 right by count in the corresponding element of ymm3/m256/m64bcst, store result using writemask k1.
EVEX.NDD.256.66.0F.W1 72 /0 ib VPRORQ ymm1 {k1}{z}, ymm2/m256/m64bcst, imm8	FV-VMI	V/V	AVX512VL AVX512F	Rotate quadwords in ymm2/m256/m64bcst right by imm8, store result using writemask k1.
EVEX.NDS.512.66.0F38.W0 14 /r VPRORVD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	FV-RVM	V/V	AVX512F	Rotate doublewords in zmm2 right by count in the corresponding element of zmm3/m512/m32bcst, store result using writemask k1.
EVEX.NDD.512.66.0F.W0 72 /0 ib VPRORD zmm1 {k1}{z}, zmm2/m512/m32bcst, imm8	FV-VMI	V/V	AVX512F	Rotate doublewords in zmm2/m512/m32bcst right by imm8, store result using writemask k1.
EVEX.NDS.512.66.0F38.W1 14 /r VPRORVQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	FV-RVM	V/V	AVX512F	Rotate quadwords in zmm2 right by count in the corresponding element of zmm3/m512/m64bcst, store result using writemask k1.
EVEX.NDD.512.66.0F.W1 72 /0 ib VPRORQ zmm1 {k1}{z}, zmm2/m512/m64bcst, imm8	FV-VMI	V/V	AVX512F	Rotate quadwords in zmm2/m512/m64bcst right by imm8, store result using writemask k1.

## Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
FV-VMI	VEX.vvvv (w)	ModRM:r/m (R)	Imm8	NA
FV-RVM	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

## Description

Rotates the bits in the individual data elements (doublewords, or quadword) in the first source operand to the right by the number of bits specified in the count operand. If the value specified by the count operand is greater than 31 (for doublewords), or 63 (for a quadword), then the count operand modulo the data size (32 or 64) is used.

EVEX.128 encoded version: The destination operand is a XMM register. The source operand is a XMM register or a memory location (for immediate form). The count operand can come either from an XMM register or a memory location or an 8-bit immediate. Bits (MAX\_VL-1:128) of the corresponding ZMM register are zeroed.

EVEX.256 encoded version: The destination operand is a YMM register. The source operand is a YMM register or a memory location (for immediate form). The count operand can come either from an XMM register or a memory location or an 8-bit immediate. Bits (MAX\_VL-1:256) of the corresponding ZMM register are zeroed.

EVEX.512 encoded version: The destination operand is a ZMM register updated according to the writemask. For the count operand in immediate form, the source operand can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32/64-bit memory location, the count operand is an 8-bit immediate. For the count operand in variable form, the first source operand (the second operand) is a ZMM register and the counter operand (the third operand) is a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32/64-bit memory location.

## Operation

```
RIGHT_ROTATE_DWORDS(SRC, COUNT_SRC)
COUNT ← COUNT_SRC modulo 32;
DEST[31:0] ← (SRC >> COUNT) | (SRC << (32 - COUNT));
```

```
RIGHT_ROTATE_QWORDS(SRC, COUNT_SRC)
COUNT ← COUNT_SRC modulo 64;
DEST[63:0] ← (SRC >> COUNT) | (SRC << (64 - COUNT));
```

**VPRORD (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

  i ← j \* 32

  IF k1[j] OR \*no writemask\* THEN

    IF (EVEX.b = 1) AND (SRC1 \*is memory\*)

      THEN DEST[i+31:i] ← RIGHT\_ROTATE\_DWORDS( SRC1[31:0], imm8)

      ELSE DEST[i+31:i] ← RIGHT\_ROTATE\_DWORDS(SRC1[i+31:i], imm8)

    FI;

  ELSE

    IF \*merging-masking\* ; merging-masking

      THEN \*DEST[i+31:i] remains unchanged\*

    ELSE \*zeroing-masking\* ; zeroing-masking

      DEST[i+31:i] ← 0

    FI

  FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VPRORVD (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\* THEN

IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

THEN DEST[i+31:i] ← RIGHT\_ROTATE\_DWORDS(SRC1[i+31:i], SRC2[31:0])

ELSE DEST[i+31:i] ← RIGHT\_ROTATE\_DWORDS(SRC1[i+31:i], SRC2[i+31:i])

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+31:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VPRORQ (EVEX encoded versions)**

(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) AND (SRC1 \*is memory\*)

THEN DEST[i+63:i] ← RIGHT\_ROTATE\_QWORDS(SRC1[63:0], imm8)

ELSE DEST[i+63:i] ← RIGHT\_ROTATE\_QWORDS(SRC1[i+63:i], imm8)

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VPRORVQ (EVEX encoded versions)**

(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) AND (SRC2 \*is memory\*)

THEN DEST[i+63:i] ← RIGHT\_ROTATE\_QWORDS(SRC1[i+63:i], SRC2[63:0])

ELSE DEST[i+63:i] ← RIGHT\_ROTATE\_QWORDS(SRC1[i+63:i], SRC2[i+63:i])

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR  
 DEST[MAX\_VL-1:VL] ← 0

### Intel C/C++ Compiler Intrinsic Equivalent

VPRORD \_\_m512i \_mm512\_ror\_epi32(\_\_m512i a, int imm);  
 VPRORD \_\_m512i \_mm512\_mask\_ror\_epi32(\_\_m512i a, \_\_mmask16 k, \_\_m512i b, int imm);  
 VPRORD \_\_m512i \_mm512\_maskz\_ror\_epi32(\_\_mmask16 k, \_\_m512i a, int imm);  
 VPRORD \_\_m256i \_mm256\_ror\_epi32(\_\_m256i a, int imm);  
 VPRORD \_\_m256i \_mm256\_mask\_ror\_epi32(\_\_m256i a, \_\_mmask8 k, \_\_m256i b, int imm);  
 VPRORD \_\_m256i \_mm256\_maskz\_ror\_epi32(\_\_mmask8 k, \_\_m256i a, int imm);  
 VPRORD \_\_m128i \_mm\_ror\_epi32(\_\_m128i a, int imm);  
 VPRORD \_\_m128i \_mm\_mask\_ror\_epi32(\_\_m128i a, \_\_mmask8 k, \_\_m128i b, int imm);  
 VPRORD \_\_m128i \_mm\_maskz\_ror\_epi32(\_\_mmask8 k, \_\_m128i a, int imm);  
 VPRORQ \_\_m512i \_mm512\_ror\_epi64(\_\_m512i a, int imm);  
 VPRORQ \_\_m512i \_mm512\_mask\_ror\_epi64(\_\_m512i a, \_\_mmask8 k, \_\_m512i b, int imm);  
 VPRORQ \_\_m512i \_mm512\_maskz\_ror\_epi64(\_\_mmask8 k, \_\_m512i a, int imm);  
 VPRORQ \_\_m256i \_mm256\_ror\_epi64(\_\_m256i a, int imm);  
 VPRORQ \_\_m256i \_mm256\_mask\_ror\_epi64(\_\_m256i a, \_\_mmask8 k, \_\_m256i b, int imm);  
 VPRORQ \_\_m256i \_mm256\_maskz\_ror\_epi64(\_\_mmask8 k, \_\_m256i a, int imm);  
 VPRORQ \_\_m128i \_mm\_ror\_epi64(\_\_m128i a, int imm);  
 VPRORQ \_\_m128i \_mm\_mask\_ror\_epi64(\_\_m128i a, \_\_mmask8 k, \_\_m128i b, int imm);  
 VPRORQ \_\_m128i \_mm\_maskz\_ror\_epi64(\_\_mmask8 k, \_\_m128i a, int imm);  
 VPRORVD \_\_m512i \_mm512\_rorv\_epi32(\_\_m512i a, \_\_m512i cnt);  
 VPRORVD \_\_m512i \_mm512\_mask\_rorv\_epi32(\_\_m512i a, \_\_mmask16 k, \_\_m512i b, \_\_m512i cnt);  
 VPRORVD \_\_m512i \_mm512\_maskz\_rorv\_epi32(\_\_mmask16 k, \_\_m512i a, \_\_m512i cnt);  
 VPRORVD \_\_m256i \_mm256\_rorv\_epi32(\_\_m256i a, \_\_m256i cnt);  
 VPRORVD \_\_m256i \_mm256\_mask\_rorv\_epi32(\_\_m256i a, \_\_mmask8 k, \_\_m256i b, \_\_m256i cnt);  
 VPRORVD \_\_m256i \_mm256\_maskz\_rorv\_epi32(\_\_mmask8 k, \_\_m256i a, \_\_m256i cnt);  
 VPRORVD \_\_m128i \_mm\_rorv\_epi32(\_\_m128i a, \_\_m128i cnt);  
 VPRORVD \_\_m128i \_mm\_mask\_rorv\_epi32(\_\_m128i a, \_\_mmask8 k, \_\_m128i b, \_\_m128i cnt);  
 VPRORVD \_\_m128i \_mm\_maskz\_rorv\_epi32(\_\_mmask8 k, \_\_m128i a, \_\_m128i cnt);  
 VPRORVQ \_\_m512i \_mm512\_rorv\_epi64(\_\_m512i a, \_\_m512i cnt);  
 VPRORVQ \_\_m512i \_mm512\_mask\_rorv\_epi64(\_\_m512i a, \_\_mmask8 k, \_\_m512i b, \_\_m512i cnt);  
 VPRORVQ \_\_m512i \_mm512\_maskz\_rorv\_epi64(\_\_mmask8 k, \_\_m512i a, \_\_m512i cnt);  
 VPRORVQ \_\_m256i \_mm256\_rorv\_epi64(\_\_m256i a, \_\_m256i cnt);  
 VPRORVQ \_\_m256i \_mm256\_mask\_rorv\_epi64(\_\_m256i a, \_\_mmask8 k, \_\_m256i b, \_\_m256i cnt);  
 VPRORVQ \_\_m256i \_mm256\_maskz\_rorv\_epi64(\_\_mmask8 k, \_\_m256i a, \_\_m256i cnt);  
 VPRORVQ \_\_m128i \_mm\_rorv\_epi64(\_\_m128i a, \_\_m128i cnt);  
 VPRORVQ \_\_m128i \_mm\_mask\_rorv\_epi64(\_\_m128i a, \_\_mmask8 k, \_\_m128i b, \_\_m128i cnt);  
 VPRORVQ \_\_m128i \_mm\_maskz\_rorv\_epi64(\_\_mmask8 k, \_\_m128i a, \_\_m128i cnt);

### SIMD Floating-Point Exceptions

None

### Other Exceptions

EVEX-encoded instruction, see Exceptions Type E4.

## VPSCATTERDD/VPSCATTERDQ/VPSCATTERQD/VPSCATTERQQ—Scatter Packed Dword, Packed Qword with Signed Dword, Signed Qword Indices

Opcode/ Instruction	Op/ En	64/32 bitMode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 A0 /vsib VPSCATTERDD vm32x {k1}, xmm1	T1S	V/V	AVX512VL AVX512F	Using signed dword indices, scatter dword values to memory using writemask k1.
EVEX.256.66.0F38.W0 A0 /vsib VPSCATTERDD vm32y {k1}, ymm1	T1S	V/V	AVX512VL AVX512F	Using signed dword indices, scatter dword values to memory using writemask k1.
EVEX.512.66.0F38.W0 A0 /vsib VPSCATTERDD vm32z {k1}, zmm1	T1S	V/V	AVX512F	Using signed dword indices, scatter dword values to memory using writemask k1.
EVEX.128.66.0F38.W1 A0 /vsib VPSCATTERDQ vm32x {k1}, xmm1	T1S	V/V	AVX512VL AVX512F	Using signed dword indices, scatter qword values to memory using writemask k1.
EVEX.256.66.0F38.W1 A0 /vsib VPSCATTERDQ vm32x {k1}, ymm1	T1S	V/V	AVX512VL AVX512F	Using signed dword indices, scatter qword values to memory using writemask k1.
EVEX.512.66.0F38.W1 A0 /vsib VPSCATTERDQ vm32y {k1}, zmm1	T1S	V/V	AVX512F	Using signed dword indices, scatter qword values to memory using writemask k1.
EVEX.128.66.0F38.W0 A1 /vsib VPSCATTERQD vm64x {k1}, xmm1	T1S	V/V	AVX512VL AVX512F	Using signed qword indices, scatter dword values to memory using writemask k1.
EVEX.256.66.0F38.W0 A1 /vsib VPSCATTERQD vm64y {k1}, xmm1	T1S	V/V	AVX512VL AVX512F	Using signed qword indices, scatter dword values to memory using writemask k1.
EVEX.512.66.0F38.W0 A1 /vsib VPSCATTERQD vm64z {k1}, ymm1	T1S	V/V	AVX512F	Using signed qword indices, scatter dword values to memory using writemask k1.
EVEX.128.66.0F38.W1 A1 /vsib VPSCATTERQQ vm64x {k1}, xmm1	T1S	V/V	AVX512VL AVX512F	Using signed qword indices, scatter qword values to memory using writemask k1.
EVEX.256.66.0F38.W1 A1 /vsib VPSCATTERQQ vm64y {k1}, ymm1	T1S	V/V	AVX512VL AVX512F	Using signed qword indices, scatter qword values to memory using writemask k1.
EVEX.512.66.0F38.W1 A1 /vsib VPSCATTERQQ vm64z {k1}, zmm1	T1S	V/V	AVX512F	Using signed qword indices, scatter qword values to memory using writemask k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
T1S	BaseReg (R): VSIB:base, VectorReg(R): VSIB:index	ModRM:reg (r)	NA	NA

### Description

Stores up to 16 elements (8 elements for qword indices) in doubleword vector or 8 elements in quadword vector to the memory locations pointed by base address `BASE_ADDR` and index vector `VINDEX`, with scale `SCALE`. The elements are specified via the `VSIB` (i.e., the index register is a vector register, holding packed indices). Elements

will only be stored if their corresponding mask bit is one. The entire mask register will be set to zero by this instruction unless it triggers an exception.

This instruction can be suspended by an exception if at least one element is already scattered (i.e., if the exception is triggered by an element other than the rightmost one with its mask bit set). When this happens, the destination register and the mask register are partially updated. If any traps or interrupts are pending from already scattered elements, they will be delivered in lieu of the exception; in this case, EFLAG.RF is set to one so an instruction breakpoint is not re-triggered when the instruction is continued.

Note that:

- Only writes to overlapping vector indices are guaranteed to be ordered with respect to each other (from LSB to MSB of the source registers). Note that this also include partially overlapping vector indices. Writes that are not overlapped may happen in any order. Memory ordering with other instructions follows the Intel-64 memory ordering model. Note that this does not account for non-overlapping indices that map into the same physical address locations.
- If two or more destination indices completely overlap, the “earlier” write(s) may be skipped.
- Faults are delivered in a right-to-left manner. That is, if a fault is triggered by an element and delivered, all elements closer to the LSB of the destination ZMM will be completed (and non-faulting). Individual elements closer to the MSB may or may not be completed. If a given element triggers multiple faults, they are delivered in the conventional order.
- Elements may be scattered in any order, but faults must be delivered in a right-to left order; thus, elements to the left of a faulting one may be gathered before the fault is delivered. A given implementation of this instruction is repeatable - given the same input values and architectural state, the same set of elements to the left of the faulting one will be gathered.
- This instruction does not perform AC checks, and so will never deliver an AC fault.
- Not valid with 16-bit effective addresses. Will deliver a #UD fault.
- If this instruction overwrites itself and then takes a fault, only a subset of elements may be completed before the fault is delivered (as described above). If the fault handler completes and attempts to re-execute this instruction, the new instruction will be executed, and the scatter will not complete.

Note that the presence of VSIB byte is enforced in this instruction. Hence, the instruction will #UD fault if ModRM.rm is different than 100b.

This instruction has special  $\text{disp8} * N$  and alignment rules. N is considered to be the size of a single vector element.

The scaled index may require more bits to represent than the address bits used by the processor (e.g., in 32-bit mode, if the scale is greater than one). In this case, the most significant bits beyond the number of address bits are ignored.

The instruction will #UD fault if the k0 mask register is specified.

The instruction will #UD fault if EVEX.Z = 1.

### Operation

BASE\_ADDR stands for the memory operand base address (a GPR); may not exist

VINDEX stands for the memory operand vector of indices (a ZMM register)

SCALE stands for the memory operand scalar (1, 2, 4 or 8)

DISP is the optional 1, 2 or 4 byte displacement

**VPSCATTERDD (EVEX encoded versions)**

(KL, VL)= (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\*

THEN MEM[BASE\_ADDR + SignExtend(VINDEX[j+31:i]) \* SCALE + DISP] ← SRC[i+31:i]

k1[j] ← 0

FI;

ENDFOR

k1[MAX\_KL-1:KL] ← 0

**VPSCATTERDQ (EVEX encoded versions)**

(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 MEM[BASE\_ADDR + SignExtend(VINDEX[k+31:k]) \* SCALE + DISP] ← SRC[j+63:i]

k1[j] ← 0

FI;

ENDFOR

k1[MAX\_KL-1:KL] ← 0

**VPSCATTERQD (EVEX encoded versions)**

(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 MEM[BASE\_ADDR + (VINDEX[k+63:k]) \* SCALE + DISP] ← SRC[i+31:i]

k1[j] ← 0

FI;

ENDFOR

k1[MAX\_KL-1:KL] ← 0

**VPSCATTERQQ (EVEX encoded versions)**

(KL, VL)= (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN MEM[BASE\_ADDR + (VINDEX[j+63:j]) \* SCALE + DISP] ← SRC[i+63:i]

FI;

ENDFOR

k1[MAX\_KL-1:KL] ← 0



### Intel C/C++ Compiler Intrinsic Equivalent

```

VPSCATTERDD void __mm512_i32scatter_epi32(void * base, __m512i vdx, __m512i a, int scale);
VPSCATTERDD void __mm256_i32scatter_epi32(void * base, __m256i vdx, __m256i a, int scale);
VPSCATTERDD void __mm_i32scatter_epi32(void * base, __m128i vdx, __m128i a, int scale);
VPSCATTERDD void __mm512_mask_i32scatter_epi32(void * base, __mmask16 k, __m512i vdx, __m512i a, int scale);
VPSCATTERDD void __mm256_mask_i32scatter_epi32(void * base, __mmask8 k, __m256i vdx, __m256i a, int scale);
VPSCATTERDD void __mm_mask_i32scatter_epi32(void * base, __mmask8 k, __m128i vdx, __m128i a, int scale);
VPSCATTERDQ void __mm512_i32scatter_epi64(void * base, __m256i vdx, __m512i a, int scale);
VPSCATTERDQ void __mm256_i32scatter_epi64(void * base, __m128i vdx, __m256i a, int scale);
VPSCATTERDQ void __mm_i32scatter_epi64(void * base, __m128i vdx, __m128i a, int scale);
VPSCATTERDQ void __mm512_mask_i32scatter_epi64(void * base, __mmask8 k, __m256i vdx, __m512i a, int scale);
VPSCATTERDQ void __mm256_mask_i32scatter_epi64(void * base, __mmask8 k, __m128i vdx, __m256i a, int scale);
VPSCATTERDQ void __mm_mask_i32scatter_epi64(void * base, __mmask8 k, __m128i vdx, __m128i a, int scale);
VPSCATTERQD void __mm512_i64scatter_epi32(void * base, __m512i vdx, __m256i a, int scale);
VPSCATTERQD void __mm256_i64scatter_epi32(void * base, __m256i vdx, __m128i a, int scale);
VPSCATTERQD void __mm_i64scatter_epi32(void * base, __m128i vdx, __m128i a, int scale);
VPSCATTERQD void __mm512_mask_i64scatter_epi32(void * base, __mmask8 k, __m512i vdx, __m256i a, int scale);
VPSCATTERQD void __mm256_mask_i64scatter_epi32(void * base, __mmask8 k, __m256i vdx, __m128i a, int scale);
VPSCATTERQD void __mm_mask_i64scatter_epi32(void * base, __mmask8 k, __m128i vdx, __m128i a, int scale);
VPSCATTERQQ void __mm512_i64scatter_epi64(void * base, __m512i vdx, __m512i a, int scale);
VPSCATTERQQ void __mm256_i64scatter_epi64(void * base, __m256i vdx, __m256i a, int scale);
VPSCATTERQQ void __mm_i64scatter_epi64(void * base, __m128i vdx, __m128i a, int scale);
VPSCATTERQQ void __mm512_mask_i64scatter_epi64(void * base, __mmask8 k, __m512i vdx, __m512i a, int scale);
VPSCATTERQQ void __mm256_mask_i64scatter_epi64(void * base, __mmask8 k, __m256i vdx, __m256i a, int scale);
VPSCATTERQQ void __mm_mask_i64scatter_epi64(void * base, __mmask8 k, __m128i vdx, __m128i a, int scale);

```

### SIMD Floating-Point Exceptions

None

### Other Exceptions

See Exceptions Type E12.

## VPSLLVW/VPSLLVD/VPSLLVQ—Variable Bit Shift Left Logical

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.NDS.128.66.0F38.W0 47 /r VPSLLVD xmm1, xmm2, xmm3/m128	RVM	V/V	AVX2	Shift doublewords in xmm2 left by amount specified in the corresponding element of xmm3/m128 while shifting in 0s.
VEX.NDS.128.66.0F38.W1 47 /r VPSLLVQ xmm1, xmm2, xmm3/m128	RVM	V/V	AVX2	Shift quadwords in xmm2 left by amount specified in the corresponding element of xmm3/m128 while shifting in 0s.
VEX.NDS.256.66.0F38.W0 47 /r VPSLLVD ymm1, ymm2, ymm3/m256	RVM	V/V	AVX2	Shift doublewords in ymm2 left by amount specified in the corresponding element of ymm3/m256 while shifting in 0s.
VEX.NDS.256.66.0F38.W1 47 /r VPSLLVQ ymm1, ymm2, ymm3/m256	RVM	V/V	AVX2	Shift quadwords in ymm2 left by amount specified in the corresponding element of ymm3/m256 while shifting in 0s.
EVEX.NDS.128.66.0F38.W1 12 /r VPSLLVW xmm1 {k1}{z}, xmm2, xmm3/m128	FVM	V/V	AVX512VL AVX512BW	Shift words in xmm2 left by amount specified in the corresponding element of xmm3/m128 while shifting in 0s using writemask k1.
EVEX.NDS.256.66.0F38.W1 12 /r VPSLLVW ymm1 {k1}{z}, ymm2, ymm3/m256	FVM	V/V	AVX512VL AVX512BW	Shift words in ymm2 left by amount specified in the corresponding element of ymm3/m256 while shifting in 0s using writemask k1.
EVEX.NDS.512.66.0F38.W1 12 /r VPSLLVW zmm1 {k1}{z}, zmm2, zmm3/m512	FVM	V/V	AVX512BW	Shift words in zmm2 left by amount specified in the corresponding element of zmm3/m512 while shifting in 0s using writemask k1.
EVEX.NDS.128.66.0F38.W0 47 /r VPSLLVD xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	FV	V/V	AVX512VL AVX512F	Shift doublewords in xmm2 left by amount specified in the corresponding element of xmm3/m128/m32bcst while shifting in 0s using writemask k1.
EVEX.NDS.256.66.0F38.W0 47 /r VPSLLVD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	FV	V/V	AVX512VL AVX512F	Shift doublewords in ymm2 left by amount specified in the corresponding element of ymm3/m256/m32bcst while shifting in 0s using writemask k1.
EVEX.NDS.512.66.0F38.W0 47 /r VPSLLVD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	FV	V/V	AVX512F	Shift doublewords in zmm2 left by amount specified in the corresponding element of zmm3/m512/m32bcst while shifting in 0s using writemask k1.
EVEX.NDS.128.66.0F38.W1 47 /r VPSLLVQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	FV	V/V	AVX512VL AVX512F	Shift quadwords in xmm2 left by amount specified in the corresponding element of xmm3/m128/m64bcst while shifting in 0s using writemask k1.
EVEX.NDS.256.66.0F38.W1 47 /r VPSLLVQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	FV	V/V	AVX512VL AVX512F	Shift quadwords in ymm2 left by amount specified in the corresponding element of ymm3/m256/m64bcst while shifting in 0s using writemask k1.
EVEX.NDS.512.66.0F38.W1 47 /r VPSLLVQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	FV	V/V	AVX512F	Shift quadwords in zmm2 left by amount specified in the corresponding element of zmm3/m512/m64bcst while shifting in 0s using writemask k1.

## Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FVM	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA
FV	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

## Description

Shifts the bits in the individual data elements (words, doublewords or quadword) in the first source operand to the left by the count value of respective data elements in the second source operand. As the bits in the data elements are shifted left, the empty low-order bits are cleared (set to 0).

The count values are specified individually in each data element of the second source operand. If the unsigned integer value specified in the respective data element of the second source operand is greater than 15 (for word), 31 (for doublewords), or 63 (for a quadword), then the destination data element are written with 0.

VEX.128 encoded version: The destination and first source operands are XMM registers. The count operand can be either an XMM register or a 128-bit memory location. Bits (MAX\_VL-1:128) of the corresponding destination register are zeroed.

VEX.256 encoded version: The destination and first source operands are YMM registers. The count operand can be either an YMM register or a 256-bit memory. Bits (MAX\_VL-1:256) of the corresponding ZMM register are zeroed.

EVEX encoded VPSLLVD/Q: The destination and first source operands are ZMM/YMM/XMM registers. The count operand can be either a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512-bit vector broadcasted from a 32/64-bit memory location. The destination is conditionally updated with writemask k1.

EVEX encoded VPSLLVW: The destination and first source operands are ZMM/YMM/XMM registers. The count operand can be either a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination is conditionally updated with writemask k1.

## Operation

**VPSLLVW (EVEX encoded version)**

(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] ← ZeroExtend(SRC1[i+15:i] << SRC2[j+15:i])
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+15:i] remains unchanged*
        ELSE ; zeroing-masking
          DEST[i+15:i] ← 0
      FI
  FI;
ENDFOR;
DEST[MAX_VL-1:VL] ← 0;

```

**VPSLLVD (VEX.128 version)**

```

COUNT_0 ← SRC2[31 : 0]
(* Repeat Each COUNT_i for the 2nd through 4th dwords of SRC2*)
COUNT_3 ← SRC2[100 : 96];
IF COUNT_0 < 32 THEN
DEST[31:0] ← ZeroExtend(SRC1[31:0] << COUNT_0);
ELSE
DEST[31:0] ← 0;
(* Repeat shift operation for 2nd through 4th dwords *)
IF COUNT_3 < 32 THEN
DEST[127:96] ← ZeroExtend(SRC1[127:96] << COUNT_3);
ELSE
DEST[127:96] ← 0;
DEST[MAX_VL-1:128] ← 0;

```

**VPSLLVD (VEX.256 version)**

```

COUNT_0 ← SRC2[31 : 0];
(* Repeat Each COUNT_i for the 2nd through 7th dwords of SRC2*)
COUNT_7 ← SRC2[228 : 224];
IF COUNT_0 < 32 THEN
DEST[31:0] ← ZeroExtend(SRC1[31:0] << COUNT_0);
ELSE
DEST[31:0] ← 0;
(* Repeat shift operation for 2nd through 7th dwords *)
IF COUNT_7 < 32 THEN
DEST[255:224] ← ZeroExtend(SRC1[255:224] << COUNT_7);
ELSE
DEST[255:224] ← 0;
DEST[MAX_VL-1:256] ← 0;

```

**VPSLLVD (EVEX encoded version)**

```

(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask* THEN
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
      THEN DEST[i+31:i] ← ZeroExtend(SRC1[i+31:i] << SRC2[31:0])
      ELSE DEST[i+31:i] ← ZeroExtend(SRC1[i+31:i] << SRC2[i+31:i])
    FI;
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+31:i] remains unchanged*
    ELSE ; zeroing-masking
      DEST[i+31:i] ← 0
    FI
  FI;
ENDFOR;
DEST[MAX_VL-1:VL] ← 0;

```

**VPSLLVQ (VEX.128 version)**

```

COUNT_0 ← SRC2[63 : 0];
COUNT_1 ← SRC2[127 : 64];
IF COUNT_0 < 64 THEN
  DEST[63:0] ← ZeroExtend(SRC1[63:0] << COUNT_0);
ELSE
  DEST[63:0] ← 0;
  IF COUNT_1 < 64 THEN
    DEST[127:64] ← ZeroExtend(SRC1[127:64] << COUNT_1);
  ELSE
    DEST[127:96] ← 0;
  DEST[MAX_VL-1:128] ← 0;

```

**VPSLLVQ (VEX.256 version)**

```

COUNT_0 ← SRC2[63 : 0];
(* Repeat Each COUNT_i for the 2nd through 4th dwords of SRC2*)
COUNT_3 ← SRC2[197 : 192];
IF COUNT_0 < 64 THEN
  DEST[63:0] ← ZeroExtend(SRC1[63:0] << COUNT_0);
ELSE
  DEST[63:0] ← 0;
  (* Repeat shift operation for 2nd through 4th dwords *)
  IF COUNT_3 < 64 THEN
    DEST[255:192] ← ZeroExtend(SRC1[255:192] << COUNT_3);
  ELSE
    DEST[255:192] ← 0;
  DEST[MAX_VL-1:256] ← 0;

```

**VPSLLVQ (EVEX encoded version)**

```

(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) AND (SRC2 *is memory*)
      THEN DEST[i+63:i] ← ZeroExtend(SRC1[i+63:i] << SRC2[63:0])
      ELSE DEST[i+63:i] ← ZeroExtend(SRC1[i+63:i] << SRC2[i+63:i])
    FI;
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+63:i] remains unchanged*
    ELSE ; zeroing-masking
      DEST[i+63:i] ← 0
    FI
  FI;
ENDFOR;
DEST[MAX_VL-1:VL] ← 0;

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VPSLLVW __m512i __mm512_sllv_epi16(__m512i a, __m512i cnt);
VPSLLVW __m512i __mm512_mask_sllv_epi16(__m512i s, __mmask32 k, __m512i a, __m512i cnt);
VPSLLVW __m512i __mm512_maskz_sllv_epi16(__mmask32 k, __m512i a, __m512i cnt);
VPSLLVW __m256i __mm256_mask_sllv_epi16(__m256i s, __mmask16 k, __m256i a, __m256i cnt);
VPSLLVW __m256i __mm256_maskz_sllv_epi16(__mmask16 k, __m256i a, __m256i cnt);
VPSLLVW __m128i __mm_mask_sllv_epi16(__m128i s, __mmask8 k, __m128i a, __m128i cnt);

```

VPSLLVW \_\_m128i \_\_mm\_maskz\_sllv\_epi16( \_\_mmask8 k, \_\_m128i a, \_\_m128i cnt);  
 VPSLLVD \_\_m512i \_\_mm512\_sllv\_epi32( \_\_m512i a, \_\_m512i cnt);  
 VPSLLVD \_\_m512i \_\_mm512\_mask\_sllv\_epi32( \_\_m512i s, \_\_mmask16 k, \_\_m512i a, \_\_m512i cnt);  
 VPSLLVD \_\_m512i \_\_mm512\_maskz\_sllv\_epi32( \_\_mmask16 k, \_\_m512i a, \_\_m512i cnt);  
 VPSLLVD \_\_m256i \_\_mm256\_mask\_sllv\_epi32( \_\_m256i s, \_\_mmask8 k, \_\_m256i a, \_\_m256i cnt);  
 VPSLLVD \_\_m256i \_\_mm256\_maskz\_sllv\_epi32( \_\_mmask8 k, \_\_m256i a, \_\_m256i cnt);  
 VPSLLVD \_\_m128i \_\_mm\_mask\_sllv\_epi32( \_\_m128i s, \_\_mmask8 k, \_\_m128i a, \_\_m128i cnt);  
 VPSLLVD \_\_m128i \_\_mm\_maskz\_sllv\_epi32( \_\_mmask8 k, \_\_m128i a, \_\_m128i cnt);  
 VPSLLVQ \_\_m512i \_\_mm512\_sllv\_epi64( \_\_m512i a, \_\_m512i cnt);  
 VPSLLVQ \_\_m512i \_\_mm512\_mask\_sllv\_epi64( \_\_m512i s, \_\_mmask8 k, \_\_m512i a, \_\_m512i cnt);  
 VPSLLVQ \_\_m512i \_\_mm512\_maskz\_sllv\_epi64( \_\_mmask8 k, \_\_m512i a, \_\_m512i cnt);  
 VPSLLVD \_\_m256i \_\_mm256\_mask\_sllv\_epi64( \_\_m256i s, \_\_mmask8 k, \_\_m256i a, \_\_m256i cnt);  
 VPSLLVD \_\_m256i \_\_mm256\_maskz\_sllv\_epi64( \_\_mmask8 k, \_\_m256i a, \_\_m256i cnt);  
 VPSLLVD \_\_m128i \_\_mm\_mask\_sllv\_epi64( \_\_m128i s, \_\_mmask8 k, \_\_m128i a, \_\_m128i cnt);  
 VPSLLVD \_\_m128i \_\_mm\_maskz\_sllv\_epi64( \_\_mmask8 k, \_\_m128i a, \_\_m128i cnt);  
 VPSLLVD \_\_m256i \_\_mm256\_sllv\_epi32( \_\_m256i m, \_\_m256i count)  
 VPSLLVQ \_\_m256i \_\_mm256\_sllv\_epi64( \_\_m256i m, \_\_m256i count)

### SIMD Floating-Point Exceptions

None

### Other Exceptions

VEX-encoded instructions, see Exceptions Type 4.

EVEX-encoded VPSLLVD/VPSLLVQ, see Exceptions Type E4.

EVEX-encoded VPSLLVW, see Exceptions Type E4.nb.

## VPSRAVW/VPSRAVD/VPSRAVQ—Variable Bit Shift Right Arithmetic

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.NDS.128.66.0F38.W0 46 /r VPSRAVD xmm1, xmm2, xmm3/m128	RVM	V/V	AVX2	Shift doublewords in xmm2 right by amount specified in the corresponding element of xmm3/m128 while shifting in sign bits.
VEX.NDS.256.66.0F38.W0 46 /r VPSRAVD ymm1, ymm2, ymm3/m256	RVM	V/V	AVX2	Shift doublewords in ymm2 right by amount specified in the corresponding element of ymm3/m256 while shifting in sign bits.
EVEX.NDS.128.66.0F38.W1 11 /r VPSRAVW xmm1 {k1}{z}, xmm2, xmm3/m128	FVM	V/V	AVX512VL AVX512BW	Shift words in xmm2 right by amount specified in the corresponding element of xmm3/m128 while shifting in sign bits using writemask k1.
EVEX.NDS.256.66.0F38.W1 11 /r VPSRAVW ymm1 {k1}{z}, ymm2, ymm3/m256	FVM	V/V	AVX512VL AVX512BW	Shift words in ymm2 right by amount specified in the corresponding element of ymm3/m256 while shifting in sign bits using writemask k1.
EVEX.NDS.512.66.0F38.W1 11 /r VPSRAVW zmm1 {k1}{z}, zmm2, zmm3/m512	FVM	V/V	AVX512BW	Shift words in zmm2 right by amount specified in the corresponding element of zmm3/m512 while shifting in sign bits using writemask k1.
EVEX.NDS.128.66.0F38.W0 46 /r VPSRAVD xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	FV	V/V	AVX512VL AVX512F	Shift doublewords in xmm2 right by amount specified in the corresponding element of xmm3/m128/m32bcst while shifting in sign bits using writemask k1.
EVEX.NDS.256.66.0F38.W0 46 /r VPSRAVD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	FV	V/V	AVX512VL AVX512F	Shift doublewords in ymm2 right by amount specified in the corresponding element of ymm3/m256/m32bcst while shifting in sign bits using writemask k1.
EVEX.NDS.512.66.0F38.W0 46 /r VPSRAVD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	FV	V/V	AVX512F	Shift doublewords in zmm2 right by amount specified in the corresponding element of zmm3/m512/m32bcst while shifting in sign bits using writemask k1.
EVEX.NDS.128.66.0F38.W1 46 /r VPSRAVQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	FV	V/V	AVX512VL AVX512F	Shift quadwords in xmm2 right by amount specified in the corresponding element of xmm3/m128/m64bcst while shifting in sign bits using writemask k1.
EVEX.NDS.256.66.0F38.W1 46 /r VPSRAVQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	FV	V/V	AVX512VL AVX512F	Shift quadwords in ymm2 right by amount specified in the corresponding element of ymm3/m256/m64bcst while shifting in sign bits using writemask k1.
EVEX.NDS.512.66.0F38.W1 46 /r VPSRAVQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	FV	V/V	AVX512F	Shift quadwords in zmm2 right by amount specified in the corresponding element of zmm3/m512/m64bcst while shifting in sign bits using writemask k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FVM	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA
FV	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

**Description**

Shifts the bits in the individual data elements (word/doublewords/quadword) in the first source operand (the second operand) to the right by the number of bits specified in the count value of respective data elements in the second source operand (the third operand). As the bits in the data elements are shifted right, the empty high-order bits are set to the MSB (sign extension).

The count values are specified individually in each data element of the second source operand. If the unsigned integer value specified in the respective data element of the second source operand is greater than 15 (for words), 31 (for doublewords), or 63 (for a quadword), then the destination data element are filled with the corresponding sign bit of the source element.

The count values are specified individually in each data element of the second source operand. If the unsigned integer value specified in the respective data element of the second source operand is greater than 16 (for word), 31 (for doublewords), or 63 (for a quadword), then the destination data element are written with 0.

VEX.128 encoded version: The destination and first source operands are XMM registers. The count operand can be either an XMM register or a 128-bit memory location. Bits (MAX\_VL-1:128) of the corresponding destination register are zeroed.

VEX.256 encoded version: The destination and first source operands are YMM registers. The count operand can be either an YMM register or a 256-bit memory. Bits (MAX\_VL-1:256) of the corresponding destination register are zeroed.

EVEX.512/256/128 encoded VPSRAVD/W: The destination and first source operands are ZMM/YMM/XMM registers. The count operand can be either a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination is conditionally updated with writemask k1.

EVEX.512/256/128 encoded VPSRAVQ: The destination and first source operands are ZMM/YMM/XMM registers. The count operand can be either a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination is conditionally updated with writemask k1.

**Operation****VPSRAVW (EVEX encoded version)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 16
  IF k1[j] OR *no writemask*
    THEN
      COUNT ← SRC2[i+3:i]
      IF COUNT < 16
        THEN DEST[i+15:i] ← SignExtend(SRC1[i+15:i] >> COUNT)
        ELSE
          FOR k ← 0 TO 15
            DEST[i+k] ← SRC1[i+15]
          ENDFOR;
        FI
      ELSE
        IF *merging-masking* ; merging-masking
          THEN *DEST[i+15:i] remains unchanged*
          ELSE ; zeroing-masking
            DEST[i+15:i] ← 0
          FI
        FI;
      ENDFOR;
    DEST[MAX_VL-1:VL] ← 0;
  
```



**VPSRAVD (VEX.128 version)**

```

COUNT_0 ← SRC2[31 : 0]
(* Repeat Each COUNT_i for the 2nd through 4th dwords of SRC2*)
COUNT_3 ← SRC2[100 : 96];
DEST[31:0] ← SignExtend(SRC1[31:0] >> COUNT_0);
(* Repeat shift operation for 2nd through 4th dwords *)
DEST[127:96] ← SignExtend(SRC1[127:96] >> COUNT_3);
DEST[MAX_VL-1:128] ← 0;

```

**VPSRAVD (VEX.256 version)**

```

COUNT_0 ← SRC2[31 : 0];
(* Repeat Each COUNT_i for the 2nd through 8th dwords of SRC2*)
COUNT_7 ← SRC2[228 : 224];
DEST[31:0] ← SignExtend(SRC1[31:0] >> COUNT_0);
(* Repeat shift operation for 2nd through 7th dwords *)
DEST[255:224] ← SignExtend(SRC1[255:224] >> COUNT_7);
DEST[MAX_VL-1:256] ← 0;

```

**VPSRAVD (EVEX encoded version)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask* THEN
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
      THEN
        COUNT ← SRC2[4:0]
        IF COUNT < 32
          THEN DEST[j+31:i] ← SignExtend(SRC1[j+31:i] >> COUNT)
          ELSE
            FOR k ← 0 TO 31
              DEST[j+k] ← SRC1[j+31]
            ENDFOR;
          FI
        ELSE
          COUNT ← SRC2[i+4:i]
          IF COUNT < 32
            THEN DEST[j+31:i] ← SignExtend(SRC1[j+31:i] >> COUNT)
            ELSE
              FOR k ← 0 TO 31
                DEST[j+k] ← SRC1[j+31]
              ENDFOR;
            FI
          FI;
        ELSE
          IF *merging-masking* ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE ; zeroing-masking
              DEST[31:0] ← 0
            FI
          FI;
        ENDFOR;
      DEST[MAX_VL-1:VL] ← 0;

```

**VPSRAVQ (EVEX encoded version)**

(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) AND (SRC2 \*is memory\*)

THEN

COUNT ← SRC2[5:0]

IF COUNT &lt; 64

THEN DEST[i+63:i] ← SignExtend(SRC1[i+63:i] &gt;&gt; COUNT)

ELSE

FOR k ← 0 TO 63

DEST[i+k] ← SRC1[i+63]

ENDFOR;

FI

ELSE

COUNT ← SRC2[i+5:i]

IF COUNT &lt; 64

THEN DEST[i+63:i] ← SignExtend(SRC1[i+63:i] &gt;&gt; COUNT)

ELSE

FOR k ← 0 TO 63

DEST[i+k] ← SRC1[i+63]

ENDFOR;

FI

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[63:0] remains unchanged\*

ELSE ; zeroing-masking

DEST[63:0] ← 0

FI

FI;

ENDFOR;

DEST[MAX\_VL-1:VL] ← 0;

**Intel C/C++ Compiler Intrinsic Equivalent**

VPSRAVD \_\_m512i \_\_mm512\_srav\_epi32(\_\_m512i a, \_\_m512i cnt);

VPSRAVD \_\_m512i \_\_mm512\_mask\_srav\_epi32(\_\_m512i s, \_\_mmask16 m, \_\_m512i a, \_\_m512i cnt);

VPSRAVD \_\_m512i \_\_mm512\_maskz\_srav\_epi32(\_\_mmask16 m, \_\_m512i a, \_\_m512i cnt);

VPSRAVD \_\_m256i \_\_mm256\_srav\_epi32(\_\_m256i a, \_\_m256i cnt);

VPSRAVD \_\_m256i \_\_mm256\_mask\_srav\_epi32(\_\_m256i s, \_\_mmask8 m, \_\_m256i a, \_\_m256i cnt);

VPSRAVD \_\_m256i \_\_mm256\_maskz\_srav\_epi32(\_\_mmask8 m, \_\_m256i a, \_\_m256i cnt);

VPSRAVD \_\_m128i \_\_mm\_srav\_epi32(\_\_m128i a, \_\_m128i cnt);

VPSRAVD \_\_m128i \_\_mm\_mask\_srav\_epi32(\_\_m128i s, \_\_mmask8 m, \_\_m128i a, \_\_m128i cnt);

VPSRAVD \_\_m128i \_\_mm\_maskz\_srav\_epi32(\_\_mmask8 m, \_\_m128i a, \_\_m128i cnt);

VPSRAVQ \_\_m512i \_\_mm512\_srav\_epi64(\_\_m512i a, \_\_m512i cnt);

VPSRAVQ \_\_m512i \_\_mm512\_mask\_srav\_epi64(\_\_m512i s, \_\_mmask8 m, \_\_m512i a, \_\_m512i cnt);

VPSRAVQ \_\_m512i \_\_mm512\_maskz\_srav\_epi64(\_\_mmask8 m, \_\_m512i a, \_\_m512i cnt);

VPSRAVQ \_\_m256i \_\_mm256\_srav\_epi64(\_\_m256i a, \_\_m256i cnt);

VPSRAVQ \_\_m256i \_\_mm256\_mask\_srav\_epi64(\_\_m256i s, \_\_mmask8 m, \_\_m256i a, \_\_m256i cnt);

VPSRAVQ \_\_m256i \_\_mm256\_maskz\_srav\_epi64(\_\_mmask8 m, \_\_m256i a, \_\_m256i cnt);

VPSRAVQ \_\_m128i \_\_mm\_srav\_epi64(\_\_m128i a, \_\_m128i cnt);

VPSRAVQ \_\_m128i \_\_mm\_mask\_srav\_epi64(\_\_m128i s, \_\_mmask8 m, \_\_m128i a, \_\_m128i cnt);

VPSRAVQ \_\_m128i \_\_mm\_maskz\_srav\_epi64(\_\_mmask8 m, \_\_m128i a, \_\_m128i cnt);

VPSRAVW \_\_m512i \_\_mm512\_srav\_epi16(\_\_m512i a, \_\_m512i cnt);  
 VPSRAVW \_\_m512i \_\_mm512\_mask\_srav\_epi16(\_\_m512i s, \_\_mmask32 m, \_\_m512i a, \_\_m512i cnt);  
 VPSRAVW \_\_m512i \_\_mm512\_maskz\_srav\_epi16(\_\_mmask32 m, \_\_m512i a, \_\_m512i cnt);  
 VPSRAVW \_\_m256i \_\_mm256\_srav\_epi16(\_\_m256i a, \_\_m256i cnt);  
 VPSRAVW \_\_m256i \_\_mm256\_mask\_srav\_epi16(\_\_m256i s, \_\_mmask16 m, \_\_m256i a, \_\_m256i cnt);  
 VPSRAVW \_\_m256i \_\_mm256\_maskz\_srav\_epi16(\_\_mmask16 m, \_\_m256i a, \_\_m256i cnt);  
 VPSRAVW \_\_m128i \_\_mm\_srav\_epi16(\_\_m128i a, \_\_m128i cnt);  
 VPSRAVW \_\_m128i \_\_mm\_mask\_srav\_epi16(\_\_m128i s, \_\_mmask8 m, \_\_m128i a, \_\_m128i cnt);  
 VPSRAVW \_\_m128i \_\_mm\_maskz\_srav\_epi32(\_\_mmask8 m, \_\_m128i a, \_\_m128i cnt);  
 VPSRAVD \_\_m256i \_\_mm256\_srav\_epi32 (\_\_m256i m, \_\_m256i count)

### SIMD Floating-Point Exceptions

None

### Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.

## VPSRLVW/VPSRLVD/VPSRLVQ—Variable Bit Shift Right Logical

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.NDS.128.66.0F38.W0 45 /r VPSRLVD xmm1, xmm2, xmm3/m128	RVM	V/V	AVX2	Shift doublewords in xmm2 right by amount specified in the corresponding element of xmm3/m128 while shifting in 0s.
VEX.NDS.128.66.0F38.W1 45 /r VPSRLVQ xmm1, xmm2, xmm3/m128	RVM	V/V	AVX2	Shift quadwords in xmm2 right by amount specified in the corresponding element of xmm3/m128 while shifting in 0s.
VEX.NDS.256.66.0F38.W0 45 /r VPSRLVD ymm1, ymm2, ymm3/m256	RVM	V/V	AVX2	Shift doublewords in ymm2 right by amount specified in the corresponding element of ymm3/m256 while shifting in 0s.
VEX.NDS.256.66.0F38.W1 45 /r VPSRLVQ ymm1, ymm2, ymm3/m256	RVM	V/V	AVX2	Shift quadwords in ymm2 right by amount specified in the corresponding element of ymm3/m256 while shifting in 0s.
EVEX.NDS.128.66.0F38.W1 10 /r VPSRLVW xmm1 {k1}{z}, xmm2, xmm3/m128	FVM	V/V	AVX512VL AVX512BW	Shift words in xmm2 right by amount specified in the corresponding element of xmm3/m128 while shifting in 0s using writemask k1.
EVEX.NDS.256.66.0F38.W1 10 /r VPSRLVW ymm1 {k1}{z}, ymm2, ymm3/m256	FVM	V/V	AVX512VL AVX512BW	Shift words in ymm2 right by amount specified in the corresponding element of ymm3/m256 while shifting in 0s using writemask k1.
EVEX.NDS.512.66.0F38.W1 10 /r VPSRLVW zmm1 {k1}{z}, zmm2, zmm3/m512	FVM	V/V	AVX512BW	Shift words in zmm2 right by amount specified in the corresponding element of zmm3/m512 while shifting in 0s using writemask k1.
EVEX.NDS.128.66.0F38.W0 45 /r VPSRLVD xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	FV	V/V	AVX512VL AVX512F	Shift doublewords in xmm2 right by amount specified in the corresponding element of xmm3/m128/m32bcst while shifting in 0s using writemask k1.
EVEX.NDS.256.66.0F38.W0 45 /r VPSRLVD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	FV	V/V	AVX512VL AVX512F	Shift doublewords in ymm2 right by amount specified in the corresponding element of ymm3/m256/m32bcst while shifting in 0s using writemask k1.
EVEX.NDS.512.66.0F38.W0 45 /r VPSRLVD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	FV	V/V	AVX512F	Shift doublewords in zmm2 right by amount specified in the corresponding element of zmm3/m512/m32bcst while shifting in 0s using writemask k1.
EVEX.NDS.128.66.0F38.W1 45 /r VPSRLVQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	FV	V/V	AVX512VL AVX512F	Shift quadwords in xmm2 right by amount specified in the corresponding element of xmm3/m128/m64bcst while shifting in 0s using writemask k1.
EVEX.NDS.256.66.0F38.W1 45 /r VPSRLVQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	FV	V/V	AVX512VL AVX512F	Shift quadwords in ymm2 right by amount specified in the corresponding element of ymm3/m256/m64bcst while shifting in 0s using writemask k1.
EVEX.NDS.512.66.0F38.W1 45 /r VPSRLVQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	FV	V/V	AVX512F	Shift quadwords in zmm2 right by amount specified in the corresponding element of zmm3/m512/m64bcst while shifting in 0s using writemask k1.

## Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FVM	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA
FV	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

## Description

Shifts the bits in the individual data elements (words, doublewords or quadword) in the first source operand to the right by the count value of respective data elements in the second source operand. As the bits in the data elements are shifted right, the empty high-order bits are cleared (set to 0).

The count values are specified individually in each data element of the second source operand. If the unsigned integer value specified in the respective data element of the second source operand is greater than 15 (for word), 31 (for doublewords), or 63 (for a quadword), then the destination data element are written with 0.

VEX.128 encoded version: The destination and first source operands are XMM registers. The count operand can be either an XMM register or a 128-bit memory location. Bits (MAX\_VL-1:128) of the corresponding destination register are zeroed.

VEX.256 encoded version: The destination and first source operands are YMM registers. The count operand can be either an YMM register or a 256-bit memory. Bits (MAX\_VL-1:256) of the corresponding ZMM register are zeroed.

EVEX encoded VPSRLVD/Q: The destination and first source operands are ZMM/YMM/XMM registers. The count operand can be either a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512-bit vector broadcasted from a 32/64-bit memory location. The destination is conditionally updated with writemask k1.

EVEX encoded VPSRLVW: The destination and first source operands are ZMM/YMM/XMM registers. The count operand can be either a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination is conditionally updated with writemask k1.

## Operation

**VPSRLVW (EVEX encoded version)**

(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] ← ZeroExtend(SRC1[i+15:i] >> SRC2[j+15:i])
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+15:i] remains unchanged*
    ELSE ; zeroing-masking
      DEST[i+15:i] ← 0
    FI
  FI;
ENDFOR;
DEST[MAX_VL-1:VL] ← 0;

```

**VPSRLVD (VEX.128 version)**

```

COUNT_0 ← SRC2[31 : 0]
(* Repeat Each COUNT_i for the 2nd through 4th dwords of SRC2*)
COUNT_3 ← SRC2[127 : 96];
IF COUNT_0 < 32 THEN
    DEST[31:0] ← ZeroExtend(SRC1[31:0] >> COUNT_0);
ELSE
    DEST[31:0] ← 0;
    (* Repeat shift operation for 2nd through 4th dwords *)
IF COUNT_3 < 32 THEN
    DEST[127:96] ← ZeroExtend(SRC1[127:96] >> COUNT_3);
ELSE
    DEST[127:96] ← 0;
DEST[MAX_VL-1:128] ← 0;

```

**VPSRLVD (VEX.256 version)**

```

COUNT_0 ← SRC2[31 : 0];
(* Repeat Each COUNT_i for the 2nd through 7th dwords of SRC2*)
COUNT_7 ← SRC2[255 : 224];
IF COUNT_0 < 32 THEN
    DEST[31:0] ← ZeroExtend(SRC1[31:0] >> COUNT_0);
ELSE
    DEST[31:0] ← 0;
    (* Repeat shift operation for 2nd through 7th dwords *)
IF COUNT_7 < 32 THEN
    DEST[255:224] ← ZeroExtend(SRC1[255:224] >> COUNT_7);
ELSE
    DEST[255:224] ← 0;
DEST[MAX_VL-1:256] ← 0;

```

**VPSRLVD (EVEX encoded version)**

```

(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask* THEN
        IF (EVEX.b = 1) AND (SRC2 *is memory*)
            THEN DEST[i+31:i] ← ZeroExtend(SRC1[i+31:i] >> SRC2[31:0])
            ELSE DEST[i+31:i] ← ZeroExtend(SRC1[i+31:i] >> SRC2[i+31:i])
        FI;
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+31:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+31:i] ← 0
        FI
    FI;
ENDFOR;
DEST[MAX_VL-1:VL] ← 0;

```

**VPSRLVQ (VEX.128 version)**

```

COUNT_0 ← SRC2[63 : 0];
COUNT_1 ← SRC2[127 : 64];
IF COUNT_0 < 64 THEN
    DEST[63:0] ← ZeroExtend(SRC1[63:0] >> COUNT_0);
ELSE
    DEST[63:0] ← 0;
IF COUNT_1 < 64 THEN
    DEST[127:64] ← ZeroExtend(SRC1[127:64] >> COUNT_1);
ELSE
    DEST[127:64] ← 0;
DEST[MAX_VL-1:128] ← 0;

```

**VPSRLVQ (VEX.256 version)**

```

COUNT_0 ← SRC2[63 : 0];
(* Repeat Each COUNT_i for the 2nd through 4th dwords of SRC2*)
COUNT_3 ← SRC2[255 : 192];
IF COUNT_0 < 64 THEN
    DEST[63:0] ← ZeroExtend(SRC1[63:0] >> COUNT_0);
ELSE
    DEST[63:0] ← 0;
(* Repeat shift operation for 2nd through 4th dwords *)
IF COUNT_3 < 64 THEN
    DEST[255:192] ← ZeroExtend(SRC1[255:192] >> COUNT_3);
ELSE
    DEST[255:192] ← 0;
DEST[MAX_VL-1:256] ← 0;

```

**VPSRLVQ (EVEX encoded version)**

```

(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) AND (SRC2 *is memory*)
            THEN DEST[i+63:i] ← ZeroExtend(SRC1[i+63:i] >> SRC2[63:0])
            ELSE DEST[i+63:i] ← ZeroExtend(SRC1[i+63:i] >> SRC2[i+63:i])
        FI;
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+63:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+63:i] ← 0
        FI
    FI;
ENDFOR;
DEST[MAX_VL-1:VL] ← 0;

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VPSRLVW __m512i __mm512_srlv_epi16(__m512i a, __m512i cnt);
VPSRLVW __m512i __mm512_mask_srlv_epi16(__m512i s, __mmask32 k, __m512i a, __m512i cnt);
VPSRLVW __m512i __mm512_maskz_srlv_epi16(__mmask32 k, __m512i a, __m512i cnt);
VPSRLVW __m256i __mm256_mask_srlv_epi16(__m256i s, __mmask16 k, __m256i a, __m256i cnt);
VPSRLVW __m256i __mm256_maskz_srlv_epi16(__mmask16 k, __m256i a, __m256i cnt);
VPSRLVW __m128i __mm_mask_srlv_epi16(__m128i s, __mmask8 k, __m128i a, __m128i cnt);

```

VPSRLVW \_\_m128i \_mm\_maskz\_srlv\_epi16( \_\_mmask8 k, \_\_m128i a, \_\_m128i cnt);  
 VPSRLVW \_\_m256i \_mm256\_srlv\_epi32( \_\_m256i m, \_\_m256i count)  
 VPSRLVD \_\_m512i \_mm512\_srlv\_epi32( \_\_m512i a, \_\_m512i cnt);  
 VPSRLVD \_\_m512i \_mm512\_mask\_srlv\_epi32( \_\_m512i s, \_\_mmask16 k, \_\_m512i a, \_\_m512i cnt);  
 VPSRLVD \_\_m512i \_mm512\_maskz\_srlv\_epi32( \_\_mmask16 k, \_\_m512i a, \_\_m512i cnt);  
 VPSRLVD \_\_m256i \_mm256\_mask\_srlv\_epi32( \_\_m256i s, \_\_mmask8 k, \_\_m256i a, \_\_m256i cnt);  
 VPSRLVD \_\_m256i \_mm256\_maskz\_srlv\_epi32( \_\_mmask8 k, \_\_m256i a, \_\_m256i cnt);  
 VPSRLVD \_\_m128i \_mm\_mask\_srlv\_epi32( \_\_m128i s, \_\_mmask8 k, \_\_m128i a, \_\_m128i cnt);  
 VPSRLVD \_\_m128i \_mm\_maskz\_srlv\_epi32( \_\_mmask8 k, \_\_m128i a, \_\_m128i cnt);  
 VPSRLVQ \_\_m512i \_mm512\_srlv\_epi64( \_\_m512i a, \_\_m512i cnt);  
 VPSRLVQ \_\_m512i \_mm512\_mask\_srlv\_epi64( \_\_m512i s, \_\_mmask8 k, \_\_m512i a, \_\_m512i cnt);  
 VPSRLVQ \_\_m512i \_mm512\_maskz\_srlv\_epi64( \_\_mmask8 k, \_\_m512i a, \_\_m512i cnt);  
 VPSRLVQ \_\_m256i \_mm256\_mask\_srlv\_epi64( \_\_m256i s, \_\_mmask8 k, \_\_m256i a, \_\_m256i cnt);  
 VPSRLVQ \_\_m256i \_mm256\_maskz\_srlv\_epi64( \_\_mmask8 k, \_\_m256i a, \_\_m256i cnt);  
 VPSRLVQ \_\_m128i \_mm\_mask\_srlv\_epi64( \_\_m128i s, \_\_mmask8 k, \_\_m128i a, \_\_m128i cnt);  
 VPSRLVQ \_\_m128i \_mm\_maskz\_srlv\_epi64( \_\_mmask8 k, \_\_m128i a, \_\_m128i cnt);  
 VPSRLVQ \_\_m256i \_mm256\_srlv\_epi64( \_\_m256i m, \_\_m256i count)  
 VPSRLVD \_\_m128i \_mm\_srlv\_epi32( \_\_m128i a, \_\_m128i cnt);  
 VPSRLVQ \_\_m128i \_mm\_srlv\_epi64( \_\_m128i a, \_\_m128i cnt);

### SIMD Floating-Point Exceptions

None

### Other Exceptions

VEX-encoded instructions, see Exceptions Type 4.

EVEX-encoded VPSRLVD/Q, see Exceptions Type E4.

EVEX-encoded VPSRLVW, see Exceptions Type E4.nb.



## VPTERNLOGD/VPTERNLOGQ—Bitwise Ternary Logic

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.DDS.128.66.0F3A.W0 25 /r ib VPTERNLOGD xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst, imm8	FV	V/V	AVX512VL AVX512F	Bitwise ternary logic taking xmm1, xmm2 and xmm3/m128/m32bcst as source operands and writing the result to xmm1 under writemask k1 with dword granularity. The immediate value determines the specific binary function being implemented.
EVEX.DDS.256.66.0F3A.W0 25 /r ib VPTERNLOGD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst, imm8	FV	V/V	AVX512VL AVX512F	Bitwise ternary logic taking ymm1, ymm2 and ymm3/m256/m32bcst as source operands and writing the result to ymm1 under writemask k1 with dword granularity. The immediate value determines the specific binary function being implemented.
EVEX.DDS.512.66.0F3A.W0 25 /r ib VPTERNLOGD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst, imm8	FV	V/V	AVX512F	Bitwise ternary logic taking zmm1, zmm2 and zmm3/m512/m32bcst as source operands and writing the result to zmm1 under writemask k1 with dword granularity. The immediate value determines the specific binary function being implemented.
EVEX.DDS.128.66.0F3A.W1 25 /r ib VPTERNLOGQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst, imm8	FV	V/V	AVX512VL AVX512F	Bitwise ternary logic taking xmm1, xmm2 and xmm3/m128/m64bcst as source operands and writing the result to xmm1 under writemask k1 with qword granularity. The immediate value determines the specific binary function being implemented.
EVEX.DDS.256.66.0F3A.W1 25 /r ib VPTERNLOGQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst, imm8	FV	V/V	AVX512VL AVX512F	Bitwise ternary logic taking ymm1, ymm2 and ymm3/m256/m64bcst as source operands and writing the result to ymm1 under writemask k1 with qword granularity. The immediate value determines the specific binary function being implemented.
EVEX.DDS.512.66.0F3A.W1 25 /r ib VPTERNLOGQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst, imm8	FV	V/V	AVX512F	Bitwise ternary logic taking zmm1, zmm2 and zmm3/m512/m64bcst as source operands and writing the result to zmm1 under writemask k1 with qword granularity. The immediate value determines the specific binary function being implemented.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
FV	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	Imm8

### Description

VPTERNLOGD/Q takes three bit vectors of 512-bit length (in the first, second and third operand) as input data to form a set of 512 indices, each index is comprised of one bit from each input vector. The imm8 byte specifies a boolean logic table producing a binary value for each 3-bit index value. The final 512-bit boolean result is written to the destination operand (the first operand) using the writemask k1 with the granularity of doubleword element or quadword element into the destination.

The destination operand is a ZMM (EVEX.512)/YMM (EVEX.256)/XMM (EVEX.128) register. The first source 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 32/64-bit memory location. The destination operand is a ZMM register conditionally updated with writemask k1.

Table 5-11 shows two examples of Boolean functions specified by immediate values 0xE2 and 0xE4, with the look up result listed in the fourth column following the three columns containing all possible values of the 3-bit index.

Table 5-11. Examples of VPTERNLOGD/Q Imm8 Boolean Function and Input Index Values

VPTERNLOGD reg1, reg2, src3, 0xE2			Bit Result with Imm8=0xE2	VPTERNLOGD reg1, reg2, src3, 0xE4			Bit Result with Imm8=0xE4
Bit(reg1)	Bit(reg2)	Bit(src3)		Bit(reg1)	Bit(reg2)	Bit(src3)	
0	0	0	0	0	0	0	0
0	0	1	1	0	0	1	0
0	1	0	0	0	1	0	1
0	1	1	0	0	1	1	0
1	0	0	0	1	0	0	0
1	0	1	1	1	0	1	1
1	1	0	1	1	1	0	1
1	1	1	1	1	1	1	1

Specifying different values in imm8 will allow any arbitrary three-input Boolean functions to be implemented in software using VPTERNLOGD/Q. Table 5-1 and Table 5-2 provide a mapping of all 256 possible imm8 values to various Boolean expressions.

### Operation

#### VPTERNLOGD (EVEX encoded versions)

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

  i ← j \* 32

  IF k1[j] OR \*no writemask\*

    THEN

      FOR k ← 0 TO 31

        IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

          THEN DEST[j][k] ← imm[(DEST[i+k] << 2) + (SRC1[ i+k ] << 1) + SRC2[ k ]]

          ELSE DEST[j][k] ← imm[(DEST[i+k] << 2) + (SRC1[ i+k ] << 1) + SRC2[ i+k ]]

        FI;

          ; table lookup of immediate bellow;

    ELSE

      IF \*merging-masking\*                                 ; merging-masking

        THEN \*DEST[31+i:i] remains unchanged\*

        ELSE                                                 ; zeroing-masking

          DEST[31+i:i] ← 0

      FI;

  FI;

ENDFOR;

DEST[MAX\_VL-1:VL] ← 0

**VPTERNLOGQ (EVEX encoded versions)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN
      FOR k ← 0 TO 63
        IF (EVEX.b = 1) AND (SRC2 *is memory*)
          THEN DEST[j][k] ← imm[(DEST[i+k] << 2) + (SRC1[i+k] << 1) + SRC2[k]]
          ELSE DEST[j][k] ← imm[(DEST[i+k] << 2) + (SRC1[i+k] << 1) + SRC2[i+k]]
        FI;
          ; table lookup of immediate below;
      ELSE
        IF *merging-masking* ; merging-masking
          THEN *DEST[63+i:i] remains unchanged*
          ELSE ; zeroing-masking
            DEST[63+i:i] ← 0
          FI;
        FI;
      ENDFOR;
    DEST[MAX_VL-1:VL] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalents**

```

VPTERNLOGD __m512i_mm512_ternarylogic_epi32(__m512i a, __m512i b, int imm);
VPTERNLOGD __m512i_mm512_mask_ternarylogic_epi32(__m512i s, __mmask16 m, __m512i a, __m512i b, int imm);
VPTERNLOGD __m512i_mm512_maskz_ternarylogic_epi32(__mmask m, __m512i a, __m512i b, int imm);
VPTERNLOGD __m256i_mm256_ternarylogic_epi32(__m256i a, __m256i b, int imm);
VPTERNLOGD __m256i_mm256_mask_ternarylogic_epi32(__m256i s, __mmask8 m, __m256i a, __m256i b, int imm);
VPTERNLOGD __m256i_mm256_maskz_ternarylogic_epi32(__mmask8 m, __m256i a, __m256i b, int imm);
VPTERNLOGD __m128i_mm_ternarylogic_epi32(__m128i a, __m128i b, int imm);
VPTERNLOGD __m128i_mm_mask_ternarylogic_epi32(__m128i s, __mmask8 m, __m128i a, __m128i b, int imm);
VPTERNLOGD __m128i_mm_maskz_ternarylogic_epi32(__mmask8 m, __m128i a, __m128i b, int imm);
VPTERNLOGQ __m512i_mm512_ternarylogic_epi64(__m512i a, __m512i b, int imm);
VPTERNLOGQ __m512i_mm512_mask_ternarylogic_epi64(__m512i s, __mmask8 m, __m512i a, __m512i b, int imm);
VPTERNLOGQ __m512i_mm512_maskz_ternarylogic_epi64(__mmask8 m, __m512i a, __m512i b, int imm);
VPTERNLOGQ __m256i_mm256_ternarylogic_epi64(__m256i a, __m256i b, int imm);
VPTERNLOGQ __m256i_mm256_mask_ternarylogic_epi64(__m256i s, __mmask8 m, __m256i a, __m256i b, int imm);
VPTERNLOGQ __m256i_mm256_maskz_ternarylogic_epi64(__mmask8 m, __m256i a, __m256i b, int imm);
VPTERNLOGQ __m128i_mm_ternarylogic_epi64(__m128i a, __m128i b, int imm);
VPTERNLOGQ __m128i_mm_mask_ternarylogic_epi64(__m128i s, __mmask8 m, __m128i a, __m128i b, int imm);
VPTERNLOGQ __m128i_mm_maskz_ternarylogic_epi64(__mmask8 m, __m128i a, __m128i b, int imm);

```

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type E4.

## VPTESTMB/VPTESTMW/VPTESTMD/VPTESTMQ—Logical AND and Set Mask

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.128.66.0F38.W0 26 /r VPTESTMB k2 {k1}, xmm2, xmm3/m128	FVM	V/V	AVX512VL AVX512BW	Bitwise AND of packed byte integers in xmm2 and xmm3/m128 and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.
EVEX.NDS.256.66.0F38.W0 26 /r VPTESTMB k2 {k1}, ymm2, ymm3/m256	FVM	V/V	AVX512VL AVX512BW	Bitwise AND of packed byte integers in ymm2 and ymm3/m256 and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.
EVEX.NDS.512.66.0F38.W0 26 /r VPTESTMB k2 {k1}, zmm2, zmm3/m512	FVM	V/V	AVX512BW	Bitwise AND of packed byte integers in zmm2 and zmm3/m512 and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.
EVEX.NDS.128.66.0F38.W1 26 /r VPTESTMW k2 {k1}, xmm2, xmm3/m128	FVM	V/V	AVX512VL AVX512BW	Bitwise AND of packed word integers in xmm2 and xmm3/m128 and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.
EVEX.NDS.256.66.0F38.W1 26 /r VPTESTMW k2 {k1}, ymm2, ymm3/m256	FVM	V/V	AVX512VL AVX512BW	Bitwise AND of packed word integers in ymm2 and ymm3/m256 and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.
EVEX.NDS.512.66.0F38.W1 26 /r VPTESTMW k2 {k1}, zmm2, zmm3/m512	FVM	V/V	AVX512BW	Bitwise AND of packed word integers in zmm2 and zmm3/m512 and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.
EVEX.NDS.128.66.0F38.W0 27 /r VPTESTMD k2 {k1}, xmm2, xmm3/m128/m32bcst	FV	V/V	AVX512VL AVX512F	Bitwise AND of packed doubleword integers in xmm2 and xmm3/m128/m32bcst and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.
EVEX.NDS.256.66.0F38.W0 27 /r VPTESTMD k2 {k1}, ymm2, ymm3/m256/m32bcst	FV	V/V	AVX512VL AVX512F	Bitwise AND of packed doubleword integers in ymm2 and ymm3/m256/m32bcst and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.
EVEX.NDS.512.66.0F38.W0 27 /r VPTESTMD k2 {k1}, zmm2, zmm3/m512/m32bcst	FV	V/V	AVX512F	Bitwise AND of packed doubleword integers in zmm2 and zmm3/m512/m32bcst and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.
EVEX.NDS.128.66.0F38.W1 27 /r VPTESTMQ k2 {k1}, xmm2, xmm3/m128/m64bcst	FV	V/V	AVX512VL AVX512F	Bitwise AND of packed quadword integers in xmm2 and xmm3/m128/m64bcst and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.
EVEX.NDS.256.66.0F38.W1 27 /r VPTESTMQ k2 {k1}, ymm2, ymm3/m256/m64bcst	FV	V/V	AVX512VL AVX512F	Bitwise AND of packed quadword integers in ymm2 and ymm3/m256/m64bcst and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.
EVEX.NDS.512.66.0F38.W1 27 /r VPTESTMQ k2 {k1}, zmm2, zmm3/m512/m64bcst	FV	V/V	AVX512F	Bitwise AND of packed quadword integers in zmm2 and zmm3/m512/m64bcst and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.

## Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
FVM	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA
FV	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

## Description

Performs a bitwise logical AND operation on the first source operand (the second operand) and second source operand (the third operand) and stores the result in the destination operand (the first operand) under the writemask. Each bit of the result is set to 1 if the bitwise AND of the corresponding elements of the first and second src operands is non-zero; otherwise it is set to 0.

**VPTESTMD/VPTESTMQ:** The first source 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 32/64-bit memory location. The destination operand is a mask register updated under the writemask.

**VPTESTMB/VPTESTMW:** The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operand is a mask register updated under the writemask.

## Operation

**VPTESTMB (EVEX encoded versions)**

(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[j] ← (SRC1[i+7:i] BITWISE AND SRC2[i+7:i] != 0)? 1 : 0;

    ELSE DEST[j] = 0 ; zeroing-masking only

  FI;

ENDFOR

DEST[MAX\_KL-1:KL] ← 0

**VPTESTMW (EVEX encoded versions)**

(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[j] ← (SRC1[i+15:i] BITWISE AND SRC2[i+15:i] != 0)? 1 : 0;

    ELSE DEST[j] = 0 ; zeroing-masking only

  FI;

ENDFOR

DEST[MAX\_KL-1:KL] ← 0

**VPTESTMD (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

  i ← j \* 32

  IF k1[j] OR \*no writemask\*

    THEN

      IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

        THEN DEST[j] ← (SRC1[i+31:i] BITWISE AND SRC2[31:0] != 0)? 1 : 0;

        ELSE DEST[j] ← (SRC1[i+31:i] BITWISE AND SRC2[i+31:i] != 0)? 1 : 0;

      FI;

    ELSE DEST[j] ← 0 ; zeroing-masking only

  FI;

ENDFOR

DEST[MAX\_KL-1:KL] ← 0

### VPTESTMQ (EVEX encoded versions)

(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) AND (SRC2 \*is memory\*)

        THEN DEST[j] ← (SRC1[i+63:i] BITWISE AND SRC2[63:0] != 0)? 1 : 0;

        ELSE DEST[j] ← (SRC1[i+63:i] BITWISE AND SRC2[i+63:i] != 0)? 1 : 0;

      FI;

    ELSE DEST[j] ← 0 ; zeroing-masking only

  FI;

ENDFOR

DEST[MAX\_KL-1:KL] ← 0

### Intel C/C++ Compiler Intrinsic Equivalents

VPTESTMB \_\_mmask64 \_mm512\_test\_epi8\_mask( \_\_m512i a, \_\_m512i b);

VPTESTMB \_\_mmask64 \_mm512\_mask\_test\_epi8\_mask(\_\_mmask64, \_\_m512i a, \_\_m512i b);

VPTESTMW \_\_mmask32 \_mm512\_test\_epi16\_mask( \_\_m512i a, \_\_m512i b);

VPTESTMW \_\_mmask32 \_mm512\_mask\_test\_epi16\_mask(\_\_mmask32, \_\_m512i a, \_\_m512i b);

VPTESTMD \_\_mmask16 \_mm512\_test\_epi32\_mask( \_\_m512i a, \_\_m512i b);

VPTESTMD \_\_mmask16 \_mm512\_mask\_test\_epi32\_mask(\_\_mmask16, \_\_m512i a, \_\_m512i b);

VPTESTMQ \_\_mmask8 \_mm512\_test\_epi64\_mask(\_\_m512i a, \_\_m512i b);

VPTESTMQ \_\_mmask8 \_mm512\_mask\_test\_epi64\_mask(\_\_mmask8, \_\_m512i a, \_\_m512i b);

### SIMD Floating-Point Exceptions

None

### Other Exceptions

VPTESTMD/Q: See Exceptions Type E4.

VPTESTMB/W: See Exceptions Type E4.nb.

## VPTTESTNMB/W/D/Q—Logical NAND and Set

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID	Description
EVEX.NDS.128.F3.0F38.W0 26 /r VPTTESTNMB k2 {k1}, xmm2, xmm3/m128	FVM	V/V	AVX512VL AVX512BW	Bitwise NAND of packed byte integers in xmm2 and xmm3/m128 and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.
EVEX.NDS.256.F3.0F38.W0 26 /r VPTTESTNMB k2 {k1}, ymm2, ymm3/m256	FVM	V/V	AVX512VL AVX512BW	Bitwise NAND of packed byte integers in ymm2 and ymm3/m256 and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.
EVEX.NDS.512.F3.0F38.W0 26 /r VPTTESTNMB k2 {k1}, zmm2, zmm3/m512	FVM	V/V	AVX512F AVX512BW	Bitwise NAND of packed byte integers in zmm2 and zmm3/m512 and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.
EVEX.NDS.128.F3.0F38.W1 26 /r VPTTESTNMW k2 {k1}, xmm2, xmm3/m128	FVM	V/V	AVX512VL AVX512BW	Bitwise NAND of packed word integers in xmm2 and xmm3/m128 and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.
EVEX.NDS.256.F3.0F38.W1 26 /r VPTTESTNMW k2 {k1}, ymm2, ymm3/m256	FVM	V/V	AVX512VL AVX512BW	Bitwise NAND of packed word integers in ymm2 and ymm3/m256 and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.
EVEX.NDS.512.F3.0F38.W1 26 /r VPTTESTNMW k2 {k1}, zmm2, zmm3/m512	FVM	V/V	AVX512F AVX512BW	Bitwise NAND of packed word integers in zmm2 and zmm3/m512 and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.
EVEX.NDS.128.F3.0F38.W0 27 /r VPTTESTNMD k2 {k1}, xmm2, xmm3/m128/m32bcst	FV	V/V	AVX512VL AVX512F	Bitwise NAND of packed doubleword integers in xmm2 and xmm3/m128/m32bcst and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.
EVEX.NDS.256.F3.0F38.W0 27 /r VPTTESTNMD k2 {k1}, ymm2, ymm3/m256/m32bcst	FV	V/V	AVX512VL AVX512F	Bitwise NAND of packed doubleword integers in ymm2 and ymm3/m256/m32bcst and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.
EVEX.NDS.512.F3.0F38.W0 27 /r VPTTESTNMD k2 {k1}, zmm2, zmm3/m512/m32bcst	FV	V/V	AVX512F	Bitwise NAND of packed doubleword integers in zmm2 and zmm3/m512/m32bcst and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.
EVEX.NDS.128.F3.0F38.W1 27 /r VPTTESTNMQ k2 {k1}, xmm2, xmm3/m128/m64bcst	FV	V/V	AVX512VL AVX512F	Bitwise NAND of packed quadword integers in xmm2 and xmm3/m128/m64bcst and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.
EVEX.NDS.256.F3.0F38.W1 27 /r VPTTESTNMQ k2 {k1}, ymm2, ymm3/m256/m64bcst	FV	V/V	AVX512VL AVX512F	Bitwise NAND of packed quadword integers in ymm2 and ymm3/m256/m64bcst and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.
EVEX.NDS.512.F3.0F38.W1 27 /r VPTTESTNMQ k2 {k1}, zmm2, zmm3/m512/m64bcst	FV	V/V	AVX512F	Bitwise NAND of packed quadword integers in zmm2 and zmm3/m512/m64bcst and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.

## Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
FVM	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA
FV	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

## Description

Performs a bitwise logical NAND operation on the byte/word/doubleword/quadword element of the first source operand (the second operand) with the corresponding element of the second source operand (the third operand) and stores the logical comparison result into each bit of the destination operand (the first operand) according to the writemask k1. Each bit of the result is set to 1 if the bitwise AND of the corresponding elements of the first and second src operands is zero; otherwise it is set to 0.

EVEX encoded VPTESTNMD/Q: The first source operand is a ZMM/YMM/XMM registers. 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 32/64-bit memory location. The destination is updated according to the writemask.

EVEX encoded VPTESTNMB/W: The first source operand is a ZMM/YMM/XMM registers. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination is updated according to the writemask.

## Operation

**VPTESTNMB**

(KL, VL) = (16, 128), (32, 256), (64, 512)

FOR j ← 0 TO KL-1

  i ← j\*8

  IF MaskBit(j) OR \*no writemask\*

    THEN

      DEST[j] ← (SRC1[i+7:i] BITWISE AND SRC2[i+7:i] == 0)? 1 : 0

    ELSE DEST[j] ← 0; zeroing masking only

  FI

ENDFOR

DEST[MAX\_KL-1:KL] ← 0

**VPTESTNMW**

(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j ← 0 TO KL-1

  i ← j\*16

  IF MaskBit(j) OR \*no writemask\*

    THEN

      DEST[j] ← (SRC1[i+15:i] BITWISE AND SRC2[i+15:i] == 0)? 1 : 0

    ELSE DEST[j] ← 0; zeroing masking only

  FI

ENDFOR

DEST[MAX\_KL-1:KL] ← 0

**VPTESTNMD**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

  i ← j\*32

  IF MaskBit(j) OR \*no writemask\*

    THEN

      IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

        THEN DEST[i+31:i] ← (SRC1[i+31:i] BITWISE AND SRC2[31:0] == 0)? 1 : 0

        ELSE DEST[j] ← (SRC1[i+31:i] BITWISE AND SRC2[i+31:i] == 0)? 1 : 0

    FI



```

        ELSE DEST[j] ← 0; zeroing masking only
    FI
ENDFOR
DEST[MAX_KL-1:KL] ← 0

VPTTESTNMQ
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j*64
    IF MaskBit(j) OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN DEST[j] ← (SRC1[i+63:i] BITWISE AND SRC2[63:0] != 0)? 1 : 0;
                ELSE DEST[j] ← (SRC1[i+63:i] BITWISE AND SRC2[i+63:i] != 0)? 1 : 0;
            FI;
        ELSE DEST[j] ← 0; zeroing masking only
    FI
ENDFOR
DEST[MAX_KL-1:KL] ← 0

```

### Intel C/C++ Compiler Intrinsic Equivalent

```

VPTTESTNMB __mmask64 __mm512_testn_epi8_mask(__m512i a, __m512i b);
VPTTESTNMB __mmask64 __mm512_mask_testn_epi8_mask(__mmask64, __m512i a, __m512i b);
VPTTESTNMB __mmask32 __mm256_testn_epi8_mask(__m256i a, __m256i b);
VPTTESTNMB __mmask32 __mm256_mask_testn_epi8_mask(__mmask32, __m256i a, __m256i b);
VPTTESTNMB __mmask16 __mm_testn_epi8_mask(__m128i a, __m128i b);
VPTTESTNMB __mmask16 __mm_mask_testn_epi8_mask(__mmask16, __m128i a, __m128i b);
VPTTESTNMW __mmask32 __mm512_testn_epi16_mask(__m512i a, __m512i b);
VPTTESTNMW __mmask32 __mm512_mask_testn_epi16_mask(__mmask32, __m512i a, __m512i b);
VPTTESTNMW __mmask16 __mm256_testn_epi16_mask(__m256i a, __m256i b);
VPTTESTNMW __mmask16 __mm256_mask_testn_epi16_mask(__mmask16, __m256i a, __m256i b);
VPTTESTNMW __mmask8 __mm_testn_epi16_mask(__m128i a, __m128i b);
VPTTESTNMW __mmask8 __mm_mask_testn_epi16_mask(__mmask8, __m128i a, __m128i b);
VPTTESTNMD __mmask16 __mm512_testn_epi32_mask(__m512i a, __m512i b);
VPTTESTNMD __mmask16 __mm512_mask_testn_epi32_mask(__mmask16, __m512i a, __m512i b);
VPTTESTNMD __mmask8 __mm256_testn_epi32_mask(__m256i a, __m256i b);
VPTTESTNMD __mmask8 __mm256_mask_testn_epi32_mask(__mmask8, __m256i a, __m256i b);
VPTTESTNMD __mmask8 __mm_testn_epi32_mask(__m128i a, __m128i b);
VPTTESTNMD __mmask8 __mm_mask_testn_epi32_mask(__mmask8, __m128i a, __m128i b);
VPTTESTNMQ __mmask8 __mm512_testn_epi64_mask(__m512i a, __m512i b);
VPTTESTNMQ __mmask8 __mm512_mask_testn_epi64_mask(__mmask8, __m512i a, __m512i b);
VPTTESTNMQ __mmask8 __mm256_testn_epi64_mask(__m256i a, __m256i b);
VPTTESTNMQ __mmask8 __mm256_mask_testn_epi64_mask(__mmask8, __m256i a, __m256i b);
VPTTESTNMQ __mmask8 __mm_testn_epi64_mask(__m128i a, __m128i b);
VPTTESTNMQ __mmask8 __mm_mask_testn_epi64_mask(__mmask8, __m128i a, __m128i b);

```

### SIMD Floating-Point Exceptions

None

### Other Exceptions

VPTTESTNMD/VPTTESTNMQ: See Exceptions Type E4.

VPTTESTNMB/VPTTESTNMW: See Exceptions Type E4.nb.

## VRANGEPD—Range Restriction Calculation For Packed Pairs of Float64 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.128.66.0F3A.W1 50 /r ib VRANGEPD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst, imm8	FV	V/V	AVX512VL AVX512DQ	Calculate two RANGE operation output value from 2 pairs of double-precision floating-point values in xmm2 and xmm3/m128/m32bcst, store the results to xmm1 under the writemask k1. Imm8 specifies the comparison and sign of the range operation.
EVEX.NDS.256.66.0F3A.W1 50 /r ib VRANGEPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst, imm8	FV	V/V	AVX512VL AVX512DQ	Calculate four RANGE operation output value from 4pairs of double-precision floating-point values in ymm2 and ymm3/m256/m32bcst, store the results to ymm1 under the writemask k1. Imm8 specifies the comparison and sign of the range operation.
EVEX.NDS.512.66.0F3A.W1 50 /r ib VRANGEPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{sae}, imm8	FV	V/V	AVX512DQ	Calculate eight RANGE operation output value from 8 pairs of double-precision floating-point values in zmm2 and zmm3/m512/m32bcst, store the results to zmm1 under the writemask k1. Imm8 specifies the comparison and sign of the range operation.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
FV	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	Imm8

### Description

This instruction calculates 2/4/8 range operation outputs from two sets of packed input double-precision FP values in the first source operand (the second operand) and the second source operand (the third operand). The range outputs are written to the destination operand (the first operand) under the writemask k1.

Bits7:4 of imm8 byte must be zero. The range operation output is performed in two parts, each configured by a two-bit control field within imm8[3:0]:

- Imm8[1:0] specifies the initial comparison operation to be one of max, min, max absolute value or min absolute value of the input value pair. Each comparison of two input values produces an intermediate result that combines with the sign selection control (Imm8[3:2]) to determine the final range operation output.
- Imm8[3:2] specifies the sign of the range operation output to be one of the following: from the first input value, from the comparison result, set or clear.

The encodings of Imm8[1:0] and Imm8[3:2] are shown in Figure 5-25.

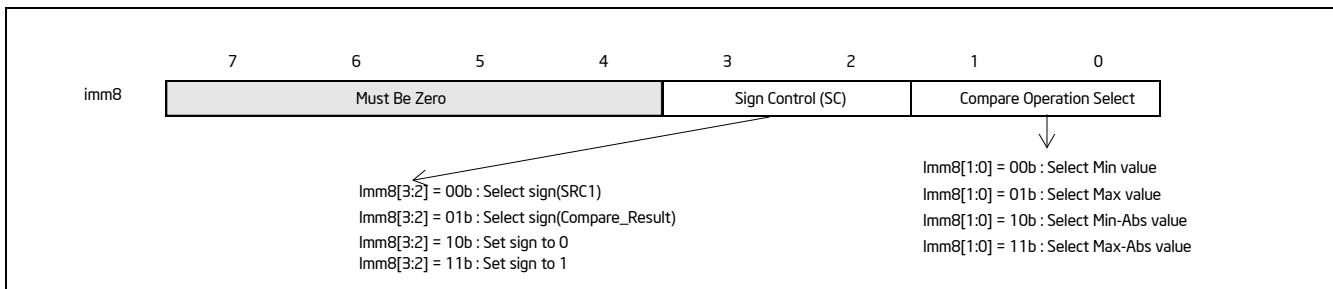


Figure 5-25. Imm8 Controls for VRANGEPD/SD/PS/SS

When one or more of the input value is a NAN, the comparison operation may signal invalid exception (IE). Details with one or more input value is NAN is listed in Table 5-12. If the comparison raises an IE, the sign select control (Imm8[3:2]) has no effect to the range operation output, this is indicated also in Table 5-12.

When both input values are zeros of opposite signs, the comparison operation of MIN/MAX in the range compare operation is slightly different from the conceptually similar FP MIN/MAX operation that are found in the instructions VMAXPD/VMINPD. The details of MIN/MAX/MIN\_ABS/MAX\_ABS operation for VRANGEPD/PS/SD/SS for magnitude-0, opposite-signed input cases are listed in Table 5-13.

Additionally, non-zero, equal-magnitude with opposite-sign input values perform MIN\_ABS or MAX\_ABS comparison operation with result listed in Table 5-14.

**Table 5-12. Signaling of Comparison Operation of One or More NaN Input Values and Effect of 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	Applicable
qNaN1	Norm2	Norm2	No	Applicable
Norm1	sNaN2	Quiet(sNaN2)	Yes	Ignored
Norm1	qNaN2	Norm1	No	Applicable

**Table 5-13. Comparison Result for Opposite-Signed Zero Cases for MIN, MIN\_ABS and MAX, MAX\_ABS**

MIN and MIN_ABS			MAX and MAX_ABS		
Src1	Src2	Result	Src1	Src2	Result
+0	-0	-0	+0	-0	+0
-0	+0	-0	-0	+0	+0

**Table 5-14. Comparison Result of Equal-Magnitude Input Cases for MIN\_ABS and MAX\_ABS, ( $|a| = |b|$ ,  $a > 0$ ,  $b < 0$ )**

MIN_ABS ( $ a  =  b $ , $a > 0$ , $b < 0$ )			MAX_ABS ( $ a  =  b $ , $a > 0$ , $b < 0$ )		
Src1	Src2	Result	Src1	Src2	Result
a	b	b	a	b	a
b	a	b	b	a	a

### Operation

```
RangeDP(SRC1[63:0], SRC2[63:0], CmpOpCtl[1:0], SignSelCtl[1:0])
```

```
{
    // Check if SNAN and report IE, see also Table 5-12
    IF (SRC1 = SNAN) THEN RETURN (QNaN(SRC1), set IE);
    IF (SRC2 = SNAN) THEN RETURN (QNaN(SRC2), set IE);

    Src1.exp ← SRC1[62:52];
    Src1.fraction ← SRC1[51:0];
    IF ((Src1.exp = 0) and (Src1.fraction != 0)) THEN// Src1 is a denormal number
        IF DAZ THEN Src1.fraction ← 0;
        ELSE IF (SRC2 <> QNaN) Set DE; FI;
    FI;

    Src2.exp ← SRC2[62:52];
    Src2.fraction ← SRC2[51:0];
```

```

IF ((Src2.exp = 0) and (Src2.fraction != 0 )) THEN// Src2 is a denormal number
  IF DAZ THEN Src2.fraction ← 0;
  ELSE IF (SRC1 <> QNAN) Set DE; FI;
FI;

IF (SRC2 = QNAN) THEN{TMP[63:0] ← SRC1[63:0]}
ELSE IF(SRC1 = QNAN) THEN{TMP[63:0] ← SRC2[63:0]}
ELSE IF (Both SRC1, SRC2 are magnitude-0 and opposite-signed) TMP[63:0] ← from Table 5-13
ELSE IF (Both SRC1, SRC2 are magnitude-equal and opposite-signed and CmpOpCtl[1:0] > 01) TMP[63:0] ← from Table 5-14
ELSE
  Case(CmpOpCtl[1:0])
  00: TMP[63:0] ← (SRC1[63:0] ≤ SRC2[63:0]) ? SRC1[63:0] : SRC2[63:0];
  01: TMP[63:0] ← (SRC1[63:0] ≤ SRC2[63:0]) ? SRC2[63:0] : SRC1[63:0];
  10: TMP[63:0] ← (ABS(SRC1[63:0]) ≤ ABS(SRC2[63:0])) ? SRC1[63:0] : SRC2[63:0];
  11: TMP[63:0] ← (ABS(SRC1[63:0]) ≤ ABS(SRC2[63:0])) ? SRC2[63:0] : SRC1[63:0];
  ESAC;
FI;

Case(SignSelCtl[1:0])
00: dest ← (SRC1[63] << 63) OR (TMP[62:0]);// Preserve Src1 sign bit
01: dest ← TMP[63:0];// Preserve sign of compare result
10: dest ← (0 << 63) OR (TMP[62:0]);// Zero out sign bit
11: dest ← (1 << 63) OR (TMP[62:0]);// Set the sign bit
ESAC;
RETURN dest[63:0];
}

```

```

CmpOpCtl[1:0]= imm8[1:0];
SignSelCtl[1:0]=imm8[3:2];

```

**VRANGEPD (EVEX encoded versions)**

(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) AND (SRC2 *is memory*)
      THEN DEST[i+63:i] ← RangeDP (SRC1[i+63:i], SRC2[63:0], CmpOpCtl[1:0], SignSelCtl[1:0]);
      ELSE DEST[i+63:i] ← RangeDP (SRC1[i+63:i], SRC2[i+63:i], DAZ, CmpOpCtl[1:0], SignSelCtl[1:0]);
    FI;
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+63:i] remains unchanged*
    ELSE ; zeroing-masking
      DEST[i+63:i] = 0
    FI;
  FI;
ENDFOR;
DEST[MAX_VL-1:VL] ← 0

```

The following example describes a common usage of this instruction for checking that the input operand is bound-  
ed between ±1023.

```
VRANGEPD zmm_dst, zmm_src, zmm_1023, 02h;
```

Where:

`zmm_dst` is the destination operand.

`zmm_src` is the input operand to compare against  $\pm 1023$  (this is SRC1).

`zmm_1023` is the reference operand, contains the value of 1023 (and this is SRC2).

`IMM=02(imm8[1:0]='10)` selects the Min Absolute value operation with selection of SRC1.sign.

In case  $|zmm\_src| < 1023$  (i.e. SRC1 is smaller than 1023 in magnitude), then its value will be written into `zmm_dst`. Otherwise, the value stored in `zmm_dst` will get the value of 1023 (received on `zmm_1023`, which is SRC2).

However, the sign control (`imm8[3:2]='00`) instructs to select the sign of SRC1 received from `zmm_src`. So, even in the case of  $|zmm\_src| \geq 1023$ , the selected sign of SRC1 is kept.

Thus, if `zmm_src < -1023`, the result of `VRANGE PD` will be the minimal value of -1023 while if `zmm_src > +1023`, the result of `VRANGE` will be the maximal value of +1023.

### Intel C/C++ Compiler Intrinsic Equivalent

```
VRANGE PD __m512d __mm512_range_pd (__m512d a, __m512d b, int imm);
VRANGE PD __m512d __mm512_range_round_pd (__m512d a, __m512d b, int imm, int sae);
VRANGE PD __m512d __mm512_mask_range_pd (__m512d s, __mmask8 k, __m512d a, __m512d b, int imm);
VRANGE PD __m512d __mm512_mask_range_round_pd (__m512d s, __mmask8 k, __m512d a, __m512d b, int imm, int sae);
VRANGE PD __m512d __mm512_maskz_range_pd (__mmask8 k, __m512d a, __m512d b, int imm);
VRANGE PD __m512d __mm512_maskz_range_round_pd (__mmask8 k, __m512d a, __m512d b, int imm, int sae);
VRANGE PD __m256d __mm256_range_pd (__m256d a, __m256d b, int imm);
VRANGE PD __m256d __mm256_mask_range_pd (__m256d s, __mmask8 k, __m256d a, __m256d b, int imm);
VRANGE PD __m256d __mm256_maskz_range_pd (__mmask8 k, __m256d a, __m256d b, int imm);
VRANGE PD __m128d __mm_range_pd (__m128d a, __m128d b, int imm);
VRANGE PD __m128d __mm_mask_range_pd (__m128d s, __mmask8 k, __m128d a, __m128d b, int imm);
VRANGE PD __m128d __mm_maskz_range_pd (__mmask8 k, __m128d a, __m128d b, int imm);
```

### SIMD Floating-Point Exceptions

Invalid, Denormal

### Other Exceptions

See Exceptions Type E2.

## VRANGEPS—Range Restriction Calculation For Packed Pairs of Float32 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.128.66.0F3A.W0 50 /r ib VRANGEPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst, imm8	FV	V/V	AVX512VL AVX512DQ	Calculate four RANGE operation output value from 4 pairs of single-precision floating-point values in xmm2 and xmm3/m128/m32bcst, store the results to xmm1 under the writemask k1. Imm8 specifies the comparison and sign of the range operation.
EVEX.NDS.256.66.0F3A.W0 50 /r ib VRANGEPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst, imm8	FV	V/V	AVX512VL AVX512DQ	Calculate eight RANGE operation output value from 8 pairs of single-precision floating-point values in ymm2 and ymm3/m256/m32bcst, store the results to ymm1 under the writemask k1. Imm8 specifies the comparison and sign of the range operation.
EVEX.NDS.512.66.0F3A.W0 50 /r ib VRANGEPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{sae}, imm8	FV	V/V	AVX512DQ	Calculate 16 RANGE operation output value from 16 pairs of single-precision floating-point values in zmm2 and zmm3/m512/m32bcst, store the results to zmm1 under the writemask k1. Imm8 specifies the comparison and sign of the range operation.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
FV	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	Imm8

### Description

This instruction calculates 4/8/16 range operation outputs from two sets of packed input single-precision FP values in the first source operand (the second operand) and the second source operand (the third operand). The range outputs are written to the destination operand (the first operand) under the writemask k1.

Bits7:4 of imm8 byte must be zero. The range operation output is performed in two parts, each configured by a two-bit control field within imm8[3:0]:

- Imm8[1:0] specifies the initial comparison operation to be one of max, min, max absolute value or min absolute value of the input value pair. Each comparison of two input values produces an intermediate result that combines with the sign selection control (Imm8[3:2]) to determine the final range operation output.
- Imm8[3:2] specifies the sign of the range operation output to be one of the following: from the first input value, from the comparison result, set or clear.

The encodings of Imm8[1:0] and Imm8[3:2] are shown in Figure 5-25.

When one or more of the input value is a NAN, the comparison operation may signal invalid exception (IE). Details with one of more input value is NAN is listed in Table 5-12. If the comparison raises an IE, the sign select control (Imm8[3:2]) has no effect to the range operation output, this is indicated also in Table 5-12.

When both input values are zeros of opposite signs, the comparison operation of MIN/MAX in the range compare operation is slightly different from the conceptually similar FP MIN/MAX operation that are found in the instructions VMAXPD/VMINPD. The details of MIN/MAX/MIN\_ABS/MAX\_ABS operation for VRANGEPS/PS/SD/SS for magnitude-0, opposite-signed input cases are listed in Table 5-13.

Additionally, non-zero, equal-magnitude with opposite-sign input values perform MIN\_ABS or MAX\_ABS comparison operation with result listed in Table 5-14.

### Operation

```
RangeSP(SRC1[31:0], SRC2[31:0], CmpOpCtl[1:0], SignSelCtl[1:0])
```

```
{
    // Check if SNAN and report IE, see also Table 5-12
    IF (SRC1=SNAN) THEN RETURN (QNAN(SRC1), set IE);
    IF (SRC2=SNAN) THEN RETURN (QNAN(SRC2), set IE);
```

```

Src1.exp ← SRC1[30:23];
Src1.fraction ← SRC1[22:0];
IF ((Src1.exp = 0 ) and (Src1.fraction != 0 )) THEN// Src1 is a denormal number
    IF DAZ THEN Src1.fraction ← 0;
    ELSE IF (SRC2 <> QNAN) Set DE; FI;
FI;
Src2.exp ← SRC2[30:23];
Src2.fraction ← SRC2[22:0];
IF ((Src2.exp = 0 ) and (Src2.fraction != 0 )) THEN// Src2 is a denormal number
    IF DAZ THEN Src2.fraction ← 0;
    ELSE IF (SRC1 <> QNAN) Set DE; FI;
FI;

IF (SRC2 = QNAN) THEN{TMP[31:0] ← SRC1[31:0]}
ELSE IF(SRC1 = QNAN) THEN{TMP[31:0] ← SRC2[31:0]}
ELSE IF (Both SRC1, SRC2 are magnitude-0 and opposite-signed) TMP[31:0] ← from Table 5-13
ELSE IF (Both SRC1, SRC2 are magnitude-equal and opposite-signed and CmpOpCtl[1:0] > 01) TMP[31:0] ← from Table 5-14
ELSE
    Case(CmpOpCtl[1:0])
    00: TMP[31:0] ← (SRC1[31:0] ≤ SRC2[31:0]) ? SRC1[31:0] : SRC2[31:0];
    01: TMP[31:0] ← (SRC1[31:0] ≤ SRC2[31:0]) ? SRC2[31:0] : SRC1[31:0];
    10: TMP[31:0] ← (ABS(SRC1[31:0]) ≤ ABS(SRC2[31:0])) ? SRC1[31:0] : SRC2[31:0];
    11: TMP[31:0] ← (ABS(SRC1[31:0]) ≤ ABS(SRC2[31:0])) ? SRC2[31:0] : SRC1[31:0];
    ESAC;
FI;
Case(SignSelCtl[1:0])
00: dest ← (SRC1[31] << 31) OR (TMP[30:0]);// Preserve Src1 sign bit
01: dest ← TMP[31:0];// Preserve sign of compare result
10: dest ← (0 << 31) OR (TMP[30:0]);// Zero out sign bit
11: dest ← (1 << 31) OR (TMP[30:0]);// Set the sign bit
ESAC;
RETURN dest[31:0];
}

CmpOpCtl[1:0]= imm8[1:0];
SignSelCtl[1:0]=imm8[3:2];

```

**VRANGEPS**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\* THEN

IF (EVEX.b == 1) AND (SRC2 \*is memory\*)

THEN DEST[i+31:i] ← RangeSP (SRC1[i+31:i], SRC2[31:0], CmpOpCtl[1:0], SignSelCtl[1:0]);

ELSE DEST[i+31:i] ← RangeSP (SRC1[i+31:i], SRC2[i+31:i], DAZ, CmpOpCtl[1:0], SignSelCtl[1:0]);

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+31:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+31:i] = 0

FI;

FI;

```
ENDFOR;
DEST[MAX_VL-1:VL] ← 0
```

The following example describes a common usage of this instruction for checking that the input operand is bounded between  $\pm 150$ .

```
VRANGEPS zmm_dst, zmm_src, zmm_150, 02h;
```

Where:

zmm\_dst is the destination operand.

zmm\_src is the input operand to compare against  $\pm 150$ .

zmm\_150 is the reference operand, contains the value of 150.

IMM=02(imm8[1:0]='10) selects the Min Absolute value operation with selection of src1.sign.

In case  $|zmm\_src| < 150$ , then its value will be written into zmm\_dst. Otherwise, the value stored in zmm\_dst will get the value of 150 (received on zmm\_150).

However, the sign control (imm8[3:2]='00) instructs to select the sign of SRC1 received from zmm\_src. So, even in the case of  $|zmm\_src| \geq 150$ , the selected sign of SRC1 is kept.

Thus, if  $zmm\_src < -150$ , the result of VRANGEPS will be the minimal value of -150 while if  $zmm\_src > +150$ , the result of VRANGE will be the maximal value of +150.

### Intel C/C++ Compiler Intrinsic Equivalent

```
VRANGEPS __m512 __mm512_range_ps (__m512 a, __m512 b, int imm);
VRANGEPS __m512 __mm512_range_round_ps (__m512 a, __m512 b, int imm, int sae);
VRANGEPS __m512 __mm512_mask_range_ps (__m512 s, __mmask16 k, __m512 a, __m512 b, int imm);
VRANGEPS __m512 __mm512_mask_range_round_ps (__m512 s, __mmask16 k, __m512 a, __m512 b, int imm, int sae);
VRANGEPS __m512 __mm512_maskz_range_ps (__mmask16 k, __m512 a, __m512 b, int imm);
VRANGEPS __m512 __mm512_maskz_range_round_ps (__mmask16 k, __m512 a, __m512 b, int imm, int sae);
VRANGEPS __m256 __mm256_range_ps (__m256 a, __m256 b, int imm);
VRANGEPS __m256 __mm256_mask_range_ps (__m256 s, __mmask8 k, __m256 a, __m256 b, int imm);
VRANGEPS __m256 __mm256_maskz_range_ps (__mmask8 k, __m256 a, __m256 b, int imm);
VRANGEPS __m128 __mm_range_ps (__m128 a, __m128 b, int imm);
VRANGEPS __m128 __mm_mask_range_ps (__m128 s, __mmask8 k, __m128 a, __m128 b, int imm);
VRANGEPS __m128 __mm_maskz_range_ps (__mmask8 k, __m128 a, __m128 b, int imm);
```

### SIMD Floating-Point Exceptions

Invalid, Denormal

### Other Exceptions

See Exceptions Type E2.



## VRANGESD—Range Restriction Calculation From a pair of Scalar Float64 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.LIG.66.0F3A.W1 51 /r VRANGESD xmm1 {k1}{z}, xmm2, xmm3/m64{sae}, imm8	T1S	V/V	AVX512DQ	Calculate a RANGE operation output value from 2 double-precision floating-point values in xmm2 and xmm3/m64, store the output to xmm1 under writemask. Imm8 specifies the comparison and sign of the range operation.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
T1S	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	Imm8

### Description

This instruction calculates a range operation output from two input double-precision FP values in the low qword element of the first source operand (the second operand) and second source operand (the third operand). The range output is written to the low qword element of the destination operand (the first operand) under the writemask k1.

Bits7:4 of imm8 byte must be zero. The range operation output is performed in two parts, each configured by a two-bit control field within imm8[3:0]:

- Imm8[1:0] specifies the initial comparison operation to be one of max, min, max absolute value or min absolute value of the input value pair. Each comparison of two input values produces an intermediate result that combines with the sign selection control (Imm8[3:2]) to determine the final range operation output.
- Imm8[3:2] specifies the sign of the range operation output to be one of the following: from the first input value, from the comparison result, set or clear.

The encodings of Imm8[1:0] and Imm8[3:2] are shown in Figure 5-25.

Bits 128:63 of the destination operand are copied from the respective element of the first source operand.

When one or more of the input value is a NAN, the comparison operation may signal invalid exception (IE). Details with one of more input value is NAN is listed in Table 5-12. If the comparison raises an IE, the sign select control (Imm8[3:2]) has no effect to the range operation output, this is indicated also in Table 5-12.

When both input values are zeros of opposite signs, the comparison operation of MIN/MAX in the range compare operation is slightly different from the conceptually similar FP MIN/MAX operation that are found in the instructions VMAXPD/VMINPD. The details of MIN/MAX/MIN\_ABS/MAX\_ABS operation for VRANGEPD/PS/SD/SS for magnitude-0, opposite-signed input cases are listed in Table 5-13.

Additionally, non-zero, equal-magnitude with opposite-sign input values perform MIN\_ABS or MAX\_ABS comparison operation with result listed in Table 5-14.

### Operation

RangeDP(SRC1[63:0], SRC2[63:0], CmpOpCtl[1:0], SignSelCtl[1:0])

```
{
  // Check if SNAN and report IE, see also Table 5-12
  IF (SRC1 = SNAN) THEN RETURN (QNAN(SRC1), set IE);
  IF (SRC2 = SNAN) THEN RETURN (QNAN(SRC2), set IE);

  Src1.exp ← SRC1[62:52];
  Src1.fraction ← SRC1[51:0];
  IF ((Src1.exp = 0) and (Src1.fraction != 0)) THEN// Src1 is a denormal number
    IF DAZ THEN Src1.fraction ← 0;
    ELSE IF (SRC2 <> QNAN) Set DE; FI;
  FI;
```

```

Src2.exp ← SRC2[62:52];
Src2.fraction ← SRC2[51:0];
IF ((Src2.exp = 0) and (Src2.fraction != 0 )) THEN// Src2 is a denormal number
    IF DAZ THEN Src2.fraction ← 0;
    ELSE IF (SRC1 <> QNAN) Set DE; FI;
FI;

IF (SRC2 = QNAN) THEN{TMP[63:0] ← SRC1[63:0]}
ELSE IF(SRC1 = QNAN) THEN{TMP[63:0] ← SRC2[63:0]}
ELSE IF (Both SRC1, SRC2 are magnitude-0 and opposite-signed) TMP[63:0] ← from Table 5-13
ELSE IF (Both SRC1, SRC2 are magnitude-equal and opposite-signed and CmpOpCtl[1:0] > 01) TMP[63:0] ← from Table 5-14
ELSE
    Case(CmpOpCtl[1:0])
    00: TMP[63:0] ← (SRC1[63:0] ≤ SRC2[63:0]) ? SRC1[63:0] : SRC2[63:0];
    01: TMP[63:0] ← (SRC1[63:0] ≤ SRC2[63:0]) ? SRC2[63:0] : SRC1[63:0];
    10: TMP[63:0] ← (ABS(SRC1[63:0]) ≤ ABS(SRC2[63:0])) ? SRC1[63:0] : SRC2[63:0];
    11: TMP[63:0] ← (ABS(SRC1[63:0]) ≤ ABS(SRC2[63:0])) ? SRC2[63:0] : SRC1[63:0];
    ESAC;
FI;

Case(SignSelCtl[1:0])
00: dest ← (SRC1[63] << 63) OR (TMP[62:0]);// Preserve Src1 sign bit
01: dest ← TMP[63:0];// Preserve sign of compare result
10: dest ← (0 << 63) OR (TMP[62:0]);// Zero out sign bit
11: dest ← (1 << 63) OR (TMP[62:0]);// Set the sign bit
ESAC;
RETURN dest[63:0];
}

CmpOpCtl[1:0]= imm8[1:0];
SignSelCtl[1:0]=imm8[3:2];

```

**VRANGESD**

```

IF k1[0] OR *no writemask*
    THEN DEST[63:0] ← RangeDP (SRC1[63:0], SRC2[63:0], CmpOpCtl[1:0], SignSelCtl[1:0]);
ELSE
    IF *merging-masking* ; merging-masking
        THEN *DEST[63:0] remains unchanged*
    ELSE ; zeroing-masking
        DEST[63:0] = 0
    FI;
FI;
DEST[127:64] ← SRC1[127:64]
DEST[MAX_VL-1:128] ← 0

```

The following example describes a common usage of this instruction for checking that the input operand is bound-  
ed between  $\pm 1023$ .

```
VRANGESD xmm_dst, xmm_src, xmm_1023, 02h;
```

Where:

xmm\_dst is the destination operand.

xmm\_src is the input operand to compare against  $\pm 1023$ .

xmm\_1023 is the reference operand, contains the value of 1023.

IMM=02(imm8[1:0]='10) selects the Min Absolute value operation with selection of src1.sign.

In case  $|xmm\_src| < 1023$ , then its value will be written into xmm\_dst. Otherwise, the value stored in xmm\_dst will get the value of 1023 (received on xmm\_1023).

However, the sign control (imm8[3:2]='00) instructs to select the sign of SRC1 received from xmm\_src. So, even in the case of  $|xmm\_src| \geq 1023$ , the selected sign of SRC1 is kept.

Thus, if  $xmm\_src < -1023$ , the result of VRANGEPD will be the minimal value of -1023 while if  $xmm\_src > +1023$ , the result of VRANGE will be the maximal value of +1023.

#### Intel C/C++ Compiler Intrinsic Equivalent

```
VRANGESD __m128d _mm_range_sd ( __m128d a, __m128d b, int imm);
```

```
VRANGESD __m128d _mm_range_round_sd ( __m128d a, __m128d b, int imm, int sae);
```

```
VRANGESD __m128d _mm_mask_range_sd ( __m128d s, __mmask8 k, __m128d a, __m128d b, int imm);
```

```
VRANGESD __m128d _mm_mask_range_round_sd ( __m128d s, __mmask8 k, __m128d a, __m128d b, int imm, int sae);
```

```
VRANGESD __m128d _mm_maskz_range_sd ( __mmask8 k, __m128d a, __m128d b, int imm);
```

```
VRANGESD __m128d _mm_maskz_range_round_sd ( __mmask8 k, __m128d a, __m128d b, int imm, int sae);
```

#### SIMD Floating-Point Exceptions

Invalid, Denormal

#### Other Exceptions

See Exceptions Type E3.

## VRANGESS—Range Restriction Calculation From a Pair of Scalar Float32 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.LIG.66.0F3A.W0 51 /r VRANGESS xmm1 {k1}{z}, xmm2, xmm3/m32{sae}, imm8	T1S	V/V	AVX512DQ	Calculate a RANGE operation output value from 2 single-precision floating-point values in xmm2 and xmm3/m32, store the output to xmm1 under writemask. Imm8 specifies the comparison and sign of the range operation.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
T1S	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

This instruction calculates a range operation output from two input single-precision FP values in the low dword element of the first source operand (the second operand) and second source operand (the third operand). The range output is written to the low dword element of the destination operand (the first operand) under the writemask k1.

Bits7:4 of imm8 byte must be zero. The range operation output is performed in two parts, each configured by a two-bit control field within imm8[3:0]:

- Imm8[1:0] specifies the initial comparison operation to be one of max, min, max absolute value or min absolute value of the input value pair. Each comparison of two input values produces an intermediate result that combines with the sign selection control (Imm8[3:2]) to determine the final range operation output.
- Imm8[3:2] specifies the sign of the range operation output to be one of the following: from the first input value, from the comparison result, set or clear.

The encodings of Imm8[1:0] and Imm8[3:2] are shown in Figure 5-25.

Bits 128:31 of the destination operand are copied from the respective elements of the first source operand.

When one or more of the input value is a NAN, the comparison operation may signal invalid exception (IE). Details with one of more input value is NAN is listed in Table 5-12. If the comparison raises an IE, the sign select control (Imm8[3:2]) has no effect to the range operation output, this is indicated also in Table 5-12.

When both input values are zeros of opposite signs, the comparison operation of MIN/MAX in the range compare operation is slightly different from the conceptually similar FP MIN/MAX operation that are found in the instructions VMAXPD/VMINPD. The details of MIN/MAX/MIN\_ABS/MAX\_ABS operation for VRANGEPD/PS/SD/SS for magnitude-0, opposite-signed input cases are listed in Table 5-13.

Additionally, non-zero, equal-magnitude with opposite-sign input values perform MIN\_ABS or MAX\_ABS comparison operation with result listed in Table 5-14.

### Operation

RangeSP(SRC1[31:0], SRC2[31:0], CmpOpCtl[1:0], SignSelCtl[1:0])

```
{
  // Check if SNAN and report IE, see also Table 5-12
  IF (SRC1=SNAN) THEN RETURN (QNAN(SRC1), set IE);
  IF (SRC2=SNAN) THEN RETURN (QNAN(SRC2), set IE);

  Src1.exp ← SRC1[30:23];
  Src1.fraction ← SRC1[22:0];
  IF ((Src1.exp = 0 ) and (Src1.fraction != 0 )) THEN// Src1 is a denormal number
    IF DAZ THEN Src1.fraction ← 0;
    ELSE IF (SRC2 <> QNAN) Set DE; FI;
  FI;
  Src2.exp ← SRC2[30:23];
```

```

Src2.fraction ← SRC2[22:0];
IF ((Src2.exp = 0 ) and (Src2.fraction != 0 )) THEN// Src2 is a denormal number
    IF DAZ THEN Src2.fraction ← 0;
    ELSE IF (SRC1 <> QNAN) Set DE; FI;
FI;

IF (SRC2 = QNAN) THEN{TMP[31:0] ← SRC1[31:0]}
ELSE IF(SRC1 = QNAN) THEN{TMP[31:0] ← SRC2[31:0]}
ELSE IF (Both SRC1, SRC2 are magnitude-0 and opposite-signed) TMP[31:0] ← from Table 5-13
ELSE IF (Both SRC1, SRC2 are magnitude-equal and opposite-signed and CmpOpCtl[1:0] > 01) TMP[31:0] ← from Table 5-14
ELSE
    Case(CmpOpCtl[1:0])
    00: TMP[31:0] ← (SRC1[31:0] ≤ SRC2[31:0]) ? SRC1[31:0] : SRC2[31:0];
    01: TMP[31:0] ← (SRC1[31:0] ≤ SRC2[31:0]) ? SRC2[31:0] : SRC1[31:0];
    10: TMP[31:0] ← (ABS(SRC1[31:0]) ≤ ABS(SRC2[31:0])) ? SRC1[31:0] : SRC2[31:0];
    11: TMP[31:0] ← (ABS(SRC1[31:0]) ≤ ABS(SRC2[31:0])) ? SRC2[31:0] : SRC1[31:0];
    ESAC;
FI;
Case(SignSelCtl[1:0])
00: dest ← (SRC1[31] << 31) OR (TMP[30:0]);// Preserve Src1 sign bit
01: dest ← TMP[31:0];// Preserve sign of compare result
10: dest ← (0 << 31) OR (TMP[30:0]);// Zero out sign bit
11: dest ← (1 << 31) OR (TMP[30:0]);// Set the sign bit
ESAC;
RETURN dest[31:0];
}

```

```

CmpOpCtl[1:0]= imm8[1:0];
SignSelCtl[1:0]=imm8[3:2];

```

### VRANGESS

```

IF k1[0] OR *no writemask*
    THEN DEST[31:0] ← RangeSP (SRC1[31:0], SRC2[31:0], CmpOpCtl[1:0], SignSelCtl[1:0]);
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE ; zeroing-masking
                DEST[31:0] = 0
        FI;
FI;
DEST[127:32] ← SRC1[127:32]
DEST[MAX_VL-1:128] ← 0

```

The following example describes a common usage of this instruction for checking that the input operand is bounded between  $\pm 150$ .

```
VRANGESS zmm_dst, zmm_src, zmm_150, 02h;
```

Where:

zmm\_dst is the destination operand.

zmm\_src is the input operand to compare against  $\pm 150$ .

zmm\_150 is the reference operand, contains the value of 150.

IMM=02(imm8[1:0]='10) selects the Min Absolute value operation with selection of src1.sign.

In case  $|xmm\_src| < 150$ , then its value will be written into `zmm_dst`. Otherwise, the value stored in `xmm_dst` will get the value of 150 (received on `zmm_150`).

However, the sign control (`imm8[3:2]='00`) instructs to select the sign of SRC1 received from `xmm_src`. So, even in the case of  $|xmm\_src| \geq 150$ , the selected sign of SRC1 is kept.

Thus, if `xmm_src`  $< -150$ , the result of `VRANGESS` will be the minimal value of -150 while if `xmm_src`  $> +150$ , the result of `VRANGE` will be the maximal value of +150.

#### Intel C/C++ Compiler Intrinsic Equivalent

```
VRANGESS __m128 _mm_range_ss (__m128 a, __m128 b, int imm);
VRANGESS __m128 _mm_range_round_ss (__m128 a, __m128 b, int imm, int sae);
VRANGESS __m128 _mm_mask_range_ss (__m128 s, __mmask8 k, __m128 a, __m128 b, int imm);
VRANGESS __m128 _mm_mask_range_round_ss (__m128 s, __mmask8 k, __m128 a, __m128 b, int imm, int sae);
VRANGESS __m128 _mm_maskz_range_ss (__mmask8 k, __m128 a, __m128 b, int imm);
VRANGESS __m128 _mm_maskz_range_round_ss (__mmask8 k, __m128 a, __m128 b, int imm, int sae);
```

#### SIMD Floating-Point Exceptions

Invalid, Denormal

#### Other Exceptions

See Exceptions Type E3.

## VRCP14PD—Compute Approximate Reciprocals of Packed Float64 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W1 4C /r VRCP14PD xmm1 {k1}{z}, xmm2/m128/m64bcst	FV	V/V	AVX512VL AVX512F	Computes the approximate reciprocals of the packed double-precision floating-point values in xmm2/m128/m64bcst and stores the results in xmm1. Under writemask.
EVEX.256.66.0F38.W1 4C /r VRCP14PD ymm1 {k1}{z}, ymm2/m256/m64bcst	FV	V/V	AVX512VL AVX512F	Computes the approximate reciprocals of the packed double-precision floating-point values in ymm2/m256/m64bcst and stores the results in ymm1. Under writemask.
EVEX.512.66.0F38.W1 4C /r VRCP14PD zmm1 {k1}{z}, zmm2/m512/m64bcst	FV	V/V	AVX512F	Computes the approximate reciprocals of the packed double-precision floating-point values in zmm2/m512/m64bcst and stores the results in zmm1. Under writemask.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
FV	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

This instruction performs a SIMD computation of the approximate reciprocals of eight/four/two packed double-precision floating-point values in the source operand (the second operand) and stores the packed double-precision floating-point results in the destination operand. The maximum relative error for this approximation is less than  $2^{-14}$ .

The source operand can be a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM register conditionally updated according to the writemask.

The VRCP14PD instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an  $\infty$  with the sign of the source value is returned. A denormal source value will be treated as zero only in case of DAZ bit set in MXCSR. Otherwise it is treated correctly (i.e. not as a 0.0). Underflow results are flushed to zero only in case of FTZ bit set in MXCSR. Otherwise it will be treated correctly (i.e. correct underflow result is written) with the sign of the operand. When a source value is a SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned.

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

MXCSR exception flags are not affected by this instruction and floating-point exceptions are not reported.

**Table 5-15. VRCP14PD/VRCP14SD Special Cases**

Input value	Result value	Comments
$0 \leq X \leq 2^{-1024}$	INF	Very small denormal
$-2^{-1024} \leq X \leq -0$	-INF	Very small denormal
$X > 2^{1022}$	Underflow	Up to 18 bits of fractions are returned*
$X < -2^{1022}$	-Underflow	Up to 18 bits of fractions are returned*
$X = 2^{-n}$	$2^n$	
$X = -2^{-n}$	$-2^n$	

\* in this case the mantissa is shifted right by one or two bits

A numerically exact implementation of VRCP14xx can be found at <https://software.intel.com/en-us/articles/reference-implementations-for-1A-approximation-instructions-vrcp14-vrsqrt14-vrcp28-vrsqrt28-vexp2>.

**Operation****VRCP14PD ((EVEX encoded versions)**

(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) AND (SRC *is memory*)
      THEN DEST[i+63:i] ← APPROXIMATE(1.0/SRC[63:0]);
      ELSE DEST[i+63:i] ← APPROXIMATE(1.0/SRC[i+63:i]);
    FI;
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+63:i] remains unchanged*
    ELSE ; zeroing-masking
      DEST[i+63:i] ← 0
    FI;
  FI;
ENDFOR;
DEST[MAX_VL-1:VL] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

VRCP14PD \_\_m512d \_\_mm512\_rcp14\_pd( \_\_m512d a);

VRCP14PD \_\_m512d \_\_mm512\_mask\_rcp14\_pd( \_\_m512d s, \_\_mmask8 k, \_\_m512d a);

VRCP14PD \_\_m512d \_\_mm512\_maskz\_rcp14\_pd( \_\_mmask8 k, \_\_m512d a);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type E4.



## VRCP14SD—Compute Approximate Reciprocal of Scalar Float64 Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.LIG.66.0F38.W1 4D /r VRCP14SD xmm1 {k1}{z}, xmm2, xmm3/m64	T1S	V/V	AVX512F	Computes the approximate reciprocal of the scalar double-precision floating-point value in xmm3/m64 and stores the result in xmm1 using writemask k1. Also, upper double-precision floating-point value (bits[127:64]) from xmm2 is copied to xmm1[127:64].

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
T1S	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

This instruction performs a SIMD computation of the approximate reciprocal of the low double-precision floating-point value in the second source operand (the third operand) stores the result in the low quadword element of the destination operand (the first operand) according to the writemask k1. Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand (the second operand). The maximum relative error for this approximation is less than  $2^{-14}$ . The source operand can be an XMM register or a 64-bit memory location. The destination operand is an XMM register.

The VRCP14SD instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an  $\infty$  with the sign of the source value is returned. A denormal source value will be treated as zero only in case of DAZ bit set in MXCSR. Otherwise it is treated correctly (i.e. not as a 0.0). Underflow results are flushed to zero only in case of FTZ bit set in MXCSR. Otherwise it will be treated correctly (i.e. correct underflow result is written) with the sign of the operand. When a source value is a SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned. See Table 5-15 for special-case input values.

MXCSR exception flags are not affected by this instruction and floating-point exceptions are not reported.

A numerically exact implementation of VRCP14xx can be found at <https://software.intel.com/en-us/articles/reference-implementations-for-IA-approximation-instructions-vcrcp14-vrsqrt14-vcrcp28-vrsqrt28-vexp2>.

### Operation

#### VRCP14SD (EVEX version)

```
IF k1[0] OR *no writemask*
    THEN DEST[63:0] ← APPROXIMATE(1.0/SRC2[63:0]);
ELSE
    IF *merging-masking*           ; merging-masking
        THEN *DEST[63:0] remains unchanged*
    ELSE                             ; zeroing-masking
        DEST[63:0] ← 0
FI;
FI;
DEST[127:64] ← SRC1[127:64]
DEST[MAX_VL-1:128] ← 0
```

### Intel C/C++ Compiler Intrinsic Equivalent

```
VRCP14SD __m128d __mm_rcp14_sd( __m128d a, __m128d b);
VRCP14SD __m128d __mm_mask_rcp14_sd( __m128d s, __mmask8 k, __m128d a, __m128d b);
VRCP14SD __m128d __mm_maskz_rcp14_sd( __mmask8 k, __m128d a, __m128d b);
```

### SIMD Floating-Point Exceptions

None



**Other Exceptions**

See Exceptions Type E5.

## VRCP14PS—Compute Approximate Reciprocals of Packed Float32 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 4C /r VRCP14PS xmm1 {k1}{z}, xmm2/m128/m32bcst	FV	V/V	AVX512VL AVX512F	Computes the approximate reciprocals of the packed single-precision floating-point values in xmm2/m128/m32bcst and stores the results in xmm1. Under writemask.
EVEX.256.66.0F38.W0 4C /r VRCP14PS ymm1 {k1}{z}, ymm2/m256/m32bcst	FV	V/V	AVX512VL AVX512F	Computes the approximate reciprocals of the packed single-precision floating-point values in ymm2/m256/m32bcst and stores the results in ymm1. Under writemask.
EVEX.512.66.0F38.W0 4C /r VRCP14PS zmm1 {k1}{z}, zmm2/m512/m32bcst	FV	V/V	AVX512F	Computes the approximate reciprocals of the packed single-precision floating-point values in zmm2/m512/m32bcst and stores the results in zmm1. Under writemask.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
FV	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

This instruction performs a SIMD computation of the approximate reciprocals of the packed single-precision floating-point values in the source operand (the second operand) and stores the packed single-precision floating-point results in the destination operand (the first operand). The maximum relative error for this approximation is less than  $2^{-14}$ .

The source operand can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM register conditionally updated according to the writemask.

The VRCP14PS instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an  $\infty$  with the sign of the source value is returned. A denormal source value will be treated as zero only in case of DAZ bit set in MXCSR. Otherwise it is treated correctly (i.e. not as a 0.0). Underflow results are flushed to zero only in case of FTZ bit set in MXCSR. Otherwise it will be treated correctly (i.e. correct underflow result is written) with the sign of the operand. When a source value is a SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned.

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

MXCSR exception flags are not affected by this instruction and floating-point exceptions are not reported.

**Table 5-16. VRCP14PS/VRCP14SS Special Cases**

Input value	Result value	Comments
$0 \leq X \leq 2^{-128}$	INF	Very small denormal
$-2^{-128} \leq X \leq -0$	-INF	Very small denormal
$X > 2^{126}$	Underflow	Up to 18 bits of fractions are returned*
$X < -2^{126}$	-Underflow	Up to 18 bits of fractions are returned*
$X = 2^{-n}$	$2^n$	
$X = -2^{-n}$	$-2^n$	

\* in this case the mantissa is shifted right by one or two bits

A numerically exact implementation of VRCP14xx can be found at <https://software.intel.com/en-us/articles/reference-implementations-for-1A-approximation-instructions-vcrcp14-vrsqrt14-vcrcp28-vrsqrt28-vexp2>.

**Operation****VRCP14PS (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\* THEN

IF (EVEX.b = 1) AND (SRC \*is memory\*)

THEN DEST[i+31:i] ← APPROXIMATE(1.0/SRC[31:0]);

ELSE DEST[i+31:i] ← APPROXIMATE(1.0/SRC[i+31:i]);

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+31:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+31:i] ← 0

FI;

FI;

ENDFOR;

DEST[MAX\_VL-1:VL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

VRCP14PS \_\_m512 \_\_mm512\_rcp14\_ps( \_\_m512 a);

VRCP14PS \_\_m512 \_\_mm512\_mask\_rcp14\_ps( \_\_m512 s, \_\_mmask16 k, \_\_m512 a);

VRCP14PS \_\_m512 \_\_mm512\_maskz\_rcp14\_ps( \_\_mmask16 k, \_\_m512 a);

VRCP14PS \_\_m256 \_\_mm256\_rcp14\_ps( \_\_m256 a);

VRCP14PS \_\_m256 \_\_mm512\_mask\_rcp14\_ps( \_\_m256 s, \_\_mmask8 k, \_\_m256 a);

VRCP14PS \_\_m256 \_\_mm512\_maskz\_rcp14\_ps( \_\_mmask8 k, \_\_m256 a);

VRCP14PS \_\_m128 \_\_mm\_rcp14\_ps( \_\_m128 a);

VRCP14PS \_\_m128 \_\_mm\_mask\_rcp14\_ps( \_\_m128 s, \_\_mmask8 k, \_\_m128 a);

VRCP14PS \_\_m128 \_\_mm\_maskz\_rcp14\_ps( \_\_mmask8 k, \_\_m128 a);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type E4.

## VRCP14SS—Compute Approximate Reciprocal of Scalar Float32 Value

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.LIG.66.0F38.W0 4D /r VRCP14SS xmm1 {k1}{z}, xmm2, xmm3/m32	T1S	V/V	AVX512F	Computes the approximate reciprocal of the scalar single-precision floating-point value in xmm3/m32 and stores the results in xmm1 using writemask k1. Also, upper double-precision floating-point value (bits[127:32]) from xmm2 is copied to xmm1[127:32].

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
T1S	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

This instruction performs a SIMD computation of the approximate reciprocal of the low single-precision floating-point value in the second source operand (the third operand) and stores the result in the low quadword element of the destination operand (the first operand) according to the writemask k1. Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand (the second operand). The maximum relative error for this approximation is less than  $2^{-14}$ . The source operand can be an XMM register or a 32-bit memory location. The destination operand is an XMM register.

The VRCP14SS instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an  $\infty$  with the sign of the source value is returned. A denormal source value will be treated as zero only in case of DAZ bit set in MXCSR. Otherwise it is treated correctly (i.e. not as a 0.0). Underflow results are flushed to zero only in case of FTZ bit set in MXCSR. Otherwise it will be treated correctly (i.e. correct underflow result is written) with the sign of the operand. When a source value is a SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned. See Table 5-16 for special-case input values.

MXCSR exception flags are not affected by this instruction and floating-point exceptions are not reported.

A numerically exact implementation of VRCP14xx can be found at <https://software.intel.com/en-us/articles/reference-implementations-for-IA-approximation-instructions-vrcp14-vrsqrt14-vrcp28-vrsqrt28-vexp2>.

### Operation

#### VRCP14SS (EVEX version)

```
IF k1[0] OR *no writemask*
    THEN DEST[31:0] ← APPROXIMATE(1.0/SRC2[31:0]);
ELSE
    IF *merging-masking*           ; merging-masking
        THEN *DEST[31:0] remains unchanged*
    ELSE                             ; zeroing-masking
        DEST[31:0] ← 0
FI;
FI;
DEST[127:32] ← SRC1[127:32]
DEST[MAX_VL-1:128] ← 0
```

#### Intel C/C++ Compiler Intrinsic Equivalent

```
VRCP14SS __m128 __mm_rcp14_ss(__m128 a, __m128 b);
VRCP14SS __m128 __mm_mask_rcp14_ss(__m128 s, __mmask8 k, __m128 a, __m128 b);
VRCP14SS __m128 __mm_maskz_rcp14_ss(__mmask8 k, __m128 a, __m128 b);
```

#### SIMD Floating-Point Exceptions

None



**Other Exceptions**

See Exceptions Type E5.

## VRCP28PD—Approximation to the Reciprocal of Packed Double-Precision Floating-Point Values with Less Than $2^{-28}$ Relative Error

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.512.66.0F38.W1 CA /r VRCP28PD zmm1 {k1}{z}, zmm2/m512/m64bcst {sae}	FV	V/V	AVX512ER	Computes the approximate reciprocals ( $< 2^{-28}$ relative error) of the packed double-precision floating-point values in zmm2/m512/m64bcst and stores the results in zmm1. Under writemask.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
FV	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Computes the reciprocal approximation of the float64 values in the source operand (the second operand) and store the results to the destination operand (the first operand). The approximate reciprocal is evaluated with less than  $2^{-28}$  of maximum relative error.

Denormal input values are treated as zeros and do not signal #DE, irrespective of MXCSR.DAZ. Denormal results are flushed to zeros and do not signal #UE, irrespective of MXCSR.FZ.

If any source element is NaN, the quietized NaN source value is returned for that element. If any source element is  $\pm\infty$ ,  $\pm 0.0$  is returned for that element. Also, if any source element is  $\pm 0.0$ ,  $\pm\infty$  is returned for that element.

The source operand is a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

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

A numerically exact implementation of VRCP28xx can be found at <https://software.intel.com/en-us/articles/reference-implementations-for-IA-approximation-instructions-vrcp14-vrsqrt14-vrcp28-vrsqrt28-vexp2>.

### Operation

#### VRCP28PD (EVEX encoded versions)

(KL, VL) = (8, 512)

FOR j  $\leftarrow$  0 TO KL-1

  i  $\leftarrow$  j \* 64

  IF k1[j] OR \*no writemask\* THEN

    IF (EVEX.b = 1) AND (SRC \*is memory\*)

      THEN DEST[i+63:i]  $\leftarrow$  RCP\_28\_DP(1.0/SRC[63:0]);

      ELSE DEST[i+63:i]  $\leftarrow$  RCP\_28\_DP(1.0/SRC[i+63:i]);

    FI;

  ELSE

    IF \*merging-masking\* ; merging-masking

      THEN \*DEST[i+63:i] remains unchanged\*

    ELSE ; zeroing-masking

      DEST[i+63:i]  $\leftarrow$  0

    FI;

  FI;

ENDFOR;

**Table 5-17. VRCP28PD Special Cases**

Input value	Result value	Comments
NAN	QNAN(input)	If (SRC = SNaN) then #I
$0 \leq X < 2^{-1022}$	INF	Positive input denormal or zero; #Z
$-2^{-1022} < X \leq -0$	-INF	Negative input denormal or zero; #Z
$X > 2^{1022}$	+0.0f	
$X < -2^{1022}$	-0.0f	
$X = +\infty$	+0.0f	
$X = -\infty$	-0.0f	
$X = 2^{-n}$	$2^n$	Exact result (unless input/output is a denormal)
$X = -2^{-n}$	$-2^n$	Exact result (unless input/output is a denormal)

**Intel C/C++ Compiler Intrinsic Equivalent**

```
VRCP28PD __m512d _mm512_rcp28_round_pd( __m512d a, int sae);
VRCP28PD __m512d _mm512_mask_rcp28_round_pd(__m512d a, __mmask8 m, __m512d b, int sae);
VRCP28PD __m512d _mm512_maskz_rcp28_round_pd( __mmask8 m, __m512d b, int sae);
```

**SIMD Floating-Point Exceptions**

Invalid (if SNaN input), Divide-by-zero

**Other Exceptions**

See Exceptions Type E2.



## VRCP28SD—Approximation to the Reciprocal of Scalar Double-Precision Floating-Point Value with Less Than $2^{-28}$ Relative Error

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.LIG.66.0F38.W1 CB /r VRCP28SD xmm1 {k1}{z}, xmm2, xmm3/m64 {sae}	T1S	V/V	AVX512ER	Computes the approximate reciprocal ( $< 2^{-28}$ relative error) of the scalar double-precision floating-point value in xmm3/m64 and stores the results in xmm1. Under writemask. Also, upper double-precision floating-point value (bits[127:64]) from xmm2 is copied to xmm1[127:64].

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
T1S	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

### Description

Computes the reciprocal approximation of the low float64 value in the second source operand (the third operand) and store the result to the destination operand (the first operand). The approximate reciprocal is evaluated with less than  $2^{-28}$  of maximum relative error. The result is written into the low float64 element of the destination operand according to the writemask k1. Bits 127:64 of the destination is copied from the corresponding bits of the first source operand (the second operand).

A denormal input value is treated as zero and does not signal #DE, irrespective of MXCSR.DAZ. A denormal result is flushed to zero and does not signal #UE, irrespective of MXCSR.FZ.

If any source element is NaN, the quietized NaN source value is returned for that element. If any source element is  $\pm\infty$ ,  $\pm 0.0$  is returned for that element. Also, if any source element is  $\pm 0.0$ ,  $\pm\infty$  is returned for that element.

The first source operand is an XMM register. The second source operand is an XMM register or a 64-bit memory location. The destination operand is a XMM register, conditionally updated using writemask k1.

A numerically exact implementation of VRCP28xx can be found at <https://software.intel.com/en-us/articles/reference-implementations-for-IA-approximation-instructions-vrcp14-vrsqrt14-vrcp28-vrsqrt28-vexp2>.

### Operation

#### VRCP28SD ((EVEX encoded versions)

```

IF k1[0] OR *no writemask* THEN
    DEST[63: 0] ← RCP_28_DP(1.0/SRC2[63: 0]);
ELSE
    IF *merging-masking*           ; merging-masking
        THEN *DEST[63: 0] remains unchanged*
    ELSE                             ; zeroing-masking
        DEST[63: 0] ← 0
    FI;
FI;
ENDFOR;
DEST[127:64] ← SRC1[127: 64]
DEST[MAX_VL-1:128] ← 0

```

Table 5-18. VRCP28SD Special Cases

Input value	Result value	Comments
NAN	QNAN(input)	If (SRC = SNaN) then #I
$0 \leq X < 2^{-1022}$	INF	Positive input denormal or zero; #Z
$-2^{-1022} < X \leq -0$	-INF	Negative input denormal or zero; #Z
$X > 2^{1022}$	+0.0f	
$X < -2^{1022}$	-0.0f	
$X = +\infty$	+0.0f	
$X = -\infty$	-0.0f	
$X = 2^{-n}$	$2^n$	Exact result (unless input/output is a denormal)
$X = -2^{-n}$	$-2^n$	Exact result (unless input/output is a denormal)

**Intel C/C++ Compiler Intrinsic Equivalent**

VRCP28SD \_\_m128d \_\_mm\_rcp28\_round\_sd (\_\_m128d a, \_\_m128d b, int sae);

VRCP28SD \_\_m128d \_\_mm\_mask\_rcp28\_round\_sd(\_\_m128d s, \_\_mmask8 m, \_\_m128d a, \_\_m128d b, int sae);

VRCP28SD \_\_m128d \_\_mm\_maskz\_rcp28\_round\_sd(\_\_mmask8 m, \_\_m128d a, \_\_m128d b, int sae);

**SIMD Floating-Point Exceptions**

Invalid (if SNaN input), Divide-by-zero

**Other Exceptions**

See Exceptions Type E3.

## VRCP28PS—Approximation to the Reciprocal of Packed Single-Precision Floating-Point Values with Less Than $2^{-28}$ Relative Error

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.512.66.0F38.W0 CA /r VRCP28PS zmm1 {k1}{z}, zmm2/m512/m32bcst {sae}	FV	V/V	AVX512ER	Computes the approximate reciprocals ( $< 2^{-28}$ relative error) of the packed single-precision floating-point values in zmm2/m512/m32bcst and stores the results in zmm1. Under writemask.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
FV	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Computes the reciprocal approximation of the float32 values in the source operand (the second operand) and store the results to the destination operand (the first operand) using the writemask k1. The approximate reciprocal is evaluated with less than  $2^{-28}$  of maximum relative error prior to final rounding. The final results are rounded to  $< 2^{-23}$  relative error before written to the destination.

Denormal input values are treated as zeros and do not signal #DE, irrespective of MXCSR.DAZ. Denormal results are flushed to zeros and do not signal #UE, irrespective of MXCSR.FZ.

If any source element is NaN, the quietized NaN source value is returned for that element. If any source element is  $\pm\infty$ ,  $\pm 0.0$  is returned for that element. Also, if any source element is  $\pm 0.0$ ,  $\pm\infty$  is returned for that element.

The source operand is a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

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

A numerically exact implementation of VRCP28xx can be found at <https://software.intel.com/en-us/articles/reference-implementations-for-IA-approximation-instructions-vrcp14-vrsqrt14-vrcp28-vrsqrt28-vexp2>.

### Operation

#### VRCP28PS (EVEX encoded versions)

(KL, VL) = (16, 512)

FOR j  $\leftarrow$  0 TO KL-1

  i  $\leftarrow$  j \* 32

  IF k1[j] OR \*no writemask\* THEN

    IF (EVEX.b = 1) AND (SRC \*is memory\*)

      THEN DEST[i+31:i]  $\leftarrow$  RCP\_28\_SP(1.0/SRC[31:0]);

      ELSE DEST[i+31:i]  $\leftarrow$  RCP\_28\_SP(1.0/SRC[i+31:i]);

    FI;

  ELSE

    IF \*merging-masking\* ; merging-masking

      THEN \*DEST[i+31:i] remains unchanged\*

    ELSE ; zeroing-masking

      DEST[i+31:i]  $\leftarrow$  0

    FI;

  FI;

ENDFOR;

Table 5-19. VRCP28PS Special Cases

Input value	Result value	Comments
NAN	QNAN(input)	If (SRC = SNaN) then #I
$0 \leq X < 2^{-126}$	INF	Positive input denormal or zero; #Z
$-2^{-126} < X \leq -0$	-INF	Negative input denormal or zero; #Z
$X > 2^{126}$	+0.0f	
$X < -2^{126}$	-0.0f	
$X = +\infty$	+0.0f	
$X = -\infty$	-0.0f	
$X = 2^{-n}$	$2^n$	Exact result (unless input/output is a denormal)
$X = -2^{-n}$	$-2^n$	Exact result (unless input/output is a denormal)

**Intel C/C++ Compiler Intrinsic Equivalent**

VRCP28PS \_\_mm512\_rcp28\_round\_ps (\_\_m512 a, int sae);

VRCP28PS \_\_m512 \_\_mm512\_mask\_rcp28\_round\_ps(\_\_m512 s, \_\_mmask16 m, \_\_m512 a, int sae);

VRCP28PS \_\_m512 \_\_mm512\_maskz\_rcp28\_round\_ps(\_\_mmask16 m, \_\_m512 a, int sae);

**SIMD Floating-Point Exceptions**

Invalid (if SNaN input), Divide-by-zero

**Other Exceptions**

See Exceptions Type E2.

## VRCP28SS—Approximation to the Reciprocal of Scalar Single-Precision Floating-Point Value with Less Than $2^{-28}$ Relative Error

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.LIG.66.0F38.WO CB /r VRCP28SS xmm1 {k1}{z}, xmm2, xmm3/m32 {sae}	T1S	V/V	AVX512ER	Computes the approximate reciprocal ( $< 2^{-28}$ relative error) of the scalar single-precision floating-point value in xmm3/m32 and stores the results in xmm1. Under writemask. Also, upper 3 single-precision floating-point values (bits[127:32]) from xmm2 is copied to xmm1[127:32].

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
T1S	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

### Description

Computes the reciprocal approximation of the low float32 value in the second source operand (the third operand) and store the result to the destination operand (the first operand). The approximate reciprocal is evaluated with less than  $2^{-28}$  of maximum relative error prior to final rounding. The final result is rounded to  $< 2^{-23}$  relative error before written into the low float32 element of the destination according to writemask k1. Bits 127:32 of the destination is copied from the corresponding bits of the first source operand (the second operand).

A denormal input value is treated as zero and does not signal #DE, irrespective of MXCSR.DAZ. A denormal result is flushed to zero and does not signal #UE, irrespective of MXCSR.FZ.

If any source element is NaN, the quietized NaN source value is returned for that element. If any source element is  $\pm\infty$ ,  $\pm 0.0$  is returned for that element. Also, if any source element is  $\pm 0.0$ ,  $\pm\infty$  is returned for that element.

The first source operand is an XMM register. The second source operand is an XMM register or a 32-bit memory location. The destination operand is a XMM register, conditionally updated using writemask k1.

A numerically exact implementation of VRCP28xx can be found at <https://software.intel.com/en-us/articles/reference-implementations-for-IA-approximation-instructions-vrcp14-vrsqrt14-vrcp28-vrsqrt28-vexp2>.

### Operation

#### VRCP28SS ((EVEX encoded versions)

```

IF k1[0] OR *no writemask* THEN
    DEST[31: 0] ← RCP_28_SP(1.0/SRC2[31: 0]);
ELSE
    IF *merging-masking*           ; merging-masking
        THEN *DEST[31: 0] remains unchanged*
    ELSE                             ; zeroing-masking
        DEST[31: 0] ← 0
    FI;
FI;
ENDFOR;
DEST[127:32] ← SRC1[127: 32]
DEST[MAX_VL-1:128] ← 0

```

**Table 5-20. VRCP28SS Special Cases**

Input value	Result value	Comments
NAN	QNAN(input)	If (SRC = SNaN) then #I
$0 \leq X < 2^{-126}$	INF	Positive input denormal or zero; #Z
$-2^{-126} < X \leq -0$	-INF	Negative input denormal or zero; #Z
$X > 2^{126}$	+0.0f	
$X < -2^{126}$	-0.0f	
$X = +\infty$	+0.0f	
$X = -\infty$	-0.0f	
$X = 2^{-n}$	$2^n$	Exact result (unless input/output is a denormal)
$X = -2^{-n}$	$-2^n$	Exact result (unless input/output is a denormal)

**Intel C/C++ Compiler Intrinsic Equivalent**

```
VRCP28SS __m128_mm_rcp28_round_ss ( __m128 a, __m128 b, int sae);
VRCP28SS __m128_mm_mask_rcp28_round_ss(__m128 s, __mmask8 m, __m128 a, __m128 b, int sae);
VRCP28SS __m128_mm_maskz_rcp28_round_ss(__mmask8 m, __m128 a, __m128 b, int sae);
```

**SIMD Floating-Point Exceptions**

Invalid (if SNaN input), Divide-by-zero

**Other Exceptions**

See Exceptions Type E3.

## VREDUCEPD—Perform Reduction Transformation on Packed Float64 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F3A.W1 56 /r ib VREDUCEPD xmm1 {k1}{z}, xmm2/m128/m64bcst, imm8	FV	V/V	AVX512VL AVX512DQ	Perform reduction transformation on packed double-precision floating point values in xmm2/m128/m32bcst by subtracting a number of fraction bits specified by the imm8 field. Stores the result in xmm1 register under writemask k1.
EVEX.256.66.0F3A.W1 56 /r ib VREDUCEPD ymm1 {k1}{z}, ymm2/m256/m64bcst, imm8	FV	V/V	AVX512VL AVX512DQ	Perform reduction transformation on packed double-precision floating point values in ymm2/m256/m32bcst by subtracting a number of fraction bits specified by the imm8 field. Stores the result in ymm1 register under writemask k1.
EVEX.512.66.0F3A.W1 56 /r ib VREDUCEPD zmm1 {k1}{z}, zmm2/m512/m64bcst{sae}, imm8	FV	V/V	AVX512DQ	Perform reduction transformation on double-precision floating point values in zmm2/m512/m32bcst by subtracting a number of fraction bits specified by the imm8 field. Stores the result in zmm1 register under writemask k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
FV	ModRM:reg (w)	ModRM:r/m (r)	Imm8	NA

### Description

Perform reduction transformation of the packed binary encoded double-precision FP values in the source operand (the second operand) and store the reduced results in binary FP format to the destination operand (the first operand) under the writemask k1.

The reduction transformation subtracts the integer part and the leading M fractional bits from the binary FP source value, where M is a unsigned integer specified by imm8[7:4], see Figure 5-26. Specifically, the reduction transformation can be expressed as:

$$\text{dest} = \text{src} - (\text{ROUND}(2^M * \text{src})) * 2^{-M};$$

where "Round()" treats "src", "2<sup>M</sup>", and their product as binary FP numbers with normalized significand and biased exponents.

The magnitude of the reduced result can be expressed by considering  $\text{src} = 2^p * \text{man}_2$ , where 'man<sub>2</sub>' is the normalized significand and 'p' is the unbiased exponent

Then if RC = RNE:  $0 \leq |\text{Reduced Result}| \leq 2^{p-M-1}$

Then if RC ≠ RNE:  $0 \leq |\text{Reduced Result}| < 2^{p-M}$

This instruction might end up with a precision exception set. However, in case of SPE set (i.e. Suppress Precision Exception, which is imm8[3]=1), no precision exception is reported.

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

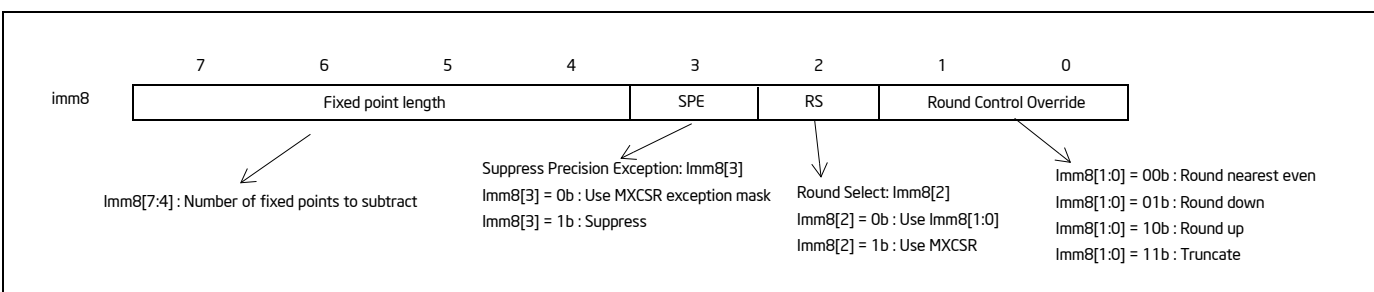


Figure 5-26. Imm8 Controls for VREDUCEPD/SD/PS/SS

Handling of special case of input values are listed in Table 5-21.

**Table 5-21. VREDUCEPD/SD/PS/SS Special Cases**

	Round Mode	Returned value
$ Src1  < 2^{-M-1}$	RNE	Src1
$ Src1  < 2^{-M}$	RPI, Src1 > 0	Round (Src1-2 <sup>-M</sup> ) *
	RPI, Src1 ≤ 0	Src1
	RNI, Src1 ≥ 0	Src1
	RNI, Src1 < 0	Round (Src1+2 <sup>-M</sup> ) *
Src1 = ±0, or Dest = ±0 (Src1!=INF)	NOT RNI	+0.0
	RNI	-0.0
Src1 = ±INF	any	+0.0
Src1 = ±NAN	n/a	QNaN(Src1)

\* Round control = (imm8.MS1)? MXCSR.RC: imm8.RC

### Operation

ReduceArgumentDP(SRC[63:0], imm8[7:0])

```
{
  // Check for NaN
  IF (SRC [63:0] = NAN) THEN
    RETURN (Convert SRC[63:0] to QNaN); FI;
  M ← imm8[7:4]; // Number of fraction bits of the normalized significand to be subtracted
  RC ← imm8[1:0]; // Round Control for ROUND() operation
  RC source ← imm[2];
  SPE ← 0; // Suppress Precision Exception
  TMP[63:0] ← 2-M * {ROUND(2M*SRC[63:0], SPE, RC_source, RC)}; // ROUND() treats SRC and 2M as standard binary FP values
  TMP[63:0] ← SRC[63:0] - TMP[63:0]; // subtraction under the same RC,SPE controls
  RETURN TMP[63:0]; // binary encoded FP with biased exponent and normalized significand
}
```

### VREDUCEPD

(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) AND (SRC \*is memory\*)

      THEN DEST[i+63:i] ← ReduceArgumentDP(SRC[63:0], imm8[7:0]);

      ELSE DEST[i+63:i] ← ReduceArgumentDP(SRC[i+63:i], imm8[7:0]);

    FI;

  ELSE

    IF \*merging-masking\* ; merging-masking

      THEN \*DEST[i+63:i] remains unchanged\*

    ELSE ; zeroing-masking

      DEST[i+63:i] = 0

    FI;

  FI;

ENDFOR;

DEST[MAX\_VL-1:VL] ← 0



**Intel C/C++ Compiler Intrinsic Equivalent**

VREDUCEPD \_\_m512d\_mm512\_mask\_reduce\_pd(\_\_m512d a, int imm, int sae)  
 VREDUCEPD \_\_m512d\_mm512\_mask\_reduce\_pd(\_\_m512d s, \_\_mmask8 k, \_\_m512d a, int imm, int sae)  
 VREDUCEPD \_\_m512d\_mm512\_maskz\_reduce\_pd(\_\_mmask8 k, \_\_m512d a, int imm, int sae)  
 VREDUCEPD \_\_m256d\_mm256\_mask\_reduce\_pd(\_\_m256d a, int imm)  
 VREDUCEPD \_\_m256d\_mm256\_mask\_reduce\_pd(\_\_m256d s, \_\_mmask8 k, \_\_m256d a, int imm)  
 VREDUCEPD \_\_m256d\_mm256\_maskz\_reduce\_pd(\_\_mmask8 k, \_\_m256d a, int imm)  
 VREDUCEPD \_\_m128d\_mm\_mask\_reduce\_pd(\_\_m128d a, int imm)  
 VREDUCEPD \_\_m128d\_mm\_mask\_reduce\_pd(\_\_m128d s, \_\_mmask8 k, \_\_m128d a, int imm)  
 VREDUCEPD \_\_m128d\_mm\_maskz\_reduce\_pd(\_\_mmask8 k, \_\_m128d a, int imm)

**SIMD Floating-Point Exceptions**

Invalid, Precision

If SPE is enabled, precision exception is not reported (regardless of MXCSR exception mask).

**Other Exceptions**

See Exceptions Type E2, additionally

#UD                      If EVEX.vvvv != 1111B.

## VREDUCESD—Perform a Reduction Transformation on a Scalar Float64 Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.LIG.66.0F3A.W1 57 /r VREDUCESD xmm1 {k1}{z}, xmm2, xmm3/m64{sae}, imm8	T1S	V/V	AVX512D Q	Perform a reduction transformation on a scalar double-precision floating point value in xmm3/m64 by subtracting a number of fraction bits specified by the imm8 field. Also, upper double precision floating-point value (bits[127:64]) from xmm2 are copied to xmm1[127:64]. Stores the result in xmm1 register.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
T1S	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Perform a reduction transformation of the binary encoded double-precision FP value in the low qword element of the second source operand (the third operand) and store the reduced result in binary FP format to the low qword element of the destination operand (the first operand) under the writemask k1. Bits 127:64 of the destination operand are copied from respective qword elements of the first source operand (the second operand).

The reduction transformation subtracts the integer part and the leading M fractional bits from the binary FP source value, where M is a unsigned integer specified by imm8[7:4], see Figure 5-26. Specifically, the reduction transformation can be expressed as:

$$\text{dest} = \text{src} - (\text{ROUND}(2^M * \text{src})) * 2^{-M};$$

where "Round()" treats "src", "2<sup>M</sup>", and their product as binary FP numbers with normalized significand and biased exponents.

The magnitude of the reduced result can be expressed by considering  $\text{src} = 2^p * \text{man}_2$ ,

where 'man<sub>2</sub>' is the normalized significand and 'p' is the unbiased exponent

Then if RC = RNE:  $0 \leq |\text{Reduced Result}| \leq 2^{p-M-1}$

Then if RC ≠ RNE:  $0 \leq |\text{Reduced Result}| < 2^{p-M}$

This instruction might end up with a precision exception set. However, in case of SPE set (i.e. Suppress Precision Exception, which is imm8[3]=1), no precision exception is reported.

The operation is write masked.

Handling of special case of input values are listed in Table 5-21.

### Operation

ReduceArgumentDP(SRC[63:0], imm8[7:0])

```
{
    // Check for NaN
    IF (SRC [63:0] = NAN) THEN
        RETURN (Convert SRC[63:0] to QNaN); FI;
    M ← imm8[7:4]; // Number of fraction bits of the normalized significand to be subtracted
    RC ← imm8[1:0]; // Round Control for ROUND() operation
    RC_source ← imm[2];
    SPE ← 0; // Suppress Precision Exception
    TMP[63:0] ← 2-M * {ROUND(2M*SRC[63:0], SPE, RC_source, RC)}; // ROUND() treats SRC and 2M as standard binary FP values
    TMP[63:0] ← SRC[63:0] - TMP[63:0]; // subtraction under the same RC,SPE controls
    RETURN TMP[63:0]; // binary encoded FP with biased exponent and normalized significand
}
```

**VREDUCESD**

```

IF k1[0] or *no writemask*
  THEN DEST[63:0] ← ReduceArgumentDP(SRC2[63:0], imm8[7:0])
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[63:0] remains unchanged*
    ELSE ; zeroing-masking
      THEN DEST[63:0] = 0
  FI;
FI;
DEST[127:64] ← SRC1[127:64]
DEST[MAX_VL-1:128] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VREDUCESD __m128d _mm_mask_reduce_sd( __m128d a, __m128d b, int imm, int sae)
VREDUCESD __m128d _mm_mask_reduce_sd(__m128d s, __mmask16 k, __m128d a, __m128d b, int imm, int sae)
VREDUCESD __m128d _mm_maskz_reduce_sd(__mmask16 k, __m128d a, __m128d b, int imm, int sae)

```

**SIMD Floating-Point Exceptions**

Invalid, Precision

If SPE is enabled, precision exception is not reported (regardless of MXCSR exception mask).

**Other Exceptions**

See Exceptions Type E3.

## VREDUCEPS—Perform Reduction Transformation on Packed Float32 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F3A.W0 56 /r ib VREDUCEPS xmm1 {k1}{z}, xmm2/m128/m32bcst, imm8	FV	V/V	AVX512VL AVX512DQ	Perform reduction transformation on packed single-precision floating point values in xmm2/m128/m32bcst by subtracting a number of fraction bits specified by the imm8 field. Stores the result in xmm1 register under writemask k1.
EVEX.256.66.0F3A.W0 56 /r ib VREDUCEPS ymm1 {k1}{z}, ymm2/m256/m32bcst, imm8	FV	V/V	AVX512VL AVX512DQ	Perform reduction transformation on packed single-precision floating point values in ymm2/m256/m32bcst by subtracting a number of fraction bits specified by the imm8 field. Stores the result in ymm1 register under writemask k1.
EVEX.512.66.0F3A.W0 56 /r ib VREDUCEPS zmm1 {k1}{z}, zmm2/m512/m32bcst{sae}, imm8	FV	V/V	AVX512DQ	Perform reduction transformation on packed single-precision floating point values in zmm2/m512/m32bcst by subtracting a number of fraction bits specified by the imm8 field. Stores the result in zmm1 register under writemask k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
FV	ModRM:reg (w)	ModRM:r/m (r)	Imm8	NA

### Description

Perform reduction transformation of the packed binary encoded single-precision FP values in the source operand (the second operand) and store the reduced results in binary FP format to the destination operand (the first operand) under the writemask k1.

The reduction transformation subtracts the integer part and the leading M fractional bits from the binary FP source value, where M is a unsigned integer specified by imm8[7:4], see Figure 5-26. Specifically, the reduction transformation can be expressed as:

$$\text{dest} = \text{src} - (\text{ROUND}(2^M * \text{src})) * 2^{-M};$$

where "Round()" treats "src", "2<sup>M</sup>", and their product as binary FP numbers with normalized significand and biased exponents.

The magnitude of the reduced result can be expressed by considering  $\text{src} = 2^p * \text{man}_2$ , where 'man<sub>2</sub>' is the normalized significand and 'p' is the unbiased exponent

Then if RC = RNE:  $0 \leq |\text{Reduced Result}| \leq 2^{p-M-1}$

Then if RC ≠ RNE:  $0 \leq |\text{Reduced Result}| < 2^{p-M}$

This instruction might end up with a precision exception set. However, in case of SPE set (i.e. Suppress Precision Exception, which is imm8[3]=1), no precision exception is reported.

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

Handling of special case of input values are listed in Table 5-21.

### Operation

ReduceArgumentSP(SRC[31:0], imm8[7:0])

```
{
  // Check for NaN
  IF (SRC [31:0] = NAN) THEN
    RETURN (Convert SRC[31:0] to QNaN); FI
  M ← imm8[7:4]; // Number of fraction bits of the normalized significand to be subtracted
  RC ← imm8[1:0]; // Round Control for ROUND() operation
  RC source ← imm[2];
```

```

SPE ← 0; // Suppress Precision Exception
TMP[31:0] ← 2-M * {ROUND(2M * SRC[31:0], SPE, RC_source, RC)}; // ROUND() treats SRC and 2M as standard binary FP values
TMP[31:0] ← SRC[31:0] - TMP[31:0]; // subtraction under the same RC, SPE controls
RETURN TMP[31:0]; // binary encoded FP with biased exponent and normalized significand
}

```

**VREDUCEPS**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\* THEN

IF (EVEX.b == 1) AND (SRC \*is memory\*)

THEN DEST[i+31:i] ← ReduceArgumentSP(SRC[31:0], imm8[7:0]);

ELSE DEST[i+31:i] ← ReduceArgumentSP(SRC[i+31:i], imm8[7:0]);

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+31:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+31:i] = 0

FI;

FI;

ENDFOR;

DEST[MAX\_VL-1:VL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

VREDUCEPS \_\_m512 \_\_mm512\_mask\_reduce\_ps( \_\_m512 a, int imm, int sae)

VREDUCEPS \_\_m512 \_\_mm512\_mask\_reduce\_ps(\_\_m512 s, \_\_mmask16 k, \_\_m512 a, int imm, int sae)

VREDUCEPS \_\_m512 \_\_mm512\_maskz\_reduce\_ps(\_\_mmask16 k, \_\_m512 a, int imm, int sae)

VREDUCEPS \_\_m256 \_\_mm256\_mask\_reduce\_ps( \_\_m256 a, int imm)

VREDUCEPS \_\_m256 \_\_mm256\_mask\_reduce\_ps(\_\_m256 s, \_\_mmask8 k, \_\_m256 a, int imm)

VREDUCEPS \_\_m256 \_\_mm256\_maskz\_reduce\_ps(\_\_mmask8 k, \_\_m256 a, int imm)

VREDUCEPS \_\_m128 \_\_mm\_mask\_reduce\_ps( \_\_m128 a, int imm)

VREDUCEPS \_\_m128 \_\_mm\_mask\_reduce\_ps(\_\_m128 s, \_\_mmask8 k, \_\_m128 a, int imm)

VREDUCEPS \_\_m128 \_\_mm\_maskz\_reduce\_ps(\_\_mmask8 k, \_\_m128 a, int imm)

**SIMD Floating-Point Exceptions**

Invalid, Precision

If SPE is enabled, precision exception is not reported (regardless of MXCSR exception mask).

**Other Exceptions**

See Exceptions Type E2, additionally

#UD If EVEX.vvvv != 1111B.

## VREDUCESS—Perform a Reduction Transformation on a Scalar Float32 Value

Opcode/ Instruction	Op/ En	64/32 bitMode Support	CPUID Feature Flag	Description
EVEX.NDS.LIG.66.0F3A.W0 57 /r /ib VREDUCESS xmm1 {k1}{z}, xmm2, xmm3/m32{sae}, imm8	T1S	V/V	AVX512DQ	Perform a reduction transformation on a scalar single-precision floating point value in xmm3/m32 by subtracting a number of fraction bits specified by the imm8 field. Also, upper single precision floating-point values (bits[127:32]) from xmm2 are copied to xmm1[127:32]. Stores the result in xmm1 register.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
T1S	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Perform a reduction transformation of the binary encoded single-precision FP value in the low dword element of the second source operand (the third operand) and store the reduced result in binary FP format to the low dword element of the destination operand (the first operand) under the writemask k1. Bits 127:32 of the destination operand are copied from respective dword elements of the first source operand (the second operand).

The reduction transformation subtracts the integer part and the leading M fractional bits from the binary FP source value, where M is a unsigned integer specified by imm8[7:4], see Figure 5-26. Specifically, the reduction transformation can be expressed as:

$$\text{dest} = \text{src} - (\text{ROUND}(2^M * \text{src})) * 2^{-M};$$

where "Round()" treats "src", "2<sup>M</sup>", and their product as binary FP numbers with normalized significand and biased exponents.

The magnitude of the reduced result can be expressed by considering  $\text{src} = 2^p * \text{man}_2$ ,

where 'man<sub>2</sub>' is the normalized significand and 'p' is the unbiased exponent

Then if RC = RNE:  $0 \leq |\text{Reduced Result}| \leq 2^{p-M-1}$

Then if RC ≠ RNE:  $0 \leq |\text{Reduced Result}| < 2^{p-M}$

This instruction might end up with a precision exception set. However, in case of SPE set (i.e. Suppress Precision Exception, which is imm8[3]=1), no precision exception is reported.

Handling of special case of input values are listed in Table 5-21.

### Operation

ReduceArgumentSP(SRC[31:0], imm8[7:0])

```
{
  // Check for NaN
  IF (SRC [31:0] = NAN) THEN
    RETURN (Convert SRC[31:0] to QNaN); FI
  M ← imm8[7:4]; // Number of fraction bits of the normalized significand to be subtracted
  RC ← imm8[1:0]; // Round Control for ROUND() operation
  RC source ← imm[2];
  SPE ← 0; // Suppress Precision Exception
  TMP[31:0] ← 2-M * {ROUND(2M*SRC[31:0], SPE, RC_source, RC)}; // ROUND() treats SRC and 2M as standard binary FP values
  TMP[31:0] ← SRC[31:0] - TMP[31:0]; // subtraction under the same RC,SPE controls
  RETURN TMP[31:0]; // binary encoded FP with biased exponent and normalized significand
}
```

**VREDUCESS**

```

IF k1[0] or *no writemask*
  THEN  DEST[31:0] ← ReduceArgumentSP(SRC2[31:0], imm8[7:0])
  ELSE
    IF *merging-masking*           ; merging-masking
      THEN *DEST[31:0] remains unchanged*
    ELSE                             ; zeroing-masking
      THEN DEST[31:0] = 0
  FI;
FI;
DEST[127:32] ← SRC1[127:32]
DEST[MAX_VL-1:128] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VREDUCESS __m128 _mm_mask_reduce_ss( __m128 a, __m128 b, int imm, int sae)
VREDUCESS __m128 _mm_mask_reduce_ss(__m128 s, __mmask16 k, __m128 a, __m128 b, int imm, int sae)
VREDUCESS __m128 _mm_maskz_reduce_ss(__mmask16 k, __m128 a, __m128 b, int imm, int sae)

```

**SIMD Floating-Point Exceptions**

Invalid, Precision

If SPE is enabled, precision exception is not reported (regardless of MXCSR exception mask).

**Other Exceptions**

See Exceptions Type E3.

## VRNDSCALEPD—Round Packed Float64 Values To Include A Given Number Of Fraction Bits

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F3A.W1 09 /r ib VRNDSCALEPD xmm1 {k1}{z}, xmm2/m128/m64bcst, imm8	FV	V/V	AVX512VL AVX512F	Rounds packed double-precision floating point values in xmm2/m128/m64bcst to a number of fraction bits specified by the imm8 field. Stores the result in xmm1 register. Under writemask.
EVEX.256.66.0F3A.W1 09 /r ib VRNDSCALEPD ymm1 {k1}{z}, ymm2/m256/m64bcst, imm8	FV	V/V	AVX512VL AVX512F	Rounds packed double-precision floating point values in ymm2/m256/m64bcst to a number of fraction bits specified by the imm8 field. Stores the result in ymm1 register. Under writemask.
EVEX.512.66.0F3A.W1 09 /r ib VRNDSCALEPD zmm1 {k1}{z}, zmm2/m512/m64bcst{sae}, imm8	FV	V/V	AVX512F	Rounds packed double-precision floating-point values in zmm2/m512/m64bcst to a number of fraction bits specified by the imm8 field. Stores the result in zmm1 register using writemask k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
FV	ModRM:reg (w)	ModRM:r/m (r)	Imm8	NA

### Description

Round the double-precision floating-point values in the source operand by the rounding mode specified in the immediate operand (see Figure 5-27) 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 64-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 double-precision floating-point value.

It should be noticed that no overflow is induced while executing this instruction (although the source is scaled by the imm8[7:4] value).

The immediate operand also specifies control fields for the rounding operation, three bit fields are defined and shown in the "Immediate Control Description" figure below. Bit 3 of the immediate byte controls the processor behavior for a precision exception, bit 2 selects the source of rounding mode control. Bits 1:0 specify a non-sticky rounding-mode value (Immediate control table below lists the encoded values for rounding-mode field).

The Precision Floating-Point Exception is signaled according to the immediate operand. If any source operand is an SNaN then it will be converted to a QNaN. If DAZ is set to `1 then 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 VRNDSCALEPD is

$$\begin{aligned} \text{ROUND}(x) &= 2^{-M} * \text{Round\_to\_INT}(x * 2^M, \text{round\_ctrl}), \\ \text{round\_ctrl} &= \text{imm}[3:0]; \\ M &= \text{imm}[7:4]; \end{aligned}$$

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

VRNDSCALEPD is a more general form of the VEX-encoded VROUNDPD instruction. In VROUNDPD, the formula of the operation on each element is

$$\begin{aligned} \text{ROUND}(x) &= \text{Round\_to\_INT}(x, \text{round\_ctrl}), \\ \text{round\_ctrl} &= \text{imm}[3:0]; \end{aligned}$$



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

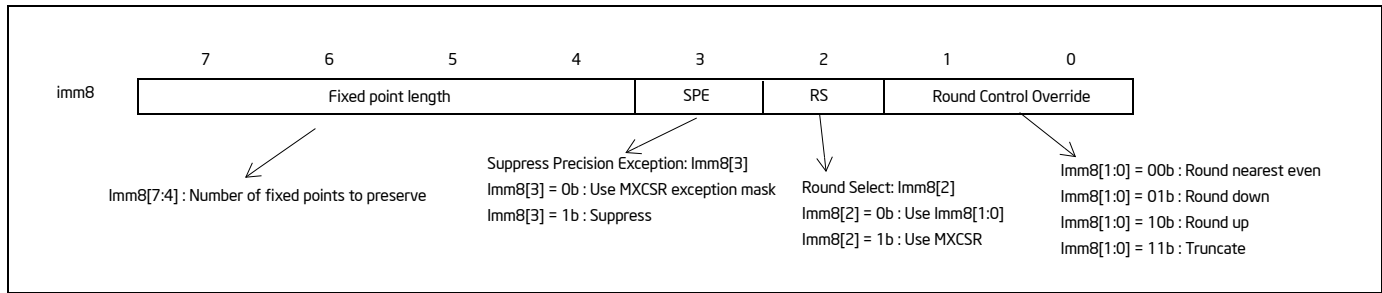


Figure 5-27. Immm8 Controls for VRNDSCALEPD/SD/PS/SS

Handling of special case of input values are listed in Table 5-22.

Table 5-22. VRNDSCALEPD/SD/PS/SS Special Cases

	Returned value
<b>Src1=±inf</b>	Src1
<b>Src1=±NAN</b>	Src1 converted to QNAN
<b>Src1=±0</b>	Src1

### Operation

```

RoundToIntegerDP(SRC[63:0], imm8[7:0]) {
    if (imm8[2] = 1)
        rounding_direction ← MXCSR:RC    ; get round control from MXCSR
    else
        rounding_direction ← imm8[1:0]    ; get round control from imm8[1:0]
    FI
    M ← imm8[7:4]                        ; get the scaling factor

    case (rounding_direction)
    00: TMP[63:0] ← round_to_nearest_even_integer(2M*SRC[63:0])
    01: TMP[63:0] ← round_to_equal_or_smaller_integer(2M*SRC[63:0])
    10: TMP[63:0] ← round_to_equal_or_larger_integer(2M*SRC[63:0])
    11: TMP[63:0] ← round_to_nearest_smallest_magnitude_integer(2M*SRC[63:0])
    ESAC

    Dest[63:0] ← 2-M* TMP[63:0]          ; scale down back to 2-M

    if (imm8[3] = 0) Then ; check SPE
        if (SRC[63:0] != Dest[63:0]) Then ; check precision lost
            set_precision() ; set #PE
        FI;
    FI;
    return(Dest[63:0])
}

```

**VRNDSCALEPD (EVEX encoded versions)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

IF \*src is a memory operand\*

THEN TMP\_SRC ← BROADCAST64(SRC, VL, k1)

ELSE TMP\_SRC ← SRC

FI;

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN DEST[i+63:i] ← RoundToIntegerDP((TMP\_SRC[j+63:i], imm8[7:0])

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+63:i] ← 0

FI;

FI;

ENDFOR;

DEST[MAX\_VL-1:VL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

VRNDSCALEPD \_\_m512d \_\_mm512\_roundscale\_pd( \_\_m512d a, int imm);

VRNDSCALEPD \_\_m512d \_\_mm512\_roundscale\_round\_pd( \_\_m512d a, int imm, int sae);

VRNDSCALEPD \_\_m512d \_\_mm512\_mask\_roundscale\_pd(\_\_m512d s, \_\_mmask8 k, \_\_m512d a, int imm);

VRNDSCALEPD \_\_m512d \_\_mm512\_mask\_roundscale\_round\_pd(\_\_m512d s, \_\_mmask8 k, \_\_m512d a, int imm, int sae);

VRNDSCALEPD \_\_m512d \_\_mm512\_maskz\_roundscale\_pd( \_\_mmask8 k, \_\_m512d a, int imm);

VRNDSCALEPD \_\_m512d \_\_mm512\_maskz\_roundscale\_round\_pd( \_\_mmask8 k, \_\_m512d a, int imm, int sae);

VRNDSCALEPD \_\_m256d \_\_mm256\_roundscale\_pd( \_\_m256d a, int imm);

VRNDSCALEPD \_\_m256d \_\_mm256\_mask\_roundscale\_pd(\_\_m256d s, \_\_mmask8 k, \_\_m256d a, int imm);

VRNDSCALEPD \_\_m256d \_\_mm256\_maskz\_roundscale\_pd( \_\_mmask8 k, \_\_m256d a, int imm);

VRNDSCALEPD \_\_m128d \_\_mm\_roundscale\_pd( \_\_m128d a, int imm);

VRNDSCALEPD \_\_m128d \_\_mm\_mask\_roundscale\_pd(\_\_m128d s, \_\_mmask8 k, \_\_m128d a, int imm);

VRNDSCALEPD \_\_m128d \_\_mm\_maskz\_roundscale\_pd( \_\_mmask8 k, \_\_m128d a, int imm);

**SIMD Floating-Point Exceptions**

Invalid, Precision

If SPE is enabled, precision exception is not reported (regardless of MXCSR exception mask).

**Other Exceptions**

See Exceptions Type E2.

## VRNDSCALESD—Round Scalar Float64 Value To Include A Given Number Of Fraction Bits

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.LIG.66.0F3A.W1 0B /r ib VRNDSCALESD xmm1 {k1}{z}, xmm2, xmm3/m64{sae}, imm8	T1S	V/V	AVX512F	Rounds scalar double-precision floating-point value in xmm3/m64 to a number of fraction bits specified by the imm8 field. Stores the result in xmm1 register.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
T1S	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	Imm8

### Description

Rounds a double-precision floating-point value in the low quadword (see Figure 5-27) element the second source operand (the third operand) by the rounding mode specified in the immediate operand and places the result in the corresponding element of the destination operand (the third operand) according to the writemask. The quadword element at bits 127:64 of the destination is copied from the first source operand (the second operand).

The destination and first source operands are XMM registers, the 2nd source operand can be an XMM register or memory location. Bits MAX\_VL-1:128 of the destination register are cleared.

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 double-precision floating-point value.

It should be noticed that no overflow is induced while executing this instruction (although the source is scaled by the imm8[7:4] value).

The immediate operand also specifies control fields for the rounding operation, three bit fields are defined and shown in the "Immediate Control Description" figure below. Bit 3 of the immediate byte controls the processor behavior for a precision exception, bit 2 selects the source of rounding mode control. Bits 1:0 specify a non-sticky rounding-mode value (Immediate control table below lists the encoded values for rounding-mode field).

The Precision Floating-Point Exception is signaled according to the immediate operand. If any source operand is an SNaN then it will be converted to a QNaN. If DAZ is set to '1 then 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 for VRNDSCALESD is

$$\begin{aligned} \text{ROUND}(x) &= 2^{-M} * \text{Round\_to\_INT}(x * 2^M, \text{round\_ctrl}), \\ \text{round\_ctrl} &= \text{imm}[3:0]; \\ M &= \text{imm}[7:4]; \end{aligned}$$

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

VRNDSCALESD is a more general form of the VEX-encoded VROUNDSD instruction. In VROUNDSD, the formula of the operation is

$$\begin{aligned} \text{ROUND}(x) &= \text{Round\_to\_INT}(x, \text{round\_ctrl}), \\ \text{round\_ctrl} &= \text{imm}[3:0]; \end{aligned}$$

EVEX encoded version: The source operand is a XMM register or a 64-bit memory location. The destination operand is a XMM register.

Handling of special case of input values are listed in Table 5-22.

**Operation**

```

RoundToIntegerDP(SRC[63:0], imm8[7:0]) {
  if (imm8[2] = 1)
    rounding_direction ← MXCSR:RC ; get round control from MXCSR
  else
    rounding_direction ← imm8[1:0] ; get round control from imm8[1:0]
  FI
  M ← imm8[7:4] ; get the scaling factor

  case (rounding_direction)
  00: TMP[63:0] ← round_to_nearest_even_integer(2M*SRC[63:0])
  01: TMP[63:0] ← round_to_equal_or_smaller_integer(2M*SRC[63:0])
  10: TMP[63:0] ← round_to_equal_or_larger_integer(2M*SRC[63:0])
  11: TMP[63:0] ← round_to_nearest_smallest_magnitude_integer(2M*SRC[63:0])
  ESAC

  Dest[63:0] ← 2-M* TMP[63:0] ; scale down back to 2-M

  if (imm8[3] = 0) Then ; check SPE
    if (SRC[63:0] != Dest[63:0]) Then ; check precision lost
      set_precision() ; set #PE
    FI;
  FI;
  return(Dest[63:0])
}

```

VRNDSCALESD (EVEX encoded version)

```

IF k1[0] or *no writemask*
  THEN DEST[63:0] ← RoundToIntegerDP(SRC2[63:0], Zero_upper_imm[7:0])
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[63:0] remains unchanged*
    ELSE ; zeroing-masking
      THEN DEST[63:0] ← 0
    FI;
  FI;
DEST[127:64] ← SRC1[127:64]
DEST[MAX_VL-1:128] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VRNDSCALESD __m128d __mm_roundscale_sd ( __m128d a, __m128d b, int imm);
VRNDSCALESD __m128d __mm_roundscale_round_sd ( __m128d a, __m128d b, int imm, int sae);
VRNDSCALESD __m128d __mm_mask_roundscale_sd ( __m128d s, __mmask8 k, __m128d a, __m128d b, int imm);
VRNDSCALESD __m128d __mm_mask_roundscale_round_sd ( __m128d s, __mmask8 k, __m128d a, __m128d b, int imm, int sae);
VRNDSCALESD __m128d __mm_maskz_roundscale_sd ( __mmask8 k, __m128d a, __m128d b, int imm);
VRNDSCALESD __m128d __mm_maskz_roundscale_round_sd ( __mmask8 k, __m128d a, __m128d b, int imm, int sae);

```

**SIMD Floating-Point Exceptions**

Invalid, Precision

If SPE is enabled, precision exception is not reported (regardless of MXCSR exception mask).

**Other Exceptions**

See Exceptions Type E3.

## VRNDSCALEPS—Round Packed Float32 Values To Include A Given Number Of Fraction Bits

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F3A.W0 08 /r ib VRNDSCALEPS xmm1 {k1}{z}, xmm2/m128/m32bcst, imm8	FV	V/V	AVX512VL AVX512F	Rounds packed single-precision floating point values in xmm2/m128/m32bcst to a number of fraction bits specified by the imm8 field. Stores the result in xmm1 register. Under writemask.
EVEX.256.66.0F3A.W0 08 /r ib VRNDSCALEPS ymm1 {k1}{z}, ymm2/m256/m32bcst, imm8	FV	V/V	AVX512VL AVX512F	Rounds packed single-precision floating point values in ymm2/m256/m32bcst to a number of fraction bits specified by the imm8 field. Stores the result in ymm1 register. Under writemask.
EVEX.512.66.0F3A.W0 08 /r ib VRNDSCALEPS zmm1 {k1}{z}, zmm2/m512/m32bcst{sae}, imm8	FV	V/V	AVX512F	Rounds packed single-precision floating-point values in zmm2/m512/m32bcst to a number of fraction bits specified by the imm8 field. Stores the result in zmm1 register using writemask.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
FV	ModRM:reg (w)	ModRM:r/m (r)	Imm8	NA

### Description

Round the single-precision floating-point values in the source operand by the rounding mode specified in the immediate operand (see Figure 5-27) and places the result in the destination operand.

The destination operand (the first operand) is a ZMM register conditionally updated according to the writemask. The source operand (the second operand) can be a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 32-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 single-precision floating-point value.

It should be noticed that no overflow is induced while executing this instruction (although the source is scaled by the imm8[7:4] value).

The immediate operand also specifies control fields for the rounding operation, three bit fields are defined and shown in the "Immediate Control Description" figure below. Bit 3 of the immediate byte controls the processor behavior for a precision exception, bit 2 selects the source of rounding mode control. Bits 1:0 specify a non-sticky rounding-mode value (Immediate control table below lists the encoded values for rounding-mode field).

The Precision Floating-Point Exception is signaled according to the immediate operand. If any source operand is an SNaN then it will be converted to a QNaN. If DAZ is set to `1 then 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 VRNDSCALEPS is

$$\text{ROUND}(x) = 2^{-M} * \text{Round\_to\_INT}(x * 2^M, \text{round\_ctrl}),$$

$$\text{round\_ctrl} = \text{imm}[3:0];$$

$$M = \text{imm}[7:4];$$

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

VRNDSCALEPS is a more general form of the VEX-encoded VROUNDPS instruction. In VROUNDPS, the formula of the operation on each element is

$$\text{ROUND}(x) = \text{Round\_to\_INT}(x, \text{round\_ctrl}),$$

$$\text{round\_ctrl} = \text{imm}[3:0];$$

Note: EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.  
Handling of special case of input values are listed in Table 5-22.

### Operation

```

RoundToIntegerSP(SRC[31:0], imm8[7:0]) {
  if (imm8[2] = 1)
    rounding_direction ← MXCSR.RC      ; get round control from MXCSR
  else
    rounding_direction ← imm8[1:0]     ; get round control from imm8[1:0]
  FI
  M ← imm8[7:4]                       ; get the scaling factor

  case (rounding_direction)
  00: TMP[31:0] ← round_to_nearest_even_integer(2M*SRC[31:0])
  01: TMP[31:0] ← round_to_equal_or_smaller_integer(2M*SRC[31:0])
  10: TMP[31:0] ← round_to_equal_or_larger_integer(2M*SRC[31:0])
  11: TMP[31:0] ← round_to_nearest_smallest_magnitude_integer(2M*SRC[31:0])
  ESAC;

  Dest[31:0] ← 2-M* TMP[31:0]         ; scale down back to 2-M
  if (imm8[3] = 0) Then                ; check SPE
    if (SRC[31:0] != Dest[31:0]) Then   ; check precision lost
      set_precision()                  ; set #PE
    FI;
  FI;
  return(Dest[31:0])
}

```

VRNDSCALEPS (EVEX encoded versions)

(KL, VL) = (4, 128), (8, 256), (16, 512)

IF \*src is a memory operand\*

THEN TMP\_SRC ← BROADCAST32(SRC, VL, k1)

ELSE TMP\_SRC ← SRC

FI;

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\*

THEN DEST[i+31:i] ← RoundToIntegerSP(TMP\_SRC[i+31:i], imm8[7:0])

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+31:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+31:i] ← 0

FI;

FI;

ENDFOR;

DEST[MAX\_VL-1:VL] ← 0

### Intel C/C++ Compiler Intrinsic Equivalent

```

VRNDSCALEPS __m512 __mm512_roundscale_ps( __m512 a, int imm);
VRNDSCALEPS __m512 __mm512_roundscale_round_ps( __m512 a, int imm, int sae);
VRNDSCALEPS __m512 __mm512_mask_roundscale_ps(__m512 s, __mmask16 k, __m512 a, int imm);
VRNDSCALEPS __m512 __mm512_mask_roundscale_round_ps(__m512 s, __mmask16 k, __m512 a, int imm, int sae);
VRNDSCALEPS __m512 __mm512_maskz_roundscale_ps( __mmask16 k, __m512 a, int imm);
VRNDSCALEPS __m512 __mm512_maskz_roundscale_round_ps( __mmask16 k, __m512 a, int imm, int sae);
VRNDSCALEPS __m256 __mm256_roundscale_ps( __m256 a, int imm);
VRNDSCALEPS __m256 __mm256_mask_roundscale_ps(__m256 s, __mmask8 k, __m256 a, int imm);
VRNDSCALEPS __m256 __mm256_maskz_roundscale_ps( __mmask8 k, __m256 a, int imm);
VRNDSCALEPS __m128 __mm_roundscale_ps( __m256 a, int imm);
VRNDSCALEPS __m128 __mm_mask_roundscale_ps(__m128 s, __mmask8 k, __m128 a, int imm);
VRNDSCALEPS __m128 __mm_maskz_roundscale_ps( __mmask8 k, __m128 a, int imm);

```

### SIMD Floating-Point Exceptions

Invalid, Precision

If SPE is enabled, precision exception is not reported (regardless of MXCSR exception mask).

### Other Exceptions

See Exceptions Type E2.

## VRNDSCALESS—Round Scalar Float32 Value To Include A Given Number Of Fraction Bits

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.LIG.66.0F3A.W0 0A /r ib VRNDSCALESS xmm1 {k1}{z}, xmm2, xmm3/m32{sae}, imm8	T1S	V/V	AVX512F	Rounds scalar single-precision floating-point value in xmm3/m32 to a number of fraction bits specified by the imm8 field. Stores the result in xmm1 register under writemask.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
T1S	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Rounds the single-precision floating-point value in the low doubleword element of the second source operand (the third operand) by the rounding mode specified in the immediate operand (see Figure 5-27) and places the result in the corresponding element of the destination operand (the first operand) according to the writemask. The doubleword elements at bits 127:32 of the destination are copied from the first source operand (the second operand).

The destination and first source operands are XMM registers, the 2nd source operand can be an XMM register or memory location. Bits MAX\_VL-1:128 of the destination register are cleared.

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 single-precision floating-point value.

It should be noticed that no overflow is induced while executing this instruction (although the source is scaled by the imm8[7:4] value).

The immediate operand also specifies control fields for the rounding operation, three bit fields are defined and shown in the “Immediate Control Description” figure below. Bit 3 of the immediate byte controls the processor behavior for a precision exception, bit 2 selects the source of rounding mode control. Bits 1:0 specify a non-sticky rounding-mode value (Immediate control tables below lists the encoded values for rounding-mode field).

The Precision Floating-Point Exception is signaled according to the immediate operand. If any source operand is an SNaN then it will be converted to a QNaN. If DAZ is set to '1 then 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 for VRNDSCALESS is

$$\text{ROUND}(x) = 2^{-M} * \text{Round\_to\_INT}(x * 2^M, \text{round\_ctrl}),$$

$$\text{round\_ctrl} = \text{imm}[3:0];$$

$$M = \text{imm}[7:4];$$

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

VRNDSCALESS is a more general form of the VEX-encoded VROUNDSS instruction. In VROUNDSS, the formula of the operation on each element is

$$\text{ROUND}(x) = \text{Round\_to\_INT}(x, \text{round\_ctrl}),$$

$$\text{round\_ctrl} = \text{imm}[3:0];$$

EVEX encoded version: The source operand is a XMM register or a 32-bit memory location. The destination operand is a XMM register.

Handling of special case of input values are listed in Table 5-22.



**Operation**

```

RoundToIntegerSP(SRC[31:0], imm8[7:0]) {
    if (imm8[2] = 1)
        rounding_direction ← MXCSR:RC ; get round control from MXCSR
    else
        rounding_direction ← imm8[1:0] ; get round control from imm8[1:0]
    FI
    M ← imm8[7:4] ; get the scaling factor

    case (rounding_direction)
    00: TMP[31:0] ← round_to_nearest_even_integer(2M*SRC[31:0])
    01: TMP[31:0] ← round_to_equal_or_smaller_integer(2M*SRC[31:0])
    10: TMP[31:0] ← round_to_equal_or_larger_integer(2M*SRC[31:0])
    11: TMP[31:0] ← round_to_nearest_smallest_magnitude_integer(2M*SRC[31:0])
    ESAC;

    Dest[31:0] ← 2-M* TMP[31:0] ; scale down back to 2-M
    if (imm8[3] = 0) Then ; check SPE
        if (SRC[31:0] != Dest[31:0]) Then ; check precision lost
            set_precision() ; set #PE
        FI;
    FI;
    return(Dest[31:0])
}

```

VRNDSCALESS (EVEX encoded version)

```

IF k1[0] or *no writemask*
    THEN DEST[31:0] ← RoundToIntegerSP(SRC2[31:0], Zero_upper_imm[7:0])
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE ; zeroing-masking
                THEN DEST[31:0] ← 0
        FI;
    FI;
DEST[127:32] ← SRC1[127:32]
DEST[MAX_VL-1:128] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VRNDSCALESS __m128 __mm_roundscale_ss (__m128 a, __m128 b, int imm);
VRNDSCALESS __m128 __mm_roundscale_round_ss (__m128 a, __m128 b, int imm, int sae);
VRNDSCALESS __m128 __mm_mask_roundscale_ss (__m128 s, __mmask8 k, __m128 a, __m128 b, int imm);
VRNDSCALESS __m128 __mm_mask_roundscale_round_ss (__m128 s, __mmask8 k, __m128 a, __m128 b, int imm, int sae);
VRNDSCALESS __m128 __mm_maskz_roundscale_ss (__mmask8 k, __m128 a, __m128 b, int imm);
VRNDSCALESS __m128 __mm_maskz_roundscale_round_ss (__mmask8 k, __m128 a, __m128 b, int imm, int sae);

```

**SIMD Floating-Point Exceptions**

Invalid, Precision

If SPE is enabled, precision exception is not reported (regardless of MXCSR exception mask).

**Other Exceptions**

See Exceptions Type E3.

## VRSQRT14PD—Compute Approximate Reciprocals of Square Roots of Packed Float64 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W1 4E /r VRSQRT14PD xmm1 {k1}{z}, xmm2/m128/m64bcst	FV	V/V	AVX512VL AVX512F	Computes the approximate reciprocal square roots of the packed double-precision floating-point values in xmm2/m128/m64bcst and stores the results in xmm1. Under writemask.
EVEX.256.66.0F38.W1 4E /r VRSQRT14PD ymm1 {k1}{z}, ymm2/m256/m64bcst	FV	V/V	AVX512VL AVX512F	Computes the approximate reciprocal square roots of the packed double-precision floating-point values in ymm2/m256/m64bcst and stores the results in ymm1. Under writemask.
EVEX.512.66.0F38.W1 4E /r VRSQRT14PD zmm1 {k1}{z}, zmm2/m512/m64bcst	FV	V/V	AVX512F	Computes the approximate reciprocal square roots of the packed double-precision floating-point values in zmm2/m512/m64bcst and stores the results in zmm1 under writemask.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
FV	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

This instruction performs a SIMD computation of the approximate reciprocals of the square roots of the eight packed double-precision floating-point values in the source operand (the second operand) and stores the packed double-precision floating-point results in the destination operand (the first operand) according to the writemask. The maximum relative error for this approximation is less than  $2^{-14}$ .

EVEX.512 encoded version: The source operand can be a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

EVEX.256 encoded version: The source operand is a YMM register, a 256-bit memory location, or a 256-bit vector broadcasted from a 64-bit memory location. The destination operand is a YMM register, conditionally updated using writemask k1.

EVEX.128 encoded version: The source operand is a XMM register, a 128-bit memory location, or a 128-bit vector broadcasted from a 64-bit memory location. The destination operand is a XMM register, conditionally updated using writemask k1.

The VRSQRT14PD instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an  $\infty$  with the sign of the source value is returned. When the source operand is an  $+\infty$  then +ZERO value is returned. A denormal source value is treated as zero only if DAZ bit is set in MXCSR. Otherwise it is treated correctly and performs the approximation with the specified masked response. When a source value is a negative value (other than 0.0) a floating-point QNaN\_indefinite is returned. When a source value is an SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned.

MXCSR exception flags are not affected by this instruction and floating-point exceptions are not reported.

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

A numerically exact implementation of VRSQRT14xx can be found at <https://software.intel.com/en-us/articles/reference-implementations-for-IA-approximation-instructions-vrcp14-vrsqrt14-vrcp28-vrsqrt28-vexp2>.

**Operation****VRSQRT14PD (EVEX encoded versions)**

(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) AND (SRC \*is memory\*)

THEN DEST[i+63:i] ← APPROXIMATE(1.0/ SQRT(SRC[63:0]));

ELSE DEST[i+63:i] ← APPROXIMATE(1.0/ SQRT(SRC[i+63:i]));

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+63:i] ← 0

FI;

FI;

ENDFOR;

DEST[MAX\_VL-1:VL] ← 0

**Table 5-23. VRSQRT14PD Special Cases**

Input value	Result value	Comments
Any denormal	Normal	Cannot generate overflow
$X = 2^{-2n}$	$2^n$	
$X < 0$	QNaN_Indefinite	Including -INF
$X = -0$	-INF	
$X = +0$	+INF	
$X = +INF$	+0	

**Intel C/C++ Compiler Intrinsic Equivalent**

VRSQRT14PD \_\_m512d \_\_mm512\_rsqrt14\_pd( \_\_m512d a);

VRSQRT14PD \_\_m512d \_\_mm512\_mask\_rsqrt14\_pd(\_\_m512d s, \_\_mmask8 k, \_\_m512d a);

VRSQRT14PD \_\_m512d \_\_mm512\_maskz\_rsqrt14\_pd( \_\_mmask8 k, \_\_m512d a);

VRSQRT14PD \_\_m256d \_\_mm256\_rsqrt14\_pd( \_\_m256d a);

VRSQRT14PD \_\_m256d \_\_mm256\_mask\_rsqrt14\_pd(\_\_m256d s, \_\_mmask8 k, \_\_m256d a);

VRSQRT14PD \_\_m256d \_\_mm256\_maskz\_rsqrt14\_pd( \_\_mmask8 k, \_\_m256d a);

VRSQRT14PD \_\_m128d \_\_mm128\_rsqrt14\_pd( \_\_m128d a);

VRSQRT14PD \_\_m128d \_\_mm128\_mask\_rsqrt14\_pd(\_\_m128d s, \_\_mmask8 k, \_\_m128d a);

VRSQRT14PD \_\_m128d \_\_mm128\_maskz\_rsqrt14\_pd( \_\_mmask8 k, \_\_m128d a);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type E4.

## VRSQRT14SD—Compute Approximate Reciprocal of Square Root of Scalar Float64 Value

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.LIG.66.0F38.W1 4F /r VRSQRT14SD xmm1 {k1}{z}, xmm2, xmm3/m64	T1S	V/V	AVX512F	Computes the approximate reciprocal square root of the scalar double-precision floating-point value in xmm3/m64 and stores the result in the low quadword element of xmm1 using writemask k1. Bits[127:64] of xmm2 is copied to xmm1[127:64].

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
T1S	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Computes the approximate reciprocal of the square roots of the scalar double-precision floating-point value in the low quadword element of the source operand (the second operand) and stores the result in the low quadword element of the destination operand (the first operand) according to the writemask. The maximum relative error for this approximation is less than  $2^{-14}$ . The source operand can be an XMM register or a 32-bit memory location. The destination operand is an XMM register.

Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (MAX\_VL-1:128) of the destination register are zeroed.

The VRSQRT14SD instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an  $\infty$  with the sign of the source value is returned. When the source operand is an  $+\infty$  then +ZERO value is returned. A denormal source value is treated as zero only if DAZ bit is set in MXCSR. Otherwise it is treated correctly and performs the approximation with the specified masked response. When a source value is a negative value (other than 0.0) a floating-point QNaN\_indefinite is returned. When a source value is an SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned.

MXCSR exception flags are not affected by this instruction and floating-point exceptions are not reported.

A numerically exact implementation of VRSQRT14xx can be found at <https://software.intel.com/en-us/articles/reference-implementations-for-IA-approximation-instructions-vrcp14-vrsqrt14-vrcp28-vrsqrt28-vexp2>.

### Operation

#### VRSQRT14SD (EVEX version)

IF k1[0] or \*no writemask\*

THEN DEST[63:0]  $\leftarrow$  APPROXIMATE(1.0/ SQRT(SRC2[63:0]))

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[63:0] remains unchanged\*

ELSE ; zeroing-masking

THEN DEST[63:0]  $\leftarrow$  0

FI;

FI;

DEST[127:64]  $\leftarrow$  SRC1[127:64]

DEST[MAX\_VL-1:128]  $\leftarrow$  0

Table 5-24. VRSQRT14SD Special Cases

Input value	Result value	Comments
Any denormal	Normal	Cannot generate overflow
$X = 2^{-2n}$	$2^n$	
$X < 0$	QNaN_Indefinite	Including -INF
$X = -0$	-INF	
$X = +0$	+INF	
$X = +INF$	+0	

**Intel C/C++ Compiler Intrinsic Equivalent**

VRSQRT14SD \_\_m128d \_mm\_rsqrt14\_sd( \_\_m128d a, \_\_m128d b);

VRSQRT14SD \_\_m128d \_mm\_mask\_rsqrt14\_sd(\_\_m128d s, \_\_mmask8 k, \_\_m128d a, \_\_m128d b);

VRSQRT14SD \_\_m128d \_mm\_maskz\_rsqrt14\_sd( \_\_mmask8d m, \_\_m128d a, \_\_m128d b);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type E5.

## VRSQRT14PS—Compute Approximate Reciprocals of Square Roots of Packed Float32 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 4E /r VRSQRT14PS xmm1 {k1}{z}, xmm2/m128/m32bcst	FV	V/V	AVX512VL AVX512F	Computes the approximate reciprocal square roots of the packed single-precision floating-point values in xmm2/m128/m32bcst and stores the results in xmm1. Under writemask.
EVEX.256.66.0F38.W0 4E /r VRSQRT14PS ymm1 {k1}{z}, ymm2/m256/m32bcst	FV	V/V	AVX512VL AVX512F	Computes the approximate reciprocal square roots of the packed single-precision floating-point values in ymm2/m256/m32bcst and stores the results in ymm1. Under writemask.
EVEX.512.66.0F38.W0 4E /r VRSQRT14PS zmm1 {k1}{z}, zmm2/m512/m32bcst	FV	V/V	AVX512F	Computes the approximate reciprocal square roots of the packed single-precision floating-point values in zmm2/m512/m32bcst and stores the results in zmm1. Under writemask.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
FV	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

This instruction performs a SIMD computation of the approximate reciprocals of the square roots of 16 packed single-precision floating-point values in the source operand (the second operand) and stores the packed single-precision floating-point results in the destination operand (the first operand) according to the writemask. The maximum relative error for this approximation is less than  $2^{-14}$ .

EVEX.512 encoded version: The source operand can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

EVEX.256 encoded version: The source operand is a YMM register, a 256-bit memory location, or a 256-bit vector broadcasted from a 32-bit memory location. The destination operand is a YMM register, conditionally updated using writemask k1.

EVEX.128 encoded version: The source operand is a XMM register, a 128-bit memory location, or a 128-bit vector broadcasted from a 32-bit memory location. The destination operand is a XMM register, conditionally updated using writemask k1.

The VRSQRT14PS instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an  $\infty$  with the sign of the source value is returned. When the source operand is an  $+\infty$  then +ZERO value is returned. A denormal source value is treated as zero only if DAZ bit is set in MXCSR. Otherwise it is treated correctly and performs the approximation with the specified masked response. When a source value is a negative value (other than 0.0) a floating-point QNaN\_indefinite is returned. When a source value is an SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned.

MXCSR exception flags are not affected by this instruction and floating-point exceptions are not reported.

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

A numerically exact implementation of VRSQRT14xx can be found at <https://software.intel.com/en-us/articles/reference-implementations-for-IA-approximation-instructions-vrcp14-vrsqrt14-vrcp28-vrsqrt28-vexp2>.

**Operation****VRSQRT14PS (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\* THEN

IF (EVEX.b = 1) AND (SRC \*is memory\*)

THEN DEST[i+31:i] ← APPROXIMATE(1.0/ SQRT(SRC[31:0]));

ELSE DEST[i+31:i] ← APPROXIMATE(1.0/ SQRT(SRC[i+31:i]));

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+31:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+31:i] ← 0

FI;

FI;

ENDFOR;

DEST[MAX\_VL-1:VL] ← 0

**Table 5-25. VRSQRT14PS Special Cases**

Input value	Result value	Comments
Any denormal	Normal	Cannot generate overflow
$X = 2^{-2n}$	$2^n$	
$X < 0$	QNaN_Indefinite	Including -INF
$X = -0$	-INF	
$X = +0$	+INF	
$X = +INF$	+0	

**Intel C/C++ Compiler Intrinsic Equivalent**

VRSQRT14PS \_\_m512 \_\_mm512\_rsqr14\_ps( \_\_m512 a);

VRSQRT14PS \_\_m512 \_\_mm512\_mask\_rsqr14\_ps(\_\_m512 s, \_\_mmask16 k, \_\_m512 a);

VRSQRT14PS \_\_m512 \_\_mm512\_maskz\_rsqr14\_ps( \_\_mmask16 k, \_\_m512 a);

VRSQRT14PS \_\_m256 \_\_mm256\_rsqr14\_ps( \_\_m256 a);

VRSQRT14PS \_\_m256 \_\_mm256\_mask\_rsqr14\_ps(\_\_m256 s, \_\_mmask8 k, \_\_m256 a);

VRSQRT14PS \_\_m256 \_\_mm256\_maskz\_rsqr14\_ps( \_\_mmask8 k, \_\_m256 a);

VRSQRT14PS \_\_m128 \_\_mm\_rsqr14\_ps( \_\_m128 a);

VRSQRT14PS \_\_m128 \_\_mm\_mask\_rsqr14\_ps(\_\_m128 s, \_\_mmask8 k, \_\_m128 a);

VRSQRT14PS \_\_m128 \_\_mm\_maskz\_rsqr14\_ps( \_\_mmask8 k, \_\_m128 a);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type 4.

## VRSQRT14SS—Compute Approximate Reciprocal of Square Root of Scalar Float32 Value

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.LIG.66.0F38.W0 4F /r VRSQRT14SS xmm1 {k1}{z}, xmm2, xmm3/m32	T1S	V/V	AVX512F	Computes the approximate reciprocal square root of the scalar single-precision floating-point value in xmm3/m32 and stores the result in the low doubleword element of xmm1 using writemask k1. Bits[127:32] of xmm2 is copied to xmm1[127:32].

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
T1S	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA

### Description

Computes the approximate reciprocal of the square root of the scalar single-precision floating-point value in the low doubleword element of the source operand (the second operand) and stores the result in the low doubleword element of the destination operand (the first operand) according to the writemask. The maximum relative error for this approximation is less than  $2^{-14}$ . The source operand can be an XMM register or a 32-bit memory location. The destination operand is an XMM register.

Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (MAX\_VL-1:128) of the destination register are zeroed.

The VRSQRT14SS instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an  $\infty$  with the sign of the source value is returned. When the source operand is an  $\infty$ , zero with the sign of the source value is returned. A denormal source value is treated as zero only if DAZ bit is set in MXCSR. Otherwise it is treated correctly and performs the approximation with the specified masked response. When a source value is a negative value (other than 0.0) a floating-point indefinite is returned. When a source value is an SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned.

MXCSR exception flags are not affected by this instruction and floating-point exceptions are not reported.

A numerically exact implementation of VRSQRT14xx can be found at <https://software.intel.com/en-us/articles/reference-implementations-for-IA-approximation-instructions-vrcp14-vrsqrt14-vrcp28-vrsqrt28-vexp2>.

### Operation

#### VRSQRT14SS (EVEX version)

IF k1[0] or \*no writemask\*

THEN DEST[31:0]  $\leftarrow$  APPROXIMATE(1.0/ SQRT(SRC2[31:0]))

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[31:0] remains unchanged\*

ELSE ; zeroing-masking

THEN DEST[31:0]  $\leftarrow$  0

FI;

FI;

DEST[127:32]  $\leftarrow$  SRC1[127:32]

DEST[MAX\_VL-1:128]  $\leftarrow$  0



**Table 5-26. VRSQRT14SS Special Cases**

Input value	Result value	Comments
Any denormal	Normal	Cannot generate overflow
$X = 2^{-2n}$	$2^n$	
$X < 0$	QNaN_Indefinite	Including -INF
$X = -0$	-INF	
$X = +0$	+INF	
$X = +INF$	+0	

**Intel C/C++ Compiler Intrinsic Equivalent**

VRSQRT14SS \_\_m128 \_mm\_rsqrt14\_ss(\_\_m128 a, \_\_m128 b);

VRSQRT14SS \_\_m128 \_mm\_mask\_rsqrt14\_ss(\_\_m128 s, \_\_mmask8 k, \_\_m128 a, \_\_m128 b);

VRSQRT14SS \_\_m128 \_mm\_maskz\_rsqrt14\_ss(\_\_mmask8 k, \_\_m128 a, \_\_m128 b);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type E5.

## VRSQRT28PD—Approximation to the Reciprocal Square Root of Packed Double-Precision Floating-Point Values with Less Than $2^{-28}$ Relative Error

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.512.66.0F38.W1 CC /r VRSQRT28PD zmm1 {k1}{z}, zmm2/m512/m64bcst {sae}	FV	V/V	AVX512ER	Computes approximations to the Reciprocal square root ( $<2^{-28}$ relative error) of the packed double-precision floating-point values from zmm2/m512/m64bcst and stores result in zmm1 with writemask k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
FV	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Computes the reciprocal square root of the float64 values in the source operand (the second operand) and store the results to the destination operand (the first operand). The approximate reciprocal is evaluated with less than  $2^{-28}$  of maximum relative error.

If any source element is NaN, the quietized NaN source value is returned for that element. Negative (non-zero) source numbers, as well as  $-\infty$ , return the canonical NaN and set the Invalid Flag (#I).

A value of  $-0$  must return  $-\infty$  and set the DivByZero flags (#Z). Negative numbers should return NaN and set the Invalid flag (#I). Note however that the instruction flush input denormals to zero of the same sign, so negative denormals return  $-\infty$  and set the DivByZero flag.

The source operand is a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

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

A numerically exact implementation of VRSQRT28xx can be found at <https://software.intel.com/en-us/articles/reference-implementations-for-IA-approximation-instructions-vrcp14-vrsqrt14-vrcp28-vrsqrt28-vexp2>.

### Operation

#### VRSQRT28PD (EVEX encoded versions)

(KL, VL) = (8, 512)

FOR j  $\leftarrow$  0 TO KL-1

  i  $\leftarrow$  j \* 64

  IF k1[j] OR \*no writemask\* THEN

    IF (EVEX.b = 1) AND (SRC \*is memory\*)

      THEN DEST[i+63:i]  $\leftarrow$  (1.0/ SQRT(SRC[63:0]));

      ELSE DEST[i+63:i]  $\leftarrow$  (1.0/ SQRT(SRC[i+63:i]));

    FI;

  ELSE

    IF \*merging-masking\* ; merging-masking

      THEN \*DEST[i+63:i] remains unchanged\*

    ELSE ; zeroing-masking

      DEST[i+63:i]  $\leftarrow$  0

    FI;

  FI;

ENDFOR;

**Table 5-27. VRSQRT28PD Special Cases**

Input value	Result value	Comments
NAN	QNAN(input)	If (SRC = SNaN) then #I
$X = 2^{-2n}$	$2^n$	
$X < 0$	QNAN_Indefinite	Including -INF
$X = -0$ or negative denormal	-INF	#Z
$X = +0$ or positive denormal	+INF	#Z
$X = +INF$	+0	

**Intel C/C++ Compiler Intrinsic Equivalent**

VRSQRT28PD \_\_m512d \_\_mm512\_rsqrt28\_round\_pd(\_\_m512d a, int sae);

VRSQRT28PD \_\_m512d \_\_mm512\_mask\_rsqrt28\_round\_pd(\_\_m512d s, \_\_mmask8 m, \_\_m512d a, int sae);

VRSQRT28PD \_\_m512d \_\_mm512\_maskz\_rsqrt28\_round\_pd(\_\_mmask8 m, \_\_m512d a, int sae);

**SIMD Floating-Point Exceptions**

Invalid (if SNaN input), Divide-by-zero

**Other Exceptions**

See Exceptions Type E2.

## VRSQRT28SD—Approximation to the Reciprocal Square Root of Scalar Double-Precision Floating-Point Value with Less Than $2^{-28}$ Relative Error

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.LIG.66.0F38.W1 CD /r VRSQRT28SD xmm1 {k1}{z}, xmm2, xmm3/m64 {sae}	T1S	V/V	AVX512ER	Computes approximate reciprocal square root ( $<2^{-28}$ relative error) of the scalar double-precision floating-point value from xmm3/m64 and stores result in xmm1 with writemask k1. Also, upper double-precision floating-point value (bits[127:64]) from xmm2 is copied to xmm1[127:64].

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
T1S	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Computes the reciprocal square root of the low float64 value in the second source operand (the third operand) and store the result to the destination operand (the first operand). The approximate reciprocal square root is evaluated with less than  $2^{-28}$  of maximum relative error. The result is written into the low float64 element of xmm1 according to the writemask k1. Bits 127:64 of the destination is copied from the corresponding bits of the first source operand (the second operand).

If any source element is NaN, the quietized NaN source value is returned for that element. Negative (non-zero) source numbers, as well as  $-\infty$ , return the canonical NaN and set the Invalid Flag (#I).

A value of  $-0$  must return  $-\infty$  and set the DivByZero flags (#Z). Negative numbers should return NaN and set the Invalid flag (#I). Note however that the instruction flush input denormals to zero of the same sign, so negative denormals return  $-\infty$  and set the DivByZero flag.

The first source operand is an XMM register. The second source operand is an XMM register or a 64-bit memory location. The destination operand is a XMM register.

A numerically exact implementation of VRSQRT28xx can be found at <https://software.intel.com/en-us/articles/reference-implementations-for-IA-approximation-instructions-vrcp14-vrsqrt14-vrcp28-vrsqrt28-vexp2>.

### Operation

#### VRSQRT28SD (EVEX encoded versions)

```

IF k1[0] OR *no writemask* THEN
    DEST[63: 0] ← (1.0/ SQRT(SRC[63: 0]));
ELSE
    IF *merging-masking*           ; merging-masking
        THEN *DEST[63: 0] remains unchanged*
    ELSE                             ; zeroing-masking
        DEST[63: 0] ← 0
FI;
FI;
ENDFOR;
DEST[127:64] ← SRC1[127: 64]
DEST[MAX_VL-1:128] ← 0

```

**Table 5-28. VRSQRT28SD Special Cases**

Input value	Result value	Comments
NAN	QNAN(input)	If (SRC = SNaN) then #I
$X = 2^{-2n}$	$2^n$	
$X < 0$	QNAN_Indefinite	Including -INF
$X = -0$ or negative denormal	-INF	#Z
$X = +0$ or positive denormal	+INF	#Z
$X = +INF$	+0	

**Intel C/C++ Compiler Intrinsic Equivalent**

VRSQRT28SD \_\_m128d \_\_mm\_rsqrt28\_round\_sd(\_\_m128d a, \_\_m128b b, int sae);

VRSQRT28SD \_\_m128d \_\_mm\_mask\_rsqrt28\_round\_pd(\_\_m128d s, \_\_mmask8 m, \_\_m128d a, \_\_m128d b, int sae);

VRSQRT28SD \_\_m128d \_\_mm\_maskz\_rsqrt28\_round\_pd(\_\_mmask8 m, \_\_m128d a, \_\_m128d b, int sae);

**SIMD Floating-Point Exceptions**

Invalid (if SNaN input), Divide-by-zero

**Other Exceptions**

See Exceptions Type E3.

## VRSQRT28PS—Approximation to the Reciprocal Square Root of Packed Single-Precision Floating-Point Values with Less Than $2^{-28}$ Relative Error

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.512.66.0F38.W0 CC /r VRSQRT28PS zmm1 {k1}{z}, zmm2/m512/m32bcst {sae}	FV	V/V	AVX512ER	Computes approximations to the Reciprocal square root (< $2^{-28}$ relative error) of the packed single-precision floating-point values from zmm2/m512/m32bcst and stores result in zmm1 with writemask k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
FV	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Computes the reciprocal square root of the float32 values in the source operand (the second operand) and store the results to the destination operand (the first operand). The approximate reciprocal is evaluated with less than  $2^{-28}$  of maximum relative error prior to final rounding. The final results is rounded to <  $2^{-23}$  relative error before written to the destination.

If any source element is NaN, the quietized NaN source value is returned for that element. Negative (non-zero) source numbers, as well as  $-\infty$ , return the canonical NaN and set the Invalid Flag (#I).

A value of  $-0$  must return  $-\infty$  and set the DivByZero flags (#Z). Negative numbers should return NaN and set the Invalid flag (#I). Note however that the instruction flush input denormals to zero of the same sign, so negative denormals return  $-\infty$  and set the DivByZero flag.

The source operand is a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

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

A numerically exact implementation of VRSQRT28xx can be found at <https://software.intel.com/en-us/articles/reference-implementations-for-IA-approximation-instructions-vrcp14-vrsqrt14-vrcp28-vrsqrt28-vexp2>.

### Operation

#### VRSQRT28PS (EVEX encoded versions)

(KL, VL) = (16, 512)

FOR j  $\leftarrow$  0 TO KL-1

  i  $\leftarrow$  j \* 32

  IF k1[j] OR \*no writemask\* THEN

    IF (EVEX.b = 1) AND (SRC \*is memory\*)

      THEN DEST[i+31:i]  $\leftarrow$  (1.0/ SQRT(SRC[31:0]));

      ELSE DEST[i+31:i]  $\leftarrow$  (1.0/ SQRT(SRC[i+31:i]));

    FI;

  ELSE

    IF \*merging-masking\* ; merging-masking

      THEN \*DEST[i+31:i] remains unchanged\*

    ELSE ; zeroing-masking

      DEST[i+31:i]  $\leftarrow$  0

    FI;

  FI;

ENDFOR;

**Table 5-29. VRSQRT28PS Special Cases**

Input value	Result value	Comments
NAN	QNAN(input)	If (SRC = SNaN) then #I
$X = 2^{-2n}$	$2^n$	
$X < 0$	QNAN_Indefinite	Including -INF
$X = -0$ or negative denormal	-INF	#Z
$X = +0$ or positive denormal	+INF	#Z
$X = +INF$	+0	

**Intel C/C++ Compiler Intrinsic Equivalent**

VRSQRT28PS \_\_m512 \_\_mm512\_rsqrt28\_round\_ps(\_\_m512 a, int sae);

VRSQRT28PS \_\_m512 \_\_mm512\_mask\_rsqrt28\_round\_ps(\_\_m512 s, \_\_mmask16 m, \_\_m512 a, int sae);

VRSQRT28PS \_\_m512 \_\_mm512\_maskz\_rsqrt28\_round\_ps(\_\_mmask16 m, \_\_m512 a, int sae);

**SIMD Floating-Point Exceptions**

Invalid (if SNaN input), Divide-by-zero

**Other Exceptions**

See Exceptions Type E2.

## VRSQRT28SS—Approximation to the Reciprocal Square Root of Scalar Single-Precision Floating-Point Value with Less Than $2^{-28}$ Relative Error

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.LIG.66.0F38.W0 CD /r VRSQRT28SS xmm1 {k1}{z}, xmm2, xmm3/m32 {sae}	T1S	V/V	AVX512ER	Computes approximate reciprocal square root ( $<2^{-28}$ relative error) of the scalar single-precision floating-point value from xmm3/m32 and stores result in xmm1 with writemask k1. Also, upper 3 single-precision floating-point value (bits[127:32]) from xmm2 is copied to xmm1[127:32].

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
T1S	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Computes the reciprocal square root of the low float32 value in the second source operand (the third operand) and store the result to the destination operand (the first operand). The approximate reciprocal square root is evaluated with less than  $2^{-28}$  of maximum relative error prior to final rounding. The final result is rounded to  $< 2^{-23}$  relative error before written to the low float32 element of the destination according to the writemask k1. Bits 127:32 of the destination is copied from the corresponding bits of the first source operand (the second operand).

If any source element is NaN, the quietized NaN source value is returned for that element. Negative (non-zero) source numbers, as well as  $-\infty$ , return the canonical NaN and set the Invalid Flag (#I).

A value of  $-0$  must return  $-\infty$  and set the DivByZero flags (#Z). Negative numbers should return NaN and set the Invalid flag (#I). Note however that the instruction flush input denormals to zero of the same sign, so negative denormals return  $-\infty$  and set the DivByZero flag.

The first source operand is an XMM register. The second source operand is an XMM register or a 32-bit memory location. The destination operand is a XMM register.

A numerically exact implementation of VRSQRT28xx can be found at <https://software.intel.com/en-us/articles/reference-implementations-for-IA-approximation-instructions-vrcp14-vrsqrt14-vrcp28-vrsqrt28-vexp2>.

### Operation

#### VRSQRT28SS (EVEX encoded versions)

```

IF k1[0] OR *no writemask* THEN
    DEST[31: 0] ← (1.0/ SQRT(SRC[31: 0]));
ELSE
    IF *merging-masking*           ; merging-masking
    THEN *DEST[31: 0] remains unchanged*
    ELSE                             ; zeroing-masking
        DEST[31: 0] ← 0
    FI;
FI;
ENDFOR;
DEST[127:32] ← SRC1[127: 32]
DEST[MAX_VL-1:128] ← 0

```



**Table 5-30. VRSQRT28SS Special Cases**

Input value	Result value	Comments
NAN	QNAN(input)	If (SRC = SNaN) then #I
$X = 2^{-2n}$	$2^n$	
$X < 0$	QNAN_Indefinite	Including -INF
$X = -0$ or negative denormal	-INF	#Z
$X = +0$ or positive denormal	+INF	#Z
$X = +INF$	+0	

**Intel C/C++ Compiler Intrinsic Equivalent**

VRSQRT28SS \_\_m128 \_\_mm\_rsqrt28\_round\_ss(\_\_m128 a, \_\_m128 b, int sae);

VRSQRT28SS \_\_m128 \_\_mm512\_mask\_rsqrt28\_round\_ss(\_\_m128 s, \_\_mmask8 m, \_\_m128 a, \_\_m128 b, int sae);

VRSQRT28SS \_\_m128 \_\_mm512\_maskz\_rsqrt28\_round\_ss(\_\_mmask8 m, \_\_m128 a, \_\_m128 b, int sae);

**SIMD Floating-Point Exceptions**

Invalid (if SNaN input), Divide-by-zero

**Other Exceptions**

See Exceptions Type E3.

## VSCALEFPD—Scale Packed Float64 Values With Float64 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.128.66.0F38.W1 2C /r VSCALEFPD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	FV	V/V	AVX512VL AVX512F	Scale the packed double-precision floating-point values in xmm2 using values from xmm3/m128/m64bcst. Under writemask k1.
EVEX.NDS.256.66.0F38.W1 2C /r VSCALEFPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	FV	V/V	AVX512VL AVX512F	Scale the packed double-precision floating-point values in ymm2 using values from ymm3/m256/m64bcst. Under writemask k1.
EVEX.NDS.512.66.0F38.W1 2C /r VSCALEFPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	FV	V/V	AVX512F	Scale the packed double-precision floating-point values in zmm2 using values from zmm3/m512/m64bcst. Under writemask k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
FV	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

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

The equation of this operation is given by:

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

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

If the result cannot be represented in double precision, then the proper overflow response (for positive scaling operand), or the proper underflow response (for negative scaling operand) is issued. The overflow and underflow responses are dependent on the rounding mode (for IEEE-compliant rounding), as well as on other settings in MXCSR (exception mask bits, FTZ bit), and on the SAE bit.

The first source operand is a ZMM/YMM/XMM register. The second 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 64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

Handling of special-case input values are listed in Table 5-31 and Table 5-32.

**Table 5-31. VSCALEFPD/SD/PS/SS Special Cases**

		Src2				Set IE
		$\pm$ NaN	+Inf	-Inf	0/Denorm/Norm	
Src1	$\pm$ QNaN	QNaN(Src1)	+INF	+0	QNaN(Src1)	IF either source is SNAN
	$\pm$ SNaN	QNaN(Src1)	QNaN(Src1)	QNaN(Src1)	QNaN(Src1)	YES
	$\pm$ Inf	QNaN(Src2)	Src1	QNaN_Indefinite	Src1	IF Src2 is SNAN or -INF
	$\pm$ 0	QNaN(Src2)	QNaN_Indefinite	Src1	Src1	IF Src2 is SNAN or +INF
	Denorm/Norm	QNaN(Src2)	$\pm$ INF (Src1 sign)	$\pm$ 0 (Src1 sign)	Compute Result	IF Src2 is SNAN

Table 5-32. Additional VSCALEFPD/SD Special Cases

Special Case	Returned value	Faults
$ \text{result}  < 2^{-1074}$	$\pm 0$ or $\pm \text{Min-Denormal}$ (Src1 sign)	Underflow
$ \text{result}  \geq 2^{1024}$	$\pm \text{INF}$ (Src1 sign) or $\pm \text{Max-normal}$ (Src1 sign)	Overflow

**Operation**

```

SCALE(SRC1, SRC2)
{
  TMP_SRC2 ← SRC2
  TMP_SRC1 ← SRC1
  IF (SRC2 is denormal AND MXCSR.DAZ) THEN TMP_SRC2=0
  IF (SRC1 is denormal AND MXCSR.DAZ) THEN TMP_SRC1=0
  /* SRC2 is a 64 bits floating-point value */
  DEST[63:0] ← TMP_SRC1[63:0] * POW(2, Floor(TMP_SRC2[63:0]))
}
VSCALEFPD (EVEX encoded versions)
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1) AND (SRC2 *is register*)
  THEN
    SET_RM(EVEX.RC);
  ELSE
    SET_RM(MXCSR.RM);
FI;
FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask* THEN
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
      THEN DEST[i+63:i] ← SCALE(SRC1[i+63:i], SRC2[63:0]);
      ELSE DEST[i+63:i] ← SCALE(SRC1[i+63:i], SRC2[i+63:i]);
    FI;
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+63:i] remains unchanged*
      ELSE ; zeroing-masking
        DEST[i+63:i] ← 0
    FI
  FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VSCALEFPD __m512d __mm512_scaleg_round_pd(__m512d a, __m512d b, int);
VSCALEFPD __m512d __mm512_mask_scaleg_round_pd(__m512d s, __mmask8 k, __m512d a, __m512d b, int);
VSCALEFPD __m512d __mm512_maskz_scaleg_round_pd(__mmask8 k, __m512d a, __m512d b, int);
VSCALEFPD __m256d __mm256_scaleg_round_pd(__m256d a, __m256d b, int);
VSCALEFPD __m256d __mm256_mask_scaleg_round_pd(__m256d s, __mmask8 k, __m256d a, __m256d b, int);
VSCALEFPD __m256d __mm256_maskz_scaleg_round_pd(__mmask8 k, __m256d a, __m256d b, int);
VSCALEFPD __m128d __mm_scaleg_round_pd(__m128d a, __m128d b, int);
VSCALEFPD __m128d __mm_mask_scaleg_round_pd(__m128d s, __mmask8 k, __m128d a, __m128d b, int);
VSCALEFPD __m128d __mm_maskz_scaleg_round_pd(__mmask8 k, __m128d a, __m128d b, int);

```

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal (for Src1).

Denormal is not reported for Src2.

**Other Exceptions**

See Exceptions Type E2.

## VSCALEFSD—Scale Scalar Float64 Values With Float64 Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.LIG.66.0F38.W1 2D /r VSCALEFSD xmm1 {k1}{z}, xmm2, xmm3/m64{er}	T1S	V/V	AVX512F	Scale the scalar double-precision floating-point values in xmm2 using the value from xmm3/m64. Under writemask k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
T1S	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

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

The equation of this operation is given by:

$$\text{xmm1} := \text{xmm2} * 2^{\text{Floor}(\text{xmm3})}$$

Floor(xmm3) means maximum integer value  $\leq$  xmm3.

If the result cannot be represented in double precision, then the proper overflow response (for positive scaling operand), or the proper underflow response (for negative scaling operand) is issued. The overflow and underflow responses are dependent on the rounding mode (for IEEE-compliant rounding), as well as on other settings in MXCSR (exception mask bits, FTZ bit), and on the SAE bit.

EVEX encoded version: The first source operand is an XMM register. The second source operand is an XMM register or a memory location. The destination operand is an XMM register conditionally updated with writemask k1.

Handling of special-case input values are listed in Table 5-31 and Table 5-32.

### Operation

```
SCALE(SRC1, SRC2)
{
    ; Check for denormal operands
    TMP_SRC2 ← SRC2
    TMP_SRC1 ← SRC1
    IF (SRC2 is denormal AND MXCSR.DAZ) THEN TMP_SRC2=0
    IF (SRC1 is denormal AND MXCSR.DAZ) THEN TMP_SRC1=0
    /* SRC2 is a 64 bits floating-point value */
    DEST[63:0] ← TMP_SRC1[63:0] * POW(2, Floor(TMP_SRC2[63:0]))
}
```

**VSCALEFSD (EVEX encoded version)**

```

IF (EVEX.b= 1) and SRC2 *is a register*
  THEN
    SET_RM(EVEX.RC);
  ELSE
    SET_RM(MXCSR.RM);
FI;
IF k1[0] OR *no writemask*
  THEN DEST[63:0] ← SCALE(SRC1[63:0], SRC2[63:0])
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[63:0] remains unchanged*
    ELSE ; zeroing-masking
      DEST[63:0] ← 0
    FI
  FI;
DEST[127:64] ← SRC1[127:64]
DEST[MAX_VL-1:128] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VSCALEFSD __m128d _mm_scalef_round_sd(__m128d a, __m128d b, int);
VSCALEFSD __m128d _mm_mask_scalef_round_sd(__m128d s, __mmask8 k, __m128d a, __m128d b, int);
VSCALEFSD __m128d _mm_maskz_scalef_round_sd(__mmask8 k, __m128d a, __m128d b, int);

```

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal (for Src1).  
Denormal is not reported for Src2.

**Other Exceptions**

See Exceptions Type E3.

## VSCALEFPS—Scale Packed Float32 Values With Float32 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.128.66.0F38.W0 2C /r VSCALEFPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	FV	V/V	AVX512VL AVX512F	Scale the packed single-precision floating-point values in xmm2 using values from xmm3/m128/m32bcst. Under writemask k1.
EVEX.NDS.256.66.0F38.W0 2C /r VSCALEFPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	FV	V/V	AVX512VL AVX512F	Scale the packed single-precision values in ymm2 using floating point values from ymm3/m256/m32bcst. Under writemask k1.
EVEX.NDS.512.66.0F38.W0 2C /r VSCALEFPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}	FV	V/V	AVX512F	Scale the packed single-precision floating-point values in zmm2 using floating-point values from zmm3/m512/m32bcst. Under writemask k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
FV	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Performs a floating-point scale of the packed single-precision floating-point values in the first source operand by multiplying it by 2 power of the float32 values in second source operand.

The equation of this operation is given by:

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

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

If the result cannot be represented in single precision, then the proper overflow response (for positive scaling operand), or the proper underflow response (for negative scaling operand) is issued. The overflow and underflow responses are dependent on the rounding mode (for IEEE-compliant rounding), as well as on other settings in MXCSR (exception mask bits, FTZ bit), and on the SAE bit.

EVEX.512 encoded version: The first source operand is a ZMM register. The second source operand is a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM register conditionally updated with writemask k1.

EVEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register, a 256-bit memory location, or a 256-bit vector broadcasted from a 32-bit memory location. The destination operand is a YMM register, conditionally updated using writemask k1.

EVEX.128 encoded version: The first source operand is an XMM register. The second source operand is a XMM register, a 128-bit memory location, or a 128-bit vector broadcasted from a 32-bit memory location. The destination operand is a XMM register, conditionally updated using writemask k1.

Handling of special-case input values are listed in Table 5-31 and Table 5-33.

**Table 5-33. Additional VSCALEFPS/SS Special Cases**

Special Case	Returned value	Faults
$ \text{result}  < 2^{-149}$	$\pm 0$ or $\pm$ Min-Denormal (Src1 sign)	Underflow
$ \text{result}  \geq 2^{128}$	$\pm$ INF (Src1 sign) or $\pm$ Max-normal (Src1 sign)	Overflow

**Operation**

```

SCALE(SRC1, SRC2)
{
    ; Check for denormal operands
    TMP_SRC2 ← SRC2
    TMP_SRC1 ← SRC1
    IF (SRC2 is denormal AND MXCSR.DAZ) THEN TMP_SRC2=0
    IF (SRC1 is denormal AND MXCSR.DAZ) THEN TMP_SRC1=0
    /* SRC2 is a 32 bits floating-point value */
    DEST[31:0] ← TMP_SRC1[31:0] * POW(2, Floor(TMP_SRC2[31:0]))
}

```

**VSCALEFPS (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

```

IF (VL = 512) AND (EVEX.b = 1) AND (SRC2 *is register*)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask* THEN
        IF (EVEX.b = 1) AND (SRC2 *is memory*)
            THEN DEST[i+31:i] ← SCALE(SRC1[i+31:i], SRC2[31:0]);
            ELSE DEST[i+31:i] ← SCALE(SRC1[i+31:i], SRC2[i+31:i]);
        FI;
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+31:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+31:i] ← 0
        FI;
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0;

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VSCALEFPS __m512 __mm512_scaleg_round_ps(__m512 a, __m512 b, int);
VSCALEFPS __m512 __mm512_mask_scaleg_round_ps(__m512 s, __mmask16 k, __m512 a, __m512 b, int);
VSCALEFPS __m512 __mm512_maskz_scaleg_round_ps(__mmask16 k, __m512 a, __m512 b, int);
VSCALEFPS __m256 __mm256_scaleg_round_ps(__m256 a, __m256 b, int);
VSCALEFPS __m256 __mm256_mask_scaleg_round_ps(__m256 s, __mmask8 k, __m256 a, __m256 b, int);
VSCALEFPS __m256 __mm256_maskz_scaleg_round_ps(__mmask8 k, __m256 a, __m256 b, int);
VSCALEFPS __m128 __mm_scaleg_round_ps(__m128 a, __m128 b, int);
VSCALEFPS __m128 __mm_mask_scaleg_round_ps(__m128 s, __mmask8 k, __m128 a, __m128 b, int);
VSCALEFPS __m128 __mm_maskz_scaleg_round_ps(__mmask8 k, __m128 a, __m128 b, int);

```

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal (for Src1).  
Denormal is not reported for Src2.

**Other Exceptions**

See Exceptions Type E2.



## VSCALEFSS—Scale Scalar Float32 Value With Float32 Value

Opcode/ Instruction	Op/ En	64/32 bitMode Support	CPUID Feature Flag	Description
EVEX.NDS.LIG.66.0F38.W0 2D /r VSCALEFSS xmm1 {k1}{z}, xmm2, xmm3/m32{er}	T1S	V/V	AVX512F	Scale the scalar single-precision floating-point value in xmm2 using floating-point value from xmm3/m32. Under writemask k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
T1S	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Performs a floating-point scale of the scalar single-precision floating-point value in the first source operand by multiplying it by 2 power of the float32 value in second source operand.

The equation of this operation is given by:

$$\text{xmm1} := \text{xmm2} * 2^{\text{floor}(\text{xmm3})}$$

Floor(xmm3) means maximum integer value  $\leq$  xmm3.

If the result cannot be represented in single precision, then the proper overflow response (for positive scaling operand), or the proper underflow response (for negative scaling operand) is issued. The overflow and underflow responses are dependent on the rounding mode (for IEEE-compliant rounding), as well as on other settings in MXCSR (exception mask bits, FTZ bit), and on the SAE bit.

EVEX encoded version: The first source operand is an XMM register. The second source operand is an XMM register or a memory location. The destination operand is an XMM register conditionally updated with writemask k1.

Handling of special-case input values are listed in Table 5-31 and Table 5-33.

### Operation

```
SCALE(SRC1, SRC2)
{
    ; Check for denormal operands
    TMP_SRC2 ← SRC2
    TMP_SRC1 ← SRC1
    IF (SRC2 is denormal AND MXCSR.DAZ) THEN TMP_SRC2=0
    IF (SRC1 is denormal AND MXCSR.DAZ) THEN TMP_SRC1=0
    /* SRC2 is a 32 bits floating-point value */
    DEST[31:0] ← TMP_SRC1[31:0] * POW(2, Floor(TMP_SRC2[31:0]))
}
```

```
VSCALEFSS (EVEX encoded version)
IF (EVEX.b= 1) and SRC2 *is a register*
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
IF k1[0] OR *no writemask*
    THEN DEST[31:0] ← SCALE(SRC1[31:0], SRC2[31:0])
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[31:0] remains unchanged*
```

```

        ELSE                                ; zeroing-masking
            DEST[31:0] ← 0
    FI
FI;
DEST[127:32] ← SRC1[127:32]
DEST[MAX_VL-1:128] ← 0

```

#### Intel C/C++ Compiler Intrinsic Equivalent

```

VSCALEFSS __m128 _mm_scalef_round_ss(__m128 a, __m128 b, int);
VSCALEFSS __m128 _mm_mask_scalef_round_ss(__m128 s, __mmask8 k, __m128 a, __m128 b, int);
VSCALEFSS __m128 _mm_maskz_scalef_round_ss(__mmask8 k, __m128 a, __m128 b, int);

```

#### SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal (for Src1).  
Denormal is not reported for Src2.

#### Other Exceptions

See Exceptions Type E3.

## VSCATTERDPS/VSCATTERDPD/VSCATTERQPS/VSCATTERQPD—Scatter Packed Single, Packed Double with Signed Dword and Qword Indices

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 A2 /vsib VSCATTERDPS vm32x {k1}, xmm1	T1S	V/V	AVX512VL AVX512F	Using signed dword indices, scatter single-precision floating-point values to memory using writemask k1.
EVEX.256.66.0F38.W0 A2 /vsib VSCATTERDPS vm32y {k1}, ymm1	T1S	V/V	AVX512VL AVX512F	Using signed dword indices, scatter single-precision floating-point values to memory using writemask k1.
EVEX.512.66.0F38.W0 A2 /vsib VSCATTERDPS vm32z {k1}, zmm1	T1S	V/V	AVX512F	Using signed dword indices, scatter single-precision floating-point values to memory using writemask k1.
EVEX.128.66.0F38.W1 A2 /vsib VSCATTERDPD vm32x {k1}, xmm1	T1S	V/V	AVX512VL AVX512F	Using signed dword indices, scatter double-precision floating-point values to memory using writemask k1.
EVEX.256.66.0F38.W1 A2 /vsib VSCATTERDPD vm32x {k1}, ymm1	T1S	V/V	AVX512VL AVX512F	Using signed dword indices, scatter double-precision floating-point values to memory using writemask k1.
EVEX.512.66.0F38.W1 A2 /vsib VSCATTERDPD vm32y {k1}, zmm1	T1S	V/V	AVX512F	Using signed dword indices, scatter double-precision floating-point values to memory using writemask k1.
EVEX.128.66.0F38.W0 A3 /vsib VSCATTERQPS vm64x {k1}, xmm1	T1S	V/V	AVX512VL AVX512F	Using signed qword indices, scatter single-precision floating-point values to memory using writemask k1.
EVEX.256.66.0F38.W0 A3 /vsib VSCATTERQPS vm64y {k1}, xmm1	T1S	V/V	AVX512VL AVX512F	Using signed qword indices, scatter single-precision floating-point values to memory using writemask k1.
EVEX.512.66.0F38.W0 A3 /vsib VSCATTERQPS vm64z {k1}, ymm1	T1S	V/V	AVX512F	Using signed qword indices, scatter single-precision floating-point values to memory using writemask k1.
EVEX.128.66.0F38.W1 A3 /vsib VSCATTERQPD vm64x {k1}, xmm1	T1S	V/V	AVX512VL AVX512F	Using signed qword indices, scatter double-precision floating-point values to memory using writemask k1.
EVEX.256.66.0F38.W1 A3 /vsib VSCATTERQPD vm64y {k1}, ymm1	T1S	V/V	AVX512VL AVX512F	Using signed qword indices, scatter double-precision floating-point values to memory using writemask k1.
EVEX.512.66.0F38.W1 A3 /vsib VSCATTERQPD vm64z {k1}, zmm1	T1S	V/V	AVX512F	Using signed qword indices, scatter double-precision floating-point values to memory using writemask k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
T1S	BaseReg (R): VSIB:base, VectorReg(R): VSIB:index	ModRM:reg (r)	NA	NA

### Description

Stores up to 16 elements (or 8 elements) in doubleword/quadword vector zmm1 to the memory locations pointed by base address `BASE_ADDR` and index vector `VINDEX`, with scale `SCALE`. The elements are specified via the VSIB (i.e., the index register is a vector register, holding packed indices). Elements will only be stored if their corre-

sponding mask bit is one. The entire mask register will be set to zero by this instruction unless it triggers an exception.

This instruction can be suspended by an exception if at least one element is already scattered (i.e., if the exception is triggered by an element other than the rightmost one with its mask bit set). When this happens, the destination register and the mask register (k1) are partially updated. If any traps or interrupts are pending from already scattered elements, they will be delivered in lieu of the exception; in this case, EFLAG.RF is set to one so an instruction breakpoint is not re-triggered when the instruction is continued.

Note that:

- Only writes to overlapping vector indices are guaranteed to be ordered with respect to each other (from LSB to MSB of the source registers). Note that this also include partially overlapping vector indices. Writes that are not overlapped may happen in any order. Memory ordering with other instructions follows the Intel-64 memory ordering model. Note that this does not account for non-overlapping indices that map into the same physical address locations.
- If two or more destination indices completely overlap, the “earlier” write(s) may be skipped.
- Faults are delivered in a right-to-left manner. That is, if a fault is triggered by an element and delivered, all elements closer to the LSB of the destination zmm will be completed (and non-faulting). Individual elements closer to the MSB may or may not be completed. If a given element triggers multiple faults, they are delivered in the conventional order.
- Elements may be scattered in any order, but faults must be delivered in a right-to left order; thus, elements to the left of a faulting one may be gathered before the fault is delivered. A given implementation of this instruction is repeatable - given the same input values and architectural state, the same set of elements to the left of the faulting one will be gathered.
- This instruction does not perform AC checks, and so will never deliver an AC fault.
- Not valid with 16-bit effective addresses. Will deliver a #UD fault.
- If this instruction overwrites itself and then takes a fault, only a subset of elements may be completed before the fault is delivered (as described above). If the fault handler completes and attempts to re-execute this instruction, the new instruction will be executed, and the scatter will not complete.

Note that the presence of VSIB byte is enforced in this instruction. Hence, the instruction will #UD fault if ModRM.rm is different than 100b.

This instruction has special  $\text{disp8} * N$  and alignment rules. N is considered to be the size of a single vector element.

The scaled index may require more bits to represent than the address bits used by the processor (e.g., in 32-bit mode, if the scale is greater than one). In this case, the most significant bits beyond the number of address bits are ignored.

The instruction will #UD fault if the k0 mask register is specified

### Operation

BASE\_ADDR stands for the memory operand base address (a GPR); may not exist

VINDEX stands for the memory operand vector of indices (a ZMM register)

SCALE stands for the memory operand scalar (1, 2, 4 or 8)

DISP is the optional 1, 2 or 4 byte displacement

**VSCATTERDPS (EVEX encoded versions)**

(KL, VL)= (4, 128), (8, 256), (16, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN MEM[BASE_ADDR + SignExtend(VINDEX[i+31:i]) * SCALE + DISP] ←
      SRC[i+31:i]
      k1[j] ← 0
  FI;
ENDFOR
k1[MAX_KL-1:KL] ← 0

```

**VSCATTERDPD (EVEX encoded versions)**

(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 MEM[BASE_ADDR + SignExtend(VINDEX[k+31:k]) * SCALE + DISP] ←
      SRC[i+63:i]
      k1[j] ← 0
  FI;
ENDFOR
k1[MAX_KL-1:KL] ← 0

```

**VSCATTERQPS (EVEX encoded versions)**

(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 MEM[BASE_ADDR + (VINDEX[k+63:k]) * SCALE + DISP] ←
      SRC[i+31:i]
      k1[j] ← 0
  FI;
ENDFOR
k1[MAX_KL-1:KL] ← 0

```

**VSCATTERQPD (EVEX encoded versions)**

(KL, VL)= (2, 128), (4, 256), (8, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN MEM[BASE_ADDR + (VINDEX[i+63:i]) * SCALE + DISP] ←
      SRC[i+63:i]
      k1[j] ← 0
  FI;
ENDFOR
k1[MAX_KL-1:KL] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VSCATTERDPD void __mm512_i32scatter_pd(void * base, __m256i vdx, __m512d a, int scale);
VSCATTERDPD void __mm512_mask_i32scatter_pd(void * base, __mmask8 k, __m256i vdx, __m512d a, int scale);
VSCATTERDPS void __mm512_i32scatter_ps(void * base, __m512i vdx, __m512 a, int scale);
VSCATTERDPS void __mm512_mask_i32scatter_ps(void * base, __mmask16 k, __m512i vdx, __m512 a, int scale);
VSCATTERQPD void __mm512_i64scatter_pd(void * base, __m512i vdx, __m512d a, int scale);
VSCATTERQPD void __mm512_mask_i64scatter_pd(void * base, __mmask8 k, __m512i vdx, __m512d a, int scale);
VSCATTERQPS void __mm512_i64scatter_ps(void * base, __m512i vdx, __m256 a, int scale);
VSCATTERQPS void __mm512_mask_i64scatter_ps(void * base, __mmask8 k, __m512i vdx, __m256 a, int scale);
VSCATTERDPD void __mm256_i32scatter_pd(void * base, __m128i vdx, __m256d a, int scale);
VSCATTERDPD void __mm256_mask_i32scatter_pd(void * base, __mmask8 k, __m128i vdx, __m256d a, int scale);
VSCATTERDPS void __mm256_i32scatter_ps(void * base, __m256i vdx, __m256 a, int scale);
VSCATTERDPS void __mm256_mask_i32scatter_ps(void * base, __mmask8 k, __m256i vdx, __m256 a, int scale);
VSCATTERQPD void __mm256_i64scatter_pd(void * base, __m256i vdx, __m256d a, int scale);
VSCATTERQPD void __mm256_mask_i64scatter_pd(void * base, __mmask8 k, __m256i vdx, __m256d a, int scale);
VSCATTERQPS void __mm256_i64scatter_ps(void * base, __m256i vdx, __m128 a, int scale);
VSCATTERQPS void __mm256_mask_i64scatter_ps(void * base, __mmask8 k, __m256i vdx, __m128 a, int scale);
VSCATTERDPD void __mm_i32scatter_pd(void * base, __m128i vdx, __m128d a, int scale);
VSCATTERDPD void __mm_mask_i32scatter_pd(void * base, __mmask8 k, __m128i vdx, __m128d a, int scale);
VSCATTERDPS void __mm_i32scatter_ps(void * base, __m128i vdx, __m128 a, int scale);
VSCATTERDPS void __mm_mask_i32scatter_ps(void * base, __mmask8 k, __m128i vdx, __m128 a, int scale);
VSCATTERQPD void __mm_i64scatter_pd(void * base, __m128i vdx, __m128d a, int scale);
VSCATTERQPD void __mm_mask_i64scatter_pd(void * base, __mmask8 k, __m128i vdx, __m128d a, int scale);
VSCATTERQPS void __mm_i64scatter_ps(void * base, __m128i vdx, __m128 a, int scale);
VSCATTERQPS void __mm_mask_i64scatter_ps(void * base, __mmask8 k, __m128i vdx, __m128 a, int scale);

```

**SIMD Floating-Point Exceptions**

Invalid, Overflow, Underflow, Precision, Denormal

**Other Exceptions**

See Exceptions Type E12.

## VSCATTERPFODPS/VSCATTERPFOQPS/VSCATTERPFODPD/VSCATTERPFOQPD—Sparse Prefetch Packed SP/DP Data Values with Signed Dword, Signed Qword Indices Using T0 Hint with Intent to Write

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.512.66.0F38.W0 C6 /5 /vsib VSCATTERPFODPS vm32z {k1}	T1S	V/V	AVX512PF	Using signed dword indices, prefetch sparse byte memory locations containing single-precision data using writemask k1 and T0 hint with intent to write.
EVEX.512.66.0F38.W0 C7 /5 /vsib VSCATTERPFOQPS vm64z {k1}	T1S	V/V	AVX512PF	Using signed qword indices, prefetch sparse byte memory locations containing single-precision data using writemask k1 and T0 hint with intent to write.
EVEX.512.66.0F38.W1 C6 /5 /vsib VSCATTERPFODPD vm32y {k1}	T1S	V/V	AVX512PF	Using signed dword indices, prefetch sparse byte memory locations containing double-precision data using writemask k1 and T0 hint with intent to write.
EVEX.512.66.0F38.W1 C7 /5 /vsib VSCATTERPFOQPD vm64z {k1}	T1S	V/V	AVX512PF	Using signed qword indices, prefetch sparse byte memory locations containing double-precision data using writemask k1 and T0 hint with intent to write.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
T1S	BaseReg (R): VSIB:base, VectorReg(R): VSIB:index	NA	NA	NA

### Description

The instruction conditionally prefetches up to sixteen 32-bit or eight 64-bit integer byte data elements. The elements are specified via the VSIB (i.e., the index register is an zmm, holding packed indices). Elements will only be prefetched if their corresponding mask bit is one.

cache lines will be brought into exclusive state (RFO) specified by a locality hint (T0):

- T0 (temporal data)—prefetch data into the first level cache.

[PS data] For dword indices, the instruction will prefetch sixteen memory locations. For qword indices, the instruction will prefetch eight values.

[PD data] For dword and qword indices, the instruction will prefetch eight memory locations.

Note that:

- (1) The prefetches may happen in any order (or not at all). The instruction is a hint.
- (2) The mask is left unchanged.
- (3) Not valid with 16-bit effective addresses. Will deliver a #UD fault.
- (4) No FP nor memory faults may be produced by this instruction.
- (5) Prefetches do not handle cache line splits
- (6) A #UD is signaled if the memory operand is encoded without the SIB byte.

### Operation

BASE\_ADDR stands for the memory operand base address (a GPR); may not exist

VINDEX stands for the memory operand vector of indices (a vector register)

SCALE stands for the memory operand scalar (1, 2, 4 or 8)

DISP is the optional 1, 2 or 4 byte displacement

PREFETCH(mem, Level, State) Prefetches a byte memory location pointed by 'mem' into the cache level specified by 'Level'; a request for exclusive/ownership is done if 'State' is 1. Note that the memory location ignore cache line splits. This operation is considered a hint for the processor and may be skipped depending on implementation.

**VSCATTERPFODPS (EVEX encoded version)**

```
(KL, VL) = (16, 512)
FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j]
    Prefetch( [BASE_ADDR + SignExtend(VINDEX[j+31:i]) * SCALE + DISP], Level=0, RFO = 1)
  FI;
ENDFOR
```

**VSCATTERPFODPD (EVEX encoded version)**

```
(KL, VL) = (8, 512)
FOR j ← 0 TO KL-1
  i ← j * 64
  k ← j * 32
  IF k1[j]
    Prefetch( [BASE_ADDR + SignExtend(VINDEX[k+31:k]) * SCALE + DISP], Level=0, RFO = 1)
  FI;
ENDFOR
```

**VSCATTERPFQPS (EVEX encoded version)**

```
(KL, VL) = (8, 256)
FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j]
    Prefetch( [BASE_ADDR + SignExtend(VINDEX[j+63:i]) * SCALE + DISP], Level=0, RFO = 1)
  FI;
ENDFOR
```

**VSCATTERPFQPD (EVEX encoded version)**

```
(KL, VL) = (8, 512)
FOR j ← 0 TO KL-1
  i ← j * 64
  k ← j * 64
  IF k1[j]
    Prefetch( [BASE_ADDR + SignExtend(VINDEX[k+63:k]) * SCALE + DISP], Level=0, RFO = 1)
  FI;
ENDFOR
```

**Intel C/C++ Compiler Intrinsic Equivalent**

```
VSCATTERPFODPD void __mm512_prefetch_i32scatter_pd(void *base, __m256i vdx, int scale, int hint);
VSCATTERPFODPD void __mm512_mask_prefetch_i32scatter_pd(void *base, __mmask8 m, __m256i vdx, int scale, int hint);
VSCATTERPFODPS void __mm512_prefetch_i32scatter_ps(void *base, __m512i vdx, int scale, int hint);
VSCATTERPFODPS void __mm512_mask_prefetch_i32scatter_ps(void *base, __mmask16 m, __m512i vdx, int scale, int hint);
VSCATTERPFQPD void __mm512_prefetch_i64scatter_pd(void *base, __m512i vdx, int scale, int hint);
VSCATTERPFQPD void __mm512_mask_prefetch_i64scatter_pd(void *base, __mmask8 m, __m512i vdx, int scale, int hint);
VSCATTERPFQPS void __mm512_prefetch_i64scatter_ps(void *base, __m512i vdx, int scale, int hint);
VSCATTERPFQPS void __mm512_mask_prefetch_i64scatter_ps(void *base, __mmask8 m, __m512i vdx, int scale, int hint);
```

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type E12NP.



## VSCATTERPF1DPS/VSCATTERPF1QPS/VSCATTERPF1DPD/VSCATTERPF1QPD—Sparse Prefetch Packed SP/DP Data Values with Signed Dword, Signed Qword Indices Using T1 Hint with Intent to Write

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.512.66.0F38.W0 C6 /6 /vsib VSCATTERPF1DPS vm32z {k1}	T1S	V/V	AVX512PF	Using signed dword indices, prefetch sparse byte memory locations containing single-precision data using writemask k1 and T1 hint with intent to write.
EVEX.512.66.0F38.W0 C7 /6 /vsib VSCATTERPF1QPS vm64z {k1}	T1S	V/V	AVX512PF	Using signed qword indices, prefetch sparse byte memory locations containing single-precision data using writemask k1 and T1 hint with intent to write.
EVEX.512.66.0F38.W1 C6 /6 /vsib VSCATTERPF1DPD vm32y {k1}	T1S	V/V	AVX512PF	Using signed dword indices, prefetch sparse byte memory locations containing double-precision data using writemask k1 and T1 hint with intent to write.
EVEX.512.66.0F38.W1 C7 /6 /vsib VSCATTERPF1QPD vm64z {k1}	T1S	V/V	AVX512PF	Using signed qword indices, prefetch sparse byte memory locations containing double-precision data using writemask k1 and T1 hint with intent to write.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
T1S	BaseReg (R): VSIB:base, VectorReg(R): VSIB:index	NA	NA	NA

### Description

The instruction conditionally prefetches up to sixteen 32-bit or eight 64-bit integer byte data elements. The elements are specified via the VSIB (i.e., the index register is an zmm, holding packed indices). Elements will only be prefetched if their corresponding mask bit is one.

cache lines will be brought into exclusive state (RFO) specified by a locality hint (T1):

- T1 (temporal data)—prefetch data into the second level cache.

[PS data] For dword indices, the instruction will prefetch sixteen memory locations. For qword indices, the instruction will prefetch eight values.

[PD data] For dword and qword indices, the instruction will prefetch eight memory locations.

Note that:

- (1) The prefetches may happen in any order (or not at all). The instruction is a hint.
- (2) The mask is left unchanged.
- (3) Not valid with 16-bit effective addresses. Will deliver a #UD fault.
- (4) No FP nor memory faults may be produced by this instruction.
- (5) Prefetches do not handle cache line splits
- (6) A #UD is signaled if the memory operand is encoded without the SIB byte.

### Operation

BASE\_ADDR stands for the memory operand base address (a GPR); may not exist

VINDEX stands for the memory operand vector of indices (a vector register)

SCALE stands for the memory operand scalar (1, 2, 4 or 8)

DISP is the optional 1, 2 or 4 byte displacement

PREFETCH(mem, Level, State) Prefetches a byte memory location pointed by 'mem' into the cache level specified by 'Level'; a request for exclusive/ownership is done if 'State' is 1. Note that the memory location ignore cache line splits. This operation is considered a hint for the processor and may be skipped depending on implementation.

**VSCATTERPF1DPS (EVEX encoded version)**

(KL, VL) = (16, 512)

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[jj]

Prefetch( [BASE\_ADDR + SignExtend(VINDEX[j+31:i]) \* SCALE + DISP], Level=1, RFO = 1)

FI;

ENDFOR

**VSCATTERPF1DPD (EVEX encoded version)**

(KL, VL) = (8, 512)

FOR j ← 0 TO KL-1

i ← j \* 64

k ← j \* 32

IF k1[jj]

Prefetch( [BASE\_ADDR + SignExtend(VINDEX[k+31:k]) \* SCALE + DISP], Level=1, RFO = 1)

FI;

ENDFOR

**VSCATTERPF1QPS (EVEX encoded version)**

(KL, VL) = (8, 512)

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[jj]

Prefetch( [BASE\_ADDR + SignExtend(VINDEX[j+63:i]) \* SCALE + DISP], Level=1, RFO = 1)

FI;

ENDFOR

**VSCATTERPF1QPD (EVEX encoded version)**

(KL, VL) = (8, 512)

FOR j ← 0 TO KL-1

i ← j \* 64

k ← j \* 64

IF k1[jj]

Prefetch( [BASE\_ADDR + SignExtend(VINDEX[k+63:k]) \* SCALE + DISP], Level=1, RFO = 1)

FI;

ENDFOR

**Intel C/C++ Compiler Intrinsic Equivalent**

VSCATTERPF1DPD void \_\_mm512\_prefetch\_i32scatter\_pd(void \*base, \_\_m256i vdx, int scale, int hint);

VSCATTERPF1DPD void \_\_mm512\_mask\_prefetch\_i32scatter\_pd(void \*base, \_\_mmask8 m, \_\_m256i vdx, int scale, int hint);

VSCATTERPF1DPS void \_\_mm512\_prefetch\_i32scatter\_ps(void \*base, \_\_m512i vdx, int scale, int hint);

VSCATTERPF1DPS void \_\_mm512\_mask\_prefetch\_i32scatter\_ps(void \*base, \_\_mmask16 m, \_\_m512i vdx, int scale, int hint);

VSCATTERPF1QPD void \_\_mm512\_prefetch\_i64scatter\_pd(void \*base, \_\_m512i vdx, int scale, int hint);

VSCATTERPF1QPD void \_\_mm512\_mask\_prefetch\_i64scatter\_pd(void \*base, \_\_mmask8 m, \_\_m512i vdx, int scale, int hint);

VSCATTERPF1QPS void \_\_mm512\_prefetch\_i64scatter\_ps(void \*base, \_\_m512i vdx, int scale, int hint);

VSCATTERPF1QPS void \_\_mm512\_mask\_prefetch\_i64scatter\_ps(void \*base, \_\_mmask8 m, \_\_m512i vdx, int scale, int hint);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type E12NP.

## VSHUFF32x4/VSHUFF64x2/VSHUFI32x4/VSHUFI64x2—Shuffle Packed Values at 128-bit Granularity

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.256.66.0F3A.W0 23 /r ib VSHUFF32X4 ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst, imm8	FV	V/V	AVX512VL AVX512F	Shuffle 128-bit packed single-precision floating-point values selected by imm8 from ymm2 and ymm3/m256/m32bcst and place results in ymm1 subject to writemask k1.
EVEX.NDS.512.66.0F3A.W0 23 /r ib VSHUFF32x4 zmm1{k1}{z}, zmm2, zmm3/m512/m32bcst, imm8	FV	V/V	AVX512F	Shuffle 128-bit packed single-precision floating-point values selected by imm8 from zmm2 and zmm3/m512/m32bcst and place results in zmm1 subject to writemask k1.
EVEX.NDS.256.66.0F3A.W1 23 /r ib VSHUFF64X2 ymm1{k1}{z}, ymm2, ymm3/m256/m64bcst, imm8	FV	V/V	AVX512VL AVX512F	Shuffle 128-bit packed double-precision floating-point values selected by imm8 from ymm2 and ymm3/m256/m64bcst and place results in ymm1 subject to writemask k1.
EVEX.NDS.512.66.0F3A.W1 23 /r ib VSHUFF64x2 zmm1{k1}{z}, zmm2, zmm3/m512/m64bcst, imm8	FV	V/V	AVX512F	Shuffle 128-bit packed double-precision floating-point values selected by imm8 from zmm2 and zmm3/m512/m64bcst and place results in zmm1 subject to writemask k1.
EVEX.NDS.256.66.0F3A.W0 43 /r ib VSHUFI32X4 ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst, imm8	FV	V/V	AVX512VL AVX512F	Shuffle 128-bit packed double-word values selected by imm8 from ymm2 and ymm3/m256/m32bcst and place results in ymm1 subject to writemask k1.
EVEX.NDS.512.66.0F3A.W0 43 /r ib VSHUFI32x4 zmm1{k1}{z}, zmm2, zmm3/m512/m32bcst, imm8	FV	V/V	AVX512F	Shuffle 128-bit packed double-word values selected by imm8 from zmm2 and zmm3/m512/m32bcst and place results in zmm1 subject to writemask k1.
EVEX.NDS.256.66.0F3A.W1 43 /r ib VSHUFI64X2 ymm1{k1}{z}, ymm2, ymm3/m256/m64bcst, imm8	FV	V/V	AVX512VL AVX512F	Shuffle 128-bit packed quad-word values selected by imm8 from ymm2 and ymm3/m256/m64bcst and place results in ymm1 subject to writemask k1.
EVEX.NDS.512.66.0F3A.W1 43 /r ib VSHUFI64x2 zmm1{k1}{z}, zmm2, zmm3/m512/m64bcst, imm8	FV	V/V	AVX512F	Shuffle 128-bit packed quad-word values selected by imm8 from zmm2 and zmm3/m512/m64bcst and place results in zmm1 subject to writemask k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
FV	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

**256-bit Version:** Moves one of the two 128-bit packed single-precision floating-point values from the first source operand (second operand) into the low 128-bit of the destination operand (first operand); moves one of the two packed 128-bit floating-point values from the second source operand (third operand) into the high 128-bit of the destination operand. The selector operand (third operand) determines which values are moved to the destination operand.

**512-bit Version:** Moves two of the four 128-bit packed single-precision floating-point values from the first source operand (second operand) into the low 256-bit of each double qword of the destination operand (first operand); moves two of the four packed 128-bit floating-point values from the second source operand (third operand) into the high 256-bit of the destination operand. The selector operand (third operand) determines which values are moved to the destination operand.

The first source operand is a vector register. The second source operand can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a vector register.

The writemask updates the destination operand with the granularity of 32/64-bit data elements.

### Operation

```
Select2(SRC, control) {
CASE (control[0]) OF
  0: TMP ← SRC[127:0];
  1: TMP ← SRC[255:128];
ESAC;
RETURN TMP
}
```

```
Select4(SRC, control) {
CASE (control[1:0]) OF
  0: TMP ← SRC[127:0];
  1: TMP ← SRC[255:128];
  2: TMP ← SRC[383:256];
  3: TMP ← SRC[511:384];
ESAC;
RETURN TMP
}
```

### VSHUFF32x4 (EVEX versions)

(KL, VL) = (8, 256), (16, 512)

```
FOR j ← 0 TO KL-1
  i ← j * 32
  IF (EVEX.b = 1) AND (SRC2 *is memory*)
    THEN TMP_SRC2[i+31:i] ← SRC2[31:0]
    ELSE TMP_SRC2[i+31:i] ← SRC2[i+31:i]
  FI;
ENDFOR;
IF VL = 256
  TMP_DEST[127:0] ← Select2(SRC1[255:0], imm8[0]);
  TMP_DEST[255:128] ← Select2(SRC2[255:0], imm8[1]);
FI;
IF VL = 512
  TMP_DEST[127:0] ← Select4(SRC1[511:0], imm8[1:0]);
  TMP_DEST[255:128] ← Select4(SRC1[511:0], imm8[3:2]);
  TMP_DEST[383:256] ← Select4(TMP_SRC2[511:0], imm8[5:4]);
  TMP_DEST[511:384] ← Select4(TMP_SRC2[511:0], imm8[7:6]);
FI;
FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+31:i] remains unchanged*
      ELSE *zeroing-masking* ; zeroing-masking
        THEN DEST[i+31:i] ← 0
      FI;
    FI;
  FI;
ENDFOR;
```

```

ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

### VSHUFF64x2 (EVEX 512-bit version)

(KL, VL) = (4, 256), (8, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 64
  IF (EVEX.b = 1) AND (SRC2 *is memory*)
    THEN TMP_SRC2[i+63:i] ← SRC2[63:0]
    ELSE TMP_SRC2[i+63:i] ← SRC2[i+63:i]
  FI;
ENDFOR;
IF VL = 256
  TMP_DEST[127:0] ← Select2(SRC1[255:0], imm8[0]);
  TMP_DEST[255:128] ← Select2(SRC2[255:0], imm8[1]);
FI;
IF VL = 512
  TMP_DEST[127:0] ← Select4(SRC1[511:0], imm8[1:0]);
  TMP_DEST[255:128] ← Select4(SRC1[511:0], imm8[3:2]);
  TMP_DEST[383:256] ← Select4(TMP_SRC2[511:0], imm8[5:4]);
  TMP_DEST[511:384] ← Select4(TMP_SRC2[511:0], imm8[7:6]);
FI;
FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+63:i] remains unchanged*
        ELSE *zeroing-masking* ; zeroing-masking
          THEN DEST[i+63:i] ← 0
      FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

### VSHUFI32x4 (EVEX 512-bit version)

(KL, VL) = (8, 256), (16, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 32
  IF (EVEX.b = 1) AND (SRC2 *is memory*)
    THEN TMP_SRC2[i+31:i] ← SRC2[31:0]
    ELSE TMP_SRC2[i+31:i] ← SRC2[i+31:i]
  FI;
ENDFOR;
IF VL = 256
  TMP_DEST[127:0] ← Select2(SRC1[255:0], imm8[0]);
  TMP_DEST[255:128] ← Select2(SRC2[255:0], imm8[1]);
FI;
IF VL = 512
  TMP_DEST[127:0] ← Select4(SRC1[511:0], imm8[1:0]);
  TMP_DEST[255:128] ← Select4(SRC1[511:0], imm8[3:2]);
  TMP_DEST[383:256] ← Select4(TMP_SRC2[511:0], imm8[5:4]);
  TMP_DEST[511:384] ← Select4(TMP_SRC2[511:0], imm8[7:6]);

```

```

FI;
FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[j+31:i] ← TMP_DEST[j+31:i]
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[j+31:i] remains unchanged*
    ELSE *zeroing-masking* ; zeroing-masking
      THEN DEST[j+31:i] ← 0
    FI
  FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**VSHUFI64x2 (EVEX 512-bit version)**

(KL, VL) = (4, 256), (8, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 64
  IF (EVEX.b = 1) AND (SRC2 *is memory*)
    THEN TMP_SRC2[j+63:i] ← SRC2[63:0]
  ELSE TMP_SRC2[j+63:i] ← SRC2[j+63:i]
  FI;
ENDFOR;
IF VL = 256
  TMP_DEST[127:0] ← Select2(SRC1[255:0], imm8[0]);
  TMP_DEST[255:128] ← Select2(SRC2[255:0], imm8[1]);
FI;
IF VL = 512
  TMP_DEST[127:0] ← Select4(SRC1[511:0], imm8[1:0]);
  TMP_DEST[255:128] ← Select4(SRC1[511:0], imm8[3:2]);
  TMP_DEST[383:256] ← Select4(TMP_SRC2[511:0], imm8[5:4]);
  TMP_DEST[511:384] ← Select4(TMP_SRC2[511:0], imm8[7:6]);
FI;
FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[j+63:i] ← TMP_DEST[j+63:i]
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[j+63:i] remains unchanged*
    ELSE *zeroing-masking* ; zeroing-masking
      THEN DEST[j+63:i] ← 0
    FI
  FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VSHUFI32x4 __m512i __mm512_shuffle_i32x4(__m512i a, __m512i b, int imm);
VSHUFI32x4 __m512i __mm512_mask_shuffle_i32x4(__m512i s, __mmask16 k, __m512i a, __m512i b, int imm);
VSHUFI32x4 __m512i __mm512_maskz_shuffle_i32x4(__mmask16 k, __m512i a, __m512i b, int imm);
VSHUFI32x4 __m256i __mm256_shuffle_i32x4(__m256i a, __m256i b, int imm);
VSHUFI32x4 __m256i __mm256_mask_shuffle_i32x4(__m256i s, __mmask8 k, __m256i a, __m256i b, int imm);

```

```

VSHUFI32x4 __m256i __mm256_mask_shuffle_i32x4(__mmask8 k, __m256i a, __m256i b, int imm);
VSHUFF32x4 __m512 __mm512_shuffle_f32x4(__m512 a, __m512 b, int imm);
VSHUFF32x4 __m512 __mm512_mask_shuffle_f32x4(__m512 s, __mmask16 k, __m512 a, __m512 b, int imm);
VSHUFF32x4 __m512 __mm512_mask_shuffle_f32x4(__mmask16 k, __m512 a, __m512 b, int imm);
VSHUFI64x2 __m512i __mm512_shuffle_i64x2(__m512i a, __m512i b, int imm);
VSHUFI64x2 __m512i __mm512_mask_shuffle_i64x2(__m512i s, __mmask8 k, __m512i b, __m512i b, int imm);
VSHUFI64x2 __m512i __mm512_mask_shuffle_i64x2(__mmask8 k, __m512i a, __m512i b, int imm);
VSHUFF64x2 __m512d __mm512_shuffle_f64x2(__m512d a, __m512d b, int imm);
VSHUFF64x2 __m512d __mm512_mask_shuffle_f64x2(__m512d s, __mmask8 k, __m512d a, __m512d b, int imm);
VSHUFF64x2 __m512d __mm512_mask_shuffle_f64x2(__mmask8 k, __m512d a, __m512d b, int imm);

```

### SIMD Floating-Point Exceptions

None

### Other Exceptions

See Exceptions Type E4NF.

#UD If EVEX.L'L = 0 for VSHUFF32x4/VSHUFF64x2.

## XADD—Exchange and Add

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
OF C0 /r	XADD r/m8, r8	MR	Valid	Valid	Exchange r8 and r/m8; load sum into r/m8.
REX + OF C0 /r	XADD r/m8*, r8*	MR	Valid	N.E.	Exchange r8 and r/m8; load sum into r/m8.
OF C1 /r	XADD r/m16, r16	MR	Valid	Valid	Exchange r16 and r/m16; load sum into r/m16.
OF C1 /r	XADD r/m32, r32	MR	Valid	Valid	Exchange r32 and r/m32; load sum into r/m32.
REX.W + OF C1 /r	XADD r/m64, r64	MR	Valid	N.E.	Exchange r64 and r/m64; load sum into r/m64.

### NOTES:

\* In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MR	ModRM:r/m (r, w)	ModRM:reg (r, w)	NA	NA

### Description

Exchanges the first operand (destination operand) with the second operand (source operand), then loads the sum of the two values into the destination operand. The destination operand can be a register or a memory location; the source operand is a register.

In 64-bit mode, the instruction's default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.

### IA-32 Architecture Compatibility

IA-32 processors earlier than the Intel486 processor do not recognize this instruction. If this instruction is used, you should provide an equivalent code sequence that runs on earlier processors.

### Operation

TEMP ← SRC + DEST;  
 SRC ← DEST;  
 DEST ← TEMP;

### Flags Affected

The CF, PF, AF, SF, ZF, and OF flags are set according to the result of the addition, which is stored in the destination operand.

### Protected Mode Exceptions

#GP(0)	If the destination is located in a non-writable segment. If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.



**Real-Address Mode Exceptions**

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

**Virtual-8086 Mode Exceptions**

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

## XORPD—Bitwise Logical XOR of Packed Double Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 57/r XORPD xmm1, xmm2/m128	RM	V/V	SSE2	Return the bitwise logical XOR of packed double-precision floating-point values in xmm1 and xmm2/mem.
VEX.NDS.128.66.0F.WIG 57 /r VXORPD xmm1,xmm2, xmm3/m128	RVM	V/V	AVX	Return the bitwise logical XOR of packed double-precision floating-point values in xmm2 and xmm3/mem.
VEX.NDS.256.66.0F.WIG 57 /r VXORPD ymm1, ymm2, ymm3/m256	RVM	V/V	AVX	Return the bitwise logical XOR of packed double-precision floating-point values in ymm2 and ymm3/mem.
EVEX.NDS.128.66.0F.W1 57 /r VXORPD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	FV	V/V	AVX512VL AVX512DQ	Return the bitwise logical XOR of packed double-precision floating-point values in xmm2 and xmm3/m128/m64bcst subject to writemask k1.
EVEX.NDS.256.66.0F.W1 57 /r VXORPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	FV	V/V	AVX512VL AVX512DQ	Return the bitwise logical XOR of packed double-precision floating-point values in ymm2 and ymm3/m256/m64bcst subject to writemask k1.
EVEX.NDS.512.66.0F.W1 57 /r VXORPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	FV	V/V	AVX512DQ	Return the bitwise logical XOR of packed double-precision floating-point values in zmm2 and zmm3/m512/m64bcst subject to writemask k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
FV	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Performs a bitwise logical XOR of the two, four or eight packed double-precision floating-point values from the first source operand and the second source operand, and stores the result in the destination operand

EVEX.512 encoded version: The first source operand is a ZMM register. The second source operand can be a ZMM register or a vector memory location. The destination operand is a ZMM register conditionally updated with writemask k1.

VEX.256 and EVEX.256 encoded versions: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register (conditionally updated with writemask k1 in case of EVEX). The upper bits (MAX\_VL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 and EVEX.128 encoded versions: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register (conditionally updated with writemask k1 in case of EVEX). The upper bits (MAX\_VL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAX\_VL-1:128) of the corresponding register destination are unmodified.

**Operation****VXORPD (EVEX encoded versions)**

(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) AND (SRC2 \*is memory\*)

THEN DEST[i+63:i] ← SRC1[i+63:i] BITWISE XOR SRC2[63:0];

ELSE DEST[i+63:i] ← SRC1[i+63:i] BITWISE XOR SRC2[i+63:i];

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[i+63:i] = 0

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VXORPD (VEX.256 encoded version)**

DEST[63:0] ← SRC1[63:0] BITWISE XOR SRC2[63:0]

DEST[127:64] ← SRC1[127:64] BITWISE XOR SRC2[127:64]

DEST[191:128] ← SRC1[191:128] BITWISE XOR SRC2[191:128]

DEST[255:192] ← SRC1[255:192] BITWISE XOR SRC2[255:192]

DEST[MAX\_VL-1:256] ← 0

**VXORPD (VEX.128 encoded version)**

DEST[63:0] ← SRC1[63:0] BITWISE XOR SRC2[63:0]

DEST[127:64] ← SRC1[127:64] BITWISE XOR SRC2[127:64]

DEST[MAX\_VL-1:128] ← 0

**XORPD (128-bit Legacy SSE version)**

DEST[63:0] ← DEST[63:0] BITWISE XOR SRC[63:0]

DEST[127:64] ← DEST[127:64] BITWISE XOR SRC[127:64]

DEST[MAX\_VL-1:128] (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**

VXORPD \_\_m512d \_\_mm512\_xor\_pd (\_\_m512d a, \_\_m512d b);

VXORPD \_\_m512d \_\_mm512\_mask\_xor\_pd (\_\_m512d a, \_\_mmask8 m, \_\_m512d b);

VXORPD \_\_m512d \_\_mm512\_maskz\_xor\_pd (\_\_mmask8 m, \_\_m512d a);

VXORPD \_\_m256d \_\_mm256\_xor\_pd (\_\_m256d a, \_\_m256d b);

VXORPD \_\_m256d \_\_mm256\_mask\_xor\_pd (\_\_m256d a, \_\_mmask8 m, \_\_m256d b);

VXORPD \_\_m256d \_\_mm256\_maskz\_xor\_pd (\_\_mmask8 m, \_\_m256d a);

XORPD \_\_m128d \_\_mm\_xor\_pd (\_\_m128d a, \_\_m128d b);

VXORPD \_\_m128d \_\_mm\_mask\_xor\_pd (\_\_m128d a, \_\_mmask8 m, \_\_m128d b);

VXORPD \_\_m128d \_\_mm\_maskz\_xor\_pd (\_\_mmask8 m, \_\_m128d a);

**SIMD Floating-Point Exceptions**

None



### Other Exceptions

Non-EVEX-encoded instructions, see Exceptions Type 4.

EVEX-encoded instructions, see Exceptions Type E4.

## XORPS—Bitwise Logical XOR of Packed Single Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
OF 57 /r XORPS xmm1, xmm2/m128	RM	V/V	SSE	Return the bitwise logical XOR of packed single-precision floating-point values in xmm1 and xmm2/mem.
VEX.NDS.128.OF.WIG 57 /r VXORPS xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Return the bitwise logical XOR of packed single-precision floating-point values in xmm2 and xmm3/mem.
VEX.NDS.256.OF.WIG 57 /r VXORPS ymm1, ymm2, ymm3/m256	RVM	V/V	AVX	Return the bitwise logical XOR of packed single-precision floating-point values in ymm2 and ymm3/mem.
EVEX.NDS.128.OF.W0 57 /r VXORPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	FV	V/V	AVX512VL AVX512DQ	Return the bitwise logical XOR of packed single-precision floating-point values in xmm2 and xmm3/m128/m32bcst subject to writemask k1.
EVEX.NDS.256.OF.W0 57 /r VXORPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	FV	V/V	AVX512VL AVX512DQ	Return the bitwise logical XOR of packed single-precision floating-point values in ymm2 and ymm3/m256/m32bcst subject to writemask k1.
EVEX.NDS.512.OF.W0 57 /r VXORPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	FV	V/V	AVX512DQ	Return the bitwise logical XOR of packed single-precision floating-point values in zmm2 and zmm3/m512/m32bcst subject to writemask k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
FV	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

### Description

Performs a bitwise logical XOR of the four, eight or sixteen packed single-precision floating-point values from the first source operand and the second source operand, and stores the result in the destination operand

EVEX.512 encoded version: The first source operand is a ZMM register. The second source operand can be a ZMM register or a vector memory location. The destination operand is a ZMM register conditionally updated with writemask k1.

VEX.256 and EVEX.256 encoded versions: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register (conditionally updated with writemask k1 in case of EVEX). The upper bits (MAX\_VL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 and EVEX.128 encoded versions: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register (conditionally updated with writemask k1 in case of EVEX). The upper bits (MAX\_VL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAX\_VL-1:128) of the corresponding register destination are unmodified.

**Operation****VXORPS (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\* THEN

IF (EVEX.b == 1) AND (SRC2 \*is memory\*)

THEN DEST[i+31:i] ← SRC1[i+31:i] BITWISE XOR SRC2[31:0];

ELSE DEST[i+31:i] ← SRC1[i+31:i] BITWISE XOR SRC2[i+31:i];

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+31:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[i+31:i] = 0

FI

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**VXORPS (VEX.256 encoded version)**

DEST[31:0] ← SRC1[31:0] BITWISE XOR SRC2[31:0]

DEST[63:32] ← SRC1[63:32] BITWISE XOR SRC2[63:32]

DEST[95:64] ← SRC1[95:64] BITWISE XOR SRC2[95:64]

DEST[127:96] ← SRC1[127:96] BITWISE XOR SRC2[127:96]

DEST[159:128] ← SRC1[159:128] BITWISE XOR SRC2[159:128]

DEST[191:160] ← SRC1[191:160] BITWISE XOR SRC2[191:160]

DEST[223:192] ← SRC1[223:192] BITWISE XOR SRC2[223:192]

DEST[255:224] ← SRC1[255:224] BITWISE XOR SRC2[255:224].

DEST[MAX\_VL-1:256] ← 0

**VXORPS (VEX.128 encoded version)**

DEST[31:0] ← SRC1[31:0] BITWISE XOR SRC2[31:0]

DEST[63:32] ← SRC1[63:32] BITWISE XOR SRC2[63:32]

DEST[95:64] ← SRC1[95:64] BITWISE XOR SRC2[95:64]

DEST[127:96] ← SRC1[127:96] BITWISE XOR SRC2[127:96]

DEST[MAX\_VL-1:128] ← 0

**XORPS (128-bit Legacy SSE version)**

DEST[31:0] ← SRC1[31:0] BITWISE XOR SRC2[31:0]

DEST[63:32] ← SRC1[63:32] BITWISE XOR SRC2[63:32]

DEST[95:64] ← SRC1[95:64] BITWISE XOR SRC2[95:64]

DEST[127:96] ← SRC1[127:96] BITWISE XOR SRC2[127:96]

DEST[MAX\_VL-1:128] (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**

VXORPS \_\_m512 \_\_mm512\_xor\_ps (\_\_m512 a, \_\_m512 b);

VXORPS \_\_m512 \_\_mm512\_mask\_xor\_ps (\_\_m512 a, \_\_mmask16 m, \_\_m512 b);

VXORPS \_\_m512 \_\_mm512\_maskz\_xor\_ps (\_\_mmask16 m, \_\_m512 a);

VXORPS \_\_m256 \_\_mm256\_xor\_ps (\_\_m256 a, \_\_m256 b);

VXORPS \_\_m256 \_\_mm256\_mask\_xor\_ps (\_\_m256 a, \_\_mmask8 m, \_\_m256 b);

VXORPS \_\_m256 \_\_mm256\_maskz\_xor\_ps (\_\_mmask8 m, \_\_m256 a);

XORPS \_\_m128 \_\_mm\_xor\_ps (\_\_m128 a, \_\_m128 b);

VXORPS \_\_m128 \_\_mm\_mask\_xor\_ps (\_\_m128 a, \_\_mmask8 m, \_\_m128 b);

VXORPS \_\_m128 \_\_mm\_maskz\_xor\_ps (\_\_mmask8 m, \_\_m128 a);

### **SIMD Floating-Point Exceptions**

None

### **Other Exceptions**

Non-EVEX-encoded instructions, see Exceptions Type 4.

EVEX-encoded instructions, see Exceptions Type E4.





## 11. Updates to Appendix A, Volume 2D

Change bars show changes to Appendix A of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2D: Instruction Set Reference*.

-----  
Changes include RDPID opcode extension corrected.

Use the opcode tables in this chapter to interpret IA-32 and Intel 64 architecture object code. Instructions are divided into encoding groups:

- 1-byte, 2-byte and 3-byte opcode encodings are used to encode integer, system, MMX technology, SSE/SSE2/SSE3/SSSE3/SSE4, and VMX instructions. Maps for these instructions are given in Table A-2 through Table A-6.
- Escape opcodes (in the format: ESC character, opcode, ModR/M byte) are used for floating-point instructions. The maps for these instructions are provided in Table A-7 through Table A-22.

## NOTE

All blanks in opcode maps are reserved and must not be used. Do not depend on the operation of undefined or blank opcodes.

## A.1 USING OPCODE TABLES

Tables in this appendix list opcodes of instructions (including required instruction prefixes, opcode extensions in associated ModR/M byte). Blank cells in the tables indicate opcodes that are reserved or undefined.

The opcode map tables are organized by hex values of the upper and lower 4 bits of an opcode byte. For 1-byte encodings (Table A-2), use the four high-order bits of an opcode to index a row of the opcode table; use the four low-order bits to index a column of the table. For 2-byte opcodes beginning with 0FH (Table A-3), skip any instruction prefixes, the 0FH byte (0FH may be preceded by 66H, F2H, or F3H) and use the upper and lower 4-bit values of the next opcode byte to index table rows and columns. Similarly, for 3-byte opcodes beginning with 0F38H or 0F3AH (Table A-4), skip any instruction prefixes, 0F38H or 0F3AH and use the upper and lower 4-bit values of the third opcode byte to index table rows and columns. See Section A.2.4, “Opcode Look-up Examples for One, Two, and Three-Byte Opcodes.”

When a ModR/M byte provides opcode extensions, this information qualifies opcode execution. For information on how an opcode extension in the ModR/M byte modifies the opcode map in Table A-2 and Table A-3, see Section A.4.

The escape (ESC) opcode tables for floating point instructions identify the eight high order bits of opcodes at the top of each page. See Section A.5. If the accompanying ModR/M byte is in the range of 00H-BFH, bits 3-5 (the top row of the third table on each page) along with the reg bits of ModR/M determine the opcode. ModR/M bytes outside the range of 00H-BFH are mapped by the bottom two tables on each page of the section.

## A.2 KEY TO ABBREVIATIONS

Operands are identified by a two-character code of the form Zz. The first character, an uppercase letter, specifies the addressing method; the second character, a lowercase letter, specifies the type of operand.

### A.2.1 Codes for Addressing Method

The following abbreviations are used to document addressing methods:

- A Direct address: the instruction has no ModR/M byte; the address of the operand is encoded in the instruction. No base register, index register, or scaling factor can be applied (for example, far JMP (EA)).
- B The VEX.vvvv field of the VEX prefix selects a general purpose register.
- C The reg field of the ModR/M byte selects a control register (for example, MOV (0F20, 0F22)).

- D The reg field of the ModR/M byte selects a debug register (for example, MOV (0F21,0F23)).
- E A ModR/M byte follows the opcode and specifies the operand. The operand is either a general-purpose register or a memory address. If it is a memory address, the address is computed from a segment register and any of the following values: a base register, an index register, a scaling factor, a displacement.
- F EFLAGS/RFLAGS Register.
- G The reg field of the ModR/M byte selects a general register (for example, AX (000)).
- H The VEX.vvvv field of the VEX prefix selects a 128-bit XMM register or a 256-bit YMM register, determined by operand type. For legacy SSE encodings this operand does not exist, changing the instruction to destructive form.
- I Immediate data: the operand value is encoded in subsequent bytes of the instruction.
- J The instruction contains a relative offset to be added to the instruction pointer register (for example, JMP (0E9), LOOP).
- L The upper 4 bits of the 8-bit immediate selects a 128-bit XMM register or a 256-bit YMM register, determined by operand type. (the MSB is ignored in 32-bit mode)
- M The ModR/M byte may refer only to memory (for example, BOUND, LES, LDS, LSS, LFS, LGS, CMPXCHG8B).
- N The R/M field of the ModR/M byte selects a packed-quadword, MMX technology register.
- O The instruction has no ModR/M byte. The offset of the operand is coded as a word or double word (depending on address size attribute) in the instruction. No base register, index register, or scaling factor can be applied (for example, MOV (A0–A3)).
- P The reg field of the ModR/M byte selects a packed quadword MMX technology register.
- Q A ModR/M byte follows the opcode and specifies the operand. The operand is either an MMX technology register or a memory address. If it is a memory address, the address is computed from a segment register and any of the following values: a base register, an index register, a scaling factor, and a displacement.
- R The R/M field of the ModR/M byte may refer only to a general register (for example, MOV (0F20-0F23)).
- S The reg field of the ModR/M byte selects a segment register (for example, MOV (8C,8E)).
- U The R/M field of the ModR/M byte selects a 128-bit XMM register or a 256-bit YMM register, determined by operand type.
- V The reg field of the ModR/M byte selects a 128-bit XMM register or a 256-bit YMM register, determined by operand type.
- W A ModR/M byte follows the opcode and specifies the operand. The operand is either a 128-bit XMM register, a 256-bit YMM register (determined by operand type), or a memory address. If it is a memory address, the address is computed from a segment register and any of the following values: a base register, an index register, a scaling factor, and a displacement.
- X Memory addressed by the DS:rSI register pair (for example, MOVS, CMPS, OUTS, or LODS).
- Y Memory addressed by the ES:rDI register pair (for example, MOVS, CMPS, INS, STOS, or SCAS).

### A.2.2 Codes for Operand Type

The following abbreviations are used to document operand types:

- a Two one-word operands in memory or two double-word operands in memory, depending on operand-size attribute (used only by the BOUND instruction).
- b Byte, regardless of operand-size attribute.
- c Byte or word, depending on operand-size attribute.
- d Doubleword, regardless of operand-size attribute.
- dq Double-quadword, regardless of operand-size attribute.

p	32-bit, 48-bit, or 80-bit pointer, depending on operand-size attribute.
pd	128-bit or 256-bit packed double-precision floating-point data.
pi	Quadword MMX technology register (for example: mm0).
ps	128-bit or 256-bit packed single-precision floating-point data.
q	Quadword, regardless of operand-size attribute.
qq	Quad-Quadword (256-bits), regardless of operand-size attribute.
s	6-byte or 10-byte pseudo-descriptor.
sd	Scalar element of a 128-bit double-precision floating data.
ss	Scalar element of a 128-bit single-precision floating data.
si	Doubleword integer register (for example: eax).
v	Word, doubleword or quadword (in 64-bit mode), depending on operand-size attribute.
w	Word, regardless of operand-size attribute.
x	dq or qq based on the operand-size attribute.
y	Doubleword or quadword (in 64-bit mode), depending on operand-size attribute.
z	Word for 16-bit operand-size or doubleword for 32 or 64-bit operand-size.

### A.2.3 Register Codes

When an opcode requires a specific register as an operand, the register is identified by name (for example, AX, CL, or ESI). The name indicates whether the register is 64, 32, 16, or 8 bits wide.

A register identifier of the form eXX or rXX is used when register width depends on the operand-size attribute. eXX is used when 16 or 32-bit sizes are possible; rXX is used when 16, 32, or 64-bit sizes are possible. For example: eAX indicates that the AX register is used when the operand-size attribute is 16 and the EAX register is used when the operand-size attribute is 32. rAX can indicate AX, EAX or RAX.

When the REX.B bit is used to modify the register specified in the reg field of the opcode, this fact is indicated by adding "/x" to the register name to indicate the additional possibility. For example, rCX/r9 is used to indicate that the register could either be rCX or r9. Note that the size of r9 in this case is determined by the operand size attribute (just as for rCX).

### A.2.4 Opcode Look-up Examples for One, Two, and Three-Byte Opcodes

This section provides examples that demonstrate how opcode maps are used.

#### A.2.4.1 One-Byte Opcode Instructions

The opcode map for 1-byte opcodes is shown in Table A-2. The opcode map for 1-byte opcodes is arranged by row (the least-significant 4 bits of the hexadecimal value) and column (the most-significant 4 bits of the hexadecimal value). Each entry in the table lists one of the following types of opcodes:

- Instruction mnemonics and operand types using the notations listed in Section A.2
- Opcodes used as an instruction prefix

For each entry in the opcode map that corresponds to an instruction, the rules for interpreting the byte following the primary opcode fall into one of the following cases:

- A ModR/M byte is required and is interpreted according to the abbreviations listed in Section A.1 and Chapter 2, "Instruction Format," of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*. Operand types are listed according to notations listed in Section A.2.
- A ModR/M byte is required and includes an opcode extension in the reg field in the ModR/M byte. Use Table A-6 when interpreting the ModR/M byte.

- Use of the ModR/M byte is reserved or undefined. This applies to entries that represent an instruction prefix or entries for instructions without operands that use ModR/M (for example: 60H, PUSH A; 06H, PUSH ES).

#### Example A-1. Look-up Example for 1-Byte Opcodes

Opcode 030500000000H for an ADD instruction is interpreted using the 1-byte opcode map (Table A-2) as follows:

- The first digit (0) of the opcode indicates the table row and the second digit (3) indicates the table column. This locates an opcode for ADD with two operands.
- The first operand (type Gv) indicates a general register that is a word or doubleword depending on the operand-size attribute. The second operand (type Ev) indicates a ModR/M byte follows that specifies whether the operand is a word or doubleword general-purpose register or a memory address.
- The ModR/M byte for this instruction is 05H, indicating that a 32-bit displacement follows (00000000H). The reg/opcode portion of the ModR/M byte (bits 3-5) is 000, indicating the EAX register.

The instruction for this opcode is ADD EAX, mem\_op, and the offset of mem\_op is 00000000H.

Some 1- and 2-byte opcodes point to group numbers (shaded entries in the opcode map table). Group numbers indicate that the instruction uses the reg/opcode bits in the ModR/M byte as an opcode extension (refer to Section A.4).

### A.2.4.2 Two-Byte Opcode Instructions

The two-byte opcode map shown in Table A-3 includes primary opcodes that are either two bytes or three bytes in length. Primary opcodes that are 2 bytes in length begin with an escape opcode 0FH. The upper and lower four bits of the second opcode byte are used to index a particular row and column in Table A-3.

Two-byte opcodes that are 3 bytes in length begin with a mandatory prefix (66H, F2H, or F3H) and the escape opcode (0FH). The upper and lower four bits of the third byte are used to index a particular row and column in Table A-3 (except when the second opcode byte is the 3-byte escape opcodes 38H or 3AH; in this situation refer to Section A.2.4.3).

For each entry in the opcode map, the rules for interpreting the byte following the primary opcode fall into one of the following cases:

- A ModR/M byte is required and is interpreted according to the abbreviations listed in Section A.1 and Chapter 2, "Instruction Format," of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*. The operand types are listed according to notations listed in Section A.2.
- A ModR/M byte is required and includes an opcode extension in the reg field in the ModR/M byte. Use Table A-6 when interpreting the ModR/M byte.
- Use of the ModR/M byte is reserved or undefined. This applies to entries that represent an instruction without operands that are encoded using ModR/M (for example: 0F77H, EMMS).

#### Example A-2. Look-up Example for 2-Byte Opcodes

Look-up opcode 0FA405000000003H for a SHLD instruction using Table A-3.

- The opcode is located in row A, column 4. The location indicates a SHLD instruction with operands Ev, Gv, and Ib. Interpret the operands as follows:
  - Ev: The ModR/M byte follows the opcode to specify a word or doubleword operand.
  - Gv: The reg field of the ModR/M byte selects a general-purpose register.
  - Ib: Immediate data is encoded in the subsequent byte of the instruction.
- The third byte is the ModR/M byte (05H). The mod and opcode/reg fields of ModR/M indicate that a 32-bit displacement is used to locate the first operand in memory and eAX as the second operand.
- The next part of the opcode is the 32-bit displacement for the destination memory operand (00000000H). The last byte stores immediate byte that provides the count of the shift (03H).
- By this breakdown, it has been shown that this opcode represents the instruction: SHLD DS:00000000H, EAX, 3.

### A.2.4.3 Three-Byte Opcode Instructions

The three-byte opcode maps shown in Table A-4 and Table A-5 includes primary opcodes that are either 3 or 4 bytes in length. Primary opcodes that are 3 bytes in length begin with two escape bytes 0F38H or 0F3AH. The upper and lower four bits of the third opcode byte are used to index a particular row and column in Table A-4 or Table A-5.

Three-byte opcodes that are 4 bytes in length begin with a mandatory prefix (66H, F2H, or F3H) and two escape bytes (0F38H or 0F3AH). The upper and lower four bits of the fourth byte are used to index a particular row and column in Table A-4 or Table A-5.

For each entry in the opcode map, the rules for interpreting the byte following the primary opcode fall into the following case:

- A ModR/M byte is required and is interpreted according to the abbreviations listed in A.1 and Chapter 2, "Instruction Format," of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*. The operand types are listed according to notations listed in Section A.2.

#### Example A-3. Look-up Example for 3-Byte Opcodes

Look-up opcode 660F3A0FC108H for a PALIGNR instruction using Table A-5.

- 66H is a prefix and 0F3AH indicate to use Table A-5. The opcode is located in row 0, column F indicating a PALIGNR instruction with operands Vdq, Wdq, and Ib. Interpret the operands as follows:
  - Vdq: The reg field of the ModR/M byte selects a 128-bit XMM register.
  - Wdq: The R/M field of the ModR/M byte selects either a 128-bit XMM register or memory location.
  - Ib: Immediate data is encoded in the subsequent byte of the instruction.
- The next byte is the ModR/M byte (C1H). The reg field indicates that the first operand is XMM0. The mod shows that the R/M field specifies a register and the R/M indicates that the second operand is XMM1.
- The last byte is the immediate byte (08H).
- By this breakdown, it has been shown that this opcode represents the instruction: PALIGNR XMM0, XMM1, 8.

### A.2.4.4 VEX Prefix Instructions

Instructions that include a VEX prefix are organized relative to the 2-byte and 3-byte opcode maps, based on the VEX.mmmmm field encoding of implied 0F, 0F38H, 0F3AH, respectively. Each entry in the opcode map of a VEX-encoded instruction is based on the value of the opcode byte, similar to non-VEX-encoded instructions.

A VEX prefix includes several bit fields that encode implied 66H, F2H, F3H prefix functionality (VEX.pp) and operand size/opcode information (VEX.L). See chapter 4 for details.

Opcode tables A2-A6 include both instructions with a VEX prefix and instructions without a VEX prefix. Many entries are only made once, but represent both the VEX and non-VEX forms of the instruction. If the VEX prefix is present all the operands are valid and the mnemonic is usually prefixed with a "v". If the VEX prefix is not present the VEX.vvvv operand is not available and the prefix "v" is dropped from the mnemonic.

A few instructions exist only in VEX form and these are marked with a superscript "v".

Operand size of VEX prefix instructions can be determined by the operand type code. 128-bit vectors are indicated by 'dq', 256-bit vectors are indicated by 'qq', and instructions with operands supporting either 128 or 256-bit, determined by VEX.L, are indicated by 'x'. For example, the entry "VMOVUPD Vx,Wx" indicates both VEX.L=0 and VEX.L=1 are supported.

### A.2.5 Superscripts Utilized in Opcode Tables

Table A-1 contains notes on particular encodings. These notes are indicated in the following opcode maps by superscripts. Gray cells indicate instruction groupings.

**Table A-1. Superscripts Utilized in Opcode Tables**

Superscript Symbol	Meaning of Symbol
1A	Bits 5, 4, and 3 of ModR/M byte used as an opcode extension (refer to Section A.4, "Opcode Extensions For One-Byte And Two-byte Opcodes").
1B	Use the 0F0B opcode (UD2 instruction) or the 0FB9H opcode when deliberately trying to generate an invalid opcode exception (#UD).
1C	Some instructions use the same two-byte opcode. If the instruction has variations, or the opcode represents different instructions, the ModR/M byte will be used to differentiate the instruction. For the value of the ModR/M byte needed to decode the instruction, see Table A-6.
i64	The instruction is invalid or not encodable in 64-bit mode. 40 through 4F (single-byte INC and DEC) are REX prefix combinations when in 64-bit mode (use FE/FF Grp 4 and 5 for INC and DEC).
o64	Instruction is only available when in 64-bit mode.
d64	When in 64-bit mode, instruction defaults to 64-bit operand size and cannot encode 32-bit operand size.
f64	The operand size is forced to a 64-bit operand size when in 64-bit mode (prefixes that change operand size are ignored for this instruction in 64-bit mode).
v	VEX form only exists. There is no legacy SSE form of the instruction. For Integer GPR instructions it means VEX prefix required.
v1	VEX128 & SSE forms only exist (no VEX256), when can't be inferred from the data size.

### A.3 ONE, TWO, AND THREE-BYTE OPCODE MAPS

See Table A-2 through Table A-5 below. The tables are multiple page presentations. Rows and columns with sequential relationships are placed on facing pages to make look-up tasks easier. Note that table footnotes are not presented on each page. Table footnotes for each table are presented on the last page of the table.

This page was  
intentionally left  
blank.



Table A-2. One-byte Opcode Map: (00H — F7H) \*

	0	1	2	3	4	5	6	7
0	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, lb	rAX, lz	PUSH ES <sup>64</sup>	POP ES <sup>64</sup>
1	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, lb	rAX, lz	PUSH SS <sup>64</sup>	POP SS <sup>64</sup>
2	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, lb	rAX, lz	SEG=ES (Prefix)	DAA <sup>64</sup>
3	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, lb	rAX, lz	SEG=SS (Prefix)	AAA <sup>64</sup>
4	INC <sup>64</sup> general register / REX <sup>64</sup> Prefixes							
	eAX REX	eCX REX.B	eDX REX.X	eBX REX.XB	eSP REX.R	eBP REX.RB	eSI REX.RX	eDI REX.RXB
5	PUSH <sup>64</sup> general register							
	rAX/r8	rCX/r9	rDX/r10	rBX/r11	rSP/r12	rBP/r13	rSI/r14	rDI/r15
6	PUSHA <sup>64</sup> / PUSHAD <sup>64</sup>	POPA <sup>64</sup> / POPAD <sup>64</sup>	BOUND <sup>64</sup> Gv, Ma	ARPL <sup>64</sup> Ew, Gw MOVSD <sup>64</sup> Gv, Ev	SEG=FS (Prefix)	SEG=GS (Prefix)	Operand Size (Prefix)	Address Size (Prefix)
7	Jcc <sup>64</sup> , Jb - Short-displacement jump on condition							
	O	NO	B/NAE/C	NB/AE/NC	Z/E	NZ/NE	BE/NA	NBE/A
8	Immediate Grp 1 <sup>1A</sup>			TEST		XCHG		
	Eb, lb	Ev, lz	Eb, lb <sup>64</sup>	Ev, lb	Eb, Gb	Ev, Gv	Eb, Gb	Ev, Gv
9	NOP PAUSE(F3) XCHG r8, rAX	XCHG word, double-word or quad-word register with rAX						
		rCX/r9	rDX/r10	rBX/r11	rSP/r12	rBP/r13	rSI/r14	rDI/r15
A	MOV							
	AL, Ob	rAX, Ov	Ob, AL	Ov, rAX	MOVS/B Yb, Xb	MOVS/W/D/Q Yv, Xv	CMPS/B Xb, Yb	CMPS/W/D Xv, Yv
B	MOV immediate byte into byte register							
	AL/R8L, lb	CL/R9L, lb	DL/R10L, lb	BL/R11L, lb	AH/R12L, lb	CH/R13L, lb	DH/R14L, lb	BH/R15L, lb
C	Shift Grp 2 <sup>1A</sup>		near RET <sup>64</sup> lw	near RET <sup>64</sup>	LES <sup>64</sup> Gz, Mp VEX+2byte	LDS <sup>64</sup> Gz, Mp VEX+1byte	Grp 11 <sup>1A</sup> - MOV	
	Eb, lb	Ev, lb					Eb, lb	Ev, lz
D	Shift Grp 2 <sup>1A</sup>				AAM <sup>64</sup> lb		AAD <sup>64</sup> lb	
	Eb, 1	Ev, 1	Eb, CL	Ev, CL				XLAT/ XLATB
E	LOOPNE <sup>64</sup> / LOOPNZ <sup>64</sup> Jb	LOOPE <sup>64</sup> / LOOPZ <sup>64</sup> Jb	LOOP <sup>64</sup> Jb	Jrcxz <sup>64</sup> / Jb	IN		OUT	
					AL, lb	eAX, lb	lb, AL	lb, eAX
F	LOCK (Prefix)		REPNE XACQUIRE (Prefix)	REP/REPE XRELEASE (Prefix)	HLT	CMC	Unary Grp 3 <sup>1A</sup>	
							Eb	Ev

Table A-2. One-byte Opcode Map: (08H – FFH) \*

	8	9	A	B	C	D	E	F		
0	OR						PUSH CS <sup>i64</sup>	2-byte escape (Table A-3)		
	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, lb	rAX, lz				
1	SBB						PUSH DS <sup>i64</sup>	POP DS <sup>i64</sup>		
	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, lb	rAX, lz				
2	SUB						SEG=CS (Prefix)	DAS <sup>i64</sup>		
	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, lb	rAX, lz				
3	CMP						SEG=DS (Prefix)	AAS <sup>i64</sup>		
	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, lb	rAX, lz				
4	DEC <sup>i64</sup> general register / REX <sup>064</sup> Prefixes									
	eAX REX.W	eCX REX.WB	eDX REX.WX	eBX REX.WXB	eSP REX.WR	eBP REX.WRB	eSI REX.WRX	eDI REX.WRXB		
5	POP <sup>d64</sup> into general register									
	rAX/r8	rCX/r9	rDX/r10	rBX/r11	rSP/r12	rBP/r13	rSI/r14	rDI/r15		
6	PUSH <sup>d64</sup> lz	IMUL Gv, Ev, lz	PUSH <sup>d64</sup> lb	IMUL Gv, Ev, lb	INS/ INSB Yb, DX	INS/ INSW/ INSD Yz, DX	OUTS/ OUTSB DX, Xb	OUTS/ OUTSW/ OUTSD DX, Xz		
7	Jcc <sup>i64</sup> , Jb- Short displacement jump on condition									
	S	NS	P/PE	NP/PO	L/NGE	NL/GE	LE/NG	NLE/G		
8	MOV						MOV Ev, Sw	LEA Gv, M	MOV Sw, Ew	Grp 1A <sup>1A</sup> POP <sup>d64</sup> Ev
	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev						
9	CBW/ CWDE/ CDQE	CWD/ CDQ/ CQO	far CALL <sup>i64</sup> Ap	FWAIT/ WAIT	PUSHF/D/Q <sup>d64</sup> / Fv	POPF/D/Q <sup>d64</sup> / Fv	SAHF	LAHF		
A	TEST		STOS/B Yb, AL	STOS/W/D/Q Yv, rAX	LODS/B AL, Xb	LODS/W/D/Q rAX, Xv	SCAS/B AL, Yb	SCAS/W/D/Q rAX, Yv		
	AL, lb	rAX, lz								
B	MOV immediate word or double into word, double, or quad register									
	rAX/r8, lv	rCX/r9, lv	rDX/r10, lv	rBX/r11, lv	rSP/r12, lv	rBP/r13, lv	rSI/r14, lv	rDI/r15, lv		
C	ENTER lw, lb	LEAVE <sup>d64</sup>	far RET lw	far RET	INT 3	INT lb	INTO <sup>i64</sup>	IRET/D/Q		
D	ESC (Escape to coprocessor instruction set)									
E	near CALL <sup>f64</sup> Jz	near <sup>f64</sup> Jz	JMP far <sup>i64</sup> Ap	short <sup>f64</sup> Jb	AL, DX	eAX, DX	DX, AL	DX, eAX		
F	CLC	STC	CLI	STI	CLD	STD	INC/DEC Grp 4 <sup>1A</sup>	INC/DEC Grp 5 <sup>1A</sup>		

NOTES:

\* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

Table A-3. Two-byte Opcode Map: 00H – 77H (First Byte is 0FH) \*

	pxf	0	1	2	3	4	5	6	7
0		Grp 6 <sup>1A</sup>	Grp 7 <sup>1A</sup>	LAR Gv, Ew	LSL Gv, Ew		SYSCALL <sup>064</sup>	CLTS	SYSRET <sup>064</sup>
1		vmovups Vps, Wps	vmovups Wps, Vps	vmovlps Vq, Hq, Mq vmovhlps Vq, Hq, Uq	vmovlps Mq, Vq	vunpcklps Vx, Hx, Wx	vunpckhps Vx, Hx, Wx	vmovhps <sup>v1</sup> Vdq, Hq, Mq vmovhlps Vdq, Hq, Uq	vmovhps <sup>v1</sup> Mq, Vq
	66	vmovupd Vpd, Wpd	vmovupd Wpd, Vpd	vmovlpd Vq, Hq, Mq	vmovlpd Mq, Vq	vunpcklpd Vx, Hx, Wx	vunpckhpd Vx, Hx, Wx	vmovhpd <sup>v1</sup> Vdq, Hq, Mq	vmovhpd <sup>v1</sup> Mq, Vq
	F3	vmovss Vx, Hx, Wss	vmovss Wss, Hx, Vss	vmovsldup Vx, Wx				vmovshdup Vx, Wx	
	F2	vmovsd Vx, Hx, Wsd	vmovsd Wsd, Hx, Vsd	vmovddup Vx, Wx					
2		MOV Rd, Cd	MOV Rd, Dd	MOV Cd, Rd	MOV Dd, Rd				
3		WRMSR	RDTSC	RDMSR	RDPMC	SYSENTER	SYSEXIT		GETSEC
4		CMOVcc, (Gv, Ev) - Conditional Move							
		O	NO	B/C/NAE	AE/NB/NC	E/Z	NE/NZ	BE/NA	A/NBE
5		vmovmskps Gy, Ups	vsqrtps Vps, Wps	vrsqrtps Vps, Wps	vrcpps Vps, Wps	vandps Vps, Hps, Wps	vandnps Vps, Hps, Wps	vorps Vps, Hps, Wps	vxorps Vps, Hps, Wps
	66	vmovmskpd Gy, Upd	vsqrtpd Vpd, Wpd			vandpd Vpd, Hpd, Wpd	vandnpd Vpd, Hpd, Wpd	vorpd Vpd, Hpd, Wpd	vxorpd Vpd, Hpd, Wpd
	F3		vsqrtss Vss, Hss, Wss	vrsqrtss Vss, Hss, Wss	vrcpss Vss, Hss, Wss				
	F2		vsqrtsd Vsd, Hsd, Wsd						
6		punpcklbw Pq, Qd	punpcklwd Pq, Qd	punpckldq Pq, Qd	packsswb Pq, Qq	pcmpgtb Pq, Qq	pcmpgtw Pq, Qq	pcmpgtd Pq, Qq	packuswb Pq, Qq
	66	vpunpcklbw Vx, Hx, Wx	vpunpcklwd Vx, Hx, Wx	vpunpckldq Vx, Hx, Wx	vpacksswb Vx, Hx, Wx	vpcmpgtb Vx, Hx, Wx	vpcmpgtw Vx, Hx, Wx	vpcmpgtd Vx, Hx, Wx	vpackuswb Vx, Hx, Wx
	F3								
7		pshufw Pq, Qq, Ib	(Grp 12 <sup>1A</sup> )	(Grp 13 <sup>1A</sup> )	(Grp 14 <sup>1A</sup> )	pcmpeqb Pq, Qq	pcmpeqw Pq, Qq	pcmpeqd Pq, Qq	emms vzeroupper <sup>v</sup> vzeroall <sup>v</sup>
	66	vpshufd Vx, Wx, Ib				vpcmpeqb Vx, Hx, Wx	vpcmpeqw Vx, Hx, Wx	vpcmpeqd Vx, Hx, Wx	
	F3	vpshufhw Vx, Wx, Ib							
	F2	vpshufw Vx, Wx, Ib							

Table A-3. Two-byte Opcode Map: 08H – 7FH (First Byte is 0FH) \*

	px	8	9	A	B	C	D	E	F
0		INVD	WBINVD		2-byte Illegal Opcodes UD2 <sup>1B</sup>		prefetchw(/1) Ev		
1		Prefetch <sup>1C</sup> (Grp 16 <sup>1A</sup> )							NOP /0 Ev
2		vmovaps Vps, Wps	vmovaps Wps, Vps	cvtpi2ps Vps, Qpi	vmovntps Mps, Vps	cvttps2pi Ppi, Wps	cvtps2pi Ppi, Wps	vucomiss Vss, Wss	vcomiss Vss, Wss
	66	vmovapd Vpd, Wpd	vmovapd Wpd, Vpd	cvtpi2pd Vpd, Qpi	vmovntpd Mpd, Vpd	cvttpd2pi Ppi, Wpd	cvtpd2pi Qpi, Wpd	vucomisd Vsd, Wsd	vcomisd Vsd, Wsd
	F3			vcvtsi2ss Vss, Hss, Ey		vcvttss2si Gy, Wss	vcvts2si Gy, Wss		
	F2			vcvtsi2sd Vsd, Hsd, Ey		vcvtt2sd2si Gy, Wsd	vcvtsd2si Gy, Wsd		
3		3-byte escape (Table A-4)		3-byte escape (Table A-5)					
4		CMOVcc(Gv, Ev) - Conditional Move							
		S	NS	P/PE	NP/PO	L/NGE	NL/GE	LE/NG	NLE/G
5		vaddps Vps, Hps, Wps	vmulps Vps, Hps, Wps	vcvtps2pd Vpd, Wps	vcvtdq2ps Vps, Wdq	vsubps Vps, Hps, Wps	vminps Vps, Hps, Wps	vdivps Vps, Hps, Wps	vmaxps Vps, Hps, Wps
	66	vaddpd Vpd, Hpd, Wpd	vmulpd Vpd, Hpd, Wpd	vcvtpd2ps Vps, Wpd	vcvtps2dq Vdq, Wps	vsubpd Vpd, Hpd, Wpd	vminpd Vpd, Hpd, Wpd	vdivpd Vpd, Hpd, Wpd	vmaxpd Vpd, Hpd, Wpd
	F3	vaddss Vss, Hss, Wss	vmulss Vss, Hss, Wss	vcvts2sd Vsd, Hx, Wss	vcvtps2dq Vdq, Wps	vsubss Vss, Hss, Wss	vminss Vss, Hss, Wss	vdivss Vss, Hss, Wss	vmaxss Vss, Hss, Wss
	F2	vaddsd Vsd, Hsd, Wsd	vmulsd Vsd, Hsd, Wsd	vcvtsd2ss Vss, Hx, Wsd		vsubsd Vsd, Hsd, Wsd	vminsd Vsd, Hsd, Wsd	vdivsd Vsd, Hsd, Wsd	vmaxsd Vsd, Hsd, Wsd
6		punpckhbw Pq, Qd	punpckhwd Pq, Qd	punpckhdq Pq, Qd	packssdw Pq, Qd			movd/q Pd, Ey	movq Pq, Qq
	66	vpunpckhbw Vx, Hx, Wx	vpunpckhwd Vx, Hx, Wx	vpunpckhdq Vx, Hx, Wx	vpackssdw Vx, Hx, Wx	vpunpcklqdq Vx, Hx, Wx	vpunpckhqdq Vx, Hx, Wx	vmovd/q Vy, Ey	vmovdqa Vx, Wx
	F3								vmovdqu Vx, Wx
7		VMREAD Ey, Gy	VMWRITE Gy, Ey					movd/q Ey, Pd	movq Qq, Pq
	66					vhaddpd Vpd, Hpd, Wpd	vhsbpd Vpd, Hpd, Wpd	vmovd/q Ey, Vy	vmovdqa Wx, Vx
	F3							vmovq Vq, Wq	vmovdqu Wx, Vx
	F2					vhaddps Vps, Hps, Wps	vhsb2ps Vps, Hps, Wps		

Table A-3. Two-byte Opcode Map: 80H – F7H (First Byte is 0FH) \*

	pxf	0	1	2	3	4	5	6	7
8		Jcc <sup>64</sup> , Jz - Long-displacement jump on condition							
		O	NO	B/CNAE	AE/NB/NC	E/Z	NE/NZ	BE/NA	A/NBE
9		SETcc, Eb - Byte Set on condition							
		O	NO	B/CNAE	AE/NB/NC	E/Z	NE/NZ	BE/NA	A/NBE
A		PUSH <sup>d64</sup> FS	POP <sup>d64</sup> FS	CPUID	BT Ev, Gv	SHLD Ev, Gv, Ib	SHLD Ev, Gv, CL		
B		CMPXCHG Eb, Gb      Ev, Gv		LSS Gv, Mp	BTR Ev, Gv	LFS Gv, Mp	LGS Gv, Mp	MOVZX Gv, Eb      Gv, Ew	
C		XADD Eb, Gb	XADD Ev, Gv	vcmps Vps,Hps,Wps,Ib	movnti My, Gy	pinsrw Pq,Ry/Mw,Ib	pextrw Gd, Nq, Ib	vshufps Vps,Hps,Wps,Ib	Grp 9 <sup>1A</sup>
	66			vcmpd Vpd,Hpd,Wpd,Ib		vpinsrw Vdq,Hdq,Ry/Mw,Ib	vpextrw Gd, Udq, Ib	vshufpd Vpd,Hpd,Wpd,Ib	
	F3			vcmpss Vss,Hss,Wss,Ib					
	F2			vcmps Vsd,Hsd,Wsd,Ib					
D			psrlw Pq, Qq	psrld Pq, Qq	psrlq Pq, Qq	paddq Pq, Qq	pmullw Pq, Qq		pmovmskb Gd, Nq
	66	vaddsubpd Vpd, Hpd, Wpd	vpsrlw Vx, Hx, Wx	vpsrld Vx, Hx, Wx	vpsrlq Vx, Hx, Wx	vpaddq Vx, Hx, Wx	vpmullw Vx, Hx, Wx	vmovq Wq, Vq	vpmovmskb Gd, Ux
	F3							movq2dq Vdq, Nq	
	F2	vaddsubps Vps, Hps, Wps						movdq2q Pq, Uq	
E		pavgb Pq, Qq	psraw Pq, Qq	psrad Pq, Qq	pavgw Pq, Qq	pmulhw Pq, Qq	pmulhw Pq, Qq		movntq Mq, Pq
	66	vpavgb Vx, Hx, Wx	vpsraw Vx, Hx, Wx	vpsrad Vx, Hx, Wx	vpavgw Vx, Hx, Wx	vpmulhw Vx, Hx, Wx	vpmulhw Vx, Hx, Wx	vcvtpd2dq Vx, Wpd	vmovntdq Mx, Vx
	F3							vcvtdq2pd Vx, Wpd	
	F2							vcvtpd2dq Vx, Wpd	
F			psllw Pq, Qq	pslld Pq, Qq	psllq Pq, Qq	pmuludq Pq, Qq	pmaddwd Pq, Qq	psadbw Pq, Qq	maskmovq Pq, Nq
	66		vpsllw Vx, Hx, Wx	vpslld Vx, Hx, Wx	vpsllq Vx, Hx, Wx	vpmuludq Vx, Hx, Wx	vpmaddwd Vx, Hx, Wx	vpsadbw Vx, Hx, Wx	vmaskmovdqu Vdq, Udq
	F2	vlddqu Vx, Mx							

Table A-3. Two-byte Opcode Map: 88H — FFH (First Byte is 0FH) \*

	pxf	8	9	A	B	C	D	E	F
8		Jcc <sup>64</sup> , Jz - Long-displacement jump on condition							
		S	NS	P/PE	NP/PO	L/NGE	NL/GE	LE/NG	NLE/G
9		SETcc, Eb - Byte Set on condition							
		S	NS	P/PE	NP/PO	L/NGE	NL/GE	LE/NG	NLE/G
A		PUSH <sup>d64</sup> GS	POP <sup>d64</sup> GS	RSM	BTS Ev, Gv	SHRD Ev, Gv, Ib	SHRD Ev, Gv, CL	(Grp 15 <sup>1A</sup> ) <sup>1C</sup>	IMUL Gv, Ev
B		JMPE (reserved for emulator on IPF)	Grp 10 <sup>1A</sup> Invalid Opcode <sup>1B</sup>	Grp 8 <sup>1A</sup> Ev, Ib	BTC Ev, Gv	BSF Gv, Ev	BSR Gv, Ev	MOVSB Gv, Eb	Gv, Ew
	F3	POPCNT Gv, Ev				TZCNT Gv, Ev	LZCNT Gv, Ev		
C		BSWAP							
		RAX/EAX/ R8/R8D	RCX/ECX/ R9/R9D	RDX/EDX/ R10/R10D	RBX/EBX/ R11/R11D	RSP/ESP/ R12/R12D	RBP/EBP/ R13/R13D	RSI/ESI/ R14/R14D	RDI/EDI/ R15/R15D
		psubusb Pq, Qq	psubusw Pq, Qq	pminub Pq, Qq	pand Pq, Qq	paddusb Pq, Qq	paddusw Pq, Qq	pmaxub Pq, Qq	pandn Pq, Qq
	66	vpsubusb Vx, Hx, Wx	vpsubusw Vx, Hx, Wx	vpminub Vx, Hx, Wx	vpand Vx, Hx, Wx	vpaddusb Vx, Hx, Wx	vpaddusw Vx, Hx, Wx	vpmaxub Vx, Hx, Wx	vpandn Vx, Hx, Wx
	F3								
	F2								
E		psubsb Pq, Qq	psubsw Pq, Qq	pminsw Pq, Qq	por Pq, Qq	paddsb Pq, Qq	paddsw Pq, Qq	pmaxsw Pq, Qq	pxor Pq, Qq
	66	vpsubsb Vx, Hx, Wx	vpsubsw Vx, Hx, Wx	vpminsw Vx, Hx, Wx	vpor Vx, Hx, Wx	vpaddsb Vx, Hx, Wx	vpaddsw Vx, Hx, Wx	vpmaxsw Vx, Hx, Wx	vpxor Vx, Hx, Wx
	F3								
	F2								
F		psubb Pq, Qq	psubw Pq, Qq	psubd Pq, Qq	psubq Pq, Qq	paddb Pq, Qq	paddw Pq, Qq	paddd Pq, Qq	
	66	vpsubb Vx, Hx, Wx	vpsubw Vx, Hx, Wx	vpsubd Vx, Hx, Wx	vpsubq Vx, Hx, Wx	vpaddb Vx, Hx, Wx	vpaddw Vx, Hx, Wx	vpaddd Vx, Hx, Wx	
	F2								

NOTES:

\* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

**Table A-4. Three-byte Opcode Map: 00H – F7H (First Two Bytes are 0F 38H) \***

	pxf	0	1	2	3	4	5	6	7
0		pshufb Pq, Qq	phaddw Pq, Qq	phadd Pq, Qq	phaddsw Pq, Qq	pmaddubsw Pq, Qq	phsubw Pq, Qq	phsubd Pq, Qq	phsubsw Pq, Qq
	66	vpshufb Vx, Hx, Wx	vphaddw Vx, Hx, Wx	vphadd Vx, Hx, Wx	vphaddsw Vx, Hx, Wx	vpaddubsw Vx, Hx, Wx	vphsubw Vx, Hx, Wx	vphsubd Vx, Hx, Wx	vphsubsw Vx, Hx, Wx
1	66	pblendvb Vdq, Wdq			vcvtph2ps <sup>v</sup> Vx, Wx, Ib	blendvps Vdq, Wdq	blendvpd Vdq, Wdq	vpermps <sup>v</sup> Vqq, Hqq, Wqq	vptest Vx, Wx
2	66	vpmovsxbw Vx, Ux/Mq	vpmovsxbd Vx, Ux/Md	vpmovsxbq Vx, Ux/Mw	vpmovsxwd Vx, Ux/Mq	vpmovsxwq Vx, Ux/Md	vpmovsxdq Vx, Ux/Mq		
3	66	vpmovzxbw Vx, Ux/Mq	vpmovzxbd Vx, Ux/Md	vpmovzxbq Vx, Ux/Mw	vpmovzxwd Vx, Ux/Mq	vpmovzxwq Vx, Ux/Md	vpmovzxdq Vx, Ux/Mq	vpermd <sup>v</sup> Vqq, Hqq, Wqq	vpcmpgtq Vx, Hx, Wx
4	66	vpmulld Vx, Hx, Wx	vphminposuw Vdq, Wdq				vpsrlvd/q <sup>v</sup> Vx, Hx, Wx	vpsravd <sup>v</sup> Vx, Hx, Wx	vpslvd/q <sup>v</sup> Vx, Hx, Wx
5									
6									
7									
8	66	INVEPT Gy, Mdq	INVVPID Gy, Mdq	INVPCID Gy, Mdq					
9	66	vgatherdd/q <sup>v</sup> Vx,Hx,Wx	vgatherqd/q <sup>v</sup> Vx,Hx,Wx	vgatherdps/d <sup>v</sup> Vx,Hx,Wx	vgatherqps/d <sup>v</sup> Vx,Hx,Wx			vfmaddsub132ps/d <sup>v</sup> Vx,Hx,Wx	vfmsubadd132ps/d <sup>v</sup> Vx,Hx,Wx
A	66							vfmaddsub213ps/d <sup>v</sup> Vx,Hx,Wx	vfmsubadd213ps/d <sup>v</sup> Vx,Hx,Wx
B	66							vfmaddsub231ps/d <sup>v</sup> Vx,Hx,Wx	vfmsubadd231ps/d <sup>v</sup> Vx,Hx,Wx
C									
D									
E									
F		MOVBE Gy, My	MOVBE My, Gy	ANDN <sup>v</sup> Gy, By, Ey	Grp 17 <sup>1A</sup>		BZHI <sup>v</sup> Gy, Ey, By		BEXTR <sup>v</sup> Gy, Ey, By
	66	MOVBE Gw, Mw	MOVBE Mw, Gw					ADCX Gy, Ey	SHLX <sup>v</sup> Gy, Ey, By
	F3						PEXT <sup>v</sup> Gy, By, Ey	ADOX Gy, Ey	SARX <sup>v</sup> Gy, Ey, By
	F2	CRC32 Gd, Eb	CRC32 Gd, Ey				PDEP <sup>v</sup> Gy, By, Ey	MULX <sup>v</sup> By,Gy,rDX,Ey	SHRX <sup>v</sup> Gy, Ey, By
	66 & F2	CRC32 Gd, Eb	CRC32 Gd, Ew						

**Table A-4. Three-byte Opcode Map: 08H – FFH (First Two Bytes are 0F 38H) \***

	pxf	8	9	A	B	C	D	E	F
0		psignb Pq, Qq	psignw Pq, Qq	psignd Pq, Qq	pmulhrsw Pq, Qq				
	66	vpsignb Vx, Hx, Wx	vpsignw Vx, Hx, Wx	vpsignd Vx, Hx, Wx	vpmulhrsw Vx, Hx, Wx	vpermilps <sup>v</sup> Vx,Hx,Wx	vpermilpd <sup>v</sup> Vx,Hx,Wx	vtestps <sup>v</sup> Vx, Wx	vtestpd <sup>v</sup> Vx, Wx
1						pabsb Pq, Qq	pabsw Pq, Qq	pabsd Pq, Qq	
	66	vbroadcastss <sup>v</sup> Vx, Wd	vbroadcastsd <sup>v</sup> Vqq, Wq	vbroadcastf128 <sup>v</sup> Vqq, Mdq		vpabsb Vx, Wx	vpabsw Vx, Wx	vpabsd Vx, Wx	
2		vpmuldq Vx, Hx, Wx	vpcmpqq Vx, Hx, Wx	vmovntdqa Vx, Mx	vpackusdw Vx, Hx, Wx	vmaskmovps <sup>v</sup> Vx,Hx,Mx	vmaskmovpd <sup>v</sup> Vx,Hx,Mx	vmaskmovps <sup>v</sup> Mx,Hx,Vx	vmaskmovpd <sup>v</sup> Mx,Hx,Vx
	66	vpmins b Vx, Hx, Wx	vpmins d Vx, Hx, Wx	vpminuw Vx, Hx, Wx	vpminud Vx, Hx, Wx	vpmaxsb Vx, Hx, Wx	vpmaxsd Vx, Hx, Wx	vpmaxuw Vx, Hx, Wx	vpmaxud Vx, Hx, Wx
3									
	66	vpbroadcast <sup>v</sup> Vx, Wx	vpbroadcastq <sup>v</sup> Vx, Wx	vpbroadcasti128 <sup>v</sup> Vqq, Mdq					
4									
	66	vpbroadcastb <sup>v</sup> Vx, Wx	vpbroadcastw <sup>v</sup> Vx, Wx						
5									
	66					vpmaskmovd/q <sup>v</sup> Vx,Hx,Mx		vpmaskmovd/q <sup>v</sup> Mx,Vx,Hx	
6									
	66	vfmadd132ps/d <sup>v</sup> Vx, Hx, Wx	vfmadd132ss/d <sup>v</sup> Vx, Hx, Wx	vfmsub132ps/d <sup>v</sup> Vx, Hx, Wx	vfmsub132ss/d <sup>v</sup> Vx, Hx, Wx	vfnmadd132ps/d <sup>v</sup> Vx, Hx, Wx	vfnmadd132ss/d <sup>v</sup> Vx, Hx, Wx	vfnmsub132ps/d <sup>v</sup> Vx, Hx, Wx	vfnmsub132ss/d <sup>v</sup> Vx, Hx, Wx
7									
	66	vfmadd213ps/d <sup>v</sup> Vx, Hx, Wx	vfmadd213ss/d <sup>v</sup> Vx, Hx, Wx	vfmsub213ps/d <sup>v</sup> Vx, Hx, Wx	vfmsub213ss/d <sup>v</sup> Vx, Hx, Wx	vfnmadd213ps/d <sup>v</sup> Vx, Hx, Wx	vfnmadd213ss/d <sup>v</sup> Vx, Hx, Wx	vfnmsub213ps/d <sup>v</sup> Vx, Hx, Wx	vfnmsub213ss/d <sup>v</sup> Vx, Hx, Wx
8									
	66	vfmadd231ps/d <sup>v</sup> Vx, Hx, Wx	vfmadd231ss/d <sup>v</sup> Vx, Hx, Wx	vfmsub231ps/d <sup>v</sup> Vx, Hx, Wx	vfmsub231ss/d <sup>v</sup> Vx, Hx, Wx	vfnmadd231ps/d <sup>v</sup> Vx, Hx, Wx	vfnmadd231ss/d <sup>v</sup> Vx, Hx, Wx	vfnmsub231ps/d <sup>v</sup> Vx, Hx, Wx	vfnmsub231ss/d <sup>v</sup> Vx, Hx, Wx
9									
	66	sha1nexte Vdq,Wdq	sha1msg1 Vdq,Wdq	sha1msg2 Vdq,Wdq	sha256rds2 Vdq,Wdq	sha256msg1 Vdq,Wdq	sha256msg2 Vdq,Wdq		
A									
	66				VAESIMC Vdq, Wdq	VAEENC Vdq,Hdq,Wdq	VAEENCLAST Vdq,Hdq,Wdq	VAEDEC Vdq,Hdq,Wdq	VAEDECLAST Vdq,Hdq,Wdq
B									
	66								
C									
	66								
D									
	66								
E									
	66								
F									
	66 & F2								

**NOTES:**

\* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.



**Table A-5. Three-byte Opcode Map: 00H – F7H (First two bytes are 0F 3AH) \***

	px	0	1	2	3	4	5	6	7
0	66	vpermq <sup>v</sup> Vqq, Wqq, lb	vpermpd <sup>v</sup> Vqq, Wqq, lb	vblendd <sup>v</sup> Vx, Hx, Wx, lb		vpermilps <sup>y</sup> Vx, Wx, lb	vpermilpd <sup>y</sup> Vx, Wx, lb	vperm2f128 <sup>y</sup> Vqq, Hqq, Wqq, lb	
1	66					vpextrb Rd/Mb, Vdq, lb	vpextrw Rd/Mw, Vdq, lb	vpextrd/q Ey, Vdq, lb	vextractps Ed, Vdq, lb
2	66	vpinsrb Vdq, Hdq, Ry/Mb, lb	vinstrps Vdq, Hdq, Udq/Md, lb	vpinsrd/q Vdq, Hdq, Ey, lb					
3									
4	66	vdpps Vx, Hx, Wx, lb	vdppd Vdq, Hdq, Wdq, lb	vmpsadbw Vx, Hx, Wx, lb		vpcimulqdq Vdq, Hdq, Wdq, lb		vperm2i128 <sup>y</sup> Vqq, Hqq, Wqq, lb	
5									
6	66	vpcmpstrm Vdq, Wdq, lb	vpcmpstri Vdq, Wdq, lb	vpcmpstrm Vdq, Wdq, lb	vpcmpstri Vdq, Wdq, lb				
7									
8									
9									
A									
B									
C									
D									
E									
F	F2	RORX <sup>y</sup> Gy, Ey, lb							

**Table A-5. Three-byte Opcode Map: 08H – FFH (First Two Bytes are 0F 3AH) \***

	px	8	9	A	B	C	D	E	F
0									palgnr Pq, Qq, Ib
	66	vroundps Vx, Wx, Ib	vroundpd Vx, Wx, Ib	vroundss Vss, Wss, Ib	vroundsd Vsd, Wsd, Ib	vblendps Vx, Hx, Wx, Ib	vblendpd Vx, Hx, Wx, Ib	vpblendw Vx, Hx, Wx, Ib	vpalgnr Vx, Hx, Wx, Ib
1	66	vinserff128 <sup>v</sup> Vqq, Hqq, Wqq, Ib	vextractf128 <sup>v</sup> Wdq, Vqq, Ib				vcvtps2ph <sup>y</sup> Wx, Vx, Ib		
2									
3	66	vinserfi128 <sup>v</sup> Vqq, Hqq, Wqq, Ib	vextractfi128 <sup>v</sup> Wdq, Vqq, Ib						
4	66			vblendvps <sup>v</sup> Vx, Hx, Wx, Lx	vblendvpd <sup>v</sup> Vx, Hx, Wx, Lx	vpblendvb <sup>v</sup> Vx, Hx, Wx, Lx			
5									
6									
7									
8									
9									
A									
B									
C						sha1m4s4 Vdq, Wdq, Ib			
D	66								VAESKEYGEN Vdq, Wdq, Ib
E									
F									

**NOTES:**

\* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

## A.4 OPCODE EXTENSIONS FOR ONE-BYTE AND TWO-BYTE OPCODES

Some 1-byte and 2-byte opcodes use bits 3-5 of the ModR/M byte (the nnn field in Figure A-1) as an extension of the opcode.

mod	nnn	R/M
-----	-----	-----

Figure A-1. ModR/M Byte nnn Field (Bits 5, 4, and 3)

Opcodes that have opcode extensions are indicated in Table A-6 and organized by group number. Group numbers (from 1 to 16, second column) provide a table entry point. The encoding for the r/m field for each instruction can be established using the third column of the table.

### A.4.1 Opcode Look-up Examples Using Opcode Extensions

An Example is provided below.

#### Example A-4. Interpreting an ADD Instruction

An ADD instruction with a 1-byte opcode of 80H is a Group 1 instruction:

- Table A-6 indicates that the opcode extension field encoded in the ModR/M byte for this instruction is 000B.
- The r/m field can be encoded to access a register (11B) or a memory address using a specified addressing mode (for example: mem = 00B, 01B, 10B).

#### Example A-5. Looking Up 0F01C3H

Look up opcode 0F01C3 for a VMRESUME instruction by using Table A-2, Table A-3 and Table A-6:

- 0F tells us that this instruction is in the 2-byte opcode map.
- 01 (row 0, column 1 in Table A-3) reveals that this opcode is in Group 7 of Table A-6.
- C3 is the ModR/M byte. The first two bits of C3 are 11B. This tells us to look at the second of the Group 7 rows in Table A-6.
- The Op/Reg bits [5,4,3] are 000B. This tells us to look in the 000 column for Group 7.
- Finally, the R/M bits [2,1,0] are 011B. This identifies the opcode as the VMRESUME instruction.

### A.4.2 Opcode Extension Tables

See Table A-6 below.

**Table A-6. Opcode Extensions for One- and Two-byte Opcodes by Group Number \***

Opcode	Group	Mod 7,6	pfx	Encoding of Bits 5,4,3 of the ModR/M Byte (bits 2,1,0 in parenthesis)								
				000	001	010	011	100	101	110	111	
80-83	1	mem, 11B		ADD	OR	ADC	SBB	AND	SUB	XOR	CMP	
8F	1A	mem, 11B		POP								
C0,C1 reg, imm D0, D1 reg, 1 D2, D3 reg, CL	2	mem, 11B		ROL	ROR	RCL	RCR	SHL/SAL	SHR		SAR	
F6, F7	3	mem, 11B		TEST lb/lz		NOT	NEG	MUL AL/rAX	IMUL AL/rAX	DIV AL/rAX	IDIV AL/rAX	
FE	4	mem, 11B		INC Eb	DEC Eb							
FF	5	mem, 11B		INC Ev	DEC Ev	near CALL <sup>164</sup> Ev	far CALL Ep	near JMP <sup>164</sup> Ev	far JMP Mp	PUSH <sup>d64</sup> Ev		
0F 00	6	mem, 11B		SLDT Rv/Mw	STR Rv/Mw	LLDT Ew	LTR Ew	VERR Ew	VERW Ew			
0F 01	7	mem		SGDT Ms	SIDT Ms	LGDT Ms	LIDT Ms	SMSW Mw/Rv		LMSW Ew	INVLPG Mb	
		11B		VMCALL (001) VMLAUNCH (010) VMRESUME (011) VMXOFF (100)	MONITOR (000) MWAIT (001) CLAC (010) STAC (011) ENCLS (111)	XGETBV (000) XSETBV (001)  VMFUNC (100) XEND (101) XTEST (110) ENCLU(111)				SWAPGS <sup>064</sup> (000) RDTSCP (001)		
0F BA	8	mem, 11B						BT	BTS	BTR	BTC	
0F C7	9	mem			CMPXCH8B Mq CMPXCHG16B Mdq					VMPTRLD Mq	VMPTRST Mq	
			66							VMCLEAR Mq		
		11B	F3								VMXON Mq	
			F3								RDRAND Rv	RDSEED Rv
0F B9	10	mem 11B										
C6	11	mem		MOV Eb, lb								
11B										XABORT (000) lb		
C7	11	mem		MOV Ev, lz								
11B											XBEGIN (000) Jz	
0F 71	12	mem				psrlw Nq, lb		psraw Nq, lb		psllw Nq, lb		
		11B	66			vpsrlw Hx,Ux,lb		vpsraw Hx,Ux,lb		vpsllw Hx,Ux,lb		
0F 72	13	mem				psrld Nq, lb		psrad Nq, lb		pslld Nq, lb		
		11B	66			vpsrld Hx,Ux,lb		vpsrad Hx,Ux,lb		vpslld Hx,Ux,lb		
0F 73	14	mem				psrlq Nq, lb				psllq Nq, lb		
		11B	66			vpsrlq Hx,Ux,lb	vpsrldq Hx,Ux,lb			vpsllq Hx,Ux,lb	vpslldq Hx,Ux,lb	

**Table A-6. Opcode Extensions for One- and Two-byte Opcodes by Group Number \* (Contd.)**

Opcode	Group	Mod 7,6	pfx	Encoding of Bits 5,4,3 of the ModR/M Byte (bits 2,1,0 in parenthesis)							
				000	001	010	011	100	101	110	111
0F AE	15	mem		fxsave	fxrstor	ldmxcsr	stmxcsr	XSAVE	XRSTOR	XSAVEOPT	clflush
		11B	F3	RDFSBASE Ry	RDGSBASE Ry	WRFSBASE Ry	WRGSBASE Ry		lfence	mfence	sfence
0F 18	16	mem		prefetch NTA	prefetch T0	prefetch T1	prefetch T2				
		11B									
VEX.0F38 F3	17	mem			BLSR <sup>v</sup> By, Ey	BLSMSK <sup>v</sup> By, Ey	BLSI <sup>v</sup> By, Ey				
		11B									

**NOTES:**

\* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

## A.5 ESCAPE OPCODE INSTRUCTIONS

Opcode maps for coprocessor escape instruction opcodes (x87 floating-point instruction opcodes) are in Table A-7 through Table A-22. These maps are grouped by the first byte of the opcode, from D8-DF. Each of these opcodes has a ModR/M byte. If the ModR/M byte is within the range of 00H-BFH, bits 3-5 of the ModR/M byte are used as an opcode extension, similar to the technique used for 1- and 2-byte opcodes (see A.4). If the ModR/M byte is outside the range of 00H through BFH, the entire ModR/M byte is used as an opcode extension.

### A.5.1 Opcode Look-up Examples for Escape Instruction Opcodes

Examples are provided below.

#### Example A-6. Opcode with ModR/M Byte in the 00H through BFH Range

DD0504000000H can be interpreted as follows:

- The instruction encoded with this opcode can be located in Section . Since the ModR/M byte (05H) is within the 00H through BFH range, bits 3 through 5 (000) of this byte indicate the opcode for an FLD double-real instruction (see Table A-9).
- The double-real value to be loaded is at 00000004H (the 32-bit displacement that follows and belongs to this opcode).

#### Example A-7. Opcode with ModR/M Byte outside the 00H through BFH Range

D8C1H can be interpreted as follows:

- This example illustrates an opcode with a ModR/M byte outside the range of 00H through BFH. The instruction can be located in Section A.4.
- In Table A-8, the ModR/M byte C1H indicates row C, column 1 (the FADD instruction using ST(0), ST(1) as operands).

### A.5.2 Escape Opcode Instruction Tables

Tables are listed below.

#### A.5.2.1 Escape Opcodes with D8 as First Byte

Table A-7 and A-8 contain maps for the escape instruction opcodes that begin with D8H. Table A-7 shows the map if the ModR/M byte is in the range of 00H-BFH. Here, the value of bits 3-5 (the nnn field in Figure A-1) selects the instruction.

**Table A-7. D8 Opcode Map When ModR/M Byte is Within 00H to BFH \***

nnn Field of ModR/M Byte (refer to Figure A.4)							
000B	001B	010B	011B	100B	101B	110B	111B
FADD single-real	FMUL single-real	FCOM single-real	FCOMP single-real	FSUB single-real	FSUBR single-real	FDIV single-real	FDIVR single-real

#### NOTES:

- \* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

Table A-8 shows the map if the ModR/M byte is outside the range of 00H-BFH. Here, the first digit of the ModR/M byte selects the table row and the second digit selects the column.

**Table A-8. D8 Opcode Map When ModR/M Byte is Outside 00H to BFH \***

	0	1	2	3	4	5	6	7
C	FADD							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
D	FCOM							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),T(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
E	FSUB							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
F	FDIV							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)

	8	9	A	B	C	D	E	F
C	FMUL							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
D	FCOMP							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),T(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
E	FSUBR							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
F	FDIVR							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)

**NOTES:**

\* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

**A.5.2.2 Escape Opcodes with D9 as First Byte**

Table A-9 and A-10 contain maps for escape instruction opcodes that begin with D9H. Table A-9 shows the map if the ModR/M byte is in the range of 00H-BFH. Here, the value of bits 3-5 (the nnn field in Figure A-1) selects the instruction.

**Table A-9. D9 Opcode Map When ModR/M Byte is Within 00H to BFH \***

nnn Field of ModR/M Byte							
000B	001B	010B	011B	100B	101B	110B	111B
FLD single-real		FST single-real	FSTP single-real	FLDENV 14/28 bytes	FLDCW 2 bytes	FSTENV 14/28 bytes	FSTCW 2 bytes

**NOTES:**

\* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

Table A-10 shows the map if the ModR/M byte is outside the range of 00H-BFH. Here, the first digit of the ModR/M byte selects the table row and the second digit selects the column.

**Table A-10. D9 Opcode Map When ModR/M Byte is Outside 00H to BFH \***

	0	1	2	3	4	5	6	7
C	FLD							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
D	FNOP							
E	FCHS	FABS			FTST	FXAM		
F	F2XM1	FYL2X	FPTAN	FPATAN	EXTRACT	FPREM1	FDECSTP	FINCSTP

	8	9	A	B	C	D	E	F
C	FXCH							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
D								
E	FLD1	FLDL2T	FLDL2E	FLDPI	FLDLG2	FLDLN2	FLDZ	
F	FPREM	FYL2XP1	FSQRT	FSINCOS	FRNDINT	FSCALE	FSIN	FCOS

**NOTES:**

\* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

**A.5.2.3 Escape Opcodes with DA as First Byte**

Table A-11 and A-12 contain maps for escape instruction opcodes that begin with DAH. Table A-11 shows the map if the ModR/M byte is in the range of 00H-BFH. Here, the value of bits 3-5 (the nnn field in Figure A-1) selects the instruction.

**Table A-11. DA Opcode Map When ModR/M Byte is Within 00H to BFH \***

nnn Field of ModR/M Byte							
000B	001B	010B	011B	100B	101B	110B	111B
FIADD dword-integer	FIMUL dword-integer	FICOM dword-integer	FICOMP dword-integer	FISUB dword-integer	FISUBR dword-integer	FIDIV dword-integer	FIDIVR dword-integer

**NOTES:**

\* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.



Table A-12 shows the map if the ModR/M byte is outside the range of 00H-BFH. Here, the first digit of the ModR/M byte selects the table row and the second digit selects the column.

**Table A-12. DA Opcode Map When ModR/M Byte is Outside 00H to BFH \***

	0	1	2	3	4	5	6	7
C	FCMOVB							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
D	FCMOVBE							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
E								
F								

	8	9	A	B	C	D	E	F
C	FCMOVE							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
D	FCMOVU							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
E		FUCOMPP						
F								

**NOTES:**

\* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

**A.5.2.4 Escape Opcodes with DB as First Byte**

Table A-13 and A-14 contain maps for escape instruction opcodes that begin with DBH. Table A-13 shows the map if the ModR/M byte is in the range of 00H-BFH. Here, the value of bits 3-5 (the nnn field in Figure A-1) selects the instruction.

**Table A-13. DB Opcode Map When ModR/M Byte is Within 00H to BFH \***

nnn Field of ModR/M Byte							
000B	001B	010B	011B	100B	101B	110B	111B
FILD dword-integer	FISTTP dword-integer	FIST dword-integer	FISTP dword-integer		FLD extended-real		FSTP extended-real

**NOTES:**

\* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

Table A-14 shows the map if the ModR/M byte is outside the range of 00H-BFH. Here, the first digit of the ModR/M byte selects the table row and the second digit selects the column.

**Table A-14. DB Opcode Map When ModR/M Byte is Outside 00H to BFH \***

	0	1	2	3	4	5	6	7
C	FCMOVNB							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
D	FCMOVNBE							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
E			FCLEX	FINIT				
F	FCOMI							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)

	8	9	A	B	C	D	E	F
C	FCMOVNE							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
D	FCMOVNU							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
E	FUCOMI							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
F								

**NOTES:**

\* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

**A.5.2.5 Escape Opcodes with DC as First Byte**

Table A-15 and A-16 contain maps for escape instruction opcodes that begin with DCH. Table A-15 shows the map if the ModR/M byte is in the range of 00H-BFH. Here, the value of bits 3-5 (the nnn field in Figure A-1) selects the instruction.

**Table A-15. DC Opcode Map When ModR/M Byte is Within 00H to BFH \***

nnn Field of ModR/M Byte (refer to Figure A-1)							
000B	001B	010B	011B	100B	101B	110B	111B
FADD double-real	FMUL double-real	FCOM double-real	FCOMP double-real	FSUB double-real	FSUBR double-real	FDIV double-real	FDIVR double-real

**NOTES:**

\* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

Table A-16 shows the map if the ModR/M byte is outside the range of 00H-BFH. In this case the first digit of the ModR/M byte selects the table row and the second digit selects the column.

**Table A-16. DC Opcode Map When ModR/M Byte is Outside 00H to BFH \***

	0	1	2	3	4	5	6	7
C	FADD							
	ST(0),ST(0)	ST(1),ST(0)	ST(2),ST(0)	ST(3),ST(0)	ST(4),ST(0)	ST(5),ST(0)	ST(6),ST(0)	ST(7),ST(0)
D								
E	FSUBR							
	ST(0),ST(0)	ST(1),ST(0)	ST(2),ST(0)	ST(3),ST(0)	ST(4),ST(0)	ST(5),ST(0)	ST(6),ST(0)	ST(7),ST(0)
F	FDIVR							
	ST(0),ST(0)	ST(1),ST(0)	ST(2),ST(0)	ST(3),ST(0)	ST(4),ST(0)	ST(5),ST(0)	ST(6),ST(0)	ST(7),ST(0)

	8	9	A	B	C	D	E	F
C	FMUL							
	ST(0),ST(0)	ST(1),ST(0)	ST(2),ST(0)	ST(3),ST(0)	ST(4),ST(0)	ST(5),ST(0)	ST(6),ST(0)	ST(7),ST(0)
D								
E	FSUB							
	ST(0),ST(0)	ST(1),ST(0)	ST(2),ST(0)	ST(3),ST(0)	ST(4),ST(0)	ST(5),ST(0)	ST(6),ST(0)	ST(7),ST(0)
F	FDIV							
	ST(0),ST(0)	ST(1),ST(0)	ST(2),ST(0)	ST(3),ST(0)	ST(4),ST(0)	ST(5),ST(0)	ST(6),ST(0)	ST(7),ST(0)

**NOTES:**

\* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

**A.5.2.6 Escape Opcodes with DD as First Byte**

Table A-17 and A-18 contain maps for escape instruction opcodes that begin with DDH. Table A-17 shows the map if the ModR/M byte is in the range of 00H-BFH. Here, the value of bits 3-5 (the nnn field in Figure A-1) selects the instruction.

**Table A-17. DD Opcode Map When ModR/M Byte is Within 00H to BFH \***

nnn Field of ModR/M Byte							
000B	001B	010B	011B	100B	101B	110B	111B
FLD double-real	FISTTP integer64	FST double-real	FSTP double-real	FRSTOR 98/108bytes		FSAVE 98/108bytes	FSTSW 2 bytes

**NOTES:**

\* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

Table A-18 shows the map if the ModR/M byte is outside the range of 00H-BFH. The first digit of the ModR/M byte selects the table row and the second digit selects the column.

**Table A-18. DD Opcode Map When ModR/M Byte is Outside 00H to BFH \***

	0	1	2	3	4	5	6	7
C	FFREE							
	ST(0)	ST(1)	ST(2)	ST(3)	ST(4)	ST(5)	ST(6)	ST(7)
D	FST							
	ST(0)	ST(1)	ST(2)	ST(3)	ST(4)	ST(5)	ST(6)	ST(7)
E	FUCOM							
	ST(0),ST(0)	ST(1),ST(0)	ST(2),ST(0)	ST(3),ST(0)	ST(4),ST(0)	ST(5),ST(0)	ST(6),ST(0)	ST(7),ST(0)
F								

	8	9	A	B	C	D	E	F
C								
D	FSTP							
	ST(0)	ST(1)	ST(2)	ST(3)	ST(4)	ST(5)	ST(6)	ST(7)
E	FUCOMP							
	ST(0)	ST(1)	ST(2)	ST(3)	ST(4)	ST(5)	ST(6)	ST(7)
F								

**NOTES:**

\* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

**A.5.2.7 Escape Opcodes with DE as First Byte**

Table A-19 and A-20 contain opcode maps for escape instruction opcodes that begin with DEH. Table A-19 shows the opcode map if the ModR/M byte is in the range of 00H-BFH. In this case, the value of bits 3-5 (the nnn field in Figure A-1) selects the instruction.

**Table A-19. DE Opcode Map When ModR/M Byte is Within 00H to BFH \***

nnn Field of ModR/M Byte							
000B	001B	010B	011B	100B	101B	110B	111B
FIADD word-integer	FIMUL word-integer	FICOM word-integer	FICOMP word-integer	FISUB word-integer	FISUBR word-integer	FIDIV word-integer	FIDIVR word-integer

**NOTES:**

\* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

Table A-20 shows the opcode map if the ModR/M byte is outside the range of 00H-BFH. The first digit of the ModR/M byte selects the table row and the second digit selects the column.

**Table A-20. DE Opcode Map When ModR/M Byte is Outside 00H to BFH \***

	0	1	2	3	4	5	6	7
C	FADDP							
	ST(0),ST(0)	ST(1),ST(0)	ST(2),ST(0)	ST(3),ST(0)	ST(4),ST(0)	ST(5),ST(0)	ST(6),ST(0)	ST(7),ST(0)
D								
E	FSUBRP							
	ST(0),ST(0)	ST(1),ST(0)	ST(2),ST(0)	ST(3),ST(0)	ST(4),ST(0)	ST(5),ST(0)	ST(6),ST(0)	ST(7),ST(0)
F	FDIVRP							
	ST(0),ST(0)	ST(1),ST(0)	ST(2),ST(0)	ST(3),ST(0)	ST(4),ST(0)	ST(5),ST(0)	ST(6),ST(0)	ST(7),ST(0)

	8	9	A	B	C	D	E	F
C	FMULP							
	ST(0),ST(0)	ST(1),ST(0)	ST(2),ST(0)	ST(3),ST(0)	ST(4),ST(0)	ST(5),ST(0)	ST(6),ST(0)	ST(7),ST(0)
D		FCOMP						
E	FSUBP							
	ST(0),ST(0)	ST(1),ST(0)	ST(2),ST(0)	ST(3),ST(0)	ST(4),ST(0)	ST(5),ST(0)	ST(6),ST(0)	ST(7),ST(0)
F	FDIVP							
	ST(0),ST(0)	ST(1),ST(0)	ST(2),ST(0)	ST(3),ST(0)	ST(4),ST(0)	ST(5),ST(0)	ST(6),ST(0)	ST(7),ST(0)

**NOTES:**

\* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

**A.5.2.8 Escape Opcodes with DF As First Byte**

Table A-21 and A-22 contain the opcode maps for escape instruction opcodes that begin with DFH. Table A-21 shows the opcode map if the ModR/M byte is in the range of 00H-BFH. Here, the value of bits 3-5 (the nnn field in Figure A-1) selects the instruction.

**Table A-21. DF Opcode Map When ModR/M Byte is Within 00H to BFH \***

nnn Field of ModR/M Byte							
000B	001B	010B	011B	100B	101B	110B	111B
FILD word-integer	FISTTP word-integer	FIST word-integer	FISTP word-integer	FBLD packed-BCD	FILD qword-integer	FBSTP packed-BCD	FISTP qword-integer

**NOTES:**

\* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

Table A-22 shows the opcode map if the ModR/M byte is outside the range of 00H-BFH. The first digit of the ModR/M byte selects the table row and the second digit selects the column.

**Table A-22. DF Opcode Map When ModR/M Byte is Outside 00H to BFH \***

	0	1	2	3	4	5	6	7
C								
D								
E	FSTSW AX							
F	FCOMIP							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)

	8	9	A	B	C	D	E	F
C								
D								
E	FUCOMIP							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
F								

**NOTES:**

\* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

This page was  
intentionally left  
blank.

## 12. Updates to Appendix C, Volume 2D

Change bars show changes to Appendix C of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2D: Instruction Set Reference*.

-----

Changes include minor updates to reflect volume addition, arrangement of information and document order numbers.



# INTEL® C/C++ COMPILER INTRINSICS AND FUNCTIONAL EQUIVALENTS

## APPENDIX C

The two tables in this appendix itemize the Intel C/C++ compiler intrinsics and functional equivalents for the Intel MMX technology, SSE, SSE2, SSE3, and SSSE3 instructions.

There may be additional intrinsics that do not have an instruction equivalent. It is strongly recommended that the reader reference the compiler documentation for the complete list of supported intrinsics. Please refer to <http://www.intel.com/support/performance/tools/>.

Table C-1 presents simple intrinsics and Table C-2 presents composite intrinsics. Some intrinsics are “composites” because they require more than one instruction to implement them.

Intel C/C++ Compiler intrinsic names reflect the following naming conventions:

`_mm_<intrin_op>_<suffix>`

where:

`<intrin_op>` Indicates the intrinsics basic operation; for example, add for addition and sub for subtraction

`<suffix>` Denotes the type of data operated on by the instruction. The first one or two letters of each suffix denotes whether the data is packed (p), extended packed (ep), or scalar (s).

The remaining letters denote the type:

s single-precision floating point  
d double-precision floating point  
i128 signed 128-bit integer  
i64 signed 64-bit integer  
u64 unsigned 64-bit integer  
i32 signed 32-bit integer  
u32 unsigned 32-bit integer  
i16 signed 16-bit integer  
u16 unsigned 16-bit integer  
i8 signed 8-bit integer  
u8 unsigned 8-bit integer

The variable `r` is generally used for the intrinsic's return value. A number appended to a variable name indicates the element of a packed object. For example, `r0` is the lowest word of `r`.

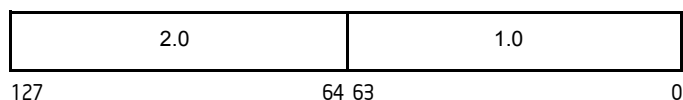
The packed values are represented in right-to-left order, with the lowest value being used for scalar operations. Consider the following example operation:

```
double a[2] = {1.0, 2.0};  
__m128d t = _mm_load_pd(a);
```

The result is the same as either of the following:

```
__m128d t = _mm_set_pd(2.0, 1.0);  
__m128d t = _mm_setr_pd(1.0, 2.0);
```

In other words, the XMM register that holds the value `t` will look as follows:



The “scalar” element is 1.0. Due to the nature of the instruction, some intrinsics require their arguments to be immediates (constant integer literals).

To use an intrinsic in your code, insert a line with the following syntax:

```
data_type intrinsic_name (parameters)
```

Where:

data_type	Is the return data type, which can be either void, int, __m64, __m128, __m128d, or __m128i. Only the __mm_empty intrinsic returns void.
intrinsic_name	Is the name of the intrinsic, which behaves like a function that you can use in your C/C++ code instead of in-lining the actual instruction.
parameters	Represents the parameters required by each intrinsic.

## C.1 SIMPLE INTRINSICS

### NOTE

For detailed descriptions of the intrinsics in Table C-1, see the corresponding mnemonic in Chapter 3, “Instruction Set Reference, A-L” of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A*, Chapter 4, “Instruction Set Reference, M-U” of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2B*, or Chapter 5, “Instruction Set Reference, V-Z,” of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2C*.

**Table C-1. Simple Intrinsics**

Mnemonic	Intrinsic
ADDPD	__m128d __mm_add_pd(__m128d a, __m128d b)
ADDPS	__m128 __mm_add_ps(__m128 a, __m128 b)
ADDSD	__m128d __mm_add_sd(__m128d a, __m128d b)
ADDSS	__m128 __mm_add_ss(__m128 a, __m128 b)
ADDSUBPD	__m128d __mm_addsub_pd(__m128d a, __m128d b)
ADDSUBPS	__m128 __mm_addsub_ps(__m128 a, __m128 b)
AESDEC	__m128i __mm_aesdec (__m128i, __m128i)
AESDECLAST	__m128i __mm_aesdeclast (__m128i, __m128i)
AESENC	__m128i __mm_aesenc (__m128i, __m128i)
AESENCLAST	__m128i __mm_aesenclast (__m128i, __m128i)
AESIMC	__m128i __mm_aesimc (__m128i)
AESKEYGENASSIST	__m128i __mm_aesimc (__m128i, const int)
ANDNPD	__m128d __mm_andnot_pd(__m128d a, __m128d b)
ANDNPS	__m128 __mm_andnot_ps(__m128 a, __m128 b)
ANDPD	__m128d __mm_and_pd(__m128d a, __m128d b)
ANDPS	__m128 __mm_and_ps(__m128 a, __m128 b)
BLENDDPD	__m128d __mm_blend_pd(__m128d v1, __m128d v2, const int mask)
BLENDPS	__m128 __mm_blend_ps(__m128 v1, __m128 v2, const int mask)
BLENDVDPD	__m128d __mm_blendv_pd(__m128d v1, __m128d v2, __m128d v3)
BLENDVPS	__m128 __mm_blendv_ps(__m128 v1, __m128 v2, __m128 v3)
CLFLUSH	void __mm_clflush(void const *p)
CMPPD	__m128d __mm_cmpeq_pd(__m128d a, __m128d b)

Table C-1. Simple Intrinsics (Contd.)

Mnemonic	Intrinsic
	__m128d_mm_cmplt_pd(__m128d a, __m128d b)
	__m128d_mm_cmple_pd(__m128d a, __m128d b)
	__m128d_mm_cmpgt_pd(__m128d a, __m128d b)
	__m128d_mm_cmpge_pd(__m128d a, __m128d b)
	__m128d_mm_cmpneq_pd(__m128d a, __m128d b)
	__m128d_mm_cmpnlt_pd(__m128d a, __m128d b)
	__m128d_mm_cmpngt_pd(__m128d a, __m128d b)
	__m128d_mm_cmpnge_pd(__m128d a, __m128d b)
	__m128d_mm_cmpord_pd(__m128d a, __m128d b)
	__m128d_mm_cmpunord_pd(__m128d a, __m128d b)
	__m128d_mm_cmpnle_pd(__m128d a, __m128d b)
CMPPS	__m128_mm_cmpeq_ps(__m128 a, __m128 b)
	__m128_mm_cmplt_ps(__m128 a, __m128 b)
	__m128_mm_cmple_ps(__m128 a, __m128 b)
	__m128_mm_cmpgt_ps(__m128 a, __m128 b)
	__m128_mm_cmpge_ps(__m128 a, __m128 b)
	__m128_mm_cmpneq_ps(__m128 a, __m128 b)
	__m128_mm_cmpnlt_ps(__m128 a, __m128 b)
	__m128_mm_cmpngt_ps(__m128 a, __m128 b)
	__m128_mm_cmpnge_ps(__m128 a, __m128 b)
	__m128_mm_cmpord_ps(__m128 a, __m128 b)
	__m128_mm_cmpunord_ps(__m128 a, __m128 b)
	__m128_mm_cmpnle_ps(__m128 a, __m128 b)
CMPSD	__m128d_mm_cmpeq_sd(__m128d a, __m128d b)
	__m128d_mm_cmplt_sd(__m128d a, __m128d b)
	__m128d_mm_cmple_sd(__m128d a, __m128d b)
	__m128d_mm_cmpgt_sd(__m128d a, __m128d b)
	__m128d_mm_cmpge_sd(__m128d a, __m128d b)
	__m128d_mm_cmpneq_sd(__m128d a, __m128d b)
	__m128d_mm_cmpnlt_sd(__m128d a, __m128d b)
	__m128d_mm_cmpnle_sd(__m128d a, __m128d b)
	__m128d_mm_cmpngt_sd(__m128d a, __m128d b)
	__m128d_mm_cmpnge_sd(__m128d a, __m128d b)
	__m128d_mm_cmpord_sd(__m128d a, __m128d b)
	__m128d_mm_cmpunord_sd(__m128d a, __m128d b)
CMPPS	__m128_mm_cmpeq_ss(__m128 a, __m128 b)
	__m128_mm_cmplt_ss(__m128 a, __m128 b)
	__m128_mm_cmple_ss(__m128 a, __m128 b)
	__m128_mm_cmpgt_ss(__m128 a, __m128 b)
	__m128_mm_cmpge_ss(__m128 a, __m128 b)
	__m128_mm_cmpneq_ss(__m128 a, __m128 b)

**Table C-1. Simple Intrinsics (Contd.)**

Mnemonic	Intrinsic
	__m128 _mm_cmpnlt_ss(__m128 a, __m128 b)
	__m128 _mm_cmpnle_ss(__m128 a, __m128 b)
	__m128 _mm_cmpngt_ss(__m128 a, __m128 b)
	__m128 _mm_cmpnge_ss(__m128 a, __m128 b)
	__m128 _mm_cmpord_ss(__m128 a, __m128 b)
	__m128 _mm_cmpunord_ss(__m128 a, __m128 b)
COMISD	int _mm_comieq_sd(__m128d a, __m128d b)
	int _mm_comilt_sd(__m128d a, __m128d b)
	int _mm_comile_sd(__m128d a, __m128d b)
	int _mm_comigt_sd(__m128d a, __m128d b)
	int _mm_comige_sd(__m128d a, __m128d b)
	int _mm_comineq_sd(__m128d a, __m128d b)
COMISS	int _mm_comieq_ss(__m128 a, __m128 b)
	int _mm_comilt_ss(__m128 a, __m128 b)
	int _mm_comile_ss(__m128 a, __m128 b)
	int _mm_comigt_ss(__m128 a, __m128 b)
	int _mm_comige_ss(__m128 a, __m128 b)
	int _mm_comineq_ss(__m128 a, __m128 b)
CRC32	unsigned int _mm_crc32_u8(unsigned int crc, unsigned char data)
	unsigned int _mm_crc32_u16(unsigned int crc, unsigned short data)
	unsigned int _mm_crc32_u32(unsigned int crc, unsigned int data)
	unsigned __int64 _mm_crc32_u64(unsigned __int64 crc, unsigned __int64 data)
CVTDQ2PD	__m128d _mm_cvtepi32_pd(__m128i a)
CVTDQ2PS	__m128 _mm_cvtepi32_ps(__m128i a)
CVTPD2DQ	__m128i _mm_cvtpd_epi32(__m128d a)
CVTPD2PI	__m64 _mm_cvtpd_pi32(__m128d a)
CVTPD2PS	__m128 _mm_cvtpd_ps(__m128d a)
CVTPI2PD	__m128d _mm_cvtpi32_pd(__m64 a)
CVTPI2PS	__m128 _mm_cvt_pi2ps(__m128 a, __m64 b)
CVTPS2DQ	__m128i _mm_cvtps_epi32(__m128 a)
CVTPS2PD	__m128d _mm_cvtps_pd(__m128 a)
CVTPS2PI	__m64 _mm_cvt_ps2pi(__m128 a)
CVTSD2SI	int _mm_cvtsd_si32(__m128d a)
CVTSD2SS	__m128 _mm_cvtsd_ss(__m128 a, __m128d b)
CVTSI2SD	__m128d _mm_cvtsi32_sd(__m128d a, int b)
CVTSI2SS	__m128 _mm_cvt_si2ss(__m128 a, int b)
CVTSI2SS	__m128 _mm_cvtsi32_ss(__m128 a, int b)
CVTSI2SS	__m128 _mm_cvtsi64_ss(__m128 a, __int64 b)
CVTSS2SD	__m128d _mm_cvtss_sd(__m128d a, __m128 b)
CVTSS2SI	int _mm_cvt_ss2si(__m128 a)
CVTSS2SI	int _mm_cvtss_si32(__m128 a)

Table C-1. Simple Intrinsics (Contd.)

Mnemonic	Intrinsic
CVTTPD2DQ	__m128i _mm_cvttpd_epi32(__m128d a)
CVTTPD2PI	__m64 _mm_cvttpd_pi32(__m128d a)
CVTTPS2DQ	__m128i _mm_cvttps_epi32(__m128 a)
CVTTPS2PI	__m64 _mm_cvttps_pi32(__m128 a) __m64 _mm_cvttps_pi32(__m128 a)
CVTTSD2SI	int _mm_cvttss_si32(__m128d a)
CVTTSS2SI	int _mm_cvtt_ss2si(__m128 a) int _mm_cvttss_si32(__m128 a) __m64 _mm_cvtsi32_si64(int i) int _mm_cvtsi64_si32(__m64 m)
DIVPD	__m128d _mm_div_pd(__m128d a, __m128d b)
DIVPS	__m128 _mm_div_ps(__m128 a, __m128 b)
DIVSD	__m128d _mm_div_sd(__m128d a, __m128d b)
DIVSS	__m128 _mm_div_ss(__m128 a, __m128 b)
DPPD	__m128d _mm_dp_pd(__m128d a, __m128d b, const int mask)
DPPS	__m128 _mm_dp_ps(__m128 a, __m128 b, const int mask)
EMMS	void _mm_empty()
EXTRACTPS	int _mm_extract_ps(__m128 src, const int ndx)
HADDPD	__m128d _mm_hadd_pd(__m128d a, __m128d b)
HADDPS	__m128 _mm_hadd_ps(__m128 a, __m128 b)
HSUBPD	__m128d _mm_hsub_pd(__m128d a, __m128d b)
HSUBPS	__m128 _mm_hsub_ps(__m128 a, __m128 b)
INSERTPS	__m128 _mm_insert_ps(__m128 dst, __m128 src, const int ndx)
LDDQU	__m128i _mm_lddqu_si128(__m128i const *p)
LDMXCSR	__mm_setcsr(unsigned int i)
LFENCE	void _mm_lfence(void)
MASKMOVDQU	void _mm_maskmoveu_si128(__m128i d, __m128i n, char *p)
MASKMOVQ	void _mm_maskmove_si64(__m64 d, __m64 n, char *p)
MAXPD	__m128d _mm_max_pd(__m128d a, __m128d b)
MAXPS	__m128 _mm_max_ps(__m128 a, __m128 b)
MAXSD	__m128d _mm_max_sd(__m128d a, __m128d b)
MAXSS	__m128 _mm_max_ss(__m128 a, __m128 b)
MFENCE	void _mm_mfence(void)
MINPD	__m128d _mm_min_pd(__m128d a, __m128d b)
MINPS	__m128 _mm_min_ps(__m128 a, __m128 b)
MINSD	__m128d _mm_min_sd(__m128d a, __m128d b)
MINSS	__m128 _mm_min_ss(__m128 a, __m128 b)
MONITOR	void _mm_monitor(void const *p, unsigned extensions, unsigned hints)
MOVAPD	__m128d _mm_load_pd(double * p) void _mm_store_pd(double *p, __m128d a)
MOVAPS	__m128 _mm_load_ps(float * p) void _mm_store_ps(float *p, __m128 a)

Table C-1. Simple Intrinsics (Contd.)

Mnemonic	Intrinsic
MOVD	__m128i _mm_cvtsi32_si128(int a)
	int _mm_cvtsi128_si32(__m128i a)
	__m64 _mm_cvtsi32_si64(int a)
	int _mm_cvtsi64_si32(__m64 a)
MOVDDUP	__m128d _mm_movedup_pd(__m128d a)
	__m128d _mm_loaddup_pd(double const * dp)
MOVDDQA	__m128i _mm_load_si128(__m128i * p)
	void _mm_store_si128(__m128i *p, __m128i a)
MOVDDQU	__m128i _mm_loadu_si128(__m128i * p)
	void _mm_storeu_si128(__m128i *p, __m128i a)
MOVDDQ2Q	__m64 _mm_movepi64_pi64(__m128i a)
MOVHLPS	__m128 _mm_movehl_ps(__m128 a, __m128 b)
MOVHPD	__m128d _mm_loadh_pd(__m128d a, double * p)
	void _mm_storeh_pd(double * p, __m128d a)
MOVHPS	__m128 _mm_loadh_pi(__m128 a, __m64 * p)
	void _mm_storeh_pi(__m64 * p, __m128 a)
MOVLPD	__m128d _mm_loadl_pd(__m128d a, double * p)
	void _mm_storel_pd(double * p, __m128d a)
MOVLPS	__m128 _mm_loadl_pi(__m128 a, __m64 *p)
	void _mm_storel_pi(__m64 * p, __m128 a)
MOVLHPS	__m128 _mm_movelh_ps(__m128 a, __m128 b)
MOVMSKPD	int _mm_movemask_pd(__m128d a)
MOVMSKPS	int _mm_movemask_ps(__m128 a)
MOVNTDQA	__m128i _mm_stream_load_si128(__m128i *p)
MOVNTDQ	void _mm_stream_si128(__m128i * p, __m128i a)
MOVNTPD	void _mm_stream_pd(double * p, __m128d a)
MOVNTPS	void _mm_stream_ps(float * p, __m128 a)
MOVNTI	void _mm_stream_si32(int * p, int a)
MOVNTQ	void _mm_stream_pi(__m64 * p, __m64 a)
MOVQ	__m128i _mm_loadl_epi64(__m128i * p)
	void _mm_storel_epi64(__m128i * p, __m128i a)
	__m128i _mm_move_epi64(__m128i a)
MOVQ2DQ	__m128i _mm_movpi64_epi64(__m64 a)
MOVSD	__m128d _mm_load_sd(double * p)
	void _mm_store_sd(double * p, __m128d a)
	__m128d _mm_move_sd(__m128d a, __m128d b)
MOVSHDUP	__m128 _mm_movehdup_ps(__m128 a)
MOVSLDUP	__m128 _mm_moveldup_ps(__m128 a)
MOVSS	__m128 _mm_load_ss(float * p)
	void _mm_store_ss(float * p, __m128 a)
	__m128 _mm_move_ss(__m128 a, __m128 b)

Table C-1. Simple Intrinsics (Contd.)

Mnemonic	Intrinsic
MOVUPD	__m128d _mm_loadu_pd(double * p)
	void_mm_storeu_pd(double *p, __m128d a)
MOVUPS	__m128 _mm_loadu_ps(float * p)
	void_mm_storeu_ps(float *p, __m128 a)
MPSADBW	__m128i _mm_mpsadbw_epu8(__m128i s1, __m128i s2, const int mask)
MULPD	__m128d _mm_mul_pd(__m128d a, __m128d b)
MULPS	__m128 _mm_mul_ss(__m128 a, __m128 b)
MULSD	__m128d _mm_mul_sd(__m128d a, __m128d b)
MULSS	__m128 _mm_mul_ss(__m128 a, __m128 b)
MWAIT	void_mm_mwait(unsigned extensions, unsigned hints)
ORPD	__m128d _mm_or_pd(__m128d a, __m128d b)
ORPS	__m128 _mm_or_ps(__m128 a, __m128 b)
PABSB	__m64 _mm_abs_pi8 (__m64 a)
	__m128i _mm_abs_epi8 (__m128i a)
PABSD	__m64 _mm_abs_pi32 (__m64 a)
	__m128i _mm_abs_epi32 (__m128i a)
PABSW	__m64 _mm_abs_pi16 (__m64 a)
	__m128i _mm_abs_epi16 (__m128i a)
PACKSSWB	__m128i _mm_packs_epi16(__m128i m1, __m128i m2)
PACKSSWB	__m64 _mm_packs_pi16(__m64 m1, __m64 m2)
PACKSSDW	__m128i _mm_packs_epi32 (__m128i m1, __m128i m2)
PACKSSDW	__m64 _mm_packs_pi32 (__m64 m1, __m64 m2)
PACKUSDW	__m128i _mm_packus_epi32(__m128i m1, __m128i m2)
PACKUSWB	__m128i _mm_packus_epi16(__m128i m1, __m128i m2)
PACKUSWB	__m64 _mm_packs_pu16(__m64 m1, __m64 m2)
PADDB	__m128i _mm_add_epi8(__m128i m1, __m128i m2)
PADDB	__m64 _mm_add_pi8(__m64 m1, __m64 m2)
PADDW	__m128i _mm_add_epi16(__m128i m1, __m128i m2)
PADDW	__m64 _mm_add_pi16(__m64 m1, __m64 m2)
PADDQ	__m128i _mm_add_epi32(__m128i m1, __m128i m2)
PADDQ	__m64 _mm_add_pi32(__m64 m1, __m64 m2)
PADDQ	__m128i _mm_add_epi64(__m128i m1, __m128i m2)
PADDQ	__m64 _mm_add_si64(__m64 m1, __m64 m2)
PADDSB	__m128i _mm_adds_epi8(__m128i m1, __m128i m2)
PADDSB	__m64 _mm_adds_pi8(__m64 m1, __m64 m2)
PADDSW	__m128i _mm_adds_epi16(__m128i m1, __m128i m2)
PADDSW	__m64 _mm_adds_pi16(__m64 m1, __m64 m2)
PADDUSB	__m128i _mm_adds_epu8(__m128i m1, __m128i m2)
PADDUSB	__m64 _mm_adds_pu8(__m64 m1, __m64 m2)
PADDUSW	__m128i _mm_adds_epu16(__m128i m1, __m128i m2)
PADDUSW	__m64 _mm_adds_pu16(__m64 m1, __m64 m2)

**Table C-1. Simple Intrinsics (Contd.)**

<b>Mnemonic</b>	<b>Intrinsic</b>
PALIGNR	<code>__m64 _mm_alignr_pi8 (__m64 a, __m64 b, int n)</code>
	<code>__m128i _mm_alignr_epi8 (__m128i a, __m128i b, int n)</code>
PAND	<code>__m128i _mm_and_si128(__m128i m1, __m128i m2)</code>
PAND	<code>__m64 _mm_and_si64(__m64 m1, __m64 m2)</code>
PANDN	<code>__m128i _mm_andnot_si128(__m128i m1, __m128i m2)</code>
PANDN	<code>__m64 _mm_andnot_si64(__m64 m1, __m64 m2)</code>
PAUSE	<code>void _mm_pause(void)</code>
PAVGB	<code>__m128i _mm_avg_epu8(__m128i a, __m128i b)</code>
PAVGB	<code>__m64 _mm_avg_pu8(__m64 a, __m64 b)</code>
PAVGW	<code>__m128i _mm_avg_epu16(__m128i a, __m128i b)</code>
PAVGW	<code>__m64 _mm_avg_pu16(__m64 a, __m64 b)</code>
PBLENDVB	<code>__m128i _mm_blendv_epi (__m128i v1, __m128i v2, __m128i mask)</code>
PBLENDW	<code>__m128i _mm_blend_epi16(__m128i v1, __m128i v2, const int mask)</code>
PCLMULQDQ	<code>__m128i _mm_clmulepi64_si128 (__m128i, __m128i, const int)</code>
PCMPEQB	<code>__m128i _mm_cmpeq_epi8(__m128i m1, __m128i m2)</code>
PCMPEQB	<code>__m64 _mm_cmpeq_pi8(__m64 m1, __m64 m2)</code>
PCMPEQQ	<code>__m128i _mm_cmpeq_epi64(__m128i a, __m128i b)</code>
PCMPEQW	<code>__m128i _mm_cmpeq_epi16 (__m128i m1, __m128i m2)</code>
PCMPEQW	<code>__m64 _mm_cmpeq_pi16 (__m64 m1, __m64 m2)</code>
PCMPEQD	<code>__m128i _mm_cmpeq_epi32(__m128i m1, __m128i m2)</code>
PCMPEQD	<code>__m64 _mm_cmpeq_pi32(__m64 m1, __m64 m2)</code>
PCMPESTRI	<code>int _mm_cmpestri (__m128i a, int la, __m128i b, int lb, const int mode)</code>
	<code>int _mm_cmpestra (__m128i a, int la, __m128i b, int lb, const int mode)</code>
	<code>int _mm_cmpestrc (__m128i a, int la, __m128i b, int lb, const int mode)</code>
	<code>int _mm_cmpestro (__m128i a, int la, __m128i b, int lb, const int mode)</code>
	<code>int _mm_cmpestrs (__m128i a, int la, __m128i b, int lb, const int mode)</code>
	<code>int _mm_cmpestrz (__m128i a, int la, __m128i b, int lb, const int mode)</code>
PCMPESTRM	<code>__m128i _mm_cmpestrm (__m128i a, int la, __m128i b, int lb, const int mode)</code>
	<code>int _mm_cmpestra (__m128i a, int la, __m128i b, int lb, const int mode)</code>
	<code>int _mm_cmpestrc (__m128i a, int la, __m128i b, int lb, const int mode)</code>
	<code>int _mm_cmpestro (__m128i a, int la, __m128i b, int lb, const int mode)</code>
	<code>int _mm_cmpestrs (__m128i a, int la, __m128i b, int lb, const int mode)</code>
	<code>int _mm_cmpestrz (__m128i a, int la, __m128i b, int lb, const int mode)</code>
PCMPGTB	<code>__m128i _mm_cmpgt_epi8 (__m128i m1, __m128i m2)</code>
PCMPGTB	<code>__m64 _mm_cmpgt_pi8 (__m64 m1, __m64 m2)</code>
PCMPGTW	<code>__m128i _mm_cmpgt_epi16(__m128i m1, __m128i m2)</code>
PCMPGTW	<code>__m64 _mm_cmpgt_pi16 (__m64 m1, __m64 m2)</code>
PCMPGTD	<code>__m128i _mm_cmpgt_epi32(__m128i m1, __m128i m2)</code>
PCMPGTD	<code>__m64 _mm_cmpgt_pi32(__m64 m1, __m64 m2)</code>
PCMPISTRI	<code>__m128i _mm_cmpestrm (__m128i a, int la, __m128i b, int lb, const int mode)</code>
	<code>int _mm_cmpestra (__m128i a, int la, __m128i b, int lb, const int mode)</code>



Table C-1. Simple Intrinsics (Contd.)

Mnemonic	Intrinsic
	int_mm_cmpestrc (__m128i a, int la, __m128i b, int lb, const int mode)
	int_mm_cmpestro (__m128i a, int la, __m128i b, int lb, const int mode)
	int_mm_cmpestrs (__m128i a, int la, __m128i b, int lb, const int mode)
	int_mm_cmpistrz (__m128i a, __m128i b, const int mode)
PCMPISTRM	__m128i_mm_cmpistrm (__m128i a, __m128i b, const int mode)
	int_mm_cmpistra (__m128i a, __m128i b, const int mode)
	int_mm_cmpestrc (__m128i a, __m128i b, const int mode)
	int_mm_cmpestro (__m128i a, __m128i b, const int mode)
	int_mm_cmpestrs (__m128i a, __m128i b, const int mode)
	int_mm_cmpistrz (__m128i a, __m128i b, const int mode)
PCMPGTQ	__m128i_mm_cmpgt_epi64(__m128i a, __m128i b)
PEXTRB	int_mm_extract_epi8 (__m128i src, const int ndx)
PEXTRD	int_mm_extract_epi32 (__m128i src, const int ndx)
PEXTRQ	__int64_mm_extract_epi64 (__m128i src, const int ndx)
PEXTRW	int_mm_extract_epi16(__m128i a, int n)
PEXTRW	int_mm_extract_pi16(__m64 a, int n)
	int_mm_extract_epi16 (__m128i src, int ndx)
PHADDD	__m64_mm_hadd_pi32 (__m64 a, __m64 b)
	__m128i_mm_hadd_epi32 (__m128i a, __m128i b)
PHADDSW	__m64_mm_hadds_pi16 (__m64 a, __m64 b)
	__m128i_mm_hadds_epi16 (__m128i a, __m128i b)
PHADDW	__m64_mm_hadd_pi16 (__m64 a, __m64 b)
	__m128i_mm_hadd_epi16 (__m128i a, __m128i b)
PHMINPOSUW	__m128i_mm_minpos_epu16(__m128i packed_words)
PHSUBD	__m64_mm_hsub_pi32 (__m64 a, __m64 b)
	__m128i_mm_hsub_epi32 (__m128i a, __m128i b)
PHSUBSW	__m64_mm_hsubs_pi16 (__m64 a, __m64 b)
	__m128i_mm_hsubs_epi16 (__m128i a, __m128i b)
PHSUBW	__m64_mm_hsub_pi16 (__m64 a, __m64 b)
	__m128i_mm_hsub_epi16 (__m128i a, __m128i b)
PINSRB	__m128i_mm_insert_epi8(__m128i s1, int s2, const int ndx)
PINSRD	__m128i_mm_insert_epi32(__m128i s2, int s, const int ndx)
PINSRQ	__m128i_mm_insert_epi64(__m128i s2, __int64 s, const int ndx)
PINSRW	__m128i_mm_insert_epi16(__m128i a, int d, int n)
PINSRW	__m64_mm_insert_pi16(__m64 a, int d, int n)
PMADDUBSW	__m64_mm_maddubs_pi16 (__m64 a, __m64 b)
	__m128i_mm_maddubs_epi16 (__m128i a, __m128i b)
PMADDWD	__m128i_mm_madd_epi16(__m128i m1 __m128i m2)
PMADDWD	__m64_mm_madd_pi16(__m64 m1, __m64 m2)
PMASB	__m128i_mm_max_epi8 (__m128i a, __m128i b)
PMASD	__m128i_mm_max_epi32 (__m128i a, __m128i b)

Table C-1. Simple Intrinsics (Contd.)

Mnemonic	Intrinsic
PMAXSW	__m128i _mm_max_epi16(__m128i a, __m128i b)
PMAXSW	__m64 _mm_max_pi16(__m64 a, __m64 b)
PMAXUB	__m128i _mm_max_epu8(__m128i a, __m128i b)
PMAXUB	__m64 _mm_max_pu8(__m64 a, __m64 b)
PMAXUD	__m128i _mm_max_epu32(__m128i a, __m128i b)
PMAXUW	__m128i _mm_max_epu16(__m128i a, __m128i b)
PMINSB	__m128i _mm_min_epi8(__m128i a, __m128i b)
PMINSD	__m128i _mm_min_epi32(__m128i a, __m128i b)
PMINSW	__m128i _mm_min_epi16(__m128i a, __m128i b)
PMINSW	__m64 _mm_min_pi16(__m64 a, __m64 b)
PMINUB	__m128i _mm_min_epu8(__m128i a, __m128i b)
PMINUB	__m64 _mm_min_pu8(__m64 a, __m64 b)
PMINUD	__m128i _mm_min_epu32(__m128i a, __m128i b)
PMINUW	__m128i _mm_min_epu16(__m128i a, __m128i b)
PMOVMASKB	int _mm_movemask_epi8(__m128i a)
PMOVMASKB	int _mm_movemask_pi8(__m64 a)
PMOVSXBW	__m128i _mm_cvtepi8_epi16(__m128i a)
PMOVSXBD	__m128i _mm_cvtepi8_epi32(__m128i a)
PMOVSXBQ	__m128i _mm_cvtepi8_epi64(__m128i a)
PMOVSXWD	__m128i _mm_cvtepi16_epi32(__m128i a)
PMOVSXWQ	__m128i _mm_cvtepi16_epi64(__m128i a)
PMOVSXDQ	__m128i _mm_cvtepi32_epi64(__m128i a)
PMOVZXBW	__m128i _mm_cvtepu8_epi16(__m128i a)
PMOVZXBQ	__m128i _mm_cvtepu8_epi64(__m128i a)
PMOVZXBD	__m128i _mm_cvtepu16_epi32(__m128i a)
PMOVZXBQ	__m128i _mm_cvtepu16_epi64(__m128i a)
PMOVZXWD	__m128i _mm_cvtepu32_epi64(__m128i a)
PMOVZXWQ	__m128i _mm_cvtepu32_epi64(__m128i a)
PMULDQ	__m128i _mm_mul_epi32(__m128i a, __m128i b)
PMULHRW	__m64 _mm_mulhrs_pi16(__m64 a, __m64 b)
	__m128i _mm_mulhrs_epi16(__m128i a, __m128i b)
PMULHUW	__m128i _mm_mulhi_epu16(__m128i a, __m128i b)
PMULHUW	__m64 _mm_mulhi_pu16(__m64 a, __m64 b)
PMULHW	__m128i _mm_mulhi_epi16(__m128i m1, __m128i m2)
PMULHW	__m64 _mm_mulhi_pi16(__m64 m1, __m64 m2)
PMULLUD	__m128i _mm_mullo_epi32(__m128i a, __m128i b)
PMULLW	__m128i _mm_mullo_epi16(__m128i m1, __m128i m2)
PMULLW	__m64 _mm_mullo_pi16(__m64 m1, __m64 m2)
PMULLDQ	__m64 _mm_mul_su32(__m64 m1, __m64 m2)
	__m128i _mm_mul_epu32(__m128i m1, __m128i m2)

Table C-1. Simple Intrinsics (Contd.)

Mnemonic	Intrinsic
POPCNT	int __mm_popcnt_u32(unsigned int a)
	int64_t __mm_popcnt_u64(unsigned __int64 a)
POR	__m64 __mm_or_si64(__m64 m1, __m64 m2)
POR	__m128i __mm_or_si128(__m128i m1, __m128i m2)
PREFETCHH	void __mm_prefetch(char *a, int sel)
PSADBW	__m128i __mm_sad_epu8(__m128i a, __m128i b)
PSADBW	__m64 __mm_sad_pu8(__m64 a, __m64 b)
PSHUFB	__m64 __mm_shuffle_pi8 (__m64 a, __m64 b)
	__m128i __mm_shuffle_epi8 (__m128i a, __m128i b)
PSHUFD	__m128i __mm_shuffle_epi32(__m128i a, int n)
PSHUFW	__m128i __mm_shufflehi_epi16(__m128i a, int n)
PSHUFLW	__m128i __mm_shufflelo_epi16(__m128i a, int n)
PSHUFW	__m64 __mm_shuffle_pi16(__m64 a, int n)
PSIGNB	__m64 __mm_sign_pi8 (__m64 a, __m64 b)
	__m128i __mm_sign_epi8 (__m128i a, __m128i b)
PSIGND	__m64 __mm_sign_pi32 (__m64 a, __m64 b)
	__m128i __mm_sign_epi32 (__m128i a, __m128i b)
PSIGNW	__m64 __mm_sign_pi16 (__m64 a, __m64 b)
	__m128i __mm_sign_epi16 (__m128i a, __m128i b)
PSLLW	__m128i __mm_sll_epi16(__m128i m, __m128i count)
PSLLW	__m128i __mm_slli_epi16(__m128i m, int count)
PSLLW	__m64 __mm_sll_pi16(__m64 m, __m64 count)
	__m64 __mm_slli_pi16(__m64 m, int count)
PSLLD	__m128i __mm_slli_epi32(__m128i m, int count)
	__m128i __mm_sll_epi32(__m128i m, __m128i count)
PSLLD	__m64 __mm_slli_pi32(__m64 m, int count)
	__m64 __mm_sll_pi32(__m64 m, __m64 count)
PSLLQ	__m64 __mm_sll_si64(__m64 m, __m64 count)
	__m64 __mm_slli_si64(__m64 m, int count)
PSLLQ	__m128i __mm_sll_epi64(__m128i m, __m128i count)
	__m128i __mm_slli_epi64(__m128i m, int count)
PSLLDQ	__m128i __mm_slli_si128(__m128i m, int imm)
PSRAW	__m128i __mm_sra_epi16(__m128i m, __m128i count)
	__m128i __mm_srai_epi16(__m128i m, int count)
PSRAW	__m64 __mm_sra_pi16(__m64 m, __m64 count)
	__m64 __mm_srai_pi16(__m64 m, int count)
PSRAD	__m128i __mm_sra_epi32 (__m128i m, __m128i count)
	__m128i __mm_srai_epi32 (__m128i m, int count)
PSRAD	__m64 __mm_sra_pi32 (__m64 m, __m64 count)
	__m64 __mm_srai_pi32 (__m64 m, int count)
PSRLW	__m128i __mm_srl_epi16 (__m128i m, __m128i count)

**Table C-1. Simple Intrinsics (Contd.)**

Mnemonic	Intrinsic
	__m128i _mm_srli_epi16 (__m128i m, int count)
	__m64 _mm_srli_pi16 (__m64 m, __m64 count)
	__m64 _mm_srli_pi16(__m64 m, int count)
PSRLD	__m128i _mm_srli_epi32 (__m128i m, __m128i count)
	__m128i _mm_srli_epi32 (__m128i m, int count)
PSRLD	__m64 _mm_srli_pi32 (__m64 m, __m64 count)
	__m64 _mm_srli_pi32 (__m64 m, int count)
PSRLQ	__m128i _mm_srli_epi64 (__m128i m, __m128i count)
	__m128i _mm_srli_epi64 (__m128i m, int count)
PSRLQ	__m64 _mm_srli_si64 (__m64 m, __m64 count)
	__m64 _mm_srli_si64 (__m64 m, int count)
PSRLDQ	__m128i _mm_srli_si128(__m128i m, int imm)
PSUBB	__m128i _mm_sub_epi8(__m128i m1, __m128i m2)
PSUBB	__m64 _mm_sub_pi8(__m64 m1, __m64 m2)
PSUBW	__m128i _mm_sub_epi16(__m128i m1, __m128i m2)
PSUBW	__m64 _mm_sub_pi16(__m64 m1, __m64 m2)
PSUBD	__m128i _mm_sub_epi32(__m128i m1, __m128i m2)
PSUBD	__m64 _mm_sub_pi32(__m64 m1, __m64 m2)
PSUBQ	__m128i _mm_sub_epi64(__m128i m1, __m128i m2)
PSUBQ	__m64 _mm_sub_si64(__m64 m1, __m64 m2)
PSUBSB	__m128i _mm_subs_epi8(__m128i m1, __m128i m2)
PSUBSB	__m64 _mm_subs_pi8(__m64 m1, __m64 m2)
PSUBSW	__m128i _mm_subs_epi16(__m128i m1, __m128i m2)
PSUBSW	__m64 _mm_subs_pi16(__m64 m1, __m64 m2)
PSUBUSB	__m128i _mm_subs_epu8(__m128i m1, __m128i m2)
PSUBUSB	__m64 _mm_subs_pu8(__m64 m1, __m64 m2)
PSUBUSW	__m128i _mm_subs_epu16(__m128i m1, __m128i m2)
PSUBUSW	__m64 _mm_subs_pu16(__m64 m1, __m64 m2)
PTEST	int _mm_testz_si128(__m128i s1, __m128i s2)
	int _mm_testc_si128(__m128i s1, __m128i s2)
	int _mm_testnzc_si128(__m128i s1, __m128i s2)
PUNPCKHBW	__m64 _mm_unpackhi_pi8(__m64 m1, __m64 m2)
PUNPCKHBW	__m128i _mm_unpackhi_epi8(__m128i m1, __m128i m2)
PUNPCKHWD	__m64 _mm_unpackhi_pi16(__m64 m1, __m64 m2)
PUNPCKHWD	__m128i _mm_unpackhi_epi16(__m128i m1, __m128i m2)
PUNPCKHDQ	__m64 _mm_unpackhi_pi32(__m64 m1, __m64 m2)
PUNPCKHDQ	__m128i _mm_unpackhi_epi32(__m128i m1, __m128i m2)
PUNPCKHQDQ	__m128i _mm_unpackhi_epi64(__m128i m1, __m128i m2)
PUNPCKLBW	__m64 _mm_unpacklo_pi8 (__m64 m1, __m64 m2)
PUNPCKLBW	__m128i _mm_unpacklo_epi8 (__m128i m1, __m128i m2)
PUNPCKLWD	__m64 _mm_unpacklo_pi16(__m64 m1, __m64 m2)

Table C-1. Simple Intrinsics (Contd.)

Mnemonic	Intrinsic
PUNPCKLWD	__m128i _mm_unpacklo_epi16(__m128i m1, __m128i m2)
PUNPCKLDQ	__m64 _mm_unpacklo_pi32(__m64 m1, __m64 m2)
PUNPCKLDQ	__m128i _mm_unpacklo_epi32(__m128i m1, __m128i m2)
PUNPCKLQDQ	__m128i _mm_unpacklo_epi64(__m128i m1, __m128i m2)
PXOR	__m64 _mm_xor_si64(__m64 m1, __m64 m2)
PXOR	__m128i _mm_xor_si128(__m128i m1, __m128i m2)
RCPSS	__m128 _mm_rcp_ps(__m128 a)
RCPSS	__m128 _mm_rcp_ss(__m128 a)
ROUNDPD	__m128 mm_round_pd(__m128d s1, int iRoundMode)
	__m128 mm_floor_pd(__m128d s1)
	__m128 mm_ceil_pd(__m128d s1)
ROUNDPS	__m128 mm_round_ps(__m128 s1, int iRoundMode)
	__m128 mm_floor_ps(__m128 s1)
	__m128 mm_ceil_ps(__m128 s1)
ROUNDSD	__m128d mm_round_sd(__m128d dst, __m128d s1, int iRoundMode)
	__m128d mm_floor_sd(__m128d dst, __m128d s1)
	__m128d mm_ceil_sd(__m128d dst, __m128d s1)
ROUNDSS	__m128 mm_round_ss(__m128 dst, __m128 s1, int iRoundMode)
	__m128 mm_floor_ss(__m128 dst, __m128 s1)
	__m128 mm_ceil_ss(__m128 dst, __m128 s1)
RSQRTPS	__m128 _mm_rsqrt_ps(__m128 a)
RSQRTSS	__m128 _mm_rsqrt_ss(__m128 a)
SFENCE	void mm_sfence(void)
SHUFPS	__m128d _mm_shuffle_pd(__m128d a, __m128d b, unsigned int imm8)
SHUFPS	__m128 _mm_shuffle_ps(__m128 a, __m128 b, unsigned int imm8)
SQRTPD	__m128d _mm_sqrt_pd(__m128d a)
SQRTPS	__m128 _mm_sqrt_ps(__m128 a)
SQRTSD	__m128d _mm_sqrt_sd(__m128d a)
SQRTSS	__m128 _mm_sqrt_ss(__m128 a)
STMXCSR	_mm_getcsr(void)
SUBPD	__m128d _mm_sub_pd(__m128d a, __m128d b)
SUBPS	__m128 _mm_sub_ps(__m128 a, __m128 b)
SUBSD	__m128d _mm_sub_sd(__m128d a, __m128d b)
SUBSS	__m128 _mm_sub_ss(__m128 a, __m128 b)
UCOMISD	int mm_ucomieq_sd(__m128d a, __m128d b)
	int mm_ucomilt_sd(__m128d a, __m128d b)
	int mm_ucomile_sd(__m128d a, __m128d b)
	int mm_ucomigt_sd(__m128d a, __m128d b)
	int mm_ucomige_sd(__m128d a, __m128d b)
	int mm_ucomineq_sd(__m128d a, __m128d b)
UCOMISS	int mm_ucomieq_ss(__m128 a, __m128 b)

Table C-1. Simple Intrinsics (Contd.)

Mnemonic	Intrinsic
	int __mm_ucomilt_ss(__m128 a, __m128 b)
	int __mm_ucomile_ss(__m128 a, __m128 b)
	int __mm_ucomigt_ss(__m128 a, __m128 b)
	int __mm_ucomige_ss(__m128 a, __m128 b)
	int __mm_ucomineq_ss(__m128 a, __m128 b)
UNPCKHPD	__m128d __mm_unpackhi_pd(__m128d a, __m128d b)
UNPCKHPS	__m128 __mm_unpackhi_ps(__m128 a, __m128 b)
UNPCKLPD	__m128d __mm_unpacklo_pd(__m128d a, __m128d b)
UNPCKLPS	__m128 __mm_unpacklo_ps(__m128 a, __m128 b)
XORPD	__m128d __mm_xor_pd(__m128d a, __m128d b)
XORPS	__m128 __mm_xor_ps(__m128 a, __m128 b)

## C.2 COMPOSITE INTRINSICS

Table C-2. Composite Intrinsics

Mnemonic	Intrinsic
(composite)	__m128i __mm_set_epi64(__m64 q1, __m64 q0)
(composite)	__m128i __mm_set_epi32(int i3, int i2, int i1, int i0)
(composite)	__m128i __mm_set_epi16(short w7, short w6, short w5, short w4, short w3, short w2, short w1, short w0)
(composite)	__m128i __mm_set_epi8(char w15, char w14, char w13, char w12, char w11, char w10, char w9, char w8, char w7, char w6, char w5, char w4, char w3, char w2, char w1, char w0)
(composite)	__m128i __mm_set1_epi64(__m64 q)
(composite)	__m128i __mm_set1_epi32(int a)
(composite)	__m128i __mm_set1_epi16(short a)
(composite)	__m128i __mm_set1_epi8(char a)
(composite)	__m128i __mm_setr_epi64(__m64 q1, __m64 q0)
(composite)	__m128i __mm_setr_epi32(int i3, int i2, int i1, int i0)
(composite)	__m128i __mm_setr_epi16(short w7, short w6, short w5, short w4, short w3, short w2, short w1, short w0)
(composite)	__m128i __mm_setr_epi8(char w15, char w14, char w13, char w12, char w11, char w10, char w9, char w8, char w7, char w6, char w5, char w4, char w3, char w2, char w1, char w0)
(composite)	__m128i __mm_setzero_si128()
(composite)	__m128 __mm_set_ps1(float w) __m128 __mm_set1_ps(float w)
(composite)	__m128cmm_set1_pd(double w)
(composite)	__m128d __mm_set_sd(double w)
(composite)	__m128d __mm_set_pd(double z, double y)
(composite)	__m128 __mm_set_ps(float z, float y, float x, float w)
(composite)	__m128d __mm_setr_pd(double z, double y)
(composite)	__m128 __mm_setr_ps(float z, float y, float x, float w)
(composite)	__m128d __mm_setzero_pd(void)
(composite)	__m128 __mm_setzero_ps(void)

**Table C-2. Composite Ininsics (Contd.)**

<b>Mnemonic</b>	<b>Intrinsic</b>
MOVSD + shuffle	__m128d _mm_load_pd(double * p) __m128d _mm_load1_pd(double *p)
MOVSS + shuffle	__m128 _mm_load_ps1(float * p) __m128 _mm_load1_ps(float *p)
MOVAPD + shuffle	__m128d _mm_loadr_pd(double * p)
MOVAPS + shuffle	__m128 _mm_loadr_ps(float * p)
MOVSD + shuffle	void _mm_store1_pd(double *p, __m128d a)
MOVSS + shuffle	void _mm_store_ps1(float * p, __m128 a) void _mm_store1_ps(float *p, __m128 a)
MOVAPD + shuffle	_mm_storer_pd(double * p, __m128d a)
MOVAPS + shuffle	_mm_storer_ps(float * p, __m128 a)





### 13. Updates to Chapter 8, Volume 3A

Change bars show changes to Chapter 8 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A: System Programming Guide, Part 1*.

-----

Changes include microcode update addition with updates to process.

The Intel 64 and IA-32 architectures provide mechanisms for managing and improving the performance of multiple processors connected to the same system bus. These include:

- Bus locking and/or cache coherency management for performing atomic operations on system memory.
- Serializing instructions.
- An advance programmable interrupt controller (APIC) located on the processor chip (see Chapter 10, “Advanced Programmable Interrupt Controller (APIC)”). This feature was introduced by the Pentium processor.
- A second-level cache (level 2, L2). For the Pentium 4, Intel Xeon, and P6 family processors, the L2 cache is included in the processor package and is tightly coupled to the processor. For the Pentium and Intel486 processors, pins are provided to support an external L2 cache.
- A third-level cache (level 3, L3). For Intel Xeon processors, the L3 cache is included in the processor package and is tightly coupled to the processor.
- Intel Hyper-Threading Technology. This extension to the Intel 64 and IA-32 architectures enables a single processor core to execute two or more threads concurrently (see Section 8.5, “Intel® Hyper-Threading Technology and Intel® Multi-Core Technology”).

These mechanisms are particularly useful in symmetric-multiprocessing (SMP) systems. However, they can also be used when an Intel 64 or IA-32 processor and a special-purpose processor (such as a communications, graphics, or video processor) share the system bus.

These multiprocessing mechanisms have the following characteristics:

- To maintain system memory coherency — When two or more processors are attempting simultaneously to access the same address in system memory, some communication mechanism or memory access protocol must be available to promote data coherency and, in some instances, to allow one processor to temporarily lock a memory location.
- To maintain cache consistency — When one processor accesses data cached on another processor, it must not receive incorrect data. If it modifies data, all other processors that access that data must receive the modified data.
- To allow predictable ordering of writes to memory — In some circumstances, it is important that memory writes be observed externally in precisely the same order as programmed.
- To distribute interrupt handling among a group of processors — When several processors are operating in a system in parallel, it is useful to have a centralized mechanism for receiving interrupts and distributing them to available processors for servicing.
- To increase system performance by exploiting the multi-threaded and multi-process nature of contemporary operating systems and applications.

The caching mechanism and cache consistency of Intel 64 and IA-32 processors are discussed in Chapter 11. The APIC architecture is described in Chapter 10. Bus and memory locking, serializing instructions, memory ordering, and Intel Hyper-Threading Technology are discussed in the following sections.

## 8.1 LOCKED ATOMIC OPERATIONS

The 32-bit IA-32 processors support locked atomic operations on locations in system memory. These operations are typically used to manage shared data structures (such as semaphores, segment descriptors, system segments, or page tables) in which two or more processors may try simultaneously to modify the same field or flag. The processor uses three interdependent mechanisms for carrying out locked atomic operations:

- Guaranteed atomic operations
- Bus locking, using the LOCK# signal and the LOCK instruction prefix

- Cache coherency protocols that ensure that atomic operations can be carried out on cached data structures (cache lock); this mechanism is present in the Pentium 4, Intel Xeon, and P6 family processors

These mechanisms are interdependent in the following ways. Certain basic memory transactions (such as reading or writing a byte in system memory) are always guaranteed to be handled atomically. That is, once started, the processor guarantees that the operation will be completed before another processor or bus agent is allowed access to the memory location. The processor also supports bus locking for performing selected memory operations (such as a read-modify-write operation in a shared area of memory) that typically need to be handled atomically, but are not automatically handled this way. Because frequently used memory locations are often cached in a processor's L1 or L2 caches, atomic operations can often be carried out inside a processor's caches without asserting the bus lock. Here the processor's cache coherency protocols ensure that other processors that are caching the same memory locations are managed properly while atomic operations are performed on cached memory locations.

### NOTE

Where there are contested lock accesses, software may need to implement algorithms that ensure fair access to resources in order to prevent lock starvation. The hardware provides no resource that guarantees fairness to participating agents. It is the responsibility of software to manage the fairness of semaphores and exclusive locking functions.

The mechanisms for handling locked atomic operations have evolved with the complexity of IA-32 processors. More recent IA-32 processors (such as the Pentium 4, Intel Xeon, and P6 family processors) and Intel 64 provide a more refined locking mechanism than earlier processors. These mechanisms are described in the following sections.

### 8.1.1 Guaranteed Atomic Operations

The Intel486 processor (and newer processors since) guarantees that the following basic memory operations will always be carried out atomically:

- Reading or writing a byte
- Reading or writing a word aligned on a 16-bit boundary
- Reading or writing a doubleword aligned on a 32-bit boundary

The Pentium processor (and newer processors since) guarantees that the following additional memory operations will always be carried out atomically:

- Reading or writing a quadword aligned on a 64-bit boundary
- 16-bit accesses to uncached memory locations that fit within a 32-bit data bus

The P6 family processors (and newer processors since) guarantee that the following additional memory operation will always be carried out atomically:

- Unaligned 16-, 32-, and 64-bit accesses to cached memory that fit within a cache line

Accesses to cacheable memory that are split across cache lines and page boundaries are not guaranteed to be atomic by the Intel Core 2 Duo, Intel<sup>®</sup> Atom™, Intel Core Duo, Pentium M, Pentium 4, Intel Xeon, P6 family, Pentium, and Intel486 processors. The Intel Core 2 Duo, Intel Atom, Intel Core Duo, Pentium M, Pentium 4, Intel Xeon, and P6 family processors provide bus control signals that permit external memory subsystems to make split accesses atomic; however, nonaligned data accesses will seriously impact the performance of the processor and should be avoided.

An x87 instruction or an SSE instructions that accesses data larger than a quadword may be implemented using multiple memory accesses. If such an instruction stores to memory, some of the accesses may complete (writing to memory) while another causes the operation to fault for architectural reasons (e.g. due an page-table entry that is marked "not present"). In this case, the effects of the completed accesses may be visible to software even though the overall instruction caused a fault. If TLB invalidation has been delayed (see Section 4.10.4.4), such page faults may occur even if all accesses are to the same page.

## 8.1.2 Bus Locking

Intel 64 and IA-32 processors provide a LOCK# signal that is asserted automatically during certain critical memory operations to lock the system bus or equivalent link. While this output signal is asserted, requests from other processors or bus agents for control of the bus are blocked. Software can specify other occasions when the LOCK semantics are to be followed by prepending the LOCK prefix to an instruction.

In the case of the Intel386, Intel486, and Pentium processors, explicitly locked instructions will result in the assertion of the LOCK# signal. It is the responsibility of the hardware designer to make the LOCK# signal available in system hardware to control memory accesses among processors.

For the P6 and more recent processor families, if the memory area being accessed is cached internally in the processor, the LOCK# signal is generally not asserted; instead, locking is only applied to the processor's caches (see Section 8.1.4, "Effects of a LOCK Operation on Internal Processor Caches").

### 8.1.2.1 Automatic Locking

The operations on which the processor automatically follows the LOCK semantics are as follows:

- When executing an XCHG instruction that references memory.
- **When setting the B (busy) flag of a TSS descriptor** — The processor tests and sets the busy flag in the type field of the TSS descriptor when switching to a task. To ensure that two processors do not switch to the same task simultaneously, the processor follows the LOCK semantics while testing and setting this flag.
- **When updating segment descriptors** — When loading a segment descriptor, the processor will set the accessed flag in the segment descriptor if the flag is clear. During this operation, the processor follows the LOCK semantics so that the descriptor will not be modified by another processor while it is being updated. For this action to be effective, operating-system procedures that update descriptors should use the following steps:
  - Use a locked operation to modify the access-rights byte to indicate that the segment descriptor is not-present, and specify a value for the type field that indicates that the descriptor is being updated.
  - Update the fields of the segment descriptor. (This operation may require several memory accesses; therefore, locked operations cannot be used.)
  - Use a locked operation to modify the access-rights byte to indicate that the segment descriptor is valid and present.
- The Intel386 processor always updates the accessed flag in the segment descriptor, whether it is clear or not. The Pentium 4, Intel Xeon, P6 family, Pentium, and Intel486 processors only update this flag if it is not already set.
- **When updating page-directory and page-table entries** — When updating page-directory and page-table entries, the processor uses locked cycles to set the accessed and dirty flag in the page-directory and page-table entries.
- **Acknowledging interrupts** — After an interrupt request, an interrupt controller may use the data bus to send the interrupt's vector to the processor. The processor follows the LOCK semantics during this time to ensure that no other data appears on the data bus while the vector is being transmitted.

### 8.1.2.2 Software Controlled Bus Locking

To explicitly force the LOCK semantics, software can use the LOCK prefix with the following instructions when they are used to modify a memory location. An invalid-opcode exception (#UD) is generated when the LOCK prefix is used with any other instruction or when no write operation is made to memory (that is, when the destination operand is in a register).

- The bit test and modify instructions (BTS, BTR, and BTC).
- The exchange instructions (XADD, CMPXCHG, and CMPXCHG8B).
- The LOCK prefix is automatically assumed for XCHG instruction.
- The following single-operand arithmetic and logical instructions: INC, DEC, NOT, and NEG.
- The following two-operand arithmetic and logical instructions: ADD, ADC, SUB, SBB, AND, OR, and XOR.

A locked instruction is guaranteed to lock only the area of memory defined by the destination operand, but may be interpreted by the system as a lock for a larger memory area.

Software should access semaphores (shared memory used for signalling between multiple processors) using identical addresses and operand lengths. For example, if one processor accesses a semaphore using a word access, other processors should not access the semaphore using a byte access.

### NOTE

Do not implement semaphores using the WC memory type. Do not perform non-temporal stores to a cache line containing a location used to implement a semaphore.

The integrity of a bus lock is not affected by the alignment of the memory field. The LOCK semantics are followed for as many bus cycles as necessary to update the entire operand. However, it is recommended that locked accesses be aligned on their natural boundaries for better system performance:

- Any boundary for an 8-bit access (locked or otherwise).
- 16-bit boundary for locked word accesses.
- 32-bit boundary for locked doubleword accesses.
- 64-bit boundary for locked quadword accesses.

Locked operations are atomic with respect to all other memory operations and all externally visible events. Only instruction fetch and page table accesses can pass locked instructions. Locked instructions can be used to synchronize data written by one processor and read by another processor.

For the P6 family processors, locked operations serialize all outstanding load and store operations (that is, wait for them to complete). This rule is also true for the Pentium 4 and Intel Xeon processors, with one exception. Load operations that reference weakly ordered memory types (such as the WC memory type) may not be serialized.

Locked instructions should not be used to ensure that data written can be fetched as instructions.

### NOTE

The locked instructions for the current versions of the Pentium 4, Intel Xeon, P6 family, Pentium, and Intel486 processors allow data written to be fetched as instructions. However, Intel recommends that developers who require the use of self-modifying code use a different synchronizing mechanism, described in the following sections.

## 8.1.3 Handling Self- and Cross-Modifying Code

The act of a processor writing data into a currently executing code segment with the intent of executing that data as code is called **self-modifying code**. IA-32 processors exhibit model-specific behavior when executing self-modified code, depending upon how far ahead of the current execution pointer the code has been modified.

As processor microarchitectures become more complex and start to speculatively execute code ahead of the retirement point (as in P6 and more recent processor families), the rules regarding which code should execute, pre- or post-modification, become blurred. To write self-modifying code and ensure that it is compliant with current and future versions of the IA-32 architectures, use one of the following coding options:

(\* OPTION 1 \*)

Store modified code (as data) into code segment;  
Jump to new code or an intermediate location;  
Execute new code;

(\* OPTION 2 \*)

Store modified code (as data) into code segment;  
Execute a serializing instruction; (\* For example, CPUID instruction \*)  
Execute new code;

The use of one of these options is not required for programs intended to run on the Pentium or Intel486 processors, but are recommended to ensure compatibility with the P6 and more recent processor families.

Self-modifying code will execute at a lower level of performance than non-self-modifying or normal code. The degree of the performance deterioration will depend upon the frequency of modification and specific characteristics of the code.

The act of one processor writing data into the currently executing code segment of a second processor with the intent of having the second processor execute that data as code is called **cross-modifying code**. As with self-modifying code, IA-32 processors exhibit model-specific behavior when executing cross-modifying code, depending upon how far ahead of the executing processors current execution pointer the code has been modified.

To write cross-modifying code and ensure that it is compliant with current and future versions of the IA-32 architecture, the following processor synchronization algorithm must be implemented:

```
(* Action of Modifying Processor *)
Memory_Flag ← 0; (* Set Memory_Flag to value other than 1 *)
Store modified code (as data) into code segment;
Memory_Flag ← 1;

(* Action of Executing Processor *)
WHILE (Memory_Flag ≠ 1)
    Wait for code to update;
ELIHW;
Execute serializing instruction; (* For example, CPUID instruction *)
Begin executing modified code;
```

(The use of this option is not required for programs intended to run on the Intel486 processor, but is recommended to ensure compatibility with the Pentium 4, Intel Xeon, P6 family, and Pentium processors.)

Like self-modifying code, cross-modifying code will execute at a lower level of performance than non-cross-modifying (normal) code, depending upon the frequency of modification and specific characteristics of the code.

The restrictions on self-modifying code and cross-modifying code also apply to the Intel 64 architecture.

### 8.1.4 Effects of a LOCK Operation on Internal Processor Caches

For the Intel486 and Pentium processors, the LOCK# signal is always asserted on the bus during a LOCK operation, even if the area of memory being locked is cached in the processor.

For the P6 and more recent processor families, if the area of memory being locked during a LOCK operation is cached in the processor that is performing the LOCK operation as write-back memory and is completely contained in a cache line, the processor may not assert the LOCK# signal on the bus. Instead, it will modify the memory location internally and allow its cache coherency mechanism to ensure that the operation is carried out atomically. This operation is called "cache locking." The cache coherency mechanism automatically prevents two or more processors that have cached the same area of memory from simultaneously modifying data in that area.

## 8.2 MEMORY ORDERING

The term **memory ordering** refers to the order in which the processor issues reads (loads) and writes (stores) through the system bus to system memory. The Intel 64 and IA-32 architectures support several memory-ordering models depending on the implementation of the architecture. For example, the Intel386 processor enforces **program ordering** (generally referred to as **strong ordering**), where reads and writes are issued on the system bus in the order they occur in the instruction stream under all circumstances.

To allow performance optimization of instruction execution, the IA-32 architecture allows departures from strong-ordering model called **processor ordering** in Pentium 4, Intel Xeon, and P6 family processors. These **processor-ordering** variations (called here the **memory-ordering model**) allow performance enhancing operations such as allowing reads to go ahead of buffered writes. The goal of any of these variations is to increase instruction execution speeds, while maintaining memory coherency, even in multiple-processor systems.

Section 8.2.1 and Section 8.2.2 describe the memory-ordering implemented by Intel486, Pentium, Intel Core 2 Duo, Intel Atom, Intel Core Duo, Pentium 4, Intel Xeon, and P6 family processors. Section 8.2.3 gives examples

illustrating the behavior of the memory-ordering model on IA-32 and Intel-64 processors. Section 8.2.4 considers the special treatment of stores for string operations and Section 8.2.5 discusses how memory-ordering behavior may be modified through the use of specific instructions.

### 8.2.1 Memory Ordering in the Intel® Pentium® and Intel486™ Processors

The Pentium and Intel486 processors follow the processor-ordered memory model; however, they operate as strongly-ordered processors under most circumstances. Reads and writes always appear in programmed order at the system bus—except for the following situation where processor ordering is exhibited. Read misses are permitted to go ahead of buffered writes on the system bus when all the buffered writes are cache hits and, therefore, are not directed to the same address being accessed by the read miss.

In the case of I/O operations, both reads and writes always appear in programmed order.

Software intended to operate correctly in processor-ordered processors (such as the Pentium 4, Intel Xeon, and P6 family processors) should not depend on the relatively strong ordering of the Pentium or Intel486 processors. Instead, it should ensure that accesses to shared variables that are intended to control concurrent execution among processors are explicitly required to obey program ordering through the use of appropriate locking or serializing operations (see Section 8.2.5, “Strengthening or Weakening the Memory-Ordering Model”).

### 8.2.2 Memory Ordering in P6 and More Recent Processor Families

The Intel Core 2 Duo, Intel Atom, Intel Core Duo, Pentium 4, and P6 family processors also use a processor-ordered memory-ordering model that can be further defined as “write ordered with store-buffer forwarding.” This model can be characterized as follows.

In a single-processor system for memory regions defined as write-back cacheable, the memory-ordering model respects the following principles (**Note** the memory-ordering principles for single-processor and multiple-processor systems are written from the perspective of software executing on the processor, where the term “processor” refers to a logical processor. For example, a physical processor supporting multiple cores and/or Intel Hyper-Threading Technology is treated as a multi-processor systems.):

- Reads are not reordered with other reads.
- Writes are not reordered with older reads.
- Writes to memory are not reordered with other writes, with the following exceptions:
  - streaming stores (writes) executed with the non-temporal move instructions (MOVNTI, MOVNTQ, MOVNTDQ, MOVNTPS, and MOVNTPD); and
  - string operations (see Section 8.2.4.1).
- No write to memory may be reordered with an execution of the CLFLUSH instruction; a write may be reordered with an execution of the CLFLUSHOPT instruction that flushes a cache line other than the one being written.<sup>1</sup> Executions of the CLFLUSH instruction are not reordered with each other. Executions of CLFLUSHOPT that access different cache lines may be reordered with each other. An execution of CLFLUSHOPT may be reordered with an execution of CLFLUSH that accesses a different cache line.
- Reads may be reordered with older writes to different locations but not with older writes to the same location.
- Reads or writes cannot be reordered with I/O instructions, locked instructions, or serializing instructions.
- Reads cannot pass earlier LFENCE and MFENCE instructions.
- Writes and executions of CLFLUSH and CLFLUSHOPT cannot pass earlier LFENCE, SFENCE, and MFENCE instructions.
- LFENCE instructions cannot pass earlier reads.
- SFENCE instructions cannot pass earlier writes or executions of CLFLUSH and CLFLUSHOPT.
- MFENCE instructions cannot pass earlier reads, writes, or executions of CLFLUSH and CLFLUSHOPT.

---

1. Earlier versions of this manual specified that writes to memory may be reordered with executions of the CLFLUSH instruction. No processors implementing the CLFLUSH instruction allow such reordering.



In a multiple-processor system, the following ordering principles apply:

- Individual processors use the same ordering principles as in a single-processor system.
- Writes by a single processor are observed in the same order by all processors.
- Writes from an individual processor are NOT ordered with respect to the writes from other processors.
- Memory ordering obeys causality (memory ordering respects transitive visibility).
- Any two stores are seen in a consistent order by processors other than those performing the stores
- Locked instructions have a total order.

See the example in Figure 8-1. Consider three processors in a system and each processor performs three writes, one to each of three defined locations (A, B, and C). Individually, the processors perform the writes in the same program order, but because of bus arbitration and other memory access mechanisms, the order that the three processors write the individual memory locations can differ each time the respective code sequences are executed on the processors. The final values in location A, B, and C would possibly vary on each execution of the write sequence.

The processor-ordering model described in this section is virtually identical to that used by the Pentium and Intel486 processors. The only enhancements in the Pentium 4, Intel Xeon, and P6 family processors are:

- Added support for speculative reads, while still adhering to the ordering principles above.
- Store-buffer forwarding, when a read passes a write to the same memory location.
- Out of order store from long string store and string move operations (see Section 8.2.4, "Fast-String Operation and Out-of-Order Stores," below).

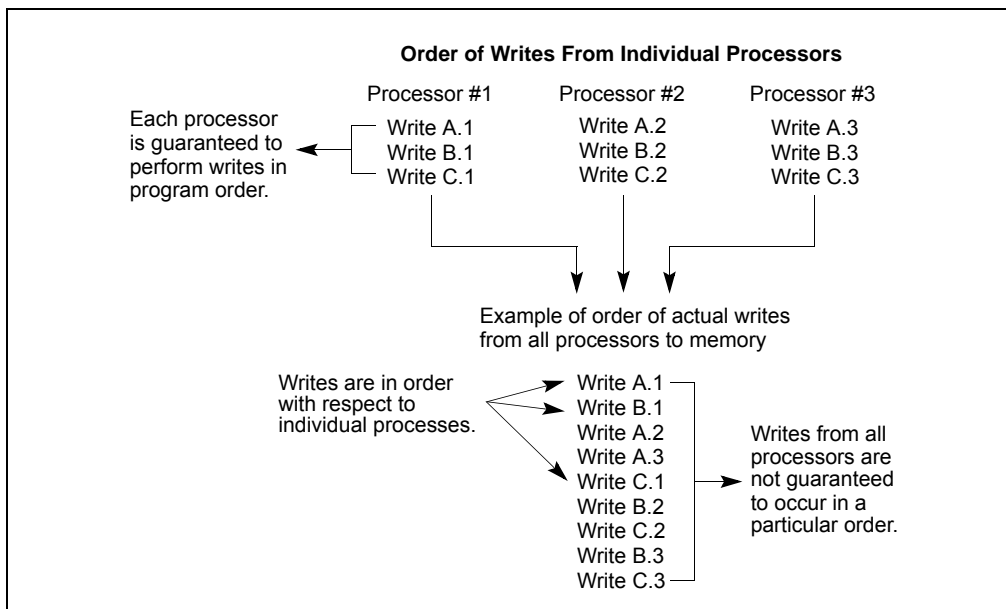


Figure 8-1. Example of Write Ordering in Multiple-Processor Systems

**NOTE**

In P6 processor family, store-buffer forwarding to reads of WC memory from streaming stores to the same address does not occur due to errata.

**8.2.3 Examples Illustrating the Memory-Ordering Principles**

This section provides a set of examples that illustrate the behavior of the memory-ordering principles introduced in Section 8.2.2. They are designed to give software writers an understanding of how memory ordering may affect the results of different sequences of instructions.



These examples are limited to accesses to memory regions defined as write-back cacheable (WB). (Section 8.2.3.1 describes other limitations on the generality of the examples.) The reader should understand that they describe only software-visible behavior. A logical processor may reorder two accesses even if one of examples indicates that they may not be reordered. Such an example states only that software cannot detect that such a reordering occurred. Similarly, a logical processor may execute a memory access more than once as long as the behavior visible to software is consistent with a single execution of the memory access.

### 8.2.3.1 Assumptions, Terminology, and Notation

As noted above, the examples in this section are limited to accesses to memory regions defined as write-back cacheable (WB). They apply only to ordinary loads stores and to locked read-modify-write instructions. They do not necessarily apply to any of the following: out-of-order stores for string instructions (see Section 8.2.4); accesses with a non-temporal hint; reads from memory by the processor as part of address translation (e.g., page walks); and updates to segmentation and paging structures by the processor (e.g., to update “accessed” bits).

The principles underlying the examples in this section apply to individual memory accesses and to locked read-modify-write instructions. The Intel-64 memory-ordering model guarantees that, for each of the following memory-access instructions, the constituent memory operation appears to execute as a single memory access:

- Instructions that read or write a single byte.
- Instructions that read or write a word (2 bytes) whose address is aligned on a 2 byte boundary.
- Instructions that read or write a doubleword (4 bytes) whose address is aligned on a 4 byte boundary.
- Instructions that read or write a quadword (8 bytes) whose address is aligned on an 8 byte boundary.

Any locked instruction (either the XCHG instruction or another read-modify-write instruction with a LOCK prefix) appears to execute as an indivisible and uninterruptible sequence of load(s) followed by store(s) regardless of alignment.

Other instructions may be implemented with multiple memory accesses. From a memory-ordering point of view, there are no guarantees regarding the relative order in which the constituent memory accesses are made. There is also no guarantee that the constituent operations of a store are executed in the same order as the constituent operations of a load.

Section 8.2.3.2 through Section 8.2.3.7 give examples using the MOV instruction. The principles that underlie these examples apply to load and store accesses in general and to other instructions that load from or store to memory. Section 8.2.3.8 and Section 8.2.3.9 give examples using the XCHG instruction. The principles that underlie these examples apply to other locked read-modify-write instructions.

This section uses the term “processor” is to refer to a logical processor. The examples are written using Intel-64 assembly-language syntax and use the following notational conventions:

- Arguments beginning with an “r”, such as r1 or r2 refer to registers (e.g., EAX) visible only to the processor being considered.
- Memory locations are denoted with x, y, z.
- Stores are written as *mov [\_x], val*, which implies that *val* is being stored into the memory location *x*.
- Loads are written as *mov r, [\_x]*, which implies that the contents of the memory location *x* are being loaded into the register *r*.

As noted earlier, the examples refer only to software visible behavior. When the succeeding sections make statement such as “the two stores are reordered,” the implication is only that “the two stores appear to be reordered from the point of view of software.”

### 8.2.3.2 Neither Loads Nor Stores Are Reordered with Like Operations

The Intel-64 memory-ordering model allows neither loads nor stores to be reordered with the same kind of operation. That is, it ensures that loads are seen in program order and that stores are seen in program order. This is illustrated by the following example:

**Example 8-1. Stores Are Not Reordered with Other Stores**

Processor 0	Processor 1
mov [_x], 1 mov [_y], 1	mov r1, [_y] mov r2, [_x]
Initially x = y = 0 r1 = 1 and r2 = 0 is not allowed	

The disallowed return values could be exhibited only if processor 0's two stores are reordered (with the two loads occurring between them) or if processor 1's two loads are reordered (with the two stores occurring between them).

If r1 = 1, the store to y occurs before the load from y. Because the Intel-64 memory-ordering model does not allow stores to be reordered, the earlier store to x occurs before the load from y. Because the Intel-64 memory-ordering model does not allow loads to be reordered, the store to x also occurs before the later load from x. This r2 = 1.

### 8.2.3.3 Stores Are Not Reordered with Earlier Loads

The Intel-64 memory-ordering model ensures that a store by a processor may not occur before a previous load by the same processor. This is illustrated by the following example:

**Example 8-2. Stores Are Not Reordered with Older Loads**

Processor 0	Processor 1
mov r1, [_x] mov [_y], 1	mov r2, [_y] mov [_x], 1
Initially x = y = 0 r1 = 1 and r2 = 1 is not allowed	

Assume r1 = 1.

- Because r1 = 1, processor 1's store to x occurs before processor 0's load from x.
- Because the Intel-64 memory-ordering model prevents each store from being reordered with the earlier load by the same processor, processor 1's load from y occurs before its store to x.
- Similarly, processor 0's load from x occurs before its store to y.
- Thus, processor 1's load from y occurs before processor 0's store to y, implying r2 = 0.

### 8.2.3.4 Loads May Be Reordered with Earlier Stores to Different Locations

The Intel-64 memory-ordering model allows a load to be reordered with an earlier store to a different location. However, loads are not reordered with stores to the same location.

The fact that a load may be reordered with an earlier store to a different location is illustrated by the following example:

**Example 8-3. Loads May be Reordered with Older Stores**

Processor 0	Processor 1
mov [_x], 1 mov r1, [_y]	mov [_y], 1 mov r2, [_x]
Initially x = y = 0 r1 = 0 and r2 = 0 is allowed	

At each processor, the load and the store are to different locations and hence may be reordered. Any interleaving of the operations is thus allowed. One such interleaving has the two loads occurring before the two stores. This would result in each load returning value 0.

The fact that a load may not be reordered with an earlier store to the same location is illustrated by the following example:

**Example 8-4. Loads Are not Reordered with Older Stores to the Same Location**

Processor 0
mov [_x], 1 mov r1, [_x]
Initially x = 0 r1 = 0 is not allowed

The Intel-64 memory-ordering model does not allow the load to be reordered with the earlier store because the accesses are to the same location. Therefore, r1 = 1 must hold.

**8.2.3.5 Intra-Processor Forwarding Is Allowed**

The memory-ordering model allows concurrent stores by two processors to be seen in different orders by those two processors; specifically, each processor may perceive its own store occurring before that of the other. This is illustrated by the following example:

**Example 8-5. Intra-Processor Forwarding is Allowed**

Processor 0	Processor 1
mov [_x], 1 mov r1, [_x] mov r2, [_y]	mov [_y], 1 mov r3, [_y] mov r4, [_x]
Initially x = y = 0 r2 = 0 and r4 = 0 is allowed	

The memory-ordering model imposes no constraints on the order in which the two stores appear to execute by the two processors. This fact allows processor 0 to see its store before seeing processor 1's, while processor 1 sees its store before seeing processor 0's. (Each processor is self consistent.) This allows r2 = 0 and r4 = 0.

In practice, the reordering in this example can arise as a result of store-buffer forwarding. While a store is temporarily held in a processor's store buffer, it can satisfy the processor's own loads but is not visible to (and cannot satisfy) loads by other processors.

**8.2.3.6 Stores Are Transitively Visible**

The memory-ordering model ensures transitive visibility of stores; stores that are causally related appear to all processors to occur in an order consistent with the causality relation. This is illustrated by the following example:

**Example 8-6. Stores Are Transitively Visible**

Processor 0	Processor 1	Processor 2
mov [_x], 1	mov r1, [_x] mov [_y], 1	mov r2, [_y] mov r3, [_x]
Initially x = y = 0 r1 = 1, r2 = 1, r3 = 0 is not allowed		

Assume that r1 = 1 and r2 = 1.

- Because r1 = 1, processor 0's store occurs before processor 1's load.

- Because the memory-ordering model prevents a store from being reordered with an earlier load (see Section 8.2.3.3), processor 1’s load occurs before its store. Thus, processor 0’s store causally precedes processor 1’s store.
- Because processor 0’s store causally precedes processor 1’s store, the memory-ordering model ensures that processor 0’s store appears to occur before processor 1’s store from the point of view of all processors.
- Because  $r2 = 1$ , processor 1’s store occurs before processor 2’s load.
- Because the Intel-64 memory-ordering model prevents loads from being reordered (see Section 8.2.3.2), processor 2’s load occur in order.
- The above items imply that processor 0’s store to x occurs before processor 2’s load from x. This implies that  $r3 = 1$ .

### 8.2.3.7 Stores Are Seen in a Consistent Order by Other Processors

As noted in Section 8.2.3.5, the memory-ordering model allows stores by two processors to be seen in different orders by those two processors. However, any two stores must appear to execute in the same order to all processors other than those performing the stores. This is illustrated by the following example:

**Example 8-7. Stores Are Seen in a Consistent Order by Other Processors**

Processor 0	Processor 1	Processor 2	Processor 3
mov [_x], 1	mov [_y], 1	mov r1, [_x] mov r2, [_y]	mov r3, [_y] mov r4, [_x]
Initially $x = y = 0$ $r1 = 1, r2 = 0, r3 = 1, r4 = 0$ is not allowed			

By the principles discussed in Section 8.2.3.2,

- processor 2’s first and second load cannot be reordered,
- processor 3’s first and second load cannot be reordered.
- If  $r1 = 1$  and  $r2 = 0$ , processor 0’s store appears to precede processor 1’s store with respect to processor 2.
- Similarly,  $r3 = 1$  and  $r4 = 0$  imply that processor 1’s store appears to precede processor 0’s store with respect to processor 1.

Because the memory-ordering model ensures that any two stores appear to execute in the same order to all processors (other than those performing the stores), this set of return values is not allowed

### 8.2.3.8 Locked Instructions Have a Total Order

The memory-ordering model ensures that all processors agree on a single execution order of all locked instructions, including those that are larger than 8 bytes or are not naturally aligned. This is illustrated by the following example:

**Example 8-8. Locked Instructions Have a Total Order**

Processor 0	Processor 1	Processor 2	Processor 3
xchg [_x], r1	xchg [_y], r2	mov r3, [_x] mov r4, [_y]	mov r5, [_y] mov r6, [_x]
Initially $r1 = r2 = 1, x = y = 0$ $r3 = 1, r4 = 0, r5 = 1, r6 = 0$ is not allowed			

Processor 2 and processor 3 must agree on the order of the two executions of XCHG. Without loss of generality, suppose that processor 0’s XCHG occurs first.

- If  $r5 = 1$ , processor 1’s XCHG into y occurs before processor 3’s load from y.

- Because the Intel-64 memory-ordering model prevents loads from being reordered (see Section 8.2.3.2), processor 3’s loads occur in order and, therefore, processor 1’s XCHG occurs before processor 3’s load from x.
- Since processor 0’s XCHG into x occurs before processor 1’s XCHG (by assumption), it occurs before processor 3’s load from x. Thus,  $r_6 = 1$ .

A similar argument (referring instead to processor 2’s loads) applies if processor 1’s XCHG occurs before processor 0’s XCHG.

### 8.2.3.9 Loads and Stores Are Not Reordered with Locked Instructions

The memory-ordering model prevents loads and stores from being reordered with locked instructions that execute earlier or later. The examples in this section illustrate only cases in which a locked instruction is executed before a load or a store. The reader should note that reordering is prevented also if the locked instruction is executed after a load or a store.

The first example illustrates that loads may not be reordered with earlier locked instructions:

#### Example 8-9. Loads Are not Reordered with Locks

Processor 0	Processor 1
xchg [_x], r1 mov r2, [_y]	xchg [_y], r3 mov r4, [_x]
Initially $x = y = 0, r_1 = r_3 = 1$ $r_2 = 0$ and $r_4 = 0$ is not allowed	

As explained in Section 8.2.3.8, there is a total order of the executions of locked instructions. Without loss of generality, suppose that processor 0’s XCHG occurs first.

Because the Intel-64 memory-ordering model prevents processor 1’s load from being reordered with its earlier XCHG, processor 0’s XCHG occurs before processor 1’s load. This implies  $r_4 = 1$ .

A similar argument (referring instead to processor 2’s accesses) applies if processor 1’s XCHG occurs before processor 0’s XCHG.

The second example illustrates that a store may not be reordered with an earlier locked instruction:

#### Example 8-10. Stores Are not Reordered with Locks

Processor 0	Processor 1
xchg [_x], r1 mov [_y], 1	mov r2, [_y] mov r3, [_x]
Initially $x = y = 0, r_1 = 1$ $r_2 = 1$ and $r_3 = 0$ is not allowed	

Assume  $r_2 = 1$ .

- Because  $r_2 = 1$ , processor 0’s store to y occurs before processor 1’s load from y.
- Because the memory-ordering model prevents a store from being reordered with an earlier locked instruction, processor 0’s XCHG into x occurs before its store to y. Thus, processor 0’s XCHG into x occurs before processor 1’s load from y.
- Because the memory-ordering model prevents loads from being reordered (see Section 8.2.3.2), processor 1’s loads occur in order and, therefore, processor 1’s XCHG into x occurs before processor 1’s load from x. Thus,  $r_3 = 1$ .

### 8.2.4 Fast-String Operation and Out-of-Order Stores

Section 7.3.9.3 of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1* described an optimization of repeated string operations called **fast-string operation**.

As explained in that section, the stores produced by fast-string operation may appear to execute out of order. Software dependent upon sequential store ordering should not use string operations for the entire data structure to be stored. Data and semaphores should be separated. Order-dependent code should write to a discrete semaphore variable after any string operations to allow correctly ordered data to be seen by all processors. Atomicity of load and store operations is guaranteed only for native data elements of the string with native data size, and only if they are included in a single cache line.

Section 8.2.4.1 and Section 8.2.4.2 provide further explain and examples.

### 8.2.4.1 Memory-Ordering Model for String Operations on Write-Back (WB) Memory

This section deals with the memory-ordering model for string operations on write-back (WB) memory for the Intel 64 architecture.

The memory-ordering model respects the follow principles:

1. Stores within a single string operation may be executed out of order.
2. Stores from separate string operations (for example, stores from consecutive string operations) do not execute out of order. All the stores from an earlier string operation will complete before any store from a later string operation.
3. String operations are not reordered with other store operations.

Fast string operations (e.g. string operations initiated with the MOVS/STOS instructions and the REP prefix) may be interrupted by exceptions or interrupts. The interrupts are precise but may be delayed - for example, the interruptions may be taken at cache line boundaries, after every few iterations of the loop, or after operating on every few bytes. Different implementations may choose different options, or may even choose not to delay interrupt handling, so software should not rely on the delay. When the interrupt/trap handler is reached, the source/destination registers point to the next string element to be operated on, while the EIP stored in the stack points to the string instruction, and the ECX register has the value it held following the last successful iteration. The return from that trap/interrupt handler should cause the string instruction to be resumed from the point where it was interrupted.

The string operation memory-ordering principles, (item 2 and 3 above) should be interpreted by taking the incorruptibility of fast string operations into account. For example, if a fast string operation gets interrupted after k iterations, then stores performed by the interrupt handler will become visible after the fast string stores from iteration 0 to k, and before the fast string stores from the (k+1)th iteration onward.

Stores within a single string operation may execute out of order (item 1 above) only if fast string operation is enabled. Fast string operations are enabled/disabled through the IA32\_MISC\_ENABLE model specific register.

### 8.2.4.2 Examples Illustrating Memory-Ordering Principles for String Operations

The following examples uses the same notation and convention as described in Section 8.2.3.1.

In Example 8-11, processor 0 does one round of (128 iterations) doubleword string store operation via rep:stosd, writing the value 1 (value in EAX) into a block of 512 bytes from location `_x` (kept in ES:EDI) in ascending order. Since each operation stores a doubleword (4 bytes), the operation is repeated 128 times (value in ECX). The block of memory initially contained 0. Processor 1 is reading two memory locations that are part of the memory block being updated by processor 0, i.e, reading locations in the range `_x` to `(_x+511)`.

#### Example 8-11. Stores Within a String Operation May be Reordered

Processor 0	Processor 1
rep:stosd [ <code>_x</code> ]	mov r1, [ <code>_z</code> ] mov r2, [ <code>_y</code> ]
Initially on processor 0: EAX = 1, ECX=128, ES:EDI = <code>_x</code> Initially [ <code>_x</code> ] to 511[ <code>_x</code> ]= 0, <code>_x</code> <= <code>_y</code> < <code>_z</code> < <code>_x</code> +512 r1 = 1 and r2 = 0 is allowed	

## MULTIPLE-PROCESSOR MANAGEMENT

It is possible for processor 1 to perceive that the repeated string stores in processor 0 are happening out of order. Assume that fast string operations are enabled on processor 0.

In Example 8-12, processor 0 does two separate rounds of `rep stosd` operation of 128 doubleword stores, writing the value 1 (value in EAX) into the first block of 512 bytes from location `_x` (kept in ES:EDI) in ascending order. It then writes 1 into a second block of memory from `(_x+512)` to `(_x+1023)`. All of the memory locations initially contain 0. The block of memory initially contained 0. Processor 1 performs two load operations from the two blocks of memory.

### Example 8-12. Stores Across String Operations Are not Reordered

Processor 0	Processor 1
<code>rep:stosd [_x]</code>	<code>mov r1, [_z]</code>
<code>mov ecx, \$128</code>	<code>mov r2, [_y]</code>
<code>rep:stosd 512[_x]</code>	
Initially on processor 0: EAX = 1, ECX=128, ES:EDI =_x Initially <code>[_x]</code> to <code>1023[_x]</code> = 0, <code>_x &lt;= _y &lt; _x+512 &lt; _z &lt; _x+1024</code> <code>r1 = 1</code> and <code>r2 = 0</code> is not allowed	

It is not possible in the above example for processor 1 to perceive any of the stores from the later string operation (to the second 512 block) in processor 0 before seeing the stores from the earlier string operation to the first 512 block.

The above example assumes that writes to the second block (`_x+512` to `_x+1023`) does not get executed while processor 0's string operation to the first block has been interrupted. If the string operation to the first block by processor 0 is interrupted, and a write to the second memory block is executed by the interrupt handler, then that change in the second memory block will be visible before the string operation to the first memory block resumes.

In Example 8-13, processor 0 does one round of (128 iterations) doubleword string store operation via `rep:stosd`, writing the value 1 (value in EAX) into a block of 512 bytes from location `_x` (kept in ES:EDI) in ascending order. It then writes to a second memory location outside the memory block of the previous string operation. Processor 1 performs two read operations, the first read is from an address outside the 512-byte block but to be updated by processor 0, the second read is from inside the block of memory of string operation.

### Example 8-13. String Operations Are not Reordered with later Stores

Processor 0	Processor 1
<code>rep:stosd [_x]</code>	<code>mov r1, [_z]</code>
<code>mov [_z], \$1</code>	<code>mov r2, [_y]</code>
Initially on processor 0: EAX = 1, ECX=128, ES:EDI =_x Initially <code>[_y] = [_z] = 0</code> , <code>[_x]</code> to <code>511[_x]</code> = 0, <code>_x &lt;= _y &lt; _x+512</code> , <code>_z</code> is a separate memory location <code>r1 = 1</code> and <code>r2 = 0</code> is not allowed	

Processor 1 cannot perceive the later store by processor 0 until it sees all the stores from the string operation. Example 8-13 assumes that processor 0's store to `[_z]` is not executed while the string operation has been interrupted. If the string operation is interrupted and the store to `[_z]` by processor 0 is executed by the interrupt handler, then changes to `[_z]` will become visible before the string operation resumes.

Example 8-14 illustrates the visibility principle when a string operation is interrupted.

**Example 8-14. Interrupted String Operation**

Processor 0	Processor 1
rep:stosd [_x] // interrupted before es:edi reach _y mov [_z], \$1 // interrupt handler	mov r1, [_z] mov r2, [_y]
Initially on processor 0: EAX = 1, ECX=128, ES:EDI = _x Initially [_y] = [_z] = 0, [_x] to 511[_x]= 0, _x <= _y < _x+512, _z is a separate memory location r1 = 1 and r2 = 0 is allowed	

In Example 8-14, processor 0 started a string operation to write to a memory block of 512 bytes starting at address \_x. Processor 0 got interrupted after k iterations of store operations. The address \_y has not yet been updated by processor 0 when processor 0 got interrupted. The interrupt handler that took control on processor 0 writes to the address \_z. Processor 1 may see the store to \_z from the interrupt handler, before seeing the remaining stores to the 512-byte memory block that are executed when the string operation resumes.

Example 8-15 illustrates the ordering of string operations with earlier stores. No store from a string operation can be visible before all prior stores are visible.

**Example 8-15. String Operations Are not Reordered with Earlier Stores**

Processor 0	Processor 1
mov [_z], \$1 rep:stosd [_x]	mov r1, [_y] mov r2, [_z]
Initially on processor 0: EAX = 1, ECX=128, ES:EDI = _x Initially [_y] = [_z] = 0, [_x] to 511[_x]= 0, _x <= _y < _x+512, _z is a separate memory location r1 = 1 and r2 = 0 is not allowed	

**8.2.5 Strengthening or Weakening the Memory-Ordering Model**

The Intel 64 and IA-32 architectures provide several mechanisms for strengthening or weakening the memory-ordering model to handle special programming situations. These mechanisms include:

- The I/O instructions, locking instructions, the LOCK prefix, and serializing instructions force stronger ordering on the processor.
- The SFENCE instruction (introduced to the IA-32 architecture in the Pentium III processor) and the LFENCE and MFENCE instructions (introduced in the Pentium 4 processor) provide memory-ordering and serialization capabilities for specific types of memory operations.
- The memory type range registers (MTRRs) can be used to strengthen or weaken memory ordering for specific area of physical memory (see Section 11.11, "Memory Type Range Registers (MTRRs)"). MTRRs are available only in the Pentium 4, Intel Xeon, and P6 family processors.
- The page attribute table (PAT) can be used to strengthen memory ordering for a specific page or group of pages (see Section 11.12, "Page Attribute Table (PAT)"). The PAT is available only in the Pentium 4, Intel Xeon, and Pentium III processors.

These mechanisms can be used as follows:

Memory mapped devices and other I/O devices on the bus are often sensitive to the order of writes to their I/O buffers. I/O instructions can be used to (the IN and OUT instructions) impose strong write ordering on such accesses as follows. Prior to executing an I/O instruction, the processor waits for all previous instructions in the program to complete and for all buffered writes to drain to memory. Only instruction fetch and page tables walks can pass I/O instructions. Execution of subsequent instructions do not begin until the processor determines that the I/O instruction has been completed.



Synchronization mechanisms in multiple-processor systems may depend upon a strong memory-ordering model. Here, a program can use a locking instruction such as the XCHG instruction or the LOCK prefix to ensure that a read-modify-write operation on memory is carried out atomically. Locking operations typically operate like I/O operations in that they wait for all previous instructions to complete and for all buffered writes to drain to memory (see Section 8.1.2, “Bus Locking”).

Program synchronization can also be carried out with serializing instructions (see Section 8.3). These instructions are typically used at critical procedure or task boundaries to force completion of all previous instructions before a jump to a new section of code or a context switch occurs. Like the I/O and locking instructions, the processor waits until all previous instructions have been completed and all buffered writes have been drained to memory before executing the serializing instruction.

The SFENCE, LFENCE, and MFENCE instructions provide a performance-efficient way of ensuring load and store memory ordering between routines that produce weakly-ordered results and routines that consume that data. The functions of these instructions are as follows:

- **SFENCE** — Serializes all store (write) operations that occurred prior to the SFENCE instruction in the program instruction stream, but does not affect load operations.
- **LFENCE** — Serializes all load (read) operations that occurred prior to the LFENCE instruction in the program instruction stream, but does not affect store operations.<sup>2</sup>
- **MFENCE** — Serializes all store and load operations that occurred prior to the MFENCE instruction in the program instruction stream.

Note that the SFENCE, LFENCE, and MFENCE instructions provide a more efficient method of controlling memory ordering than the CPUID instruction.

The MTRRs were introduced in the P6 family processors to define the cache characteristics for specified areas of physical memory. The following are two examples of how memory types set up with MTRRs can be used strengthen or weaken memory ordering for the Pentium 4, Intel Xeon, and P6 family processors:

- The strong uncached (UC) memory type forces a strong-ordering model on memory accesses. Here, all reads and writes to the UC memory region appear on the bus and out-of-order or speculative accesses are not performed. This memory type can be applied to an address range dedicated to memory mapped I/O devices to force strong memory ordering.
- For areas of memory where weak ordering is acceptable, the write back (WB) memory type can be chosen. Here, reads can be performed speculatively and writes can be buffered and combined. For this type of memory, cache locking is performed on atomic (locked) operations that do not split across cache lines, which helps to reduce the performance penalty associated with the use of the typical synchronization instructions, such as XCHG, that lock the bus during the entire read-modify-write operation. With the WB memory type, the XCHG instruction locks the cache instead of the bus if the memory access is contained within a cache line.

The PAT was introduced in the Pentium III processor to enhance the caching characteristics that can be assigned to pages or groups of pages. The PAT mechanism typically used to strengthen caching characteristics at the page level with respect to the caching characteristics established by the MTRRs. Table 11-7 shows the interaction of the PAT with the MTRRs.

Intel recommends that software written to run on Intel Core 2 Duo, Intel Atom, Intel Core Duo, Pentium 4, Intel Xeon, and P6 family processors assume the processor-ordering model or a weaker memory-ordering model. The Intel Core 2 Duo, Intel Atom, Intel Core Duo, Pentium 4, Intel Xeon, and P6 family processors do not implement a strong memory-ordering model, except when using the UC memory type. Despite the fact that Pentium 4, Intel Xeon, and P6 family processors support processor ordering, Intel does not guarantee that future processors will support this model. To make software portable to future processors, it is recommended that operating systems provide critical region and resource control constructs and API's (application program interfaces) based on I/O, locking, and/or serializing instructions be used to synchronize access to shared areas of memory in multiple-processor systems. Also, software should not depend on processor ordering in situations where the system hardware does not support this memory-ordering model.

- 
2. Specifically, LFENCE does not execute until all prior instructions have completed locally, and no later instruction begins execution until LFENCE completes. As a result, an instruction that loads from memory and that precedes an LFENCE receives data from memory prior to completion of the LFENCE. An LFENCE that follows an instruction that stores to memory might complete before the data being stored have become globally visible. Instructions following an LFENCE may be fetched from memory before the LFENCE, but they will not execute until the LFENCE completes.

## 8.3 SERIALIZING INSTRUCTIONS

The Intel 64 and IA-32 architectures define several **serializing instructions**. These instructions force the processor to complete all modifications to flags, registers, and memory by previous instructions and to drain all buffered writes to memory before the next instruction is fetched and executed. For example, when a MOV to control register instruction is used to load a new value into control register CR0 to enable protected mode, the processor must perform a serializing operation before it enters protected mode. This serializing operation ensures that all operations that were started while the processor was in real-address mode are completed before the switch to protected mode is made.

The concept of serializing instructions was introduced into the IA-32 architecture with the Pentium processor to support parallel instruction execution. Serializing instructions have no meaning for the Intel486 and earlier processors that do not implement parallel instruction execution.

It is important to note that executing of serializing instructions on P6 and more recent processor families constrain speculative execution because the results of speculatively executed instructions are discarded. The following instructions are serializing instructions:

- **Privileged serializing instructions** — INVD, INVEPT, INVLPG, INVVPID, LGDT, LIDT, LLDT, LTR, MOV (to control register, with the exception of MOV CR8<sup>3</sup>), MOV (to debug register), WBINVD, and WRMSR<sup>4</sup>.
- **Non-privileged serializing instructions** — CPUID, IRET, and RSM.

When the processor serializes instruction execution, it ensures that all pending memory transactions are completed (including writes stored in its store buffer) before it executes the next instruction. Nothing can pass a serializing instruction and a serializing instruction cannot pass any other instruction (read, write, instruction fetch, or I/O). For example, CPUID can be executed at any privilege level to serialize instruction execution with no effect on program flow, except that the EAX, EBX, ECX, and EDX registers are modified.

The following instructions are memory-ordering instructions, not serializing instructions. These drain the data memory subsystem. They do not serialize the instruction execution stream:<sup>5</sup>

- **Non-privileged memory-ordering instructions** — SFENCE, LFENCE, and MFENCE.

The SFENCE, LFENCE, and MFENCE instructions provide more granularity in controlling the serialization of memory loads and stores (see Section 8.2.5, “Strengthening or Weakening the Memory-Ordering Model”).

The following additional information is worth noting regarding serializing instructions:

- The processor does not write back the contents of modified data in its data cache to external memory when it serializes instruction execution. Software can force modified data to be written back by executing the WBINVD instruction, which is a serializing instruction. The amount of time or cycles for WBINVD to complete will vary due to the size of different cache hierarchies and other factors. As a consequence, the use of the WBINVD instruction can have an impact on interrupt/event response time.
- When an instruction is executed that enables or disables paging (that is, changes the PG flag in control register CR0), the instruction should be followed by a jump instruction. The target instruction of the jump instruction is fetched with the new setting of the PG flag (that is, paging is enabled or disabled), but the jump instruction itself is fetched with the previous setting. The Pentium 4, Intel Xeon, and P6 family processors do not require the jump operation following the move to register CR0 (because any use of the MOV instruction in a Pentium 4, Intel Xeon, or P6 family processor to write to CR0 is completely serializing). However, to maintain backwards and forward compatibility with code written to run on other IA-32 processors, it is recommended that the jump operation be performed.
- Whenever an instruction is executed to change the contents of CR3 while paging is enabled, the next instruction is fetched using the translation tables that correspond to the new value of CR3. Therefore the next instruction and the sequentially following instructions should have a mapping based upon the new value of CR3. (Global entries in the TLBs are not invalidated, see Section 4.10.4, “Invalidation of TLBs and Paging-Structure Caches.”)

---

3. MOV CR8 is not defined architecturally as a serializing instruction.

4. WRMSR to the IA32\_TSC\_DEADLINE MSR (MSR index 6E0H) and the X2APIC MSRs (MSR indices 802H to 83FH) are not serializing.

5. LFENCE does provide some guarantees on instruction ordering. It does not execute until all prior instructions have completed locally, and no later instruction begins execution until LFENCE completes.

- The Pentium processor and more recent processor families use branch-prediction techniques to improve performance by prefetching the destination of a branch instruction before the branch instruction is executed. Consequently, instruction execution is not deterministically serialized when a branch instruction is executed.

## 8.4 MULTIPLE-PROCESSOR (MP) INITIALIZATION

The IA-32 architecture (beginning with the P6 family processors) defines a multiple-processor (MP) initialization protocol called the *Multiprocessor Specification Version 1.4*. This specification defines the boot protocol to be used by IA-32 processors in multiple-processor systems. (Here, **multiple processors** is defined as two or more processors.) The MP initialization protocol has the following important features:

- It supports controlled booting of multiple processors without requiring dedicated system hardware.
- It allows hardware to initiate the booting of a system without the need for a dedicated signal or a predefined boot processor.
- It allows all IA-32 processors to be booted in the same manner, including those supporting Intel Hyper-Threading Technology.
- The MP initialization protocol also applies to MP systems using Intel 64 processors.

The mechanism for carrying out the MP initialization protocol differs depending on the Intel processor generations. The following bullets summarize the evolution of the changes:

- **For P6 family or older processors supporting MP operations**— The selection of the BSP and APs (see Section 8.4.1, “BSP and AP Processors”) is handled through arbitration on the APIC bus, using BIPI and FIPI messages. These processor generations have CPUID signatures of (family=06H, extended\_model=0, model<=0DH), or family <06H. See Section 8.11.1, “Overview of the MP Initialization Process For P6 Family Processors” for a complete discussion of MP initialization for P6 family processors.
- **Early generations of IA processors with family 0FH** — The selection of the BSP and APs (see Section 8.4.1, “BSP and AP Processors”) is handled through arbitration on the system bus, using BIPI and FIPI messages (see Section 8.4.3, “MP Initialization Protocol Algorithm for MP Systems”). These processor generations have CPUID signatures of family=0FH, model=0H, stepping<=09H.
- **Later generations of IA processors with family 0FH, and IA processors with system bus** — The selection of the BSP and APs is handled through a special system bus cycle, without using BIPI and FIPI message arbitration (see Section 8.4.3, “MP Initialization Protocol Algorithm for MP Systems”). These processor generations have CPUID signatures of family=0FH with (model=0H, stepping>=0AH) or (model >0, all steppings); or family=06H, extended\_model=0, model>=0EH.
- **All other modern IA processor generations supporting MP operations**— The selection of the BSP and APs in the system is handled by platform-specific arrangement of the combination of hardware, BIOS, and/or configuration input options. The basis of the selection mechanism is similar to those of the Later generations of family 0FH and other Intel processor using system bus (see Section 8.4.3, “MP Initialization Protocol Algorithm for MP Systems”). These processor generations have CPUID signatures of family=06H, extended\_model>0.

The family, model, and stepping ID for a processor is given in the EAX register when the CPUID instruction is executed with a value of 1 in the EAX register.

### 8.4.1 BSP and AP Processors

The MP initialization protocol defines two classes of processors: the bootstrap processor (BSP) and the application processors (APs). Following a power-up or RESET of an MP system, system hardware dynamically selects one of the processors on the system bus as the BSP. The remaining processors are designated as APs.

As part of the BSP selection mechanism, the BSP flag is set in the IA32\_APIC\_BASE MSR (see Figure 10-5) of the BSP, indicating that it is the BSP. This flag is cleared for all other processors.

The BSP executes the BIOS’s boot-strap code to configure the APIC environment, sets up system-wide data structures, and starts and initializes the APs. When the BSP and APs are initialized, the BSP then begins executing the operating-system initialization code.

Following a power-up or reset, the APs complete a minimal self-configuration, then wait for a startup signal (a SIPI message) from the BSP processor. Upon receiving a SIPI message, an AP executes the BIOS AP configuration code, which ends with the AP being placed in halt state.

For Intel 64 and IA-32 processors supporting Intel Hyper-Threading Technology, the MP initialization protocol treats each of the logical processors on the system bus or coherent link domain as a separate processor (with a unique APIC ID). During boot-up, one of the logical processors is selected as the BSP and the remainder of the logical processors are designated as APs.

## 8.4.2 MP Initialization Protocol Requirements and Restrictions

The MP initialization protocol imposes the following requirements and restrictions on the system:

- The MP protocol is executed only after a power-up or RESET. If the MP protocol has completed and a BSP is chosen, subsequent INITs (either to a specific processor or system wide) do not cause the MP protocol to be repeated. Instead, each logical processor examines its BSP flag (in the IA32\_APIC\_BASE MSR) to determine whether it should execute the BIOS boot-strap code (if it is the BSP) or enter a wait-for-SIPI state (if it is an AP).
- All devices in the system that are capable of delivering interrupts to the processors must be inhibited from doing so for the duration of the MP initialization protocol. The time during which interrupts must be inhibited includes the window between when the BSP issues an INIT-SIPI-SIPI sequence to an AP and when the AP responds to the last SIPI in the sequence.

## 8.4.3 MP Initialization Protocol Algorithm for MP Systems

Following a power-up or RESET of an MP system, the processors in the system execute the MP initialization protocol algorithm to initialize each of the logical processors on the system bus or coherent link domain. In the course of executing this algorithm, the following boot-up and initialization operations are carried out:

1. Each logical processor is assigned a unique APIC ID, based on system topology. The unique ID is a 32-bit value if the processor supports CPUID leaf 0BH, otherwise the unique ID is an 8-bit value. (see Section 8.4.5, "Identifying Logical Processors in an MP System").
2. Each logical processor is assigned a unique arbitration priority based on its APIC ID.
3. Each logical processor executes its internal BIST simultaneously with the other logical processors in the system.
4. Upon completion of the BIST, the logical processors use a hardware-defined selection mechanism to select the BSP and the APs from the available logical processors on the system bus. The BSP selection mechanism differs depending on the family, model, and stepping IDs of the processors, as follows:
  - Later generations of IA processors within family 0FH (see Section 8.4), IA processors with system bus (family=06H, extended\_model=0, model>=0EH), or all other modern Intel processors (family=06H, extended\_model>0):
    - The logical processors begin monitoring the BNR# signal, which is toggling. When the BNR# pin stops toggling, each processor attempts to issue a NOP special cycle on the system bus.
    - The logical processor with the highest arbitration priority succeeds in issuing a NOP special cycle and is nominated the BSP. This processor sets the BSP flag in its IA32\_APIC\_BASE MSR, then fetches and begins executing BIOS boot-strap code, beginning at the reset vector (physical address FFFF FFF0H).
    - The remaining logical processors (that failed in issuing a NOP special cycle) are designated as APs. They leave their BSP flags in the clear state and enter a "wait-for-SIPI state."
  - Early generations of IA processors within family 0FH (family=0FH, model=0H, stepping<=09H), P6 family or older processors supporting MP operations (family=06H, extended\_model=0, model<=0DH; or family<06H):
    - Each processor broadcasts a BIPI to "all including self." The first processor that broadcasts a BIPI (and thus receives its own BIPI vector), selects itself as the BSP and sets the BSP flag in its IA32\_APIC\_BASE

MSR. (See Section 8.11.1, “Overview of the MP Initialization Process For P6 Family Processors” for a description of the BIPI, FIPI, and SIPI messages.)

- The remainder of the processors (which were not selected as the BSP) are designated as APs. They leave their BSP flags in the clear state and enter a “wait-for-SIPI state.”
  - The newly established BSP broadcasts an FIPI message to “all including self,” which the BSP and APs treat as an end of MP initialization signal. Only the processor with its BSP flag set responds to the FIPI message. It responds by fetching and executing the BIOS boot-strap code, beginning at the reset vector (physical address FFFF FFF0H).
5. As part of the boot-strap code, the BSP creates an ACPI table and/or an MP table and adds its initial APIC ID to these tables as appropriate.
  6. At the end of the boot-strap procedure, the BSP sets a processor counter to 1, then broadcasts a SIPI message to all the APs in the system. Here, the SIPI message contains a vector to the BIOS AP initialization code (at 000VV000H, where VV is the vector contained in the SIPI message).
  7. The first action of the AP initialization code is to set up a race (among the APs) to a BIOS initialization semaphore. The first AP to the semaphore begins executing the initialization code. (See Section 8.4.4, “MP Initialization Example,” for semaphore implementation details.) As part of the AP initialization procedure, the AP adds its APIC ID number to the ACPI and/or MP tables as appropriate and increments the processor counter by 1. At the completion of the initialization procedure, the AP executes a CLI instruction and halts itself.
  8. When each of the APs has gained access to the semaphore and executed the AP initialization code, the BSP establishes a count for the number of processors connected to the system bus, completes executing the BIOS boot-strap code, and then begins executing operating-system boot-strap and start-up code.
  9. While the BSP is executing operating-system boot-strap and start-up code, the APs remain in the halted state. In this state they will respond only to INITs, NMIs, and SMIs. They will also respond to snoops and to assertions of the STPCLK# pin.

The following section gives an example (with code) of the MP initialization protocol for of multiple processors operating in an MP configuration.

Chapter 35, “Model-Specific Registers (MSRs),” describes how to program the LINT[0:1] pins of the processor’s local APICs after an MP configuration has been completed.

## 8.4.4 MP Initialization Example

The following example illustrates the use of the MP initialization protocol used to initialize processors in an MP system after the BSP and APs have been established. The code runs on Intel 64 or IA-32 processors that use a protocol. This includes P6 Family processors, Pentium 4 processors, Intel Core Duo, Intel Core 2 Duo and Intel Xeon processors.

The following constants and data definitions are used in the accompanying code examples. They are based on the addresses of the APIC registers defined in Table 10-1.

```

ICR_LOW      EQU 0FEE00300H
SVR          EQU 0FEE000F0H
APIC_ID      EQU 0FEE00020H
LVT3        EQU 0FEE00370H
APIC_ENABLED EQU 0100H
BOOT_ID      DD ?
COUNT      EQU 00H
VACANT       EQU 00H
    
```

### 8.4.4.1 Typical BSP Initialization Sequence

After the BSP and APs have been selected (by means of a hardware protocol, see Section 8.4.3, “MP Initialization Protocol Algorithm for MP Systems”), the BSP begins executing BIOS boot-strap code (POST) at the normal IA-32 architecture starting address (FFFF FFF0H). The boot-strap code typically performs the following operations:

1. Initializes memory.
2. Loads the microcode update into the processor.
3. Initializes the MTRRs.
4. Enables the caches.
5. Executes the CPUID instruction with a value of 0H in the EAX register, then reads the EBX, ECX, and EDX registers to determine if the BSP is "GenuineIntel."
6. Executes the CPUID instruction with a value of 1H in the EAX register, then saves the values in the EAX, ECX, and EDX registers in a system configuration space in RAM for use later.
7. Loads start-up code for the AP to execute into a 4-KByte page in the lower 1 MByte of memory.
8. Switches to protected mode and ensures that the APIC address space is mapped to the strong uncacheable (UC) memory type.
9. Determine the BSP's APIC ID from the local APIC ID register (default is 0), the code snippet below is an example that applies to logical processors in a system whose local APIC units operate in xAPIC mode that APIC registers are accessed using memory mapped interface:

```

MOV ESI, APIC_ID; Address of local APIC ID register
MOV EAX, [ESI];
AND EAX, 0FF00000H; Zero out all other bits except APIC ID
MOV BOOT_ID, EAX; Save in memory

```

Saves the APIC ID in the ACPI and/or MP tables and optionally in the system configuration space in RAM.

10. Converts the base address of the 4-KByte page for the AP's bootup code into 8-bit vector. The 8-bit vector defines the address of a 4-KByte page in the real-address mode address space (1-MByte space). For example, a vector of 0BDH specifies a start-up memory address of 000BD000H.
11. Enables the local APIC by setting bit 8 of the APIC spurious vector register (SVR).
 

```

MOV ESI, SVR; Address of SVR
MOV EAX, [ESI];
OR EAX, APIC_ENABLED; Set bit 8 to enable (0 on reset)
MOV [ESI], EAX;

```
12. Sets up the LVT error handling entry by establishing an 8-bit vector for the APIC error handler.
 

```

MOV ESI, LVT3;
MOV EAX, [ESI];
AND EAX, 0FFFFFF0H; Clear out previous vector.
OR EAX, 000000xxH; xx is the 8-bit vector the APIC error handler.
MOV [ESI], EAX;

```
13. Initializes the Lock Semaphore variable VACANT to 00H. The APs use this semaphore to determine the order in which they execute BIOS AP initialization code.
14. Performs the following operation to set up the BSP to detect the presence of APs in the system and the number of processors (within a finite duration, minimally 100 milliseconds):
  - Sets the value of the COUNT variable to 1.
  - In the AP BIOS initialization code, the AP will increment the COUNT variable to indicate its presence. The finite duration while waiting for the COUNT to be updated can be accomplished with a timer. When the timer expires, the BSP checks the value of the COUNT variable. If the timer expires and the COUNT variable has not been incremented, no APs are present or some error has occurred.
15. Broadcasts an INIT-SIPI-SIPI IPI sequence to the APs to wake them up and initialize them. If software knows how many logical processors it expects to wake up, it may choose to poll the COUNT variable. If the expected processors show up before the 100 millisecond timer expires, the timer can be canceled and skip to step 16. The left-hand-side of the procedure illustrated in Table 8-1 provides an algorithm when the expected processor count is unknown. The right-hand-side of Table 8-1 can be used when the expected processor count is known.



**Table 8-1. Broadcast INIT-SIPI-SIPI Sequence and Choice of Timeouts**

INIT-SIPI-SIPI when the expected processor count is unknown	INIT-SIPI-SIPI when the expected processor count is known
MOV ESI, ICR_LOw; Load address of ICR low dword into ESI. MOV EAX, 000C4500H; Load ICR encoding for broadcast INIT IPI ; to all APs into EAX. MOV [ESI], EAX; Broadcast INIT IPI to all APs ; 10-millisecond delay loop. MOV EAX, 000C46XXH; Load ICR encoding for broadcast SIPI IP ; to all APs into EAX, where xx is the vector computed in step 10. MOV [ESI], EAX; Broadcast SIPI IPI to all APs ; 200-microsecond delay loop MOV [ESI], EAX; Broadcast second SIPI IPI to all APs ; Waits for the timer interrupt until the timer expires	MOV ESI, ICR_LOw; Load address of ICR low dword into ESI. MOV EAX, 000C4500H; Load ICR encoding for broadcast INIT IPI ; to all APs into EAX. MOV [ESI], EAX; Broadcast INIT IPI to all APs ; 10-millisecond delay loop. MOV EAX, 000C46XXH; Load ICR encoding for broadcast SIPI IP ; to all APs into EAX, where xx is the vector computed in step 10. MOV [ESI], EAX; Broadcast SIPI IPI to all APs ; 200 microsecond delay loop with check to see if COUNT has ; reached the expected processor count. If COUNT reaches ; expected processor count, cancel timer and go to step 16. MOV [ESI], EAX; Broadcast second SIPI IPI to all APs ; Wait for the timer interrupt polling COUNT. If COUNT reaches ; expected processor count, cancel timer and go to step 16. ; If timer expires, go to step 16.

16. Reads and evaluates the COUNT variable and establishes a processor count.

17. If necessary, reconfigures the APIC and continues with the remaining system diagnostics as appropriate.

### 8.4.4.2 Typical AP Initialization Sequence

When an AP receives the SIPI, it begins executing BIOS AP initialization code at the vector encoded in the SIPI. The AP initialization code typically performs the following operations:

1. Waits on the BIOS initialization Lock Semaphore. When control of the semaphore is attained, initialization continues.
2. Loads the microcode update into the processor.
3. Initializes the MTRRs (using the same mapping that was used for the BSP).
4. Enables the cache.
5. Executes the CPUID instruction with a value of 0H in the EAX register, then reads the EBX, ECX, and EDX registers to determine if the AP is "GenuineIntel."
6. Executes the CPUID instruction with a value of 1H in the EAX register, then saves the values in the EAX, ECX, and EDX registers in a system configuration space in RAM for use later.
7. Switches to protected mode and ensures that the APIC address space is mapped to the strong uncacheable (UC) memory type.
8. Determines the AP's APIC ID from the local APIC ID register, and adds it to the MP and ACPI tables and optionally to the system configuration space in RAM.
9. Initializes and configures the local APIC by setting bit 8 in the SVR register and setting up the LVT3 (error LVT) for error handling (as described in steps 9 and 10 in Section 8.4.4.1, "Typical BSP Initialization Sequence").
10. Configures the APs SMI execution environment. (Each AP and the BSP must have a different SMBASE address.)
11. Increments the COUNT variable by 1.
12. Releases the semaphore.
13. Executes one of the following:

- the CLI and HLT instructions (if MONITOR/MWAIT is not supported), or
- the CLI, MONITOR and MWAIT sequence to enter a deep C-state.

14. Waits for an INIT IPI.

### 8.4.5 Identifying Logical Processors in an MP System

After the BIOS has completed the MP initialization protocol, each logical processor can be uniquely identified by its local APIC ID. Software can access these APIC IDs in either of the following ways:

- **Read APIC ID for a local APIC** — Code running on a logical processor can read APIC ID in one of two ways depending on the local APIC unit is operating in x2APIC mode (see *Intel® 64 Architecture x2APIC Specification*) or in xAPIC mode:
  - If the local APIC unit supports x2APIC and is operating in x2APIC mode, 32-bit APIC ID can be read by executing a RDMSR instruction to read the processor's x2APIC ID register. This method is equivalent to executing CPUID leaf 0BH described below.
  - If the local APIC unit is operating in xAPIC mode, 8-bit APIC ID can be read by executing a MOV instruction to read the processor's local APIC ID register (see Section 10.4.6, "Local APIC ID"). This is the ID to use for directing physical destination mode interrupts to the processor.
- **Read ACPI or MP table** — As part of the MP initialization protocol, the BIOS creates an ACPI table and an MP table. These tables are defined in the Multiprocessor Specification Version 1.4 and provide software with a list of the processors in the system and their local APIC IDs. The format of the ACPI table is derived from the ACPI specification, which is an industry standard power management and platform configuration specification for MP systems.
- **Read Initial APIC ID** (If the process does not support CPUID leaf 0BH) — An APIC ID is assigned to a logical processor during power up. This is the initial APIC ID reported by CPUID.1:EBX[31:24] and may be different from the current value read from the local APIC. The initial APIC ID can be used to determine the topological relationship between logical processors for multi-processor systems that do not support CPUID leaf 0BH.
 

Bits in the 8-bit initial APIC ID can be interpreted using several bit masks. Each bit mask can be used to extract an identifier to represent a hierarchical level of the multi-threading resource topology in an MP system (See Section 8.9.1, "Hierarchical Mapping of Shared Resources"). The initial APIC ID may consist of up to four bit-fields. In a non-clustered MP system, the field consists of up to three bit fields.
- **Read 32-bit APIC ID from CPUID leaf 0BH** (If the processor supports CPUID leaf 0BH) — A unique APIC ID is assigned to a logical processor during power up. This APIC ID is reported by CPUID.0BH:EDX[31:0] as a 32-bit value. Use the 32-bit APIC ID and CPUID leaf 0BH to determine the topological relationship between logical processors if the processor supports CPUID leaf 0BH.
 

Bits in the 32-bit x2APIC ID can be extracted into sub-fields using CPUID leaf 0BH parameters. (See Section 8.9.1, "Hierarchical Mapping of Shared Resources").

Figure 8-2 shows two examples of APIC ID bit fields in earlier single-core processors. In single-core Intel Xeon processors, the APIC ID assigned to a logical processor during power-up and initialization is 8 bits. Bits 2:1 form a 2-bit physical package identifier (which can also be thought of as a socket identifier). In systems that configure physical processors in clusters, bits 4:3 form a 2-bit cluster ID. Bit 0 is used in the Intel Xeon processor MP to identify the two logical processors within the package (see Section 8.9.3, "Hierarchical ID of Logical Processors in an MP System"). For Intel Xeon processors that do not support Intel Hyper-Threading Technology, bit 0 is always set to 0; for Intel Xeon processors supporting Intel Hyper-Threading Technology, bit 0 performs the same function as it does for Intel Xeon processor MP.

For more recent multi-core processors, see Section 8.9.1, "Hierarchical Mapping of Shared Resources" for a complete description of the topological relationships between logical processors and bit field locations within an initial APIC ID across Intel 64 and IA-32 processor families.

Note the number of bit fields and the width of bit-fields are dependent on processor and platform hardware capabilities. Software should determine these at runtime. When initial APIC IDs are assigned to logical processors, the value of APIC ID assigned to a logical processor will respect the bit-field boundaries corresponding core, physical package, etc. Additional examples of the bit fields in the initial APIC ID of multi-threading capable systems are shown in Section 8.9.



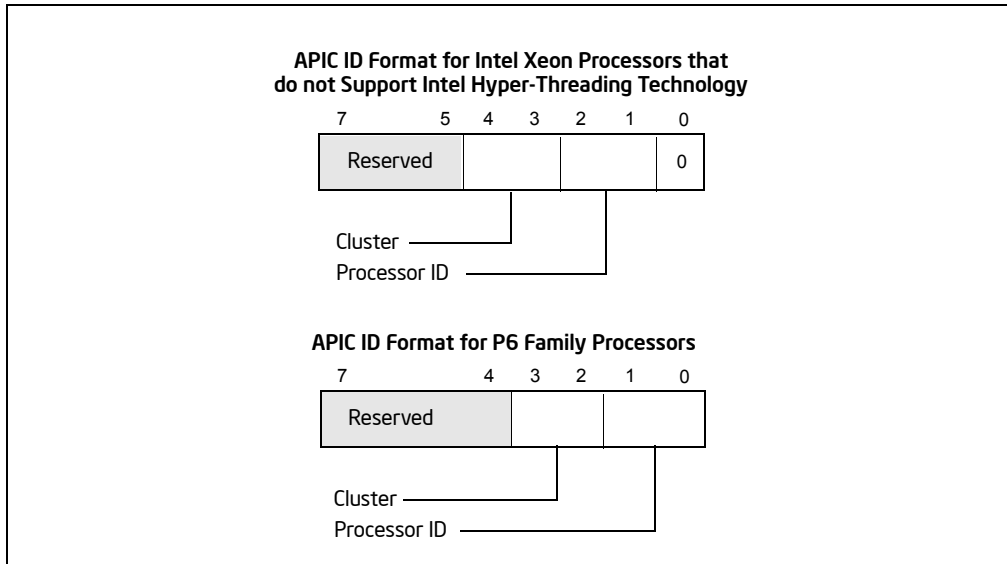


Figure 8-2. Interpretation of APIC ID in Early MP Systems

For P6 family processors, the APIC ID that is assigned to a processor during power-up and initialization is 4 bits (see Figure 8-2). Here, bits 0 and 1 form a 2-bit processor (or socket) identifier and bits 2 and 3 form a 2-bit cluster ID.

## 8.5 INTEL® HYPER-THREADING TECHNOLOGY AND INTEL® MULTI-CORE TECHNOLOGY

Intel Hyper-Threading Technology and Intel multi-core technology are extensions to Intel 64 and IA-32 architectures that enable a single physical processor to execute two or more separate code streams (called *threads*) concurrently. In Intel Hyper-Threading Technology, a single processor core provides two logical processors that share execution resources (see Section 8.7, “Intel® Hyper-Threading Technology Architecture”). In Intel multi-core technology, a physical processor package provides two or more processor cores. Both configurations require chipsets and a BIOS that support the technologies.

Software should not rely on processor names to determine whether a processor supports Intel Hyper-Threading Technology or Intel multi-core technology. Use the CPUID instruction to determine processor capability (see Section 8.6.2, “Initializing Multi-Core Processors”).

## 8.6 DETECTING HARDWARE MULTI-THREADING SUPPORT AND TOPOLOGY

Use the CPUID instruction to detect the presence of hardware multi-threading support in a physical processor. Hardware multi-threading can support several varieties of multigrade and/or Intel Hyper-Threading Technology. CPUID instruction provides several sets of parameter information to aid software enumerating topology information. The relevant topology enumeration parameters provided by CPUID include:

- **Hardware Multi-Threading feature flag (CPUID.1:EDX[28] = 1)** — Indicates when set that the physical package is capable of supporting Intel Hyper-Threading Technology and/or multiple cores.
- **Processor topology enumeration parameters for 8-bit APIC ID:**
  - **Addressable IDs for Logical processors in the same Package (CPUID.1:EBX[23:16])** — Indicates the maximum number of addressable ID for logical processors in a physical package. Within a physical package, there may be addressable IDs that are not occupied by any logical processors. This parameter does not represent the hardware capability of the physical processor.<sup>6</sup>

- **Addressable IDs for processor cores in the same Package<sup>7</sup> (CPUID.(EAX=4, ECX=0<sup>8</sup>):EAX[31:26] + 1 = Y)** — Indicates the maximum number of addressable IDs attributable to processor cores (Y) in the physical package.
- **Extended Processor Topology Enumeration parameters for 32-bit APIC ID:** Intel 64 processors supporting CPUID leaf 0BH will assign unique APIC IDs to each logical processor in the system. CPUID leaf 0BH reports the 32-bit APIC ID and provide topology enumeration parameters. See CPUID instruction reference pages in *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*.

The CPUID feature flag may indicate support for hardware multi-threading when only one logical processor available in the package. In this case, the decimal value represented by bits 16 through 23 in the EBX register will have a value of 1.

Software should note that the number of logical processors enabled by system software may be less than the value of "Addressable IDs for Logical processors". Similarly, the number of cores enabled by system software may be less than the value of "Addressable IDs for processor cores".

Software can detect the availability of the CPUID extended topology enumeration leaf (0BH) by performing two steps:

- Check maximum input value for basic CPUID information by executing CPUID with EAX= 0. If CPUID.0H:EAX is greater than or equal to 11 (0BH), then proceed to next step,
- Check CPUID.EAX=0BH, ECX=0H:EBX is non-zero.

If both of the above conditions are true, extended topology enumeration leaf is available. Note the presence of CPUID leaf 0BH in a processor does not guarantee support that the local APIC supports x2APIC. If CPUID.(EAX=0BH, ECX=0H):EBX returns zero and maximum input value for basic CPUID information is greater than 0BH, then CPUID.0BH leaf is not supported on that processor.

## 8.6.1 Initializing Processors Supporting Hyper-Threading Technology

The initialization process for an MP system that contains processors supporting Intel Hyper-Threading Technology is the same as for conventional MP systems (see Section 8.4, "Multiple-Processor (MP) Initialization"). One logical processor in the system is selected as the BSP and other processors (or logical processors) are designated as APs. The initialization process is identical to that described in Section 8.4.3, "MP Initialization Protocol Algorithm for MP Systems," and Section 8.4.4, "MP Initialization Example."

During initialization, each logical processor is assigned an APIC ID that is stored in the local APIC ID register for each logical processor. If two or more processors supporting Intel Hyper-Threading Technology are present, each logical processor on the system bus is assigned a unique ID (see Section 8.9.3, "Hierarchical ID of Logical Processors in an MP System"). Once logical processors have APIC IDs, software communicates with them by sending APIC IPI messages.

## 8.6.2 Initializing Multi-Core Processors

The initialization process for an MP system that contains multi-core Intel 64 or IA-32 processors is the same as for conventional MP systems (see Section 8.4, "Multiple-Processor (MP) Initialization"). A logical processor in one core is selected as the BSP; other logical processors are designated as APs.

During initialization, each logical processor is assigned an APIC ID. Once logical processors have APIC IDs, software may communicate with them by sending APIC IPI messages.

- 
6. Operating system and BIOS may implement features that reduce the number of logical processors available in a platform to applications at runtime to less than the number of physical packages times the number of hardware-capable logical processors per package.
  7. Software must check CPUID for its support of leaf 4 when implementing support for multi-core. If CPUID leaf 4 is not available at runtime, software should handle the situation as if there is only one core per package.
  8. Maximum number of cores in the physical package must be queried by executing CPUID with EAX=4 and a valid ECX input value. Valid ECX input values start from 0.

### 8.6.3 Executing Multiple Threads on an Intel® 64 or IA-32 Processor Supporting Hardware Multi-Threading

Upon completing the operating system boot-up procedure, the bootstrap processor (BSP) executes operating system code. Other logical processors are placed in the halt state. To execute a code stream (thread) on a halted logical processor, the operating system issues an interprocessor interrupt (IPI) addressed to the halted logical processor. In response to the IPI, the processor wakes up and begins executing the code identified by the vector received as part of the IPI.

To manage execution of multiple threads on logical processors, an operating system can use conventional symmetric multiprocessing (SMP) techniques. For example, the operating-system can use a time-slice or load balancing mechanism to periodically interrupt each of the active logical processors. Upon interrupting a logical processor, the operating system checks its run queue for a thread waiting to be executed and dispatches the thread to the interrupted logical processor.

### 8.6.4 Handling Interrupts on an IA-32 Processor Supporting Hardware Multi-Threading

Interrupts are handled on processors supporting Intel Hyper-Threading Technology as they are on conventional MP systems. External interrupts are received by the I/O APIC, which distributes them as interrupt messages to specific logical processors (see Figure 8-3).

Logical processors can also send IPIs to other logical processors by writing to the ICR register of its local APIC (see Section 10.6, “Issuing Interprocessor Interrupts”). This also applies to dual-core processors.

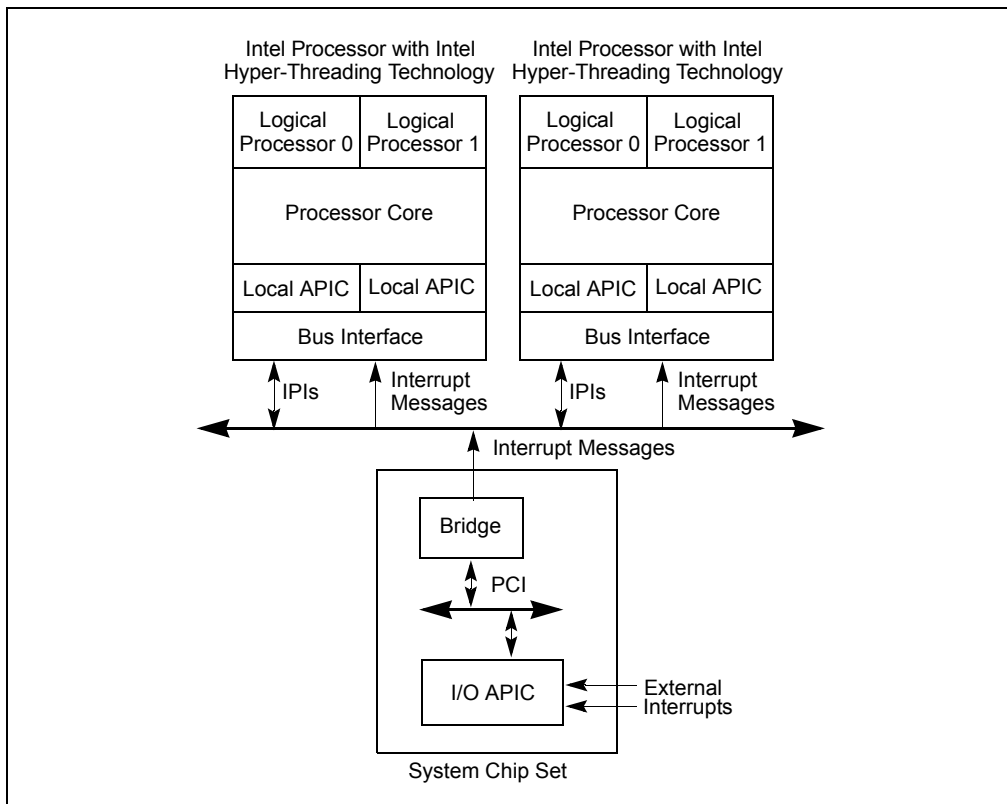


Figure 8-3. Local APICs and I/O APIC in MP System Supporting Intel HT Technology

## 8.7 INTEL® HYPER-THREADING TECHNOLOGY ARCHITECTURE

Figure 8-4 shows a generalized view of an Intel processor supporting Intel Hyper-Threading Technology, using the original Intel Xeon processor MP as an example. This implementation of the Intel Hyper-Threading Technology

consists of two logical processors (each represented by a separate architectural state) which share the processor's execution engine and the bus interface. Each logical processor also has its own advanced programmable interrupt controller (APIC).

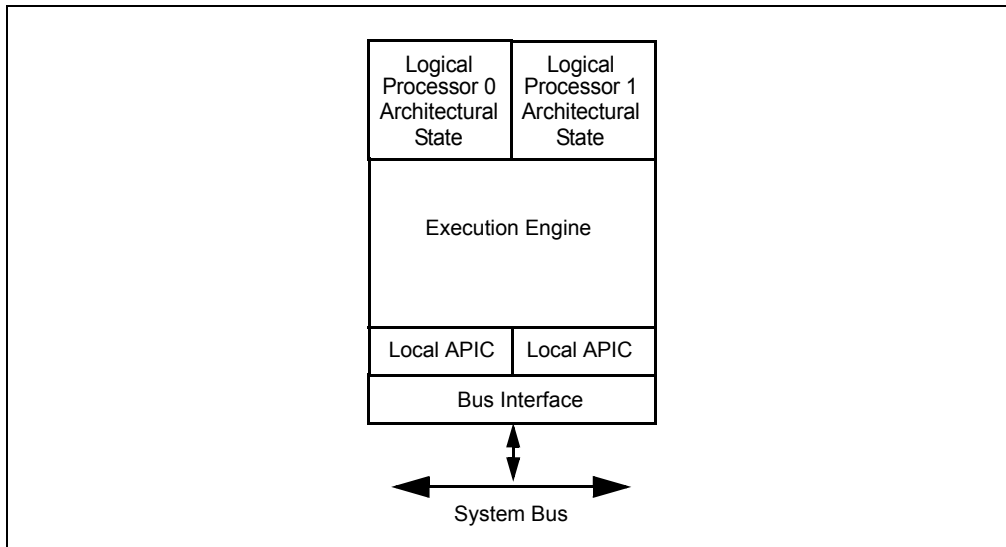


Figure 8-4. IA-32 Processor with Two Logical Processors Supporting Intel HT Technology

### 8.7.1 State of the Logical Processors

The following features are part of the architectural state of logical processors within Intel 64 or IA-32 processors supporting Intel Hyper-Threading Technology. The features can be subdivided into three groups:

- Duplicated for each logical processor
- Shared by logical processors in a physical processor
- Shared or duplicated, depending on the implementation

The following features are duplicated for each logical processor:

- General purpose registers (EAX, EBX, ECX, EDX, ESI, EDI, ESP, and EBP)
- Segment registers (CS, DS, SS, ES, FS, and GS)
- EFLAGS and EIP registers. Note that the CS and EIP/RIP registers for each logical processor point to the instruction stream for the thread being executed by the logical processor.
- x87 FPU registers (ST0 through ST7, status word, control word, tag word, data operand pointer, and instruction pointer)
- MMX registers (MM0 through MM7)
- XMM registers (XMM0 through XMM7) and the MXCSR register
- Control registers and system table pointer registers (GDTR, LDTR, IDTR, task register)
- Debug registers (DR0, DR1, DR2, DR3, DR6, DR7) and the debug control MSRs
- Machine check global status (IA32\_MCG\_STATUS) and machine check capability (IA32\_MCG\_CAP) MSRs
- Thermal clock modulation and ACPI Power management control MSRs
- Time stamp counter MSRs
- Most of the other MSR registers, including the page attribute table (PAT). See the exceptions below.
- Local APIC registers.
- Additional general purpose registers (R8-R15), XMM registers (XMM8-XMM15), control register, IA32\_EFER on Intel 64 processors.

The following features are shared by logical processors:

- Memory type range registers (MTRRs)

Whether the following features are shared or duplicated is implementation-specific:

- IA32\_MISC\_ENABLE MSR (MSR address 1A0H)
- Machine check architecture (MCA) MSRs (except for the IA32\_MCG\_STATUS and IA32\_MCG\_CAP MSRs)
- Performance monitoring control and counter MSRs

### 8.7.2 APIC Functionality

When a processor supporting Intel Hyper-Threading Technology support is initialized, each logical processor is assigned a local APIC ID (see Table 10-1). The local APIC ID serves as an ID for the logical processor and is stored in the logical processor's APIC ID register. If two or more processors supporting Intel Hyper-Threading Technology are present in a dual processor (DP) or MP system, each logical processor on the system bus is assigned a unique local APIC ID (see Section 8.9.3, "Hierarchical ID of Logical Processors in an MP System").

Software communicates with local processors using the APIC's interprocessor interrupt (IPI) messaging facility. Setup and programming for APICs is identical in processors that support and do not support Intel Hyper-Threading Technology. See Chapter 10, "Advanced Programmable Interrupt Controller (APIC)," for a detailed discussion.

### 8.7.3 Memory Type Range Registers (MTRR)

MTRRs in a processor supporting Intel Hyper-Threading Technology are shared by logical processors. When one logical processor updates the setting of the MTRRs, settings are automatically shared with the other logical processors in the same physical package.

The architectures require that all MP systems based on Intel 64 and IA-32 processors (this includes logical processors) must use an identical MTRR memory map. This gives software a consistent view of memory, independent of the processor on which it is running. See Section 11.11, "Memory Type Range Registers (MTRRs)," for information on setting up MTRRs.

### 8.7.4 Page Attribute Table (PAT)

Each logical processor has its own PAT MSR (IA32\_PAT). However, as described in Section 11.12, "Page Attribute Table (PAT)," the PAT MSR settings must be the same for all processors in a system, including the logical processors.

### 8.7.5 Machine Check Architecture

In the Intel HT Technology context as implemented by processors based on Intel NetBurst<sup>®</sup> microarchitecture, all of the machine check architecture (MCA) MSRs (except for the IA32\_MCG\_STATUS and IA32\_MCG\_CAP MSRs) are duplicated for each logical processor. This permits logical processors to initialize, configure, query, and handle machine-check exceptions simultaneously within the same physical processor. The design is compatible with machine check exception handlers that follow the guidelines given in Chapter 15, "Machine-Check Architecture."

The IA32\_MCG\_STATUS MSR is duplicated for each logical processor so that its machine check in progress bit field (MCIP) can be used to detect recursion on the part of MCA handlers. In addition, the MSR allows each logical processor to determine that a machine-check exception is in progress independent of the actions of another logical processor in the same physical package.

Because the logical processors within a physical package are tightly coupled with respect to shared hardware resources, both logical processors are notified of machine check errors that occur within a given physical processor. If machine-check exceptions are enabled when a fatal error is reported, all the logical processors within a physical package are dispatched to the machine-check exception handler. If machine-check exceptions are disabled, the logical processors enter the shutdown state and assert the IERR# signal.

When enabling machine-check exceptions, the MCE flag in control register CR4 should be set for each logical processor.

On Intel Atom family processors that support Intel Hyper-Threading Technology, the MCA facilities are shared between all logical processors on the same processor core.

## 8.7.6 Debug Registers and Extensions

Each logical processor has its own set of debug registers (DR0, DR1, DR2, DR3, DR6, DR7) and its own debug control MSR. These can be set to control and record debug information for each logical processor independently. Each logical processor also has its own last branch records (LBR) stack.

## 8.7.7 Performance Monitoring Counters

Performance counters and their companion control MSRs are shared between the logical processors within a processor core for processors based on Intel NetBurst microarchitecture. As a result, software must manage the use of these resources. The performance counter interrupts, events, and precise event monitoring support can be set up and allocated on a per thread (per logical processor) basis.

See Section 18.16, “Performance Monitoring and Intel Hyper-Threading Technology in Processors Based on Intel NetBurst<sup>®</sup> Microarchitecture,” for a discussion of performance monitoring in the Intel Xeon processor MP.

In Intel Atom processor family that support Intel Hyper-Threading Technology, the performance counters (general-purpose and fixed-function counters) and their companion control MSRs are duplicated for each logical processor.

## 8.7.8 IA32\_MISC\_ENABLE MSR

The IA32\_MISC\_ENABLE MSR (MSR address 1A0H) is generally shared between the logical processors in a processor core supporting Intel Hyper-Threading Technology. However, some bit fields within IA32\_MISC\_ENABLE MSR may be duplicated per logical processor. The partition of shared or duplicated bit fields within IA32\_MISC\_ENABLE is implementation dependent. Software should program duplicated fields carefully on all logical processors in the system to ensure consistent behavior.

## 8.7.9 Memory Ordering

The logical processors in an Intel 64 or IA-32 processor supporting Intel Hyper-Threading Technology obey the same rules for memory ordering as Intel 64 or IA-32 processors without Intel HT Technology (see Section 8.2, “Memory Ordering”). Each logical processor uses a processor-ordered memory model that can be further defined as “write-ordered with store buffer forwarding.” All mechanisms for strengthening or weakening the memory-ordering model to handle special programming situations apply to each logical processor.

## 8.7.10 Serializing Instructions

As a general rule, when a logical processor in a processor supporting Intel Hyper-Threading Technology executes a serializing instruction, only that logical processor is affected by the operation. An exception to this rule is the execution of the WBINVD, INVD, and WRMSR instructions; and the MOV CR instruction when the state of the CD flag in control register CR0 is modified. Here, both logical processors are serialized.

## 8.7.11 Microcode Update Resources

In an Intel processor supporting Intel Hyper-Threading Technology, the microcode update facilities are shared between the logical processors; either logical processor can initiate an update. Each logical processor has its own BIOS signature MSR (IA32\_BIOS\_SIGN\_ID at MSR address 8BH). When a logical processor performs an update for the physical processor, the IA32\_BIOS\_SIGN\_ID MSRs for resident logical processors are updated with identical information. If logical processors initiate an update simultaneously, the processor core provides the necessary synchronization needed to ensure that only one update is performed at a time.

**NOTE**

Some processors (prior to the introduction of Intel 64 Architecture and based on Intel NetBurst microarchitecture) do not support simultaneous loading of microcode update to the sibling logical processors in the same core. All other processors support logical processors initiating an update simultaneously. Intel recommends a common approach that the microcode loader use the sequential technique described in Section 9.11.6.3.

**8.7.12 Self Modifying Code**

Intel processors supporting Intel Hyper-Threading Technology support self-modifying code, where data writes modify instructions cached or currently in flight. They also support cross-modifying code, where on an MP system writes generated by one processor modify instructions cached or currently in flight on another. See Section 8.1.3, "Handling Self- and Cross-Modifying Code," for a description of the requirements for self- and cross-modifying code in an IA-32 processor.

**8.7.13 Implementation-Specific Intel HT Technology Facilities**

The following non-architectural facilities are implementation-specific in IA-32 processors supporting Intel Hyper-Threading Technology:

- Caches
- Translation lookaside buffers (TLBs)
- Thermal monitoring facilities

The Intel Xeon processor MP implementation is described in the following sections.

**8.7.13.1 Processor Caches**

For processors supporting Intel Hyper-Threading Technology, the caches are shared. Any cache manipulation instruction that is executed on one logical processor has a global effect on the cache hierarchy of the physical processor. Note the following:

- **WBINVD instruction** — The entire cache hierarchy is invalidated after modified data is written back to memory. All logical processors are stopped from executing until after the write-back and invalidate operation is completed. A special bus cycle is sent to all caching agents. The amount of time or cycles for WBINVD to complete will vary due to the size of different cache hierarchies and other factors. As a consequence, the use of the WBINVD instruction can have an impact on interrupt/event response time.
- **INVD instruction** — The entire cache hierarchy is invalidated without writing back modified data to memory. All logical processors are stopped from executing until after the invalidate operation is completed. A special bus cycle is sent to all caching agents.
- **CLFLUSH and CLFLUSHOPT instructions** — The specified cache line is invalidated from the cache hierarchy after any modified data is written back to memory and a bus cycle is sent to all caching agents, regardless of which logical processor caused the cache line to be filled.
- **CD flag in control register CR0** — Each logical processor has its own CR0 control register, and thus its own CD flag in CR0. The CD flags for the two logical processors are ORed together, such that when any logical processor sets its CD flag, the entire cache is nominally disabled.

**8.7.13.2 Processor Translation Lookaside Buffers (TLBs)**

In processors supporting Intel Hyper-Threading Technology, data cache TLBs are shared. The instruction cache TLB may be duplicated or shared in each logical processor, depending on implementation specifics of different processor families.

Entries in the TLBs are tagged with an ID that indicates the logical processor that initiated the translation. This tag applies even for translations that are marked global using the page-global feature for memory paging. See Section 4.10, "Caching Translation Information," for information about global translations.



When a logical processor performs a TLB invalidation operation, only the TLB entries that are tagged for that logical processor are guaranteed to be flushed. This protocol applies to all TLB invalidation operations, including writes to control registers CR3 and CR4 and uses of the INVLPG instruction.

### 8.7.13.3 Thermal Monitor

In a processor that supports Intel Hyper-Threading Technology, logical processors share the catastrophic shutdown detector and the automatic thermal monitoring mechanism (see Section 14.7, “Thermal Monitoring and Protection”). Sharing results in the following behavior:

- If the processor’s core temperature rises above the preset catastrophic shutdown temperature, the processor core halts execution, which causes both logical processors to stop execution.
- When the processor’s core temperature rises above the preset automatic thermal monitor trip temperature, the frequency of the processor core is automatically modulated, which effects the execution speed of both logical processors.

For software controlled clock modulation, each logical processor has its own IA32\_CLOCK\_MODULATION MSR, allowing clock modulation to be enabled or disabled on a logical processor basis. Typically, if software controlled clock modulation is going to be used, the feature must be enabled for all the logical processors within a physical processor and the modulation duty cycle must be set to the same value for each logical processor. If the duty cycle values differ between the logical processors, the processor clock will be modulated at the highest duty cycle selected.

### 8.7.13.4 External Signal Compatibility

This section describes the constraints on external signals received through the pins of a processor supporting Intel Hyper-Threading Technology and how these signals are shared between its logical processors.

- **STPCLK#** — A single STPCLK# pin is provided on the physical package of the Intel Xeon processor MP. External control logic uses this pin for power management within the system. When the STPCLK# signal is asserted, the processor core transitions to the stop-grant state, where instruction execution is halted but the processor core continues to respond to snoop transactions. Regardless of whether the logical processors are active or halted when the STPCLK# signal is asserted, execution is stopped on both logical processors and neither will respond to interrupts.

In MP systems, the STPCLK# pins on all physical processors are generally tied together. As a result this signal affects all the logical processors within the system simultaneously.

- **LINT0 and LINT1 pins** — A processor supporting Intel Hyper-Threading Technology has only one set of LINT0 and LINT1 pins, which are shared between the logical processors. When one of these pins is asserted, both logical processors respond unless the pin has been masked in the APIC local vector tables for one or both of the logical processors.

Typically in MP systems, the LINT0 and LINT1 pins are not used to deliver interrupts to the logical processors. Instead all interrupts are delivered to the local processors through the I/O APIC.

- **A20M# pin** — On an IA-32 processor, the A20M# pin is typically provided for compatibility with the Intel 286 processor. Asserting this pin causes bit 20 of the physical address to be masked (forced to zero) for all external bus memory accesses. Processors supporting Intel Hyper-Threading Technology provide one A20M# pin, which affects the operation of both logical processors within the physical processor.

The functionality of A20M# is used primarily by older operating systems and not used by modern operating systems. On newer Intel 64 processors, A20M# may be absent.

## 8.8 MULTI-CORE ARCHITECTURE

This section describes the architecture of Intel 64 and IA-32 processors supporting dual-core and quad-core technology. The discussion is applicable to the Intel Pentium processor Extreme Edition, Pentium D, Intel Core Duo, Intel Core 2 Duo, Dual-core Intel Xeon processor, Intel Core 2 Quad processors, and quad-core Intel Xeon processors. Features vary across different microarchitectures and are detectable using CPUID.



In general, each processor core has dedicated microarchitectural resources identical to a single-processor implementation of the underlying microarchitecture without hardware multi-threading capability. Each logical processor in a dual-core processor (whether supporting Intel Hyper-Threading Technology or not) has its own APIC functionality, PAT, machine check architecture, debug registers and extensions. Each logical processor handles serialization instructions or self-modifying code on its own. Memory order is handled the same way as in Intel Hyper-Threading Technology.

The topology of the cache hierarchy (with respect to whether a given cache level is shared by one or more processor cores or by all logical processors in the physical package) depends on the processor implementation. Software must use the deterministic cache parameter leaf of CPUID instruction to discover the cache-sharing topology between the logical processors in a multi-threading environment.

### 8.8.1 Logical Processor Support

The topological composition of processor cores and logical processors in a multi-core processor can be discovered using CPUID. Within each processor core, one or more logical processors may be available.

System software must follow the requirement MP initialization sequences (see Section 8.4, “Multiple-Processor (MP) Initialization”) to recognize and enable logical processors. At runtime, software can enumerate those logical processors enabled by system software to identify the topological relationships between these logical processors. (See Section 8.9.5, “Identifying Topological Relationships in a MP System”).

### 8.8.2 Memory Type Range Registers (MTRR)

MTRR is shared between two logical processors sharing a processor core if the physical processor supports Intel Hyper-Threading Technology. MTRR is not shared between logical processors located in different cores or different physical packages.

The Intel 64 and IA-32 architectures require that all logical processors in an MP system use an identical MTRR memory map. This gives software a consistent view of memory, independent of the processor on which it is running.

See Section 11.11, “Memory Type Range Registers (MTRRs).”

### 8.8.3 Performance Monitoring Counters

Performance counters and their companion control MSRs are shared between two logical processors sharing a processor core if the processor core supports Intel Hyper-Threading Technology and is based on Intel NetBurst microarchitecture. They are not shared between logical processors in different cores or different physical packages. As a result, software must manage the use of these resources, based on the topology of performance monitoring resources. Performance counter interrupts, events, and precise event monitoring support can be set up and allocated on a per thread (per logical processor) basis.

See Section 18.16, “Performance Monitoring and Intel Hyper-Threading Technology in Processors Based on Intel NetBurst<sup>®</sup> Microarchitecture.”

### 8.8.4 IA32\_MISC\_ENABLE MSR

Some bit fields in IA32\_MISC\_ENABLE MSR (MSR address 1A0H) may be shared between two logical processors sharing a processor core, or may be shared between different cores in a physical processor. See Chapter 35, “Model-Specific Registers (MSRs).”

### 8.8.5 Microcode Update Resources

Microcode update facilities are shared between two logical processors sharing a processor core if the physical package supports Intel Hyper-Threading Technology. They are not shared between logical processors in different

cores or different physical packages. Either logical processor that has access to the microcode update facility can initiate an update.

Each logical processor has its own BIOS signature MSR (IA32\_BIOS\_SIGN\_ID at MSR address 8BH). When a logical processor performs an update for the physical processor, the IA32\_BIOS\_SIGN\_ID MSRs for resident logical processors are updated with identical information.

All microcode update steps during processor initialization should use the same update data on all cores in all physical packages of the same stepping. Any subsequent microcode update must apply consistent update data to all cores in all physical packages of the same stepping. If the processor detects an attempt to load an older microcode update when a newer microcode update had previously been loaded, it may reject the older update to stay with the newer update.

#### NOTE

Some processors (prior to the introduction of Intel 64 Architecture and based on Intel NetBurst microarchitecture) do not support simultaneous loading of microcode update to the sibling logical processors in the same core. All other processors support logical processors initiating an update simultaneously. Intel recommends a common approach that the microcode loader use the sequential technique described in Section 9.11.6.3.

## 8.9 PROGRAMMING CONSIDERATIONS FOR HARDWARE MULTI-THREADING CAPABLE PROCESSORS

In a multi-threading environment, there may be certain hardware resources that are physically shared at some level of the hardware topology. In the multi-processor systems, typically bus and memory sub-systems are physically shared between multiple sockets. Within a hardware multi-threading capable processors, certain resources are provided for each processor core, while other resources may be provided for each logical processors (see Section 8.7, “Intel® Hyper-Threading Technology Architecture,” and Section 8.8, “Multi-Core Architecture”).

From a software programming perspective, control transfer of processor operation is managed at the granularity of logical processor (operating systems dispatch a runnable task by allocating an available logical processor on the platform). To manage the topology of shared resources in a multi-threading environment, it may be useful for software to understand and manage resources that are shared by more than one logical processors.

### 8.9.1 Hierarchical Mapping of Shared Resources

The APIC\_ID value associated with each logical processor in a multi-processor system is unique (see Section 8.6, “Detecting Hardware Multi-Threading Support and Topology”). This 8-bit or 32-bit value can be decomposed into sub-fields, where each sub-field corresponds a hierarchical level of the topological mapping of hardware resources.

The decomposition of an APIC\_ID may consist of several sub fields representing the topology within a physical processor package, the higher-order bits of an APIC ID may also be used by cluster vendors to represent the topology of cluster nodes of each coherent multiprocessor systems. If the processor does not support CPUID leaf 0BH, the 8-bit initial APIC ID can represent 4 levels of hierarchy:

- **Cluster** — Some multi-threading environments consists of multiple clusters of multi-processor systems. The CLUSTER\_ID sub-field is usually supported by vendor firmware to distinguish different clusters. For non-clustered systems, CLUSTER\_ID is usually 0 and system topology is reduced to three levels of hierarchy.
- **Package** — A multi-processor system consists of two or more sockets, each mates with a physical processor package. The PACKAGE\_ID sub-field distinguishes different physical packages within a cluster.
- **Core** — A physical processor package consists of one or more processor cores. The CORE\_ID sub-field distinguishes processor cores in a package. For a single-core processor, the width of this bit field is 0.
- **SMT** — A processor core provides one or more logical processors sharing execution resources. The SMT\_ID sub-field distinguishes logical processors in a core. The width of this bit field is non-zero if a processor core provides more than one logical processors.

SMT and CORE sub-fields are bit-wise contiguous in the APIC\_ID field (see Figure 8-5).

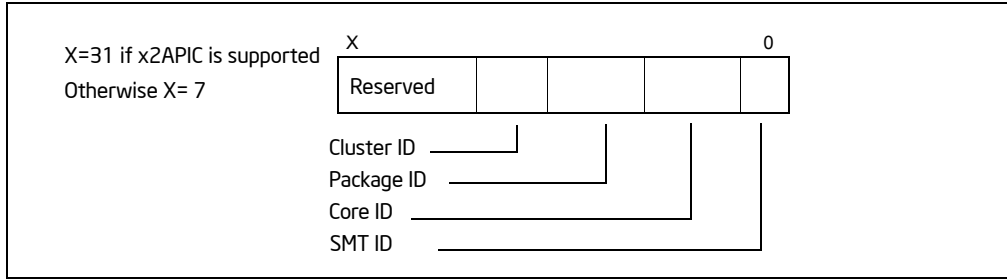


Figure 8-5. Generalized Four level Interpretation of the APIC ID

If the processor supports CPUID leaf 0BH, the 32-bit APIC ID can represent cluster plus several levels of topology within the physical processor package. The exact number of hierarchical levels within a physical processor package must be enumerated through CPUID leaf 0BH. Common processor families may employ topology similar to that represented by 8-bit Initial APIC ID. In general, CPUID leaf 0BH can support topology enumeration algorithm that decompose a 32-bit APIC ID into more than four sub-fields (see Figure 8-6).

The width of each sub-field depends on hardware and software configurations. Field widths can be determined at runtime using the algorithm discussed below (Example 8-16 through Example 8-20).

Figure 7-6 depicts the relationships of three of the hierarchical sub-fields in a hypothetical MP system. The value of valid APIC\_IDs need not be contiguous across package boundary or core boundaries.

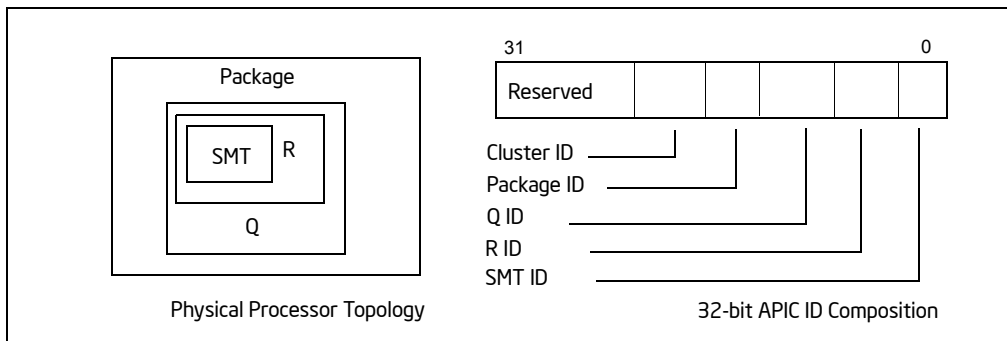


Figure 8-6. Conceptual Five-level Topology and 32-bit APIC ID Composition

### 8.9.2 Hierarchical Mapping of CPUID Extended Topology Leaf

CPUID leaf 0BH provides enumeration parameters for software to identify each hierarchy of the processor topology in a deterministic manner. Each hierarchical level of the topology starting from the SMT level is represented numerically by a sub-leaf index within the CPUID 0BH leaf. Each level of the topology is mapped to a sub-field in the APIC ID, following the general relationship depicted in Figure 8-6. This mechanism allows software to query the exact number of levels within a physical processor package and the bit-width of each sub-field of x2APIC ID directly. For example,

- Starting from sub-leaf index 0 and incrementing ECX until CPUID.(EAX=0BH, ECX=N):ECX[15:8] returns an invalid "level type" encoding. The number of levels within the physical processor package is "N" (excluding PACKAGE). Using Figure 8-6 as an example, CPUID.(EAX=0BH, ECX=3):ECX[15:8] will report 00H, indicating sub leaf 03H is invalid. This is also depicted by a pseudo code example:

**Example 8-16. Number of Levels Below the Physical Processor Package**

```

Byte type = 1;
s = 0;
While ( type ) {
    EAX = 0BH; // query each sub leaf of CPUID leaf 0BH
    ECX = s;
    CPUID;
    type = ECX[15:8]; // examine level type encoding
    s ++;
}
N = ECX[7:0];

```

- Sub-leaf index 0 (ECX= 0 as input) provides enumeration parameters to extract the SMT sub-field of x2APIC ID. If EAX = 0BH, and ECX =0 is specified as input when executing CPUID, CPUID.(EAX=0BH, ECX=0):EAX[4:0] reports a value (a right-shift count) that allow software to extract part of x2APIC ID to distinguish the next higher topological entities above the SMT level. This value also corresponds to the bit-width of the sub-field of x2APIC ID corresponding the hierarchical level with sub-leaf index 0.
- For each subsequent higher sub-leaf index m, CPUID.(EAX=0BH, ECX=m):EAX[4:0] reports the right-shift count that will allow software to extract part of x2APIC ID to distinguish higher-level topological entities. This means the right-shift value at of sub-leaf m, corresponds to the least significant (m+1) subfields of the 32-bit x2APIC ID.

**Example 8-17. BitWidth Determination of x2APIC ID Subfields**

```

For m = 0, m < N, m ++;
{ cumulative_width[m] = CPUID.(EAX=0BH, ECX= m): EAX[4:0]; }
BitWidth[0] = cumulative_width[0];
For m = 1, m < N, m ++;
    BitWidth[m] = cumulative_width[m] - cumulative_width[m-1];

```

Currently, only the following encoding of hierarchical level type are defined: 0 (invalid), 1 (SMT), and 2 (core). Software must not assume any "level type" encoding value to be related to any sub-leaf index, except sub-leaf 0.

Example 8-16 and Example 8-17 represent the general technique for using CPUID leaf 0BH to enumerate processor topology of more than two levels of hierarchy inside a physical package. Most processor families to date requires only "SMT" and "CORE" levels within a physical package. The examples in later sections will focus on these three-level topology only.

**8.9.3 Hierarchical ID of Logical Processors in an MP System**

For Intel 64 and IA-32 processors, system hardware establishes an 8-bit initial APIC ID (or 32-bit APIC ID if the processor supports CPUID leaf 0BH) that is unique for each logical processor following power-up or RESET (see Section 8.6.1). Each logical processor on the system is allocated an initial APIC ID. BIOS may implement features that tell the OS to support less than the total number of logical processors on the system bus. Those logical processors that are not available to applications at runtime are halted during the OS boot process. As a result, the number valid local APIC\_IDS that can be queried by `affinitizing-current-thread-context` (See Example 8-22) is limited to the number of logical processors enabled at runtime by the OS boot process.

Table 8-2 shows an example of the 8-bit APIC IDs that are initially reported for logical processors in a system with four Intel Xeon MP processors that support Intel Hyper-Threading Technology (a total of 8 logical processors, each physical package has two processor cores and supports Intel Hyper-Threading Technology). Of the two logical processors within a Intel Xeon processor MP, logical processor 0 is designated the primary logical processor and logical processor 1 as the secondary logical processor.

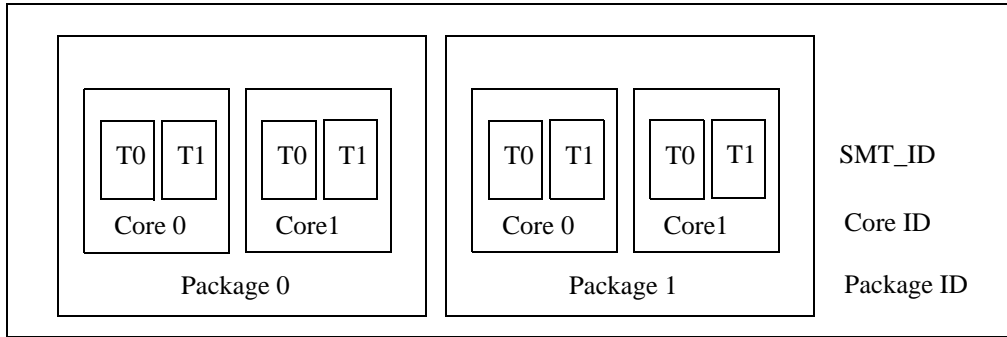


Figure 8-7. Topological Relationships between Hierarchical IDs in a Hypothetical MP Platform

Table 8-2. Initial APIC IDs for the Logical Processors in a System that has Four Intel Xeon MP Processors Supporting Intel Hyper-Threading Technology<sup>1</sup>

Initial APIC ID	Package ID	Core ID	SMT ID
0H	0H	0H	0H
1H	0H	0H	1H
2H	1H	0H	0H
3H	1H	0H	1H
4H	2H	0H	0H
5H	2H	0H	1H
6H	3H	0H	0H
7H	3H	0H	1H

**NOTE:**

1. Because information on the number of processor cores in a physical package was not available in early single-core processors supporting Intel Hyper-Threading Technology, the core ID can be treated as 0.

Table 8-3 shows the initial APIC IDs for a hypothetical situation with a dual processor system. Each physical package providing two processor cores, and each processor core also supporting Intel Hyper-Threading Technology.

Table 8-3. Initial APIC IDs for the Logical Processors in a System that has Two Physical Processors Supporting Dual-Core and Intel Hyper-Threading Technology

Initial APIC ID	Package ID	Core ID	SMT ID
0H	0H	0H	0H
1H	0H	0H	1H
2H	0H	1H	0H
3H	0H	1H	1H
4H	1H	0H	0H
5H	1H	0H	1H
6H	1H	1H	0H
7H	1H	1H	1H

### 8.9.3.1 Hierarchical ID of Logical Processors with x2APIC ID

Table 8-4 shows an example of possible x2APIC ID assignments for a dual processor system that support x2APIC. Each physical package providing four processor cores, and each processor core also supporting Intel Hyper-Threading Technology. Note that the x2APIC ID need not be contiguous in the system.

**Table 8-4. Example of Possible x2APIC ID Assignment in a System that has Two Physical Processors Supporting x2APIC and Intel Hyper-Threading Technology**

x2APIC ID	Package ID	Core ID	SMT ID
0H	0H	0H	0H
1H	0H	0H	1H
2H	0H	1H	0H
3H	0H	1H	1H
4H	0H	2H	0H
5H	0H	2H	1H
6H	0H	3H	0H
7H	0H	3H	1H
10H	1H	0H	0H
11H	1H	0H	1H
12H	1H	1H	0H
13H	1H	1H	1H
14H	1H	2H	0H
15H	1H	2H	1H
16H	1H	3H	0H
17H	1H	3H	1H

### 8.9.4 Algorithm for Three-Level Mappings of APIC\_ID

Software can gather the initial APIC\_IDs for each logical processor supported by the operating system at runtime<sup>9</sup> and extract identifiers corresponding to the three levels of sharing topology (package, core, and SMT). The three-level algorithms below focus on a non-clustered MP system for simplicity. They do not assume APIC IDs are contiguous or that all logical processors on the platform are enabled.

Intel supports multi-threading systems where all physical processors report identical values in CPUID leaf 0BH, CPUID.1:EBX[23:16]), CPUID.4<sup>10</sup>:EAX[31:26], and CPUID.4<sup>11</sup>:EAX[25:14]. The algorithms below assume the target system has symmetry across physical package boundaries with respect to the number of logical processors per package, number of cores per package, and cache topology within a package.

The extraction algorithm (for three-level mappings from an APIC ID) uses the general procedure depicted in Example 8-18, and is supplemented by more detailed descriptions on the derivation of topology enumeration parameters for extraction bit masks:

1. Detect hardware multi-threading support in the processor.

9. As noted in Section 8.6 and Section 8.9.3, the number of logical processors supported by the OS at runtime may be less than the total number logical processors available in the platform hardware.

10. Maximum number of addressable ID for processor cores in a physical processor is obtained by executing CPUID with EAX=4 and a valid ECX index, The ECX index start at 0.

11. Maximum number addressable ID for processor cores sharing the target cache level is obtained by executing CPUID with EAX = 4 and the ECX index corresponding to the target cache level.

2. Derive a set of bit masks that can extract the sub ID of each hierarchical level of the topology. The algorithm to derive extraction bit masks for SMT\_ID/CORE\_ID/PACKAGE\_ID differs based on APIC ID is 32-bit (see step 3 below) or 8-bit (see step 4 below):
3. If the processor supports CPUID leaf 0BH, each APIC ID contains a 32-bit value, the topology enumeration parameters needed to derive three-level extraction bit masks are:
  - a. Query the right-shift value for the SMT level of the topology using CPUID leaf 0BH with ECX =0H as input. The number of bits to shift-right on x2APIC ID (EAX[4:0]) can distinguish different higher-level entities above SMT (e.g. processor cores) in the same physical package. This is also the width of the bit mask to extract the SMT\_ID.
  - b. Query CPUID leaf 0BH for the amount of bit shift to distinguish next higher-level entities (e.g. physical processor packages) in the system. This describes an explicit three-level-topology situation for commonly available processors. Consult Example 8-17 to adapt to situations beyond three-level topology of a physical processor. The width of the extraction bit mask can be used to derive the cumulative extraction bitmask to extract the sub IDs of logical processors (including different processor cores) in the same physical package. The extraction bit mask to distinguish merely different processor cores can be derived by xor'ing the SMT extraction bit mask from the cumulative extraction bit mask.
  - c. Query the 32-bit x2APIC ID for the logical processor where the current thread is executing.
  - d. Derive the extraction bit masks corresponding to SMT\_ID, CORE\_ID, and PACKAGE\_ID, starting from SMT\_ID.
  - e. Apply each extraction bit mask to the 32-bit x2APIC ID to extract sub-field IDs.
4. If the processor does not support CPUID leaf 0BH, each initial APIC ID contains an 8-bit value, the topology enumeration parameters needed to derive extraction bit masks are:
  - a. Query the size of address space for sub IDs that can accommodate logical processors in a physical processor package. This size parameters (CPUID.1:EBX[23:16]) can be used to derive the width of an extraction bitmask to enumerate the sub IDs of different logical processors in the same physical package.
  - b. Query the size of address space for sub IDs that can accommodate processor cores in a physical processor package. This size parameters can be used to derive the width of an extraction bitmask to enumerate the sub IDs of processor cores in the same physical package.
  - c. Query the 8-bit initial APIC ID for the logical processor where the current thread is executing.
  - d. Derive the extraction bit masks using respective address sizes corresponding to SMT\_ID, CORE\_ID, and PACKAGE\_ID, starting from SMT\_ID.
  - e. Apply each extraction bit mask to the 8-bit initial APIC ID to extract sub-field IDs.

### Example 8-18. Support Routines for Detecting Hardware Multi-Threading and Identifying the Relationships Between Package, Core and Logical Processors

#### 1. Detect support for Hardware Multi-Threading Support in a processor.

```
// Returns a non-zero value if CPUID reports the presence of hardware multi-threading
// support in the physical package where the current logical processor is located.
// This does not guarantee BIOS or OS will enable all logical processors in the physical
// package and make them available to applications.
// Returns zero if hardware multi-threading is not present.
```

```
#define HWMT_BIT 10000000H

unsigned int HWMTSupported(void)
{
    // ensure cpuid instruction is supported
    // execute cpuid with eax = 0 to get vendor string
    // execute cpuid with eax = 1 to get feature flag and signature

    // Check to see if this a Genuine Intel Processor

    if (vendor string EQ GenuineIntel) {
        return (feature_flag_edx & HWMT_BIT); // bit 28
    }
    return 0;
}
```

### Example 8-19. Support Routines for Identifying Package, Core and Logical Processors from 32-bit x2APIC ID

#### a. Derive the extraction bitmask for logical processors in a processor core and associated mask offset for different cores.

```
int DeriveSMT_Mask_Offsets (void)
{
    if (!HWMTSupported()) return -1;
    execute cpuid with eax = 11, ECX = 0;
    If (returned level type encoding in ECX[15:8] does not match SMT) return -1;
    Mask_SMT_shift = EAX[4:0]; // # bits shift right of APIC ID to distinguish different cores
    SMT_MASK = ~((-1) << Mask_SMT_shift); // shift left to derive extraction bitmask for SMT_ID
    return 0;
}
```



- b. **Derive the extraction bitmask for processor cores in a physical processor package and associated mask offset for different packages.**

```
int DeriveCore_Mask_Offsets (void)
{
    if (!HWMTSupported()) return -1;
    execute cpuid with eax = 11, ECX = 0;
    while( ECX[15:8] ) { // level type encoding is valid
        If (returned level type encoding in ECX[15:8] matches CORE) {
            Mask_Core_shift = EAX[4:0]; // needed to distinguish different physical packages
            COREPlusSMT_MASK = ~( (-1) << Mask_Core_shift);
            CORE_MASK = COREPlusSMT_MASK ^ SMT_MASK;
            PACKAGE_MASK = (-1) << Mask_Core_shift;
            return 0
        }
        ECX ++;
        execute cpuid with eax = 11;
    }
    return -1;
}
```

- c. **Query the x2APIC ID of a logical processor.**

APIC\_IDs for each logical processor.

```
unsigned char Getx2APIC_ID (void)
{
    unsigned reg_edx = 0;
    execute cpuid with eax = 11, ECX = 0
    store returned value of edx
    return (unsigned) (reg_edx);
}
```

**Example 8-20. Support Routines for Identifying Package, Core and Logical Processors from 8-bit Initial APIC ID**

- a. **Find the size of address space for logical processors in a physical processor package.**

```
#define NUM_LOGICAL_BITS 00FF0000H
// Use the mask above and CPUID.1.EBX[23:16] to obtain the max number of addressable IDs
// for logical processors in a physical package,
```

```
//Returns the size of address space of logical processors in a physical processor package;
// Software should not assume the value to be a power of 2.
```

```
unsigned char MaxLPIDsPerPackage(void)
{
    if (!HWMTSupported()) return 1;
    execute cpuid with eax = 1
    store returned value of ebx
    return (unsigned char) ((reg_ebx & NUM_LOGICAL_BITS) >> 16);
}
```

**b. Find the size of address space for processor cores in a physical processor package.**

```
// Returns the max number of addressable IDs for processor cores in a physical processor package;
// Software should not assume cpuid reports this value to be a power of 2.
```

```
unsigned MaxCoreIDsPerPackage(void)
{
    if (!HWMTSupported()) return (unsigned char) 1;
    if cpuid supports leaf number 4
    { // we can retrieve multi-core topology info using leaf 4
        execute cpuid with eax = 4, ecx = 0
        store returned value of eax
        return (unsigned) ((reg_eax >> 26) + 1);
    }
    else // must be a single-core processor
        return 1;
}
```

**c. Query the initial APIC ID of a logical processor.**

```
#define INITIAL_APIC_ID_BITS FF00000H // CPUID.1.EBX[31:24] initial APIC ID
```

```
// Returns the 8-bit unique initial APIC ID for the processor running the code.
// Software can use OS services to affinitize the current thread to each logical processor
// available under the OS to gather the initial APIC_IDs for each logical processor.
```

```
unsigned GetInitAPIC_ID (void)
{
    unsigned int reg_ebx = 0;
    execute cpuid with eax = 1
    store returned value of ebx
    return (unsigned) ((reg_ebx & INITIAL_APIC_ID_BITS) >> 24);
}
```

**d. Find the width of an extraction bitmask from the maximum count of the bit-field (address size).**

```
// Returns the mask bit width of a bit field from the maximum count that bit field can represent.
// This algorithm does not assume 'address size' to have a value equal to power of 2.
// Address size for SMT_ID can be calculated from MaxLPIDsPerPackage()/MaxCoreIDsPerPackage()
// Then use the routine below to derive the corresponding width of SMT extraction bitmask
// Address size for CORE_ID is MaxCoreIDsPerPackage(),
// Derive the bitwidth for CORE extraction mask similarly
```

```
unsigned FindMaskWidth(unsigned Max_Count)
{
    unsigned int mask_width, cnt = Max_Count;
    __asm {
        mov eax, cnt
        mov ecx, 0
        mov mask_width, ecx
        dec eax
        bsr cx, ax
        jz next
        inc cx
        mov mask_width, ecx
        next:
        mov eax, mask_width
    }
    return mask_width;
}
```

**e. Extract a sub ID from an 8-bit full ID, using address size of the sub ID and shift count.**

```
// The routine below can extract SMT_ID, CORE_ID, and PACKAGE_ID respectively from the init APIC_ID
// To extract SMT_ID, MaxSubIDvalue is set to the address size of SMT_ID, Shift_Count = 0
// To extract CORE_ID, MaxSubIDvalue is the address size of CORE_ID, Shift_Count is width of SMT extraction bitmask.
// Returns the value of the sub ID, this is not a zero-based value
```

```
Unsigned char GetSubID(unsigned char Full_ID, unsigned char MaxSubIDvalue, unsigned char Shift_Count)
{
    MaskWidth = FindMaskWidth(MaxSubIDvalue);
    MaskBits = ((uchar) (FFH << Shift_Count)) ^ ((uchar) (FFH << Shift_Count + MaskWidth));
    SubID = Full_ID & MaskBits;
    Return SubID;
}
```

Software must not assume local APIC\_ID values in an MP system are consecutive. Non-consecutive local APIC\_IDs may be the result of hardware configurations or debug features implemented in the BIOS or OS.

An identifier for each hierarchical level can be extracted from an 8-bit APIC\_ID using the support routines illustrated in Example 8-20. The appropriate bit mask and shift value to construct the appropriate bit mask for each level must be determined dynamically at runtime.

### 8.9.5 Identifying Topological Relationships in a MP System

To detect the number of physical packages, processor cores, or other topological relationships in a MP system, the following procedures are recommended:

- Extract the three-level identifiers from the APIC ID of each logical processor enabled by system software. The sequence is as follows (See the pseudo code shown in Example 8-21 and support routines shown in Example 8-18):

- The extraction start from the right-most bit field, corresponding to SMT\_ID, the innermost hierarchy in a three-level topology (See Figure 8-7). For the right-most bit field, the shift value of the working mask is zero. The width of the bit field is determined dynamically using the maximum number of logical processor per core, which can be derived from information provided from CPUID.
- To extract the next bit-field, the shift value of the working mask is determined from the width of the bit mask of the previous step. The width of the bit field is determined dynamically using the maximum number of cores per package.
- To extract the remaining bit-field, the shift value of the working mask is determined from the maximum number of logical processor per package. So the remaining bits in the APIC ID (excluding those bits already extracted in the two previous steps) are extracted as the third identifier. This applies to a non-clustered MP system, or if there is no need to distinguish between PACKAGE\_ID and CLUSTER\_ID.

If there is need to distinguish between PACKAGE\_ID and CLUSTER\_ID, PACKAGE\_ID can be extracted using an algorithm similar to the extraction of CORE\_ID, assuming the number of physical packages in each node of a clustered system is symmetric.

- Assemble the three-level identifiers of SMT\_ID, CORE\_ID, PACKAGE\_IDs into arrays for each enabled logical processor. This is shown in Example 8-22a.
- To detect the number of physical packages: use PACKAGE\_ID to identify those logical processors that reside in the same physical package. This is shown in Example 8-22b. This example also depicts a technique to construct a mask to represent the logical processors that reside in the same package.
- To detect the number of processor cores: use CORE\_ID to identify those logical processors that reside in the same core. This is shown in Example 8-22. This example also depicts a technique to construct a mask to represent the logical processors that reside in the same core.

In Example 8-21, the numerical ID value can be obtained from the value extracted with the mask by shifting it right by shift count. Algorithms below do not shift the value. The assumption is that the SubID values can be compared for equivalence without the need to shift.

#### Example 8-21. Pseudo Code Depicting Three-level Extraction Algorithm

```

For Each local_APIC_ID{
    // Calculate SMT_MASK, the bit mask pattern to extract SMT_ID,
    // SMT_MASK is determined using topology enumeration parameters
    // from CPUID leaf 0BH (Example 8-19);
    // otherwise, SMT_MASK is determined using CPUID leaf 01H and leaf 04H (Example 8-20).
    // This algorithm assumes there is symmetry across core boundary, i.e. each core within a
    // package has the same number of logical processors
    // SMT_ID always starts from bit 0, corresponding to the right-most bit-field
    SMT_ID = APIC_ID & SMT_MASK;

// Extract CORE_ID:
    // CORE_MASK is determined in Example 8-19 or Example 8-20
    CORE_ID = (APIC_ID & CORE_MASK);

    // Extract PACKAGE_ID:
    // Assume single cluster.
    // Shift out the mask width for maximum logical processors per package
    // PACKAGE_MASK is determined in Example 8-19 or Example 8-20
    PACKAGE_ID = (APIC_ID & PACKAGE_MASK);
}

```

**Example 8-22. Compute the Number of Packages, Cores, and Processor Relationships in a MP System**

a) Assemble lists of PACKAGE\_ID, CORE\_ID, and SMT\_ID of each enabled logical processors

```
//The BIOS and/or OS may limit the number of logical processors available to applications
// after system boot. The below algorithm will compute topology for the processors visible
// to the thread that is computing it.

// Extract the 3-levels of IDs on every processor
// SystemAffinity is a bitmask of all the processors started by the OS. Use OS specific APIs to
// obtain it.
// ThreadAffinityMask is used to affinitize the topology enumeration thread to each processor
using OS specific APIs.
// Allocate per processor arrays to store the Package_ID, Core_ID and SMT_ID for every started
// processor.
```

```
ThreadAffinityMask = 1;
ProcessorNum = 0;
while (ThreadAffinityMask != 0 && ThreadAffinityMask <= SystemAffinity) {
    // Check to make sure we can utilize this processor first.
    if (ThreadAffinityMask & SystemAffinity){
        Set thread to run on the processor specified in ThreadAffinityMask
        Wait if necessary and ensure thread is running on specified processor

        APIC_ID = GetAPIC_ID(); // 32 bit ID in Example 8-19 or 8-bit ID in Example 8-20
        Extract the Package_ID, Core_ID and SMT_ID as explained in three level extraction
        algorithm of Example 8-21
        PackageID[ProcessorNUM] = PACKAGE_ID;
        CoreID[ProcessorNum] = CORE_ID;
        SmtID[ProcessorNum] = SMT_ID;
        ProcessorNum++;
    }
    ThreadAffinityMask <<= 1;
}
NumStartedLPs = ProcessorNum;
```

b) Using the list of PACKAGE\_ID to count the number of physical packages in a MP system and construct, for each package, a multi-bit mask corresponding to those logical processors residing in the same package.

```
// Compute the number of packages by counting the number of processors
// with unique PACKAGE_IDs in the PackageID array.
// Compute the mask of processors in each package.
```

PackageIDBucket is an array of unique PACKAGE\_ID values. Allocate an array of NumStartedLPs count of entries in this array.  
 PackageProcessorMask is a corresponding array of the bit mask of processors belonging to the same package, these are processors with the same PACKAGE\_ID  
 The algorithm below assumes there is symmetry across package boundary if more than one socket is populated in an MP system.  
 // Bucket Package IDs and compute processor mask for every package.

```
PackageNum = 1;
PackageIDBucket[0] = PackageID[0];
ProcessorMask = 1;
PackageProcessorMask[0] = ProcessorMask;
```

```

For (ProcessorNum = 1; ProcessorNum < NumStartedLPs; ProcessorNum++) {
    ProcessorMask << = 1;
    For (i=0; i < PackageNum; i++) {
        // we may be comparing bit-fields of logical processors residing in different
        // packages, the code below assume package symmetry
        If (PackageID[ProcessorNum] = PackageIDBucket[i]) {
            PackageProcessorMask[i] |= ProcessorMask;
            Break; // found in existing bucket, skip to next iteration
        }
    }
    if (i = PackageNum) {
        //PACKAGE_ID did not match any bucket, start new bucket
        PackageIDBucket[i] = PackageID[ProcessorNum];
        PackageProcessorMask[i] = ProcessorMask;
        PackageNum++;
    }
}
// PackageNum has the number of Packages started in OS
// PackageProcessorMask[] array has the processor set of each package

```

c) Using the list of CORE\_ID to count the number of cores in a MP system and construct, for each core, a multi-bit mask corresponding to those logical processors residing in the same core.

Processors in the same core can be determined by bucketing the processors with the same PACKAGE\_ID and CORE\_ID. Note that code below can BIT OR the values of PACKAGE and CORE ID because they have not been shifted right.

The algorithm below assumes there is symmetry across package boundary if more than one socket is populated in an MP system.

```

//Bucketing PACKAGE and CORE IDs and computing processor mask for every core
CoreNum = 1;
CoreIDBucket[0] = PackageID[0] | CoreID[0];
ProcessorMask = 1;
CoreProcessorMask[0] = ProcessorMask;
For (ProcessorNum = 1; ProcessorNum < NumStartedLPs; ProcessorNum++) {
    ProcessorMask << = 1;
    For (i=0; i < CoreNum; i++) {
        // we may be comparing bit-fields of logical processors residing in different
        // packages, the code below assume package symmetry
        If ((PackageID[ProcessorNum] | CoreID[ProcessorNum]) = CoreIDBucket[i]) {
            CoreProcessorMask[i] |= ProcessorMask;
            Break; // found in existing bucket, skip to next iteration
        }
    }
    if (i = CoreNum) {
        //Did not match any bucket, start new bucket
        CoreIDBucket[i] = PackageID[ProcessorNum] | CoreID[ProcessorNum];
        CoreProcessorMask[i] = ProcessorMask;
        CoreNum++;
    }
}
// CoreNum has the number of cores started in the OS
// CoreProcessorMask[] array has the processor set of each core

```

Other processor relationships such as processor mask of sibling cores can be computed from set operations of the PackageProcessorMask[] and CoreProcessorMask[].

The algorithm shown above can be adapted to work with earlier generations of single-core IA-32 processors that support Intel Hyper-Threading Technology and in situations that the deterministic cache parameter leaf is not supported (provided CPUID supports initial APIC ID). A reference code example is available (see *Intel® 64 Architecture Processor Topology Enumeration*).

## 8.10 MANAGEMENT OF IDLE AND BLOCKED CONDITIONS

When a logical processor in an MP system (including multi-core processor or processors supporting Intel Hyper-Threading Technology) is idle (no work to do) or blocked (on a lock or semaphore), additional management of the core execution engine resource can be accomplished by using the HLT (halt), PAUSE, or the MONITOR/MWAIT instructions.

### 8.10.1 HLT Instruction

The HLT instruction stops the execution of the logical processor on which it is executed and places it in a halted state until further notice (see the description of the HLT instruction in Chapter 3 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*). When a logical processor is halted, active logical processors continue to have full access to the shared resources within the physical package. Here shared resources that were being used by the halted logical processor become available to active logical processors, allowing them to execute at greater efficiency. When the halted logical processor resumes execution, shared resources are again shared among all active logical processors. (See Section 8.10.6.3, "Halt Idle Logical Processors," for more information about using the HLT instruction with processors supporting Intel Hyper-Threading Technology.)

### 8.10.2 PAUSE Instruction

The PAUSE instruction can improve the performance of processors supporting Intel Hyper-Threading Technology when executing "spin-wait loops" and other routines where one thread is accessing a shared lock or semaphore in a tight polling loop. When executing a spin-wait loop, the processor can suffer a severe performance penalty when exiting the loop because it detects a possible memory order violation and flushes the core processor's pipeline. The PAUSE instruction provides a hint to the processor that the code sequence is a spin-wait loop. The processor uses this hint to avoid the memory order violation and prevent the pipeline flush. In addition, the PAUSE instruction depipelined the spin-wait loop to prevent it from consuming execution resources excessively and consume power needlessly. (See Section 8.10.6.1, "Use the PAUSE Instruction in Spin-Wait Loops," for more information about using the PAUSE instruction with IA-32 processors supporting Intel Hyper-Threading Technology.)

### 8.10.3 Detecting Support MONITOR/MWAIT Instruction

Streaming SIMD Extensions 3 introduced two instructions (MONITOR and MWAIT) to help multithreaded software improve thread synchronization. In the initial implementation, MONITOR and MWAIT are available to software at ring 0. The instructions are conditionally available at levels greater than 0. Use the following steps to detect the availability of MONITOR and MWAIT:

- Use CPUID to query the MONITOR bit (CPUID.1.ECX[3] = 1).
- If CPUID indicates support, execute MONITOR inside a TRY/EXCEPT exception handler and trap for an exception. If an exception occurs, MONITOR and MWAIT are not supported at a privilege level greater than 0. See Example 8-23.

**Example 8-23. Verifying MONITOR/MWAIT Support**

```

boolean MONITOR_MWAIT_works = TRUE;
try {
    _asm {
        xor ecx, ecx
        xor edx, edx
        mov eax, MemArea
        monitor
    }
    // Use monitor
} except (UNWIND) {
    // if we get here, MONITOR/MWAIT is not supported
    MONITOR_MWAIT_works = FALSE;
}

```

**8.10.4 MONITOR/MWAIT Instruction**

Operating systems usually implement idle loops to handle thread synchronization. In a typical idle-loop scenario, there could be several “busy loops” and they would use a set of memory locations. An impacted processor waits in a loop and poll a memory location to determine if there is available work to execute. The posting of work is typically a write to memory (the work-queue of the waiting processor). The time for initiating a work request and getting it scheduled is on the order of a few bus cycles.

From a resource sharing perspective (logical processors sharing execution resources), use of the HLT instruction in an OS idle loop is desirable but has implications. Executing the HLT instruction on a idle logical processor puts the targeted processor in a non-execution state. This requires another processor (when posting work for the halted logical processor) to wake up the halted processor using an inter-processor interrupt. The posting and servicing of such an interrupt introduces a delay in the servicing of new work requests.

In a shared memory configuration, exits from busy loops usually occur because of a state change applicable to a specific memory location; such a change tends to be triggered by writes to the memory location by another agent (typically a processor).

MONITOR/MWAIT complement the use of HLT and PAUSE to allow for efficient partitioning and un-partitioning of shared resources among logical processors sharing physical resources. MONITOR sets up an effective address range that is monitored for write-to-memory activities; MWAIT places the processor in an optimized state (this may vary between different implementations) until a write to the monitored address range occurs.

In the initial implementation of MONITOR and MWAIT, they are available at CPL = 0 only.

Both instructions rely on the state of the processor’s monitor hardware. The monitor hardware can be either armed (by executing the MONITOR instruction) or triggered (due to a variety of events, including a store to the monitored memory region). If upon execution of MWAIT, monitor hardware is in a triggered state: MWAIT behaves as a NOP and execution continues at the next instruction in the execution stream. The state of monitor hardware is not architecturally visible except through the behavior of MWAIT.

Multiple events other than a write to the triggering address range can cause a processor that executed MWAIT to wake up. These include events that would lead to voluntary or involuntary context switches, such as:

- External interrupts, including NMI, SMI, INIT, BINIT, MCERR, A20M#
- Faults, Aborts (including Machine Check)
- Architectural TLB invalidations including writes to CR0, CR3, CR4 and certain MSR writes; execution of LMSW (occurring prior to issuing MWAIT but after setting the monitor)
- Voluntary transitions due to fast system call and far calls (occurring prior to issuing MWAIT but after setting the monitor)

Power management related events (such as Thermal Monitor 2 or chipset driven STPCLK# assertion) will not cause the monitor event pending flag to be cleared. Faults will not cause the monitor event pending flag to be cleared.



Software should not allow for voluntary context switches in between MONITOR/MWAIT in the instruction flow. Note that execution of MWAIT does not re-arm the monitor hardware. This means that MONITOR/MWAIT need to be executed in a loop. Also note that exits from the MWAIT state could be due to a condition other than a write to the triggering address; software should explicitly check the triggering data location to determine if the write occurred. Software should also check the value of the triggering address following the execution of the monitor instruction (and prior to the execution of the MWAIT instruction). This check is to identify any writes to the triggering address that occurred during the course of MONITOR execution.

The address range provided to the MONITOR instruction must be of write-back caching type. Only write-back memory type stores to the monitored address range will trigger the monitor hardware. If the address range is not in memory of write-back type, the address monitor hardware may not be set up properly or the monitor hardware may not be armed. Software is also responsible for ensuring that

- Writes that are not intended to cause the exit of a busy loop do not write to a location within the address region being monitored by the monitor hardware,
- Writes intended to cause the exit of a busy loop are written to locations within the monitored address region.

Not doing so will lead to more false wakeups (an exit from the MWAIT state not due to a write to the intended data location). These have negative performance implications. It might be necessary for software to use padding to prevent false wakeups. CPUID provides a mechanism for determining the size data locations for monitoring as well as a mechanism for determining the size of a the pad.

### 8.10.5 Monitor/Mwait Address Range Determination

To use the MONITOR/MWAIT instructions, software should know the length of the region monitored by the MONITOR/MWAIT instructions and the size of the coherence line size for cache-snoop traffic in a multiprocessor system. This information can be queried using the CPUID monitor leaf function (EAX = 05H). You will need the smallest and largest monitor line size:

- To avoid missed wake-ups: make sure that the data structure used to monitor writes fits within the smallest monitor line-size. Otherwise, the processor may not wake up after a write intended to trigger an exit from MWAIT.
- To avoid false wake-ups; use the largest monitor line size to pad the data structure used to monitor writes. Software must make sure that beyond the data structure, no unrelated data variable exists in the triggering area for MWAIT. A pad may be needed to avoid this situation.

These above two values bear no relationship to cache line size in the system and software should not make any assumptions to that effect. Within a single-cluster system, the two parameters should default to be the same (the size of the monitor triggering area is the same as the system coherence line size).

Based on the monitor line sizes returned by the CPUID, the OS should dynamically allocate structures with appropriate padding. If static data structures must be used by an OS, attempt to adapt the data structure and use a dynamically allocated data buffer for thread synchronization. When the latter technique is not possible, consider not using MONITOR/MWAIT when using static data structures.

To set up the data structure correctly for MONITOR/MWAIT on multi-clustered systems: interaction between processors, chipsets, and the BIOS is required (system coherence line size may depend on the chipset used in the system; the size could be different from the processor's monitor triggering area). The BIOS is responsible to set the correct value for system coherence line size using the IA32\_MONITOR\_FILTER\_LINE\_SIZE MSR. Depending on the relative magnitude of the size of the monitor triggering area versus the value written into the IA32\_MONITOR\_FILTER\_LINE\_SIZE MSR, the smaller of the parameters will be reported as the *Smallest Monitor Line Size*. The larger of the parameters will be reported as the *Largest Monitor Line Size*.

### 8.10.6 Required Operating System Support

This section describes changes that must be made to an operating system to run on processors supporting Intel Hyper-Threading Technology. It also describes optimizations that can help an operating system make more efficient use of the logical processors sharing execution resources. The required changes and suggested optimizations are representative of the types of modifications that appear in Windows\* XP and Linux\* kernel 2.4.0 operating systems for Intel processors supporting Intel Hyper-Threading Technology. Additional optimizations for processors

supporting Intel Hyper-Threading Technology are described in the *Intel® 64 and IA-32 Architectures Optimization Reference Manual*.

### 8.10.6.1 Use the PAUSE Instruction in Spin-Wait Loops

Intel recommends that a PAUSE instruction be placed in all spin-wait loops that run on Intel processors supporting Intel Hyper-Threading Technology and multi-core processors.

Software routines that use spin-wait loops include multiprocessor synchronization primitives (spin-locks, semaphores, and mutex variables) and idle loops. Such routines keep the processor core busy executing a load-compare-branch loop while a thread waits for a resource to become available. Including a PAUSE instruction in such a loop greatly improves efficiency (see Section 8.10.2, “PAUSE Instruction”). The following routine gives an example of a spin-wait loop that uses a PAUSE instruction:

```
Spin_Lock:
    CMP lockvar, 0    ;Check if lock is free
    JE Get_Lock
    PAUSE            ;Short delay
    JMP Spin_Lock
Get_Lock:
    MOV EAX, 1
    XCHG EAX, lockvar ;Try to get lock
    CMP EAX, 0      ;Test if successful
    JNE Spin_Lock
Critical_Section:
    <critical section code>
    MOV lockvar, 0
```

...

Continue:

The spin-wait loop above uses a “test, test-and-set” technique for determining the availability of the synchronization variable. This technique is recommended when writing spin-wait loops.

In IA-32 processor generations earlier than the Pentium 4 processor, the PAUSE instruction is treated as a NOP instruction.

### 8.10.6.2 Potential Usage of MONITOR/MWAIT in C0 Idle Loops

An operating system may implement different handlers for different idle states. A typical OS idle loop on an ACPI-compatible OS is shown in Example 8-24:

#### Example 8-24. A Typical OS Idle Loop

```
// WorkQueue is a memory location indicating there is a thread
// ready to run. A non-zero value for WorkQueue is assumed to
// indicate the presence of work to be scheduled on the processor.
// The idle loop is entered with interrupts disabled.

WHILE (1) {
    IF (WorkQueue) THEN {
        // Schedule work at WorkQueue.
    }
    ELSE {
        // No work to do - wait in appropriate C-state handler depending
        // on Idle time accumulated
        IF (IdleTime >= IdleTimeThreshold) THEN {
            // Call appropriate C1, C2, C3 state handler, C1 handler
```

```

        // shown below
    }
}
// C1 handler uses a Halt instruction
VOID C1Handler()
{ STI
  HLT
}

```

The MONITOR and MWAIT instructions may be considered for use in the C0 idle state loops, if MONITOR and MWAIT are supported.

### Example 8-25. An OS Idle Loop with MONITOR/MWAIT in the C0 Idle Loop

```

// WorkQueue is a memory location indicating there is a thread
// ready to run. A non-zero value for WorkQueue is assumed to
// indicate the presence of work to be scheduled on the processor.
// The following example assumes that the necessary padding has been
// added surrounding WorkQueue to eliminate false wakeups
// The idle loop is entered with interrupts disabled.

```

```

WHILE (1) {
  IF (WorkQueue) THEN {
    // Schedule work at WorkQueue.
  }
  ELSE {
    // No work to do - wait in appropriate C-state handler depending
    // on Idle time accumulated.
    IF (IdleTime >= IdleTimeThreshold) THEN {
      // Call appropriate C1, C2, C3 state handler, C1
      // handler shown below
      MONITOR WorkQueue // Setup of eax with WorkQueue
                        // LinearAddress,
                        // ECX, EDX = 0
      IF (WorkQueue ≠ 0) THEN {
        MWAIT
      }
    }
  }
}
// C1 handler uses a Halt instruction.
VOID C1Handler()
{ STI
  HLT
}

```

### 8.10.6.3 Halt Idle Logical Processors

If one of two logical processors is idle or in a spin-wait loop of long duration, explicitly halt that processor by means of a HLT instruction.

In an MP system, operating systems can place idle processors into a loop that continuously checks the run queue for runnable software tasks. Logical processors that execute idle loops consume a significant amount of core's execution resources that might otherwise be used by the other logical processors in the physical package. For this reason, halting idle logical processors optimizes the performance.<sup>12</sup> If all logical processors within a physical package are halted, the processor will enter a power-saving state.

### 8.10.6.4 Potential Usage of MONITOR/MWAIT in C1 Idle Loops

An operating system may also consider replacing HLT with MONITOR/MWAIT in its C1 idle loop. An example is shown in Example 8-26:

#### Example 8-26. An OS Idle Loop with MONITOR/MWAIT in the C1 Idle Loop

```
// WorkQueue is a memory location indicating there is a thread
// ready to run. A non-zero value for WorkQueue is assumed to
// indicate the presence of work to be scheduled on the processor.
// The following example assumes that the necessary padding has been
// added surrounding WorkQueue to eliminate false wakeups
// The idle loop is entered with interrupts disabled.
```

```
WHILE (1) {
    IF (WorkQueue) THEN {
        // Schedule work at WorkQueue
    }
    ELSE {
        // No work to do - wait in appropriate C-state handler depending
        // on Idle time accumulated
        IF (IdleTime >= IdleTimeThreshold) THEN {
            // Call appropriate C1, C2, C3 state handler, C1
            // handler shown below
        }
    }
}

VOID C1Handler()

{ MONITOR WorkQueue // Setup of eax with WorkQueue LinearAddress,
    // ECX, EDX = 0
  IF (WorkQueue ≠ 0) THEN {
    STI
    MWAIT // EAX, ECX = 0
  }
}
```

### 8.10.6.5 Guidelines for Scheduling Threads on Logical Processors Sharing Execution Resources

Because the logical processors, the order in which threads are dispatched to logical processors for execution can affect the overall efficiency of a system. The following guidelines are recommended for scheduling threads for execution.

- Dispatch threads to one logical processor per processor core before dispatching threads to the other logical processor sharing execution resources in the same processor core.
- In an MP system with two or more physical packages, distribute threads out over all the physical processors, rather than concentrate them in one or two physical processors.
- Use processor affinity to assign a thread to a specific processor core or package, depending on the cache-sharing topology. The practice increases the chance that the processor's caches will contain some of the thread's code and data when it is dispatched for execution after being suspended.

---

12. Excessive transitions into and out of the HALT state could also incur performance penalties. Operating systems should evaluate the performance trade-offs for their operating system.

### 8.10.6.6 Eliminate Execution-Based Timing Loops

Intel discourages the use of timing loops that depend on a processor's execution speed to measure time. There are several reasons:

- Timing loops cause problems when they are calibrated on a IA-32 processor running at one frequency and then executed on a processor running at another frequency.
- Routines for calibrating execution-based timing loops produce unpredictable results when run on an IA-32 processor supporting Intel Hyper-Threading Technology. This is due to the sharing of execution resources between the logical processors within a physical package.

To avoid the problems described, timing loop routines must use a timing mechanism for the loop that does not depend on the execution speed of the logical processors in the system. The following sources are generally available:

- A high resolution system timer (for example, an Intel 8254).
- A high resolution timer within the processor (such as, the local APIC timer or the time-stamp counter).

For additional information, see the *Intel® 64 and IA-32 Architectures Optimization Reference Manual*.

### 8.10.6.7 Place Locks and Semaphores in Aligned, 128-Byte Blocks of Memory

When software uses locks or semaphores to synchronize processes, threads, or other code sections; Intel recommends that only one lock or semaphore be present within a cache line (or 128 byte sector, if 128-byte sector is supported). In processors based on Intel NetBurst microarchitecture (which support 128-byte sector consisting of two cache lines), following this recommendation means that each lock or semaphore should be contained in a 128-byte block of memory that begins on a 128-byte boundary. The practice minimizes the bus traffic required to service locks.

## 8.11 MP INITIALIZATION FOR P6 FAMILY PROCESSORS

This section describes the MP initialization process for systems that use multiple P6 family processors. This process uses the MP initialization protocol that was introduced with the Pentium Pro processor (see Section 8.4, "Multiple-Processor (MP) Initialization"). For P6 family processors, this protocol is typically used to boot 2 or 4 processors that reside on single system bus; however, it can support from 2 to 15 processors in a multi-clustered system when the APIC busses are tied together. Larger systems are not supported.

### 8.11.1 Overview of the MP Initialization Process For P6 Family Processors

During the execution of the MP initialization protocol, one processor is selected as the bootstrap processor (BSP) and the remaining processors are designated as application processors (APs), see Section 8.4.1, "BSP and AP Processors." Thereafter, the BSP manages the initialization of itself and the APs. This initialization includes executing BIOS initialization code and operating-system initialization code.

The MP protocol imposes the following requirements and restrictions on the system:

- An APIC clock (APICLK) must be provided.
- The MP protocol will be executed only after a power-up or RESET. If the MP protocol has been completed and a BSP has been chosen, subsequent INITs (either to a specific processor or system wide) do not cause the MP protocol to be repeated. Instead, each processor examines its BSP flag (in the APIC\_BASE MSR) to determine whether it should execute the BIOS boot-strap code (if it is the BSP) or enter a wait-for-SIPI state (if it is an AP).
- All devices in the system that are capable of delivering interrupts to the processors must be inhibited from doing so for the duration of the MP initialization protocol. The time during which interrupts must be inhibited includes the window between when the BSP issues an INIT-SIPI-SIPI sequence to an AP and when the AP responds to the last SIPI in the sequence.

The following special-purpose interprocessor interrupts (IPIs) are used during the boot phase of the MP initialization protocol. These IPIs are broadcast on the APIC bus.

- **Boot IPI (BIPI)**—Initiates the arbitration mechanism that selects a BSP from the group of processors on the system bus and designates the remainder of the processors as APs. Each processor on the system bus broadcasts a BIPI to all the processors following a power-up or RESET.
- **Final Boot IPI (FIPI)**—Initiates the BIOS initialization procedure for the BSP. This IPI is broadcast to all the processors on the system bus, but only the BSP responds to it. The BSP responds by beginning execution of the BIOS initialization code at the reset vector.
- **Startup IPI (SIPI)**—Initiates the initialization procedure for an AP. The SIPI message contains a vector to the AP initialization code in the BIOS.

Table 8-5 describes the various fields of the boot phase IPIs.

**Table 8-5. Boot Phase IPI Message Format**

Type	Destination Field	Destination Shorthand	Trigger Mode	Level	Destination Mode	Delivery Mode	Vector (Hex)
BIPI	Not used	All including self	Edge	Deassert	Don't Care	Fixed (000)	40 to 4E*
FIPI	Not used	All including self	Edge	Deassert	Don't Care	Fixed (000)	10
SIPI	Used	All excluding self	Edge	Assert	Physical	StartUp (110)	00 to FF

**NOTE:**

\* For all P6 family processors.

For BIPI messages, the lower 4 bits of the vector field contain the APIC ID of the processor issuing the message and the upper 4 bits contain the “generation ID” of the message. All P6 family processor will have a generation ID of 4H. BIPIs will therefore use vector values ranging from 40H to 4EH (4FH can not be used because FH is not a valid APIC ID).

### 8.11.2 MP Initialization Protocol Algorithm

Following a power-up or RESET of a system, the P6 family processors in the system execute the MP initialization protocol algorithm to initialize each of the processors on the system bus. In the course of executing this algorithm, the following boot-up and initialization operations are carried out:

1. Each processor on the system bus is assigned a unique APIC ID, based on system topology (see Section 8.4.5, “Identifying Logical Processors in an MP System”). This ID is written into the local APIC ID register for each processor.
2. Each processor executes its internal BIST simultaneously with the other processors on the system bus. Upon completion of the BIST (at T0), each processor broadcasts a BIPI to “all including self” (see Figure 8-1).
3. APIC arbitration hardware causes all the APICs to respond to the BIPIs one at a time (at T1, T2, T3, and T4).
4. When the first BIPI is received (at time T1), each APIC compares the four least significant bits of the BIPI’s vector field with its APIC ID. If the vector and APIC ID match, the processor selects itself as the BSP by setting the BSP flag in its IA32\_APIC\_BASE MSR. If the vector and APIC ID do not match, the processor selects itself as an AP by entering the “wait for SIPI” state. (Note that in Figure 8-1, the BIPI from processor 1 is the first BIPI to be handled, so processor 1 becomes the BSP.)
5. The newly established BSP broadcasts an FIPI message to “all including self.” The FIPI is guaranteed to be handled only after the completion of the BIPIs that were issued by the non-BSP processors.

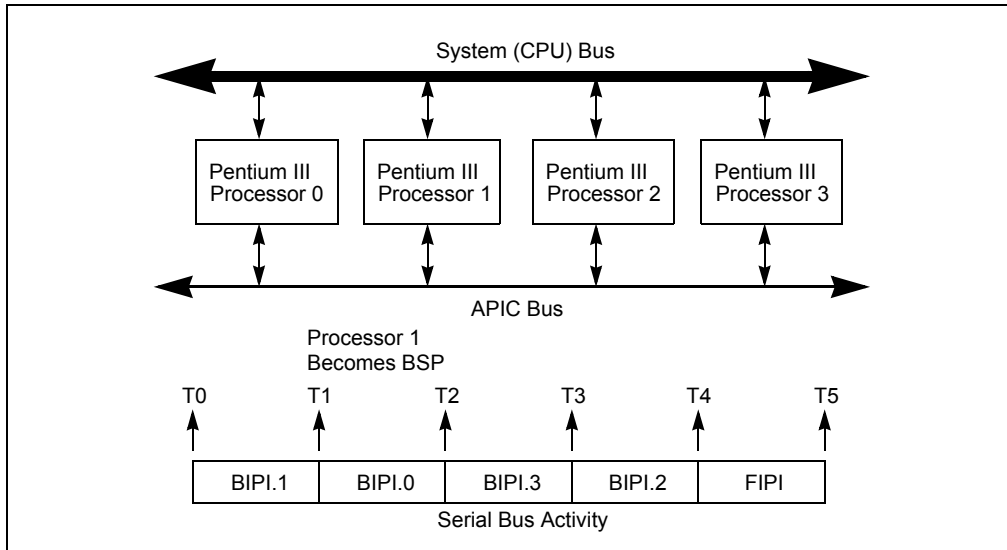


Figure 8-1. MP System With Multiple Pentium III Processors

6. After the BSP has been established, the outstanding BIPIs are received one at a time (at T2, T3, and T4) and ignored by all processors.
7. When the FIPI is finally received (at T5), only the BSP responds to it. It responds by fetching and executing BIOS boot-strap code, beginning at the reset vector (physical address FFFF FFF0H).
8. As part of the boot-strap code, the BSP creates an ACPI table and an MP table and adds its initial APIC ID to these tables as appropriate.
9. At the end of the boot-strap procedure, the BSP broadcasts a SIPI message to all the APs in the system. Here, the SIPI message contains a vector to the BIOS AP initialization code (at 000V V000H, where VV is the vector contained in the SIPI message).
10. All APs respond to the SIPI message by racing to a BIOS initialization semaphore. The first one to the semaphore begins executing the initialization code. (See MP init code for semaphore implementation details.) As part of the AP initialization procedure, the AP adds its APIC ID number to the ACPI and MP tables as appropriate. At the completion of the initialization procedure, the AP executes a CLI instruction (to clear the IF flag in the EFLAGS register) and halts itself.
11. When each of the APs has gained access to the semaphore and executed the AP initialization code and all written their APIC IDs into the appropriate places in the ACPI and MP tables, the BSP establishes a count for the number of processors connected to the system bus, completes executing the BIOS boot-strap code, and then begins executing operating-system boot-strap and start-up code.
12. While the BSP is executing operating-system boot-strap and start-up code, the APs remain in the halted state. In this state they will respond only to INITs, NMIs, and SMIs. They will also respond to snoops and to assertions of the STPCLK# pin.

See Section 8.4.4, "MP Initialization Example," for an annotated example the use of the MP protocol to boot IA-32 processors in an MP. This code should run on any IA-32 processor that used the MP protocol.

### 8.11.2.1 Error Detection and Handling During the MP Initialization Protocol

Errors may occur on the APIC bus during the MP initialization phase. These errors may be transient or permanent and can be caused by a variety of failure mechanisms (for example, broken traces, soft errors during bus usage, etc.). All serial bus related errors will result in an APIC checksum or acceptance error.

The MP initialization protocol makes the following assumptions regarding errors that occur during initialization:

- If errors are detected on the APIC bus during execution of the MP initialization protocol, the processors that detect the errors are shut down.

- The MP initialization protocol will be executed by processors even if they fail their BIST sequences.





#### 14. Updates to Chapter 9, Volume 3A

Change bars show changes to Chapter 9 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A: System Programming Guide, Part 1*.

-----  
Changes include initialization updates, including revised processor state and reset table, and new table on variance of RESET values.

This chapter describes the facilities provided for managing processor wide functions and for initializing the processor. The subjects covered include: processor initialization, x87 FPU initialization, processor configuration, feature determination, mode switching, the MSRs (in the Pentium, P6 family, Pentium 4, and Intel Xeon processors), and the MTRRs (in the P6 family, Pentium 4, and Intel Xeon processors).

### 9.1 INITIALIZATION OVERVIEW

Following power-up or an assertion of the RESET# pin, each processor on the system bus performs a hardware initialization of the processor (known as a hardware reset) and an optional built-in self-test (BIST). A hardware reset sets each processor's registers to a known state and places the processor in real-address mode. It also invalidates the internal caches, translation lookaside buffers (TLBs) and the branch target buffer (BTB). At this point, the action taken depends on the processor family:

- **Pentium 4 processors (CPUID DisplayFamily 0FH)** — All the processors on the system bus (including a single processor in a uniprocessor system) execute the multiple processor (MP) initialization protocol. The processor that is selected through this protocol as the bootstrap processor (BSP) then immediately starts executing software-initialization code in the current code segment beginning at the offset in the EIP register. The application (non-BSP) processors (APs) go into a Wait For Startup IPI (SIPI) state while the BSP is executing initialization code. See Section 8.4, "Multiple-Processor (MP) Initialization," for more details. Note that in a uniprocessor system, the single Pentium 4 or Intel Xeon processor automatically becomes the BSP.
- **IA-32 and Intel 64 processors (CPUID DisplayFamily 06H)** — The action taken is the same as for the Pentium 4 processors (as described in the previous paragraph).
- **Pentium processors** — In either a single- or dual- processor system, a single Pentium processor is always pre-designated as the primary processor. Following a reset, the primary processor behaves as follows in both single- and dual-processor systems. Using the dual-processor (DP) ready initialization protocol, the primary processor immediately starts executing software-initialization code in the current code segment beginning at the offset in the EIP register. The secondary processor (if there is one) goes into a halt state.
- **Intel486 processor** — The primary processor (or single processor in a uniprocessor system) immediately starts executing software-initialization code in the current code segment beginning at the offset in the EIP register. (The Intel486 does not automatically execute a DP or MP initialization protocol to determine which processor is the primary processor.)

The software-initialization code performs all system-specific initialization of the BSP or primary processor and the system logic.

At this point, for MP (or DP) systems, the BSP (or primary) processor wakes up each AP (or secondary) processor to enable those processors to execute self-configuration code.

When all processors are initialized, configured, and synchronized, the BSP or primary processor begins executing an initial operating-system or executive task.

The x87 FPU is also initialized to a known state during hardware reset. x87 FPU software initialization code can then be executed to perform operations such as setting the precision of the x87 FPU and the exception masks. No special initialization of the x87 FPU is required to switch operating modes.

Asserting the INIT# pin on the processor invokes a similar response to a hardware reset. The major difference is that during an INIT, the internal caches, MSRs, MTRRs, and x87 FPU state are left unchanged (although, the TLBs and BTB are invalidated as with a hardware reset). An INIT provides a method for switching from protected to real-address mode while maintaining the contents of the internal caches.

### 9.1.1 Processor State After Reset

Following power-up, The state of control register CR0 is 60000010H (see Figure 9-1). This places the processor in real-address mode with paging disabled.

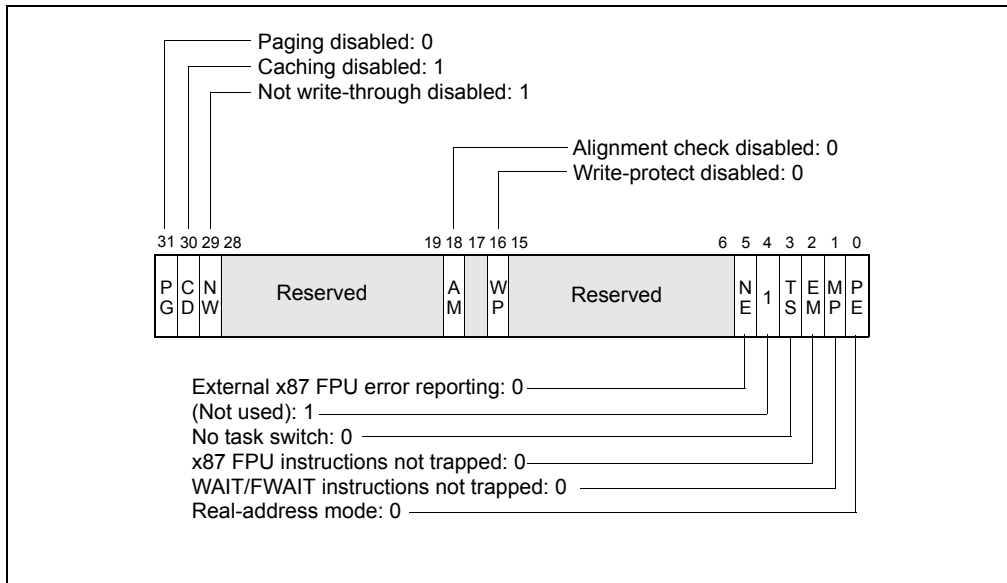


Figure 9-1. Contents of CR0 Register after Reset

The state of the flags and other registers following power-up for the Pentium 4, Pentium Pro, and Pentium processors are shown in Section 22.39, "Initial State of Pentium, Pentium Pro and Pentium 4 Processors" of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*.

Table 9-1 shows processor states of IA-32 and Intel 64 processors with CPUID DisplayFamily signature of 06H at the following events: power-up, Rest, and INIT. In a few cases, the behavior of some registers behave slightly different across warm RESET, the variant cases are marked in Table 9-1 and described in more detail in Table 9-2.

Table 9-1. IA-32 and Intel 64 Processor States Following Power-up, Reset, or INIT

Register	Power up	Reset	INIT
EFLAGS <sup>1</sup>	00000002H	00000002H	00000002H
EIP	0000FFF0H	0000FFF0H	0000FFF0H
CR0	60000010H <sup>2</sup>	60000010H <sup>2</sup>	60000010H <sup>2</sup>
CR2, CR3, CR4	00000000H	00000000H	00000000H
CS	Selector = F000H Base = FFFF0000H Limit = FFFFH AR = Present, R/W, Accessed	Selector = F000H Base = FFFF0000H Limit = FFFFH AR = Present, R/W, Accessed	Selector = F000H Base = FFFF0000H Limit = FFFFH AR = Present, R/W, Accessed
SS, DS, ES, FS, GS	Selector = 0000H Base = 00000000H Limit = FFFFH AR = Present, R/W, Accessed	Selector = 0000H Base = 00000000H Limit = FFFFH AR = Present, R/W, Accessed	Selector = 0000H Base = 00000000H Limit = FFFFH AR = Present, R/W, Accessed
EDX	000n06xxH <sup>3</sup>	000n06xxH <sup>3</sup>	000n06xxH <sup>3</sup>
EAX	0 <sup>4</sup>	0 <sup>4</sup>	0 <sup>4</sup>
EBX, ECX, ESI, EDI, EBP, ESP	00000000H	00000000H	00000000H
ST0 through ST7 <sup>5</sup>	+0.0	+0.0	FINIT/FNINIT: Unchanged

Table 9-1. IA-32 and Intel 64 Processor States Following Power-up, Reset, or INIT (Contd.)

Register	Power up	Reset	INIT
x87 FPU Control Word <sup>5</sup>	0040H	0040H	FINIT/FNINIT: 037FH
x87 FPU Status Word <sup>5</sup>	0000H	0000H	FINIT/FNINIT: 0000H
x87 FPU Tag Word <sup>5</sup>	5555H	5555H	FINIT/FNINIT: FFFFH
x87 FPU Data Operand and CS Seg. Selectors <sup>5</sup>	0000H	0000H	FINIT/FNINIT: 0000H
x87 FPU Data Operand and Inst. Pointers <sup>5</sup>	00000000H	00000000H	FINIT/FNINIT: 00000000H
MM0 through MM7 <sup>5</sup>	0000000000000000H	0000000000000000H	INIT or FINIT/FNINIT: Unchanged
XMM0 through XMM7	0H	0H	Unchanged
MXCSR	1F80H	1F80H	Unchanged
GDTR, IDTR	Base = 00000000H Limit = FFFFH AR = Present, R/W	Base = 00000000H Limit = FFFFH AR = Present, R/W	Base = 00000000H Limit = FFFFH AR = Present, R/W
LDTR, Task Register	Selector = 0000H Base = 00000000H Limit = FFFFH AR = Present, R/W	Selector = 0000H Base = 00000000H Limit = FFFFH AR = Present, R/W	Selector = 0000H Base = 00000000H Limit = FFFFH AR = Present, R/W
DR0, DR1, DR2, DR3	00000000H	00000000H	00000000H
DR6	FFFF0FF0H	FFFF0FF0H	FFFF0FF0H
DR7	00000400H	00000400H	00000400H
R8-R15	0000000000000000H	0000000000000000H	0000000000000000H
XMM8-XMM15	0H	0H	Unchanged
XCRO	3H	3H	Unchanged
IA32_XSS	0H	0H	0H
YMM_H[255:128]	0H	0H	Unchanged
BNDCFGU	0H	0H	0H
BND0-BND3	0H	0H	0H
IA32_BNDCFGS	0H	0H	0H
OPMASK	0H	0H	Unchanged
ZMM_H[511:256]	0H	0H	Unchanged
ZMMHi16[511:0]	0H	0H	Unchanged
PKRU	0H	0H	Unchanged
Intel Processor Trace MSRs	0H	0H <sup>w</sup>	Unchanged
Time-Stamp Counter	0H	0H <sup>w</sup>	Unchanged
IA32_TSC_AUX	0H	0H	Unchanged
IA32_TSC_ADJUST	0H	0H	Unchanged
IA32_TSC_DEADLINE	0H	0H	Unchanged
IA32_SYSENTER_CS/ESP/EIP	0H	0H	Unchanged
IA32_EFER	0000000000000000H	0000000000000000H	0000000000000000H
IA32_STAR/LSTAR	0H	0H	Unchanged
IA32_FS_BASE/GS_BASE	0H	0H	0H

**Table 9-1. IA-32 and Intel 64 Processor States Following Power-up, Reset, or INIT (Contd.)**

Register	Power up	Reset	INIT
IA32_PMCx, IA32_PERFEVTSELx	0H	0H	Unchanged
IA32_FIXED_CTRx, IA32_FIXED_CTR_CTRL, Global Perf Counter Controls	0H	0H	Unchanged
Data and Code Cache, TLBs	Invalid <sup>6</sup>	Invalid <sup>6</sup>	Unchanged
Fixed MTRRs	Disabled	Disabled	Unchanged
Variable MTRRs	Disabled	Disabled	Unchanged
Machine-Check Banks	Undefined	Undefined <sup>w</sup>	Unchanged
Last Branch Record Stack	0	0 <sup>w</sup>	Unchanged
APIC	Enabled	Enabled	Unchanged
X2APIC	Disabled	Disabled	Unchanged
MSR_FEATURE_CONFIG	0	0 <sup>w</sup>	Unchanged
IA32_DEBUG_INTERFACE	0	0 <sup>w</sup>	Unchanged

**NOTES:**

1. The 10 most-significant bits of the EFLAGS register are undefined following a reset. Software should not depend on the states of any of these bits.
  2. The CD and NW flags are unchanged, bit 4 is set to 1, all other bits are cleared.
  3. Where “n” is the Extended Model Value for the respective processor, and “xx” = don’t care.
  4. If Built-In Self-Test (BIST) is invoked on power up or reset, EAX is 0 only if all tests passed. (BIST cannot be invoked during an INIT.)
  5. The state of the x87 FPU and MMX registers is not changed by the execution of an INIT.
  6. Internal caches are invalid after power-up and RESET, but left unchanged with an INIT.
- w: Warm RESET behavior differs from power-on RESET with details listed in Table 9-2.

**Table 9-2. Variance of RESET Values in Selected Intel Architecture Processors**

State	XREF	Value	Feature Flag or DisplayFamily_DisplayModel Signatures
Time-Stamp Counter	Warm RESET	Unmodified across warm Reset	06_2DH, 06_3EH
Machine-Check Banks	Warm RESET	IA32_MCi_Status banks are unmodified across warm Reset	06_2DH, 06_3EH, 06_3FH, 06_4FH, 06_56H
Last Branch Record Stack	Warm RESET	LBR stack MSRs are unmodified across warm Reset	06_1AH, 06_1CH, DisplayFamiy= 06 and DisplayModel > 1DH
MSR_FEATURE_CONFIG	Warm RESET	Unmodified across warm Reset	06_2AH, 06_2CH, 06_2DH, 06_2FH, 06_3AH, DisplayFamiy= 06 and DisplayModel > 37H
Intel Processor Trace MSRs	Warm RESET	Clears IA32_RTIT_CTL.TraceEn, the rest of MSRs are unmodified	If CPUID.(EAX=14H, ECX=0H):EBX[bit 2] = 1
IA32_DEBUG_INTERFACE	Warm RESET	Unmodified across warm Reset	If CPUID.01H:ECX.[11] = 1

## 9.1.2 Processor Built-In Self-Test (BIST)

Hardware may request that the BIST be performed at power-up. The EAX register is cleared (0H) if the processor passes the BIST. A nonzero value in the EAX register after the BIST indicates that a processor fault was detected. If the BIST is not requested, the contents of the EAX register after a hardware reset is 0H.

The overhead for performing a BIST varies between processor families. For example, the BIST takes approximately 30 million processor clock periods to execute on the Pentium 4 processor. This clock count is model-specific; Intel reserves the right to change the number of periods for any Intel 64 or IA-32 processor, without notification.

## 9.1.3 Model and Stepping Information

Following a hardware reset, the EDX register contains component identification and revision information (see Figure 9-2). For example, the model, family, and processor type returned for the first processor in the Intel Pentium 4 family is as follows: model (0000B), family (1111B), and processor type (00B).

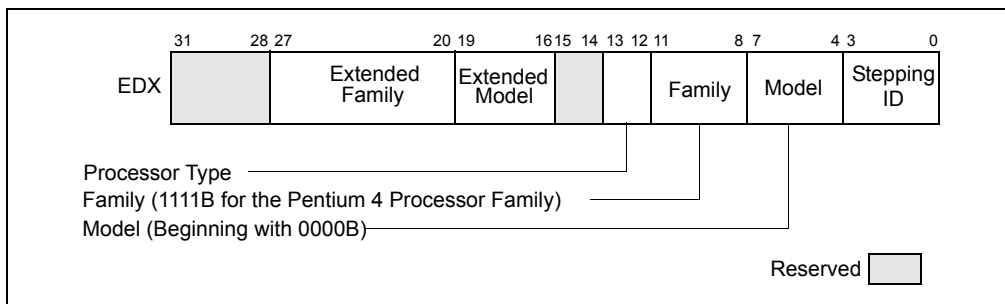


Figure 9-2. Version Information in the EDX Register after Reset

The stepping ID field contains a unique identifier for the processor's stepping ID or revision level. The extended family and extended model fields were added to the IA-32 architecture in the Pentium 4 processors.

## 9.1.4 First Instruction Executed

The first instruction that is fetched and executed following a hardware reset is located at physical address FFFFFFF0H. This address is 16 bytes below the processor's uppermost physical address. The EPROM containing the software-initialization code must be located at this address.

The address FFFFFFF0H is beyond the 1-MByte addressable range of the processor while in real-address mode. The processor is initialized to this starting address as follows. The CS register has two parts: the visible segment selector part and the hidden base address part. In real-address mode, the base address is normally formed by shifting the 16-bit segment selector value 4 bits to the left to produce a 20-bit base address. However, during a hardware reset, the segment selector in the CS register is loaded with F000H and the base address is loaded with FFFF0000H. The starting address is thus formed by adding the base address to the value in the EIP register (that is, FFFF0000 + FFF0H = FFFFFFF0H).

The first time the CS register is loaded with a new value after a hardware reset, the processor will follow the normal rule for address translation in real-address mode (that is, [CS base address = CS segment selector \* 16]). To insure that the base address in the CS register remains unchanged until the EPROM based software-initialization code is completed, the code must not contain a far jump or far call or allow an interrupt to occur (which would cause the CS selector value to be changed).

## 9.2 X87 FPU INITIALIZATION

Software-initialization code can determine the whether the processor contains an x87 FPU by using the CPUID instruction. The code must then initialize the x87 FPU and set flags in control register CR0 to reflect the state of the x87 FPU environment.

A hardware reset places the x87 FPU in the state shown in Table 9-1. This state is different from the state the x87 FPU is placed in following the execution of an FINIT or FNINIT instruction (also shown in Table 9-1). If the x87 FPU is to be used, the software-initialization code should execute an FINIT/FNINIT instruction following a hardware reset. These instructions, tag all data registers as empty, clear all the exception masks, set the TOP-of-stack value to 0, and select the default rounding and precision controls setting (round to nearest and 64-bit precision).

If the processor is reset by asserting the INIT# pin, the x87 FPU state is not changed.

### 9.2.1 Configuring the x87 FPU Environment

Initialization code must load the appropriate values into the MP, EM, and NE flags of control register CR0. These bits are cleared on hardware reset of the processor. Figure 9-3 shows the suggested settings for these flags, depending on the IA-32 processor being initialized. Initialization code can test for the type of processor present before setting or clearing these flags.

**Table 9-3. Recommended Settings of EM and MP Flags on IA-32 Processors**

EM	MP	NE	IA-32 processor
1	0	1	Intel486™ SX, Intel386™ DX, and Intel386™ SX processors only, without the presence of a math coprocessor.
0	1	1 or 0*	Pentium 4, Intel Xeon, P6 family, Pentium, Intel486™ DX, and Intel 487 SX processors, and Intel386 DX and Intel386 SX processors when a companion math coprocessor is present.
0	1	1 or 0*	More recent Intel 64 or IA-32 processors

**NOTE:**

\* The setting of the NE flag depends on the operating system being used.

The EM flag determines whether floating-point instructions are executed by the x87 FPU (EM is cleared) or a device-not-available exception (#NM) is generated for all floating-point instructions so that an exception handler can emulate the floating-point operation (EM = 1). Ordinarily, the EM flag is cleared when an x87 FPU or math coprocessor is present and set if they are not present. If the EM flag is set and no x87 FPU, math coprocessor, or floating-point emulator is present, the processor will hang when a floating-point instruction is executed.

The MP flag determines whether WAIT/FWAIT instructions react to the setting of the TS flag. If the MP flag is clear, WAIT/FWAIT instructions ignore the setting of the TS flag; if the MP flag is set, they will generate a device-not-available exception (#NM) if the TS flag is set. Generally, the MP flag should be set for processors with an integrated x87 FPU and clear for processors without an integrated x87 FPU and without a math coprocessor present. However, an operating system can choose to save the floating-point context at every context switch, in which case there would be no need to set the MP bit.

Table 2-2 shows the actions taken for floating-point and WAIT/FWAIT instructions based on the settings of the EM, MP, and TS flags.

The NE flag determines whether unmasked floating-point exceptions are handled by generating a floating-point error exception internally (NE is set, native mode) or through an external interrupt (NE is cleared). In systems where an external interrupt controller is used to invoke numeric exception handlers (such as MS-DOS-based systems), the NE bit should be cleared.

### 9.2.2 Setting the Processor for x87 FPU Software Emulation

Setting the EM flag causes the processor to generate a device-not-available exception (#NM) and trap to a software exception handler whenever it encounters a floating-point instruction. (Table 9-3 shows when it is appropriate to use this flag.) Setting this flag has two functions:

- It allows x87 FPU code to run on an IA-32 processor that has neither an integrated x87 FPU nor is connected to an external math coprocessor, by using a floating-point emulator.
- It allows floating-point code to be executed using a special or nonstandard floating-point emulator, selected for a particular application, regardless of whether an x87 FPU or math coprocessor is present.



To emulate floating-point instructions, the EM, MP, and NE flag in control register CR0 should be set as shown in Table 9-4.

**Table 9-4. Software Emulation Settings of EM, MP, and NE Flags**

CRO Bit	Value
EM	1
MP	0
NE	1

Regardless of the value of the EM bit, the Intel486 SX processor generates a device-not-available exception (#NM) upon encountering any floating-point instruction.

## 9.3 CACHE ENABLING

IA-32 processors (beginning with the Intel486 processor) and Intel 64 processors contain internal instruction and data caches. These caches are enabled by clearing the CD and NW flags in control register CR0. (They are set during a hardware reset.) Because all internal cache lines are invalid following reset initialization, it is not necessary to invalidate the cache before enabling caching. Any external caches may require initialization and invalidation using a system-specific initialization and invalidation code sequence.

Depending on the hardware and operating system or executive requirements, additional configuration of the processor's caching facilities will probably be required. Beginning with the Intel486 processor, page-level caching can be controlled with the PCD and PWT flags in page-directory and page-table entries. Beginning with the P6 family processors, the memory type range registers (MTRRs) control the caching characteristics of the regions of physical memory. (For the Intel486 and Pentium processors, external hardware can be used to control the caching characteristics of regions of physical memory.) See Chapter 11, "Memory Cache Control," for detailed information on configuration of the caching facilities in the Pentium 4, Intel Xeon, and P6 family processors and system memory.

## 9.4 MODEL-SPECIFIC REGISTERS (MSRS)

Most IA-32 processors (starting from Pentium processors) and Intel 64 processors contain a model-specific registers (MSRs). A given MSR may not be supported across all families and models for Intel 64 and IA-32 processors. Some MSRs are designated as architectural to simplify software programming; a feature introduced by an architectural MSR is expected to be supported in future processors. Non-architectural MSRs are not guaranteed to be supported or to have the same functions on future processors.

MSRs that provide control for a number of hardware and software-related features, include:

- Performance-monitoring counters (see Chapter 23, "Introduction to Virtual Machine Extensions").
- Debug extensions (see Chapter 23, "Introduction to Virtual Machine Extensions").
- Machine-check exception capability and its accompanying machine-check architecture (see Chapter 15, "Machine-Check Architecture").
- MTRRs (see Section 11.11, "Memory Type Range Registers (MTRRs)").
- Thermal and power management.
- Instruction-specific support (for example: SYSENTER, SYSEXIT, SWAPGS, etc.).
- Processor feature/mode support (for example: IA32\_EFER, IA32\_FEATURE\_CONTROL).

The MSRs can be read and written to using the RDMSR and WRMSR instructions, respectively.

When performing software initialization of an IA-32 or Intel 64 processor, many of the MSRs will need to be initialized to set up things like performance-monitoring events, run-time machine checks, and memory types for physical memory.

Lists of available performance-monitoring events are given in Chapter 19, “Performance Monitoring Events”, and lists of available MSRs are given in Chapter 35, “Model-Specific Registers (MSRs)”. The references earlier in this section show where the functions of the various groups of MSRs are described in this manual.

## 9.5 MEMORY TYPE RANGE REGISTERS (MTRRS)

Memory type range registers (MTRRs) were introduced into the IA-32 architecture with the Pentium Pro processor. They allow the type of caching (or no caching) to be specified in system memory for selected physical address ranges. They allow memory accesses to be optimized for various types of memory such as RAM, ROM, frame buffer memory, and memory-mapped I/O devices.

In general, initializing the MTRRs is normally handled by the software initialization code or BIOS and is not an operating system or executive function. At the very least, all the MTRRs must be cleared to 0, which selects the uncached (UC) memory type. See Section 11.11, “Memory Type Range Registers (MTRRs),” for detailed information on the MTRRs.

## 9.6 INITIALIZING SSE/SSE2/SSE3/SSSE3 EXTENSIONS

For processors that contain SSE/SSE2/SSE3/SSSE3 extensions, steps must be taken when initializing the processor to allow execution of these instructions.

1. Check the CPUID feature flags for the presence of the SSE/SSE2/SSE3/SSSE3 extensions (respectively: EDX bits 25 and 26, ECX bit 0 and 9) and support for the FXSAVE and FXRSTOR instructions (EDX bit 24). Also check for support for the CLFLUSH instruction (EDX bit 19). The CPUID feature flags are loaded in the EDX and ECX registers when the CPUID instruction is executed with a 1 in the EAX register.
2. Set the OSFXSR flag (bit 9 in control register CR4) to indicate that the operating system supports saving and restoring the SSE/SSE2/SSE3/SSSE3 execution environment (XMM and MXCSR registers) with the FXSAVE and FXRSTOR instructions, respectively. See Section 2.5, “Control Registers,” for a description of the OSFXSR flag.
3. Set the OSXMMEXCPT flag (bit 10 in control register CR4) to indicate that the operating system supports the handling of SSE/SSE2/SSE3 SIMD floating-point exceptions (#XM). See Section 2.5, “Control Registers,” for a description of the OSXMMEXCPT flag.
4. Set the mask bits and flags in the MXCSR register according to the mode of operation desired for SSE/SSE2/SSE3 SIMD floating-point instructions. See “MXCSR Control and Status Register” in Chapter 10, “Programming with Streaming SIMD Extensions (SSE),” of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for a detailed description of the bits and flags in the MXCSR register.

## 9.7 SOFTWARE INITIALIZATION FOR REAL-ADDRESS MODE OPERATION

Following a hardware reset (either through a power-up or the assertion of the RESET# pin) the processor is placed in real-address mode and begins executing software initialization code from physical address FFFFFFF0H. Software initialization code must first set up the necessary data structures for handling basic system functions, such as a real-mode IDT for handling interrupts and exceptions. If the processor is to remain in real-address mode, software must then load additional operating-system or executive code modules and data structures to allow reliable execution of application programs in real-address mode.

If the processor is going to operate in protected mode, software must load the necessary data structures to operate in protected mode and then switch to protected mode. The protected-mode data structures that must be loaded are described in Section 9.8, “Software Initialization for Protected-Mode Operation.”

### 9.7.1 Real-Address Mode IDT

In real-address mode, the only system data structure that must be loaded into memory is the IDT (also called the “interrupt vector table”). By default, the address of the base of the IDT is physical address 0H. This address can be

changed by using the LIDT instruction to change the base address value in the IDTR. Software initialization code needs to load interrupt- and exception-handler pointers into the IDT before interrupts can be enabled.

The actual interrupt- and exception-handler code can be contained either in EPROM or RAM; however, the code must be located within the 1-MByte addressable range of the processor in real-address mode. If the handler code is to be stored in RAM, it must be loaded along with the IDT.

## 9.7.2 NMI Interrupt Handling

The NMI interrupt is always enabled (except when multiple NMIs are nested). If the IDT and the NMI interrupt handler need to be loaded into RAM, there will be a period of time following hardware reset when an NMI interrupt cannot be handled. During this time, hardware must provide a mechanism to prevent an NMI interrupt from halting code execution until the IDT and the necessary NMI handler software is loaded. Here are two examples of how NMIs can be handled during the initial states of processor initialization:

- A simple IDT and NMI interrupt handler can be provided in EPROM. This allows an NMI interrupt to be handled immediately after reset initialization.
- The system hardware can provide a mechanism to enable and disable NMIs by passing the NMI# signal through an AND gate controlled by a flag in an I/O port. Hardware can clear the flag when the processor is reset, and software can set the flag when it is ready to handle NMI interrupts.

## 9.8 SOFTWARE INITIALIZATION FOR PROTECTED-MODE OPERATION

The processor is placed in real-address mode following a hardware reset. At this point in the initialization process, some basic data structures and code modules must be loaded into physical memory to support further initialization of the processor, as described in Section 9.7, "Software Initialization for Real-Address Mode Operation." Before the processor can be switched to protected mode, the software initialization code must load a minimum number of protected mode data structures and code modules into memory to support reliable operation of the processor in protected mode. These data structures include the following:

- A IDT.
- A GDT.
- A TSS.
- (Optional) An LDT.
- If paging is to be used, at least one page directory and one page table.
- A code segment that contains the code to be executed when the processor switches to protected mode.
- One or more code modules that contain the necessary interrupt and exception handlers.

Software initialization code must also initialize the following system registers before the processor can be switched to protected mode:

- The GDTR.
- (Optional.) The IDTR. This register can also be initialized immediately after switching to protected mode, prior to enabling interrupts.
- Control registers CR1 through CR4.
- (Pentium 4, Intel Xeon, and P6 family processors only.) The memory type range registers (MTRRs).

With these data structures, code modules, and system registers initialized, the processor can be switched to protected mode by loading control register CR0 with a value that sets the PE flag (bit 0).

### 9.8.1 Protected-Mode System Data Structures

The contents of the protected-mode system data structures loaded into memory during software initialization, depend largely on the type of memory management the protected-mode operating-system or executive is going to support: flat, flat with paging, segmented, or segmented with paging.

To implement a flat memory model without paging, software initialization code must at a minimum load a GDT with one code and one data-segment descriptor. A null descriptor in the first GDT entry is also required. The stack can be placed in a normal read/write data segment, so no dedicated descriptor for the stack is required. A flat memory model with paging also requires a page directory and at least one page table (unless all pages are 4 MBytes in which case only a page directory is required). See Section 9.8.3, "Initializing Paging."

Before the GDT can be used, the base address and limit for the GDT must be loaded into the GDTR register using an LGDT instruction.

A multi-segmented model may require additional segments for the operating system, as well as segments and LDTs for each application program. LDTs require segment descriptors in the GDT. Some operating systems allocate new segments and LDTs as they are needed. This provides maximum flexibility for handling a dynamic programming environment. However, many operating systems use a single LDT for all tasks, allocating GDT entries in advance. An embedded system, such as a process controller, might pre-allocate a fixed number of segments and LDTs for a fixed number of application programs. This would be a simple and efficient way to structure the software environment of a real-time system.

### 9.8.2 Initializing Protected-Mode Exceptions and Interrupts

Software initialization code must at a minimum load a protected-mode IDT with gate descriptor for each exception vector that the processor can generate. If interrupt or trap gates are used, the gate descriptors can all point to the same code segment, which contains the necessary exception handlers. If task gates are used, one TSS and accompanying code, data, and task segments are required for each exception handler called with a task gate.

If hardware allows interrupts to be generated, gate descriptors must be provided in the IDT for one or more interrupt handlers.

Before the IDT can be used, the base address and limit for the IDT must be loaded into the IDTR register using an LIDT instruction. This operation is typically carried out immediately after switching to protected mode.

### 9.8.3 Initializing Paging

Paging is controlled by the PG flag in control register CR0. When this flag is clear (its state following a hardware reset), the paging mechanism is turned off; when it is set, paging is enabled. Before setting the PG flag, the following data structures and registers must be initialized:

- Software must load at least one page directory and one page table into physical memory. The page table can be eliminated if the page directory contains a directory entry pointing to itself (here, the page directory and page table reside in the same page), or if only 4-MByte pages are used.
- Control register CR3 (also called the PDBR register) is loaded with the physical base address of the page directory.
- (Optional) Software may provide one set of code and data descriptors in the GDT or in an LDT for supervisor mode and another set for user mode.

With this paging initialization complete, paging is enabled and the processor is switched to protected mode at the same time by loading control register CR0 with an image in which the PG and PE flags are set. (Paging cannot be enabled before the processor is switched to protected mode.)

### 9.8.4 Initializing Multitasking

If the multitasking mechanism is not going to be used and changes between privilege levels are not allowed, it is not necessary to load a TSS into memory or to initialize the task register.

If the multitasking mechanism is going to be used and/or changes between privilege levels are allowed, software initialization code must load at least one TSS and an accompanying TSS descriptor. (A TSS is required to change privilege levels because pointers to the privileged-level 0, 1, and 2 stack segments and the stack pointers for these stacks are obtained from the TSS.) TSS descriptors must not be marked as busy when they are created; they should be marked busy by the processor only as a side-effect of performing a task switch. As with descriptors for LDTs, TSS descriptors reside in the GDT.

After the processor has switched to protected mode, the LTR instruction can be used to load a segment selector for a TSS descriptor into the task register. This instruction marks the TSS descriptor as busy, but does not perform a task switch. The processor can, however, use the TSS to locate pointers to privilege-level 0, 1, and 2 stacks. The segment selector for the TSS must be loaded before software performs its first task switch in protected mode, because a task switch copies the current task state into the TSS.

After the LTR instruction has been executed, further operations on the task register are performed by task switching. As with other segments and LDTs, TSSs and TSS descriptors can be either pre-allocated or allocated as needed.

## 9.8.5 Initializing IA-32e Mode

On Intel 64 processors, the IA32\_EFER MSR is cleared on system reset. The operating system must be in protected mode with paging enabled before attempting to initialize IA-32e mode. IA-32e mode operation also requires physical-address extensions with four levels of enhanced paging structures (see Section 4.5, “IA-32e Paging”).

Operating systems should follow this sequence to initialize IA-32e mode:

1. Starting from protected mode, disable paging by setting CR0.PG = 0. Use the MOV CR0 instruction to disable paging (the instruction must be located in an identity-mapped page).
2. Enable physical-address extensions (PAE) by setting CR4.PAE = 1. Failure to enable PAE will result in a #GP fault when an attempt is made to initialize IA-32e mode.
3. Load CR3 with the physical base address of the Level 4 page map table (PML4).
4. Enable IA-32e mode by setting IA32\_EFER.LME = 1.
5. Enable paging by setting CR0.PG = 1. This causes the processor to set the IA32\_EFER.LMA bit to 1. The MOV CR0 instruction that enables paging and the following instructions must be located in an identity-mapped page (until such time that a branch to non-identity mapped pages can be effected).

64-bit mode paging tables must be located in the first 4 GBytes of physical-address space prior to activating IA-32e mode. This is necessary because the MOV CR3 instruction used to initialize the page-directory base must be executed in legacy mode prior to activating IA-32e mode (setting CR0.PG = 1 to enable paging). Because MOV CR3 is executed in protected mode, only the lower 32 bits of the register are written, limiting the table location to the low 4 GBytes of memory. Software can relocate the page tables anywhere in physical memory after IA-32e mode is activated.

The processor performs 64-bit mode consistency checks whenever software attempts to modify any of the enable bits directly involved in activating IA-32e mode (IA32\_EFER.LME, CR0.PG, and CR4.PAE). It will generate a general protection fault (#GP) if consistency checks fail. 64-bit mode consistency checks ensure that the processor does not enter an undefined mode or state with unpredictable behavior.

64-bit mode consistency checks fail in the following circumstances:

- An attempt is made to enable or disable IA-32e mode while paging is enabled.
- IA-32e mode is enabled and an attempt is made to enable paging prior to enabling physical-address extensions (PAE).
- IA-32e mode is active and an attempt is made to disable physical-address extensions (PAE).
- If the current CS has the L-bit set on an attempt to activate IA-32e mode.
- If the TR contains a 16-bit TSS.

### 9.8.5.1 IA-32e Mode System Data Structures

After activating IA-32e mode, the system-descriptor-table registers (GDTR, LDTR, IDTR, TR) continue to reference legacy protected-mode descriptor tables. Tables referenced by the descriptors all reside in the lower 4 GBytes of linear-address space. After activating IA-32e mode, 64-bit operating-systems should use the LGDT, LLDT, LIDT, and LTR instructions to load the system-descriptor-table registers with references to 64-bit descriptor tables.

### 9.8.5.2 IA-32e Mode Interrupts and Exceptions

Software must not allow exceptions or interrupts to occur between the time IA-32e mode is activated and the update of the interrupt-descriptor-table register (IDTR) that establishes references to a 64-bit interrupt-descriptor table (IDT). This is because the IDT remains in legacy form immediately after IA-32e mode is activated.

If an interrupt or exception occurs prior to updating the IDTR, a legacy 32-bit interrupt gate will be referenced and interpreted as a 64-bit interrupt gate with unpredictable results. External interrupts can be disabled by using the CLI instruction.

Non-maskable interrupts (NMI) must be disabled using external hardware.

### 9.8.5.3 64-bit Mode and Compatibility Mode Operation

IA-32e mode uses two code segment-descriptor bits (CS.L and CS.D, see Figure 3-8) to control the operating modes after IA-32e mode is initialized. If CS.L = 1 and CS.D = 0, the processor is running in 64-bit mode. With this encoding, the default operand size is 32 bits and default address size is 64 bits. Using instruction prefixes, operand size can be changed to 64 bits or 16 bits; address size can be changed to 32 bits.

When IA-32e mode is active and CS.L = 0, the processor operates in compatibility mode. In this mode, CS.D controls default operand and address sizes exactly as it does in the IA-32 architecture. Setting CS.D = 1 specifies default operand and address size as 32 bits. Clearing CS.D to 0 specifies default operand and address size as 16 bits (the CS.L = 1, CS.D = 1 bit combination is reserved).

Compatibility mode execution is selected on a code-segment basis. This mode allows legacy applications to coexist with 64-bit applications running in 64-bit mode. An operating system running in IA-32e mode can execute existing 16-bit and 32-bit applications by clearing their code-segment descriptor's CS.L bit to 0.

In compatibility mode, the following system-level mechanisms continue to operate using the IA-32e-mode architectural semantics:

- Linear-to-physical address translation uses the 64-bit mode extended page-translation mechanism.
- Interrupts and exceptions are handled using the 64-bit mode mechanisms.
- System calls (calls through call gates and SYSENTER/SYSEXIT) are handled using the IA-32e mode mechanisms.

### 9.8.5.4 Switching Out of IA-32e Mode Operation

To return from IA-32e mode to paged-protected mode operation. Operating systems must use the following sequence:

1. Switch to compatibility mode.
2. Deactivate IA-32e mode by clearing CR0.PG = 0. This causes the processor to set IA32\_EFER.LMA = 0. The MOV CR0 instruction used to disable paging and subsequent instructions must be located in an identity-mapped page.
3. Load CR3 with the physical base address of the legacy page-table-directory base address.
4. Disable IA-32e mode by setting IA32\_EFER.LME = 0.
5. Enable legacy paged-protected mode by setting CR0.PG = 1
6. A branch instruction must follow the MOV CR0 that enables paging. Both the MOV CR0 and the branch instruction must be located in an identity-mapped page.

Registers only available in 64-bit mode (R8-R15 and XMM8-XMM15) are preserved across transitions from 64-bit mode into compatibility mode then back into 64-bit mode. However, values of R8-R15 and XMM8-XMM15 are undefined after transitions from 64-bit mode through compatibility mode to legacy or real mode and then back through compatibility mode to 64-bit mode.



## 9.9 MODE SWITCHING

To use the processor in protected mode after hardware or software reset, a mode switch must be performed from real-address mode. Once in protected mode, software generally does not need to return to real-address mode. To run software written to run in real-address mode (8086 mode), it is generally more convenient to run the software in virtual-8086 mode, than to switch back to real-address mode.

### 9.9.1 Switching to Protected Mode

Before switching to protected mode from real mode, a minimum set of system data structures and code modules must be loaded into memory, as described in Section 9.8, “Software Initialization for Protected-Mode Operation.” Once these tables are created, software initialization code can switch into protected mode.

Protected mode is entered by executing a MOV CR0 instruction that sets the PE flag in the CR0 register. (In the same instruction, the PG flag in register CR0 can be set to enable paging.) Execution in protected mode begins with a CPL of 0.

Intel 64 and IA-32 processors have slightly different requirements for switching to protected mode. To insure upwards and downwards code compatibility with Intel 64 and IA-32 processors, we recommend that you follow these steps:

1. Disable interrupts. A CLI instruction disables maskable hardware interrupts. NMI interrupts can be disabled with external circuitry. (Software must guarantee that no exceptions or interrupts are generated during the mode switching operation.)
2. Execute the LGDT instruction to load the GDTR register with the base address of the GDT.
3. Execute a MOV CR0 instruction that sets the PE flag (and optionally the PG flag) in control register CR0.
4. Immediately following the MOV CR0 instruction, execute a far JMP or far CALL instruction. (This operation is typically a far jump or call to the next instruction in the instruction stream.)
5. The JMP or CALL instruction immediately after the MOV CR0 instruction changes the flow of execution and serializes the processor.
6. If paging is enabled, the code for the MOV CR0 instruction and the JMP or CALL instruction must come from a page that is identity mapped (that is, the linear address before the jump is the same as the physical address after paging and protected mode is enabled). The target instruction for the JMP or CALL instruction does not need to be identity mapped.
7. If a local descriptor table is going to be used, execute the LLDT instruction to load the segment selector for the LDT in the LDTR register.
8. Execute the LTR instruction to load the task register with a segment selector to the initial protected-mode task or to a writable area of memory that can be used to store TSS information on a task switch.
9. After entering protected mode, the segment registers continue to hold the contents they had in real-address mode. The JMP or CALL instruction in step 4 resets the CS register. Perform one of the following operations to update the contents of the remaining segment registers.
  - Reload segment registers DS, SS, ES, FS, and GS. If the ES, FS, and/or GS registers are not going to be used, load them with a null selector.
  - Perform a JMP or CALL instruction to a new task, which automatically resets the values of the segment registers and branches to a new code segment.
10. Execute the LIDT instruction to load the IDTR register with the address and limit of the protected-mode IDT.
11. Execute the STI instruction to enable maskable hardware interrupts and perform the necessary hardware operation to enable NMI interrupts.

Random failures can occur if other instructions exist between steps 3 and 4 above. Failures will be readily seen in some situations, such as when instructions that reference memory are inserted between steps 3 and 4 while in system management mode.

## 9.9.2 Switching Back to Real-Address Mode

The processor switches from protected mode back to real-address mode if software clears the PE bit in the CR0 register with a MOV CR0 instruction. A procedure that re-enters real-address mode should perform the following steps:

1. Disable interrupts. A CLI instruction disables maskable hardware interrupts. NMI interrupts can be disabled with external circuitry.
2. If paging is enabled, perform the following operations:
  - Transfer program control to linear addresses that are identity mapped to physical addresses (that is, linear addresses equal physical addresses).
  - Insure that the GDT and IDT are in identity mapped pages.
  - Clear the PG bit in the CR0 register.
  - Move 0H into the CR3 register to flush the TLB.
3. Transfer program control to a readable segment that has a limit of 64 KBytes (FFFFH). This operation loads the CS register with the segment limit required in real-address mode.
4. Load segment registers SS, DS, ES, FS, and GS with a selector for a descriptor containing the following values, which are appropriate for real-address mode:
  - Limit = 64 KBytes (0FFFFH)
  - Byte granular (G = 0)
  - Expand up (E = 0)
  - Writable (W = 1)
  - Present (P = 1)
  - Base = any value

The segment registers must be loaded with non-null segment selectors or the segment registers will be unusable in real-address mode. Note that if the segment registers are not reloaded, execution continues using the descriptor attributes loaded during protected mode.

5. Execute an LIDT instruction to point to a real-address mode interrupt table that is within the 1-MByte real-address mode address range.
6. Clear the PE flag in the CR0 register to switch to real-address mode.
7. Execute a far JMP instruction to jump to a real-address mode program. This operation flushes the instruction queue and loads the appropriate base-address value in the CS register.
8. Load the SS, DS, ES, FS, and GS registers as needed by the real-address mode code. If any of the registers are not going to be used in real-address mode, write 0s to them.
9. Execute the STI instruction to enable maskable hardware interrupts and perform the necessary hardware operation to enable NMI interrupts.

### NOTE

All the code that is executed in steps 1 through 9 must be in a single page and the linear addresses in that page must be identity mapped to physical addresses.

## 9.10 INITIALIZATION AND MODE SWITCHING EXAMPLE

This section provides an initialization and mode switching example that can be incorporated into an application. This code was originally written to initialize the Intel386 processor, but it will execute successfully on the Pentium 4, Intel Xeon, P6 family, Pentium, and Intel486 processors. The code in this example is intended to reside in EPROM and to run following a hardware reset of the processor. The function of the code is to do the following:

- Establish a basic real-address mode operating environment.



- Load the necessary protected-mode system data structures into RAM.
- Load the system registers with the necessary pointers to the data structures and the appropriate flag settings for protected-mode operation.
- Switch the processor to protected mode.

Figure 9-3 shows the physical memory layout for the processor following a hardware reset and the starting point of this example. The EPROM that contains the initialization code resides at the upper end of the processor’s physical memory address range, starting at address FFFFFFFFH and going down from there. The address of the first instruction to be executed is at FFFFFFF0H, the default starting address for the processor following a hardware reset.

The main steps carried out in this example are summarized in Table 9-5. The source listing for the example (with the filename STARTUP.ASM) is given in Example 9-1. The line numbers given in Table 9-5 refer to the source listing.

The following are some additional notes concerning this example:

- When the processor is switched into protected mode, the original code segment base-address value of FFFF0000H (located in the hidden part of the CS register) is retained and execution continues from the current offset in the EIP register. The processor will thus continue to execute code in the EPROM until a far jump or call is made to a new code segment, at which time, the base address in the CS register will be changed.
- Maskable hardware interrupts are disabled after a hardware reset and should remain disabled until the necessary interrupt handlers have been installed. The NMI interrupt is not disabled following a reset. The NMI# pin must thus be inhibited from being asserted until an NMI handler has been loaded and made available to the processor.
- The use of a temporary GDT allows simple transfer of tables from the EPROM to anywhere in the RAM area. A GDT entry is constructed with its base pointing to address 0 and a limit of 4 GBytes. When the DS and ES registers are loaded with this descriptor, the temporary GDT is no longer needed and can be replaced by the application GDT.
- This code loads one TSS and no LDTs. If more TSSs exist in the application, they must be loaded into RAM. If there are LDTs they may be loaded as well.

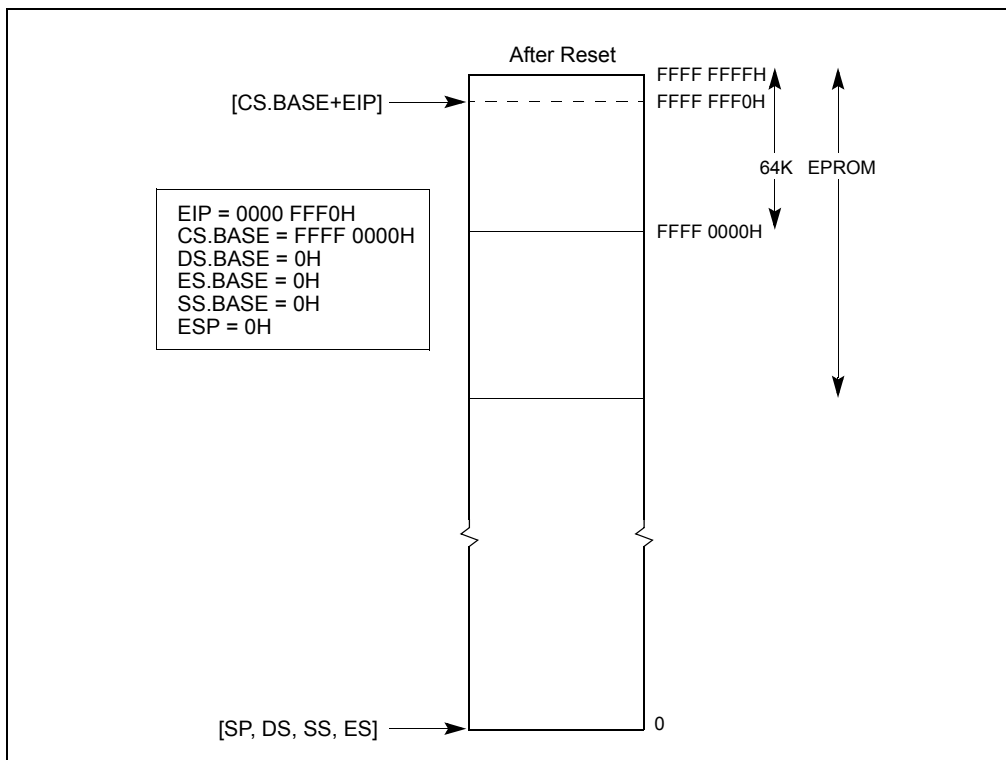


Figure 9-3. Processor State After Reset

**Table 9-5. Main Initialization Steps in STARTUP.ASM Source Listing**

STARTUP.ASM Line Numbers		Description
From	To	
157	157	Jump (short) to the entry code in the EPROM
162	169	Construct a temporary GDT in RAM with one entry: 0 - null 1 - R/W data segment, base = 0, limit = 4 GBytes
171	172	Load the GDTR to point to the temporary GDT
174	177	Load CRO with PE flag set to switch to protected mode
179	181	Jump near to clear real mode instruction queue
184	186	Load DS, ES registers with GDT[1] descriptor, so both point to the entire physical memory space
188	195	Perform specific board initialization that is imposed by the new protected mode
196	218	Copy the application's GDT from ROM into RAM
220	238	Copy the application's IDT from ROM into RAM
241	243	Load application's GDTR
244	245	Load application's IDTR
247	261	Copy the application's TSS from ROM into RAM
263	267	Update TSS descriptor and other aliases in GDT (GDT alias or IDT alias)
277	277	Load the task register (without task switch) using LTR instruction
282	286	Load SS, ESP with the value found in the application's TSS
287	287	Push EFLAGS value found in the application's TSS
288	288	Push CS value found in the application's TSS
289	289	Push EIP value found in the application's TSS
290	293	Load DS, ES with the value found in the application's TSS
296	296	Perform IRET; pop the above values and enter the application code

### 9.10.1 Assembler Usage

In this example, the Intel assembler ASM386 and build tools BLD386 are used to assemble and build the initialization code module. The following assumptions are used when using the Intel ASM386 and BLD386 tools.

- The ASM386 will generate the right operand size opcodes according to the code-segment attribute. The attribute is assigned either by the ASM386 invocation controls or in the code-segment definition.
- If a code segment that is going to run in real-address mode is defined, it must be set to a USE 16 attribute. If a 32-bit operand is used in an instruction in this code segment (for example, MOV EAX, EBX), the assembler automatically generates an operand prefix for the instruction that forces the processor to execute a 32-bit operation, even though its default code-segment attribute is 16-bit.
- Intel's ASM386 assembler allows specific use of the 16- or 32-bit instructions, for example, LGDTW, LGDTD, IRETD. If the generic instruction LGDT is used, the default- segment attribute will be used to generate the right opcode.

### 9.10.2 STARTUP.ASM Listing

Example 9-1 provides high-level sample code designed to move the processor into protected mode. This listing does not include any opcode and offset information.

**Example 9-1. STARTUP.ASM**

MS-DOS\* 5.0(045-N) 386(TM) MACRO ASSEMBLER STARTUP 09:44:51 08/19/92 PAGE 1

MS-DOS 5.0(045-N) 386(TM) MACRO ASSEMBLER V4.0, ASSEMBLY OF MODULE STARTUP  
 OBJECT MODULE PLACED IN startup.obj  
 ASSEMBLER INVOKED BY: f:\386tools\ASM386.EXE startup.a58 pw (132 )

```

LINE      SOURCE

1         NAME      STARTUP
2
3         ;;;;;;;;;;;;;;
4         ;
5         ;   ASSUMPTIONS:
6         ;
7         ;       1.  The bottom 64K of memory is ram, and can be used for
8         ;           scratch space by this module.
9         ;
10        ;       2.  The system has sufficient free usable ram to copy the
11        ;           initial GDT, IDT, and TSS
12        ;
13        ;;;;;;;;;;;;;;
14
15        ; configuration data - must match with build definition
16
17        CS_BASE      EQU      0FFFF0000H
18
19        ; CS_BASE is the linear address of the segment STARTUP_CODE
20        ; - this is specified in the build language file
21
22        RAM_START    EQU      400H
23
24        ; RAM_START is the start of free, usable ram in the linear
25        ; memory space.  The GDT, IDT, and initial TSS will be
26        ; copied above this space, and a small data segment will be
27        ; discarded at this linear address.  The 32-bit word at
28        ; RAM_START will contain the linear address of the first
29        ; free byte above the copied tables - this may be useful if
30        ; a memory manager is used.
31
32        TSS_INDEX    EQU      10
33
34        ; TSS_INDEX is the index of the TSS of the first task to
35        ; run after startup
36
37
38        ;;;;;;;;;;;;;;
39
40        ; ----- STRUCTURES and EQU -----
41        ; structures for system data
42
43        ; TSS structure
44        TASK_STATE   STRUC
45        link         DW ?

```

## PROCESSOR MANAGEMENT AND INITIALIZATION

```
46     link_h    DW ?
47     ESP0     DD ?
48     SS0      DW ?
49     SS0_h    DW ?
50     ESP1     DD ?
51     SS1      DW ?
52     SS1_h    DW ?
53     ESP2     DD ?
54     SS2      DW ?
55     SS2_h    DW ?
56     CR3_reg  DD ?
57     EIP_reg  DD ?
58     EFLAGS_reg DD ?
59     EAX_reg  DD ?
60     ECX_reg  DD ?
61     EDX_reg  DD ?
62     EBX_reg  DD ?
63     ESP_reg  DD ?
64     EBP_reg  DD ?
65     ESI_reg  DD ?
66     EDI_reg  DD ?
67     ES_reg   DW ?
68     ES_h     DW ?
69     CS_reg   DW ?
70     CS_h     DW ?
71     SS_reg   DW ?
72     SS_h     DW ?
73     DS_reg   DW ?
74     DS_h     DW ?
75     FS_reg   DW ?
76     FS_h     DW ?
77     GS_reg   DW ?
78     GS_h     DW ?
79     LDT_reg  DW ?
80     LDT_h    DW ?
81     TRAP_reg DW ?
82     IO_map_base DW ?
83 TASK_STATE ENDS
84
85 ; basic structure of a descriptor
86 DESC     STRUC
87     lim_0_15 DW ?
88     bas_0_15 DW ?
89     bas_16_23 DB ?
90     access   DB ?
91     gran     DB ?
92     bas_24_31 DB ?
93 DESC     ENDS
94
95 ; structure for use with LGDT and LIDT instructions
96 TABLE_REG STRUC
97     table_lim DW ?
98     table_linear DD ?
99 TABLE_REG ENDS
```

```

100
101 ; offset of GDT and IDT descriptors in builder generated GDT
102 GDT_DESC_OFF EQU 1*SIZE(DESC)
103 IDT_DESC_OFF EQU 2*SIZE(DESC)
104
105 ; equates for building temporary GDT in RAM
106 LINEAR_SEL EQU 1*SIZE(DESC)
107 LINEAR_PROTO_LO EQU 00000FFFFH ; LINEAR_ALIAS
108 LINEAR_PROTO_HI EQU 000CF9200H
109
110 ; Protection Enable Bit in CR0
111 PE_BIT EQU 1B
112
113 ; -----
114
115 ; ----- DATA SEGMENT-----
116
117 ; Initially, this data segment starts at linear 0, according
118 ; to the processor's power-up state.
119
120 STARTUP_DATA SEGMENT RW
121
122 free_mem_linear_base LABEL DWORD
123 TEMP_GDT LABEL BYTE ; must be first in segment
124 TEMP_GDT_NULL_DESC DESC <>
125 TEMP_GDT_LINEAR_DESC DESC <>
126
127 ; scratch areas for LGDT and LIDT instructions
128 TEMP_GDT_SCRATCH TABLE_REG <>
129 APP_GDT_RAM TABLE_REG <>
130 APP_IDT_RAM TABLE_REG <>
131 ; align end_data
132 fill DW ?
133
134 ; last thing in this segment - should be on a dword boundary
135 end_data LABEL BYTE
136
137 STARTUP_DATA ENDS
138 ; -----
139
140
141 ; ----- CODE SEGMENT-----
142 STARTUP_CODE SEGMENT ER PUBLIC USE16
143
144 ; filled in by builder
145 PUBLIC GDT_EPROM
146 GDT_EPROM TABLE_REG <>
147
148 ; filled in by builder
149 PUBLIC IDT_EPROM
150 IDT_EPROM TABLE_REG <>
151
152 ; entry point into startup code - the bootstrap will vector
153 ; here with a near JMP generated by the builder. This

```

## PROCESSOR MANAGEMENT AND INITIALIZATION

```
154 ; label must be in the top 64K of linear memory.
155
156     PUBLIC  STARTUP
157 STARTUP:
158
159 ; DS,ES address the bottom 64K of flat linear memory
160     ASSUME  DS:STARTUP_DATA, ES:STARTUP_DATA
161 ; See Figure 9-4
162 ; load GDTR with temporary GDT
163     LEA    EBX,TEMP_GDT ; build the TEMP_GDT in low ram,
164     MOV    DWORD PTR [EBX],0 ; where we can address
165     MOV    DWORD PTR [EBX]+4,0
166     MOV    DWORD PTR [EBX]+8, LINEAR_PROTO_LO
167     MOV    DWORD PTR [EBX]+12, LINEAR_PROTO_HI
168     MOV    TEMP_GDT_scratch.table_linear,EBX
169     MOV    TEMP_GDT_scratch.table_lim,15
170
171     DB 66H; execute a 32 bit LGDT
172     LGDT  TEMP_GDT_scratch
173
174 ; enter protected mode
175     MOV    EBX,CR0
176     OR    EBX,PE_BIT
177     MOV    CR0,EBX
178
179 ; clear prefetch queue
180     JMP    CLEAR_LABEL
181 CLEAR_LABEL:
182
183 ; make DS and ES address 4G of linear memory
184     MOV    CX,LINEAR_SEL
185     MOV    DS,CX
186     MOV    ES,CX
187
188 ; do board specific initialization
189 ;
190 ;
191 ; .....
192 ;
193
194
195 ; See Figure 9-5
196 ; copy EPROM GDT to ram at:
197 ;          RAM_START + size (STARTUP_DATA)
198     MOV    EAX,RAM_START
199     ADD    EAX,OFFSET (end_data)
200     MOV    EBX,RAM_START
201     MOV    ECX, CS_BASE
202     ADD    ECX, OFFSET (GDT_EPROM)
203     MOV    ESI, [ECX].table_linear
204     MOV    EDI,EAX
205     MOVZX  ECX, [ECX].table_lim
206     MOV    APP_GDT_ram[EBX].table_lim,CX
```

```

207     INC     ECX
208     MOV     EDX,EAX
209     MOV     APP_GDT_ram[EBX].table_linear,EAX
210     ADD     EAX,ECX
211     REP MOVSB     BYTE PTR ES:[EDI],BYTE PTR DS:[ESI]
212
213     ; fixup GDT base in descriptor
214     MOV     ECX,EDX
215     MOV     [EDX].bas_0_15+GDT_DESC_OFF,CX
216     ROR     ECX,16
217     MOV     [EDX].bas_16_23+GDT_DESC_OFF,CL
218     MOV     [EDX].bas_24_31+GDT_DESC_OFF,CH
219
220     ; copy EPROM IDT to ram at:
221     ; RAM_START+size(STARTUP_DATA)+SIZE (EPROM GDT)
222     MOV     ECX, CS_BASE
223     ADD     ECX, OFFSET (IDT_EPROM)
224     MOV     ESI, [ECX].table_linear
225     MOV     EDI,EAX
226     MOVZX  ECX, [ECX].table_lim
227     MOV     APP_IDT_ram[EBX].table_lim,CX
228     INC     ECX
229     MOV     APP_IDT_ram[EBX].table_linear,EAX
230     MOV     EBX,EAX
231     ADD     EAX,ECX
232     REP MOVSB     BYTE PTR ES:[EDI],BYTE PTR DS:[ESI]
233
234     ; fixup IDT pointer in GDT
235     MOV     [EDX].bas_0_15+IDT_DESC_OFF,BX
236     ROR     EBX,16
237     MOV     [EDX].bas_16_23+IDT_DESC_OFF,BL
238     MOV     [EDX].bas_24_31+IDT_DESC_OFF,BH
239
240     ; load GDTR and IDTR
241     MOV     EBX,RAM_START
242     DB     66H           ; execute a 32 bit LGDT
243     LGDT   APP_GDT_ram[EBX]
244     DB     66H           ; execute a 32 bit LIDT
245     LIDT   APP_IDT_ram[EBX]
246
247     ; move the TSS
248     MOV     EDI,EAX
249     MOV     EBX,TSS_INDEX*SIZE(DESC)
250     MOV     ECX,GDT_DESC_OFF ;build linear address for TSS
251     MOV     GS,CX
252     MOV     DH,GS:[EBX].bas_24_31
253     MOV     DL,GS:[EBX].bas_16_23
254     ROL     EDX,16
255     MOV     DX,GS:[EBX].bas_0_15
256     MOV     ESI,EDX
257     LSL     ECX,EBX
258     INC     ECX
259     MOV     EDX,EAX
260     ADD     EAX,ECX

```

## PROCESSOR MANAGEMENT AND INITIALIZATION

```
261     REP MOVSB     BYTE PTR ES:[EDI],BYTE PTR DS:[ESI]
262
263         ; fixup TSS pointer
264     MOV     GS:[EBX].bas_0_15,DX
265     ROL     EDX,16
266     MOV     GS:[EBX].bas_24_31,DH
267     MOV     GS:[EBX].bas_16_23,DL
268     ROL     EDX,16
269     ;save start of free ram at linear location RAMSTART
270     MOV     free_mem_linear_base+RAM_START,EAX
271
272     ;assume no LDT used in the initial task - if necessary,
273     ;code to move the LDT could be added, and should resemble
274     ;that used to move the TSS
275
276     ; load task register
277     LTR     BX ; No task switch, only descriptor loading
278     ; See Figure 9-6
279     ; load minimal set of registers necessary to simulate task
280     ; switch
281
282
283     MOV     AX,[EDX].SS_reg ; start loading registers
284     MOV     EDI,[EDX].ESP_reg
285     MOV     SS,AX
286     MOV     ESP,EDI ; stack now valid
287     PUSH   DWORD PTR [EDX].EFLAGS_reg
288     PUSH   DWORD PTR [EDX].CS_reg
289     PUSH   DWORD PTR [EDX].EIP_reg
290     MOV     AX,[EDX].DS_reg
291     MOV     BX,[EDX].ES_reg
292     MOV     DS,AX ; DS and ES no longer linear memory
293     MOV     ES,BX
294
295     ; simulate far jump to initial task
296     IRETD
297
298     STARTUP_CODE ENDS
*** WARNING #377 IN 298, (PASS 2) SEGMENT CONTAINS PRIVILEGED INSTRUCTION(S)
299
300     END STARTUP, DS:STARTUP_DATA, SS:STARTUP_DATA
301
302
```

ASSEMBLY COMPLETE, 1 WARNING, NO ERRORS.



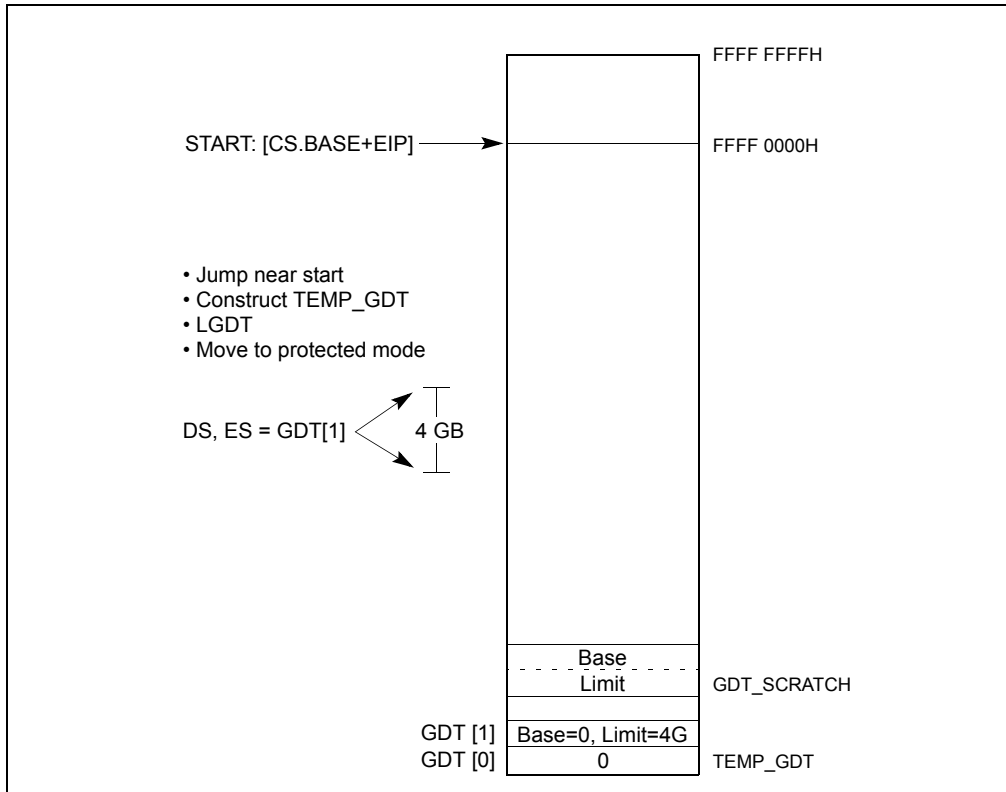


Figure 9-4. Constructing Temporary GDT and Switching to Protected Mode (Lines 162-172 of List File)

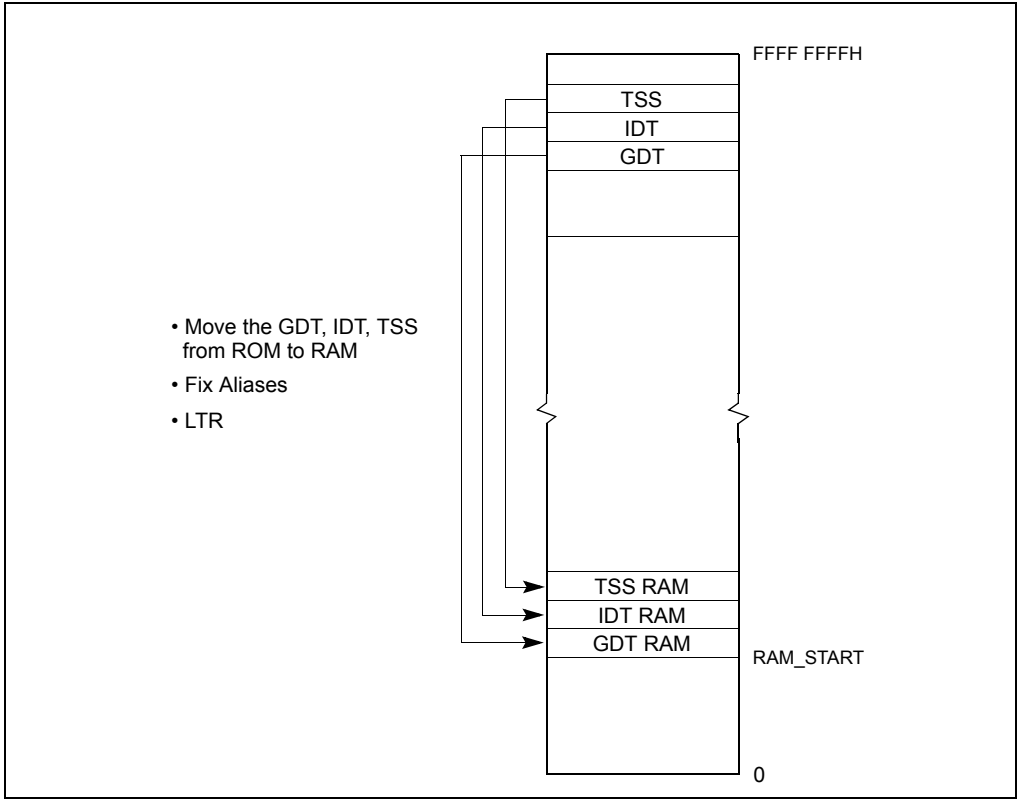


Figure 9-5. Moving the GDT, IDT, and TSS from ROM to RAM (Lines 196-261 of List File)

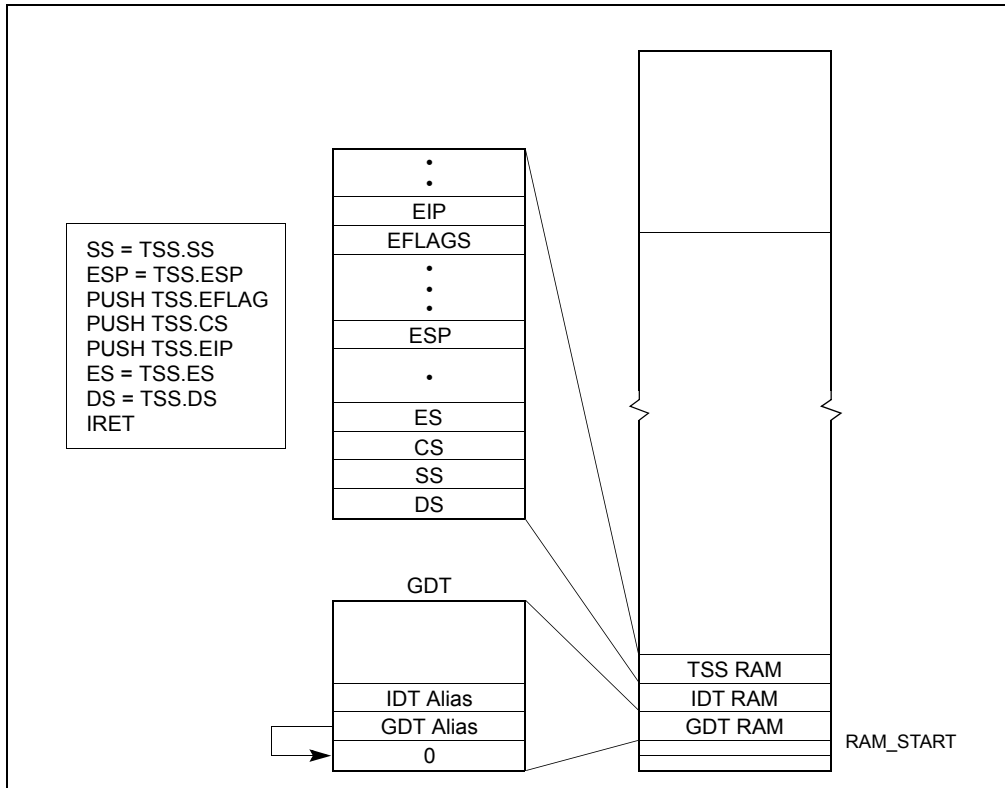


Figure 9-6. Task Switching (Lines 282-296 of List File)

### 9.10.3 MAIN.ASM Source Code

The file MAIN.ASM shown in Example 9-2 defines the data and stack segments for this application and can be substituted with the main module task written in a high-level language that is invoked by the IRET instruction executed by STARTUP.ASM.

#### Example 9-2. MAIN.ASM

```
NAME    main_module
data    SEGMENT RW
        dw 1000 dup(?)
DATA    ENDS
stack   stackseg 800
CODE SEGMENT ER use32 PUBLIC
main_start:
        nop
        nop
        nop
CODE    ENDS
END main_start, ds:data, ss:stack
```

### 9.10.4 Supporting Files

The batch file shown in Example 9-3 can be used to assemble the source code files STARTUP.ASM and MAIN.ASM and build the final application.

**Example 9-3. Batch File to Assemble and Build the Application**

```

ASM386 STARTUP.ASM
ASM386 MAIN.ASM
BLD386 STARTUP.OBJ, MAIN.OBJ buildfile(EPROM.BLD) bootstrap(STARTUP) Bootload

```

BLD386 performs several operations in this example:  
 It allocates physical memory location to segments and tables.  
 It generates tables using the build file and the input files.  
 It links object files and resolves references.  
 It generates a boot-loadable file to be programmed into the EPROM.

Example 9-4 shows the build file used as an input to BLD386 to perform the above functions.

**Example 9-4. Build File**

```

INIT_BLD_EXAMPLE;

SEGMENT
    *SEGMENTS(DPL = 0)
    ,   startup.startup_code(BASE = 0FFFF0000H)
    ;

TASK
    BOOT_TASK(OBJECT = startup, INITIAL,DPL = 0,
              NOT INTENABLED)
    ,   PROTECTED_MODE_TASK(OBJECT = main_module,DPL = 0,
                            NOT INTENABLED)
    ;

TABLE
    GDT (
        LOCATION = GDT_EPROM
        ,   ENTRY = (
            10:   PROTECTED_MODE_TASK
            ,   startup.startup_code
            ,   startup.startup_data
            ,   main_module.data
            ,   main_module.code
            ,   main_module.stack
            )
        ),
    IDT (
        LOCATION = IDT_EPROM
        );

MEMORY
    (
        RESERVE = (0..3FFFH
                  -- Area for the GDT, IDT, TSS copied from ROM
                  ,   60000H..0FFFEFFFFH)
        ,   RANGE = (ROM_AREA = ROM (0FFFF0000H..0FFFFFFFHH)
                    -- Eprom size 64K
                    ,   RANGE = (RAM_AREA = RAM (4000H..05FFFFH))

```

);

END

Table 9-6 shows the relationship of each build item with an ASM source file.

**Table 9-6. Relationship Between BLD Item and ASM Source File**

Item	ASM386 and Startup.A58	BLD386 Controls and BLD file	Effect
Bootstrap	public startup startup:	bootstrap start(startup)	Near jump at OFFFFFFFF0H to start.
GDT location	public GDT_EPROM GDT_EPROM TABLE_REG <>	TABLE GDT(location = GDT_EPROM)	The location of the GDT will be programmed into the GDT_EPROM location.
IDT location	public IDT_EPROM IDT_EPROM TABLE_REG <>	TABLE IDT(location = IDT_EPROM)	The location of the IDT will be programmed into the IDT_EPROM location.
RAM start	RAM_START equ 400H	memory (reserve = (0..3FFFH))	RAM_START is used as the ram destination for moving the tables. It must be excluded from the application's segment area.
Location of the application TSS in the GDT	TSS_INDEX EQU 10	TABLE GDT( ENTRY = (10: PROTECTED_MODE_ TASK))	Put the descriptor of the application TSS in GDT entry 10.
EPROM size and location	size and location of the initialization code	SEGMENT startup.code (base = OFFF0000H) ...memory (RANGE( ROM_AREA = ROM(x..y))	Initialization code size must be less than 64K and resides at upper most 64K of the 4-GByte memory space.

## 9.11 MICROCODE UPDATE FACILITIES

The P6 family and later processors have the capability to correct errata by loading an Intel-supplied data block into the processor. The data block is called a microcode update. This section describes the mechanisms the BIOS needs to provide in order to use this feature during system initialization. It also describes a specification that permits the incorporation of future updates into a system BIOS.

Intel considers the release of a microcode update for a silicon revision to be the equivalent of a processor stepping and completes a full-stepping level validation for releases of microcode updates.

A microcode update is used to correct errata in the processor. The BIOS, which has an update loader, is responsible for loading the update on processors during system initialization (Figure 9-7). There are two steps to this process: the first is to incorporate the necessary update data blocks into the BIOS; the second is to load update data blocks into the processor.

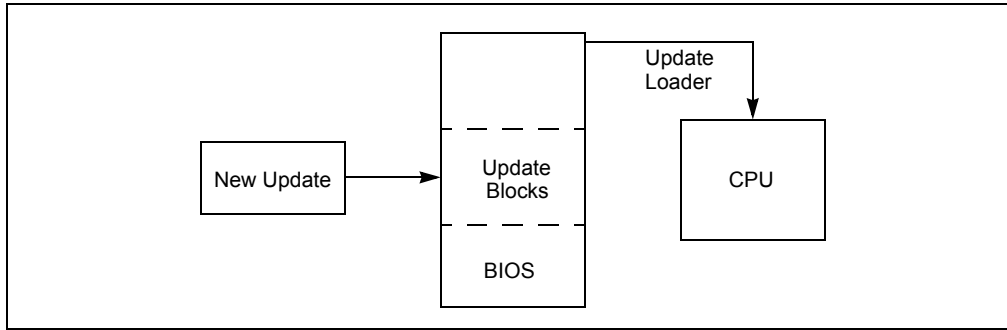


Figure 9-7. Applying Microcode Updates

### 9.11.1 Microcode Update

A microcode update consists of an Intel-supplied binary that contains a descriptive header and data. No executable code resides within the update. Each microcode update is tailored for a specific list of processor signatures. A mismatch of the processor’s signature with the signature contained in the update will result in a failure to load. A processor signature includes the extended family, extended model, type, family, model, and stepping of the processor (starting with processor family 0FH, model 03H, a given microcode update may be associated with one of multiple processor signatures; see Section 9.11.2 for detail).

Microcode updates are composed of a multi-byte header, followed by encrypted data and then by an optional extended signature table. Table 9-7 provides a definition of the fields; Table 9-8 shows the format of an update.

The header is 48 bytes. The first 4 bytes of the header contain the header version. The update header and its reserved fields are interpreted by software based upon the header version. An encoding scheme guards against tampering and provides a means for determining the authenticity of any given update. For microcode updates with a data size field equal to 00000000H, the size of the microcode update is 2048 bytes. The first 48 bytes contain the microcode update header. The remaining 2000 bytes contain encrypted data.

For microcode updates with a data size not equal to 00000000H, the total size field specifies the size of the microcode update. The first 48 bytes contain the microcode update header. The second part of the microcode update is the encrypted data. The data size field of the microcode update header specifies the encrypted data size, its value must be a multiple of the size of DWORD. The total size field of the microcode update header specifies the encrypted data size plus the header size; its value must be in multiples of 1024 bytes (1 KBytes). The optional extended signature table if implemented follows the encrypted data, and its size is calculated by (Total Size – (Data Size + 48)).

#### NOTE

The optional extended signature table is supported starting with processor family 0FH, model 03H.

Table 9-7. Microcode Update Field Definitions

Field Name	Offset (bytes)	Length (bytes)	Description
Header Version	0	4	Version number of the update header.
Update Revision	4	4	Unique version number for the update, the basis for the update signature provided by the processor to indicate the current update functioning within the processor. Used by the BIOS to authenticate the update and verify that the processor loads successfully. The value in this field cannot be used for processor stepping identification alone. This is a signed 32-bit number.
Date	8	4	Date of the update creation in binary format: mmddyyyy (e.g. 07/18/98 is 07181998H).

Table 9-7. Microcode Update Field Definitions (Contd.)

Field Name	Offset (bytes)	Length (bytes)	Description
Processor Signature	12	4	<i>Extended family, extended model, type, family, model, and stepping</i> of processor that requires this particular update revision (e.g., 00000650H). Each microcode update is designed specifically for a given extended family, extended model, <i>type, family, model, and stepping</i> of the processor.  The BIOS uses the processor signature field in conjunction with the CPUID instruction to determine whether or not an update is appropriate to load on a processor. The information encoded within this field exactly corresponds to the bit representations returned by the CPUID instruction.
Checksum	16	4	Checksum of Update Data and Header. Used to verify the integrity of the update header and data. Checksum is correct when the summation of all the DWORDs (including the extended Processor Signature Table) that comprise the microcode update result in 00000000H.
Loader Revision	20	4	Version number of the loader program needed to correctly load this update. The initial version is 00000001H.
Processor Flags	24	4	Platform type information is encoded in the lower 8 bits of this 4-byte field. Each bit represents a particular platform type for a given CPUID. The BIOS uses the processor flags field in conjunction with the platform Id bits in MSR (17H) to determine whether or not an update is appropriate to load on a processor. Multiple bits may be set representing support for multiple platform IDs.
Data Size	28	4	Specifies the size of the encrypted data in bytes, and must be a multiple of DWORDs. If this value is 00000000H, then the microcode update encrypted data is 2000 bytes (or 500 DWORDs).
Total Size	32	4	Specifies the total size of the microcode update in bytes. It is the summation of the header size, the encrypted data size and the size of the optional extended signature table. This value is always a multiple of 1024.
Reserved	36	12	Reserved fields for future expansion
Update Data	48	Data Size or 2000	Update data
Extended Signature Count	Data Size + 48	4	Specifies the number of extended signature structures (Processor Signature[n], processor flags[n] and checksum[n]) that exist in this microcode update.
Extended Checksum	Data Size + 52	4	Checksum of update extended processor signature table. Used to verify the integrity of the extended processor signature table. Checksum is correct when the summation of the DWORDs that comprise the extended processor signature table results in 00000000H.
Reserved	Data Size + 56	12	Reserved fields

**Table 9-7. Microcode Update Field Definitions (Contd.)**

Field Name	Offset (bytes)	Length (bytes)	Description
Processor Signature[n]	Data Size + 68 + (n * 12)	4	<p><i>Extended family, extended model, type, family, model, and stepping</i> of processor that requires this particular update revision (e.g., 00000650H). Each microcode update is designed specifically for a given extended family, extended model, <i>type, family, model, and stepping</i> of the processor.</p> <p>The BIOS uses the processor signature field in conjunction with the CPUID instruction to determine whether or not an update is appropriate to load on a processor. The information encoded within this field exactly corresponds to the bit representations returned by the CPUID instruction.</p>
Processor Flags[n]	Data Size + 72 + (n * 12)	4	Platform type information is encoded in the lower 8 bits of this 4-byte field. Each bit represents a particular platform type for a given CPUID. The BIOS uses the processor flags field in conjunction with the platform Id bits in MSR (17H) to determine whether or not an update is appropriate to load on a processor. Multiple bits may be set representing support for multiple platform IDs.
Checksum[n]	Data Size + 76 + (n * 12)	4	<p>Used by utility software to decompose a microcode update into multiple microcode updates where each of the new updates is constructed without the optional Extended Processor Signature Table.</p> <p>To calculate the Checksum, substitute the Primary Processor Signature entry and the Processor Flags entry with the corresponding Extended Patch entry. Delete the Extended Processor Signature Table entries. The Checksum is correct when the summation of all DWORDs that comprise the created Extended Processor Patch results in 00000000H.</p>

**Table 9-8. Microcode Update Format**

31	24	16	8	0	Bytes
<b>Header Version</b>					0
<b>Update Revision</b>					4
Month: 8		Day: 8		Year: 16	8
<b>Processor Signature (CPUID)</b>					12
Res: 4	Extended Family: 8	Extended Mode: 4	Reserved: 2	Type: 2	Family: 4
					Model: 4
					Stepping: 4
<b>Checksum</b>					16
<b>Loader Revision</b>					20
<b>Processor Flags</b>					24
Reserved (24 bits)					P7 P6 P5 P4 P3 P2 P1 P0
<b>Data Size</b>					28
<b>Total Size</b>					32
<b>Reserved (12 Bytes)</b>					36



**Table 9-8. Microcode Update Format (Contd.)**

31	24	16	8	0	Bytes
Update Data (Data Size bytes, or 2000 Bytes if Data Size = 00000000H)					48
Extended Signature Count 'n'					Data Size + 48
Extended Processor Signature Table Checksum					Data Size + 52
Reserved (12 Bytes)					Data Size + 56
Processor Signature[n]					Data Size + 68 + (n * 12)
Processor Flags[n]					Data Size + 72 + (n * 12)
Checksum[n]					Data Size + 76 + (n * 12)

### 9.11.2 Optional Extended Signature Table

The extended signature table is a structure that may be appended to the end of the encrypted data when the encrypted data only supports a single processor signature (optional case). The extended signature table will always be present when the encrypted data supports multiple processor steppings and/or models (required case).

The extended signature table consists of a 20-byte extended signature header structure, which contains the extended signature count, the extended processor signature table checksum, and 12 reserved bytes (Table 9-9). Following the extended signature header structure, the extended signature table contains 0-to-n extended processor signature structures.

Each processor signature structure consist of the processor signature, processor flags, and a checksum (Table 9-10).

The extended signature count in the extended signature header structure indicates the number of processor signature structures that exist in the extended signature table.

The extended processor signature table checksum is a checksum of all DWORDs that comprise the extended signature table. That includes the extended signature count, extended processor signature table checksum, 12 reserved bytes and the n processor signature structures. A valid extended signature table exists when the result of a DWORD checksum is 00000000H.

**Table 9-9. Extended Processor Signature Table Header Structure**

Extended Signature Count 'n'	Data Size + 48
Extended Processor Signature Table Checksum	Data Size + 52
Reserved (12 Bytes)	Data Size + 56

**Table 9-10. Processor Signature Structure**

Processor Signature[n]	Data Size + 68 + (n * 12)
Processor Flags[n]	Data Size + 72 + (n * 12)
Checksum[n]	Data Size + 76 + (n * 12)

### 9.11.3 Processor Identification

Each microcode update is designed to for a specific processor or set of processors. To determine the correct microcode update to load, software must ensure that one of the processor signatures embedded in the microcode update matches the 32-bit processor signature returned by the CPUID instruction when executed by the target processor with EAX = 1. Attempting to load a microcode update that does not match a processor signature embedded in the microcode update with the processor signature returned by CPUID will cause the BIOS to reject the update.

Example 9-5 shows how to check for a valid processor signature match between the processor and microcode update.

#### Example 9-5. Pseudo Code to Validate the Processor Signature

```

ProcessorSignature ← CPUID(1):EAX

If (Update.HeaderVersion = 00000001h)
{
    // first check the ProcessorSignature field
    If (ProcessorSignature = Update.ProcessorSignature)
        Success

    // if extended signature is present
    Else If (Update.TotalSize > (Update.DataSize + 48))
    {

        //
        // Assume the Data Size has been used to calculate the
        // location of Update.ProcessorSignature[0].
        //

        For (N ← 0; ((N < Update.ExtendedSignatureCount) AND
            (ProcessorSignature ≠ Update.ProcessorSignature[N])); N++);

            // if the loops ended when the iteration count is
            // less than the number of processor signatures in
            // the table, we have a match
            If (N < Update.ExtendedSignatureCount)
                Success
            Else
                Fail
    }
    Else
        Fail
Else
    Fail

```

### 9.11.4 Platform Identification

In addition to verifying the processor signature, the intended processor platform type must be determined to properly target the microcode update. The intended processor platform type is determined by reading the IA32\_PLATFORM\_ID register, (MSR 17H). This 64-bit register must be read using the RDMSR instruction.

The three platform ID bits, when read as a binary coded decimal (BCD) number, indicate the bit position in the microcode update header's processor flags field associated with the installed processor. The processor flags in the 48-byte header and the processor flags field associated with the extended processor signature structures may have multiple bits set. Each set bit represents a different platform ID that the update supports.

Register Name: IA32\_PLATFORM\_ID  
MSR Address: 017H

Access: Read Only

IA32\_PLATFORM\_ID is a 64-bit register accessed only when referenced as a Qword through a RDMSR instruction.

**Table 9-11. Processor Flags**

Bit	Descriptions																																				
63:53	Reserved																																				
52:50	Platform Id Bits (RO). The field gives information concerning the intended platform for the processor. See also Table 9-8.  <table style="margin-left: 40px; border-collapse: collapse;"> <tr> <td style="padding-right: 10px;">52</td> <td style="padding-right: 10px;">51</td> <td style="padding-right: 10px;">50</td> <td></td> </tr> <tr> <td>0</td> <td>0</td> <td>0</td> <td>Processor Flag 0</td> </tr> <tr> <td>0</td> <td>0</td> <td>1</td> <td>Processor Flag 1</td> </tr> <tr> <td>0</td> <td>1</td> <td>0</td> <td>Processor Flag 2</td> </tr> <tr> <td>0</td> <td>1</td> <td>1</td> <td>Processor Flag 3</td> </tr> <tr> <td>1</td> <td>0</td> <td>0</td> <td>Processor Flag 4</td> </tr> <tr> <td>1</td> <td>0</td> <td>1</td> <td>Processor Flag 5</td> </tr> <tr> <td>1</td> <td>1</td> <td>0</td> <td>Processor Flag 6</td> </tr> <tr> <td>1</td> <td>1</td> <td>1</td> <td>Processor Flag 7</td> </tr> </table>	52	51	50		0	0	0	Processor Flag 0	0	0	1	Processor Flag 1	0	1	0	Processor Flag 2	0	1	1	Processor Flag 3	1	0	0	Processor Flag 4	1	0	1	Processor Flag 5	1	1	0	Processor Flag 6	1	1	1	Processor Flag 7
52	51	50																																			
0	0	0	Processor Flag 0																																		
0	0	1	Processor Flag 1																																		
0	1	0	Processor Flag 2																																		
0	1	1	Processor Flag 3																																		
1	0	0	Processor Flag 4																																		
1	0	1	Processor Flag 5																																		
1	1	0	Processor Flag 6																																		
1	1	1	Processor Flag 7																																		
49:0	Reserved																																				

To validate the platform information, software may implement an algorithm similar to the algorithms in Example 9-6.

#### Example 9-6. Pseudo Code Example of Processor Flags Test

```

Flag ← 1 << IA32_PLATFORM_ID[52:50]

If (Update.HeaderVersion = 00000001h)
{
  If (Update.ProcessorFlags & Flag)
  {
    Load Update
  }
  Else
  {

    //
    // Assume the Data Size has been used to calculate the
    // location of Update.ProcessorSignature[N] and a match
    // on Update.ProcessorSignature[N] has already succeeded
    //

    If (Update.ProcessorFlags[n] & Flag)
    {
      Load Update
    }
  }
}

```

### 9.11.5 Microcode Update Checksum

Each microcode update contains a DWORD checksum located in the update header. It is software's responsibility to ensure that a microcode update is not corrupt. To check for a corrupt microcode update, software must perform an unsigned DWORD (32-bit) checksum of the microcode update. Even though some fields are signed, the checksum

procedure treats all DWORDs as unsigned. Microcode updates with a header version equal to 00000001H must sum all DWORDs that comprise the microcode update. A valid checksum check will yield a value of 00000000H. Any other value indicates the microcode update is corrupt and should not be loaded.

The checksum algorithm shown by the pseudo code in Example 9-7 treats the microcode update as an array of unsigned DWORDs. If the data size DWORD field at byte offset 32 equals 00000000H, the size of the encrypted data is 2000 bytes, resulting in 500 DWORDs. Otherwise the microcode update size in DWORDs = (*Total Size* / 4), where the total size is a multiple of 1024 bytes (1 KBytes).

#### Example 9-7. Pseudo Code Example of Checksum Test

```
N ← 512

If (Update.DataSize ≠ 00000000H)
    N ← Update.TotalSize / 4

ChkSum ← 0
For (I ← 0; I < N; I++)
{
    ChkSum ← ChkSum + MicrocodeUpdate[I]
}

If (ChkSum = 00000000H)
    Success
Else
    Fail
```

### 9.11.6 Microcode Update Loader

This section describes an update loader used to load an update into a P6 family or later processors. It also discusses the requirements placed on the BIOS to ensure proper loading. The update loader described contains the minimal instructions needed to load an update. The specific instruction sequence that is required to load an update is dependent upon the loader revision field contained within the update header. This revision is expected to change infrequently (potentially, only when new processor models are introduced).

Example 9-8 below represents the update loader with a loader revision of 00000001H. Note that the microcode update must be aligned on a 16-byte boundary and the size of the microcode update must be 1-KByte granular.

#### Example 9-8. Assembly Code Example of Simple Microcode Update Loader

```
mov  ecx,79h           ; MSR to write in ECX
xor  eax,eax          ; clear EAX
xor  ebx,ebx          ; clear EBX
mov  ax,cs             ; Segment of microcode update
shl  eax,4
mov  bx,offset Update ; Offset of microcode update
add  eax,ebx           ; Linear Address of Update in EAX
add  eax,48d           ; Offset of the Update Data within the Update
xor  edx,edx           ; Zero in EDX
WRMSR                  ; microcode update trigger
```

The loader shown in Example 9-8 assumes that *update* is the address of a microcode update (header and data) embedded within the code segment of the BIOS. It also assumes that the processor is operating in real mode. The data may reside anywhere in memory, aligned on a 16-byte boundary, that is accessible by the processor within its current operating mode.

Before the BIOS executes the microcode update trigger (WRMSR) instruction, the following must be true:

- In 64-bit mode, EAX contains the lower 32-bits of the microcode update linear address. In protected mode, EAX contains the full 32-bit linear address of the microcode update.
- In 64-bit mode, EDX contains the upper 32-bits of the microcode update linear address. In protected mode, EDX equals zero.
- ECX contains 79H (address of IA32\_BIOS\_UPDT\_TRIG).

Other requirements are:

- If the update is loaded while the processor is in real mode, then the update data may not cross a segment boundary.
- If the update is loaded while the processor is in real mode, then the update data may not exceed a segment limit.
- If paging is enabled, pages that are currently present must map the update data.
- The microcode update data requires a 16-byte boundary alignment.

### 9.11.6.1 Hard Resets in Update Loading

The effects of a loaded update are cleared from the processor upon a hard reset. Therefore, each time a hard reset is asserted during the BIOS POST, the update must be reloaded on all processors that observed the reset. The effects of a loaded update are, however, maintained across a processor INIT. There are no side effects caused by loading an update into a processor multiple times.

### 9.11.6.2 Update in a Multiprocessor System

A multiprocessor (MP) system requires loading each processor with update data appropriate for its CPUID and platform ID bits. The BIOS is responsible for ensuring that this requirement is met and that the loader is located in a module executed by all processors in the system. If a system design permits multiple steppings of Pentium 4, Intel Xeon, and P6 family processors to exist concurrently; then the BIOS must verify individual processors against the update header information to ensure appropriate loading. Given these considerations, it is most practical to load the update during MP initialization.

### 9.11.6.3 Update in a System Supporting Intel Hyper-Threading Technology

Intel Hyper-Threading Technology has implications on the loading of the microcode update. The update must be loaded for each core in a physical processor. Thus, for a processor supporting Intel Hyper-Threading Technology, only one logical processor per core is required to load the microcode update. Each individual logical processor can independently load the update. However, MP initialization must provide some mechanism (e.g. a software semaphore) to force serialization of microcode update loads and to prevent simultaneous load attempts to the same core.

### 9.11.6.4 Update in a System Supporting Dual-Core Technology

Dual-core technology has implications on the loading of the microcode update. The microcode update facility is not shared between processor cores in the same physical package. The update must be loaded for each core in a physical processor.

If processor core supports Intel Hyper-Threading Technology, the guideline described in Section 9.11.6.3 also applies.

### 9.11.6.5 Update Loader Enhancements

The update loader presented in Section 9.11.6, "Microcode Update Loader," is a minimal implementation that can be enhanced to provide additional functionality. Potential enhancements are described below:

- BIOS can incorporate multiple updates to support multiple steppings of the Pentium 4, Intel Xeon, and P6 family processors. This feature provides for operating in a mixed stepping environment on an MP system and enables a user to upgrade to a later version of the processor. In this case, modify the loader to check the CPUID

and platform ID bits of the processor that it is running on against the available headers before loading a particular update. The number of updates is only limited by available BIOS space.

- A loader can load the update and test the processor to determine if the update was loaded correctly. See Section 9.11.7, "Update Signature and Verification."
- A loader can verify the integrity of the update data by performing a checksum on the double words of the update summing to zero. See Section 9.11.5, "Microcode Update Checksum."
- A loader can provide power-on messages indicating successful loading of an update.

### 9.11.7 Update Signature and Verification

The P6 family and later processors provide capabilities to verify the authenticity of a particular update and to identify the current update revision. This section describes the model-specific extensions of processors that support this feature. The update verification method below assumes that the BIOS will only verify an update that is more recent than the revision currently loaded in the processor.

CPUID returns a value in a model specific register in addition to its usual register return values. The semantics of CPUID cause it to deposit an update ID value in the 64-bit model-specific register at address 08BH (IA32\_BIOS\_SIGN\_ID). If no update is present in the processor, the value in the MSR remains unmodified. The BIOS must pre-load a zero into the MSR before executing CPUID. If a read of the MSR at 8BH still returns zero after executing CPUID, this indicates that no update is present.

The update ID value returned in the EDX register after RDMSR executes indicates the revision of the update loaded in the processor. This value, in combination with the CPUID value returned in the EAX register, uniquely identifies a particular update. The signature ID can be directly compared with the update revision field in a microcode update header for verification of a correct load. No consecutive updates released for a given stepping of a processor may share the same signature. The processor signature returned by CPUID differentiates updates for different step-pings.

#### 9.11.7.1 Determining the Signature

An update that is successfully loaded into the processor provides a signature that matches the update revision of the currently functioning revision. This signature is available any time after the actual update has been loaded. Requesting the signature does not have a negative impact upon a loaded update.

The procedure for determining this signature shown in Example 9-9.

#### Example 9-9. Assembly Code to Retrieve the Update Revision

```

MOV    ECX, 08BH           ;IA32_BIOS_SIGN_ID
XOR    EAX, EAX           ;clear EAX
XOR    EDX, EDX          ;clear EDX
WRMSR                ;Load 0 to MSR at 8BH
MOV    EAX, 1
cpuid
MOV    ECX, 08BH         ;IA32_BIOS_SIGN_ID
rdmsr                ;Read Model Specific Register
    
```

If there is an update active in the processor, its revision is returned in the EDX register after the RDMSR instruction executes.

IA32_BIOS_SIGN_ID	Microcode Update Signature Register
MSR Address:	08BH Accessed as a Qword
Default Value:	XXXX XXXX XXXX XXXXh
Access:	Read/Write

The IA32\_BIOS\_SIGN\_ID register is used to report the microcode update signature when CPUID executes. The signature is returned in the upper DWORD (Table 9-12).

**Table 9-12. Microcode Update Signature**

Bit	Description
63:32	Microcode update signature. This field contains the signature of the currently loaded microcode update when read following the execution of the CPUID instruction, function 1. It is required that this register field be pre-loaded with zero prior to executing the CPUID, function 1. If the field remains equal to zero, then there is no microcode update loaded. Another non-zero value will be the signature.
31:0	Reserved.

### 9.11.7.2 Authenticating the Update

An update may be authenticated by the BIOS using the signature primitive, described above, and the algorithm in Example 9-10.

#### Example 9-10. Pseudo Code to Authenticate the Update

```
Z ← Obtain Update Revision from the Update Header to be authenticated;
X ← Obtain Current Update Signature from MSR 8BH;

If (Z > X)
{
  Load Update that is to be authenticated;
  Y ← Obtain New Signature from MSR 8BH;

  If (Z = Y)
    Success
  Else
    Fail
}
Else
  Fail
```

Example 9-10 requires that the BIOS only authenticate updates that contain a numerically larger revision than the currently loaded revision, where Current Signature ( $X$ ) < New Update Revision ( $Z$ ). A processor with no loaded update is considered to have a revision equal to zero.

This authentication procedure relies upon the decoding provided by the processor to verify an update from a potentially hostile source. As an example, this mechanism in conjunction with other safeguards provides security for dynamically incorporating field updates into the BIOS.

### 9.11.8 Optional Processor Microcode Update Specifications

This section an interface that an OEM-BIOS may provide to its client system software to manage processor microcode updates. System software may choose to build its own facility to manage microcode updates (e.g. similar to the facility described in Section 9.11.6) or rely on a facility provided by the BIOS to perform microcode updates.

Sections 9.11.8.1-9.11.8.9 describes an extension (Function 0D042H) to the real mode INT 15H service. INT 15H 0D042H function is one of several alternatives that a BIOS may choose to implement microcode update facility and offer to its client application (e.g. an OS). Other alternative microcode update facility that BIOS can choose are dependent on platform-specific capabilities, including the Capsule Update mechanism from the UEFI specification ([www.uefi.org](http://www.uefi.org)). In this discussion, the application is referred to as the calling program or caller.

The real mode INT15 call specification described here is an Intel extension to an OEM BIOS. This extension allows an application to read and modify the contents of the microcode update data in NVRAM. The update loader, which is part of the system BIOS, cannot be updated by the interface. All of the functions defined in the specification must be implemented for a system to be considered compliant with the specification. The INT15 functions are accessible only from real mode.

### 9.11.8.1 Responsibilities of the BIOS

If a BIOS passes the presence test (INT 15H, AX = 0D042H, BL = 0H), it must implement all of the sub-functions defined in the INT 15H, AX = 0D042H specification. There are no optional functions. BIOS must load the appropriate update for each processor during system initialization.

A Header Version of an update block containing the value 0FFFFFFFH indicates that the update block is unused and available for storing a new update.

The BIOS is responsible for providing a region of non-volatile storage (NVRAM) for each potential processor stepping within a system. This storage unit consists of one or more update blocks. An update block is a contiguous 2048-byte block of memory. The BIOS for a single processor system need only provide update blocks to store one microcode update. If the BIOS for a multiple processor system is intended to support mixed processor steppings, then the BIOS needs to provide enough update blocks to store each unique microcode update or for each processor socket on the OEM's system board.

The BIOS is responsible for managing the NVRAM update blocks. This includes garbage collection, such as removing microcode updates that exist in NVRAM for which a corresponding processor does not exist in the system. This specification only provides the mechanism for ensuring security, the uniqueness of an entry, and that stale entries are not loaded. The actual update block management is implementation specific on a per-BIOS basis.

As an example, the BIOS may use update blocks sequentially in ascending order with CPU signatures sorted versus the first available block. In addition, garbage collection may be implemented as a setup option to clear all NVRAM slots or as BIOS code that searches and eliminates unused entries during boot.

### NOTES

For IA-32 processors starting with family 0FH and model 03H and Intel 64 processors, the microcode update may be as large as 16 KBytes. Thus, BIOS must allocate 8 update blocks for each microcode update. In a MP system, a common microcode update may be sufficient for each socket in the system.

For IA-32 processors earlier than family 0FH and model 03H, the microcode update is 2 KBytes. An MP-capable BIOS that supports multiple steppings must allocate a block for each socket in the system.

A single-processor BIOS that supports variable-sized microcode update and fixed-sized microcode update must allocate one 16-KByte region and a second region of at least 2 KBytes.

The following algorithm (Example 9-11) describes the steps performed during BIOS initialization used to load the updates into the processor(s). The algorithm assumes:

- The BIOS ensures that no update contained within NVRAM has a header version or loader version that does not match one currently supported by the BIOS.
- The update contains a correct checksum.
- The BIOS ensures that (at most) one update exists for each processor stepping.
- Older update revisions are not allowed to overwrite more recent ones.

These requirements are checked by the BIOS during the execution of the write update function of this interface. The BIOS sequentially scans through all of the update blocks in NVRAM starting with index 0. The BIOS scans until it finds an update where the processor fields in the header match the processor signature (extended family, extended model, type, family, model, and stepping) as well as the platform bits of the current processor.

#### Example 9-11. Pseudo Code, Checks Required Prior to Loading an Update

```

For each processor in the system
{
    Determine the Processor Signature via CPUID function 1;
    Determine the Platform Bits ← 1 << IA32_PLATFORM_ID[52:50];

    For (I ← UpdateBlock 0, I < NumOfBlocks; I++)
    {
        If (Update.Header_Version = 00000001H)
        {

```



```

If ((Update.ProcessorSignature = Processor Signature) &&
    (Update.ProcessorFlags & Platform Bits))
{
    Load Update.UpdateData into the Processor;
    Verify update was correctly loaded into the processor
    Go on to next processor
    Break;
}
Else If (Update.TotalSize > (Update.DataSize + 48))
{
    N ← 0
    While (N < Update.ExtendedSignatureCount)
    {
        If ((Update.ProcessorSignature[N] =
            Processor Signature) &&
            (Update.ProcessorFlags[N] & Platform Bits))
        {
            Load Update.UpdateData into the Processor;
            Verify update correctly loaded into the processor
            Go on to next processor
            Break;
        }
        N ← N + 1
    }
    I ← I + (Update.TotalSize / 2048)
    If ((Update.TotalSize MOD 2048) = 0)
        I ← I + 1
    }
}
}
}
}

```

## NOTES

The platform Id bits in IA32\_PLATFORM\_ID are encoded as a three-bit binary coded decimal field. The platform bits in the microcode update header are individually bit encoded. The algorithm must do a translation from one format to the other prior to doing a check.

When performing the INT 15H, 0D042H functions, the BIOS must assume that the caller has no knowledge of platform specific requirements. It is the responsibility of BIOS calls to manage all chipset and platform specific prerequisites for managing the NVRAM device. When writing the update data using the Write Update sub-function, the BIOS must maintain implementation specific data requirements (such as the update of NVRAM checksum). The BIOS should also attempt to verify the success of write operations on the storage device used to record the update.

### 9.11.8.2 Responsibilities of the Calling Program

This section of the document lists the responsibilities of a calling program using the interface specifications to load microcode update(s) into BIOS NVRAM.

- The calling program should call the INT 15H, 0D042H functions from a pure real mode program and should be executing on a system that is running in pure real mode.
- The caller should issue the presence test function (sub function 0) and verify the signature and return codes of that function.
- It is important that the calling program provides the required scratch RAM buffers for the BIOS and the proper stack size as specified in the interface definition.
- The calling program should read any update data that already exists in the BIOS in order to make decisions about the appropriateness of loading the update. The BIOS must refuse to overwrite a newer update with an

older version. The update header contains information about version and processor specifics for the calling program to make an intelligent decision about loading.

- There can be no ambiguous updates. The BIOS must refuse to allow multiple updates for the same CPU to exist at the same time; it also must refuse to load updates for processors that don't exist on the system.
- The calling application should implement a verify function that is run after the update write function successfully completes. This function reads back the update and verifies that the BIOS returned an image identical to the one that was written.

Example 9-12 represents a calling program.

**Example 9-12. INT 15 D042 Calling Program Pseudo-code**

```
//
// We must be in real mode
//
If the system is not in Real mode exit
//
// Detect presence of Genuine Intel processor(s) that can be updated
// using(CPUID)
//
If no Intel processors exist that can be updated exit
//
// Detect the presence of the Intel microcode update extensions
//
If the BIOS fails the PresenceTestexit
//
// If the APIC is enabled, see if any other processors are out there
//
Read IA32_APICBASE
If APIC enabled
{
    Send Broadcast Message to all processors except self via APIC
    Have all processors execute CPUID, record the Processor Signature
    (i.e., Extended Family, Extended Model, Type, Family, Model, Stepping)
    Have all processors read IA32_PLATFORM_ID[52:50], record Platform
    Id Bits

    If current processor cannot be updated
        exit
}
//
// Determine the number of unique update blocks needed for this system
//
NumBlocks = 0
For each processor
{
    If ((this is a unique processor stepping) AND
        (we have a unique update in the database for this processor))
    {
        Checksum the update from the database;
        If Checksum fails
            exit
        NumBlocks ← NumBlocks + size of microcode update / 2048
    }
}
//
// Do we have enough update slots for all CPUs?
//
```

```

If there are more blocks required to support the unique processor steppings than update blocks
provided by the BIOS exit
//
// Do we need any update blocks at all?  If not, we are done
//
If (NumBlocks = 0)
    exit
//
// Record updates for processors in NVRAM.
//
For (I=0; I<NumBlocks; I++)
{
    //
    // Load each Update
    //
    Issue the WriteUpdate function

    If (STORAGE_FULL) returned
    {
        Display Error -- BIOS is not managing NVRAM appropriately
        exit
    }

    If (INVALID_REVISION) returned
    {
        Display Message: More recent update already loaded in NVRAM for
        this stepping
        continue
    }

    If any other error returned
    {
        Display Diagnostic
        exit
    }

    //
    // Verify the update was loaded correctly
    //
    Issue the ReadUpdate function

    If an error occurred
    {
        Display Diagnostic
        exit
    }
    //
    // Compare the Update read to that written
    //
    If (Update read ≠ Update written)
    {
        Display Diagnostic
        exit
    }

    I ← I + (size of microcode update / 2048)
}
//
// Enable Update Loading, and inform user

```

//  
Issue the Update Control function with Task = Enable.

### 9.11.8.3 Microcode Update Functions

Table 9-13 defines the processor microcode update functions that implementations of INT 15H 0D042H must support.

**Table 9-13. Microcode Update Functions**

Microcode Update Function	Function Number	Description	Required/Optional
Presence test	00H	Returns information about the supported functions.	Required
Write update data	01H	Writes one of the update data areas (slots).	Required
Update control	02H	Globally controls the loading of updates.	Required
Read update data	03H	Reads one of the update data areas (slots).	Required

### 9.11.8.4 INT 15H-based Interface

If an OEM-BIOS is implementing INT 15H 0D042H interface and offer to its client, the BIOS should allow additional microcode updates to be added to system flash.

The program that calls this interface is responsible for providing three 64-kilobyte RAM areas for BIOS use during calls to the read and write functions. These RAM scratch pads can be used by the BIOS for any purpose, but only for the duration of the function call. The calling routine places real mode segments pointing to the RAM blocks in the CX, DX and SI registers. Calls to functions in this interface must be made with a minimum of 32 kilobytes of stack available to the BIOS.

In general, each function returns with CF cleared and AH contains the returned status. The general return codes and other constant definitions are listed in Section 9.11.8.9, "Return Codes."

The OEM error field (AL) is provided for the OEM to return additional error information specific to the platform. If the BIOS provides no additional information about the error, OEM error must be set to SUCCESS. The OEM error field is undefined if AH contains either SUCCESS (00H) or NOT\_IMPLEMENTED (86H). In all other cases, it must be set with either SUCCESS or a value meaningful to the OEM.

The following sections describe functions provided by the INT15H-based interface.

### 9.11.8.5 Function 00H—Presence Test

This function verifies that the BIOS has implemented required microcode update functions. Table 9-14 lists the parameters and return codes for the function.

**Table 9-14. Parameters for the Presence Test**

Input		
AX	Function Code	0D042H
BL	Sub-function	00H - Presence test
Output		
CF	Carry Flag	Carry Set - Failure - AH contains status Carry Clear - All return values valid
AH	Return Code	
AL	OEM Error	Additional OEM information.
EBX	Signature Part 1	'INTE' - Part one of the signature
ECX	Signature Part 2	'LPEP' - Part two of the signature
EDX	Loader Version	Version number of the microcode update loader

**Table 9-14. Parameters for the Presence Test (Contd.)**

Input		
SI	Update Count	Number of 2048 update blocks in NVRAM the BIOS allocated to storing microcode updates
Return Codes (see Table 9-19 for code definitions)		
SUCCESS		The function completed successfully.
NOT_IMPLEMENTED		The function is not implemented.

In order to assure that the BIOS function is present, the caller must verify the carry flag, the return code, and the 64-bit signature. The update count reflects the number of 2048-byte blocks available for storage within one non-volatile RAM.

The loader version number refers to the revision of the update loader program that is included in the system BIOS image.

### 9.11.8.6 Function 01H—Write Microcode Update Data

This function integrates a new microcode update into the BIOS storage device. Table 9-15 lists the parameters and return codes for the function.

**Table 9-15. Parameters for the Write Update Data Function**

Input		
AX	Function Code	0D042H
BL	Sub-function	01H - Write update
ES:DI	Update Address	Real Mode pointer to the Intel Update structure. This buffer is 2048 bytes in length if the processor supports only fixed-size microcode update or...  Real Mode pointer to the Intel Update structure. This buffer is 64 KBytes in length if the processor supports a variable-size microcode update.
CX	Scratch Pad1	Real mode segment address of 64 KBytes of RAM block
DX	Scratch Pad2	Real mode segment address of 64 KBytes of RAM block
SI	Scratch Pad3	Real mode segment address of 64 KBytes of RAM block
SS:SP	Stack pointer	32 KBytes of stack minimum
Output		
CF	Carry Flag	Carry Set - Failure - AH Contains status Carry Clear - All return values valid
AH	Return Code	Status of the call
AL	OEM Error	Additional OEM information
Return Codes (see Table 9-19 for code definitions)		
SUCCESS		The function completed successfully.
NOT_IMPLEMENTED		The function is not implemented.
WRITE_FAILURE		A failure occurred because of the inability to write the storage device.
ERASE_FAILURE		A failure occurred because of the inability to erase the storage device.
READ_FAILURE		A failure occurred because of the inability to read the storage device.

**Table 9-15. Parameters for the Write Update Data Function (Contd.)**

Input	
STORAGE_FULL	The BIOS non-volatile storage area is unable to accommodate the update because all available update blocks are filled with updates that are needed for processors in the system.
CPU_NOT_PRESENT	The processor stepping does not currently exist in the system.
INVALID_HEADER	The update header contains a header or loader version that is not recognized by the BIOS.
INVALID_HEADER_CS	The update does not checksum correctly.
SECURITY_FAILURE	The processor rejected the update.
INVALID_REVISION	The same or more recent revision of the update exists in the storage device.

**Description**

The BIOS is responsible for selecting an appropriate update block in the non-volatile storage for storing the new update. This BIOS is also responsible for ensuring the integrity of the information provided by the caller, including authenticating the proposed update before incorporating it into storage.

Before writing the update block into NVRAM, the BIOS should ensure that the update structure meets the following criteria in the following order:

1. The update header version should be equal to an update header version recognized by the BIOS.
2. The update loader version in the update header should be equal to the update loader version contained within the BIOS image.
3. The update block must checksum. This checksum is computed as a 32-bit summation of all double words in the structure, including the header, data, and processor signature table.

The BIOS selects update block(s) in non-volatile storage for storing the candidate update. The BIOS can select any available update block as long as it guarantees that only a single update exists for any given processor stepping in non-volatile storage. If the update block selected already contains an update, the following additional criteria apply to overwrite it:

- The processor signature in the proposed update must be equal to the processor signature in the header of the current update in NVRAM (Processor Signature + platform ID bits).
- The update revision in the proposed update should be greater than the update revision in the header of the current update in NVRAM.

If no unused update blocks are available and the above criteria are not met, the BIOS can overwrite update block(s) for a processor stepping that is no longer present in the system. This can be done by scanning the update blocks and comparing the processor steppings, identified in the MP Specification table, to the processor steppings that currently exist in the system.

Finally, before storing the proposed update in NVRAM, the BIOS must verify the authenticity of the update via the mechanism described in Section 9.11.6, "Microcode Update Loader." This includes loading the update into the current processor, executing the CPUID instruction, reading MSR 08Bh, and comparing a calculated value with the update revision in the proposed update header for equality.

When performing the write update function, the BIOS must record the entire update, including the header, the update data, and the extended processor signature table (if applicable). When writing an update, the original contents may be overwritten, assuming the above criteria have been met. It is the responsibility of the BIOS to ensure that more recent updates are not overwritten through the use of this BIOS call, and that only a single update exists within the NVRAM for any processor stepping and platform ID.

Figure 9-8 and Figure 9-9 show the process the BIOS follows to choose an update block and ensure the integrity of the data when it stores the new microcode update.

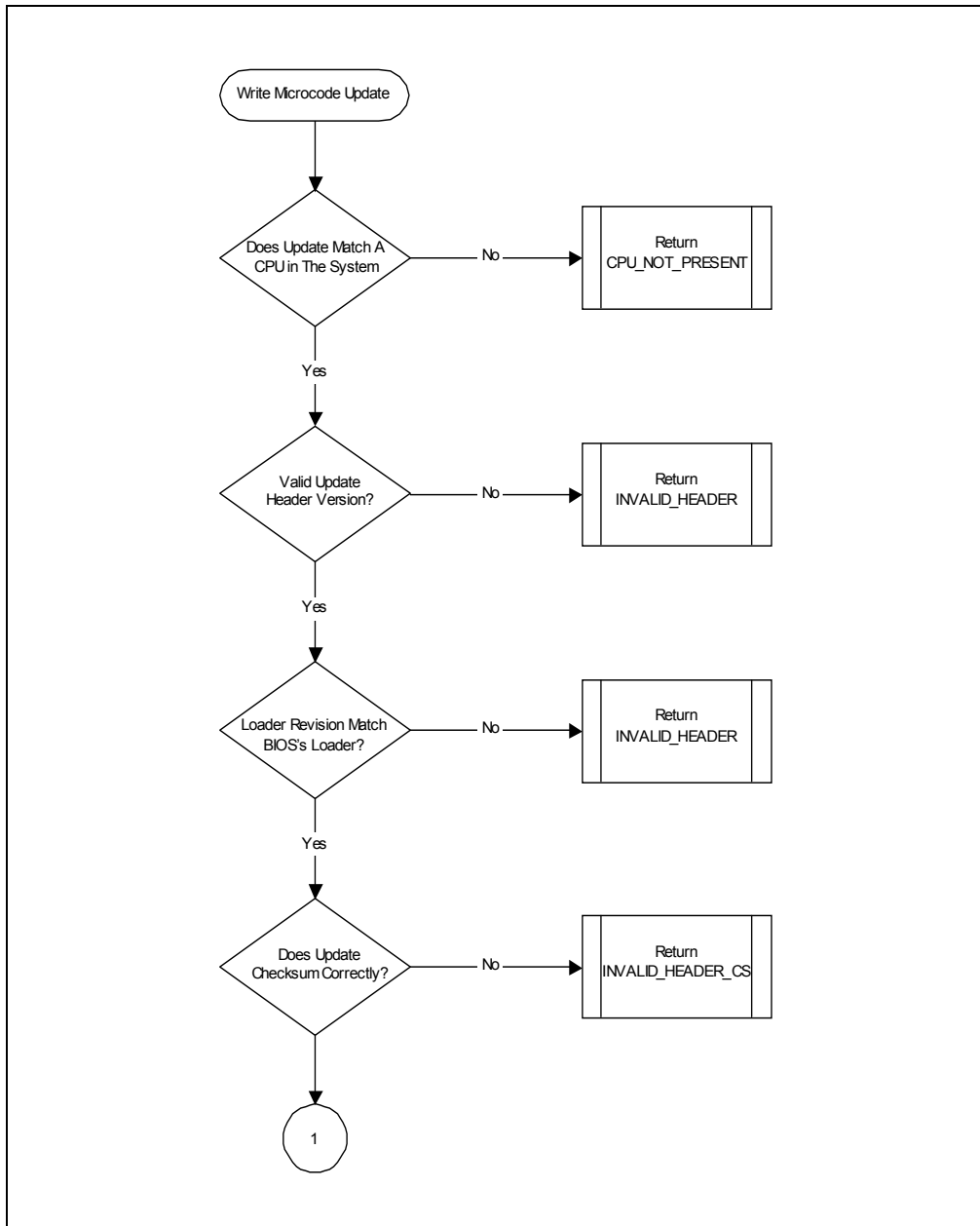


Figure 9-8. Microcode Update Write Operation Flow [1]

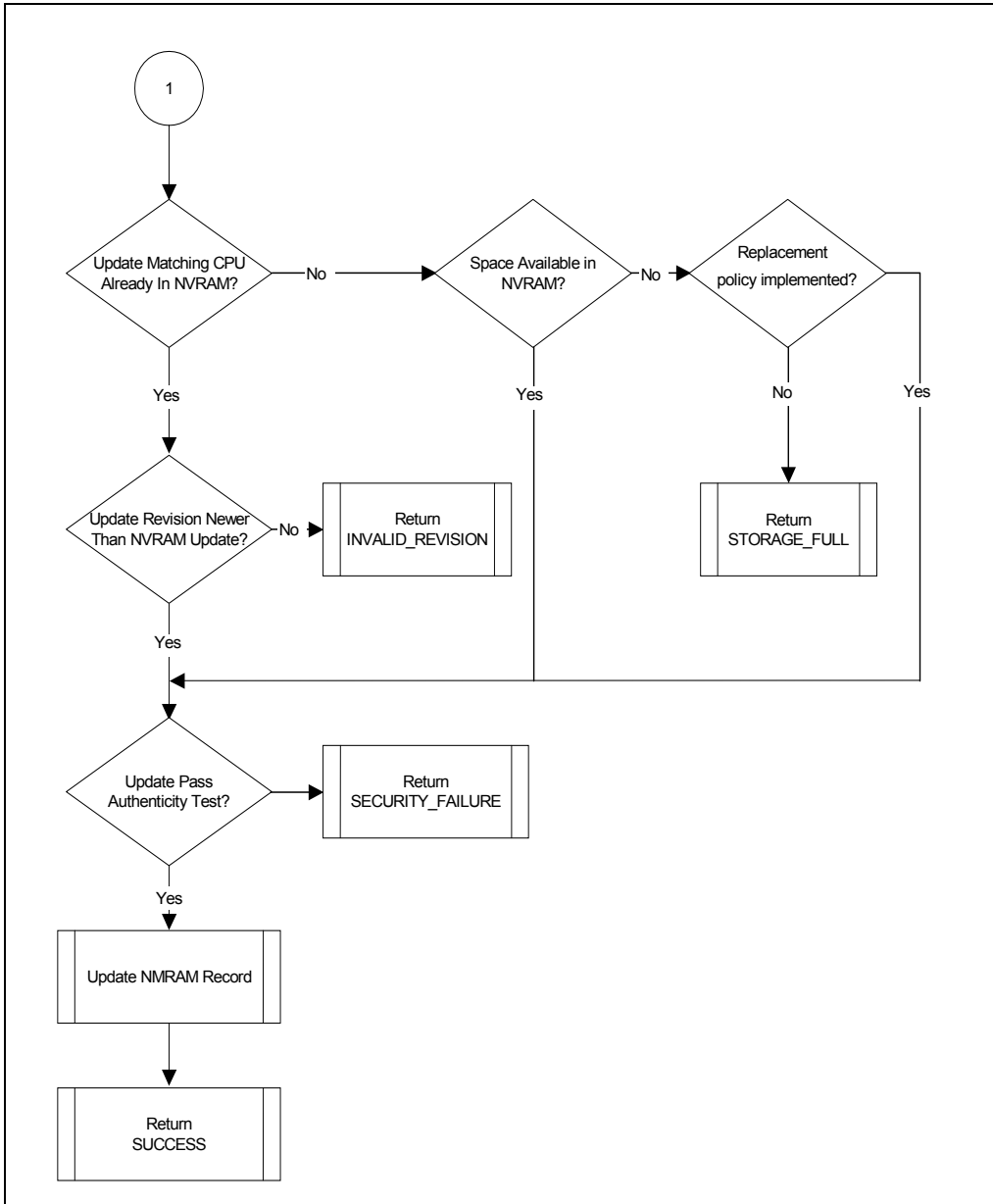


Figure 9-9. Microcode Update Write Operation Flow [2]

### 9.11.8.7 Function 02H—Microcode Update Control

This function enables loading of binary updates into the processor. Table 9-16 lists the parameters and return codes for the function.



**Table 9-16. Parameters for the Control Update Sub-function**

Input		
AX	Function Code	0D042H
BL	Sub-function	02H - Control update
BH	Task	See the description below.
CX	Scratch Pad1	Real mode segment of 64 KBytes of RAM block
DX	Scratch Pad2	Real mode segment of 64 KBytes of RAM block
SI	Scratch Pad3	Real mode segment of 64 KBytes of RAM block
SS:SP	Stack pointer	32 kilobytes of stack minimum
Output		
CF	Carry Flag	Carry Set - Failure - AH contains status Carry Clear - All return values valid.
AH	Return Code	Status of the call
AL	OEM Error	Additional OEM Information.
BL	Update Status	Either enable or disable indicator
Return Codes (see Table 9-19 for code definitions)		
SUCCESS		Function completed successfully.
READ_FAILURE		A failure occurred because of the inability to read the storage device.

This control is provided on a global basis for all updates and processors. The caller can determine the current status of update loading (enabled or disabled) without changing the state. The function does not allow the caller to disable loading of binary updates, as this poses a security risk.

The caller specifies the requested operation by placing one of the values from Table 9-17 in the BH register. After successfully completing this function, the BL register contains either the enable or the disable designator. Note that if the function fails, the update status return value is undefined.

**Table 9-17. Mnemonic Values**

Mnemonic	Value	Meaning
Enable	1	Enable the Update loading at initialization time.
Query	2	Determine the current state of the update control without changing its status.

The READ\_FAILURE error code returned by this function has meaning only if the control function is implemented in the BIOS NVRAM. The state of this feature (enabled/disabled) can also be implemented using CMOS RAM bits where READ failure errors cannot occur.

### 9.11.8.8 Function 03H—Read Microcode Update Data

This function reads a currently installed microcode update from the BIOS storage into a caller-provided RAM buffer. Table 9-18 lists the parameters and return codes.

**Table 9-18. Parameters for the Read Microcode Update Data Function**

Input		
AX	Function Code	0D042H
BL	Sub-function	03H - Read Update
ES:DI	Buffer Address	Real Mode pointer to the Intel Update structure that will be written with the binary data

**Table 9-18. Parameters for the Read Microcode Update Data Function (Contd.)**

ECX	Scratch Pad1	Real Mode Segment address of 64 KBytes of RAM Block (lower 16 bits)
ECX	Scratch Pad2	Real Mode Segment address of 64 KBytes of RAM Block (upper 16 bits)
DX	Scratch Pad3	Real Mode Segment address of 64 KBytes of RAM Block
SS:SP	Stack pointer	32 KBytes of Stack Minimum
SI	Update Number	This is the index number of the update block to be read. This value is zero based and must be less than the update count returned from the presence test function.
<b>Output</b>		
CF	Carry Flag	Carry Set - Failure - AH contains Status
Carry Clear - All return values are valid.		
AH	Return Code	Status of the Call
AL	OEM Error	Additional OEM Information
<b>Return Codes (see Table 9-19 for code definitions)</b>		
SUCCESS		The function completed successfully.
READ_FAILURE		There was a failure because of the inability to read the storage device.
UPDATE_NUM_INVALID		Update number exceeds the maximum number of update blocks implemented by the BIOS.
NOT_EMPTY		The specified update block is a subsequent block in use to store a valid microcode update that spans multiple blocks.  The specified block is not a header block and is not empty.

The read function enables the caller to read any microcode update data that already exists in a BIOS and make decisions about the addition of new updates. As a result of a successful call, the BIOS copies the microcode update into the location pointed to by ES:DI, with the contents of all Update block(s) that are used to store the specified microcode update.

If the specified block is not a header block, but does contain valid data from a microcode update that spans multiple update blocks, then the BIOS must return Failure with the NOT\_EMPTY error code in AH.

An update block is considered unused and available for storing a new update if its Header Version contains the value 0FFFFFFFH after return from this function call. The actual implementation of NVRAM storage management is not specified here and is BIOS dependent. As an example, the actual data value used to represent an empty block by the BIOS may be zero, rather than 0FFFFFFFH. The BIOS is responsible for translating this information into the header provided by this function.

### 9.11.8.9 Return Codes

After the call has been made, the return codes listed in Table 9-19 are available in the AH register.

Table 9-19. Return Code Definitions

Return Code	Value	Description
SUCCESS	00H	The function completed successfully.
NOT_IMPLEMENTED	86H	The function is not implemented.
ERASE_FAILURE	90H	A failure because of the inability to erase the storage device.
WRITE_FAILURE	91H	A failure because of the inability to write the storage device.
READ_FAILURE	92H	A failure because of the inability to read the storage device.
STORAGE_FULL	93H	The BIOS non-volatile storage area is unable to accommodate the update because all available update blocks are filled with updates that are needed for processors in the system.
CPU_NOT_PRESENT	94H	The processor stepping does not currently exist in the system.
INVALID_HEADER	95H	The update header contains a header or loader version that is not recognized by the BIOS.
INVALID_HEADER_CS	96H	The update does not checksum correctly.
SECURITY_FAILURE	97H	The update was rejected by the processor.
INVALID_REVISION	98H	The same or more recent revision of the update exists in the storage device.
UPDATE_NUM_INVALID	99H	The update number exceeds the maximum number of update blocks implemented by the BIOS.
NOT_EMPTY	9AH	The specified update block is a subsequent block in use to store a valid microcode update that spans multiple blocks.  The specified block is not a header block and is not empty.



## 15. Updates to Chapter 18, Volume 3B

Change bars show changes to Chapter 18 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B: System Programming Guide, Part 2*.

-----  
Changes include a new performance monitoring section on Intel® Xeon Phi™ processors.

Intel 64 and IA-32 architectures provide facilities for monitoring performance via a PMU (Performance Monitoring Unit).

### 18.1 PERFORMANCE MONITORING OVERVIEW

Performance monitoring was introduced in the Pentium processor with a set of model-specific performance-monitoring counter MSRs. These counters permit selection of processor performance parameters to be monitored and measured. The information obtained from these counters can be used for tuning system and compiler performance.

In Intel P6 family of processors, the performance monitoring mechanism was enhanced to permit a wider selection of events to be monitored and to allow greater control events to be monitored. Next, Intel processors based on Intel NetBurst microarchitecture introduced a distributed style of performance monitoring mechanism and performance events.

The performance monitoring mechanisms and performance events defined for the Pentium, P6 family, and Intel processors based on Intel NetBurst microarchitecture are not architectural. They are all model specific (not compatible among processor families). Intel Core Solo and Intel Core Duo processors support a set of architectural performance events and a set of non-architectural performance events. Newer Intel processor generations support enhanced architectural performance events and non-architectural performance events.

Starting with Intel Core Solo and Intel Core Duo processors, there are two classes of performance monitoring capabilities. The first class supports events for monitoring performance using counting or interrupt-based event sampling usage. These events are non-architectural and vary from one processor model to another. They are similar to those available in Pentium M processors. These non-architectural performance monitoring events are specific to the microarchitecture and may change with enhancements. They are discussed in Section 18.3, "Performance Monitoring (Intel® Core™ Solo and Intel® Core™ Duo Processors)." Non-architectural events for a given microarchitecture can not be enumerated using CPUID; and they are listed in Chapter 19, "Performance-Monitoring Events."

The second class of performance monitoring capabilities is referred to as architectural performance monitoring. This class supports the same counting and interrupt-based event sampling usages, with a smaller set of available events. The visible behavior of architectural performance events is consistent across processor implementations. Availability of architectural performance monitoring capabilities is enumerated using the CPUID.0AH. These events are discussed in Section 18.2.

See also:

- Section 18.2, "Architectural Performance Monitoring"
- Section 18.3, "Performance Monitoring (Intel® Core™ Solo and Intel® Core™ Duo Processors)"
- Section 18.4, "Performance Monitoring (Processors Based on Intel® Core™ Microarchitecture)"
- Section 18.5, "Performance Monitoring (45 nm and 32 nm Intel® Atom™ Processors)"
- Section 18.6, "Performance Monitoring for Silvermont Microarchitecture"
- Section 18.7, "Performance Monitoring for Goldmont Microarchitecture"
- Section 18.8, "Performance Monitoring for Processors Based on Intel® Microarchitecture Code Name Nehalem"
- Section 18.8.4, "Performance Monitoring for Processors Based on Intel® Microarchitecture Code Name Westmere"
- Section 18.9, "Performance Monitoring for Processors Based on Intel® Microarchitecture Code Name Sandy Bridge"
- Section 18.9.8, "Intel® Xeon® Processor E5 Family Uncore Performance Monitoring Facility"

- Section 18.10, “3rd Generation Intel® Core™ Processor Performance Monitoring Facility”
- Section 18.11, “4th Generation Intel® Core™ Processor Performance Monitoring Facility”
- Section 18.12, “Intel® Core™ M Processor Performance Monitoring Facility”
- Section 18.13, “6th Generation Intel® Core™ Processor Performance Monitoring Facility”
- Section 18.15, “Performance Monitoring (Processors Based on Intel NetBurst® Microarchitecture)”
- Section 18.16, “Performance Monitoring and Intel Hyper-Threading Technology in Processors Based on Intel NetBurst® Microarchitecture”
- Section 18.19, “Performance Monitoring and Dual-Core Technology”
- Section 18.20, “Performance Monitoring on 64-bit Intel Xeon Processor MP with Up to 8-MByte L3 Cache”
- Section 18.22, “Performance Monitoring (P6 Family Processor)”
- Section 18.23, “Performance Monitoring (Pentium Processors)”

## 18.2 ARCHITECTURAL PERFORMANCE MONITORING

Performance monitoring events are architectural when they behave consistently across microarchitectures. Intel Core Solo and Intel Core Duo processors introduced architectural performance monitoring. The feature provides a mechanism for software to enumerate performance events and provides configuration and counting facilities for events.

Architectural performance monitoring does allow for enhancement across processor implementations. The CPUID.0AH leaf provides version ID for each enhancement. Intel Core Solo and Intel Core Duo processors support base level functionality identified by version ID of 1. Processors based on Intel Core microarchitecture support, at a minimum, the base level functionality of architectural performance monitoring. Intel Core 2 Duo processor T 7700 and newer processors based on Intel Core microarchitecture support both the base level functionality and enhanced architectural performance monitoring identified by version ID of 2.

45 nm and 32 nm Intel Atom processors and Intel Atom processors based on the Silvermont microarchitecture support the functionality provided by versionID 1, 2, and 3; CPUID.0AH:EAX[7:0] reports versionID = 3 to indicate the aggregate of architectural performance monitoring capabilities. Intel Atom processors based on the Airmont microarchitecture support the same performance monitoring capabilities as those based on the Silvermont microarchitecture.

Intel Core processors and related Intel Xeon processor families based on the Nehalem through Broadwell microarchitectures support version ID 1, 2, and 3. Intel processors based on the Skylake microarchitecture support versionID 4.

Next generation Intel Atom processors is based on the Goldmont microarchitecture. Intel processors based on the Goldmont microarchitecture support versionID 4.

### 18.2.1 Architectural Performance Monitoring Version 1

Configuring an architectural performance monitoring event involves programming performance event select registers. There are a finite number of performance event select MSRs (IA32\_PERFEVTSELx MSRs). The result of a performance monitoring event is reported in a performance monitoring counter (IA32\_PMCx MSR). Performance monitoring counters are paired with performance monitoring select registers.

Performance monitoring select registers and counters are architectural in the following respects:

- Bit field layout of IA32\_PERFEVTSELx is consistent across microarchitectures.
- Addresses of IA32\_PERFEVTSELx MSRs remain the same across microarchitectures.
- Addresses of IA32\_PMC MSRs remain the same across microarchitectures.
- Each logical processor has its own set of IA32\_PERFEVTSELx and IA32\_PMCx MSRs. Configuration facilities and counters are not shared between logical processors sharing a processor core.

Architectural performance monitoring provides a CPUID mechanism for enumerating the following information:

- Number of performance monitoring counters available in a logical processor (each IA32\_PERFEVTSELx MSR is paired to the corresponding IA32\_PMCx MSR)
- Number of bits supported in each IA32\_PMCx
- Number of architectural performance monitoring events supported in a logical processor

Software can use CPUID to discover architectural performance monitoring availability (CPUID.0AH). The architectural performance monitoring leaf provides an identifier corresponding to the version number of architectural performance monitoring available in the processor.

The version identifier is retrieved by querying CPUID.0AH:EAX[bits 7:0] (see Chapter 3, “Instruction Set Reference, A-L,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A*). If the version identifier is greater than zero, architectural performance monitoring capability is supported. Software queries the CPUID.0AH for the version identifier first; it then analyzes the value returned in CPUID.0AH.EAX, CPUID.0AH.EBX to determine the facilities available.

In the initial implementation of architectural performance monitoring; software can determine how many IA32\_PERFEVTSELx/ IA32\_PMCx MSR pairs are supported per core, the bit-width of PMC, and the number of architectural performance monitoring events available.

### 18.2.1.1 Architectural Performance Monitoring Version 1 Facilities

Architectural performance monitoring facilities include a set of performance monitoring counters and performance event select registers. These MSRs have the following properties:

- IA32\_PMCx MSRs start at address 0C1H and occupy a contiguous block of MSR address space; the number of MSRs per logical processor is reported using CPUID.0AH:EAX[15:8].
- IA32\_PERFEVTSELx MSRs start at address 186H and occupy a contiguous block of MSR address space. Each performance event select register is paired with a corresponding performance counter in the 0C1H address block.
- The bit width of an IA32\_PMCx MSR is reported using the CPUID.0AH:EAX[23:16]. This the number of valid bits for read operation. On write operations, the lower-order 32 bits of the MSR may be written with any value, and the high-order bits are sign-extended from the value of bit 31.
- Bit field layout of IA32\_PERFEVTSELx MSRs is defined architecturally.

See Figure 18-1 for the bit field layout of IA32\_PERFEVTSELx MSRs. The bit fields are:

- **Event select field (bits 0 through 7)** — Selects the event logic unit used to detect microarchitectural conditions (see Table 18-1, for a list of architectural events and their 8-bit codes). The set of values for this field is defined architecturally; each value corresponds to an event logic unit for use with an architectural performance event. The number of architectural events is queried using CPUID.0AH:EAX. A processor may support only a subset of pre-defined values.

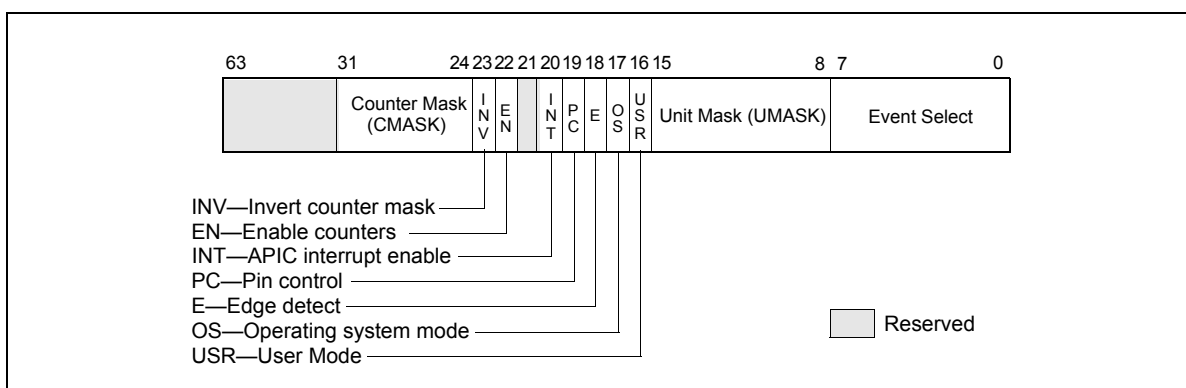


Figure 18-1. Layout of IA32\_PERFEVTSELx MSRs



- Unit mask (UMASK) field (bits 8 through 15)** — These bits qualify the condition that the selected event logic unit detects. Valid UMASK values for each event logic unit are specific to the unit. For each architectural performance event, its corresponding UMASK value defines a specific microarchitectural condition.

A pre-defined microarchitectural condition associated with an architectural event may not be applicable to a given processor. The processor then reports only a subset of pre-defined architectural events. Pre-defined architectural events are listed in Table 18-1; support for pre-defined architectural events is enumerated using CPUID.0AH:EBX. Architectural performance events available in the initial implementation are listed in Table 19-1.
- USR (user mode) flag (bit 16)** — Specifies that the selected microarchitectural condition is counted when the logical processor is operating at privilege levels 1, 2 or 3. This flag can be used with the OS flag.
- OS (operating system mode) flag (bit 17)** — Specifies that the selected microarchitectural condition is counted when the logical processor is operating at privilege level 0. This flag can be used with the USR flag.
- E (edge detect) flag (bit 18)** — Enables (when set) edge detection of the selected microarchitectural condition. The logical processor counts the number of deasserted to asserted transitions for any condition that can be expressed by the other fields. The mechanism does not permit back-to-back assertions to be distinguished.

This mechanism allows software to measure not only the fraction of time spent in a particular state, but also the average length of time spent in such a state (for example, the time spent waiting for an interrupt to be serviced).
- PC (pin control) flag (bit 19)** — When set, the logical processor toggles the PM*i* pins and increments the counter when performance-monitoring events occur; when clear, the processor toggles the PM*i* pins when the counter overflows. The toggling of a pin is defined as assertion of the pin for a single bus clock followed by deassertion.
- INT (APIC interrupt enable) flag (bit 20)** — When set, the logical processor generates an exception through its local APIC on counter overflow.
- EN (Enable Counters) Flag (bit 22)** — When set, performance counting is enabled in the corresponding performance-monitoring counter; when clear, the corresponding counter is disabled. The event logic unit for a UMASK must be disabled by setting IA32\_PERFEVTSELx[bit 22] = 0, before writing to IA32\_PMCx.
- INV (invert) flag (bit 23)** — When set, inverts the counter-mask (CMASK) comparison, so that both greater than or equal to and less than comparisons can be made (0: greater than or equal; 1: less than). Note if counter-mask is programmed to zero, INV flag is ignored.
- Counter mask (CMASK) field (bits 24 through 31)** — When this field is not zero, a logical processor compares this mask to the events count of the detected microarchitectural condition during a single cycle. If the event count is greater than or equal to this mask, the counter is incremented by one. Otherwise the counter is not incremented.

This mask is intended for software to characterize microarchitectural conditions that can count multiple occurrences per cycle (for example, two or more instructions retired per clock; or bus queue occupations). If the counter-mask field is 0, then the counter is incremented each cycle by the event count associated with multiple occurrences.

### 18.2.1.2 Pre-defined Architectural Performance Events

Table 18-1 lists architecturally defined events.

**Table 18-1. UMask and Event Select Encodings for Pre-Defined Architectural Performance Events**

Bit Position CPUID.AH.EBX	Event Name	UMask	Event Select
0	UnHalted Core Cycles	00H	3CH
1	Instruction Retired	00H	C0H
2	UnHalted Reference Cycles	01H	3CH
3	LLC Reference	4FH	2EH
4	LLC Misses	41H	2EH

**Table 18-1. UMask and Event Select Encodings for Pre-Defined Architectural Performance Events**

5	Branch Instruction Retired	00H	C4H
6	Branch Misses Retired	00H	C5H

A processor that supports architectural performance monitoring may not support all the predefined architectural performance events (Table 18-1). The non-zero bits in CPUID.0AH:EBX indicate the events that are not available.

The behavior of each architectural performance event is expected to be consistent on all processors that support that event. Minor variations between microarchitectures are noted below:

- **UnHalted Core Cycles** — Event select 3CH, Umask 00H

This event counts core clock cycles when the clock signal on a specific core is running (not halted). The counter does not advance in the following conditions:

- an ACPI C-state other than C0 for normal operation
- HLT
- STPCLK# pin asserted
- being throttled by TM1
- during the frequency switching phase of a performance state transition (see Chapter 14, “Power and Thermal Management”)

The performance counter for this event counts across performance state transitions using different core clock frequencies

- **Instructions Retired** — Event select C0H, Umask 00H

This event counts the number of instructions at retirement. For instructions that consist of multiple micro-ops, this event counts the retirement of the last micro-op of the instruction. An instruction with a REP prefix counts as one instruction (not per iteration). Faults before the retirement of the last micro-op of a multi-ops instruction are not counted.

This event does not increment under VM-exit conditions. Counters continue counting during hardware interrupts, traps, and inside interrupt handlers.

- **UnHalted Reference Cycles** — Event select 3CH, Umask 01H

This event counts reference clock cycles while the clock signal on the core is running. The reference clock operates at a fixed frequency, irrespective of core frequency changes due to performance state transitions. Processors may implement this behavior differently. See Table 19-23 and Table 19-27 in Chapter 19, “Performance-Monitoring Events.”

- **Last Level Cache References** — Event select 2EH, Umask 4FH

This event counts requests originating from the core that reference a cache line in the last level cache. The event count includes speculation and cache line fills due to the first-level cache hardware prefetcher, but may exclude cache line fills due to other hardware-prefetchers.

Because cache hierarchy, cache sizes and other implementation-specific characteristics; value comparison to estimate performance differences is not recommended.

- **Last Level Cache Misses** — Event select 2EH, Umask 41H

This event counts each cache miss condition for references to the last level cache. The event count may include speculation and cache line fills due to the first-level cache hardware prefetcher, but may exclude cache line fills due to other hardware-prefetchers.

Because cache hierarchy, cache sizes and other implementation-specific characteristics; value comparison to estimate performance differences is not recommended.

- **Branch Instructions Retired** — Event select C4H, Umask 00H

This event counts branch instructions at retirement. It counts the retirement of the last micro-op of a branch instruction.

- **All Branch Mispredict Retired** — Event select C5H, Umask 00H

This event counts mispredicted branch instructions at retirement. It counts the retirement of the last micro-op of a branch instruction in the architectural path of execution and experienced misprediction in the branch prediction hardware.

Branch prediction hardware is implementation-specific across microarchitectures; value comparison to estimate performance differences is not recommended.

### NOTE

Programming decisions or software precisions on functionality should not be based on the event values or dependent on the existence of performance monitoring events.

## 18.2.2 Architectural Performance Monitoring Version 2

The enhanced features provided by architectural performance monitoring version 2 include the following:

- **Fixed-function performance counter register and associated control register** — Three of the architectural performance events are counted using three fixed-function MSRs (IA32\_FIXED\_CTR0 through IA32\_FIXED\_CTR2). Each of the fixed-function PMC can count only one architectural performance event. Configuring the fixed-function PMCs is done by writing to bit fields in the MSR (IA32\_FIXED\_CTR\_CTRL) located at address 38DH. Unlike configuring performance events for general-purpose PMCs (IA32\_PMCx) via UMASK field in (IA32\_PERFEVTSELx), configuring, programming IA32\_FIXED\_CTR\_CTRL for fixed-function PMCs do not require any UMASK.
- **Simplified event programming** — Most frequent operation in programming performance events are enabling/disabling event counting and checking the status of counter overflows. Architectural performance event version 2 provides three architectural MSRs:
  - IA32\_PERF\_GLOBAL\_CTRL allows software to enable/disable event counting of all or any combination of fixed-function PMCs (IA32\_FIXED\_CTRx) or any general-purpose PMCs via a single WRMSR.
  - IA32\_PERF\_GLOBAL\_STATUS allows software to query counter overflow conditions on any combination of fixed-function PMCs or general-purpose PMCs via a single RDMSR.
  - IA32\_PERF\_GLOBAL\_OVF\_CTRL allows software to clear counter overflow conditions on any combination of fixed-function PMCs or general-purpose PMCs via a single WRMSR.
- **PMI Overhead Mitigation** — Architectural performance monitoring version 2 introduces two bit field interface in IA32\_DEBUGCTL for PMI service routine to accumulate performance monitoring data and LBR records with reduced perturbation from servicing the PMI. The two bit fields are:
  - IA32\_DEBUGCTL.Freeze\_LBR\_On\_PMI(bit 11). In architectural performance monitoring version 2, only the legacy semantic behavior is supported. See Section 17.4.7 for details of the legacy Freeze LBRs on PMI control.
  - IA32\_DEBUGCTL.Freeze\_PerfMon\_On\_PMI(bit 12). In architectural performance monitoring version 2, only the legacy semantic behavior is supported. See Section 17.4.7 for details of the legacy Freeze LBRs on PMI control.

The facilities provided by architectural performance monitoring version 2 can be queried from CPUID leaf 0AH by examining the content of register EDX:

- Bits 0 through 4 of CPUID.0AH.EDX indicates the number of fixed-function performance counters available per core,
- Bits 5 through 12 of CPUID.0AH.EDX indicates the bit-width of fixed-function performance counters. Bits beyond the width of the fixed-function counter are reserved and must be written as zeros.

### NOTE

Early generation of processors based on Intel Core microarchitecture may report in CPUID.0AH:EDX of support for version 2 but indicating incorrect information of version 2 facilities.

The IA32\_FIXED\_CTR\_CTRL MSR include multiple sets of 4-bit field, each 4 bit field controls the operation of a fixed-function performance counter. Figure 18-2 shows the layout of 4-bit controls for each fixed-function PMC. Two sub-fields are currently defined within each control. The definitions of the bit fields are:

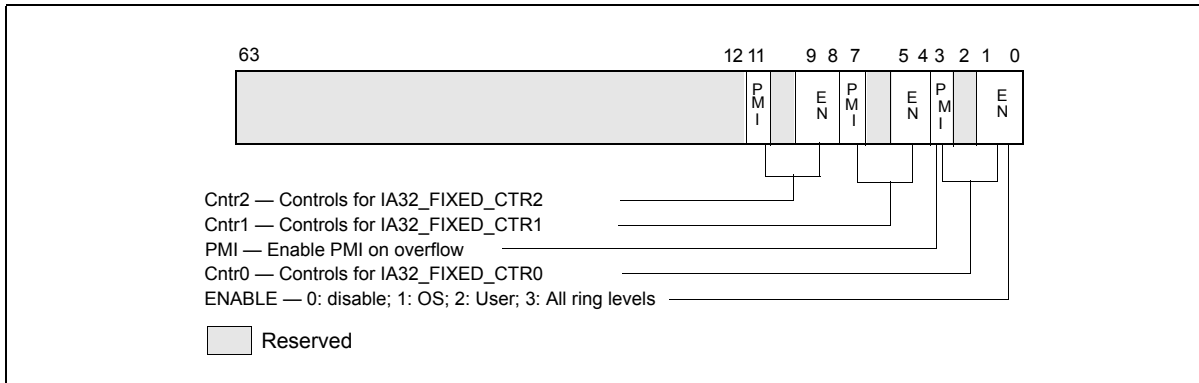


Figure 18-2. Layout of IA32\_FIXED\_CTR\_CTRL MSR

- Enable field (lowest 2 bits within each 4-bit control)** — When bit 0 is set, performance counting is enabled in the corresponding fixed-function performance counter to increment while the target condition associated with the architecture performance event occurred at ring 0. When bit 1 is set, performance counting is enabled in the corresponding fixed-function performance counter to increment while the target condition associated with the architecture performance event occurred at ring greater than 0. Writing 0 to both bits stops the performance counter. Writing a value of 11B enables the counter to increment irrespective of privilege levels.
- PMI field (the fourth bit within each 4-bit control)** — When set, the logical processor generates an exception through its local APIC on overflow condition of the respective fixed-function counter.

IA32\_PERF\_GLOBAL\_CTRL MSR provides single-bit controls to enable counting of each performance counter. Figure 18-3 shows the layout of IA32\_PERF\_GLOBAL\_CTRL. Each enable bit in IA32\_PERF\_GLOBAL\_CTRL is AND’ed with the enable bits for all privilege levels in the respective IA32\_PERFEVTSELx or IA32\_PERF\_FIXED\_CTR\_CTRL MSRs to start/stop the counting of respective counters. Counting is enabled if the AND’ed results is true; counting is disabled when the result is false.

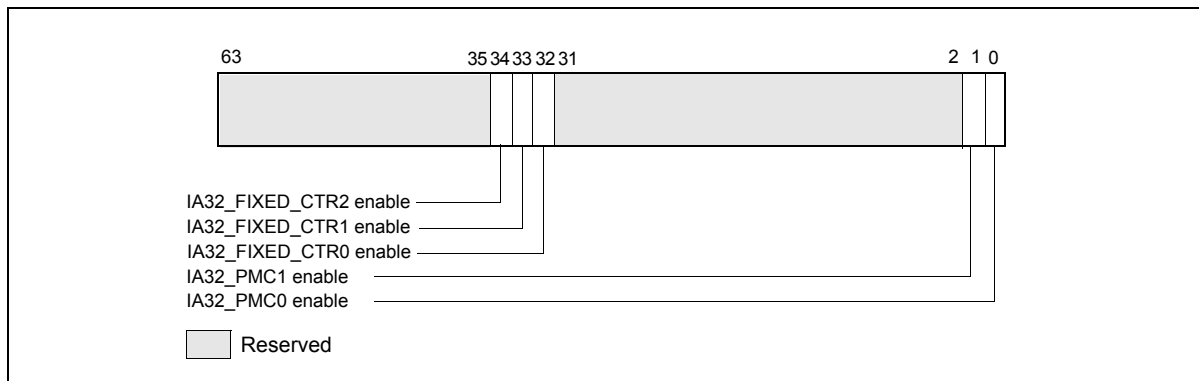


Figure 18-3. Layout of IA32\_PERF\_GLOBAL\_CTRL MSR

The fixed-function performance counters supported by architectural performance version 2 is listed in Table 18-8, the pairing between each fixed-function performance counter to an architectural performance event is also shown.

IA32\_PERF\_GLOBAL\_STATUS MSR provides single-bit status for software to query the overflow condition of each performance counter. IA32\_PERF\_GLOBAL\_STATUS[bit 62] indicates overflow conditions of the DS area data buffer. IA32\_PERF\_GLOBAL\_STATUS[bit 63] provides a CondChgd bit to indicate changes to the state of performance monitoring hardware. Figure 18-4 shows the layout of IA32\_PERF\_GLOBAL\_STATUS. A value of 1 in bits 0, 1, 32 through 34 indicates a counter overflow condition has occurred in the associated counter.

When a performance counter is configured for PEBS, overflow condition in the counter generates a performance-monitoring interrupt signaling a PEBS event. On a PEBS event, the processor stores data records into the buffer area (see Section 18.15.5), clears the counter overflow status, and sets the “OvfBuffer” bit in IA32\_PERF\_GLOBAL\_STATUS.

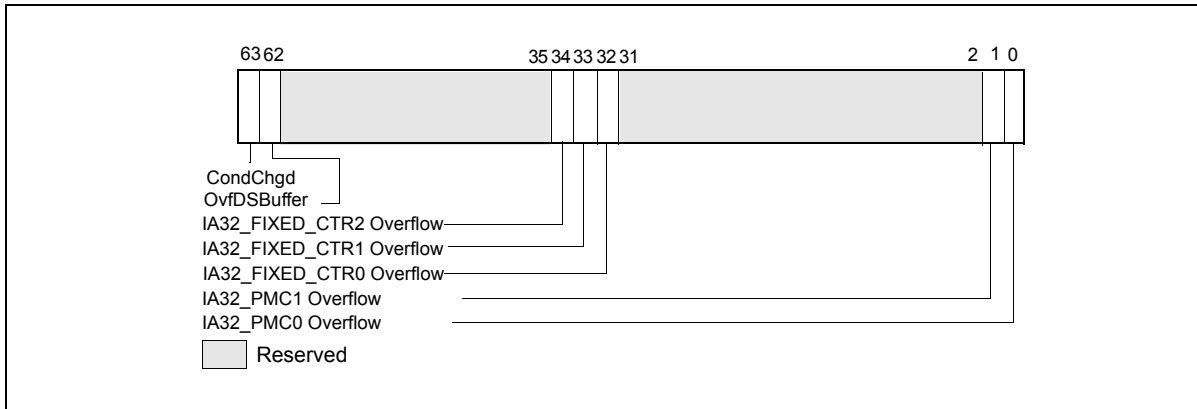


Figure 18-4. Layout of IA32\_PERF\_GLOBAL\_STATUS MSR

IA32\_PERF\_GLOBAL\_OVF\_CTL MSR allows software to clear overflow indicator(s) of any general-purpose or fixed-function counters via a single WRMSR. Software should clear overflow indications when

- Setting up new values in the event select and/or UMASK field for counting or interrupt-based event sampling.
- Reloading counter values to continue collecting next sample.
- Disabling event counting or interrupt-based event sampling.

The layout of IA32\_PERF\_GLOBAL\_OVF\_CTL is shown in Figure 18-5.

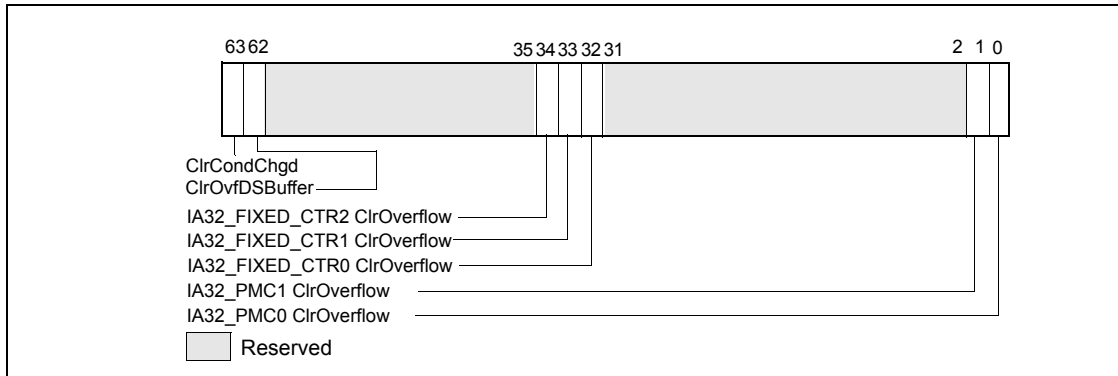


Figure 18-5. Layout of IA32\_PERF\_GLOBAL\_OVF\_CTRL MSR

### 18.2.3 Architectural Performance Monitoring Version 3

Processors supporting architectural performance monitoring version 3 also supports version 1 and 2, as well as capability enumerated by CPUID leaf 0AH. Specifically, version 3 provides the following enhancement in performance monitoring facilities if a processor core comprising of more than one logical processor, i.e. a processor core supporting Intel Hyper-Threading Technology or simultaneous multi-threading capability:

- Anythread counting for processor core supporting two or more logical processors. The interface that supports AnyThread counting include:
  - Each IA32\_PERFEVTSELx MSR (starting at MSR address 186H) support the bit field layout defined in Figure 18-6.

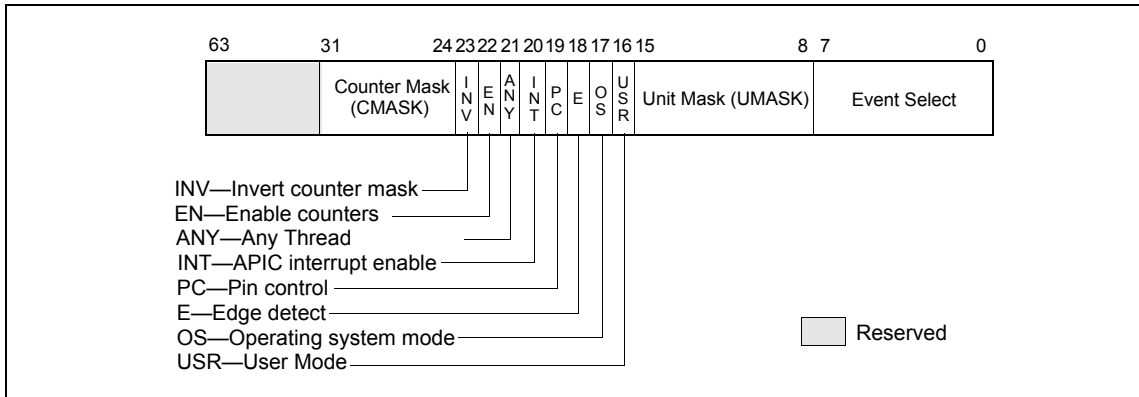


Figure 18-6. Layout of IA32\_PERFEVTSELx MSRs Supporting Architectural Performance Monitoring Version 3

**Bit 21 (AnyThread)** of IA32\_PERFEVTSELx is supported in architectural performance monitoring version 3 for processor core comprising of two or more logical processors. When set to 1, it enables counting the associated event conditions (including matching the thread's CPL with the OS/USR setting of IA32\_PERFEVTSELx) occurring across all logical processors sharing a processor core. When bit 21 is 0, the counter only increments the associated event conditions (including matching the thread's CPL with the OS/USR setting of IA32\_PERFEVTSELx) occurring in the logical processor which programmed the IA32\_PERFEVTSELx MSR.

- Each fixed-function performance counter IA32\_FIXED\_CTRx (starting at MSR address 309H) is configured by a 4-bit control block in the IA32\_PERF\_FIXED\_CTR\_CTRL MSR. The control block also allow thread-specificity configuration using an AnyThread bit. The layout of IA32\_PERF\_FIXED\_CTR\_CTRL MSR is shown.

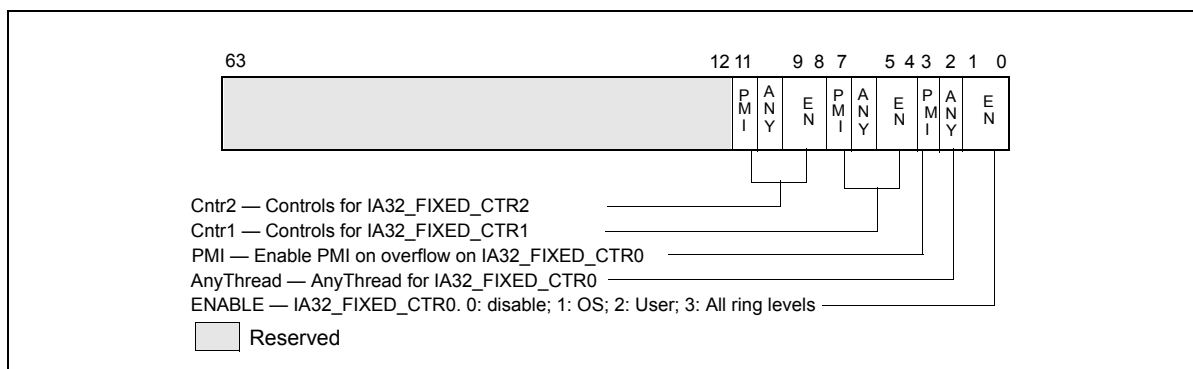


Figure 18-7. IA32\_FIXED\_CTR\_CTRL MSR Supporting Architectural Performance Monitoring Version 3

Each control block for a fixed-function performance counter provides a **AnyThread** (bit position  $2 + 4*N$ ,  $N = 0, 1, \text{etc.}$ ) bit. When set to 1, it enables counting the associated event conditions (including matching the thread's CPL with the ENABLE setting of the corresponding control block of IA32\_PERF\_FIXED\_CTR\_CTRL) occurring across all logical processors sharing a processor core. When an **AnyThread** bit is 0 in IA32\_PERF\_FIXED\_CTR\_CTRL, the corresponding fixed counter only increments the associated event conditions occurring in the logical processor which programmed the IA32\_PERF\_FIXED\_CTR\_CTRL MSR.

- The IA32\_PERF\_GLOBAL\_CTRL, IA32\_PERF\_GLOBAL\_STATUS, IA32\_PERF\_GLOBAL\_OVF\_CTRL MSRs provide single-bit controls/status for each general-purpose and fixed-function performance counter. Figure 18-8 and Figure 18-9 show the layout of these MSRs for N general-purpose performance counters (where N is reported by CPUID.0AH:EAX[15:8]) and three fixed-function counters.

**Note:** The number of general-purpose performance monitoring counters (i.e. N in Figure 18-9) can vary across processor generations within a processor family, across processor families, or could be different depending on the configuration chosen at boot time in the BIOS regarding Intel Hyper Threading Technology, (e.g.  $N=2$  for 45 nm Intel Atom processors;  $N=4$  for processors based on the Nehalem microarchitecture; for processors

based on the Sandy Bridge microarchitecture, N = 4 if Intel Hyper Threading Technology is active and N=8 if not active).

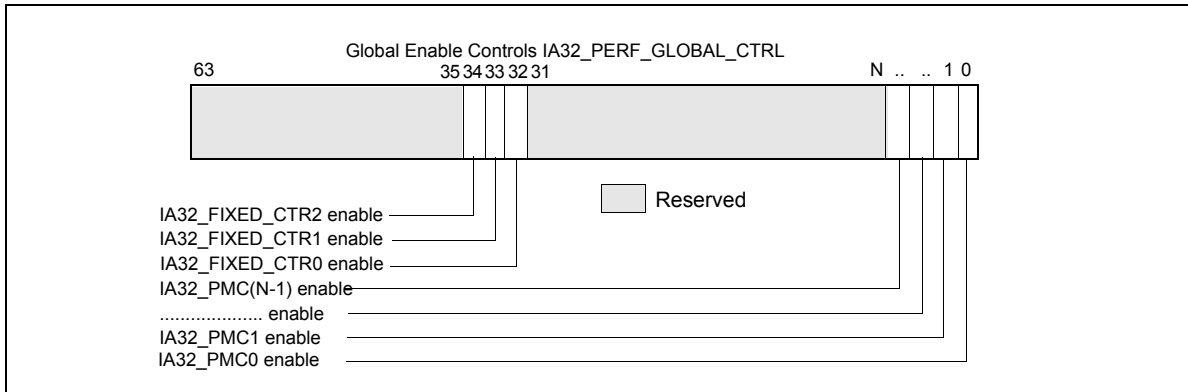


Figure 18-8. Layout of Global Performance Monitoring Control MSR

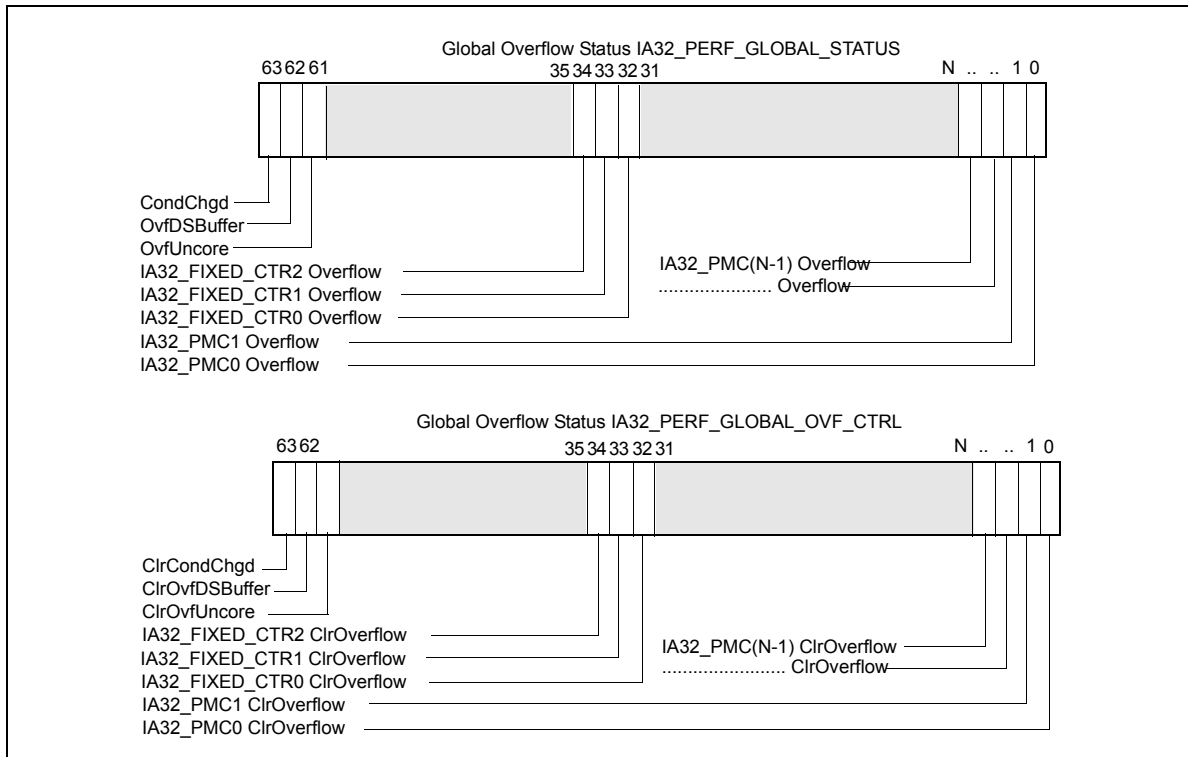


Figure 18-9. Global Performance Monitoring Overflow Status and Control MSRs

### 18.2.3.1 AnyThread Counting and Software Evolution

The motivation for characterizing software workload over multiple software threads running on multiple logical processors of the same processor core originates from a time earlier than the introduction of the AnyThread interface in IA32\_PERFEVTSELx and IA32\_FIXED\_CTR\_CTRL. While Anythread counting provides some benefits in simple software environments of an earlier era, the evolution contemporary software environments introduce certain concepts and pre-requisites that AnyThread counting does not comply with.



One example is the proliferation of software environments that support multiple virtual machines (VM) under VMX (see Chapter 23, “Introduction to Virtual-Machine Extensions”) where each VM represents a domain separated from one another.

A Virtual Machine Monitor (VMM) that manages the VMs may allow individual VM to employ performance monitoring facilities to profile the performance characteristics of a workload. The use of the AnyThread interface in IA32\_PERFEVTSELx and IA32\_FIXED\_CTR\_CTRL is discouraged with software environments supporting virtualization or requiring domain separation.

Specifically, Intel recommends VMM:

- configure the MSR bitmap to cause VM-exits for WRMSR to IA32\_PERFEVTSELx and IA32\_FIXED\_CTR\_CTRL in VMX non-Root operation (see CHAPTER 24 for additional information),
- clear the AnyThread bit of IA32\_PERFEVTSELx and IA32\_FIXED\_CTR\_CTRL in the MSR-load lists for VM exits and VM entries (see CHAPTER 24, CHAPTER 26, and CHAPTER 27).

Even when operating in simpler legacy software environments which might not emphasize the pre-requisites of a virtualized software environment, the use of the AnyThread interface should be moderated and follow any event-specific guidance where explicitly noted (see relevant sections of Chapter 19, “Performance Monitoring Events”).

## 18.2.4 Architectural Performance Monitoring Version 4

Processors supporting architectural performance monitoring version 4 also supports version 1, 2, and 3, as well as capability enumerated by CPUID leaf 0AH. Version 4 introduced a streamlined PMI overhead mitigation interface that replaces the legacy semantic behavior but retains the same control interface in IA32\_DEBUGCTL.Freeze\_LBRs\_On\_PMI and Freeze\_PerfMon\_On\_PMI. Specifically version 4 provides the following enhancement:

- New indicators (LBR\_FRZ, CTR\_FRZ) in IA32\_PERF\_GLOBAL\_STATUS, see Section 18.2.4.1.
- Streamlined Freeze/PMI Overhead management interfaces to use IA32\_DEBUGCTL.Freeze\_LBRs\_On\_PMI and IA32\_DEBUGCTL.Freeze\_PerfMon\_On\_PMI: see Section 18.2.4.1. Legacy semantics of Freeze\_LBRs\_On\_PMI and Freeze\_PerfMon\_On\_PMI (applicable to version 2 and 3) are not supported with version 4 or higher.
- Fine-grain separation of control interface to manage overflow/status of IA32\_PERF\_GLOBAL\_STATUS and read-only performance counter enabling interface in IA32\_PERF\_GLOBAL\_STATUS: see Section 18.2.4.2.
- Performance monitoring resource in-use MSR to facilitate cooperative sharing protocol between perfmon-managing privilege agents.

### 18.2.4.1 Enhancement in IA32\_PERF\_GLOBAL\_STATUS

The IA32\_PERF\_GLOBAL\_STATUS MSR provides the following indicators with architectural performance monitoring version 4:

- IA32\_PERF\_GLOBAL\_STATUS.LBR\_FRZ[bit 58]: This bit is set due to the following conditions:
  - IA32\_DEBUGCTL.FREEZE\_LBR\_ON\_PMI has been set by the profiling agent, and
  - A performance counter, configured to generate PMI, has overflowed to signal a PMI. Consequently the LBR stack is frozen.

Effectively, the IA32\_PERF\_GLOBAL\_STATUS.LBR\_FRZ bit also serve as an read-only control to enable capturing data in the LBR stack. To enable capturing LBR records, the following expression must hold with architectural perfmon version 4 or higher:

–  $(\text{IA32\_DEBUGCTL.LBR} \ \& \ (!\text{IA32\_PERF\_GLOBAL\_STATUS.LBR\_FRZ})) = 1$

- IA32\_PERF\_GLOBAL\_STATUS.CTR\_FRZ[bit 59]: This bit is set due to the following conditions:
  - IA32\_DEBUGCTL.FREEZE\_PERFMON\_ON\_PMI has been set by the profiling agent, and
  - A performance counter, configured to generate PMI, has overflowed to signal a PMI. Consequently, all the performance counters are frozen.



Effectively, the IA32\_PERF\_GLOBAL\_STATUS.CTR\_FRZ bit also serve as an read-only control to enable programmable performance counters and fixed counters in the core PMU. To enable counting with the performance counters, the following expression must hold with architectural perfmon version 4 or higher:

- $(IA32\_PERFEVTSELn.EN \ \& \ IA32\_PERF\_GLOBAL\_CTRL.PMCn \ \& \ (!IA32\_PERF\_GLOBAL\_STATUS.CTR\_FRZ)) = 1$  for programmable counter 'n', or
- $(IA32\_PERF\_FIXED\_CTRL.ENi \ \& \ IA32\_PERF\_GLOBAL\_CTRL.FCi \ \& \ (!IA32\_PERF\_GLOBAL\_STATUS.CTR\_FRZ)) = 1$  for fixed counter 'i'

The read-only enable interface IA32\_PERF\_GLOBAL\_STATUS.CTR\_FRZ provides a more efficient flow for a PMI handler to use IA32\_DEBUGCTL.Freeza\_Perfmon\_On\_PMI to filter out data that may distort target workload analysis, see Table 17-3. It should be noted the IA32\_PERF\_GLOBAL\_CTRL register continue to serve as the primary interface to control all performance counters of the logical processor.

For example, when the Freeze-On-PMI mode is not being used, a PMI handler would be setting IA32\_PERF\_GLOBAL\_CTRL as the very last step to commence the overall operation after configuring the individual counter registers, controls and PEBS facility. This does not only assure atomic monitoring but also avoids unnecessary complications (e.g. race conditions) when software attempts to change the core PMU configuration while some counters are kept enabled.

Additionally, IA32\_PERF\_GLOBAL\_STATUS.TraceToPAPMI[bit 55]: On processors that support Intel Processor Trace and configured to store trace output packets to physical memory using the ToPA scheme, bit 55 is set when a PMI occurred due to a ToPA entry memory buffer was completely filled.

IA32\_PERF\_GLOBAL\_STATUS also provides an indicator to distinguish interaction of performance monitoring operations with other side-band activities, which apply Intel SGX on processors that support SGX (For additional information about Intel SGX, see "Intel® Software Guard Extensions Programming Reference".):

- IA32\_PERF\_GLOBAL\_STATUS.ASCI[bit 60]: This bit is set when data accumulated in any of the configured performance counters (i.e. IA32\_PMCx or IA32\_FIXED\_CTRx) may include contributions from direct or indirect operation of Intel SGX to protect an enclave (since the last time IA32\_PERF\_GLOBAL\_STATUS.ASCI was cleared).

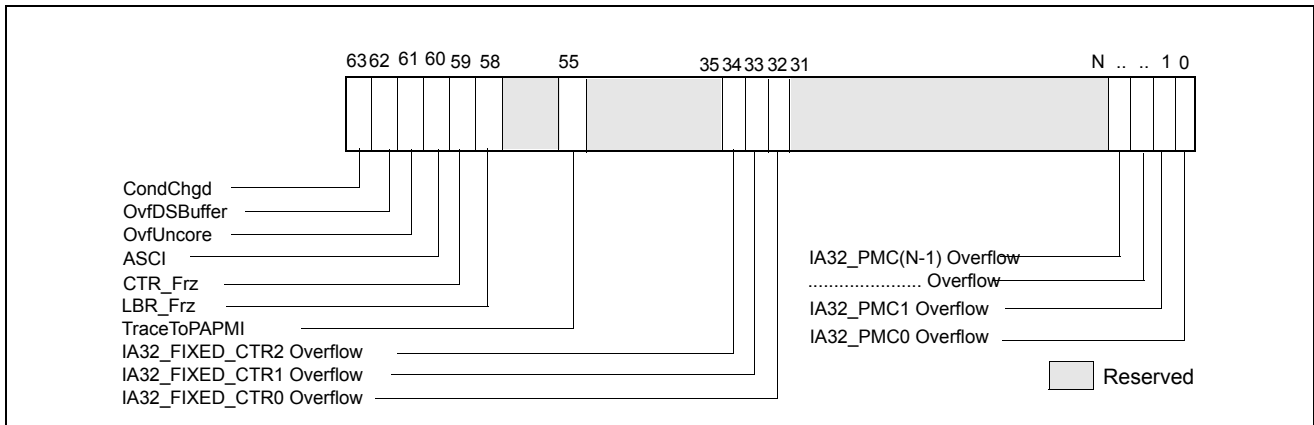


Figure 18-10. IA32\_PERF\_GLOBAL\_STATUS MSR and Architectural Perfmon Version 4

Note, a processor’s support for IA32\_PERF\_GLOBAL\_STATUS.TraceToPAPMI[bit 55] is enumerated as a result of CPUID enumerated capability of Intel Processor Trace and the use of the ToPA buffer scheme. Support of IA32\_PERF\_GLOBAL\_STATUS.ASCI[bit 60] is enumerated by the CPUID enumeration of Intel SGX.

### 18.2.4.2 IA32\_PERF\_GLOBAL\_STATUS\_RESET and IA32\_PERF\_GLOBAL\_STATUS\_SET MSRS

With architectural performance monitoring version 3 and lower, clearing of the set bits in IA32\_PERF\_GLOBAL\_STATUS MSR by software is done via IA32\_PERF\_GLOBAL\_OVF\_CTRL MSR. Starting with architectural performance monitoring version 4, software can manage the overflow and other indicators in IA32\_PERF\_GLOBAL\_STATUS using separate interfaces to set or clear individual bits.

The address and the architecturally-defined bits of IA32\_PERF\_GLOBAL\_OVF\_CTRL is inherited by IA32\_PERF\_GLOBAL\_STATUS\_RESET (see Figure 18-11). Further, IA32\_PERF\_GLOBAL\_STATUS\_RESET provides additional bit fields to clear the new indicators in IA32\_PERF\_GLOBAL\_STATUS described in Section 18.2.4.1.

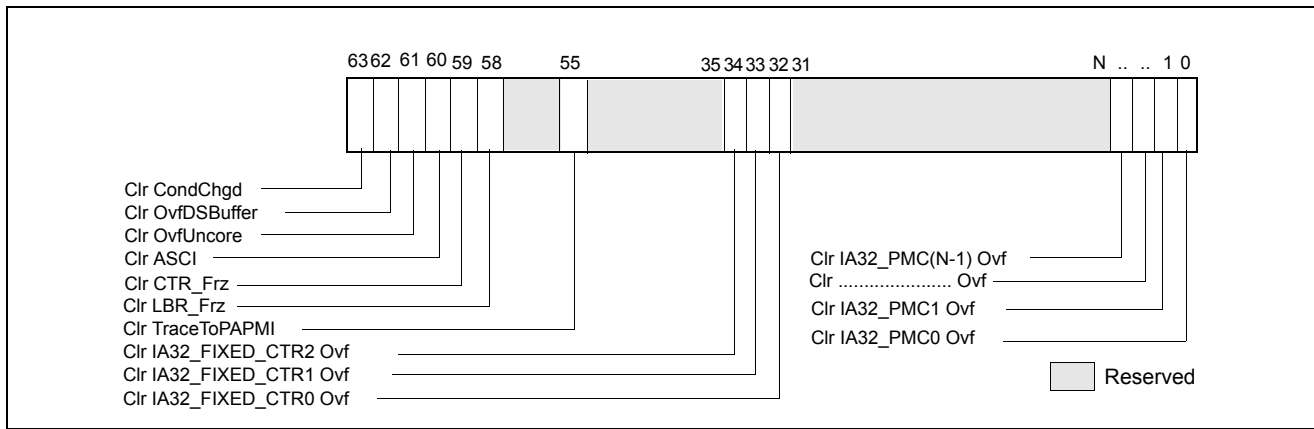


Figure 18-11. IA32\_PERF\_GLOBAL\_STATUS\_RESET MSR and Architectural Perfmon Version 4

The IA32\_PERF\_GLOBAL\_STATUS\_SET MSR is introduced with architectural performance monitoring version 4. It allows software to set individual bits in IA32\_PERF\_GLOBAL\_STATUS. The IA32\_PERF\_GLOBAL\_STATUS\_SET interface can be used by a VMM to virtualize the state of IA32\_PERF\_GLOBAL\_STATUS across VMs.

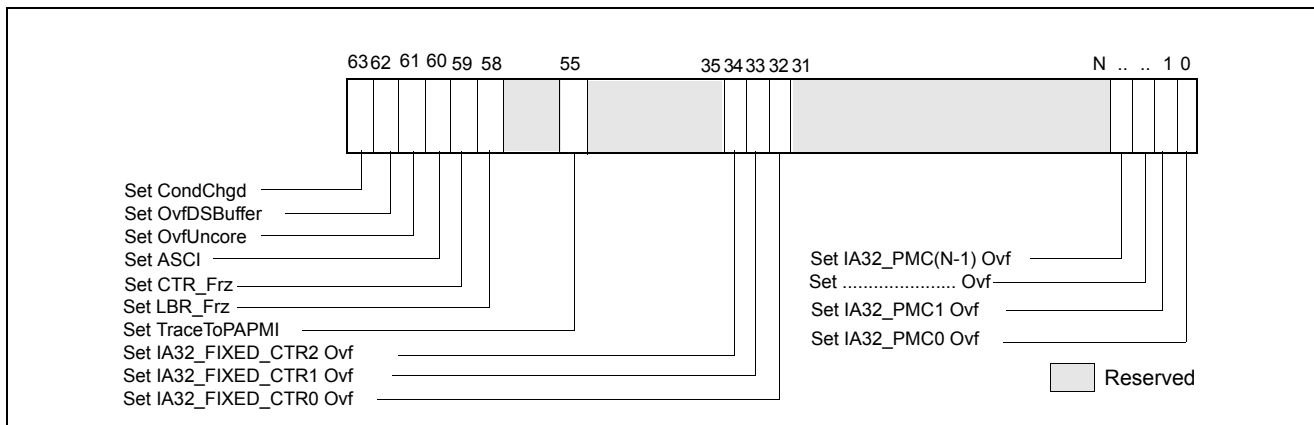


Figure 18-12. IA32\_PERF\_GLOBAL\_STATUS\_SET MSR and Architectural Perfmon Version 4

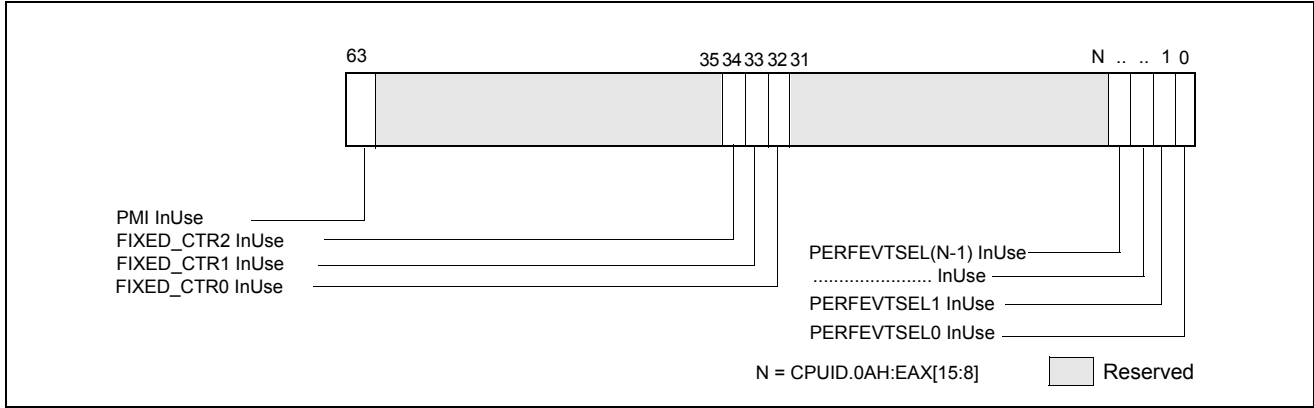
### 18.2.4.3 IA32\_PERF\_GLOBAL\_INUSE MSR

In a contemporary software environment, multiple privileged service agents may wish to employ the processor’s performance monitoring facilities. The IA32\_MISC\_ENABLES.PERFMON\_AVAILABLE[bit 7] interface could not serve the need of multiple agent adequately. A white paper, “Performance Monitoring Unit Sharing Guideline”<sup>1</sup>, proposed a cooperative sharing protocol that is voluntary for participating software agents.

Architectural performance monitoring version 4 introduces a new MSR, IA32\_PERF\_GLOBAL\_INUSE, that simplifies the task of multiple cooperating agents to implement the sharing protocol.

The layout of IA32\_PERF\_GLOBAL\_INUSE is shown in Figure 18-13.

1. Available at <http://www.intel.com/sdm>



**Figure 18-13. IA32\_PERF\_GLOBAL\_INUSE MSR and Architectural Perfmon Version 4**

The IA32\_PERF\_GLOBAL\_INUSE MSR provides an “InUse” bit for each programmable performance counter and fixed counter in the processor. Additionally, it includes an indicator if the PMI mechanism has been configured by a profiling agent.

- IA32\_PERF\_GLOBAL\_INUSE.PERFEVTSEL0\_InUse[bit 0]: This bit reflects the logical state of (IA32\_PERFEVTSEL0[7:0] != 0).
- IA32\_PERF\_GLOBAL\_INUSE.PERFEVTSEL1\_InUse[bit 1]: This bit reflects the logical state of (IA32\_PERFEVTSEL1[7:0] != 0).
- IA32\_PERF\_GLOBAL\_INUSE.PERFEVTSEL2\_InUse[bit 2]: This bit reflects the logical state of (IA32\_PERFEVTSEL2[7:0] != 0).
- IA32\_PERF\_GLOBAL\_INUSE.PERFEVTSELn\_InUse[bit n]: This bit reflects the logical state of (IA32\_PERFEVTSELn[7:0] != 0), n < CPUID.0AH:EAX[15:8].
- IA32\_PERF\_GLOBAL\_INUSE.FC0\_InUse[bit 32]: This bit reflects the logical state of (IA32\_FIXED\_CTR\_CTRL[1:0] != 0).
- IA32\_PERF\_GLOBAL\_INUSE.FC1\_InUse[bit 33]: This bit reflects the logical state of (IA32\_FIXED\_CTR\_CTRL[5:4] != 0).
- IA32\_PERF\_GLOBAL\_INUSE.FC2\_InUse[bit 34]: This bit reflects the logical state of (IA32\_FIXED\_CTR\_CTRL[9:8] != 0).
- IA32\_PERF\_GLOBAL\_INUSE.PMI\_InUse[bit 63]: This bit is set if any one of the following bit is set:
  - IA32\_PERFEVTSELn.INT[bit 20], n < CPUID.0AH:EAX[15:8];
  - IA32\_FIXED\_CTR\_CTRL.ENi\_PMI, i = 0, 1, 2;
  - IA32\_PEBES\_ENABLES.EN\_PMCi, i = 0, 1, 2, 3

### 18.2.5 Full-Width Writes to Performance Counter Registers

The general-purpose performance counter registers IA32\_PMCx are writable via WRMSR instruction. However, the value written into IA32\_PMCx by WRMSR is the signed extended 64-bit value of the EAX[31:0] input of WRMSR.

A processor that supports full-width writes to the general-purpose performance counters enumerated by CPUID.0AH:EAX[15:8] will set IA32\_PERF\_CAPABILITIES[13] to enumerate its full-width-write capability See Figure 18-49.

If IA32\_PERF\_CAPABILITIES.FW\_WRITE[bit 13] = 1, each IA32\_PMCi is accompanied by a corresponding alias address starting at 4C1H for IA32\_A\_PMC0.

The bit width of the performance monitoring counters is specified in CPUID.0AH:EAX[23:16].

If IA32\_A\_PMCi is present, the 64-bit input value (EDX:EAX) of WRMSR to IA32\_A\_PMCi will cause IA32\_PMCi to be updated by:

```
COUNTERWIDTH = CPUID.0AH:EAX[23:16] bit width of the performance monitoring counter
IA32_PMCi[COUNTERWIDTH-1:32] ← EDX[COUNTERWIDTH-33:0];
IA32_PMCi[31:0] ← EAX[31:0];
EDX[63:COUNTERWIDTH] are reserved
```

## 18.3 PERFORMANCE MONITORING (INTEL® CORE™ SOLO AND INTEL® CORE™ DUO PROCESSORS)

In Intel Core Solo and Intel Core Duo processors, non-architectural performance monitoring events are programmed using the same facilities (see Figure 18-1) used for architectural performance events.

Non-architectural performance events use event select values that are model-specific. Event mask (Umask) values are also specific to event logic units. Some microarchitectural conditions detectable by a Umask value may have specificity related to processor topology (see Section 8.6, “Detecting Hardware Multi-Threading Support and Topology,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*). As a result, the unit mask field (for example, IA32\_PERFEVTSELx[bits 15:8]) may contain sub-fields that specify topology information of processor cores.

The sub-field layout within the Umask field may support two-bit encoding that qualifies the relationship between a microarchitectural condition and the originating core. This data is shown in Table 18-2. The two-bit encoding for core-specificity is only supported for a subset of Umask values (see Chapter 19, “Performance Monitoring Events”) and for Intel Core Duo processors. Such events are referred to as core-specific events.

**Table 18-2. Core Specificity Encoding within a Non-Architectural Umask**

IA32_PERFEVTSELx MSRs	
Bit 15:14 Encoding	Description
11B	All cores
10B	Reserved
01B	This core
00B	Reserved

Some microarchitectural conditions allow detection specificity only at the boundary of physical processors. Some bus events belong to this category, providing specificity between the originating physical processor (a bus agent) versus other agents on the bus. Sub-field encoding for agent specificity is shown in Table 18-3.

**Table 18-3. Agent Specificity Encoding within a Non-Architectural Umask**

IA32_PERFEVTSELx MSRs	
Bit 13 Encoding	Description
0	This agent
1	Include all agents

Some microarchitectural conditions are detectable only from the originating core. In such cases, unit mask does not support core-specificity or agent-specificity encodings. These are referred to as core-only conditions.

Some microarchitectural conditions allow detection specificity that includes or excludes the action of hardware prefetches. A two-bit encoding may be supported to qualify hardware prefetch actions. Typically, this applies only to some L2 or bus events. The sub-field encoding for hardware prefetch qualification is shown in Table 18-4.

**Table 18-4. HW Prefetch Qualification Encoding within a Non-Architectural Umask**

IA32_PERFEVTSELx MSRs	
Bit 13:12 Encoding	Description
11B	All inclusive
10B	Reserved
01B	Hardware prefetch only
00B	Exclude hardware prefetch

Some performance events may (a) support none of the three event-specific qualification encodings (b) may support core-specificity and agent specificity simultaneously (c) or may support core-specificity and hardware prefetch qualification simultaneously. Agent-specificity and hardware prefetch qualification are mutually exclusive.

In addition, some L2 events permit qualifications that distinguish cache coherent states. The sub-field definition for cache coherency state qualification is shown in Table 18-5. If no bits in the MESI qualification sub-field are set for an event that requires setting MESI qualification bits, the event count will not increment.

**Table 18-5. MESI Qualification Definitions within a Non-Architectural Umask**

IA32_PERFEVTSELx MSRs	
Bit Position 11:8	Description
Bit 11	Counts modified state
Bit 10	Counts exclusive state
Bit 9	Counts shared state
Bit 8	Counts Invalid state

## 18.4 PERFORMANCE MONITORING (PROCESSORS BASED ON INTEL® CORE™ MICROARCHITECTURE)

In addition to architectural performance monitoring, processors based on the Intel Core microarchitecture support non-architectural performance monitoring events.

Architectural performance events can be collected using general-purpose performance counters. Non-architectural performance events can be collected using general-purpose performance counters (coupled with two IA32\_PERFEVTSELx MSRs for detailed event configurations), or fixed-function performance counters (see Section 18.4.1). IA32\_PERFEVTSELx MSRs are architectural; their layout is shown in Figure 18-1. Starting with Intel Core 2 processor T 7700, fixed-function performance counters and associated counter control and status MSR becomes part of architectural performance monitoring version 2 facilities (see also Section 18.2.2).

Non-architectural performance events in processors based on Intel Core microarchitecture use event select values that are model-specific. Valid event mask (Umask) bits are listed in Chapter 19. The UMASK field may contain sub-fields identical to those listed in Table 18-2, Table 18-3, Table 18-4, and Table 18-5. One or more of these sub-fields may apply to specific events on an event-by-event basis. Details are listed in Table 19-23 in Chapter 19, "Performance-Monitoring Events."

In addition, the UMASK field may also contain a sub-field that allows detection specificity related to snoop responses. Bits of the snoop response qualification sub-field are defined in Table 18-6.

**Table 18-6. Bus Snoop Qualification Definitions within a Non-Architectural Umask**

IA32_PERFEVTSELx MSRs	
Bit Position 11:8	Description
Bit 11	HITM response
Bit 10	Reserved

**Table 18-6. Bus Snoop Qualification Definitions within a Non-Architectural Umask**

IA32_PERFEVTSELx MSRs	
Bit Position 11:8	Description
Bit 9	HIT response
Bit 8	CLEAN response

There are also non-architectural events that support qualification of different types of snoop operation. The corresponding bit field for snoop type qualification are listed in Table 18-7.

**Table 18-7. Snoop Type Qualification Definitions within a Non-Architectural Umask**

IA32_PERFEVTSELx MSRs	
Bit Position 9:8	Description
Bit 9	CMP2I snoops
Bit 8	CMP2S snoops

No more than one sub-field of MESI, snoop response, and snoop type qualification sub-fields can be supported in a performance event.

#### NOTE

Software must write known values to the performance counters prior to enabling the counters. The content of general-purpose counters and fixed-function counters are undefined after INIT or RESET.

### 18.4.1 Fixed-function Performance Counters

Processors based on Intel Core microarchitecture provide three fixed-function performance counters. Bits beyond the width of the fixed counter are reserved and must be written as zeros. Model-specific fixed-function performance counters on processors that support Architectural Perfmon version 1 are 40 bits wide.

Each of the fixed-function counter is dedicated to count a pre-defined performance monitoring events. The performance monitoring events associated with fixed-function counters and the addresses of these counters are listed in Table 18-8.

**Table 18-8. Association of Fixed-Function Performance Counters with Architectural Performance Events**

Event Name	Fixed-Function PMC	PMC Address
INST_RETIRED.ANY	MSR_PERF_FIXED_CTR0//IA32_FIXED_CTR0	309H
CPU_CLK_UNHALTED.CORE	MSR_PERF_FIXED_CTR1//IA32_FIXED_CTR1	30AH
CPU_CLK_UNHALTED.REF	MSR_PERF_FIXED_CTR2//IA32_FIXED_CTR2	30BH

Programming the fixed-function performance counters does not involve any of the IA32\_PERFEVTSELx MSRs, and does not require specifying any event masks. Instead, the MSR MSR\_PERF\_FIXED\_CTR\_CTRL provides multiple sets of 4-bit fields; each 4-bit field controls the operation of a fixed-function performance counter (PMC). See Figures 18-14. Two sub-fields are defined for each control. See Figure 18-14; bit fields are:

- **Enable field (low 2 bits in each 4-bit control)** — When bit 0 is set, performance counting is enabled in the corresponding fixed-function performance counter to increment when the target condition associated with the architecture performance event occurs at ring 0.

When bit 1 is set, performance counting is enabled in the corresponding fixed-function performance counter to increment when the target condition associated with the architecture performance event occurs at ring greater than 0.

Writing 0 to both bits stops the performance counter. Writing 11B causes the counter to increment irrespective of privilege levels.

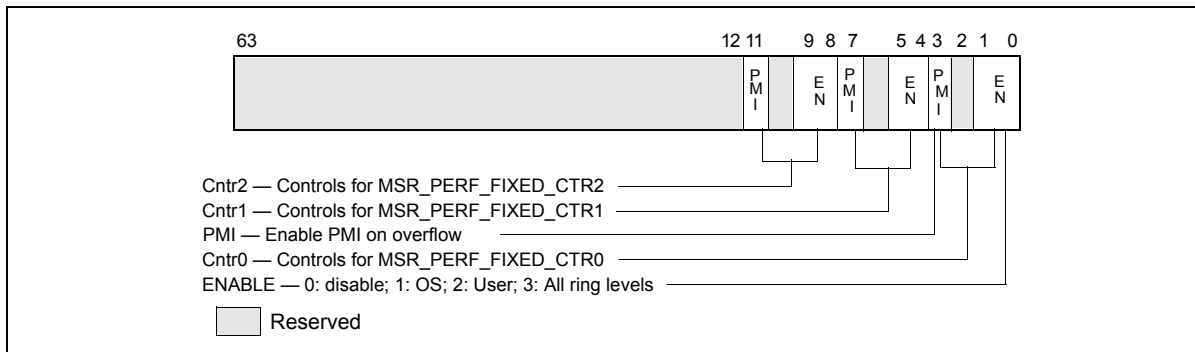


Figure 18-14. Layout of MSR\_PERF\_FIXED\_CTR\_CTRL MSR

- **PMI field (fourth bit in each 4-bit control)** — When set, the logical processor generates an exception through its local APIC on overflow condition of the respective fixed-function counter.

### 18.4.2 Global Counter Control Facilities

Processors based on Intel Core microarchitecture provides simplified performance counter control that simplifies the most frequent operations in programming performance events, i.e. enabling/disabling event counting and checking the status of counter overflows. This is done by the following three MSRs:

- MSR\_PERF\_GLOBAL\_CTRL enables/disables event counting for all or any combination of fixed-function PMCs (MSR\_PERF\_FIXED\_CTRx) or general-purpose PMCs via a single WRMSR.
- MSR\_PERF\_GLOBAL\_STATUS allows software to query counter overflow conditions on any combination of fixed-function PMCs (MSR\_PERF\_FIXED\_CTRx) or general-purpose PMCs via a single RDMSR.
- MSR\_PERF\_GLOBAL\_OVF\_CTRL allows software to clear counter overflow conditions on any combination of fixed-function PMCs (MSR\_PERF\_FIXED\_CTRx) or general-purpose PMCs via a single WRMSR.

MSR\_PERF\_GLOBAL\_CTRL MSR provides single-bit controls to enable counting in each performance counter (see Figure 18-15). Each enable bit in MSR\_PERF\_GLOBAL\_CTRL is AND’ed with the enable bits for all privilege levels in the respective IA32\_PERFEVTSELx or MSR\_PERF\_FIXED\_CTR\_CTRL MSRs to start/stop the counting of respective counters. Counting is enabled if the AND’ed results is true; counting is disabled when the result is false.

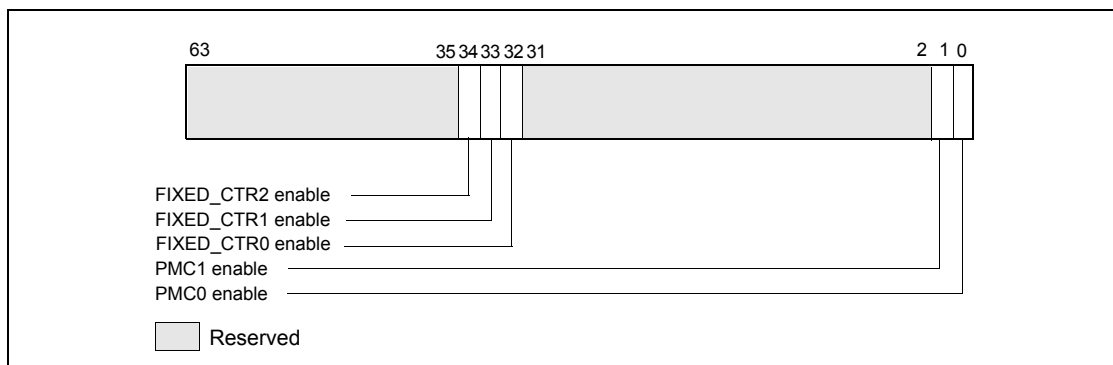


Figure 18-15. Layout of MSR\_PERF\_GLOBAL\_CTRL MSR

MSR\_PERF\_GLOBAL\_STATUS MSR provides single-bit status used by software to query the overflow condition of each performance counter. MSR\_PERF\_GLOBAL\_STATUS[bit 62] indicates overflow conditions of the DS area data buffer. MSR\_PERF\_GLOBAL\_STATUS[bit 63] provides a CondChgd bit to indicate changes to the state of performance monitoring hardware (see Figure 18-16). A value of 1 in bits 34:32, 1, 0 indicates an overflow condition has occurred in the associated counter.

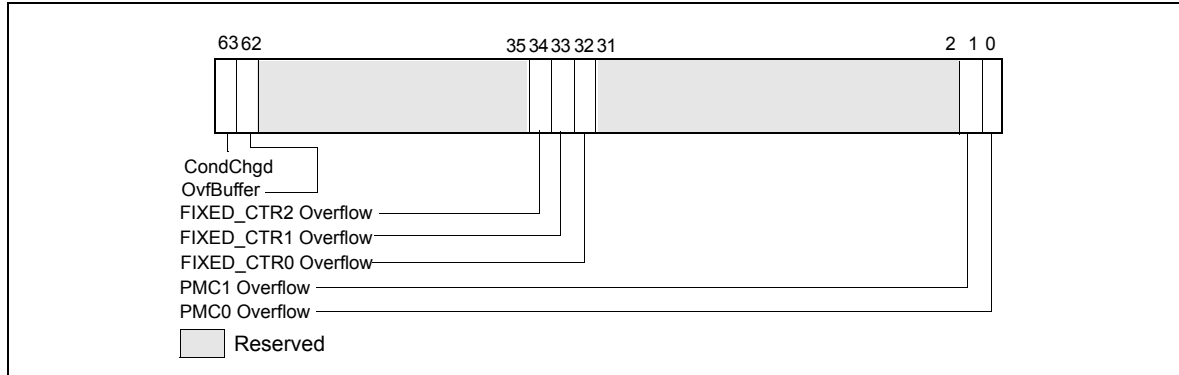


Figure 18-16. Layout of MSR\_PERF\_GLOBAL\_STATUS MSR

When a performance counter is configured for PEBS, an overflow condition in the counter will arm PEBS. On the subsequent event following overflow, the processor will generate a PEBS event. On a PEBS event, the processor will perform bounds checks based on the parameters defined in the DS Save Area (see Section 17.4.9). Upon successful bounds checks, the processor will store the data record in the defined buffer area, clear the counter overflow status, and reload the counter. If the bounds checks fail, the PEBS will be skipped entirely. In the event that the PEBS buffer fills up, the processor will set the OvFBuffer bit in MSR\_PERF\_GLOBAL\_STATUS.

MSR\_PERF\_GLOBAL\_OVF\_CTL MSR allows software to clear overflow the indicators for general-purpose or fixed-function counters via a single WRMSR (see Figure 18-17). Clear overflow indications when:

- Setting up new values in the event select and/or UMASK field for counting or interrupt-based event sampling.
- Reloading counter values to continue collecting next sample.
- Disabling event counting or interrupt-based event sampling.

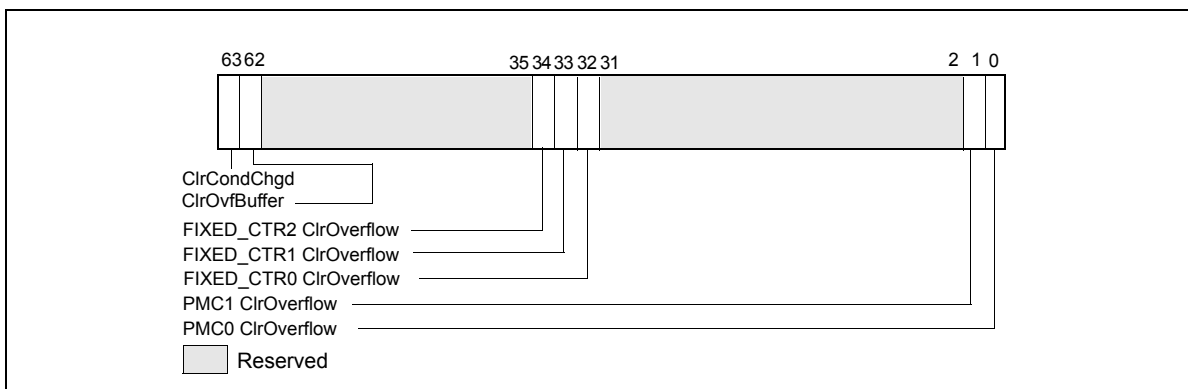


Figure 18-17. Layout of MSR\_PERF\_GLOBAL\_OVF\_CTL MSR



### 18.4.3 At-Retirement Events

Many non-architectural performance events are impacted by the speculative nature of out-of-order execution. A subset of non-architectural performance events on processors based on Intel Core microarchitecture are enhanced with a tagging mechanism (similar to that found in Intel NetBurst<sup>®</sup> microarchitecture) that exclude contributions that arise from speculative execution. The at-retirement events available in processors based on Intel Core microarchitecture does not require special MSR programming control (see Section 18.15.6, “At-Retirement Counting”), but is limited to IA32\_PMC0. See Table 18-9 for a list of events available to processors based on Intel Core microarchitecture.

**Table 18-9. At-Retirement Performance Events for Intel Core Microarchitecture**

Event Name	UMask	Event Select
ITLB_MISS_RETIRED	00H	C9H
MEM_LOAD_RETIRED.L1D_MISS	01H	CBH
MEM_LOAD_RETIRED.L1D_LINE_MISS	02H	CBH
MEM_LOAD_RETIRED.L2_MISS	04H	CBH
MEM_LOAD_RETIRED.L2_LINE_MISS	08H	CBH
MEM_LOAD_RETIRED.DTLB_MISS	10H	CBH

### 18.4.4 Processor Event Based Sampling (PEBS)

Processors based on Intel Core microarchitecture also support processor event based sampling (PEBS). This feature was introduced by processors based on Intel NetBurst microarchitecture.

PEBS uses a debug store mechanism and a performance monitoring interrupt to store a set of architectural state information for the processor. The information provides architectural state of the instruction executed after the instruction that caused the event (See Section 18.4.4.2 and Section 17.4.9).

In cases where the same instruction causes BTS and PEBS to be activated, PEBS is processed before BTS are processed. The PMI request is held until the processor completes processing of PEBS and BTS.

For processors based on Intel Core microarchitecture, precise events that can be used with PEBS are listed in Table 18-10. The procedure for detecting availability of PEBS is the same as described in Section 18.15.7.1.

**Table 18-10. PEBS Performance Events for Intel Core Microarchitecture**

Event Name	UMask	Event Select
INSTR_RETIRED.ANY_P	00H	C0H
X87_OPS_RETIRED.ANY	FEH	C1H
BR_INST_RETIRED.MISPRED	00H	C5H
SIMD_INST_RETIRED.ANY	1FH	C7H
MEM_LOAD_RETIRED.L1D_MISS	01H	CBH
MEM_LOAD_RETIRED.L1D_LINE_MISS	02H	CBH
MEM_LOAD_RETIRED.L2_MISS	04H	CBH
MEM_LOAD_RETIRED.L2_LINE_MISS	08H	CBH
MEM_LOAD_RETIRED.DTLB_MISS	10H	CBH

### 18.4.4.1 Setting up the PEBS Buffer

For processors based on Intel Core microarchitecture, PEBS is available using IA32\_PMC0 only. Use the following procedure to set up the processor and IA32\_PMC0 counter for PEBS:

1. Set up the precise event buffering facilities. Place values in the precise event buffer base, precise event index, precise event absolute maximum, precise event interrupt threshold, and precise event counter reset fields of the DS buffer management area. In processors based on Intel Core microarchitecture, PEBS records consist of 64-bit address entries. See Figure 17-8 to set up the precise event records buffer in memory.
2. Enable PEBS. Set the Enable PEBS on PMC0 flag (bit 0) in IA32\_PEBS\_ENABLE MSR.
3. Set up the IA32\_PMC0 performance counter and IA32\_PERFEVTSEL0 for an event listed in Table 18-10.

### 18.4.4.2 PEBS Record Format

The PEBS record format may be extended across different processor implementations. The IA32\_PERF\_CAPABILITIES MSR defines a mechanism for software to handle the evolution of PEBS record format in processors that support architectural performance monitoring with version id equals 2 or higher. The bit fields of IA32\_PERF\_CAPABILITIES are defined in Table 35-2 of Chapter 35, "Model-Specific Registers (MSRs)". The relevant bit fields that governs PEBS are:

- **PEBSTrap [bit 6]:** When set, PEBS recording is trap-like. After the PEBS-enabled counter has overflowed, PEBS record is recorded for the next PEBS-able event at the completion of the sampled instruction causing the PEBS event. When clear, PEBS recording is fault-like. The PEBS record is recorded before the sampled instruction causing the PEBS event.
- **PEBSSaveArchRegs [bit 7]:** When set, PEBS will save architectural register and state information according to the encoded value of the PEBSRecordFormat field. When clear, only the return instruction pointer and flags are recorded. On processors based on Intel Core microarchitecture, this bit is always 1
- **PEBSRecordFormat [bits 11:8]:** Valid encodings are:
  - 0000B: Only general-purpose registers, instruction pointer and RFLAGS registers are saved in each PEBS record (seeSection 18.15.7).
  - 0001B: PEBS record includes additional information of IA32\_PERF\_GLOBAL\_STATUS and load latency data. (seeSection 18.8.1.1).
  - 0010B: PEBS record includes additional information of IA32\_PERF\_GLOBAL\_STATUS, load latency data, and TSX tuning information. (seeSection 18.11.2).
  - 0011B: PEBS record includes additional information of load latency data, TSX tuning information, TSC data, and the applicable counter field replaces IA32\_PERF\_GLOBAL\_STATUS at offset 90H. (see Section 18.13.1.1).

### 18.4.4.3 Writing a PEBS Interrupt Service Routine

The PEBS facilities share the same interrupt vector and interrupt service routine (called the DS ISR) with the Interrupt-based event sampling and BTS facilities. To handle PEBS interrupts, PEBS handler code must be included in the DS ISR. See Section 17.4.9.1, "64 Bit Format of the DS Save Area," for guidelines when writing the DS ISR.

The service routine can query MSR\_PERF\_GLOBAL\_STATUS to determine which counter(s) caused of overflow condition. The service routine should clear overflow indicator by writing to MSR\_PERF\_GLOBAL\_OVF\_CTL.

A comparison of the sequence of requirements to program PEBS for processors based on Intel Core and Intel NetBurst microarchitectures is listed in Table 18-11.

**Table 18-11. Requirements to Program PEBS**

	For Processors based on Intel Core microarchitecture	For Processors based on Intel NetBurst microarchitecture
Verify PEBS support of processor/OS	<ul style="list-style-type: none"> <li>IA32_MISC_ENABLE.EMON_AVAILABLE (bit 7) is set.</li> <li>IA32_MISC_ENABLE.PEBS_UNAVAILABLE (bit 12) is clear.</li> </ul>	
Ensure counters are in disabled	<p>On initial set up or changing event configurations, write MSR_PERF_GLOBAL_CTRL MSR (38FH) with 0.</p> <p>On subsequent entries:</p> <ul style="list-style-type: none"> <li>Clear all counters if “Counter Freeze on PMI” is not enabled.</li> <li>If IA32_DebugCTL.Freeze is enabled, counters are automatically disabled. Counters MUST be stopped before writing.<sup>1</sup></li> </ul>	Optional
Disable PEBS.	Clear ENABLE PMCO bit in IA32_PEBS_ENABLE MSR (3F1H).	Optional
Check overflow conditions.	Check MSR_PERF_GLOBAL_STATUS MSR (38EH) handle any overflow conditions.	Check OVF flag of each CCCR for overflow condition
Clear overflow status.	Clear MSR_PERF_GLOBAL_STATUS MSR (38EH) using IA32_PERF_GLOBAL_OVF_CTRL MSR (390H).	Clear OVF flag of each CCCR.
Write “sample-after” values.	Configure the counter(s) with the sample after value.	
Configure specific counter configuration MSR.	<ul style="list-style-type: none"> <li>Set local enable bit 22 - 1.</li> <li>Do NOT set local counter PMI/INT bit, bit 20 - 0.</li> <li>Event programmed must be PEBS capable.</li> </ul>	<ul style="list-style-type: none"> <li>Set appropriate OVF_PMI bits - 1.</li> <li>Only CCCR for MSR_IQ_COUNTER4 support PEBS.</li> </ul>
Allocate buffer for PEBS states.	Allocate a buffer in memory for the precise information.	
Program the IA32_DS_AREA MSR.	Program the IA32_DS_AREA MSR.	
Configure the PEBS buffer management records.	Configure the PEBS buffer management records in the DS buffer management area.	
Configure/Enable PEBS.	Set Enable PMCO bit in IA32_PEBS_ENABLE MSR (3F1H).	Configure MSR_PEBS_ENABLE, MSR_PEBS_MATRIX_VERT and MSR_PEBS_MATRIX_HORZ as needed.
Enable counters.	Set Enable bits in MSR_PERF_GLOBAL_CTRL MSR (38FH).	Set each CCCR enable bit 12 - 1.

**NOTES:**

1. Counters read while enabled are not guaranteed to be precise with event counts that occur in timing proximity to the RDMSR.

**18.4.4.4 Re-configuring PEBS Facilities**

When software needs to reconfigure PEBS facilities, it should allow a quiescent period between stopping the prior event counting and setting up a new PEBS event. The quiescent period is to allow any latent residual PEBS records to complete its capture at their previously specified buffer address (provided by IA32\_DS\_AREA).

## 18.5 PERFORMANCE MONITORING (45 NM AND 32 NM INTEL® ATOM™ PROCESSORS)

45 nm and 32 nm Intel Atom processors report architectural performance monitoring versionID = 3 (supporting the aggregate capabilities of versionID 1, 2, and 3; see Section 18.2.3) and a host of non-architectural monitoring capabilities. These 45 nm and 32 nm Intel Atom processors provide two general-purpose performance counters (IA32\_PMC0, IA32\_PMC1) and three fixed-function performance counters (IA32\_FIXED\_CTR0, IA32\_FIXED\_CTR1, IA32\_FIXED\_CTR2).

Non-architectural performance monitoring in Intel Atom processor family uses the IA32\_PERFEVTSELx MSR to configure a set of non-architecture performance monitoring events to be counted by the corresponding general-purpose performance counter. The list of non-architectural performance monitoring events is listed in Table 19-26.

Architectural and non-architectural performance monitoring events in 45 nm and 32 nm Intel Atom processors support thread qualification using bit 21 (AnyThread) of IA32\_PERFEVTSELx MSR, i.e. if IA32\_PERFEVTSELx.AnyThread = 1, event counts include monitored conditions due to either logical processors in the same processor core.

The bit fields within each IA32\_PERFEVTSELx MSR are defined in Figure 18-6 and described in Section 18.2.1.1 and Section 18.2.3.

Valid event mask (Umask) bits are listed in Chapter 19. The UMASK field may contain sub-fields that provide the same qualifying actions like those listed in Table 18-2, Table 18-3, Table 18-4, and Table 18-5. One or more of these sub-fields may apply to specific events on an event-by-event basis. Details are listed in Table 19-26 in Chapter 19, "Performance-Monitoring Events." Precise Event Based Monitoring is supported using IA32\_PMC0 (see also Section 17.4.9, "BTS and DS Save Area").

## 18.6 PERFORMANCE MONITORING FOR SILVERMONT MICROARCHITECTURE

Intel processors based on the Silvermont microarchitecture report architectural performance monitoring versionID = 3 (see Section 18.2.3) and a host of non-architectural monitoring capabilities. Intel processors based on the Silvermont microarchitecture provide two general-purpose performance counters (IA32\_PMC0, IA32\_PMC1) and three fixed-function performance counters (IA32\_FIXED\_CTR0, IA32\_FIXED\_CTR1, IA32\_FIXED\_CTR2). Intel Atom processors based on the Airmont microarchitecture support the same performance monitoring capabilities as those based on the Silvermont microarchitecture.

Non-architectural performance monitoring in the Silvermont microarchitecture uses the IA32\_PERFEVTSELx MSR to configure a set of non-architecture performance monitoring events to be counted by the corresponding general-purpose performance counter. The list of non-architectural performance monitoring events is listed in Table 19-25.

The bit fields (except bit 21) within each IA32\_PERFEVTSELx MSR are defined in Figure 18-6 and described in Section 18.2.1.1 and Section 18.2.3. Architectural and non-architectural performance monitoring events in the Silvermont microarchitecture ignore the AnyThread qualification regardless of its setting in IA32\_PERFEVTSELx MSR.

### 18.6.1 Enhancements of Performance Monitoring in the Processor Core

The notable enhancements in the monitoring of performance events in the processor core include:

- The width of counter reported by CPUID.0AH:EAX[23:16] is 40 bits.
- Off-core response counting facility. This facility in the processor core allows software to count certain transaction responses between the processor core to sub-systems outside the processor core (uncore). Counting off-core response requires additional event qualification configuration facility in conjunction with IA32\_PERFEVTSELx. Two off-core response MSRs are provided to use in conjunction with specific event codes that must be specified with IA32\_PERFEVTSELx.
- Average request latency measurement. The off-core response counting facility can be combined to use two performance counters to count the occurrences and weighted cycles of transaction requests.

### 18.6.1.1 Processor Event Based Sampling (PEBS)

In the Silvermont microarchitecture, the PEBS facility can be used with precise events. PEBS is supported using IA32\_PMC0 (see also Section 17.4.9).

PEBS uses a debug store mechanism to store a set of architectural state information for the processor. The information provides architectural state of the instruction executed after the instruction that caused the event (See Section 18.4.4).

The list of precise events supported in the Silvermont microarchitecture is shown in Table 18-12.

**Table 18-12. PEBS Performance Events for the Silvermont Microarchitecture**

Event Name	Event Select	Sub-event	UMask
BR_INST_RETIRED	C4H	ALL_BRANCHES	00H
		JCC	7EH
		TAKEN_JCC	FEH
		CALL	F9H
		REL_CALL	FDH
		IND_CALL	FBH
		NON_RETURN_IND	EBH
		FAR_BRANCH	BFH
		RETURN	F7H
BR_MISP_RETIRED	C5H	ALL_BRANCHES	00H
		JCC	7EH
		TAKEN_JCC	FEH
		IND_CALL	FBH
		NON_RETURN_IND	EBH
		RETURN	F7H
MEM_UOPS_RETIRED	04H	L2_HIT_LOADS	02H
		L2_MISS_LOADS	04H
		DLTB_MISS_LOADS	08H
		HITM	20H
REHABQ	03H	LD_BLOCK_ST_FORWARD	01H
		LD_SPLITS	08H

PEBS Record Format The PEBS record format supported by processors based on the Intel Silvermont microarchitecture is shown in Table 18-13, and each field in the PEBS record is 64 bits long.

**Table 18-13. PEBS Record Format for the Silvermont Microarchitecture**

Byte Offset	Field	Byte Offset	Field
00H	R/EFLAGS	60H	R10
08H	R/EIP	68H	R11
10H	R/EAX	70H	R12
18H	R/EBX	78H	R13
20H	R/ECX	80H	R14
28H	R/EDX	88H	R15
30H	R/ESI	90H	IA32_PERF_GLOBAL_STATUS
38H	R/EDI	98H	Reserved
40H	R/EBP	A0H	Reserved
48H	R/ESP	A8H	Reserved
50H	R8	B0H	EventingRIP
58H	R9	B8H	Reserved

## 18.6.2 Offcore Response Event

Event number 0B7H support offcore response monitoring using an associated configuration MSR, MSR\_OFFCORE\_RSP0 (address 1A6H) in conjunction with umask value 01H or MSR\_OFFCORE\_RSP1 (address 1A7H) in conjunction with umask value 02H. Table 18-14 lists the event code, mask value and additional off-core configuration MSR that must be programmed to count off-core response events using IA32\_PMCx.

In the Silvermont microarchitecture, each MSR\_OFFCORE\_RSPx is shared by two processor cores.

**Table 18-14. OffCore Response Event Encoding**

Counter	Event code	UMask	Required Off-core Response MSR
PMCO-1	B7H	01H	MSR_OFFCORE_RSP0 (address 1A6H)
PMCO-1	B7H	02H	MSR_OFFCORE_RSP1 (address 1A7H)

The layout of MSR\_OFFCORE\_RSP0 and MSR\_OFFCORE\_RSP1 are shown in Figure 18-18 and Figure 18-19. Bits 15:0 specifies the request type of a transaction request to the uncore. Bits 30:16 specifies supplier information, bits 37:31 specifies snoop response information.

Additionally, MSR\_OFFCORE\_RSP0 provides bit 38 to enable measurement of average latency of specific type of offcore transaction requests using two programmable counter simultaneously, see Section 18.6.3 for details.

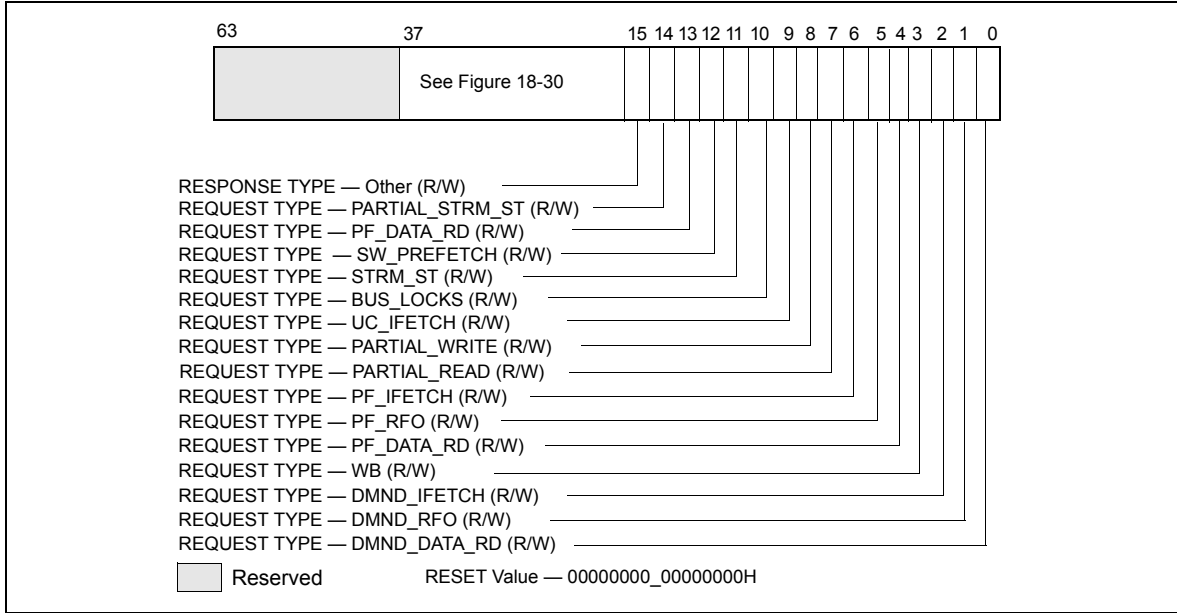


Figure 18-18. Request\_Type Fields for MSR\_OFFCORE\_RSPx

Table 18-15. MSR\_OFFCORE\_RSPx Request\_Type Field Definition

Bit Name	Offset	Description
DMND_DATA_RD	0	(R/W). Counts the number of demand and DCU prefetch data reads of full and partial cachelines as well as demand data page table entry cacheline reads. Does not count L2 data read prefetches or instruction fetches.
DMND_RFO	1	(R/W). Counts the number of demand and DCU prefetch reads for ownership (RFO) requests generated by a write to data cacheline. Does not count L2 RFO prefetches.
DMND_IFETCH	2	(R/W). Counts the number of demand and DCU prefetch instruction cacheline reads. Does not count L2 code read prefetches.
WB	3	(R/W). Counts the number of writeback (modified to exclusive) transactions.
PF_DATA_RD	4	(R/W). Counts the number of data cacheline reads generated by L2 prefetchers.
PF_RFO	5	(R/W). Counts the number of RFO requests generated by L2 prefetchers.
PF_IFETCH	6	(R/W). Counts the number of code reads generated by L2 prefetchers.
PARTIAL_READ	7	(R/W). Counts the number of demand reads of partial cache lines (including UC and WC).
PARTIAL_WRITE	8	(R/W). Counts the number of demand RFO requests to write to partial cache lines (includes UC, WT and WP)
UC_IFETCH	9	(R/W). Counts the number of UC instruction fetches.
BUS_LOCKS	10	(R/W). Bus lock and split lock requests
STRM_ST	11	(R/W). Streaming store requests
SW_PREFETCH	12	(R/W). Counts software prefetch requests
PF_DATA_RD	13	(R/W). Counts DCU hardware prefetcher data read requests
PARTIAL_STRM_ST	14	(R/W). Streaming store requests
ANY	15	(R/W). Any request that crosses IDI, including I/O.

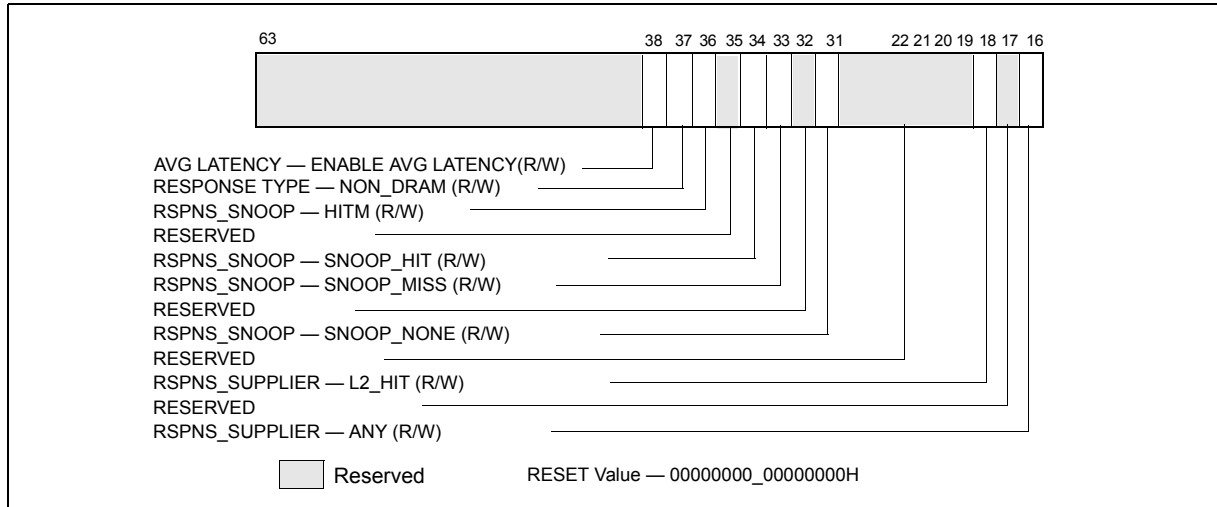


Figure 18-19. Response\_Supplier and Snoop Info Fields for MSR\_OFFCORE\_RSPx

To properly program this extra register, software must set at least one request type bit (Table 18-15) and a valid response type pattern (Table 18-16, Table 18-17). Otherwise, the event count reported will be zero. It is permissible and useful to set multiple request and response type bits in order to obtain various classes of off-core response events. Although MSR\_OFFCORE\_RSPx allow an agent software to program numerous combinations that meet the above guideline, not all combinations produce meaningful data.

Table 18-16. MSR\_OFFCORE\_RSP\_x Response Supplier Info Field Definition

Subtype	Bit Name	Offset	Description
Common	ANY_RESPONSE	16	(R/W). Catch all value for any response types.
Supplier Info	Reserved	17	Reserved
	L2_HIT	18	(R/W). Cache reference hit L2 in either M/E/S states.
	Reserved	30:19	Reserved

To specify a complete offcore response filter, software must properly program bits in the request and response type fields. A valid request type must have at least one bit set in the non-reserved bits of 15:0. A valid response type must be a non-zero value of the following expression:

ANY | [(‘OR’ of Supplier Info Bits) & (‘OR’ of Snoop Info Bits)]

If “ANY” bit is set, the supplier and snoop info bits are ignored.



**Table 18-17. MSR\_OFFCORE\_RSPx Snoop Info Field Definition**

Subtype	Bit Name	Offset	Description
Snoop Info	SNP_NONE	31	(R/W). No details on snoop-related information
	Reserved	32	Reserved
	SNOOP_MISS	33	(R/W). Counts the number of snoop misses when L2 misses
	SNOOP_HIT	34	(R/W). Counts the number of snoops hit in the other module where no modified copies were found
	Reserved	35	Reserved
	HITM	36	(R/W). Counts the number of snoops hit in the other module where modified copies were found in other core's L1 cache.
	NON_DRAM	37	(R/W). Target was non-DRAM system address. This includes MMIO transactions.
	AVG_LATENCY	38	(R/W). Enable average latency measurement by counting weighted cycles of outstanding offcore requests of the request type specified in bits 15:0 and any response (bits 37:16 cleared to 0).  This bit is available in MSR_OFFCORE_RESP0. The weighted cycles is accumulated in the specified programmable counter IA32_PMCx and the occurrence of specified requests are counted in the other programmable counter.

### 18.6.3 Average Offcore Request Latency Measurement

Average latency for offcore transactions can be determined by using both MSR\_OFFCORE\_RSP registers. Using two performance monitoring counters, program the two OFFCORE\_RESPONSE event encodings into the corresponding IA32\_PERFEVTSELx MSRs. Count the weighted cycles via MSR\_OFFCORE\_RSP0 by programming a request type in MSR\_OFFCORE\_RSP0.[15:0] and setting MSR\_OFFCORE\_RSP0.OUTSTANDING[38] to 1, while setting the remaining bits to 0. Count the number of requests via MSR\_OFFCORE\_RSP1 by programming the same request type from MSR\_OFFCORE\_RSP0 into MSR\_OFFCORE\_RSP1[bit 15:0], and setting MSR\_OFFCORE\_RSP1.ANY\_RESPONSE[16] = 1, while setting the remaining bits to 0. The average latency can be obtained by dividing the value of the IA32\_PMCx register that counted weight cycles by the register that counted requests.

## 18.7 PERFORMANCE MONITORING FOR GOLDMONT MICROARCHITECTURE

Next generation Intel Atom processors are based on the Goldmont microarchitecture. They report architectural performance monitoring versionID = 4 (see Section 18.2.4) and support non-architectural monitoring capabilities described in this section.

Architectural performance monitoring version 4 capabilities are described in Section 18.2.4.

The bit fields (except bit 21) within each IA32\_PERFEVTSELx MSR are defined in Figure 18-6 and described in Section 18.2.1.1 and Section 18.2.3. Architectural and non-architectural performance monitoring events in the Goldmont microarchitecture ignore the AnyThread qualification regardless of its setting in the IA32\_PERFEVTSELx MSR.

The core PMU's capability is similar to that of the Silvermont microarchitecture described in Section 18.6, with some differences and enhancements summarized in Table 18-18.

**Table 18-18. Core PMU Comparison Between the Goldmont and Silvermont Microarchitectures**

Box	The Goldmont microarchitecture	The Silvermont microarchitecture	Comment
# of Fixed counters per core	3	3	Use CPUID to enumerate # of counters.
# of general-purpose counters per core	4	2	
Counter width (R,W)	R:48, W: 32/48	R:40, W:32	See Section 18.2.2.
Architectural Performance Monitoring version ID	4	3	Use CPUID to enumerate # of counters.
PMI Overhead Mitigation	<ul style="list-style-type: none"> <li>▪ Freeze_Perfmon_on_PMI with streamlined semantics.</li> <li>▪ Freeze_on_LBR with legacy semantics for branch profiling.</li> <li>▪ Freeze_while_SMM.</li> </ul>	<ul style="list-style-type: none"> <li>▪ Freeze_Perfmon_on_PMI with legacy semantics.</li> <li>▪ Freeze_on_LBR with legacy semantics for branch profiling.</li> <li>▪ Freeze_while_SMM.</li> </ul>	See Section 17.4.7. Legacy semantics not supported with version 4 or higher.
Counter and Buffer Overflow Status Management	<ul style="list-style-type: none"> <li>▪ Query via IA32_PERF_GLOBAL_STATUS</li> <li>▪ Reset via IA32_PERF_GLOBAL_STATUS_RESET</li> <li>▪ Set via IA32_PERF_GLOBAL_STATUS_SET</li> </ul>	<ul style="list-style-type: none"> <li>▪ Query via IA32_PERF_GLOBAL_STATUS</li> <li>▪ Reset via IA32_PERF_GLOBAL_OVF_CTRL</li> </ul>	See Section 18.2.4.
IA32_PERF_GLOBAL_STATUS Indicators of Overflow/Overhead/Interference	<ul style="list-style-type: none"> <li>▪ Individual counter overflow</li> <li>▪ PEBS buffer overflow</li> <li>▪ ToPA buffer overflow</li> <li>▪ CTR_Frz, LBR_Frz</li> </ul>	<ul style="list-style-type: none"> <li>▪ Individual counter overflow</li> <li>▪ PEBS buffer overflow</li> </ul>	See Section 18.2.4.
Enable control in IA32_PERF_GLOBAL_STATUS	<ul style="list-style-type: none"> <li>▪ CTR_Frz,</li> <li>▪ LBR_Frz</li> </ul>	No	See Section 18.2.4.1.
Perfmon Counter In-Use Indicator	Query IA32_PERF_GLOBAL_INUSE	No	See Section 18.2.4.3.
Processor Event Based Sampling (PEBS) Events	General-Purpose Counter 0 only. Supports all events (precise and non-precise). Precise events are listed in Table 18-19.	See Section 18.6.1.1. General-Purpose Counter 0 only. Only supports precise events (see Table 18-12).	IA32_PMC0 only.
PEBS record format encoding	0011b	0010b	
Reduce skid PEBS	IA32_PMC0 only	No	
Data Address Profiling	Yes	No	
PEBS record layout	Table 18-20; enhanced fields at offsets 90H- 98H; and TSC record field at COH.	Table 18-13.	
PEBS EventingIP	Yes	Yes	
Off-core Response Event	MSR 1A6H and 1A7H, each core has its own register.	MSR 1A6H and 1A7H, shared by a pair of cores.	Nehalem supports 1A6H only.

## 18.7.1 Processor Event Based Sampling (PEBS)

Processor event based sampling (PEBS) on the Goldmont microarchitecture is enhanced over prior generations with respect to sampling support of precise events and non-precise events. In the Goldmont microarchitecture, PEBS is supported using IA32\_PMC0 for all events (see Section 17.4.9).

PEBS uses a debug store mechanism to store a set of architectural state information for the processor at the time the sample was generated.

Precise events work the same way as on the Silvermont microarchitecture. They can capture precise eventingIP associated with a retired instruction that caused the event. The PEBS record provides architectural state of the instruction executed after the instruction that caused the event (See Section 18.4.4 and Section 17.4.9). The PEBS record also provides architectural state of the processor after the instruction that caused the event completes. The list of precise events supported in the Goldmont microarchitecture is shown in Table 18-19.

In the Goldmont microarchitecture, the PEBS facility also supports the use of non-precise events to record processor state information into PEBS records with the same format as with precise events.

However, a non-precise event may not be attributable to a particular retired instruction or the time of instruction execution. When the counter overflows, a PEBS record will be generated at the next opportunity. Consider the event ICACHE.HIT. When the counter overflows, the processor is fetching future instructions. The PEBS record will be generated at the next opportunity and capture the state at the processor's current retirement point. Other examples of a non-precise events are CPU\_CLK\_UNHALTED.CORE\_P and HARDWARE\_INTERRUPTS.RECEIVED. There may be many instructions in various stages of execution, multiple or zero instructions being retired each cycle as CPU\_CLK\_UNHALTED.CORE\_P increments. HARDWARE\_INTERRUPTS.RECEIVED increments independent of any instructions being executed. The PEBS record will be generated at the next opportunity, capturing the processor state when the machine received the interrupt, even if interrupts are masked. The PEBS facility thus allows for identification of the instructions which were executing when the event overflowed.

After generating a record, the PEBS facility reloads the counter and resumes execution, just as is done for precise events. Unlike interrupt-based sampling, which requires an interrupt service routine to collect the sample and reload the counter, the PEBS facility can collect samples even when interrupts are masked and without using NMI. Since a PEBS record is generated immediately when a counter for a non-precise event is enabled, it may also be generated after an overflow is set by an MSR write to IA32\_PERF\_GLOBAL\_STATUS\_SET.

**Table 18-19. Precise Events Supported by the Goldmont Microarchitecture**

Event Name	Event Select	Sub-event	UMask
LD_BLOCKS	03H	DATA_UNKNOWN	01H
		STORE_FORWARD	02H
		4K_ALIAS	04H
		UTLB_MISS	08H
		ALL_BLOCK	10H
MISALIGN_MEM_REF	13H	LOAD_PAGE_SPLIT	02H
		STORE_PAGE_SPLIT	04H
INST_RETIRED	COH	ANY	00H
UOPS_RETITRED	C2H	ANY	00H
		LD_SPLITSMS	01H
BR_INST_RETIRED	C4H	ALL_BRANCHES	00H
		JCC	7EH
		TAKEN_JCC	FEH
		CALL	F9H
		REL_CALL	FDH
		IND_CALL	FBH
		NON_RETURN_IND	EBH
		FAR_BRANCH	BFH

**Table 18-19. Precise Events Supported by the Goldmont Microarchitecture (Contd.)**

Event Name	Event Select	Sub-event	UMask
		RETURN	F7H
BR_MISP_RETIRED	C5H	ALL_BRANCHES	00H
		JCC	7EH
		TAKEN_JCC	FEH
		IND_CALL	FBH
		NON_RETURN_IND	EBH
		RETURN	F7H
MEM_UOPS_RETIRED	D0H	ALL_LOADS	81H
		ALL_STORES	82H
		ALL	83H
		DLTB_MISS_LOADS	11H
		DLTB_MISS_STORES	12H
		DLTB_MISS	13H
MEM_LOAD_UOPS_RETIRED	D1H	L1_HIT	01H
		L2_HIT	02H
		L1_MISS	08H
		L2_MISS	10H
		HITM	20H
		WCB_HIT	40H
		DRAM_HIT	80H

The PEBS record format supported by processors based on the Intel Silvermont microarchitecture is shown in Table 18-13, and each field in the PEBS record is 64 bits long.

**Table 18-20. PEBS Record Format for the Goldmont Microarchitecture**

Byte Offset	Field	Byte Offset	Field
00H	R/EFLAGS	68H	R11
08H	R/EIP	70H	R12
10H	R/EAX	78H	R13
18H	R/EBX	80H	R14
20H	R/ECX	88H	R15
28H	R/EDX	90H	Applicable Counters
30H	R/ESI	98H	Data Linear Address
38H	R/EDI	A0H	Reserved
40H	R/EBP	A8H	Reserved
48H	R/ESP	B0H	EventingRIP
50H	R8	B8H	Reserved
58H	R9	COH	TSC
60H	R10		

On Goldmont microarchitecture, all 64 bits of architectural registers are written into the PEBS record regardless of processor mode.

With PEBS record format encoding 0011b, offset 90H reports the “applicable counter” field, which is a multi-counter PEBS resolution index allowing software to correlate the PEBS record entry with the eventing PEBS overflow when multiple counters are configured to record PEBS records. Additionally, offset C0H captures a snapshot of the TSC that provides a time line annotation for each PEBS record entry.

### 18.7.1.1 PEBS Data Linear Address Profiling

Goldmont supports the Data Linear Address field introduced in Haswell. It does not support the Data Source Encoding or Latency Value fields that are also part of Data Address Profiling. The fields are present in the record but are reserved.

For Goldmont, the Data Linear Address field will record the linear address of memory accesses in the previous instruction (e.g. the one that triggered a precise event that caused the PEBS record to be generated).

### 18.7.1.2 Reduced Skid PEBS

For precise events, upon triggering a PEBS assist, there will be a finite delay between the time the counter overflows and when the microcode starts to carry out its data collection obligations. The Reduced Skid mechanism mitigates the “skid” problem by providing an early indication of when the counter is about to overflow, allowing the machine to more precisely trap on the instruction that actually caused the counter overflow thus greatly reducing skid.

This mechanism is a superset of the PDIR mechanism available in the Sandy Bridge microarchitecture. See Section 18.9.4.4

In the Goldmont microarchitecture, the mechanism applies to all precise events including INST\_RETIRED. However, the Reduced Skid mechanism is disabled for any counter when the INV, ANY, E, or CMASK fields are set.

To ensure the Reduced Skid mechanism operates correctly, disable PEBS via the IA32\_PEBS\_ENABLE or IA32\_PERF\_GLOBAL\_CTRL MSRs before writing to the configuration registers (IA32\_PERFEVTSELx) or to the counters (IA32\_PMCx and IA32\_A\_PMCx).

### 18.7.1.3 Enhancements to IA32\_PERF\_GLOBAL\_STATUS.OvfDSBuffer[62]

In addition to IA32\_PERF\_GLOBAL\_STATUS.OvfDSBuffer[62] being set when PEBS\_Index reaches the PEBS\_Interrupt\_Threshold, the bit is also set when PEBS\_Index is out of bounds. That is, the bit will be set when PEBS\_Index < PEBS\_Buffer\_Base or PEBS\_Index > PEBS\_Absolute\_Maximum. Note that when an out of bound condition is encountered, the overflow bits in IA32\_PERF\_GLOBAL\_STATUS will be cleared according to Applicable Counters, however the IA32\_PMCx values will not be reloaded with the Reset values stored in the DS\_AREA.

## 18.7.2 Offcore Response Event

Event number 0B7H support offcore response monitoring using an associated configuration MSR, MSR\_OFFCORE\_RSP0 (address 1A6H) in conjunction with umask value 01H or MSR\_OFFCORE\_RSP1 (address 1A7H) in conjunction with umask value 02H. Table 18-14 lists the event code, mask value and additional off-core configuration MSR that must be programmed to count off-core response events using IA32\_PMCx.

The Goldmont microarchitecture provides unique pairs of MSR\_OFFCORE\_RSPx registers per core.

The layout of MSR\_OFFCORE\_RSP0 and MSR\_OFFCORE\_RSP1 are organized as follows:

- Bits 15:0 specifies the request type of a transaction request to the uncore. This is described in Table 18-21.
- Bits 30:16 specifies common supplier information or an L2 Hit, and is described in Table 18-16.
- If L2 misses, then Bits 37:31 can be used to specify snoop response information and is described in Table 18-22.
- For outstanding requests, bit 38 can enable measurement of average latency of specific type of offcore transaction requests using two programmable counter simultaneously; see Section 18.6.3 for details.

**Table 18-21. MSR\_OFFCORE\_RSPx Request\_Type Field Definition**

Bit Name	Offset	Description
DEMAND_DATA_RD	0	(R/W) Counts cacheline read requests due to demand reads (excludes prefetches).
DEMAND_RFO	1	(R/W) Counts cacheline read for ownership (RFO) requests due to demand writes (excludes prefetches).
DEMAND_CODE_RD	2	(R/W) Counts demand instruction cacheline and I-side prefetch requests that miss the instruction cache.
COREWB	3	(R/W) Counts writeback transactions caused by L1 or L2 cache evictions.
PF_L2_DATA_RD	4	(R/W) Counts data cacheline reads generated by hardware L2 cache prefetcher.
PF_L2_RFO	5	(R/W) Counts reads for ownership (RFO) requests generated by L2 prefetcher.
Reserved	6	Reserved.
PARTIAL_READS	7	(R/W) Counts demand data partial reads, including data in uncacheable (UC) or uncacheable (WC) write combining memory types.
PARTIAL_WRITES	8	(R/W) Counts partial writes, including uncacheable (UC), write through (WT) and write protected (WP) memory type writes.
UC_CODE_READS	9	(R/W) Counts code reads in uncacheable (UC) memory region.
BUS_LOCKS	10	(R/W) Counts bus lock and split lock requests.
FULL_STREAMING_STORES	11	(R/W) Counts full cacheline writes due to streaming stores.
SW_PREFETCH	12	(R/W) Counts cacheline requests due to software prefetch instructions.
PF_L1_DATA_RD	13	(R/W) Counts data cacheline reads generated by hardware L1 data cache prefetcher.
PARTIAL_STREAMING_STORES	14	(R/W) Counts partial cacheline writes due to streaming stores.
ANY_REQUEST	15	(R/W) Counts requests to the uncore subsystem.

To properly program this extra register, software must set at least one request type bit (Table 18-15) and a valid response type pattern (either Table 18-16 or Table 18-22). Otherwise, the event count reported will be zero. It is permissible and useful to set multiple request and response type bits in order to obtain various classes of off-core response events. Although MSR\_OFFCORE\_RSPx allow an agent software to program numerous combinations that meet the above guideline, not all combinations produce meaningful data.

**Table 18-22. MSR\_OFFCORE\_RSPx For L2 Miss and Outstanding Requests**

Subtype	Bit Name	Offset	Description
L2_MISS (Snoop Info)	Reserved	32:31	Reserved
	L2_MISS.SNOOP_MISS_OR_NO_SNOOP_NEEDED	33	(R/W). A true miss to this module, for which a snoop request missed the other module or no snoop was performed/needed.
	L2_MISS.HIT_OTHER_CORE_NO_FWD	34	(R/W) A snoop hit in the other processor module, but no data forwarding is required.
	Reserved	35	Reserved
	L2_MISS.HITM_OTHER_CORE	36	(R/W) Counts the number of snoops hit in the other module or other core's L1 where modified copies were found.
	L2_MISS.NON_DRAM	37	(R/W) Target was a non-DRAM system address. This includes MMIO transactions.
Outstanding requests <sup>1</sup>	OUTSTANDING	38	(R/W) Counts weighted cycles of outstanding offcore requests of the request type specified in bits 15:0, from the time the XQ receives the request and any response is received. Bits 37:16 must be set to 0. This bit is only available in MSR_OFFCORE_RESP0.

**NOTES:**

1. See Section 18.6.3, “Average Offcore Request Latency Measurement” for details on how to use this bit to extract average latency.

To specify a complete offcore response filter, software must properly program bits in the request and response type fields. A valid request type must have at least one bit set in the non-reserved bits of 15:0. A valid response type must be a non-zero value of the following expression:

```
[ANY 'OR' (L2 Hit) ] 'XOR' ( Snoop Info Bits) 'XOR' (Avg Latency)
```

### 18.7.3 Average Offcore Request Latency Measurement

In Goldmont microarchitecture, measurement of average latency of offcore transaction requests is the same as described in Section 18.6.3.

## 18.8 PERFORMANCE MONITORING FOR PROCESSORS BASED ON INTEL® MICROARCHITECTURE CODE NAME NEHALEM

Intel Core i7 processor family<sup>2</sup> supports architectural performance monitoring capability with version ID 3 (see Section 18.2.3) and a host of non-architectural monitoring capabilities. The Intel Core i7 processor family is based on Intel® microarchitecture code name Nehalem, and provides four general-purpose performance counters (IA32\_PMC0, IA32\_PMC1, IA32\_PMC2, IA32\_PMC3) and three fixed-function performance counters (IA32\_FIXED\_CTR0, IA32\_FIXED\_CTR1, IA32\_FIXED\_CTR2) in the processor core.

Non-architectural performance monitoring in Intel Core i7 processor family uses the IA32\_PERFVTSELx MSR to configure a set of non-architecture performance monitoring events to be counted by the corresponding general-purpose performance counter. The list of non-architectural performance monitoring events is listed in Table 19-26. Non-architectural performance monitoring events fall into two broad categories:

- Performance monitoring events in the processor core: These include many events that are similar to performance monitoring events available to processor based on Intel Core microarchitecture. Additionally, there are several enhancements in the performance monitoring capability for detecting microarchitectural conditions in the processor core or in the interaction of the processor core to the off-core sub-systems in the physical processor package. The off-core sub-systems in the physical processor package is loosely referred to as “uncore”.
- Performance monitoring events in the uncore: The uncore sub-system is shared by more than one processor cores in the physical processor package. It provides additional performance monitoring facility outside of IA32\_PMCx and performance monitoring events that are specific to the uncore sub-system.

Architectural and non-architectural performance monitoring events in Intel Core i7 processor family support thread qualification using bit 21 of IA32\_PERFVTSELx MSR.

The bit fields within each IA32\_PERFVTSELx MSR are defined in Figure 18-6 and described in Section 18.2.1.1 and Section 18.2.3.

---

2. Intel Xeon processor 5500 series and 3400 series are also based on Intel microarchitecture code name Nehalem, so the performance monitoring facilities described in this section generally also apply.

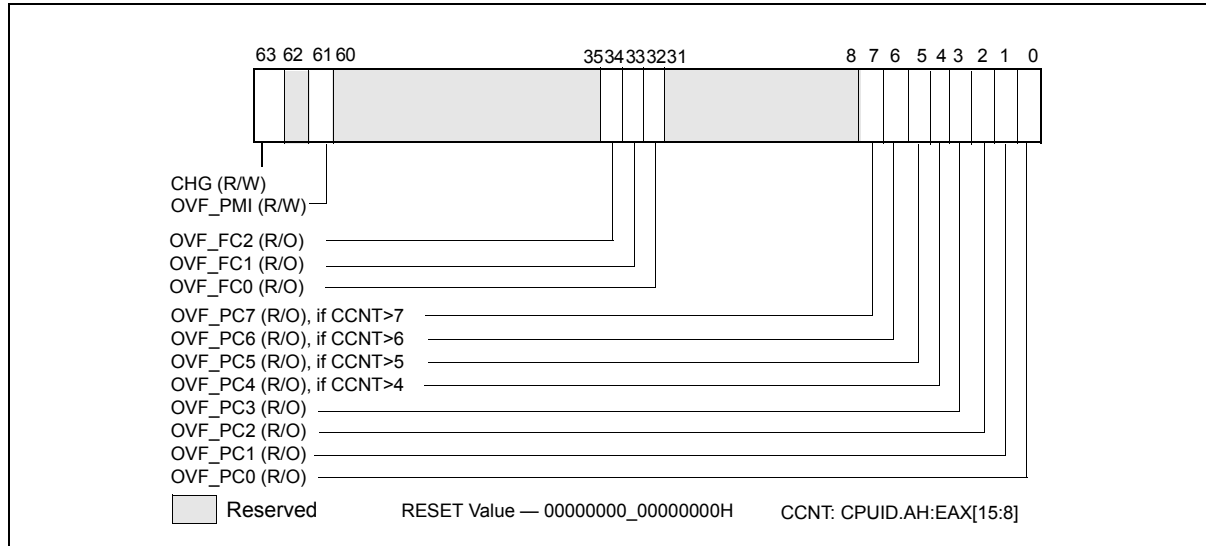


Figure 18-20. IA32\_PERF\_GLOBAL\_STATUS MSR

## 18.8.1 Enhancements of Performance Monitoring in the Processor Core

The notable enhancements in the monitoring of performance events in the processor core include:

- Four general purpose performance counters, IA32\_PMCx, associated counter configuration MSRs, IA32\_PERFEVTSELx, and global counter control MSR supporting simplified control of four counters. Each of the four performance counter can support processor event based sampling (PEBS) and thread-qualification of architectural and non-architectural performance events. Width of IA32\_PMCx supported by hardware has been increased. The width of counter reported by CPUID.0AH:EAX[23:16] is 48 bits. The PEBS facility in Intel micro-architecture code name Nehalem has been enhanced to include new data format to capture additional information, such as load latency.
- Load latency sampling facility. Average latency of memory load operation can be sampled using load-latency facility in processors based on Intel microarchitecture code name Nehalem. This field measures the load latency from load's first dispatch of till final data writeback from the memory subsystem. The latency is reported for retired demand load operations and in core cycles (it accounts for re-dispatches). This facility is used in conjunction with the PEBS facility.
- Off-core response counting facility. This facility in the processor core allows software to count certain transaction responses between the processor core to sub-systems outside the processor core (uncore). Counting off-core response requires additional event qualification configuration facility in conjunction with IA32\_PERFEVTSELx. Two off-core response MSRs are provided to use in conjunction with specific event codes that must be specified with IA32\_PERFEVTSELx.

### 18.8.1.1 Processor Event Based Sampling (PEBS)

All four general-purpose performance counters, IA32\_PMCx, can be used for PEBS if the performance event supports PEBS. Software uses IA32\_MISC\_ENABLE[7] and IA32\_MISC\_ENABLE[12] to detect whether the performance monitoring facility and PEBS functionality are supported in the processor. The MSR IA32\_PEBS\_ENABLE provides 4 bits that software must use to enable which IA32\_PMCx overflow condition will cause the PEBS record to be captured.

Additionally, the PEBS record is expanded to allow latency information to be captured. The MSR IA32\_PEBS\_ENABLE provides 4 additional bits that software must use to enable latency data recording in the PEBS record upon the respective IA32\_PMCx overflow condition. The layout of IA32\_PEBS\_ENABLE for processors based on Intel microarchitecture code name Nehalem is shown in Figure 18-21.



When a counter is enabled to capture machine state (PEBS\_EN\_PMCx = 1), the processor will write machine state information to a memory buffer specified by software as detailed below. When the counter IA32\_PMCx overflows from maximum count to zero, the PEBS hardware is armed.

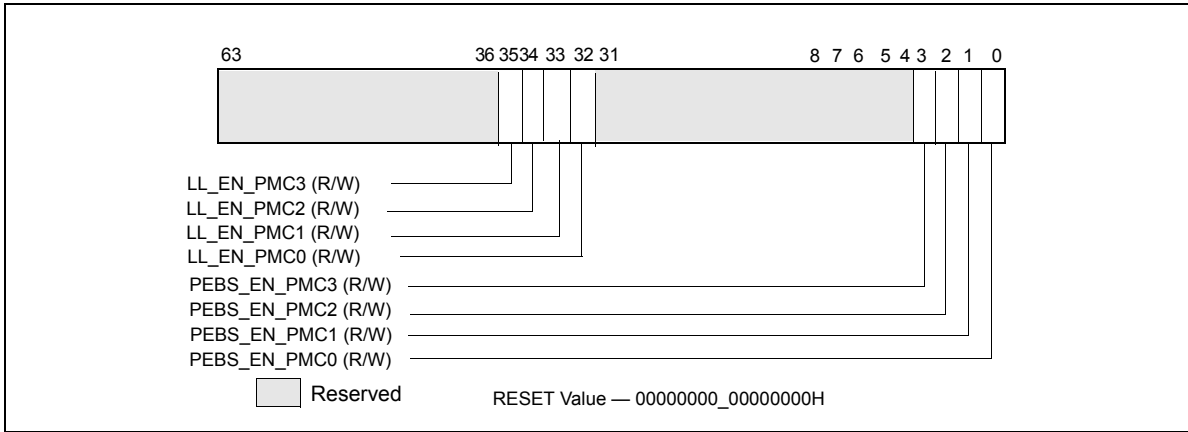


Figure 18-21. Layout of IA32\_PEBS\_ENABLE MSR

Upon occurrence of the next PEBS event, the PEBS hardware triggers an assist and causes a PEBS record to be written. The format of the PEBS record is indicated by the bit field IA32\_PERF\_CAPABILITIES[11:8] (see Figure 18-49).

The behavior of PEBS assists is reported by IA32\_PERF\_CAPABILITIES[6] (see Figure 18-49). The return instruction pointer (RIP) reported in the PEBS record will point to the instruction after (+1) the instruction that causes the PEBS assist. The machine state reported in the PEBS record is the machine state after the instruction that causes the PEBS assist is retired. For instance, if the instructions:

```
mov eax, [eax] ; causes PEBS assist
nop
```

are executed, the PEBS record will report the address of the nop, and the value of EAX in the PEBS record will show the value read from memory, not the target address of the read operation.

The PEBS record format is shown in Table 18-23, and each field in the PEBS record is 64 bits long. The PEBS record format, along with debug/store area storage format, does not change regardless of IA-32e mode is active or not. CPUID.01H:ECX.DTES64[bit 2] reports whether the processor's DS storage format support is mode-independent. When set, it uses 64-bit DS storage format.

**Table 18-23. PEBS Record Format for Intel Core i7 Processor Family**

Byte Offset	Field	Byte Offset	Field
00H	R/EFLAGS	58H	R9
08H	R/EIP	60H	R10
10H	R/EAX	68H	R11
18H	R/EBX	70H	R12
20H	R/ECX	78H	R13
28H	R/EDX	80H	R14
30H	R/ESI	88H	R15
38H	R/EDI	90H	IA32_PERF_GLOBAL_STATUS
40H	R/EBP	98H	Data Linear Address
48H	R/ESP	A0H	Data Source Encoding
50H	R8	A8H	Latency value (core cycles)

In IA-32e mode, the full 64-bit value is written to the register. If the processor is not operating in IA-32e mode, 32-bit value is written to registers with bits 63:32 zeroed. Registers not defined when the processor is not in IA-32e mode are written to zero.

Bytes AFH:90H are enhancement to the PEBS record format. Support for this enhanced PEBS record format is indicated by IA32\_PERF\_CAPABILITIES[11:8] encoding of 0001B.

The value written to bytes 97H:90H is the state of the IA32\_PERF\_GLOBAL\_STATUS register before the PEBS assist occurred. This value is written so software can determine which counters overflowed when this PEBS record was written. Note that this field indicates the overflow status for all counters, regardless of whether they were programmed for PEBS or not.

### Programming PEBS Facility

Only a subset of non-architectural performance events in the processor support PEBS. The subset of precise events are listed in Table 18-10. In addition to using IA32\_PERFEVTSELx to specify event unit/mask settings and setting the EN\_PMCx bit in the IA32\_PEBS\_ENABLE register for the respective counter, the software must also initialize the DS\_BUFFER\_MANAGEMENT\_AREA data structure in memory to support capturing PEBS records for precise events.

### NOTE

PEBS events are only valid when the following fields of IA32\_PERFEVTSELx are all zero: AnyThread, Edge, Invert, CMask.

The beginning linear address of the DS\_BUFFER\_MANAGEMENT\_AREA data structure must be programmed into the IA32\_DS\_AREA register. The layout of the DS\_BUFFER\_MANAGEMENT\_AREA is shown in Figure 18-22.

- **PEBS Buffer Base:** This field is programmed with the linear address of the first byte of the PEBS buffer allocated by software. The processor reads this field to determine the base address of the PEBS buffer. Software should allocate this memory from the non-paged pool.
- **PEBS Index:** This field is initially programmed with the same value as the PEBS Buffer Base field, or the beginning linear address of the PEBS buffer. The processor reads this field to determine the location of the next PEBS record to write to. After a PEBS record has been written, the processor also updates this field with the address of the next PEBS record to be written. The figure above illustrates the state of PEBS Index after the first PEBS record is written.
- **PEBS Absolute Maximum:** This field represents the absolute address of the maximum length of the allocated PEBS buffer plus the starting address of the PEBS buffer. The processor will not write any PEBS record beyond the end of PEBS buffer, when **PEBS Index** equals **PEBS Absolute Maximum**. No signaling is generated when PEBS buffer is full. Software must reset the **PEBS Index** field to the beginning of the PEBS buffer address to continue capturing PEBS records.

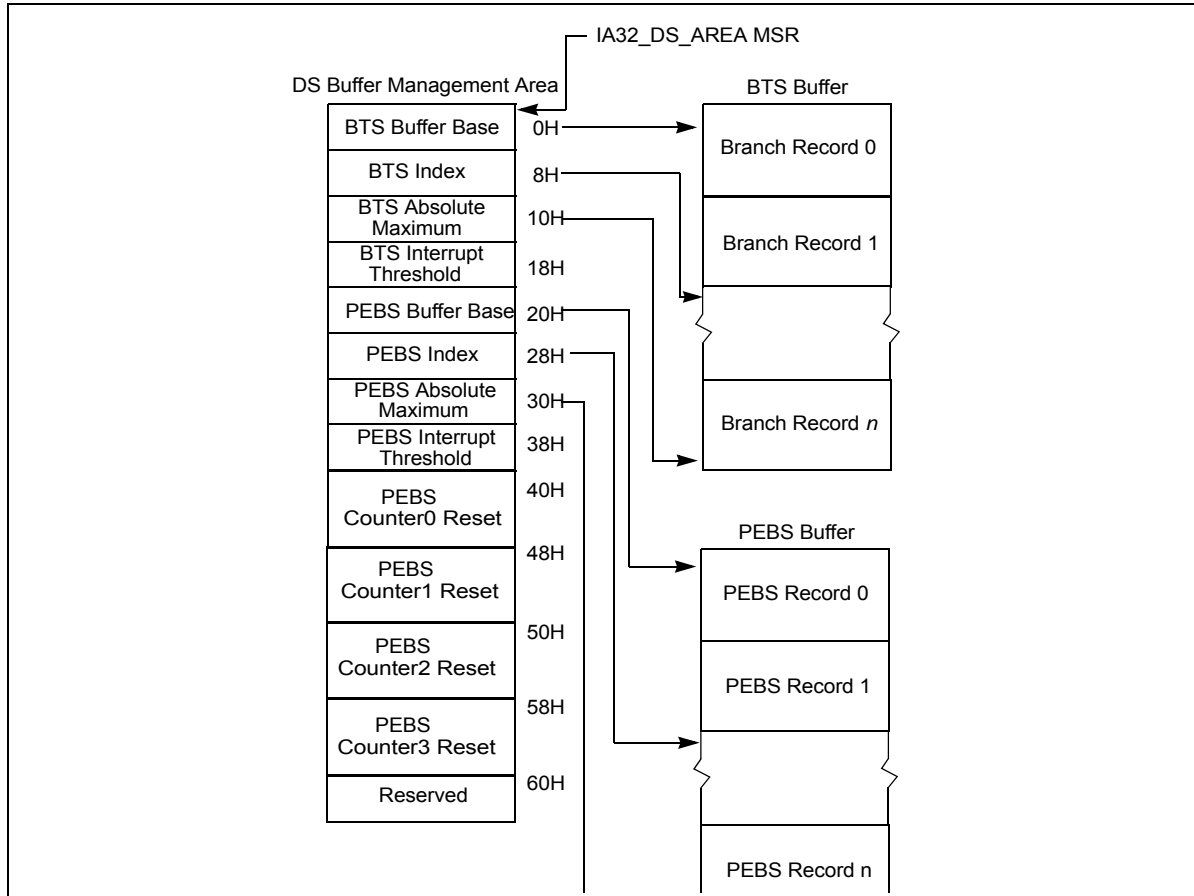


Figure 18-22. PEBS Programming Environment

- PEBS Interrupt Threshold:** This field specifies the threshold value to trigger a performance interrupt and notify software that the PEBS buffer is nearly full. This field is programmed with the linear address of the first byte of the PEBS record within the PEBS buffer that represents the threshold record. After the processor writes a PEBS record and updates **PEBS Index**, if the **PEBS Index** reaches the threshold value of this field, the processor will generate a performance interrupt. This is the same interrupt that is generated by a performance counter overflow, as programmed in the Performance Monitoring Counters vector in the Local Vector Table of the Local APIC. When a performance interrupt due to PEBS buffer full is generated, the `IA32_PERF_GLOBAL_STATUS.PEBS_Ovf` bit will be set.
- PEBS CounterX Reset:** This field allows software to set up PEBS counter overflow condition to occur at a rate useful for profiling workload, thereby generating multiple PEBS records to facilitate characterizing the profile the execution of test code. After each PEBS record is written, the processor checks each counter to see if it overflowed and was enabled for PEBS (the corresponding bit in `IA32_PEBS_ENABLED` was set). If these conditions are met, then the reset value for each overflowed counter is loaded from the DS Buffer Management Area. For example, if counter `IA32_PMC0` caused a PEBS record to be written, then the value of "PEBS Counter 0 Reset" would be written to counter `IA32_PMC0`. If a counter is not enabled for PEBS, its value will not be modified by the PEBS assist.

### Performance Counter Prioritization

Performance monitoring interrupts are triggered by a counter transitioning from maximum count to zero (assuming `IA32_PerfEvtSelX.INT` is set). This same transition will cause PEBS hardware to arm, but not trigger. PEBS hardware triggers upon detection of the first PEBS event after the PEBS hardware has been armed (a 0 to 1 transition of the counter). At this point, a PEBS assist will be undertaken by the processor.

Performance counters (fixed and general-purpose) are prioritized in index order. That is, counter IA32\_PMC0 takes precedence over all other counters. Counter IA32\_PMC1 takes precedence over counters IA32\_PMC2 and IA32\_PMC3, and so on. This means that if simultaneous overflows or PEBS assists occur, the appropriate action will be taken for the highest priority performance counter. For example, if IA32\_PMC1 cause an overflow interrupt and IA32\_PMC2 causes an PEBS assist simultaneously, then the overflow interrupt will be serviced first.

The PEBS threshold interrupt is triggered by the PEBS assist, and is by definition prioritized lower than the PEBS assist. Hardware will not generate separate interrupts for each counter that simultaneously overflows. General-purpose performance counters are prioritized over fixed counters.

If a counter is programmed with a precise (PEBS-enabled) event and programmed to generate a counter overflow interrupt, the PEBS assist is serviced before the counter overflow interrupt is serviced. If in addition the PEBS interrupt threshold is met, the

threshold interrupt is generated after the PEBS assist completes, followed by the counter overflow interrupt (two separate interrupts are generated).

Uncore counters may be programmed to interrupt one or more processor cores (see Section 18.8.2). It is possible for interrupts posted from the uncore facility to occur coincident with counter overflow interrupts from the processor core. Software must check core and uncore status registers to determine the exact origin of counter overflow interrupts.

### 18.8.1.2 Load Latency Performance Monitoring Facility

The load latency facility provides software a means to characterize the average load latency to different levels of cache/memory hierarchy. This facility requires processor supporting enhanced PEBS record format in the PEBS buffer, see Table 18-23. This field measures the load latency from load's first dispatch of till final data writeback from the memory subsystem. The latency is reported for retired demand load operations and in core cycles (it accounts for re-dispatches).

To use this feature software must assure:

- One of the IA32\_PERFEVTSELx MSR is programmed to specify the event unit MEM\_INST\_RETIRED, and the LATENCY\_ABOVE\_THRESHOLD event mask must be specified (IA32\_PerfEvtSelX[15:0] = 100H). The corresponding counter IA32\_PMCx will accumulate event counts for architecturally visible loads which exceed the programmed latency threshold specified separately in a MSR. Stores are ignored when this event is programmed. The CMASK or INV fields of the IA32\_PerfEvtSelX register used for counting load latency must be 0. Writing other values will result in undefined behavior.
- The MSR\_PEBS\_LD\_LAT\_THRESHOLD MSR is programmed with the desired latency threshold in core clock cycles. Loads with latencies greater than this value are eligible for counting and latency data reporting. The minimum value that may be programmed in this register is 3 (the minimum detectable load latency is 4 core clock cycles).
- The PEBS enable bit in the IA32\_PEBS\_ENABLE register is set for the corresponding IA32\_PMCx counter register. This means that both the PEBS\_EN\_CTRX and LL\_EN\_CTRX bits must be set for the counter(s) of interest. For example, to enable load latency on counter IA32\_PMC0, the IA32\_PEBS\_ENABLE register must be programmed with the 64-bit value 00000001\_00000001H.

When the load-latency facility is enabled, load operations are randomly selected by hardware and tagged to carry information related to data source locality and latency. Latency and data source information of tagged loads are updated internally.

When a PEBS assist occurs, the last update of latency and data source information are captured by the assist and written as part of the PEBS record. The PEBS sample after value (SAV), specified in PEBS CounterX Reset, operates orthogonally to the tagging mechanism. Loads are randomly tagged to collect latency data. The SAV controls the number of tagged loads with latency information that will be written into the PEBS record field by the PEBS assists. The load latency data written to the PEBS record will be for the last tagged load operation which retired just before the PEBS assist was invoked.

The load-latency information written into a PEBS record (see Table 18-23, bytes AFH:98H) consists of:

- **Data Linear Address:** This is the linear address of the target of the load operation.
- **Latency Value:** This is the elapsed cycles of the tagged load operation between dispatch to GO, measured in processor core clock domain.

- Data Source:** The encoded value indicates the origin of the data obtained by the load instruction. The encoding is shown in Table 18-24. In the descriptions local memory refers to system memory physically attached to a processor package, and remote memory referrals to system memory physically attached to another processor package.

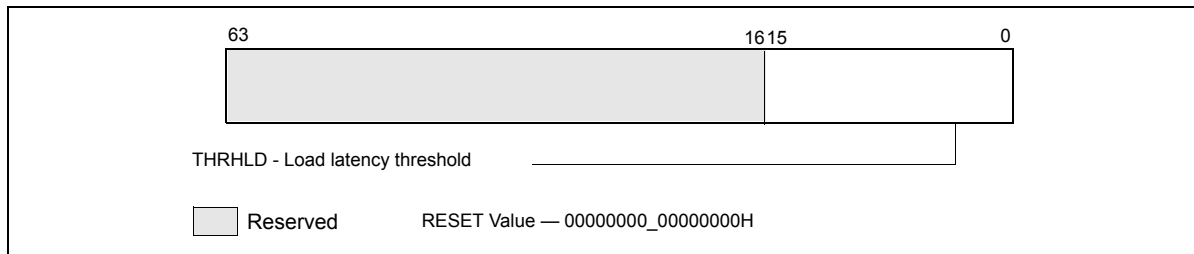
**Table 18-24. Data Source Encoding for Load Latency Record**

Encoding	Description
00H	Unknown L3 cache miss
01H	Minimal latency core cache hit. This request was satisfied by the L1 data cache.
02H	Pending core cache HIT. Outstanding core cache miss to same cache-line address was already underway.
03H	This data request was satisfied by the L2.
04H	L3 HIT. Local or Remote home requests that hit L3 cache in the uncore with no coherency actions required (snooping).
05H	L3 HIT. Local or Remote home requests that hit the L3 cache and was serviced by another processor core with a cross core snoop where no modified copies were found. (clean).
06H	L3 HIT. Local or Remote home requests that hit the L3 cache and was serviced by another processor core with a cross core snoop where modified copies were found. (HITM).
07H <sup>1</sup>	Reserved/LLC Snoop HitM. Local or Remote home requests that hit the last level cache and was serviced by another core with a cross core snoop where modified copies found
08H	L3 MISS. Local homed requests that missed the L3 cache and was serviced by forwarded data following a cross package snoop where no modified copies found. (Remote home requests are not counted).
09H	Reserved
0AH	L3 MISS. Local home requests that missed the L3 cache and was serviced by local DRAM (go to shared state).
0BH	L3 MISS. Remote home requests that missed the L3 cache and was serviced by remote DRAM (go to shared state).
0CH	L3 MISS. Local home requests that missed the L3 cache and was serviced by local DRAM (go to exclusive state).
0DH	L3 MISS. Remote home requests that missed the L3 cache and was serviced by remote DRAM (go to exclusive state).
0EH	I/O, Request of input/output operation
0FH	The request was to un-cacheable memory.

**NOTES:**

- Bit 7 is supported only for processor with CPUID DisplayFamily\_DisplayModel signature of 06\_2A, and 06\_2E; otherwise it is reserved.

The layout of MSR\_PEBS\_LD\_LAT\_THRESHOLD is shown in Figure 18-23.



**Figure 18-23. Layout of MSR\_PEBS\_LD\_LAT MSR**

Bits 15:0 specifies the threshold load latency in core clock cycles. Performance events with latencies greater than this value are counted in IA32\_PMCx and their latency information is reported in the PEBS record. Otherwise, they are ignored. The minimum value that may be programmed in this field is 3.

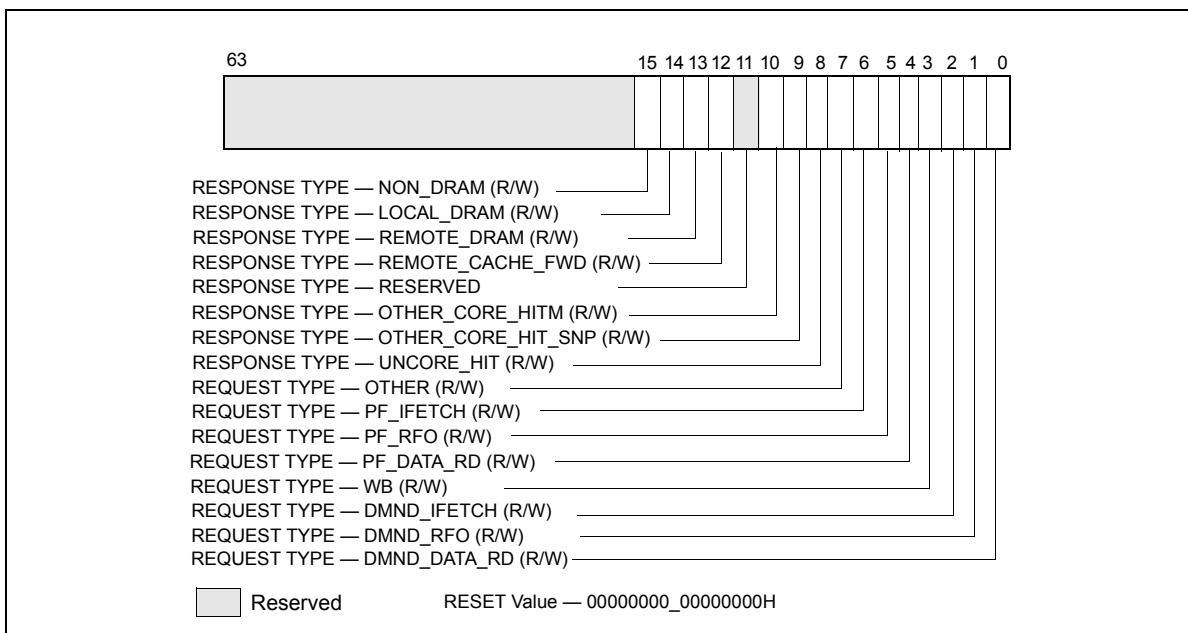
### 18.8.1.3 Off-core Response Performance Monitoring in the Processor Core

Programming a performance event using the off-core response facility can choose any of the four IA32\_PERFEVTSELx MSR with specific event codes and predefine mask bit value. Each event code for off-core response monitoring requires programming an associated configuration MSR, MSR\_OFFCORE\_RSP\_0. There is only one off-core response configuration MSR. Table 18-25 lists the event code, mask value and additional off-core configuration MSR that must be programmed to count off-core response events using IA32\_PMCx.

**Table 18-25. Off-Core Response Event Encoding**

Event code in IA32_PERFEVTSELx	Mask Value in IA32_PERFEVTSELx	Required Off-core Response MSR
B7H	01H	MSR_OFFCORE_RSP_0 (address 1A6H)

The layout of MSR\_OFFCORE\_RSP\_0 is shown in Figure 18-24. Bits 7:0 specifies the request type of a transaction request to the uncore. Bits 15:8 specifies the response of the uncore subsystem.



**Figure 18-24. Layout of MSR\_OFFCORE\_RSP\_0 and MSR\_OFFCORE\_RSP\_1 to Configure Off-core Response Events**

**Table 18-26. MSR\_OFFCORE\_RSP\_0 and MSR\_OFFCORE\_RSP\_1 Bit Field Definition**

Bit Name	Offset	Description
DMND_DATA_RD	0	(R/W). Counts the number of demand and DCU prefetch data reads of full and partial cachelines as well as demand data page table entry cacheline reads. Does not count L2 data read prefetches or instruction fetches.
DMND_RFO	1	(R/W). Counts the number of demand and DCU prefetch reads for ownership (RFO) requests generated by a write to data cacheline. Does not count L2 RFO.
DMND_IFETCH	2	(R/W). Counts the number of demand and DCU prefetch instruction cacheline reads. Does not count L2 code read prefetches.
WB	3	(R/W). Counts the number of writeback (modified to exclusive) transactions.
PF_DATA_RD	4	(R/W). Counts the number of data cacheline reads generated by L2 prefetchers.
PF_RFO	5	(R/W). Counts the number of RFO requests generated by L2 prefetchers.

**Table 18-26. MSR\_OFFCORE\_RSP\_0 and MSR\_OFFCORE\_RSP\_1 Bit Field Definition (Contd.)**

Bit Name	Offset	Description
PF_IFETCH	6	(R/W). Counts the number of code reads generated by L2 prefetchers.
OTHER	7	(R/W). Counts one of the following transaction types, including L3 invalidate, I/O, full or partial writes, WC or non-temporal stores, CLFLUSH, Fences, lock, unlock, split lock.
UNCORE_HIT	8	(R/W). L3 Hit: local or remote home requests that hit L3 cache in the uncore with no coherency actions required (snooping).
OTHER_CORE_HIT_SNP	9	(R/W). L3 Hit: local or remote home requests that hit L3 cache in the uncore and was serviced by another core with a cross core snoop where no modified copies were found (clean).
OTHER_CORE_HIT_TM	10	(R/W). L3 Hit: local or remote home requests that hit L3 cache in the uncore and was serviced by another core with a cross core snoop where modified copies were found (HITM).
Reserved	11	Reserved
REMOTE_CACHE_FWD	12	(R/W). L3 Miss: local homed requests that missed the L3 cache and was serviced by forwarded data following a cross package snoop where no modified copies found. (Remote home requests are not counted)
REMOTE_DRAM	13	(R/W). L3 Miss: remote home requests that missed the L3 cache and were serviced by remote DRAM.
LOCAL_DRAM	14	(R/W). L3 Miss: local home requests that missed the L3 cache and were serviced by local DRAM.
NON_DRAM	15	(R/W). Non-DRAM requests that were serviced by IOH.

## 18.8.2 Performance Monitoring Facility in the Uncore

The “uncore” in Intel microarchitecture code name Nehalem refers to subsystems in the physical processor package that are shared by multiple processor cores. Some of the sub-systems in the uncore include the L3 cache, Intel QuickPath Interconnect link logic, and integrated memory controller. The performance monitoring facilities inside the uncore operates in the same clock domain as the uncore (U-clock domain), which is usually different from the processor core clock domain. The uncore performance monitoring facilities described in this section apply to Intel Xeon processor 5500 series and processors with the following CPUID signatures: 06\_1AH, 06\_1EH, 06\_1FH (see Chapter 35). An overview of the uncore performance monitoring facilities is described separately.

The performance monitoring facilities available in the U-clock domain consist of:

- Eight General-purpose counters (MSR\_UNCORE\_PerfCntr0 through MSR\_UNCORE\_PerfCntr7). The counters are 48 bits wide. Each counter is associated with a configuration MSR, MSR\_UNCORE\_PerfEvtSelx, to specify event code, event mask and other event qualification fields. A set of global uncore performance counter enabling/overflow/status control MSRs are also provided for software.
- Performance monitoring in the uncore provides an address/opcode match MSR that provides event qualification control based on address value or QPI command opcode.
- One fixed-function counter, MSR\_UNCORE\_FixedCntr0. The fixed-function uncore counter increments at the rate of the U-clock when enabled.

The frequency of the uncore clock domain can be determined from the uncore clock ratio which is available in the PCI configuration space register at offset C0H under device number 0 and Function 0.

### 18.8.2.1 Uncore Performance Monitoring Management Facility

MSR\_UNCORE\_PERF\_GLOBAL\_CTRL provides bit fields to enable/disable general-purpose and fixed-function counters in the uncore. Figure 18-25 shows the layout of MSR\_UNCORE\_PERF\_GLOBAL\_CTRL for an uncore that is shared by four processor cores in a physical package.

- EN\_PCn (bit n, n = 0, 7): When set, enables counting for the general-purpose uncore counter MSR\_UNCORE\_PerfCntr n.
- EN\_FC0 (bit 32): When set, enables counting for the fixed-function uncore counter MSR\_UNCORE\_FixedCntr0.

- EN\_PMI\_COREn (bit n, n = 0, 3 if four cores are present): When set, processor core n is programmed to receive an interrupt signal from any interrupt enabled uncore counter. PMI delivery due to an uncore counter overflow is enabled by setting IA32\_DEBUGCTL.Offcore\_PMI\_EN to 1.
- PMI\_FRZ (bit 63): When set, all U-clock uncore counters are disabled when any one of them signals a performance interrupt. Software must explicitly re-enable the counter by setting the enable bits in MSR\_UNCORE\_PERF\_GLOBAL\_CTRL upon exit from the ISR.

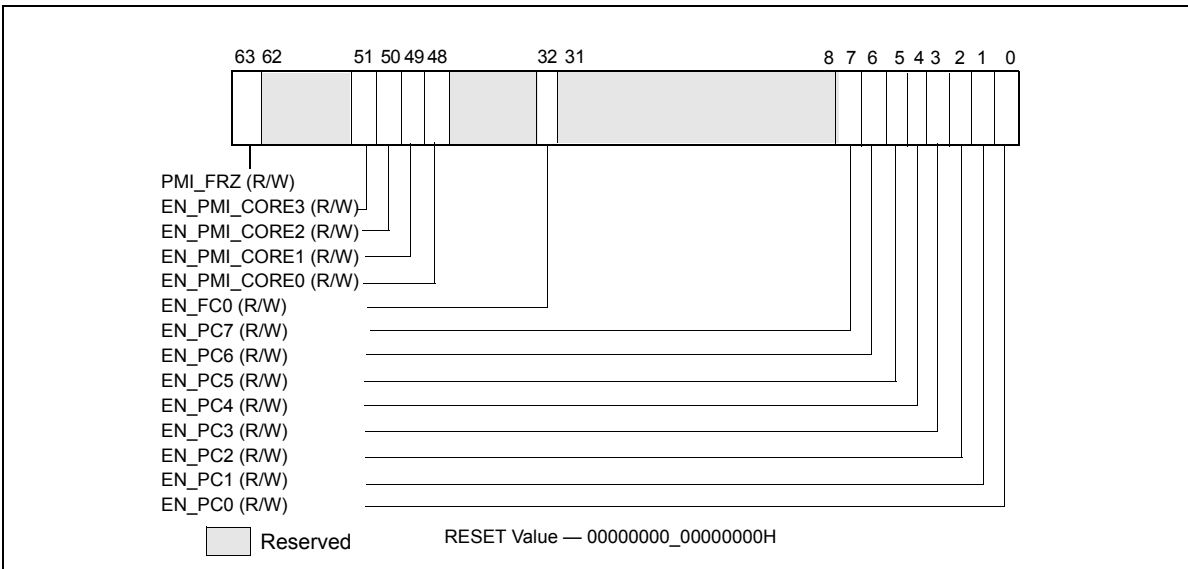


Figure 18-25. Layout of MSR\_UNCORE\_PERF\_GLOBAL\_CTRL MSR

MSR\_UNCORE\_PERF\_GLOBAL\_STATUS provides overflow status of the U-clock performance counters in the uncore. This is a read-only register. If an overflow status bit is set the corresponding counter has overflowed. The register provides a condition change bit (bit 63) which can be quickly checked by software to determine if a significant change has occurred since the last time the condition change status was cleared. Figure 18-26 shows the layout of MSR\_UNCORE\_PERF\_GLOBAL\_STATUS.

- OVF\_PCn (bit n, n = 0, 7): When set, indicates general-purpose uncore counter MSR\_UNCORE\_PerfCntr n has overflowed.
- OVF\_FC0 (bit 32): When set, indicates the fixed-function uncore counter MSR\_UNCORE\_FixedCntr0 has overflowed.
- OVF\_PMI (bit 61): When set indicates that an uncore counter overflowed and generated an interrupt request.
- CHG (bit 63): When set indicates that at least one status bit in MSR\_UNCORE\_PERF\_GLOBAL\_STATUS register has changed state.

MSR\_UNCORE\_PERF\_GLOBAL\_OVF\_CTRL allows software to clear the status bits in the UNCORE\_PERF\_GLOBAL\_STATUS register. This is a write-only register, and individual status bits in the global status register are cleared by writing a binary one to the corresponding bit in this register. Writing zero to any bit position in this register has no effect on the uncore PMU hardware.



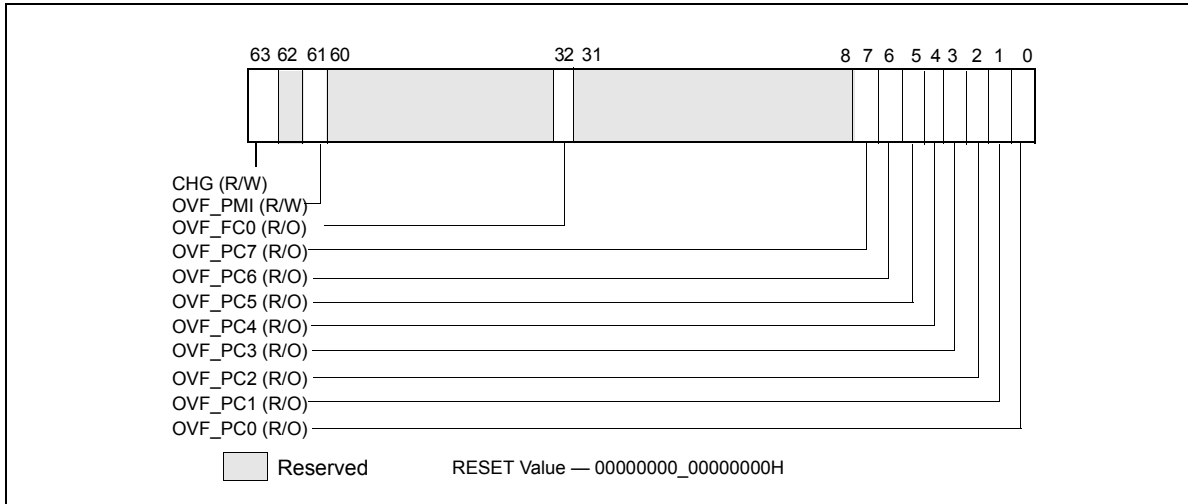


Figure 18-26. Layout of MSR\_UNCORE\_PERF\_GLOBAL\_STATUS MSR

Figure 18-27 shows the layout of MSR\_UNCORE\_PERF\_GLOBAL\_OVF\_CTRL.

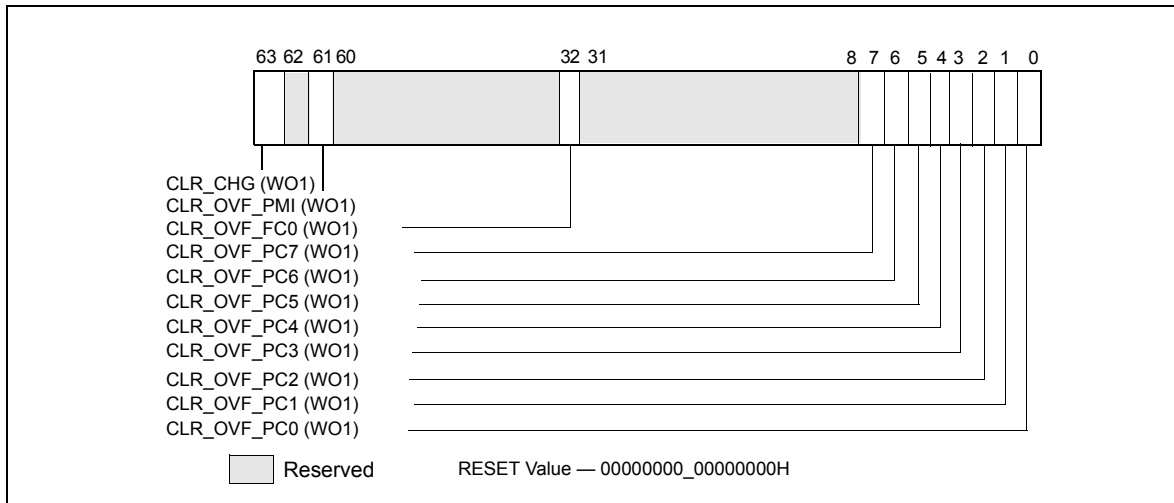


Figure 18-27. Layout of MSR\_UNCORE\_PERF\_GLOBAL\_OVF\_CTRL MSR

- CLR\_OVF\_PCn (bit n, n = 0, 7): Set this bit to clear the overflow status for general-purpose uncore counter MSR\_UNCORE\_PerfCntr n. Writing a value other than 1 is ignored.
- CLR\_OVF\_FC0 (bit 32): Set this bit to clear the overflow status for the fixed-function uncore counter MSR\_UNCORE\_FixedCntr0. Writing a value other than 1 is ignored.
- CLR\_OVF\_PMI (bit 61): Set this bit to clear the OVF\_PMI flag in MSR\_UNCORE\_PERF\_GLOBAL\_STATUS. Writing a value other than 1 is ignored.
- CLR\_CHG (bit 63): Set this bit to clear the CHG flag in MSR\_UNCORE\_PERF\_GLOBAL\_STATUS register. Writing a value other than 1 is ignored.

### 18.8.2.2 Uncore Performance Event Configuration Facility

MSR\_UNCORE\_PerfEvtSel0 through MSR\_UNCORE\_PerfEvtSel7 are used to select performance event and configure the counting behavior of the respective uncore performance counter. Each uncore PerfEvtSel MSR is paired with an uncore performance counter. Each uncore counter must be locally configured using the corresponding MSR\_UNCORE\_PerfEvtSelx and counting must be enabled using the respective EN\_PCx bit in MSR\_UNCORE\_PERF\_GLOBAL\_CTRL. Figure 18-28 shows the layout of MSR\_UNCORE\_PERFEVTSELx.

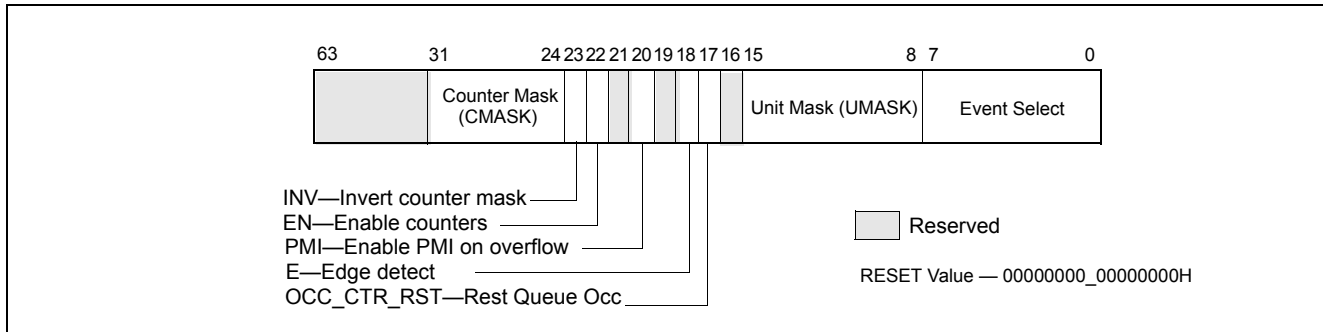


Figure 18-28. Layout of MSR\_UNCORE\_PERFEVTSELx MSRs

- Event Select (bits 7:0): Selects the event logic unit used to detect uncore events.
- Unit Mask (bits 15:8) : Condition qualifiers for the event selection logic specified in the Event Select field.
- OCC\_CTR\_RST (bit17): When set causes the queue occupancy counter associated with this event to be cleared (zeroed). Writing a zero to this bit will be ignored. It will always read as a zero.
- Edge Detect (bit 18): When set causes the counter to increment when a deasserted to asserted transition occurs for the conditions that can be expressed by any of the fields in this register.
- PMI (bit 20): When set, the uncore will generate an interrupt request when this counter overflowed. This request will be routed to the logical processors as enabled in the PMI enable bits (EN\_PMI\_COREx) in the register MSR\_UNCORE\_PERF\_GLOBAL\_CTRL.
- EN (bit 22): When clear, this counter is locally disabled. When set, this counter is locally enabled and counting starts when the corresponding EN\_PCx bit in MSR\_UNCORE\_PERF\_GLOBAL\_CTRL is set.
- INV (bit 23): When clear, the Counter Mask field is interpreted as greater than or equal to. When set, the Counter Mask field is interpreted as less than.
- Counter Mask (bits 31:24): When this field is clear, it has no effect on counting. When set to a value other than zero, the logical processor compares this field to the event counts on each core clock cycle. If INV is clear and the event counts are greater than or equal to this field, the counter is incremented by one. If INV is set and the event counts are less than this field, the counter is incremented by one. Otherwise the counter is not incremented.

Figure 18-29 shows the layout of MSR\_UNCORE\_FIXED\_CTR\_CTRL.

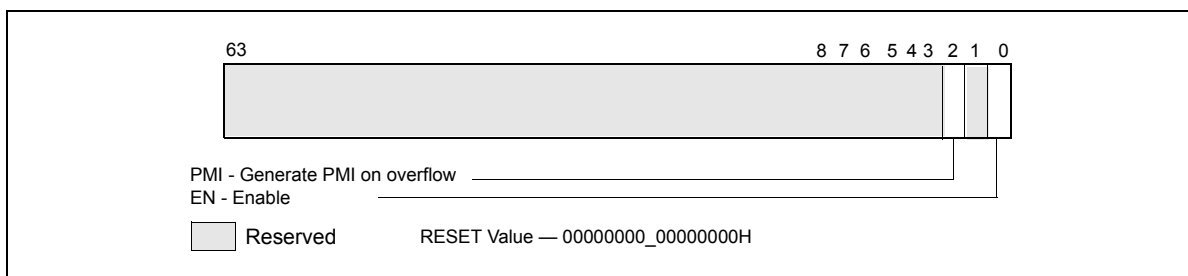


Figure 18-29. Layout of MSR\_UNCORE\_FIXED\_CTR\_CTRL MSR

- EN (bit 0): When clear, the uncore fixed-function counter is locally disabled. When set, it is locally enabled and counting starts when the EN\_FC0 bit in MSR\_UNCORE\_PERF\_GLOBAL\_CTRL is set.
- PMI (bit 2): When set, the uncore will generate an interrupt request when the uncore fixed-function counter overflowed. This request will be routed to the logical processors as enabled in the PMI enable bits (EN\_PMI\_COREx) in the register MSR\_UNCORE\_PERF\_GLOBAL\_CTRL.

Both the general-purpose counters (MSR\_UNCORE\_PerfCnt) and the fixed-function counter (MSR\_UNCORE\_FixedCnt0) are 48 bits wide. They support both counting and interrupt based sampling usages. The event logic unit can filter event counts to specific regions of code or transaction types incoming to the home node logic.

### 18.8.2.3 Uncore Address/Opcode Match MSR

The Event Select field [7:0] of MSR\_UNCORE\_PERFEVTSELx is used to select different uncore event logic unit. When the event "ADDR\_OPCODE\_MATCH" is selected in the Event Select field, software can filter uncore performance events according to transaction address and certain transaction responses. The address filter and transaction response filtering requires the use of MSR\_UNCORE\_ADDR\_OPCODE\_MATCH register. The layout is shown in Figure 18-30.

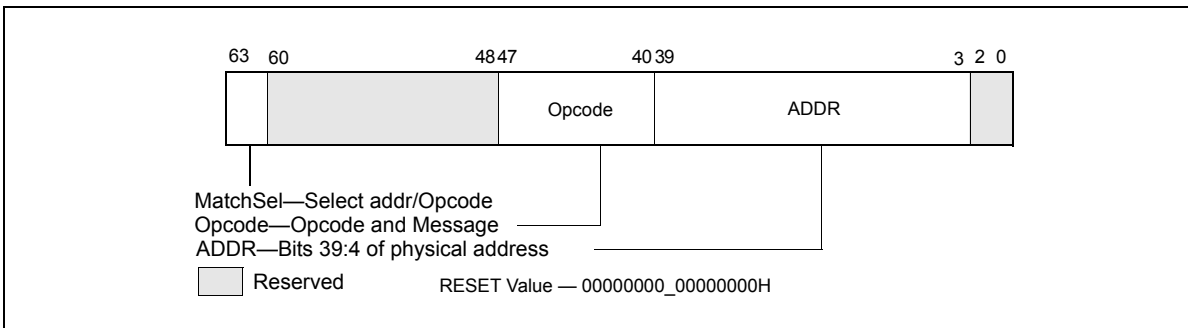


Figure 18-30. Layout of MSR\_UNCORE\_ADDR\_OPCODE\_MATCH MSR

- Addr (bits 39:3): The physical address to match if "MatchSel" field is set to select address match. The uncore performance counter will increment if the lowest 40-bit incoming physical address (excluding bits 2:0) for a transaction request matches bits 39:3.
- Opcode (bits 47:40) : Bits 47:40 allow software to filter uncore transactions based on QPI link message class/packed header opcode. These bits are consists two sub-fields:
  - Bits 43:40 specify the QPI packet header opcode,
  - Bits 47:44 specify the QPI message classes.

Table 18-27 lists the encodings supported in the opcode field.

**Table 18-27. Opcode Field Encoding for MSR\_UNCORE\_ADDR\_OPCODE\_MATCH**

Opcode [43:40]	QPI Message Class		
	Home Request [47:44] = 0000B	Snoop Response [47:44] = 0001B	Data Response [47:44] = 1110B
		1	
DMND_IFETCH	2	2	
WB	3	3	
PF_DATA_RD	4	4	
PF_RFO	5	5	
PF_IFETCH	6	6	
OTHER	7	7	
NON_DRAM	15	15	

- MatchSel (bits 63:61): Software specifies the match criteria according to the following encoding:
  - 000B: Disable addr\_opcode match hardware
  - 100B: Count if only the address field matches,
  - 010B: Count if only the opcode field matches
  - 110B: Count if either opcode field matches or the address field matches
  - 001B: Count only if both opcode and address field match
  - Other encoding are reserved

### 18.8.3 Intel® Xeon® Processor 7500 Series Performance Monitoring Facility

The performance monitoring facility in the processor core of Intel® Xeon® processor 7500 series are the same as those supported in Intel Xeon processor 5500 series. The uncore subsystem in Intel Xeon processor 7500 series are significantly different. The uncore performance monitoring facility consist of many distributed units associated with individual logic control units (referred to as boxes) within the uncore subsystem. A high level block diagram of the various box units of the uncore is shown in Figure 18-31.

Uncore PMUs are programmed via MSR interfaces. Each of the distributed uncore PMU units have several general-purpose counters. Each counter requires an associated event select MSR, and may require additional MSRs to configure sub-event conditions. The uncore PMU MSRs associated with each box can be categorized based on its functional scope: per-counter, per-box, or global across the uncore. The number counters available in each box type are different. Each box generally provides a set of MSRs to enable/disable, check status/overflow of multiple counters within each box.

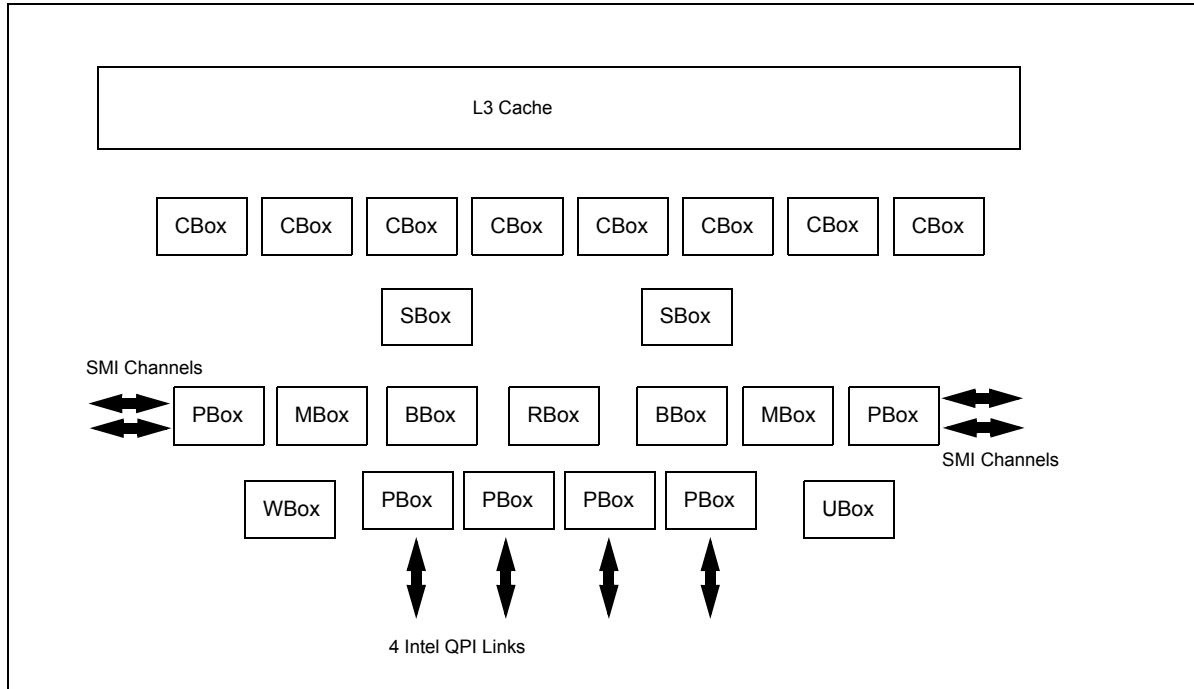


Figure 18-31. Distributed Units of the Uncore of Intel® Xeon® Processor 7500 Series

Table 18-28 summarizes the number MSRs for uncore PMU for each box.

Table 18-28. Uncore PMU MSR Summary

Box	# of Boxes	Counters per Box	Counter Width	General Purpose	Global Enable	Sub-control MSRs
C-Box	8	6	48	Yes	per-box	None
S-Box	2	4	48	Yes	per-box	Match/Mask
B-Box	2	4	48	Yes	per-box	Match/Mask
M-Box	2	6	48	Yes	per-box	Yes
R-Box	1	16 ( 2 port, 8 per port)	48	Yes	per-box	Yes
W-Box	1	4	48	Yes	per-box	None
		1	48	No	per-box	None
U-Box	1	1	48	Yes	uncore	None

The W-Box provides 4 general-purpose counters, each requiring an event select configuration MSR, similar to the general-purpose counters in other boxes. There is also a fixed-function counter that increments clockticks in the uncore clock domain.

For C,S,B,M,R, and W boxes, each box provides an MSR to enable/disable counting, configuring PMI of multiple counters within the same box, this is somewhat similar the “global control” programming interface, IA32\_PERF\_GLOBAL\_CTRL, offered in the core PMU. Similarly status information and counter overflow control for multiple counters within the same box are also provided in C,S,B,M,R, and W boxes.

In the U-Box, MSR\_U\_PMON\_GLOBAL\_CTL provides overall uncore PMU enable/disable and PMI configuration control. The scope of status information in the U-box is at per-box granularity, in contrast to the per-box status information MSR (in the C,S,B,M,R, and W boxes) providing status information of individual counter overflow. The difference in scope also apply to the overflow control MSR in the U-Box versus those in the other Boxes.

The individual MSRs that provide uncore PMU interfaces are listed in Chapter 35, Table 35-15 under the general naming style of MSR\_%box#%\_PMON\_%scope\_function%, where %box#% designates the type of box and zero-based index if there are more than one box of the same type, %scope\_function% follows the examples below:

- Multi-counter enabling MSRs: MSR\_U\_PMON\_GLOBAL\_CTL, MSR\_S0\_PMON\_BOX\_CTL, MSR\_C7\_PMON\_BOX\_CTL, etc.
- Multi-counter status MSRs: MSR\_U\_PMON\_GLOBAL\_STATUS, MSR\_S0\_PMON\_BOX\_STATUS, MSR\_C7\_PMON\_BOX\_STATUS, etc.
- Multi-counter overflow control MSRs: MSR\_U\_PMON\_GLOBAL\_OVF\_CTL, MSR\_S0\_PMON\_BOX\_OVF\_CTL, MSR\_C7\_PMON\_BOX\_OVF\_CTL, etc.
- Performance counters MSRs: the scope is implicitly per counter, e.g. MSR\_U\_PMON\_CTR, MSR\_S0\_PMON\_CTR0, MSR\_C7\_PMON\_CTR5, etc.
- Event select MSRs: the scope is implicitly per counter, e.g. MSR\_U\_PMON\_EVNT\_SEL, MSR\_S0\_PMON\_EVNT\_SEL0, MSR\_C7\_PMON\_EVNT\_SEL5, etc.
- Sub-control MSRs: the scope is implicitly per-box granularity, e.g. MSR\_M0\_PMON\_TIMESTAMP, MSR\_R0\_PMON\_IPERF0\_P1, MSR\_S1\_PMON\_MATCH.

Details of uncore PMU MSR bit field definitions can be found in a separate document "Intel Xeon Processor 7500 Series Uncore Performance Monitoring Guide".

### 18.8.4 Performance Monitoring for Processors Based on Intel® Microarchitecture Code Name Westmere

All of the performance monitoring programming interfaces (architectural and non-architectural core PMU facilities, and uncore PMU) described in Section 18.8 also apply to processors based on Intel® microarchitecture code name Westmere.

Table 18-25 describes a non-architectural performance monitoring event (event code 0B7H) and associated MSR\_OFFCORE\_RSP\_0 (address 1A6H) in the core PMU. This event and a second functionally equivalent offcore response event using event code 0BBH and MSR\_OFFCORE\_RSP\_1 (address 1A7H) are supported in processors based on Intel microarchitecture code name Westmere. The event code and event mask definitions of Non-architectural performance monitoring events are listed in Table 19-26.

The load latency facility is the same as described in Section 18.8.1.2, but added enhancement to provide more information in the data source encoding field of each load latency record. The additional information relates to STLB\_MISS and LOCK, see Table 18-33.

### 18.8.5 Intel® Xeon® Processor E7 Family Performance Monitoring Facility

The performance monitoring facility in the processor core of the Intel® Xeon® processor E7 family is the same as those supported in the Intel Xeon processor 5600 series<sup>3</sup>. The uncore subsystem in the Intel Xeon processor E7 family is similar to those of the Intel Xeon processor 7500 series. The high level construction of the uncore subsystem is similar to that shown in Figure 18-31, with the additional capability that up to 10 C-Box units are supported.

Table 18-29 summarizes the number MSRs for uncore PMU for each box.

**Table 18-29. Uncore PMU MSR Summary for Intel® Xeon® Processor E7 Family**

Box	# of Boxes	Counters per Box	Counter Width	General Purpose	Global Enable	Sub-control MSRs
C-Box	10	6	48	Yes	per-box	None
S-Box	2	4	48	Yes	per-box	Match/Mask

3. Exceptions are indicated for event code 0FH in Table 19-19; and valid bits of data source encoding field of each load latency record is limited to bits 5:4 of Table 18-33.

**Table 18-29. Uncore PMU MSR Summary for Intel® Xeon® Processor E7 Family**

Box	# of Boxes	Counters per Box	Counter Width	General Purpose	Global Enable	Sub-control MSRs
B-Box	2	4	48	Yes	per-box	Match/Mask
M-Box	2	6	48	Yes	per-box	Yes
R-Box	1	16 ( 2 port, 8 per port)	48	Yes	per-box	Yes
W-Box	1	4	48	Yes	per-box	None
		1	48	No	per-box	None
U-Box	1	1	48	Yes	uncore	None

Details of the uncore performance monitoring facility of Intel Xeon Processor E7 family is available in the “Intel® Xeon® Processor E7 Uncore Performance Monitoring Programming Reference Manual”.

## 18.9 PERFORMANCE MONITORING FOR PROCESSORS BASED ON INTEL® MICROARCHITECTURE CODE NAME SANDY BRIDGE

Intel® Core™ i7-2xxx, Intel® Core™ i5-2xxx, Intel® Core™ i3-2xxx processor series, and Intel® Xeon® processor E3-1200 family are based on Intel microarchitecture code name Sandy Bridge; this section describes the performance monitoring facilities provided in the processor core. The core PMU supports architectural performance monitoring capability with version ID 3 (see Section 18.2.3) and a host of non-architectural monitoring capabilities.

Architectural performance monitoring version 3 capabilities are described in Section 18.2.3.

The core PMU’s capability is similar to those described in Section 18.8.1 and Section 18.8.4, with some differences and enhancements relative to Intel microarchitecture code name Westmere summarized in Table 18-30.

**Table 18-30. Core PMU Comparison**

Box	Intel® microarchitecture code name Sandy Bridge	Intel® microarchitecture code name Westmere	Comment
# of Fixed counters per thread	3	3	Use CPUID to enumerate # of counters.
# of general-purpose counters per core	8	8	
Counter width (R,W)	R:48, W: 32/48	R:48, W:32	See Section 18.2.2.
# of programmable counters per thread	4 or (8 if a core not shared by two threads)	4	Use CPUID to enumerate # of counters.
PMI Overhead Mitigation	<ul style="list-style-type: none"> <li>▪ Freeze_Perfmon_on_PMI with legacy semantics.</li> <li>▪ Freeze_on_LBR with legacy semantics for branch profiling.</li> <li>▪ Freeze_while_SMM.</li> </ul>	<ul style="list-style-type: none"> <li>▪ Freeze_Perfmon_on_PMI with legacy semantics.</li> <li>▪ Freeze_on_LBR with legacy semantics for branch profiling.</li> <li>▪ Freeze_while_SMM.</li> </ul>	See Section 17.4.7.
Processor Event Based Sampling (PEBS) Events	See Table 18-32.	See Table 18-10.	IA32_PMC4-IA32_PMC7 do not support PEBS.
PEBS-Load Latency	See Section 18.9.4.2; <ul style="list-style-type: none"> <li>▪ Data source encoding</li> <li>▪ STLB miss encoding</li> <li>▪ Lock transaction encoding</li> </ul>	Data source encoding	
PEBS-Precise Store	Section 18.9.4.3	No	

Table 18-30. Core PMU Comparison (Contd.)

Box	Intel® microarchitecture code name Sandy Bridge	Intel® microarchitecture code name Westmere	Comment
PEBS-PDIR	Yes (using precise INST_RETIRED.ALL).	No	
Off-core Response Event	MSR 1A6H and 1A7H, extended request and response types.	MSR 1A6H and 1A7H, limited response types.	Nehalem supports 1A6H only.

### 18.9.1 Global Counter Control Facilities In Intel® Microarchitecture Code Name Sandy Bridge

The number of general-purpose performance counters visible to a logical processor can vary across Processors based on Intel microarchitecture code name Sandy Bridge. Software must use CPUID to determine the number performance counters/event select registers (See Section 18.2.1.1).

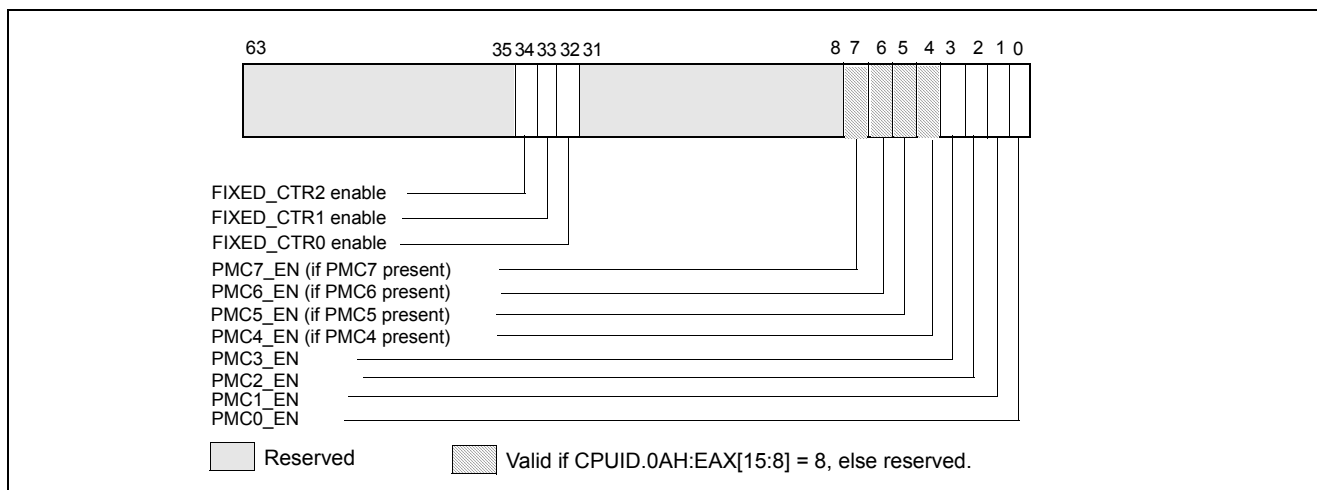


Figure 18-32. IA32\_PERF\_GLOBAL\_CTRL MSR in Intel® Microarchitecture Code Name Sandy Bridge

Figure 18-15 depicts the layout of IA32\_PERF\_GLOBAL\_CTRL MSR. The enable bits (PMC4\_EN, PMC5\_EN, PMC6\_EN, PMC7\_EN) corresponding to IA32\_PMC4-IA32\_PMC7 are valid only if CPUID.0AH:EAX[15:8] reports a value of '8'. If CPUID.0AH:EAX[15:8] = 4, attempts to set the invalid bits will cause #GP.

Each enable bit in IA32\_PERF\_GLOBAL\_CTRL is AND'ed with the enable bits for all privilege levels in the respective IA32\_PERFEVTSELx or IA32\_PERF\_FIXED\_CTR\_CTRL MSRs to start/stop the counting of respective counters. Counting is enabled if the AND'ed results is true; counting is disabled when the result is false.

IA32\_PERF\_GLOBAL\_STATUS MSR provides single-bit status used by software to query the overflow condition of each performance counter. IA32\_PERF\_GLOBAL\_STATUS[bit 62] indicates overflow conditions of the DS area data buffer (see Figure 18-33). A value of 1 in each bit of the PMCx\_OVF field indicates an overflow condition has occurred in the associated counter.



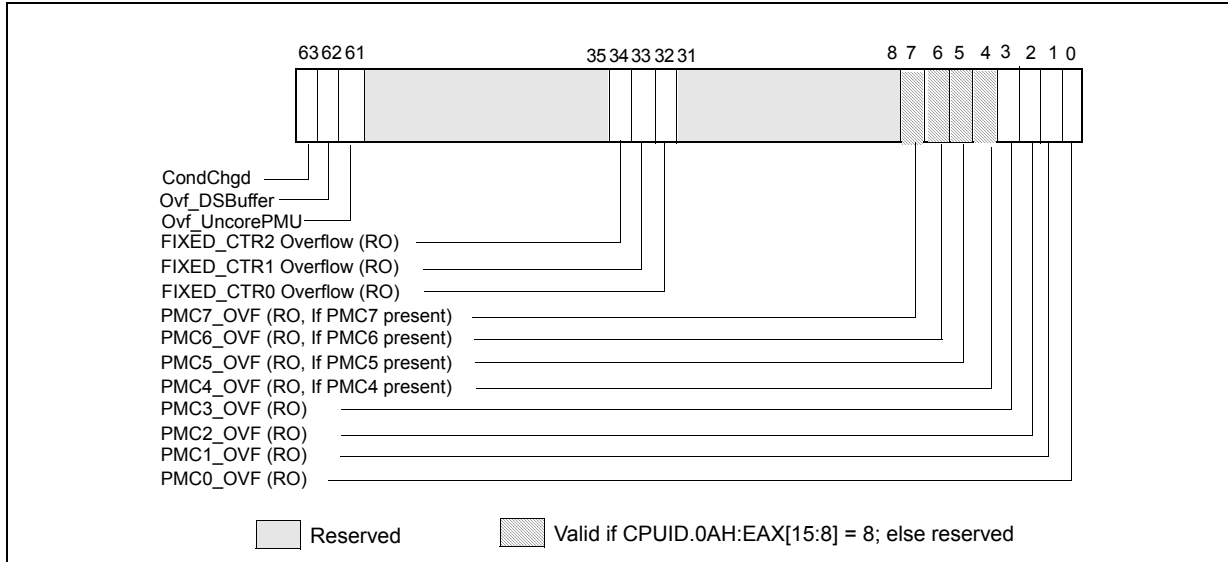


Figure 18-33. IA32\_PERF\_GLOBAL\_STATUS MSR in Intel® Microarchitecture Code Name Sandy Bridge

When a performance counter is configured for PEBS, an overflow condition in the counter will arm PEBS. On the subsequent event following overflow, the processor will generate a PEBS event. On a PEBS event, the processor will perform bounds checks based on the parameters defined in the DS Save Area (see Section 17.4.9). Upon successful bounds checks, the processor will store the data record in the defined buffer area, clear the counter overflow status, and reload the counter. If the bounds checks fail, the PEBS will be skipped entirely. In the event that the PEBS buffer fills up, the processor will set the OvfBuffer bit in MSR\_PERF\_GLOBAL\_STATUS.

IA32\_PERF\_GLOBAL\_OVF\_CTL MSR allows software to clear overflow the indicators for general-purpose or fixed-function counters via a single WRMSR (see Figure 18-34). Clear overflow indications when:

- Setting up new values in the event select and/or UMASK field for counting or interrupt based sampling
- Reloading counter values to continue sampling
- Disabling event counting or interrupt based sampling

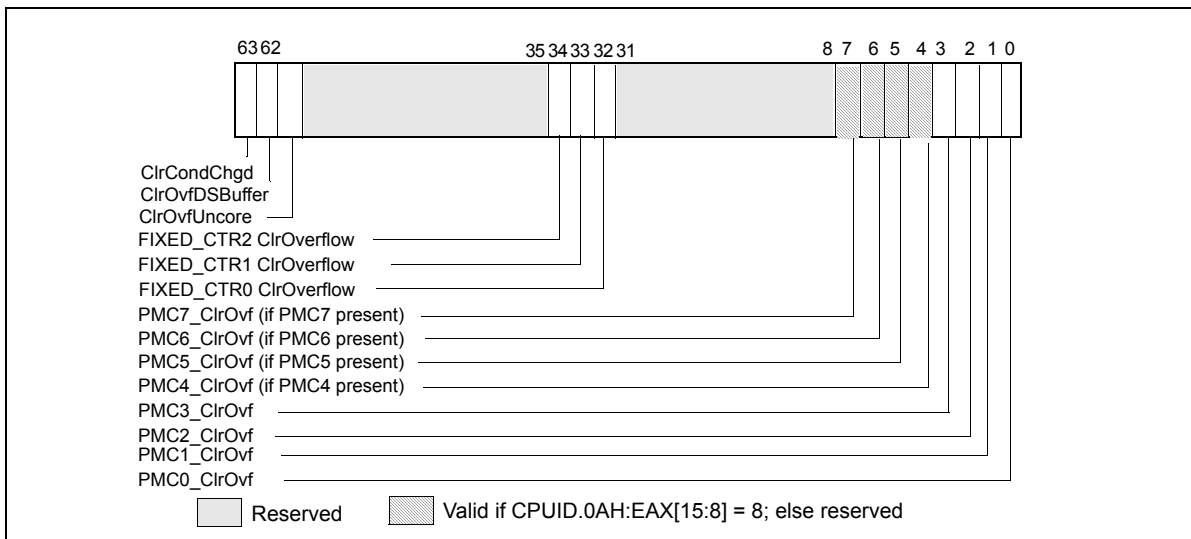


Figure 18-34. IA32\_PERF\_GLOBAL\_OVF\_CTRL MSR in Intel microarchitecture code name Sandy Bridge

## 18.9.2 Counter Coalescence

In processors based on Intel microarchitecture code name Sandy Bridge, each processor core implements eight general-purpose counters. CPUID.0AH:EAX[15:8] will report either 4 or 8 depending specific processor's product features.

If a processor core is shared by two logical processors, each logical processors can access 4 counters (IA32\_PMC0-IA32\_PMC3). This is the same as in the prior generation for processors based on Intel microarchitecture code name Nehalem.

If a processor core is not shared by two logical processors, all eight general-purpose counters are visible, and CPUID.0AH:EAX[15:8] reports 8. IA32\_PMC4-IA32\_PMC7 occupy MSR addresses 0C5H through 0C8H. Each counter is accompanied by an event select MSR (IA32\_PERFEVTSEL4-IA32\_PERFEVTSEL7).

If CPUID.0AH:EAX[15:8] report 4, access to IA32\_PMC4-IA32\_PMC7, IA32\_PMC4-IA32\_PMC7 will cause #GP. Writing 1's to bit position 7:4 of IA32\_PERF\_GLOBAL\_CTRL, IA32\_PERF\_GLOBAL\_STATUS, or IA32\_PERF\_GLOBAL\_OVF\_CTL will also cause #GP.

## 18.9.3 Full Width Writes to Performance Counters

Processors based on Intel microarchitecture code name Sandy Bridge support full-width writes to the general-purpose counters, IA32\_PMCx. Support of full-width writes are enumerated by IA32\_PERF\_CAPABILITIES.FW\_WRITES[13] (see Section 18.2.4).

The default behavior of IA32\_PMCx is unchanged, i.e. WRMSR to IA32\_PMCx results in a sign-extended 32-bit value of the input EAX written into IA32\_PMCx. Full-width writes must issue WRMSR to a dedicated alias MSR address for each IA32\_PMCx.

Software must check the presence of full-width write capability and the presence of the alias address IA32\_A\_PMCx by testing IA32\_PERF\_CAPABILITIES[13].

## 18.9.4 PEBS Support in Intel® Microarchitecture Code Name Sandy Bridge

Processors based on Intel microarchitecture code name Sandy Bridge support PEBS, similar to those offered in prior generation, with several enhanced features. The key components and differences of PEBS facility relative to Intel microarchitecture code name Westmere is summarized in Table 18-31.

**Table 18-31. PEBS Facility Comparison**

Box	Intel® microarchitecture code name Sandy Bridge	Intel® microarchitecture code name Westmere	Comment
Valid IA32_PMCx	PMC0-PMC3	PMC0-PMC3	No PEBS on PMC4-PMC7.
PEBS Buffer Programming	Section 18.8.1.1	Section 18.8.1.1	Unchanged
IA32_PEBS_ENABLE Layout	Figure 18-35	Figure 18-21	
PEBS record layout	Physical Layout same as Table 18-23.	Table 18-23	Enhanced fields at offsets 98H, A0H, A8H.
PEBS Events	See Table 18-32.	See Table 18-10.	IA32_PMC4-IA32_PMC7 do not support PEBS.
PEBS-Load Latency	See Table 18-33.	Table 18-24	
PEBS-Precise Store	Yes; see Section 18.9.4.3.	No	IA32_PMC3 only
PEBS-PDIR	Yes	No	IA32_PMC1 only
PEBS skid from EventingIP	1 (or 2 if micro+macro fusion)	1	
SAMPLING Restriction	Small SAV(CountDown) value incur higher overhead than prior generation.		

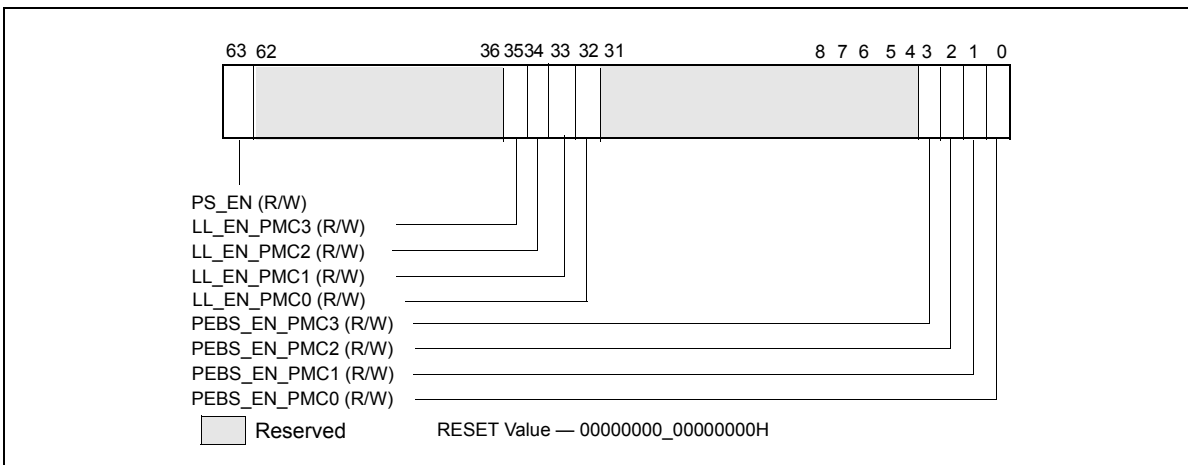
Only IA32\_PMC0 through IA32\_PMC3 support PEBS.

**NOTE**

PEBS events are only valid when the following fields of IA32\_PERFEVTSELx are all zero: AnyThread, Edge, Invert, CMask.

In a PMU with PDIR capability, PEBS behavior is unpredictable if IA32\_PERFEVTSELx or IA32\_PMCx is changed for a PEBS-enabled counter while an event is being counted. To avoid this, changes to the programming or value of a PEBS-enabled counter should be performed when the counter is disabled.

In IA32\_PEBS\_ENABLE MSR, bit 63 is defined as PS\_ENABLE: When set, this enables IA32\_PMC3 to capture precise store information. Only IA32\_PMC3 supports the precise store facility. In typical usage of PEBS, the bit fields in IA32\_PEBS\_ENABLE are written to when the agent software starts PEBS operation; the enabled bit fields should be modified only when re-programming another PEBS event or cleared when the agent uses the performance counters for non-PEBS operations.



**Figure 18-35. Layout of IA32\_PEBS\_ENABLE MSR**

**18.9.4.1 PEBS Record Format**

The layout of PEBS records physically identical to those shown in Table 18-23, but the fields at offset 98H, A0H and A8H have been enhanced to support additional PEBS capabilities.

- Load/Store Data Linear Address (Offset 98H): This field will contain the linear address of the source of the load, or linear address of the destination of the store.
- Data Source /Store Status (Offset A0H): When load latency is enabled, this field will contain three piece of information (including an encoded value indicating the source which satisfied the load operation). The source field encodings are detailed in Table 18-24. When precise store is enabled, this field will contain information indicating the status of the store, as detailed in Table 19.
- Latency Value/0 (Offset A8H): When load latency is enabled, this field contains the latency in cycles to service the load. This field is not meaningful when precise store is enabled and will be written to zero in that case. Upon writing the PEBS record, microcode clears the overflow status bits in the IA32\_PERF\_GLOBAL\_STATUS corresponding to those counters that both overflowed and were enabled in the IA32\_PEBS\_ENABLE register. The status bits of other counters remain unaffected.

The number PEBS events has expanded. The list of PEBS events supported in Intel microarchitecture code name Sandy Bridge is shown in Table 18-32.

Table 18-32. PEBS Performance Events for Intel® Microarchitecture Code Name Sandy Bridge

Event Name	Event Select	Sub-event	UMask
INST_RETIRED	COH	PREC_DIST	01H <sup>1</sup>
UOPS_RETIRED	C2H	All	01H
		Retire_Slots	02H
BR_INST_RETIRED	C4H	Conditional	01H
		Near_Call	02H
		All_branches	04H
		Near_Return	08H
		Near_Taken	20H
BR_MISP_RETIRED	C5H	Conditional	01H
		Near_Call	02H
		All_branches	04H
		Not_Taken	10H
		Taken	20H
MEM_UOPS_RETIRED	D0H	STLB_MISS_LOADS	11H
		STLB_MISS_STORE	12H
		LOCK_LOADS	21H
		SPLIT_LOADS	41H
		SPLIT_STORES	42H
		ALL_LOADS	81H
		ALL_STORES	82H
MEM_LOAD_UOPS_RETIRED	D1H	L1_Hit	01H
		L2_Hit	02H
		L3_Hit	04H
		Hit_LFB	40H
MEM_LOAD_UOPS_LLC_HIT_RETIRED	D2H	XSNP_Miss	01H
		XSNP_Hit	02H
		XSNP_Hitm	04H
		XSNP_None	08H

**NOTES:**

1. Only available on IA32\_PMC1.

### 18.9.4.2 Load Latency Performance Monitoring Facility

The load latency facility in Intel microarchitecture code name Sandy Bridge is similar to that in prior microarchitecture. It provides software a means to characterize the average load latency to different levels of cache/memory hierarchy. This facility requires processor supporting enhanced PEBS record format in the PEBS buffer, see Table 18-23 and Section 18.9.4.1. This field measures the load latency from load's first dispatch of till final data writeback from the memory subsystem. The latency is reported for retired demand load operations and in core cycles (it accounts for re-dispatches).

To use this feature software must assure:

- One of the IA32\_PERFEVTSELx MSR is programmed to specify the event unit MEM\_TRANS\_RETIRED, and the LATENCY\_ABOVE\_THRESHOLD event mask must be specified (IA32\_PerfEvtSelX[15:0] = 1CDH). The corresponding counter IA32\_PMCx will accumulate event counts for architecturally visible loads which exceed the

programmed latency threshold specified separately in a MSR. Stores are ignored when this event is programmed. The CMASK or INV fields of the IA32\_PerfEvtSelX register used for counting load latency must be 0. Writing other values will result in undefined behavior.

- The MSR\_PEBS\_LD\_LAT\_THRESHOLD MSR is programmed with the desired latency threshold in core clock cycles. Loads with latencies greater than this value are eligible for counting and latency data reporting. The minimum value that may be programmed in this register is 3 (the minimum detectable load latency is 4 core clock cycles).
- The PEBS enable bit in the IA32\_PEBS\_ENABLE register is set for the corresponding IA32\_PMCx counter register. This means that both the PEBS\_EN\_CTRX and LL\_EN\_CTRX bits must be set for the counter(s) of interest. For example, to enable load latency on counter IA32\_PMC0, the IA32\_PEBS\_ENABLE register must be programmed with the 64-bit value 00000001.00000001H.
- When Load latency event is enabled, no other PEBS event can be configured with other counters.

When the load-latency facility is enabled, load operations are randomly selected by hardware and tagged to carry information related to data source locality and latency. Latency and data source information of tagged loads are updated internally. The MEM\_TRANS\_RETIRE event for load latency counts only tagged retired loads. If a load is cancelled it will not be counted and the internal state of the load latency facility will not be updated. In this case the hardware will tag the next available load.

When a PEBS assist occurs, the last update of latency and data source information are captured by the assist and written as part of the PEBS record. The PEBS sample after value (SAV), specified in PEBS CounterX Reset, operates orthogonally to the tagging mechanism. Loads are randomly tagged to collect latency data. The SAV controls the number of tagged loads with latency information that will be written into the PEBS record field by the PEBS assists. The load latency data written to the PEBS record will be for the last tagged load operation which retired just before the PEBS assist was invoked.

The physical layout of the PEBS records is the same as shown in Table 18-23. The specificity of Data Source entry at offset A0H has been enhanced to report three piece of information.

**Table 18-33. Layout of Data Source Field of Load Latency Record**

Field	Position	Description
Source	3:0	See Table 18-24
STLB_MISS	4	0: The load did not miss the STLB (hit the DTLB or STLB). 1: The load missed the STLB.
Lock	5	0: The load was not part of a locked transaction. 1: The load was part of a locked transaction.
Reserved	63:6	Reserved

The layout of MSR\_PEBS\_LD\_LAT\_THRESHOLD is the same as shown in Figure 18-23.

### 18.9.4.3 Precise Store Facility

Processors based on Intel microarchitecture code name Sandy Bridge offer a precise store capability that complements the load latency facility. It provides a means to profile store memory references in the system.

Precise stores leverage the PEBS facility and provide additional information about sampled stores. Having precise memory reference events with linear address information for both loads and stores can help programmers improve data structure layout, eliminate remote node references, and identify cache-line conflicts in NUMA systems.

Only IA32\_PMC3 can be used to capture precise store information. After enabling this facility, counter overflows will initiate the generation of PEBS records as previously described in PEBS. Upon counter overflow hardware captures the linear address and other status information of the next store that retires. This information is then written to the PEBS record.

To enable the precise store facility, software must complete the following steps. Please note that the precise store facility relies on the PEBS facility, so the PEBS configuration requirements must be completed before attempting to capture precise store information.

- Complete the PEBS configuration steps.
- Program the MEM\_TRANS\_RETIRED.PRECISE\_STORE event in IA32\_PERFEVTSEL3. Only counter 3 (IA32\_PMC3) supports collection of precise store information.
- Set IA32\_PEBS\_ENABLE[3] and IA32\_PEBS\_ENABLE[63]. This enables IA32\_PMC3 as a PEBS counter and enables the precise store facility, respectively.

The precise store information written into a PEBS record affects entries at offset 98H, A0H and A8H of Table 18-23. The specificity of Data Source entry at offset A0H has been enhanced to report three piece of information.

**Table 18-34. Layout of Precise Store Information In PEBS Record**

Field	Offset	Description
Store Data Linear Address	98H	The linear address of the destination of the store.
Store Status	A0H	<b>L1D Hit</b> (Bit 0): The store hit the data cache closest to the core (lowest latency cache) if this bit is set, otherwise the store missed the data cache. <b>STLB Miss</b> (bit 4): The store missed the STLB if set, otherwise the store hit the STLB <b>Locked Access</b> (bit 5): The store was part of a locked access if set, otherwise the store was not part of a locked access.
Reserved	A8H	Reserved

#### 18.9.4.4 Precise Distribution of Instructions Retired (PDIR)

Upon triggering a PEBS assist, there will be a finite delay between the time the counter overflows and when the microcode starts to carry out its data collection obligations. INST\_RETIRED is a very common event that is used to sample where performance bottleneck happened and to help identify its location in instruction address space. Even if the delay is constant in core clock space, it invariably manifest as variable “skids” in instruction address space. This creates a challenge for programmers to profile a workload and pinpoint the location of bottlenecks.

The core PMU in processors based on Intel microarchitecture code name Sandy Bridge include a facility referred to as precise distribution of Instruction Retired (PDIR).

The PDIR facility mitigates the “skid” problem by providing an early indication of when the INST\_RETIRED counter is about to overflow, allowing the machine to more precisely trap on the instruction that actually caused the counter overflow thus eliminating skid.

PDIR applies only to the INST\_RETIRED.ALL precise event, and must use IA32\_PMC1 with PerfEvtSel1 property configured and bit 1 in the IA32\_PEBS\_ENABLE set to 1. INST\_RETIRED.ALL is a non-architectural performance event, it is not supported in prior generation microarchitectures. Additionally, on processors with CPUID DisplayFamily\_DisplayModel signatures of 06\_2A and 06\_2D, the tool that programs PDIR should quiesce the rest of the programmable counters in the core when PDIR is active.

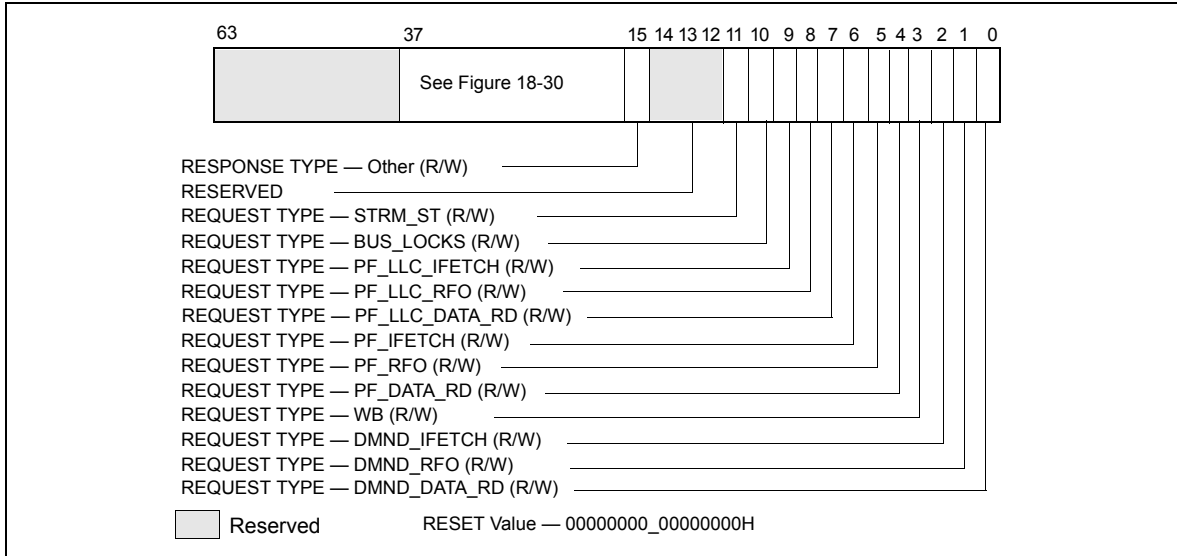
#### 18.9.5 Off-core Response Performance Monitoring

The core PMU in processors based on Intel microarchitecture code name Sandy Bridge provides off-core response facility similar to prior generation. Off-core response can be programmed only with a specific pair of event select and counter MSR, and with specific event codes and predefine mask bit value in a dedicated MSR to specify attributes of the off-core transaction. Two event codes are dedicated for off-core response event programming. Each event code for off-core response monitoring requires programming an associated configuration MSR, MSR\_OFFCORE\_RSP\_x. Table 18-35 lists the event code, mask value and additional off-core configuration MSR that must be programmed to count off-core response events using IA32\_PMCx.

**Table 18-35. Off-Core Response Event Encoding**

Counter	Event code	UMask	Required Off-core Response MSR
PMCO-3	B7H	01H	MSR_OFFCORE_RSP_0 (address 1A6H)
PMCO-3	BBH	01H	MSR_OFFCORE_RSP_1 (address 1A7H)

The layout of MSR\_OFFCORE\_RSP\_0 and MSR\_OFFCORE\_RSP\_1 are shown in Figure 18-36 and Figure 18-37. Bits 15:0 specifies the request type of a transaction request to the uncore. Bits 30:16 specifies supplier information, bits 37:31 specifies snoop response information.



**Figure 18-36. Request\_Type Fields for MSR\_OFFCORE\_RSP\_x**

**Table 18-36. MSR\_OFFCORE\_RSP\_x Request\_Type Field Definition**

Bit Name	Offset	Description
DMND_DATA_RD	0	(R/W). Counts the number of demand data reads of full and partial cachelines as well as demand data page table entry cacheline reads. Does not count L2 data read prefetches or instruction fetches.
DMND_RFO	1	(R/W). Counts the number of demand and DCU prefetch reads for ownership (RFO) requests generated by a write to data cacheline. Does not count L2 RFO prefetches.
DMND_IFETCH	2	(R/W). Counts the number of demand and DCU prefetch instruction cacheline reads. Does not count L2 code read prefetches.
WB	3	(R/W). Counts the number of writeback (modified to exclusive) transactions.
PF_DATA_RD	4	(R/W). Counts the number of data cacheline reads generated by L2 prefetchers.
PF_RFO	5	(R/W). Counts the number of RFO requests generated by L2 prefetchers.
PF_IFETCH	6	(R/W). Counts the number of code reads generated by L2 prefetchers.
PF_LLC_DATA_RD	7	(R/W). L2 prefetcher to L3 for loads.
PF_LLC_RFO	8	(R/W). RFO requests generated by L2 prefetcher
PF_LLC_IFETCH	9	(R/W). L2 prefetcher to L3 for instruction fetches.
BUS_LOCKS	10	(R/W). Bus lock and split lock requests
STRM_ST	11	(R/W). Streaming store requests
OTHER	15	(R/W). Any other request that crosses IDI, including I/O.

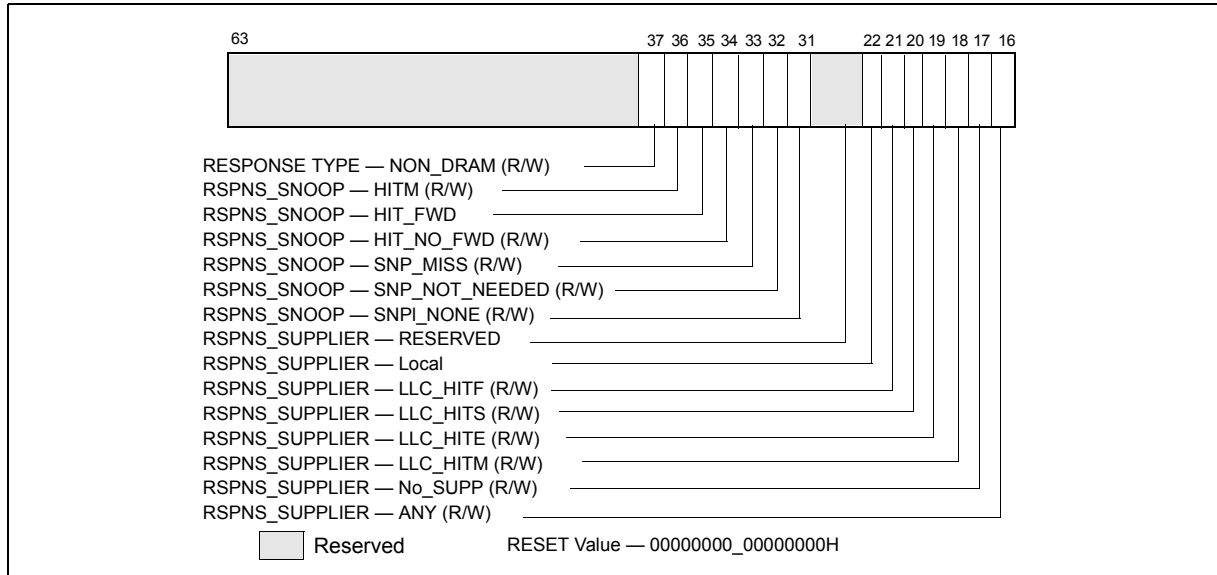


Figure 18-37. Response\_Supplier and Snoop Info Fields for MSR\_OFFCORE\_RSP\_x

To properly program this extra register, software must set at least one request type bit and a valid response type pattern. Otherwise, the event count reported will be zero. It is permissible and useful to set multiple request and response type bits in order to obtain various classes of off-core response events. Although MSR\_OFFCORE\_RSP\_x allow an agent software to program numerous combinations that meet the above guideline, not all combinations produce meaningful data.

Table 18-37. MSR\_OFFCORE\_RSP\_x Response Supplier Info Field Definition

Subtype	Bit Name	Offset	Description
Common	Any	16	(R/W). Catch all value for any response types.
Supplier Info	NO_SUPP	17	(R/W). No Supplier Information available
	LLC_HITM	18	(R/W). M-state initial lookup stat in L3.
	LLC_HITE	19	(R/W). E-state
	LLC_HITS	20	(R/W). S-state
	LLC_HITF	21	(R/W). F-state
	LOCAL	22	(R/W). Local DRAM Controller
	Reserved	30:23	Reserved

To specify a complete offcore response filter, software must properly program bits in the request and response type fields. A valid request type must have at least one bit set in the non-reserved bits of 15:0. A valid response type must be a non-zero value of the following expression:

ANY | [(‘OR’ of Supplier Info Bits) & (‘OR’ of Snoop Info Bits)]

If “ANY” bit is set, the supplier and snoop info bits are ignored.

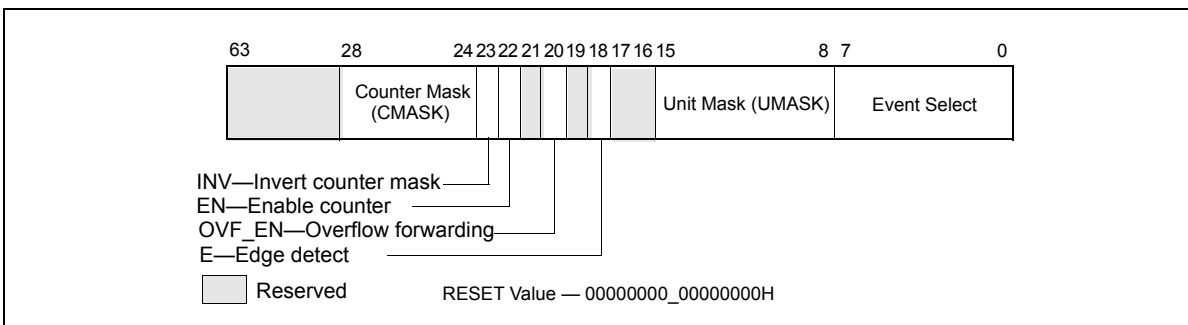


**Table 18-38. MSR\_OFFCORE\_RSP\_x Snoop Info Field Definition**

Subtype	Bit Name	Offset	Description
Snoop Info	SNP_NONE	31	(R/W). No details on snoop-related information
	SNP_NOT_NEEDED	32	(R/W). No snoop was needed to satisfy the request.
	SNP_MISS	33	(R/W). A snoop was needed and it missed all snooped caches: -For LLC Hit, ReslHitl was returned by all cores -For LLC Miss, Rspl was returned by all sockets and data was returned from DRAM.
	SNP_NO_FWD	34	(R/W). A snoop was needed and it hits in at least one snooped cache. Hit denotes a cache-line was valid before snoop effect. This includes: -Snoop Hit w/ Invalidation (LLC Hit, RFO) -Snoop Hit, Left Shared (LLC Hit/Miss, IFetch/Data_RD) -Snoop Hit w/ Invalidation and No Forward (LLC Miss, RFO Hit S) In the LLC Miss case, data is returned from DRAM.
	SNP_FWD	35	(R/W). A snoop was needed and data was forwarded from a remote socket. This includes: -Snoop Forward Clean, Left Shared (LLC Hit/Miss, IFetch/Data_RD/RFT).
	HITM	36	(R/W). A snoop was needed and it HitM-ed in local or remote cache. HitM denotes a cache-line was in modified state before effect as a results of snoop. This includes: -Snoop HitM w/ WB (LLC miss, IFetch/Data_RD) -Snoop Forward Modified w/ Invalidation (LLC Hit/Miss, RFO) -Snoop MtoS (LLC Hit, IFetch/Data_RD).
	NON_DRAM	37	(R/W). Target was non-DRAM system address. This includes MMIO transactions.

### 18.9.6 Uncore Performance Monitoring Facilities In Intel® Core™ i7-2xxx, Intel® Core™ i5-2xxx, Intel® Core™ i3-2xxx Processor Series

The uncore sub-system in Intel® Core™ i7-2xxx, Intel® Core™ i5-2xxx, Intel® Core™ i3-2xxx processor series provides a unified L3 that can support up to four processor cores. The L3 cache consists multiple slices, each slice interface with a processor via a coherence engine, referred to as a C-Box. Each C-Box provides dedicated facility of MSRs to select uncore performance monitoring events and each C-Box event select MSR is paired with a counter register, similar in style as those described in Section 18.8.2.2. The ARB unit in the uncore also provides its local performance counters and event select MSRs. The layout of the event select MSRs in the C-Boxes and the ARB unit are shown in Figure 18-38.



**Figure 18-38. Layout of Uncore PERFVTSSEL MSR for a C-Box Unit or the ARB Unit**

The bit fields of the uncore event select MSRs for a C-box unit or the ARB unit are summarized below:

- Event\_Select (bits 7:0) and UMASK (bits 15:8): Specifies the microarchitectural condition to count in a local uncore PMU counter, see Table 19-16.
- E (bit 18): Enables edge detection filtering, if 1.
- OVF\_EN (bit 20): Enables the overflow indicator from the uncore counter forwarded to MSR\_UNC\_PERF\_GLOBAL\_CTRL, if 1.
- EN (bit 22): Enables the local counter associated with this event select MSR.
- INV (bit 23): Event count increments with non-negative value if 0, with negated value if 1.
- CMASK (bits 28:24): Specifies a positive threshold value to filter raw event count input.

At the uncore domain level, there is a master set of control MSRs that centrally manages all the performance monitoring facility of uncore units. Figure 18-39 shows the layout of the uncore domain global control.

When an uncore counter overflows, a PMI can be routed to a processor core. Bits 3:0 of MSR\_UNC\_PERF\_GLOBAL\_CTRL can be used to select which processor core to handle the uncore PMI. Software must then write to bit 13 of IA32\_DEBUGCTL (at address 1D9H) to enable this capability.

- PMI\_SEL\_Core#: Enables the forwarding of an uncore PMI request to a processor core, if 1. If bit 30 (WakePMI) is '1', a wake request is sent to the respective processor core prior to sending the PMI.
- EN: Enables the fixed uncore counter, the ARB counters, and the CBO counters in the uncore PMU, if 1. This bit is cleared if bit 31 (FREEZE) is set and any enabled uncore counters overflow.
- WakePMI: Controls sending a wake request to any halted processor core before issuing the uncore PMI request. If a processor core was halted and not sent a wake request, the uncore PMI will not be serviced by the processor core.
- FREEZE: Provides the capability to freeze all uncore counters when an overflow condition occurs in a unit counter. When this bit is set, and a counter overflow occurs, the uncore PMU logic will clear the global enable bit (bit 29).

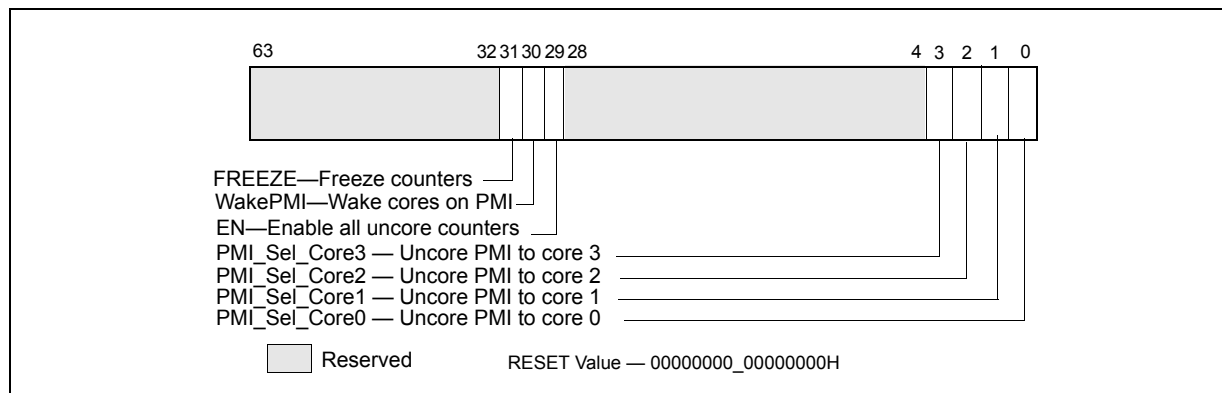


Figure 18-39. Layout of MSR\_UNC\_PERF\_GLOBAL\_CTRL MSR for Uncore

Additionally, there is also a fixed counter, counting uncore clockticks, for the uncore domain. Table 18-39 summarizes the number MSRs for uncore PMU for each box.

**Table 18-39. Uncore PMU MSR Summary**

Box	# of Boxes	Counters per Box	Counter Width	General Purpose	Global Enable	Comment
C-Box	SKU specific	2	44	Yes	Per-box	Up to 4, see Table 35-19 MSR_UNC_CBO_CONFIG
ARB	1	2	44	Yes	Uncore	
Fixed Counter	N.A.	N.A.	48	No	Uncore	

### 18.9.6.1 Uncore Performance Monitoring Events

There are certain restrictions on the uncore performance counters in each C-Box. Specifically,

- Occupancy events are supported only with counter 0 but not counter 1.

Other uncore C-Box events can be programmed with either counter 0 or 1.

The C-Box uncore performance events described in Table 19-16 can collect performance characteristics of transactions initiated by processor core. In that respect, they are similar to various sub-events in the OFFCORE\_RESPONSE family of performance events in the core PMU. Information such as data supplier locality (LLC HIT/MISS) and snoop responses can be collected via OFFCORE\_RESPONSE and qualified on a per-thread basis.

On the other hand, uncore performance event logic can not associate its counts with the same level of per-thread qualification attributes as the core PMU events can. Therefore, whenever similar event programming capabilities are available from both core PMU and uncore PMU, the recommendation is that utilizing the core PMU events may be less affected by artifacts, complex interactions and other factors.

### 18.9.7 Intel® Xeon® Processor E5 Family Performance Monitoring Facility

The Intel® Xeon® Processor E5 Family (and Intel® Core™ i7-3930K Processor) are based on Intel microarchitecture code name Sandy Bridge-E. While the processor cores share the same microarchitecture as those of the Intel® Xeon® Processor E3 Family and 2nd generation Intel Core i7-2xxx, Intel Core i5-2xxx, Intel Core i3-2xxx processor series, the uncore subsystems are different. An overview of the uncore performance monitoring facilities of the Intel Xeon processor E5 family (and Intel Core i7-3930K processor) is described in Section 18.9.8.

Thus, the performance monitoring facilities in the processor core generally are the same as those described in Section 18.9 through Section 18.9.5. However, the MSR\_OFFCORE\_RSP\_0/MSR\_OFFCORE\_RSP\_1 Response Supplier Info field shown in Table 18-37 applies to Intel Core Processors with CPUID signature of DisplayFamily\_DisplayModel encoding of 06\_2AH; Intel Xeon processor with CPUID signature of DisplayFamily\_DisplayModel encoding of 06\_2DH supports an additional field for remote DRAM controller shown in Table 18-40. Additionally, there are some small differences in the non-architectural performance monitoring events (see Table 19-14).

**Table 18-40. MSR\_OFFCORE\_RSP\_x Supplier Info Field Definitions**

Subtype	Bit Name	Offset	Description
Common	Any	16	(R/W). Catch all value for any response types.
Supplier Info	NO_SUPP	17	(R/W). No Supplier Information available
	LLC_HITM	18	(R/W). M-state initial lookup stat in L3.
	LLC_HITE	19	(R/W). E-state
	LLC_HITS	20	(R/W). S-state
	LLC_HITF	21	(R/W). F-state
	LOCAL	22	(R/W). Local DRAM Controller
	Remote	30:23	(R/W); Remote DRAM Controller (either all 0s or all 1s)

### 18.9.8 Intel® Xeon® Processor E5 Family Uncore Performance Monitoring Facility

The uncore subsystem in the Intel Xeon processor E5-2600 product family has some similarities with those of the Intel Xeon processor E7 family. Within the uncore subsystem, localized performance counter sets are provided at logic control unit scope. For example, each Cbox caching agent has a set of local performance counters, and the power controller unit (PCU) has its own local performance counters. Up to 8 C-Box units are supported in the uncore sub-system.

Table 18-41 summarizes the uncore PMU facilities providing MSR interfaces.

**Table 18-41. Uncore PMU MSR Summary for Intel® Xeon® Processor E5 Family**

Box	# of Boxes	Counters per Box	Counter Width	General Purpose	Global Enable	Sub-control MSRs
C-Box	8	4	44	Yes	per-box	None
PCU	1	4	48	Yes	per-box	Match/Mask
U-Box	1	2	44	Yes	uncore	None

Details of the uncore performance monitoring facility of Intel Xeon Processor E5 family is available in "Intel® Xeon® Processor E5 Uncore Performance Monitoring Programming Reference Manual". The MSR-based uncore PMU interfaces are listed in Table 35-22.

## 18.10 3RD GENERATION INTEL® CORE™ PROCESSOR PERFORMANCE MONITORING FACILITY

The 3rd generation Intel® Core™ processor family and Intel® Xeon® processor E3-1200v2 product family are based on the Ivy Bridge microarchitecture. The performance monitoring facilities in the processor core generally are the same as those described in Section 18.9 through Section 18.9.5. The non-architectural performance monitoring events supported by the processor core are listed in Table 19-14.

### 18.10.1 Intel® Xeon® Processor E5 v2 and E7 v2 Family Uncore Performance Monitoring Facility

The uncore subsystem in the Intel Xeon processor E5 v2 and Intel Xeon Processor E7 v2 product families are based on the Ivy Bridge-E microarchitecture. There are some similarities with those of the Intel Xeon processor E5 family based on the Sandy Bridge microarchitecture. Within the uncore subsystem, localized performance counter sets are provided at logic control unit scope.

Details of the uncore performance monitoring facility of Intel Xeon Processor E5 v2 and Intel Xeon Processor E7 v2 families are available in “Intel® Xeon® Processor E5 v2 and E7 v2 Uncore Performance Monitoring Programming Reference Manual”. The MSR-based uncore PMU interfaces are listed in Table 35-26.

## 18.11 4TH GENERATION INTEL® CORE™ PROCESSOR PERFORMANCE MONITORING FACILITY

The 4th generation Intel® Core™ processor and Intel® Xeon® processor E3-1200 v3 product family are based on the Haswell microarchitecture. The core PMU supports architectural performance monitoring capability with version ID 3 (see Section 18.2.3) and a host of non-architectural monitoring capabilities.

Architectural performance monitoring version 3 capabilities are described in Section 18.2.3.

The core PMU’s capability is similar to those described in Section 18.9 through Section 18.9.5, with some differences and enhancements summarized in Table 18-42. Additionally, the core PMU provides some enhancement to support performance monitoring when the target workload contains instruction streams using Intel® Transactional Synchronization Extensions (TSX), see Section 18.11.5. For details of Intel TSX, see Chapter 16, “Programming with Intel® Transactional Synchronization Extensions” of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*.

**Table 18-42. Core PMU Comparison**

Box	Intel® microarchitecture code name Haswell	Intel® microarchitecture code name Sandy Bridge	Comment
# of Fixed counters per thread	3	3	
# of general-purpose counters per core	8	8	
Counter width (R,W)	R:48, W: 32/48	R:48, W: 32/48	See Section 18.2.2.
# of programmable counters per thread	4 or (8 if a core not shared by two threads)	4 or (8 if a core not shared by two threads)	Use CPUID to enumerate # of counters.
PMI Overhead Mitigation	<ul style="list-style-type: none"> <li>▪ Freeze_Perfmon_on_PMI with legacy semantics.</li> <li>▪ Freeze_on_LBR with legacy semantics for branch profiling.</li> <li>▪ Freeze_while_SMM.</li> </ul>	<ul style="list-style-type: none"> <li>▪ Freeze_Perfmon_on_PMI with legacy semantics.</li> <li>▪ Freeze_on_LBR with legacy semantics for branch profiling.</li> <li>▪ Freeze_while_SMM.</li> </ul>	See Section 17.4.7.
Processor Event Based Sampling (PEBS) Events	See Table 18-32 and Section 18.11.5.1.	See Table 18-32.	IA32_PMC4-IA32_PMC7 do not support PEBS.
PEBS-Load Latency	See Section 18.9.4.2.	See Section 18.9.4.2.	
PEBS-Precise Store	No, replaced by Data Address profiling.	Section 18.9.4.3	
PEBS-PDIR	Yes (using precise INST_RETIRED.ALL)	Yes (using precise INST_RETIRED.ALL)	
PEBS-EventingIP	Yes	No	
Data Address Profiling	Yes	No	
LBR Profiling	Yes	Yes	
Call Stack Profiling	Yes, see Section 17.9.	No	Use LBR facility.
Off-core Response Event	MSR 1A6H and 1A7H; extended request and response types.	MSR 1A6H and 1A7H; extended request and response types.	
Intel TSX support for Perfmon	See Section 18.11.5.	No	

### 18.11.1 Processor Event Based Sampling (PEBS) Facility

The PEBS facility in the 4th Generation Intel Core processor is similar to those in processors based on Intel micro-architecture code name Sandy Bridge, with several enhanced features. The key components and differences of PEBS facility relative to Intel microarchitecture code name Sandy Bridge is summarized in Table 18-43.

**Table 18-43. PEBS Facility Comparison**

Box	Intel® microarchitecture code name Haswell	Intel® microarchitecture code name Sandy Bridge	Comment
Valid IA32_PMCx	PMC0-PMC3	PMC0-PMC3	No PEBS on PMC4-PMC7
PEBS Buffer Programming	Section 18.8.1.1	Section 18.8.1.1	Unchanged
IA32_PEBS_ENABLE Layout	Figure 18-21	Figure 18-35	
PEBS record layout	Table 18-44; enhanced fields at offsets 98H, A0H, A8H, B0H.	Table 18-23; enhanced fields at offsets 98H, A0H, A8H.	
Precise Events	See Table 18-32.	See Table 18-32.	IA32_PMC4-IA32_PMC7 do not support PEBS.
PEBS-Load Latency	See Table 18-33.	Table 18-33	
PEBS-Precise Store	No, replaced by data address profiling.	Yes; see Section 18.9.4.3.	
PEBS-PDIR	Yes	Yes	IA32_PMC1 only.
PEBS skid from EventingIP	1 (or 2 if micro+macro fusion)	1	
SAMPLING Restriction	Small SAV(CountDown) value incur higher overhead than prior generation.		

Only IA32\_PMC0 through IA32\_PMC3 support PEBS.

#### NOTE

PEBS events are only valid when the following fields of IA32\_PERFEVTSELx are all zero: AnyThread, Edge, Invert, CMask.

In a PMU with PDIR capability, PEBS behavior is unpredictable if IA32\_PERFEVTSELx or IA32\_PMCx is changed for a PEBS-enabled counter while an event is being counted. To avoid this, changes to the programming or value of a PEBS-enabled counter should be performed when the counter is disabled.

### 18.11.2 PEBS Data Format

The PEBS record format for the 4th Generation Intel Core processor is shown in Table 18-44. The PEBS record format, along with debug/store area storage format, does not change regardless of whether IA-32e mode is active or not. CPUID.01H:ECX.DTES64[bit 2] reports whether the processor's DS storage format support is mode-independent. When set, it uses 64-bit DS storage format.

**Table 18-44. PEBS Record Format for 4th Generation Intel Core Processor Family**

Byte Offset	Field	Byte Offset	Field
00H	R/EFLAGS	60H	R10
08H	R/EIP	68H	R11
10H	R/EAX	70H	R12
18H	R/EBX	78H	R13
20H	R/ECX	80H	R14
28H	R/EDX	88H	R15
30H	R/ESI	90H	IA32_PERF_GLOBAL_STATUS
38H	R/EDI	98H	Data Linear Address
40H	R/EBP	A0H	Data Source Encoding
48H	R/ESP	A8H	Latency value (core cycles)
50H	R8	B0H	EventingIP
58H	R9	B8H	TX Abort Information (Section 18.11.5.1)

The layout of PEBS records are almost identical to those shown in Table 18-23. Offset B0H is a new field that records the eventing IP address of the retired instruction that triggered the PEBS assist.

The PEBS records at offsets 98H, A0H, and ABH record data gathered from three of the PEBS capabilities in prior processor generations: load latency facility (Section 18.9.4.2), PDIR (Section 18.9.4.4), and the equivalent capability of precise store in prior generation (see Section 18.11.3).

In the core PMU of the 4th generation Intel Core processor, load latency facility and PDIR capabilities are unchanged. However, precise store is replaced by an enhanced capability, data address profiling, that is not restricted to store address. Data address profiling also records information in PEBS records at offsets 98H, A0H, and ABH.

### 18.11.3 PEBS Data Address Profiling

The Data Linear Address facility is also abbreviated as DataLA. The facility is a replacement or extension of the precise store facility in previous processor generations. The DataLA facility complements the load latency facility by providing a means to profile load and store memory references in the system, leverages the PEBS facility, and provides additional information about sampled loads and stores. Having precise memory reference events with linear address information for both loads and stores provides information to improve data structure layout, eliminate remote node references, and identify cache-line conflicts in NUMA systems.

The DataLA facility in the 4th generation processor supports the following events configured to use PEBS:

**Table 18-45. Precise Events That Supports Data Linear Address Profiling**

Event Name	Event Name
MEM_UOPS_RETIRED.STLB_MISS_LOADS	MEM_UOPS_RETIRED.STLB_MISS_STORES
MEM_UOPS_RETIRED.LOCK_LOADS	MEM_UOPS_RETIRED.SPLIT_STORES
MEM_UOPS_RETIRED.SPLIT_LOADS	MEM_UOPS_RETIRED.ALL_STORES
MEM_UOPS_RETIRED.ALL_LOADS	MEM_LOAD_UOPS_RETIRED.LLC_MISS_RETIRED.LOCAL_DRAM
MEM_LOAD_UOPS_RETIRED.L1_HIT	MEM_LOAD_UOPS_RETIRED.L2_HIT
MEM_LOAD_UOPS_RETIRED.L3_HIT	MEM_LOAD_UOPS_RETIRED.L1_MISS
MEM_LOAD_UOPS_RETIRED.L2_MISS	MEM_LOAD_UOPS_RETIRED.L3_MISS
MEM_LOAD_UOPS_RETIRED.HIT_LFB	MEM_LOAD_UOPS_L3_HIT_RETIRED.XSNP_MISS

**Table 18-45. Precise Events That Supports Data Linear Address Profiling (Contd.)**

Event Name	Event Name
MEM_LOAD_UOPS_L3_HIT_RETIRED.XSNP_HIT	MEM_LOAD_UOPS_L3_HIT_RETIRED.XSNP_HITM
UOPS_RETIRED.ALL (if load or store is tagged)	MEM_LOAD_UOPS_LLC_HIT_RETIRED.XSNP_NONE

DataLA can use any one of the IA32\_PMC0-IA32\_PMC3 counters. Counter overflows will initiate the generation of PEBS records. Upon counter overflow, hardware captures the linear address and possible other status information of the retiring memory uop. This information is then written to the PEBS record that is subsequently generated.

To enable the DataLA facility, software must complete the following steps. Please note that the DataLA facility relies on the PEBS facility, so the PEBS configuration requirements must be completed before attempting to capture DataLA information.

- Complete the PEBS configuration steps.
- Program the an event listed in Table 18-45 using any one of IA32\_PERFEVTSEL0-IA32\_PERFEVTSEL3.
- Set the corresponding IA32\_PEBS\_ENABLE.PEBS\_EN\_CTRx bit. This enables the corresponding IA32\_PMCx as a PEBS counter and enables the DataLA facility.

When the DataLA facility is enabled, the relevant information written into a PEBS record affects entries at offsets 98H, A0H and A8H, as shown in Table 18-46.

**Table 18-46. Layout of Data Linear Address Information In PEBS Record**

Field	Offset	Description
Data Linear Address	98H	The linear address of the load or the destination of the store.
Store Status	A0H	<ul style="list-style-type: none"> <li>▪ <b>DCU Hit</b> (Bit 0): The store hit the data cache closest to the core (L1 cache) if this bit is set, otherwise the store missed the data cache. This information is valid only for the following store events: UOPS_RETIRED.ALL (if store is tagged), MEM_UOPS_RETIRED.STLB_MISS_STORES, MEM_UOPS_RETIRED.SPLIT_STORES, MEM_UOPS_RETIRED.ALL_STORES</li> <li>▪ Other bits are zero, The STLB_MISS, LOCK bit information can be obtained by programming the corresponding store event in Table 18-45.</li> </ul>
Reserved	A8H	Always zero.

### 18.11.3.1 EventingIP Record

The PEBS record layout for processors based on Intel microarchitecture code name Haswell adds a new field at offset 0B0H. This is the eventingIP field that records the IP address of the retired instruction that triggered the PEBS assist. The EIP/RIP field at offset 08H records the IP address of the next instruction to be executed following the PEBS assist.

### 18.11.4 Off-core Response Performance Monitoring

The core PMU facility to collect off-core response events are similar to those described in Section 18.9.5. The event codes are listed in Table 18-35. Each event code for off-core response monitoring requires programming an associated configuration MSR, MSR\_OFFCORE\_RSP\_x. Software must program MSR\_OFFCORE\_RSP\_x according to:

- Transaction request type encoding (bits 15:0): see Table 18-47.
- Supplier information (bits 30:16): see Table 18-48 and Table 18-48.
- Snoop response information (bits 37:31): see Table 18-38.



**Table 18-47. MSR\_OFFCORE\_RSP\_x Request\_Type Definition (Haswell microarchitecture)**

Bit Name	Offset	Description
DMND_DATA_RD	0	(R/W). Counts the number of demand data reads of full and partial cachelines as well as demand data page table entry cacheline reads. Does not count L2 data read prefetches or instruction fetches.
DMND_RFO	1	(R/W). Counts the number of demand and DCU prefetch reads for ownership (RFO) requests generated by a write to data cacheline. Does not count L2 RFO prefetches.
DMND_IFETCH	2	(R/W). Counts the number of demand and DCU prefetch instruction cacheline reads. Does not count L2 code read prefetches.
Reserved	3	Reserved
PF_DATA_RD	4	(R/W). Counts the number of data cacheline reads generated by L2 prefetchers.
PF_RFO	5	(R/W). Counts the number of RFO requests generated by L2 prefetchers.
PF_IFETCH	6	(R/W). Counts the number of code reads generated by L2 prefetchers.
Reserved	7-14	Reserved
OTHER	15	(R/W). Any other request that crosses IDI, including I/O.

The supplier information field listed in Table 18-48 and Table 18-48. The fields vary across products (according to CPUID signatures) and is noted in the description.

**Table 18-48. MSR\_OFFCORE\_RSP\_x Supplier Info Field Definition (CPUID Signature 06\_3CH, 06\_46H)**

Subtype	Bit Name	Offset	Description
Common	Any	16	(R/W). Catch all value for any response types.
Supplier Info	NO_SUPP	17	(R/W). No Supplier Information available
	L3_HITM	18	(R/W). M-state initial lookup stat in L3.
	L3_HITE	19	(R/W). E-state
	L3_HITS	20	(R/W). S-state
	Reserved	21	Reserved
	LOCAL	22	(R/W). Local DRAM Controller
	Reserved	30:23	Reserved

**Table 18-49. MSR\_OFFCORE\_RSP\_x Supplier Info Field Definition (CUID Signature 06\_45H)**

Subtype	Bit Name	Offset	Description
Common	Any	16	(R/W). Catch all value for any response types.
Supplier Info	NO_SUPP	17	(R/W). No Supplier Information available
	L3_HITM	18	(R/W). M-state initial lookup stat in L3.
	L3_HITE	19	(R/W). E-state
	L3_HITS	20	(R/W). S-state
	Reserved	21	Reserved
	L4_HIT_LOCAL_L4	22	(R/W). L4 Cache
	L4_HIT_REMOTE_HOP0_L4	23	(R/W). L4 Cache
	L4_HIT_REMOTE_HOP1_L4	24	(R/W). L4 Cache
	L4_HIT_REMOTE_HOP2P_L4	25	(R/W). L4 Cache
	Reserved	30:26	Reserved

#### 18.11.4.1 Off-core Response Performance Monitoring in Intel Xeon Processors E5 v3 Series

Table 18-48 lists the supplier information field that apply to Intel Xeon processor E5 v3 series (CUID signature 06\_3FH).

**Table 18-50. MSR\_OFFCORE\_RSP\_x Supplier Info Field Definition**

Subtype	Bit Name	Offset	Description
Common	Any	16	(R/W). Catch all value for any response types.
Supplier Info	NO_SUPP	17	(R/W). No Supplier Information available
	L3_HITM	18	(R/W). M-state initial lookup stat in L3.
	L3_HITE	19	(R/W). E-state
	L3_HITS	20	(R/W). S-state
	L3_HITF	21	(R/W). F-state
	LOCAL	22	(R/W). Local DRAM Controller
	Reserved	26:23	Reserved
	L3_MISS_REMOTE_HOP0	27	(R/W). Hop 0 Remote supplier
	L3_MISS_REMOTE_HOP1	28	(R/W). Hop 1 Remote supplier
	L3_MISS_REMOTE_HOP2P	29	(R/W). Hop 2 or more Remote supplier
	Reserved	30	Reserved

#### 18.11.5 Performance Monitoring and Intel® TSX

Chapter 16 of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1* describes the details of Intel® Transactional Synchronization Extensions (Intel TSX). This section describes performance monitoring support for Intel TSX.

If a processor supports Intel TSX, the core PMU enhances its IA32\_PERFEVTSELx MSR with two additional bit fields for event filtering. Support for Intel TSX is indicated by either (a) CUID.(EAX=7, ECX=0):RTM[bit 11]=1, or (b) if CUID.07H.EBX.HLE [bit 4] = 1. The TSX-enhanced layout of IA32\_PERFEVTSELx is shown in Figure 18-40. The two additional bit fields are:

- **IN\_TX** (bit 32): When set, the counter will only include counts that occurred inside a transactional region, regardless of whether that region was aborted or committed. This bit may only be set if the processor supports HLE or RTM.
- **IN\_TXCP** (bit 33): When set, the counter will not include counts that occurred inside of an aborted transactional region. This bit may only be set if the processor supports HLE or RTM. This bit may only be set for IA32\_PERFEVTSEL2.

When the IA32\_PERFEVTSELx MSR is programmed with both IN\_TX=0 and IN\_TXCP=0 on a processor that supports Intel TSX, the result in a counter may include detectable conditions associated with a transaction code region for its aborted execution (if any) and completed execution.

In the initial implementation, software may need to take pre-caution when using the IN\_TXCP bit. see Table 35-27.

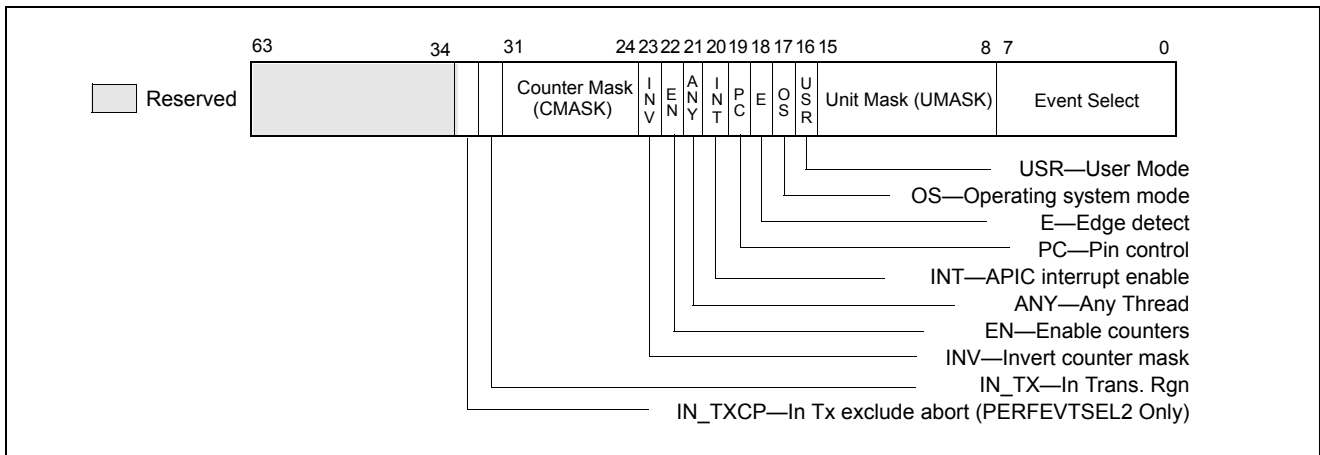


Figure 18-40. Layout of IA32\_PERFEVTSELx MSRs Supporting Intel TSX

A common usage of setting IN\_TXCP=1 is to capture the number of events that were discarded due to a transactional abort. With IA32\_PMC2 configured to count in such a manner, then when a transactional region aborts, the value for that counter is restored to the value it had prior to the aborted transactional region. As a result, any updates performed to the counter during the aborted transactional region are discarded.

On the other hand, setting IN\_TX=1 can be used to drill down on the performance characteristics of transactional code regions. When a PMCx is configured with the corresponding IA32\_PERFEVTSELx.IN\_TX=1, only eventing conditions that occur inside transactional code regions are propagated to the event logic and reflected in the counter result. Eventing conditions specified by IA32\_PERFEVTSELx but occurring outside a transactional region are discarded. The following example illustrates using three counters to drill down cycles spent inside and outside of transactional regions:

- Program IA32\_PERFEVTSEL2 to count Unhalted\_Core\_Cycles with (IN\_TXCP=1, IN\_TX=0), such that IA32\_PMC2 will count cycles spent due to aborted TSX transactions;
- Program IA32\_PERFEVTSEL0 to count Unhalted\_Core\_Cycles with (IN\_TXCP=0, IN\_TX=1), such that IA32\_PMC0 will count cycles spent by the transactional code regions;
- Program IA32\_PERFEVTSEL1 to count Unhalted\_Core\_Cycles with (IN\_TXCP=0, IN\_TX=0), such that IA32\_PMC1 will count total cycles spent by the non-transactional code and transactional code regions.

Additionally, a number of performance events are solely focused on characterizing the execution of Intel TSX transactional code, they are listed in Table 19-8.

### 18.11.5.1 Intel TSX and PEBS Support

If a PEBS event would have occurred inside a transactional region, then the transactional region first aborts, and then the PEBS event is processed.

Two of the TSX performance monitoring events in Table 19-8 also support using PEBS facility to capture additional information. They are:

- HLE\_RETIREDA.BORT ED (encoding C8H mask 04H),
- RTM\_RETIREDA.BORTED (encoding C9H mask 04H).

A transactional abort (HLE\_RETIREDA.BORTED,RTM\_RETIREDA.BORTED) can also be programmed to cause PEBS events. In this scenario, a PEBS event is processed following the abort.

Pending a PEBS record inside of a transactional region will cause a transactional abort. If a PEBS record was pending at the time of the abort or on an overflow of the TSX PEBS events listed above, only the following PEBS entries will be valid (enumerated by PEBS entry offset B8H bits[33:32] to indicate an HLE abort or an RTM abort):

- Offset B0H: EventingIP,
- Offset B8H: TX Abort Information

These fields are set for all PEBS events.

- Offset 08H (RIP/EIP) corresponds to the instruction following the outermost XACQUIRE in HLE or the first instruction of the fallback handler of the outermost XBEGIN instruction in RTM. This is useful to identify the aborted transactional region.

In the case of HLE, an aborted transaction will restart execution deterministically at the start of the HLE region. In the case of RTM, an aborted transaction will transfer execution to the RTM fallback handler.

The layout of the TX Abort Information field is given in Table 18-51.

**Table 18-51. TX Abort Information Field Definition**

Bit Name	Offset	Description
Cycles_Last_TX	31:0	The number of cycles in the last TSX region, regardless of whether that region had aborted or committed.
HLE_Abort	32	If set, the abort information corresponds to an aborted HLE execution
RTM_Abort	33	If set, the abort information corresponds to an aborted RTM execution
Instruction_Abort	34	If set, the abort was associated with the instruction corresponding to the eventing IP (offset 0B0H) within the transactional region.
Non_Instruction_Abort	35	If set, the instruction corresponding to the eventing IP may not necessarily be related to the transactional abort.
Retry	36	If set, retrying the transactional execution may have succeeded.
Data_Conflict	37	If set, another logical processor conflicted with a memory address that was part of the transactional region that aborted.
Capacity Writes	38	If set, the transactional region aborted due to exceeding resources for transactional writes.
Capacity Reads	39	If set, the transactional region aborted due to exceeding resources for transactional reads.
Reserved	63:40	Reserved

### 18.11.6 Uncore Performance Monitoring Facilities in the 4th Generation Intel® Core™ Processors

The uncore sub-system in the 4th Generation Intel® Core™ processors provides its own performance monitoring facility. The uncore PMU facility provides dedicated MSRs to select uncore performance monitoring events in a similar manner as those described in Section 18.9.6.

The ARB unit and each C-Box provide local pairs of event select MSR and counter register. The layout of the event select MSRs in the C-Boxes are identical as shown in Figure 18-38.

At the uncore domain level, there is a master set of control MSRs that centrally manages all the performance monitoring facility of uncore units. Figure 18-39 shows the layout of the uncore domain global control.

Additionally, there is also a fixed counter, counting uncore clockticks, for the uncore domain. Table 18-39 summarizes the number MSR for uncore PMU for each box.

**Table 18-52. Uncore PMU MSR Summary**

Box	# of Boxes	Counters per Box	Counter Width	General Purpose	Global Enable	Comment
C-Box	SKU specific	2	44	Yes	Per-box	Up to 4, see Table 35-19 MSR_UNC_CBO_CONFIG
ARB	1	2	44	Yes	Uncore	
Fixed Counter	N.A.	N.A.	48	No	Uncore	

The uncore performance events for the C-Box and ARB units are listed in Table 19-9.

### 18.11.7 Intel® Xeon® Processor E5 v3 Family Uncore Performance Monitoring Facility

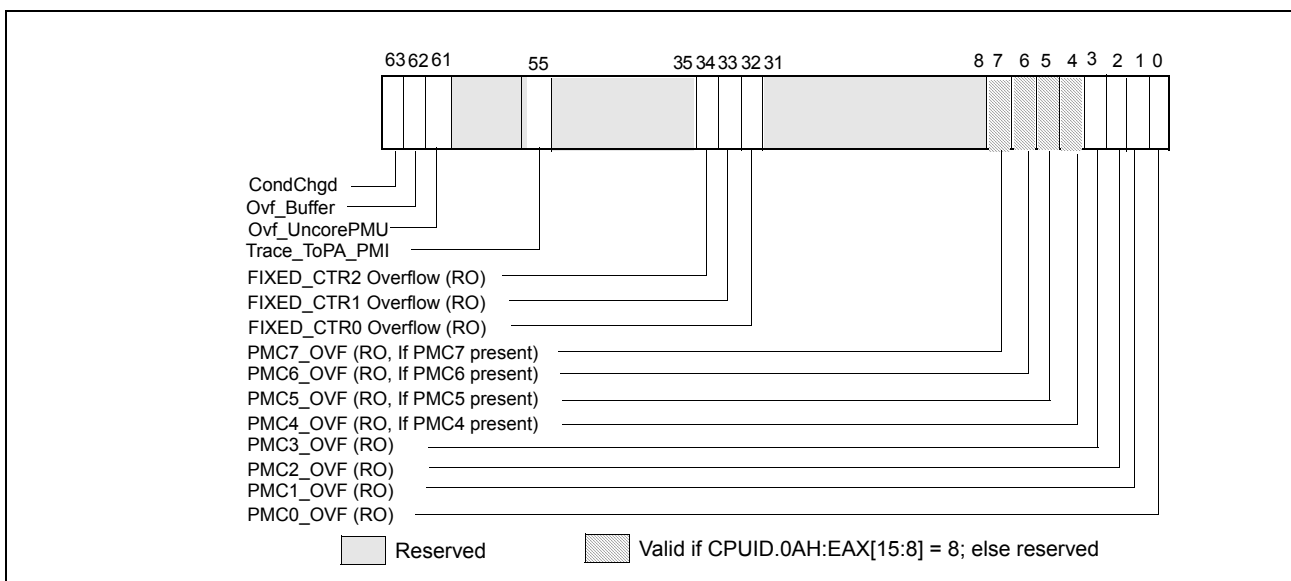
Details of the uncore performance monitoring facility of Intel Xeon Processor E5 v3 families are available in “Intel® Xeon® Processor E5 v3 Uncore Performance Monitoring Programming Reference Manual”. The MSR-based uncore PMU interfaces are listed in Table 35-31.

## 18.12 INTEL® CORE™ M PROCESSOR PERFORMANCE MONITORING FACILITY

The Intel® Core™ M processor and the 5th Generation Intel Core processor families are based on the Broadwell microarchitecture. The core PMU supports architectural performance monitoring capability with version ID 3 (see Section 18.2.3) and a host of non-architectural monitoring capabilities.

Architectural performance monitoring version 3 capabilities are described in Section 18.2.3.

The core PMU has the same capability as those described in Section 18.11. IA32\_PERF\_GLOBAL\_STATUS provide a bit indicator (bit 55) for PMI handler to distinguish PMI due to output buffer overflow condition due to accumulating packet data from Intel Processor Trace.



**Figure 18-41. IA32\_PERF\_GLOBAL\_STATUS MSR in Broadwell Microarchitecture**

Details of Intel Processor Trace is described in Chapter 36, “Intel® Processor Trace”. IA32\_PERF\_GLOBAL\_OVF\_CTRL MSR provide a corresponding reset control bit.

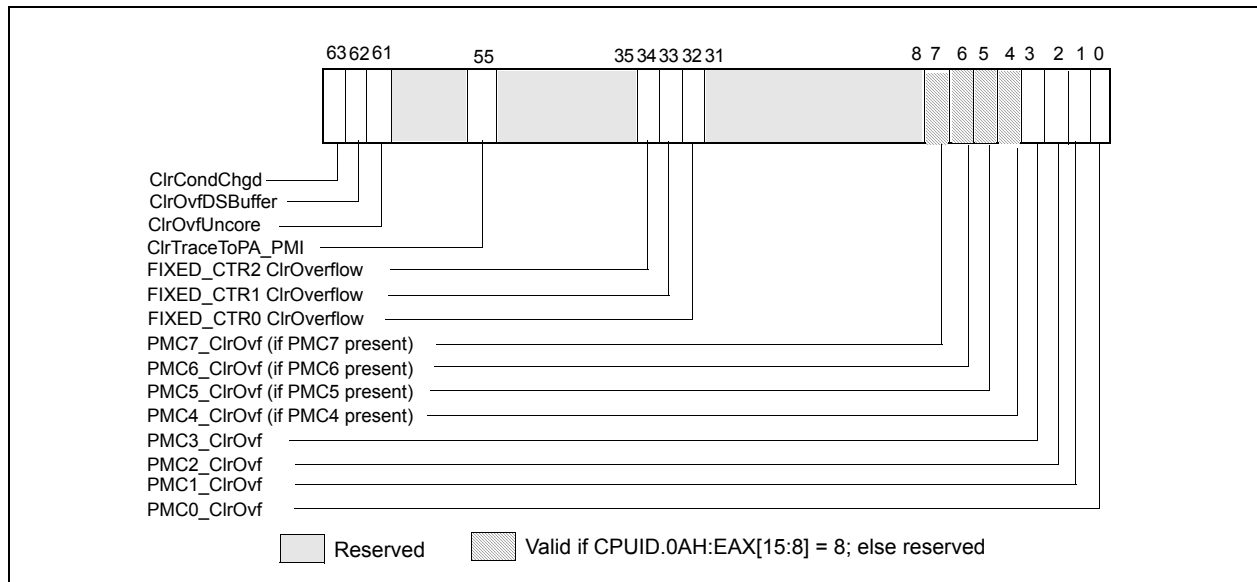


Figure 18-42. IA32\_PERF\_GLOBAL\_OVF\_CTRL MSR in Broadwell microarchitecture

The specifics of non-architectural performance events are listed in Chapter 19, “Performance Monitoring Events”.

## 18.13 6TH GENERATION INTEL® CORE™ PROCESSOR PERFORMANCE MONITORING FACILITY

The 6th generation Intel® Core™ processor is based on the Skylake microarchitecture. The core PMU supports architectural performance monitoring capability with version ID 4 (see Section 18.2.4) and a host of non-architectural monitoring capabilities.

Architectural performance monitoring version 4 capabilities are described in Section 18.2.4.

The core PMU’s capability is similar to those described in Section 18.9 through Section 18.9.5, with some differences and enhancements summarized in Table 18-42. Additionally, the core PMU provides some enhancement to support performance monitoring when the target workload contains instruction streams using Intel® Transactional Synchronization Extensions (TSX), see Section 18.11.5. For details of Intel TSX, see Chapter 16, “Programming with Intel® Transactional Synchronization Extensions” of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*.

Performance monitoring result may be affected by side-band activity on processors that support Intel SGX, details are described in Chapter 43, “Enclave Code Debug and Profiling”.

Table 18-53. Core PMU Comparison

Box	Intel® microarchitecture code name Skylake	Intel® microarchitecture code name Haswell and Broadwell	Comment
# of Fixed counters per thread	3	3	
# of general-purpose counters per core	8	8	
Counter width (R,W)	R:48, W: 32/48	R:48, W: 32/48	See Section 18.2.2.
# of programmable counters per thread	4 or (8 if a core not shared by two threads)	4 or (8 if a core not shared by two threads)	CPUID enumerates # of counters.
Architectural Perfmon version	4	3	See Section 18.2.4
PMI Overhead Mitigation	<ul style="list-style-type: none"> <li>Freeze_Perfmon_on_PMI with streamlined semantics.</li> <li>Freeze_on_LBR with streamlined semantics.</li> <li>Freeze_while_SMM.</li> </ul>	<ul style="list-style-type: none"> <li>Freeze_Perfmon_on_PMI with legacy semantics.</li> <li>Freeze_on_LBR with legacy semantics for branch profiling.</li> <li>Freeze_while_SMM.</li> </ul>	See Section 17.4.7. Legacy semantics not supported with version 4 or higher.
Counter and Buffer Overflow Status Management	<ul style="list-style-type: none"> <li>Query via IA32_PERF_GLOBAL_STATUS</li> <li>Reset via IA32_PERF_GLOBAL_STATUS_RESET</li> <li>Set via IA32_PERF_GLOBAL_STATUS_SET</li> </ul>	<ul style="list-style-type: none"> <li>Query via IA32_PERF_GLOBAL_STATUS</li> <li>Reset via IA32_PERF_GLOBAL_OVF_CTRL</li> </ul>	See Section 18.2.4.
IA32_PERF_GLOBAL_STATUS Indicators of Overflow/Overhead/Interference	<ul style="list-style-type: none"> <li>Individual counter overflow</li> <li>PEBS buffer overflow</li> <li>ToPA buffer overflow</li> <li>CTR_Frz, LBR_Frz, ASCII</li> </ul>	<ul style="list-style-type: none"> <li>Individual counter overflow</li> <li>PEBS buffer overflow</li> <li>ToPA buffer overflow (applicable to Broadwell microarchitecture)</li> </ul>	See Section 18.2.4.
Enable control in IA32_PERF_GLOBAL_STATUS	<ul style="list-style-type: none"> <li>CTR_Frz,</li> <li>LBR_Frz</li> </ul>	NA	See Section 18.2.4.1.
Perfmon Counter In-Use Indicator	Query IA32_PERF_GLOBAL_INUSE	NA	See Section 18.2.4.3.
Precise Events	See Table 18-56.	See Table 18-32.	IA32_PMC4-PMC7 do not support PEBS.
PEBS for front end events	See Section 18.13.1.4;	No	
LBR Record Format Encoding	000101b	000100b	Section 17.4.8.1
LBR Size	32 entries	16 entries	
LBR Entry	From_IP/To_IP/LBR_Info triplet	From_IP/To_IP pair	Section 17.10
LBR Timing	Yes	No	Section 17.10.1
Call Stack Profiling	Yes, see Section 17.9	Yes, see Section 17.9	Use LBR facility
Off-core Response Event	MSR 1A6H and 1A7H; Extended request and response types.	MSR 1A6H and 1A7H; Extended request and response types.	
Intel TSX support for Perfmon	See Section 18.11.5;	See Section 18.11.5;	

### 18.13.1 Processor Event Based Sampling (PEBS) Facility

The PEBS facility in the 6th Generation Intel Core processor provides a number enhancement relative to PEBS in processors based on Haswell/Broadwell microarchitectures. The key components and differences of PEBS facility relative to Haswell/Broadwell microarchitecture is summarized in Table 18-54.

**Table 18-54. PEBS Facility Comparison**

Box	Intel® microarchitecture code name Skylake	Intel® microarchitecture code name Haswell and Broadwell	Comment
Valid IA32_PMCx	PMC0-PMC3	PMC0-PMC3	No PEBS on PMC4-PMC7.
PEBS Buffer Programming	Section 18.8.1.1	Section 18.8.1.1	Unchanged
IA32_PEBS_ENABLE Layout	Figure 18-21	Figure 18-21	
PEBS-EventingIP	Yes	Yes	
PEBS record format encoding	0011b	0010b	
PEBS record layout	Table 18-55; enhanced fields at offsets 98H- B8H; and TSC record field at C0H.	Table 18-44; enhanced fields at offsets 98H, A0H, A8H, B0H.	
Multi-counter PEBS resolution	PEBS record 90H resolves the eventing counter overflow.	PEBS record 90H reflects IA32_PERF_GLOBAL_STATUS.	
Precise Events	See Table 18-56.	See Table 18-32.	IA32_PMC4-IA32_PMC7 do not support PEBS.
PEBS-PDIR	Yes	Yes	IA32_PMC1 only.
PEBS-Load Latency	See Section 18.9.4.2.	See Section 18.9.4.2.	
Data Address Profiling	Yes	Yes	
FrontEnd event support	FrontEnd_Retried event and MSR_PEBS_FRONTEND.	No	IA32_PMC0-PMC3 only.

Only IA32\_PMC0 through IA32\_PMC3 support PEBS.

## NOTES

Precise events are only valid when the following fields of IA32\_PERFEVTSELx are all zero: AnyThread, Edge, Invert, CMask.

In a PMU with PDIR capability, PEBS behavior is unpredictable if IA32\_PERFEVTSELx or IA32\_PMCx is changed for a PEBS-enabled counter while an event is being counted. To avoid this, changes to the programming or value of a PEBS-enabled counter should be performed when the counter is disabled.

### 18.13.1.1 PEBS Data Format

The PEBS record format for the 6th generation Intel Core processor is reporting with encoding 0011b in IA32\_PERF\_CAPABILITIES[11:8]. The lay out is shown in Table 18-55. The PEBS record format, along with debug/store area storage format, does not change regardless of whether IA-32e mode is active or not. CPUID.01H:ECX.DTES64[bit 2] reports whether the processor's DS storage format support is mode-independent. When set, it uses 64-bit DS storage format.



**Table 18-55. PEBS Record Format for 6th Generation Intel Core Processor Family**

Byte Offset	Field	Byte Offset	Field
00H	R/EFLAGS	68H	R11
08H	R/EIP	70H	R12
10H	R/EAX	78H	R13
18H	R/EBX	80H	R14
20H	R/ECX	88H	R15
28H	R/EDX	90H	Applicable Counter
30H	R/ESI	98H	Data Linear Address
38H	R/EDI	A0H	Data Source Encoding
40H	R/EBP	A8H	Latency value (core cycles)
48H	R/ESP	B0H	EventingIP
50H	R8	B8H	TX Abort Information (Section 18.11.5.1)
58H	R9	C0H	TSC
60H	R10		

The layout of PEBS records are largely identical to those shown in Table 18-44.

The PEBS records at offsets 98H, A0H, and ABH record data gathered from three of the PEBS capabilities in prior processor generations: load latency facility (Section 18.9.4.2), PDIR (Section 18.9.4.4), and data address profiling (Section 18.11.3).

In the core PMU of the 6th generation processor, load latency facility and PDIR capabilities and data address profiling are unchanged relative to the 4th and 5th generation Intel Core processors. Similarly, precise store is replaced by data address profiling.

With format 0010b, a snapshot of the IA32\_PERF\_GLOBAL\_STATUS may be useful to resolve the situations when more than one of IA32\_PMICx have been configured to collect PEBS data and two consecutive overflows of the PEBS-enabled counters are sufficiently far apart in time. It is also possible for the image at 90H to indicate multiple PEBS-enabled counters have overflowed. In the latter scenario, software cannot to correlate the PEBS record entry to the multiple overflowed bits.

With PEBS record format encoding 0011b, offset 90H reports the “applicable counter” field, which is a multi-counter PEBS resolution index allowing software to correlate the PEBS record entry with the eventing PEBS overflow when multiple counters are configured to record PEBS records. Additionally, offset C0H captures a snapshot of the TSC that provides a time line annotation for each PEBS record entry.

### 18.13.1.2 PEBS Events

The list of precise events supported for PEBS in the Skylake microarchitecture is shown in Table 18-56.

Table 18-56. Precise Events for the Skylake Microarchitecture

Event Name	Event Select	Sub-event	UMask
INST_RETIRED	C0H	PREC_DIST <sup>1</sup>	01H
		ALL_CYCLES <sup>2</sup>	01H
OTHER_ASSISTS	C1H	ANY	3FH
BR_INST_RETIRED	C4H	CONDITIONAL	01H
		NEAR_CALL	02H
		ALL_BRANCHES	04H
		NEAR_RETURN	08H
		NEAR_TAKEN	20H
		FAR_BRACHES	40H
BR_MISP_RETIRED	C5H	CONDITIONAL	01H
		ALL_BRANCHES	04H
		NEAR_TAKEN	20H
FRONTEND_RETIRED	C6H	<Programmable <sup>3</sup> >	01H
HLE_RETIRED	C8H	ABORTED	04H
RTM_RETIRED	C9H	ABORTED	04H
MEM_INST_RETIRED <sup>2</sup>	D0H	LOCK_LOADS	21H
		SPLIT_LOADS	41H
		SPLIT_STORES	42H
		ALL_LOADS	81H
		ALL_STORES	82H
MEM_LOAD_RETIRED <sup>4</sup>	D1H	L1_HIT	01H
		L2_HIT	02H
		L3_HIT	04H
		L1_MISS	08H
		L2_MISS	10H
		L3_MISS	20H
		HIT_LFB	40H
MEM_LOAD_L3_HIT_RETIRED <sup>2</sup>	D2H	XSNP_MISS	01H
		XSNP_HIT	02H
		XSNP_HITM	04H
		XSNP_NONE	08H

**NOTES:**

1. Only available on IA32\_PMC1.
2. INST\_RETIRED.ALL\_CYCLES is configured with additional parameters of cmask = 10 and INV = 1
3. Subevents are specified using MSR\_PEBBS\_FRONTEND, see Section 18.13.2
4. Instruction with at least one load uop experiencing the condition specified in the UMask.

**18.13.1.3 Data Address Profiling**

The PEBS Data address profiling on the 6th generation Intel Core processor is largely unchanged from prior generation. When the DataLA facility is enabled, the relevant information written into a PEBS record affects entries at offsets 98H, A0H and A8H, as shown in Table 18-46.

**Table 18-57. Layout of Data Linear Address Information In PEBS Record**

Field	Offset	Description
Data Linear Address	98H	The linear address of the load or the destination of the store.
Store Status	AOH	<ul style="list-style-type: none"> <li>▪ <b>DCU Hit</b> (Bit 0): The store hit the data cache closest to the core (L1 cache) if this bit is set, otherwise the store missed the data cache. This information is valid only for the following store events: UOPS_RETIRED.ALL (if store is tagged), MEM_INST_RETIRED.STLB_MISS_STORES, MEM_INST_RETIRED.ALL_STORES, MEM_INST_RETIRED.SPLIT_STORES.</li> <li>▪ Other bits are zero.</li> </ul>
Reserved	A8H	Always zero.

### 18.13.1.4 PEBS Facility for Front End Events

In the 6th generation Intel Core processor, the PEBS facility has been extended to allow capturing PEBS data for some microarchitectural conditions related to front end events. The frontend microarchitectural conditions supported by PEBS requires the following interfaces:

- The IA32\_PERFEVTSELx MSR must select "FrontEnd\_Retired" (C6H) in the EventSelect field (bits 7:0) and umask = 01H,
- The "FRONTEND\_RETIRED" event employs a new MSR, MSR\_PEBS\_FRONTEND, to specify the supported frontend event details, see Table 18-58.
- Program the PEBS\_EN\_PMCx field of IA32\_PEBS\_ENABLE MSR as required.

Note the AnyThread field of IA32\_PERFEVTSELx is ignored by the processor for the "FRONTEND\_RETIRED" event.

The sub-event encodings supported by MSR\_PEBS\_FRONTEND.EVTSEL is given in Table 18-58.

**Table 18-58. FrontEnd\_Retired Sub-Event Encodings Supported by MSR\_PEBS\_FRONTEND.EVTSEL**

Sub-Event Name	EVTSEL	Description
DSB_MISS	11H	Retired Instructions which experienced decode stream buffer (DSB) miss.
L1L_MISS	12H	The fetch of retired Instructions which experienced Instruction L1 Cache true miss <sup>1</sup> . Additional requests to the same cache line as an in-flight L1l cache miss will not be counted.
L2_MISS	13H	The fetch of retired Instructions which experienced L2 Cache true miss. Additional requests to the same cache line as an in-flight MLC cache miss will not be counted.
ITLB_MISS	14H	The fetch of retired Instructions which experienced ITLB true miss. Additional requests to the same cache line as an in-flight ITLB miss will not be counted.
STLB_MISS	15H	The fetch of retired Instructions which experienced STLB true miss. Additional requests to the same cache line as an in-flight STLB miss will not be counted.
IDQ_READ_BUBBLES	6H	<p>An IDQ read bubble is defined as any one of the 4 allocation slots of IDQ that is not filled by the front-end on any cycle where there is no back end stall. Using the threshold and latency fields in MSR_PEBS_FRONTEND allows counting of IDQ read bubbles of various magnitude and duration. Latency controls the number of cycles and Threshold controls the number of allocation slots that contain bubbles.</p> <p>The event counts if and only if a sequence of at least FE_LATENCY consecutive cycles contain at least FE_TRESHOLD number of bubbles each.</p>

#### NOTES:

1. A true miss is the first miss for a cacheline/page (excluding secondary misses that fall into same cacheline/page).

The layout of MSR\_PEBS\_FRONTEND is given in Table 18-59.

**Table 18-59. MSR\_PEBS\_FRONTEND Layout**

Bit Name	Offset	Description
EVTSEL	7:0	Encodes the sub-event within FrontEnd_Retired that can use PEBS facility, see Table 18-58
IDQ_Bubble_Length	19:8	Specifies the threshold of continuously elapsed cycles for the specified width of bubbles when counting IDQ_READ_BUBBLES event
IDQ_Bubble_Width	22:20	Specifies the threshold of simultaneous bubbles when counting IDQ_READ_BUBBLES event
Reserved	63:23	Reserved

### 18.13.1.5 FRONTEND\_RETIRED

The FRONTEND\_RETIRED event is designed to help software developers identify exact instructions that caused front-end issues. There are some instances in which the event will, by design, the under-counting scenarios include the following:

- The event counts only retired (non-speculative) Frontend events, i.e. events from just true program execution path are counted.
- The event will count once per cacheline (at most). If a cacheline contains multiple instructions which caused front-end misses, the count will be only 1 for that line.
- If the multibyte sequence of an instruction spans across two cachelines and causes a miss it will be recorded once. If there were additional misses in the second cacheline, they will not be counted separately.
- If a multi-uop instruction exceeds the allocation width of one cycle, the bubbles associated with these uops will be counted once per that instruction.
- If 2 instructions are fused (macro-fusion), and either of them or both cause front-end misses, it will be counted once for the fused instruction.
- If a frontend (miss) event occurs outside instruction boundary (e.g. due to processor handling of architectural event), it may be reported for the next instruction to retire.

### 18.13.2 Off-core Response Performance Monitoring

The core PMU facility to collect off-core response events are similar to those described in Section 18.9.5. Each event code for off-core response monitoring requires programming an associated configuration MSR, MSR\_OFFCORE\_RSP\_x. Software must program MSR\_OFFCORE\_RSP\_x according to:

- Transaction request type encoding (bits 15:0): see Table 18-60.
- Supplier information (bits 30:16): see Table 18-61.
- Snoop response information (bits 37:31): see Table 18-62.

**Table 18-60. MSR\_OFFCORE\_RSP\_x Request\_Type Definition (Skylake microarchitecture)**

Bit Name	Offset	Description
DMND_DATA_RD	0	(R/W). Counts the number of demand data reads of full and partial cachelines as well as demand data page table entry cacheline reads. Does not count hw or sw prefetches.
DMND_RFO	1	(R/W). Counts the number of demand reads for ownership (RFO) requests generated by a write to data cacheline. Does not count L2 RFO prefetches.
DMND_IFETCH	2	(R/W). Counts the number of demand and DCU prefetch instruction cacheline reads. Does not count L2 code read prefetches.
Reserved	6:3	Reserved
PF_L3_DATA_RD	7	(R/W). Counts the number of MLC prefetches into L3.

**Table 18-60. MSR\_OFFCORE\_RSP\_x Request\_Type Definition (Skylake microarchitecture) (Contd.)**

Bit Name	Offset	Description
PF_L3_RFO	8	(R/W). Counts the number of RFO requests generated by MLC prefetches to L3.
Reserved	10:9	Reserved
STRM_ST	11	(R/W). Counts the number of streaming store requests.
Reserved	14:12	Reserved
OTHER	15	(R/W). Any other request that crosses IDI, including I/O.

Table 18-61 lists the supplier information field that apply to 6th generation Intel Core processors. (CPUID signature 06\_4EH, 06\_5EH).

**Table 18-61. MSR\_OFFCORE\_RSP\_x Supplier Info Field Definition (CPUID Signature 06\_4EH, 06\_5EH)**

Subtype	Bit Name	Offset	Description
Common	Any	16	(R/W). Catch all value for any response types.
Supplier Info	NO_SUPP	17	(R/W). No Supplier Information available.
	L3_HITM	18	(R/W). M-state initial lookup stat in L3.
	L3_HITE	19	(R/W). E-state
	L3_HITS	20	(R/W). S-state
	Reserved	21	Reserved
	L4_HIT	22	(R/W). L4 Cache (if L4 is present in the processor)
	Reserved	25:23	Reserved
	DRAM	26	(R/W). Local Node
	Reserved	29:27	Reserved
	SPL_HIT	30	(R/W). L4 cache super line hit (if L4 is present in the processor)

Table 18-62 lists the snoop information field that apply to processors with CPUID signature 06\_4EH, 06\_5EH.

**Table 18-62. MSR\_OFFCORE\_RSP\_x Snoop Info Field Definition (CPUID Signature 06\_4EH, 06\_5EH)**

Subtype	Bit Name	Offset	Description
Snoop Info	SNOOP_NONE	31	(R/W). No details on snoop-related information
	SNOOP_NOT_NEEDED	32	(R/W). No snoop was needed to satisfy the request.
	SNOOP_MISS	33	(R/W). A snoop was needed and it missed all snooped caches: -For LLC Hit, ReslHitl was returned by all cores -For LLC Miss, Rspl was returned by all sockets and data was returned from DRAM.
	SNOOP_HIT_NO_FWD	34	(R/W). A snoop was needed and it hits in at least one snooped cache. Hit denotes a cache-line was valid before snoop effect. This includes: -Snoop Hit w/ Invalidation (LLC Hit, RFO) -Snoop Hit, Left Shared (LLC Hit/Miss, IFetch/Data_RD) -Snoop Hit w/ Invalidation and No Forward (LLC Miss, RFO Hit S) In the LLC Miss case, data is returned from DRAM.
	SNOOP_HIT_WITH_FWD	35	(R/W). A snoop was needed and data was forwarded from a remote socket. This includes: -Snoop Forward Clean, Left Shared (LLC Hit/Miss, IFetch/Data_RD/RFT).
	SNOOP_HITM	36	(R/W). A snoop was needed and it HitM-ed in local or remote cache. HitM denotes a cache-line was in modified state before effect as a results of snoop. This includes: -Snoop HitM w/ WB (LLC miss, IFetch/Data_RD) -Snoop Forward Modified w/ Invalidation (LLC Hit/Miss, RFO) -Snoop MtoS (LLC Hit, IFetch/Data_RD).
	SNOOP_NON_DRAM	37	(R/W). Target was non-DRAM system address. This includes MMIO transactions.

## 18.14 INTEL® XEON PHI™ PROCESSOR 7200/5200/3200 PERFORMANCE MONITORING

The Intel® Xeon Phi™ processor 7200/5200/3200 series are based on the Knights Landing microarchitecture. The performance monitoring capabilities are distributed between its tiles (pair of processor cores) and untile (connecting many tiles in a physical processor package). Functional details of the tiles and untile of the Knights Landing microarchitecture can be found in Chapter 16 of *Intel® 64 and IA-32 Architectures Optimization Reference Manual*.

The PMU inside a processor core supports architectural performance monitoring capability with version ID 3.

A complete description of the tile and untile PMU programming interfaces for Intel Xeon Phi processors based on the Knights Landing microarchitecture can be found in the Technical Document section at <http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-detail.html>.

Performance monitoring events supported by the tile and untile PMU can be found at <http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-detail.html> and at <https://download.01.org/perfmon/>.

## 18.15 PERFORMANCE MONITORING (PROCESSORS BASED ON INTEL NETBURST® MICROARCHITECTURE)

The performance monitoring mechanism provided in processors based on Intel NetBurst microarchitecture is different from that provided in the P6 family and Pentium processors. While the general concept of selecting, filtering, counting, and reading performance events through the WRMSR, RDMSR, and RDPMC instructions is unchanged, the setup mechanism and MSR layouts are incompatible with the P6 family and Pentium processor mechanisms. Also, the RDPMC instruction has been extended to support faster reading of counters and to read all performance counters available in processors based on Intel NetBurst microarchitecture.

The event monitoring mechanism consists of the following facilities:

- The IA32\_MISC\_ENABLE MSR, which indicates the availability in an Intel 64 or IA-32 processor of the performance monitoring and processor event-based sampling (PEBS) facilities.
- Event selection control (ESCR) MSRs for selecting events to be monitored with specific performance counters. The number available differs by family and model (43 to 45).
- 18 performance counter MSRs for counting events.
- 18 counter configuration control (CCCR) MSRs, with one CCCR associated with each performance counter. CCCRs sets up an associated performance counter for a specific method of counting.
- A debug store (DS) save area in memory for storing PEBS records.
- The IA32\_DS\_AREA MSR, which establishes the location of the DS save area.
- The debug store (DS) feature flag (bit 21) returned by the CPUID instruction, which indicates the availability of the DS mechanism.
- The MSR\_PEBS\_ENABLE MSR, which enables the PEBS facilities and replay tagging used in at-retirement event counting.
- A set of predefined events and event metrics that simplify the setting up of the performance counters to count specific events.

Table 18-63 lists the performance counters and their associated CCCRs, along with the ESCRs that select events to be counted for each performance counter. Predefined event metrics and events are listed in Chapter 19, "Performance-Monitoring Events."

**Table 18-63. Performance Counter MSRs and Associated CCCR and ESCR MSRs (Processors Based on Intel NetBurst Microarchitecture)**

Counter			CCCR		ESCR		
Name	No.	Addr	Name	Addr	Name	No.	Addr
MSR_BPU_COUNTER0	0	300H	MSR_BPU_CCCRO	360H	MSR_BSU_ESCRO	7	3A0H
					MSR_FSB_ESCRO	6	3A2H
					MSR_MOB_ESCRO	2	3AAH
					MSR_PMH_ESCRO	4	3ACH
					MSR_BPU_ESCRO	0	3B2H
					MSR_IS_ESCRO	1	3B4H
					MSR_ITLB_ESCRO	3	3B6H
					MSR_IX_ESCRO	5	3C8H
MSR_BPU_COUNTER1	1	301H	MSR_BPU_CCCR1	361H	MSR_BSU_ESCRO	7	3A0H
					MSR_FSB_ESCRO	6	3A2H
					MSR_MOB_ESCRO	2	3AAH
					MSR_PMH_ESCRO	4	3ACH
					MSR_BPU_ESCRO	0	3B2H
					MSR_IS_ESCRO	1	3B4H
					MSR_ITLB_ESCRO	3	3B6H
					MSR_IX_ESCRO	5	3C8H

**Table 18-63. Performance Counter MSRs and Associated CCCR and ESCR MSRs (Processors Based on Intel NetBurst Microarchitecture) (Contd.)**

Counter			CCCR		ESCR		
Name	No.	Addr	Name	Addr	Name	No.	Addr
MSR_BPU_COUNTER2	2	302H	MSR_BPU_CCCR2	362H	MSR_BSU_ESCR1 MSR_FSB_ESCR1 MSR_MOB_ESCR1 MSR_PMH_ESCR1 MSR_BPU_ESCR1 MSR_IS_ESCR1 MSR_ITLB_ESCR1 MSR_IX_ESCR1	7 6 2 4 0 1 3 5	3A1H 3A3H 3ABH 3ADH 3B3H 3B5H 3B7H 3C9H
MSR_BPU_COUNTER3	3	303H	MSR_BPU_CCCR3	363H	MSR_BSU_ESCR1 MSR_FSB_ESCR1 MSR_MOB_ESCR1 MSR_PMH_ESCR1 MSR_BPU_ESCR1 MSR_IS_ESCR1 MSR_ITLB_ESCR1 MSR_IX_ESCR1	7 6 2 4 0 1 3 5	3A1H 3A3H 3ABH 3ADH 3B3H 3B5H 3B7H 3C9H
MSR_MS_COUNTER0	4	304H	MSR_MS_CCCR0	364H	MSR_MS_ESCR0 MSR_TBPU_ESCR0 MSR_TC_ESCR0	0 2 1	3C0H 3C2H 3C4H
MSR_MS_COUNTER1	5	305H	MSR_MS_CCCR1	365H	MSR_MS_ESCR0 MSR_TBPU_ESCR0 MSR_TC_ESCR0	0 2 1	3C0H 3C2H 3C4H
MSR_MS_COUNTER2	6	306H	MSR_MS_CCCR2	366H	MSR_MS_ESCR1 MSR_TBPU_ESCR1 MSR_TC_ESCR1	0 2 1	3C1H 3C3H 3C5H
MSR_MS_COUNTER3	7	307H	MSR_MS_CCCR3	367H	MSR_MS_ESCR1 MSR_TBPU_ESCR1 MSR_TC_ESCR1	0 2 1	3C1H 3C3H 3C5H
MSR_FLAME_COUNTER0	8	308H	MSR_FLAME_CCCR0	368H	MSR_FIRM_ESCR0 MSR_FLAME_ESCR0 MSR_DAC_ESCR0 MSR_SAA_T_ESCR0 MSR_U2L_ESCR0	1 0 5 2 3	3A4H 3A6H 3A8H 3AEH 3B0H
MSR_FLAME_COUNTER1	9	309H	MSR_FLAME_CCCR1	369H	MSR_FIRM_ESCR0 MSR_FLAME_ESCR0 MSR_DAC_ESCR0 MSR_SAA_T_ESCR0 MSR_U2L_ESCR0	1 0 5 2 3	3A4H 3A6H 3A8H 3AEH 3B0H
MSR_FLAME_COUNTER2	10	30AH	MSR_FLAME_CCCR2	36AH	MSR_FIRM_ESCR1 MSR_FLAME_ESCR1 MSR_DAC_ESCR1 MSR_SAA_T_ESCR1 MSR_U2L_ESCR1	1 0 5 2 3	3A5H 3A7H 3A9H 3AFH 3B1H
MSR_FLAME_COUNTER3	11	30BH	MSR_FLAME_CCCR3	36BH	MSR_FIRM_ESCR1 MSR_FLAME_ESCR1 MSR_DAC_ESCR1 MSR_SAA_T_ESCR1 MSR_U2L_ESCR1	1 0 5 2 3	3A5H 3A7H 3A9H 3AFH 3B1H
MSR_IQ_COUNTER0	12	30CH	MSR_IQ_CCCR0	36CH	MSR_CRU_ESCR0 MSR_CRU_ESCR2 MSR_CRU_ESCR4 MSR_IQ_ESCR0 <sup>1</sup> MSR_RAT_ESCR0 MSR_SSU_ESCR0 MSR_ALF_ESCR0	4 5 6 0 2 3 1	3B8H 3CCH 3E0H 3BAH 3BCH 3BEH 3CAH



**Table 18-63. Performance Counter MSRs and Associated CCCR and ESCR MSRs (Processors Based on Intel NetBurst Microarchitecture) (Contd.)**

Counter			CCCR		ESCR		
Name	No.	Addr	Name	Addr	Name	No.	Addr
MSR_IQ_COUNTER1	13	30DH	MSR_IQ_CCCR1	36DH	MSR_CRU_ESCR0	4	3B8H
					MSR_CRU_ESCR2	5	3CCH
					MSR_CRU_ESCR4	6	3E0H
					MSR_IQ_ESCR0 <sup>1</sup>	0	3BAH
					MSR_RAT_ESCR0	2	3BCH
					MSR_SSU_ESCR0	3	3BEH
					MSR_ALF_ESCR0	1	3CAH
MSR_IQ_COUNTER2	14	30EH	MSR_IQ_CCCR2	36EH	MSR_CRU_ESCR1	4	3B9H
					MSR_CRU_ESCR3	5	3CDH
					MSR_CRU_ESCR5	6	3E1H
					MSR_IQ_ESCR1 <sup>1</sup>	0	3BBH
					MSR_RAT_ESCR1	2	3BDH
					MSR_ALF_ESCR1	1	3CBH
					MSR_IQ_COUNTER3	15	30FH
MSR_CRU_ESCR3	5	3CDH					
MSR_CRU_ESCR5	6	3E1H					
MSR_IQ_ESCR1 <sup>1</sup>	0	3BBH					
MSR_RAT_ESCR1	2	3BDH					
MSR_ALF_ESCR1	1	3CBH					
MSR_IQ_COUNTER4	16	310H	MSR_IQ_CCCR4	370H			
					MSR_CRU_ESCR2	5	3CCH
					MSR_CRU_ESCR4	6	3E0H
					MSR_IQ_ESCR0 <sup>1</sup>	0	3BAH
					MSR_RAT_ESCR0	2	3BCH
					MSR_SSU_ESCR0	3	3BEH
					MSR_ALF_ESCR0	1	3CAH
MSR_IQ_COUNTER5	17	311H	MSR_IQ_CCCR5	371H	MSR_CRU_ESCR1	4	3B9H
					MSR_CRU_ESCR3	5	3CDH
					MSR_CRU_ESCR5	6	3E1H
					MSR_IQ_ESCR1 <sup>1</sup>	0	3BBH
					MSR_RAT_ESCR1	2	3BDH
					MSR_ALF_ESCR1	1	3CBH

**NOTES:**

1. MSR\_IQ\_ESCR0 and MSR\_IQ\_ESCR1 are available only on early processor builds (family 0FH, models 01H-02H). These MSRs are not available on later versions.

The types of events that can be counted with these performance monitoring facilities are divided into two classes: non-retirement events and at-retirement events.

- Non-retirement events (see Table 19-28) are events that occur any time during instruction execution (such as bus transactions or cache transactions).
- At-retirement events (see Table 19-29) are events that are counted at the retirement stage of instruction execution, which allows finer granularity in counting events and capturing machine state.

The at-retirement counting mechanism includes facilities for tagging  $\mu$ ops that have encountered a particular performance event during instruction execution. Tagging allows events to be sorted between those that occurred on an execution path that resulted in architectural state being committed at retirement as well as events that occurred on an execution path where the results were eventually cancelled and never committed to architectural state (such as, the execution of a mispredicted branch).

The Pentium 4 and Intel Xeon processor performance monitoring facilities support the three usage models described below. The first two models can be used to count both non-retirement and at-retirement events; the third model is used to count a subset of at-retirement events:

- **Event counting** — A performance counter is configured to count one or more types of events. While the counter is counting, software reads the counter at selected intervals to determine the number of events that have been counted between the intervals.

- Interrupt-based event sampling** — A performance counter is configured to count one or more types of events and to generate an interrupt when it overflows. To trigger an overflow, the counter is preset to a modulus value that will cause the counter to overflow after a specific number of events have been counted. When the counter overflows, the processor generates a performance monitoring interrupt (PMI). The interrupt service routine for the PMI then records the return instruction pointer (RIP), resets the modulus, and restarts the counter. Code performance can be analyzed by examining the distribution of RIPs with a tool like the VTune™ Performance Analyzer.
- Processor event-based sampling (PEBS)** — In PEBS, the processor writes a record of the architectural state of the processor to a memory buffer after the counter overflows. The records of architectural state provide additional information for use in performance tuning. Processor-based event sampling can be used to count only a subset of at-retirement events. PEBS captures more precise processor state information compared to interrupt based event sampling, because the latter need to use the interrupt service routine to re-construct the architectural states of processor.

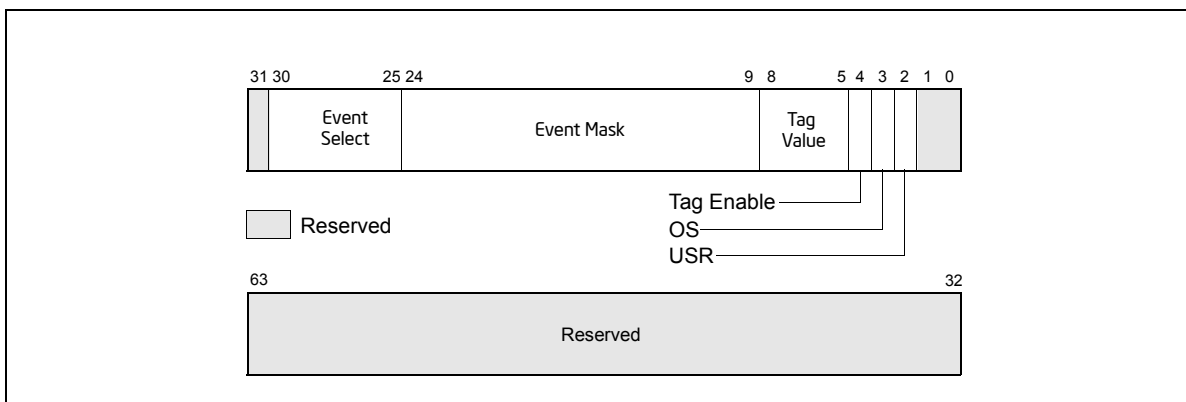
The following sections describe the MSRs and data structures used for performance monitoring in the Pentium 4 and Intel Xeon processors.

### 18.15.1 ESCR MSRs

The 45 ESCR MSRs (see Table 18-63) allow software to select specific events to be countered. Each ESCR is usually associated with a pair of performance counters (see Table 18-63) and each performance counter has several ESCRs associated with it (allowing the events counted to be selected from a variety of events).

Figure 18-43 shows the layout of an ESCR MSR. The functions of the flags and fields are:

- USR flag, bit 2** — When set, events are counted when the processor is operating at a current privilege level (CPL) of 1, 2, or 3. These privilege levels are generally used by application code and unprotected operating system code.
- OS flag, bit 3** — When set, events are counted when the processor is operating at CPL of 0. This privilege level is generally reserved for protected operating system code. (When both the OS and USR flags are set, events are counted at all privilege levels.)



**Figure 18-43. Event Selection Control Register (ESCR) for Pentium 4 and Intel Xeon Processors without Intel HT Technology Support**

- Tag enable, bit 4** — When set, enables tagging of  $\mu$ ops to assist in at-retirement event counting; when clear, disables tagging. See Section 18.15.6, "At-Retirement Counting."
- Tag value field, bits 5 through 8** — Selects a tag value to associate with a  $\mu$ op to assist in at-retirement event counting.
- Event mask field, bits 9 through 24** — Selects events to be counted from the event class selected with the event select field.
- Event select field, bits 25 through 30** — Selects a class of events to be counted. The events within this class that are counted are selected with the event mask field.

When setting up an ESCR, the event select field is used to select a specific class of events to count, such as retired branches. The event mask field is then used to select one or more of the specific events within the class to be counted. For example, when counting retired branches, four different events can be counted: branch not taken predicted, branch not taken mispredicted, branch taken predicted, and branch taken mispredicted. The OS and USR flags allow counts to be enabled for events that occur when operating system code and/or application code are being executed. If neither the OS nor USR flag is set, no events will be counted.

The ESCRs are initialized to all 0s on reset. The flags and fields of an ESCR are configured by writing to the ESCR using the WRMSR instruction. Table 18-63 gives the addresses of the ESCR MSRs.

Writing to an ESCR MSR does not enable counting with its associated performance counter; it only selects the event or events to be counted. The CCCR for the selected performance counter must also be configured. Configuration of the CCCR includes selecting the ESCR and enabling the counter.

## 18.15.2 Performance Counters

The performance counters in conjunction with the counter configuration control registers (CCCRs) are used for filtering and counting the events selected by the ESCRs. Processors based on Intel NetBurst microarchitecture provide 18 performance counters organized into 9 pairs. A pair of performance counters is associated with a particular subset of events and ESCR's (see Table 18-63). The counter pairs are partitioned into four groups:

- The BPU group, includes two performance counter pairs:
  - MSR\_BPU\_COUNTER0 and MSR\_BPU\_COUNTER1.
  - MSR\_BPU\_COUNTER2 and MSR\_BPU\_COUNTER3.
- The MS group, includes two performance counter pairs:
  - MSR\_MS\_COUNTER0 and MSR\_MS\_COUNTER1.
  - MSR\_MS\_COUNTER2 and MSR\_MS\_COUNTER3.
- The FLAME group, includes two performance counter pairs:
  - MSR\_FLAME\_COUNTER0 and MSR\_FLAME\_COUNTER1.
  - MSR\_FLAME\_COUNTER2 and MSR\_FLAME\_COUNTER3.
- The IQ group, includes three performance counter pairs:
  - MSR\_IQ\_COUNTER0 and MSR\_IQ\_COUNTER1.
  - MSR\_IQ\_COUNTER2 and MSR\_IQ\_COUNTER3.
  - MSR\_IQ\_COUNTER4 and MSR\_IQ\_COUNTER5.

The MSR\_IQ\_COUNTER4 counter in the IQ group provides support for the PEBS.

Alternate counters in each group can be cascaded: the first counter in one pair can start the first counter in the second pair and vice versa. A similar cascading is possible for the second counters in each pair. For example, within the BPU group of counters, MSR\_BPU\_COUNTER0 can start MSR\_BPU\_COUNTER2 and vice versa, and MSR\_BPU\_COUNTER1 can start MSR\_BPU\_COUNTER3 and vice versa (see Section 18.15.5.6, "Cascading Counters"). The cascade flag in the CCCR register for the performance counter enables the cascading of counters.

Each performance counter is 40-bits wide (see Figure 18-44). The RDPMC instruction is intended to allow reading of either the full counter-width (40-bits) or the low 32-bits of the counter. Reading the low 32-bits is faster than reading the full counter width and is appropriate in situations where the count is small enough to be contained in 32 bits.

The RDPMC instruction can be used by programs or procedures running at any privilege level and in virtual-8086 mode to read these counters. The PCE flag in control register CR4 (bit 8) allows the use of this instruction to be restricted to only programs and procedures running at privilege level 0.

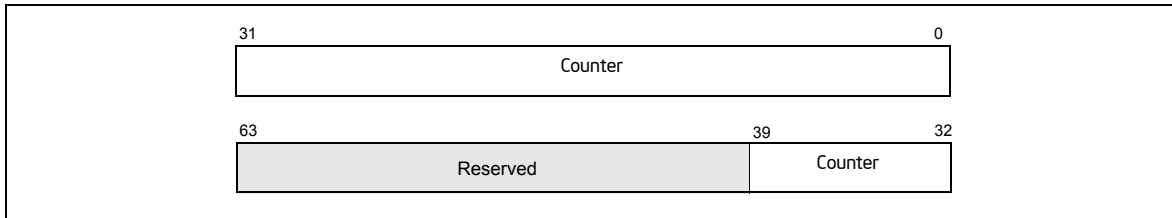


Figure 18-44. Performance Counter (Pentium 4 and Intel Xeon Processors)

The RDPMC instruction is not serializing or ordered with other instructions. Thus, it does not necessarily wait until all previous instructions have been executed before reading the counter. Similarly, subsequent instructions may begin execution before the RDPMC instruction operation is performed.

Only the operating system, executing at privilege level 0, can directly manipulate the performance counters, using the RDMSR and WRMSR instructions. A secure operating system would clear the PCE flag during system initialization to disable direct user access to the performance-monitoring counters, but provide a user-accessible programming interface that emulates the RDPMC instruction.

Some uses of the performance counters require the counters to be preset before counting begins (that is, before the counter is enabled). This can be accomplished by writing to the counter using the WRMSR instruction. To set a counter to a specified number of counts before overflow, enter a 2s complement negative integer in the counter. The counter will then count from the preset value up to -1 and overflow. Writing to a performance counter in a Pentium 4 or Intel Xeon processor with the WRMSR instruction causes all 40 bits of the counter to be written.

### 18.15.3 CCCR MSRs

Each of the 18 performance counters has one CCCR MSR associated with it (see Table 18-63). The CCCRs control the filtering and counting of events as well as interrupt generation. Figure 18-45 shows the layout of an CCCR MSR. The functions of the flags and fields are as follows:

- **Enable flag, bit 12** — When set, enables counting; when clear, the counter is disabled. This flag is cleared on reset.
- **ESCR select field, bits 13 through 15** — Identifies the ESCR to be used to select events to be counted with the counter associated with the CCCR.
- **Compare flag, bit 18** — When set, enables filtering of the event count; when clear, disables filtering. The filtering method is selected with the threshold, complement, and edge flags.
- **Complement flag, bit 19** — Selects how the incoming event count is compared with the threshold value. When set, event counts that are less than or equal to the threshold value result in a single count being delivered to the performance counter; when clear, counts greater than the threshold value result in a count being delivered to the performance counter (see Section 18.15.5.2, “Filtering Events”). The complement flag is not active unless the compare flag is set.
- **Threshold field, bits 20 through 23** — Selects the threshold value to be used for comparisons. The processor examines this field only when the compare flag is set, and uses the complement flag setting to determine the type of threshold comparison to be made. The useful range of values that can be entered in this field depend on the type of event being counted (see Section 18.15.5.2, “Filtering Events”).
- **Edge flag, bit 24** — When set, enables rising edge (false-to-true) edge detection of the threshold comparison output for filtering event counts; when clear, rising edge detection is disabled. This flag is active only when the compare flag is set.

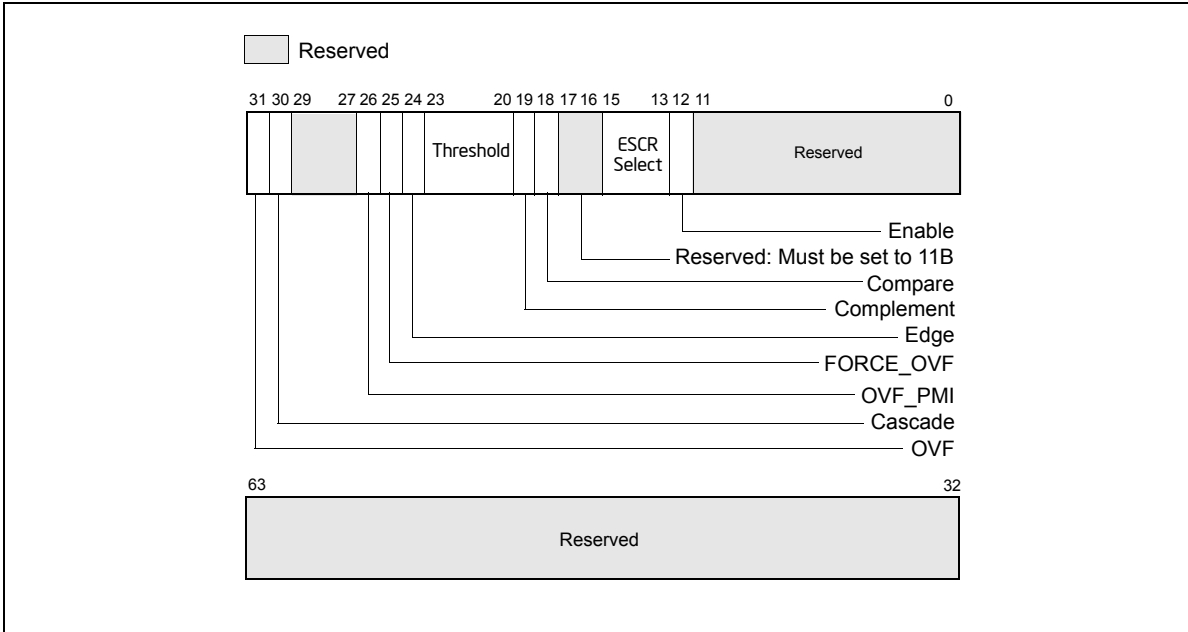


Figure 18-45. Counter Configuration Control Register (CCCR)

- **FORCE\_OVF flag, bit 25** — When set, forces a counter overflow on every counter increment; when clear, overflow only occurs when the counter actually overflows.
- **OVF\_PMI flag, bit 26** — When set, causes a performance monitor interrupt (PMI) to be generated when the counter overflows occurs; when clear, disables PMI generation. Note that the PMI is generated on the next event count after the counter has overflowed.
- **Cascade flag, bit 30** — When set, enables counting on one counter of a counter pair when its alternate counter in the other the counter pair in the same counter group overflows (see Section 18.15.2, “Performance Counters,” for further details); when clear, disables cascading of counters.
- **OVF flag, bit 31** — Indicates that the counter has overflowed when set. This flag is a sticky flag that must be explicitly cleared by software.

The CCCRs are initialized to all 0s on reset.

The events that an enabled performance counter actually counts are selected and filtered by the following flags and fields in the ESCR and CCCR registers and in the qualification order given:

1. The event select and event mask fields in the ESCR select a class of events to be counted and one or more event types within the class, respectively.
2. The OS and USR flags in the ESCR selected the privilege levels at which events will be counted.
3. The ESCR select field of the CCCR selects the ESCR. Since each counter has several ESCRs associated with it, one ESCR must be chosen to select the classes of events that may be counted.
4. The compare and complement flags and the threshold field of the CCCR select an optional threshold to be used in qualifying an event count.
5. The edge flag in the CCCR allows events to be counted only on rising-edge transitions.

The qualification order in the above list implies that the filtered output of one “stage” forms the input for the next. For instance, events filtered using the privilege level flags can be further qualified by the compare and complement flags and the threshold field, and an event that matched the threshold criteria, can be further qualified by edge detection.

The uses of the flags and fields in the CCCRs are discussed in greater detail in Section 18.15.5, “Programming the Performance Counters for Non-Retirement Events.”

### 18.15.4 Debug Store (DS) Mechanism

The debug store (DS) mechanism was introduced with processors based on Intel NetBurst microarchitecture to allow various types of information to be collected in memory-resident buffers for use in debugging and tuning programs. The DS mechanism can be used to collect two types of information: branch records and processor event-based sampling (PEBS) records. The availability of the DS mechanism in a processor is indicated with the DS feature flag (bit 21) returned by the CPUID instruction.

See Section 17.4.5, “Branch Trace Store (BTS),” and Section 18.15.7, “Processor Event-Based Sampling (PEBS),” for a description of these facilities. Records collected with the DS mechanism are saved in the DS save area. See Section 17.4.9, “BTS and DS Save Area.”

### 18.15.5 Programming the Performance Counters for Non-Retirement Events

The basic steps to program a performance counter and to count events include the following:

1. Select the event or events to be counted.
2. For each event, select an ESCR that supports the event using the values in the ESCR restrictions row in Table 19-28, Chapter 19.
3. Match the CCCR Select value and ESCR name in Table 19-28 to a value listed in Table 18-63; select a CCCR and performance counter.
4. Set up an ESCR for the specific event or events to be counted and the privilege levels at which the are to be counted.
5. Set up the CCCR for the performance counter by selecting the ESCR and the desired event filters.
6. Set up the CCCR for optional cascading of event counts, so that when the selected counter overflows its alternate counter starts.
7. Set up the CCCR to generate an optional performance monitor interrupt (PMI) when the counter overflows. If PMI generation is enabled, the local APIC must be set up to deliver the interrupt to the processor and a handler for the interrupt must be in place.
8. Enable the counter to begin counting.

#### 18.15.5.1 Selecting Events to Count

Table 19-29 in Chapter 19 lists a set of at-retirement events for processors based on Intel NetBurst microarchitecture. For each event listed in Table 19-29, setup information is provided. Table 18-64 gives an example of one of the events.

**Table 18-64. Event Example**

Event Name	Event Parameters	Parameter Value	Description
branch_retired			Counts the retirement of a branch. Specify one or more mask bits to select any combination of branch taken, not-taken, predicted and mispredicted.
	ESCR restrictions	MSR_CRU_ESCR2 MSR_CRU_ESCR3	See Table 15-3 for the addresses of the ESCR MSRs.
	Counter numbers per ESCR	ESCR2: 12, 13, 16 ESCR3: 14, 15, 17	The counter numbers associated with each ESCR are provided. The performance counters and corresponding CCCRs can be obtained from Table 15-3.
	ESCR Event Select	06H	ESCR[31:25]

Table 18-64. Event Example (Contd.)

Event Name	Event Parameters	Parameter Value	Description
	ESCR Event Mask	Bit 0: MMNP 1: MMNM 2: MMTP 3: MMTM	ESCR[24:9] Branch Not-taken Predicted Branch Not-taken Mispredicted Branch Taken Predicted Branch Taken Mispredicted
	CCCR Select	05H	CCCR[15:13]
	Event Specific Notes		P6: EMON_BR_INST_RETIRED
	Can Support PEBS	No	
	Requires Additional MSRs for Tagging	No	

For Table 19-28 and Table 19-29, Chapter 19, the name of the event is listed in the Event Name column and parameters that define the event and other information are listed in the Event Parameters column. The Parameter Value and Description columns give specific parameters for the event and additional description information. Entries in the Event Parameters column are described below.

- **ESCR restrictions** — Lists the ESCRs that can be used to program the event. Typically only one ESCR is needed to count an event.
- **Counter numbers per ESCR** — Lists which performance counters are associated with each ESCR. Table 18-63 gives the name of the counter and CCCR for each counter number. Typically only one counter is needed to count the event.
- **ESCR event select** — Gives the value to be placed in the event select field of the ESCR to select the event.
- **ESCR event mask** — Gives the value to be placed in the Event Mask field of the ESCR to select sub-events to be counted. The parameter value column defines the documented bits with relative bit position offset starting from 0, where the absolute bit position of relative offset 0 is bit 9 of the ESCR. All undocumented bits are reserved and should be set to 0.
- **CCCR select** — Gives the value to be placed in the ESCR select field of the CCCR associated with the counter to select the ESCR to be used to define the event. This value is not the address of the ESCR; it is the number of the ESCR from the Number column in Table 18-63.
- **Event specific notes** — Gives additional information about the event, such as the name of the same or a similar event defined for the P6 family processors.
- **Can support PEBS** — Indicates if PEBS is supported for the event (only supplied for at-retirement events listed in Table 19-29.)
- **Requires additional MSR for tagging** — Indicates which if any additional MSRs must be programmed to count the events (only supplied for the at-retirement events listed in Table 19-29.)

#### NOTE

The performance-monitoring events listed in Chapter 19, "Performance-Monitoring Events," are intended to be used as guides for performance tuning. The counter values reported are not guaranteed to be absolutely accurate and should be used as a relative guide for tuning. Known discrepancies are documented where applicable.

The following procedure shows how to set up a performance counter for basic counting; that is, the counter is set up to count a specified event indefinitely, wrapping around whenever it reaches its maximum count. This procedure is continued through the following four sections.

Using information in Table 19-28, Chapter 19, an event to be counted can be selected as follows:

1. Select the event to be counted.
2. Select the ESCR to be used to select events to be counted from the ESCRs field.



3. Select the number of the counter to be used to count the event from the Counter Numbers Per ESCR field.
4. Determine the name of the counter and the CCCR associated with the counter, and determine the MSR addresses of the counter, CCCR, and ESCR from Table 18-63.
5. Use the WRMSR instruction to write the ESCR Event Select and ESCR Event Mask values into the appropriate fields in the ESCR. At the same time set or clear the USR and OS flags in the ESCR as desired.
6. Use the WRMSR instruction to write the CCCR Select value into the appropriate field in the CCCR.

### NOTE

Typically all the fields and flags of the CCCR will be written with one WRMSR instruction; however, in this procedure, several WRMSR writes are used to more clearly demonstrate the uses of the various CCCR fields and flags.

This setup procedure is continued in the next section, Section 18.15.5.2, "Filtering Events."

### 18.15.5.2 Filtering Events

Each counter receives up to 4 input lines from the processor hardware from which it is counting events. The counter treats these inputs as binary inputs (input 0 has a value of 1, input 1 has a value of 2, input 2 has a value of 4, and input 3 has a value of 8). When a counter is enabled, it adds this binary input value to the counter value on each clock cycle. For each clock cycle, the value added to the counter can then range from 0 (no event) to 15.

For many events, only the 0 input line is active, so the counter is merely counting the clock cycles during which the 0 input is asserted. However, for some events two or more input lines are used. Here, the counter's threshold setting can be used to filter events. The compare, complement, threshold, and edge fields control the filtering of counter increments by input value.

If the compare flag is set, then a "greater than" or a "less than or equal to" comparison of the input value vs. a threshold value can be made. The complement flag selects "less than or equal to" (flag set) or "greater than" (flag clear). The threshold field selects a threshold value of from 0 to 15. For example, if the complement flag is cleared and the threshold field is set to 6, then any input value of 7 or greater on the 4 inputs to the counter will cause the counter to be incremented by 1, and any value less than 7 will cause an increment of 0 (or no increment) of the counter. Conversely, if the complement flag is set, any value from 0 to 6 will increment the counter and any value from 7 to 15 will not increment the counter. Note that when a threshold condition has been satisfied, the input to the counter is always 1, not the input value that is presented to the threshold filter.

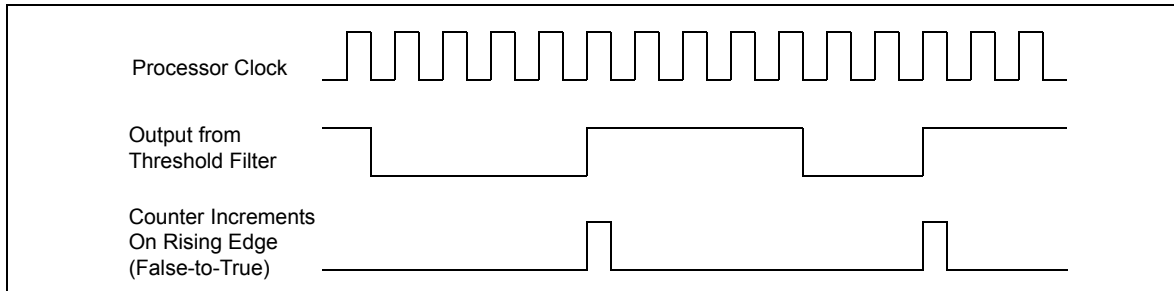
The edge flag provides further filtering of the counter inputs when a threshold comparison is being made. The edge flag is only active when the compare flag is set. When the edge flag is set, the resulting output from the threshold filter (a value of 0 or 1) is used as an input to the edge filter. Each clock cycle, the edge filter examines the last and current input values and sends a count to the counter only when it detects a "rising edge" event; that is, a false-to-true transition. Figure 18-46 illustrates rising edge filtering.

The following procedure shows how to configure a CCCR to filter events using the threshold filter and the edge filter. This procedure is a continuation of the setup procedure introduced in Section 18.15.5.1, "Selecting Events to Count."

7. (Optional) To set up the counter for threshold filtering, use the WRMSR instruction to write values in the CCCR compare and complement flags and the threshold field:
  - Set the compare flag.
  - Set or clear the complement flag for less than or equal to or greater than comparisons, respectively.
  - Enter a value from 0 to 15 in the threshold field.
8. (Optional) Select rising edge filtering by setting the CCCR edge flag.

This setup procedure is continued in the next section, Section 18.15.5.3, "Starting Event Counting."





**Figure 18-46. Effects of Edge Filtering**

### 18.15.5.3 Starting Event Counting

Event counting by a performance counter can be initiated in either of two ways. The typical way is to set the enable flag in the counter's CCCR. Following the instruction to set the enable flag, event counting begins and continues until it is stopped (see Section 18.15.5.5, "Halting Event Counting").

The following procedural step shows how to start event counting. This step is a continuation of the setup procedure introduced in Section 18.15.5.2, "Filtering Events."

9. To start event counting, use the WRMSR instruction to set the CCCR enable flag for the performance counter.

This setup procedure is continued in the next section, Section 18.15.5.4, "Reading a Performance Counter's Count."

The second way that a counter can be started by using the cascade feature. Here, the overflow of one counter automatically starts its alternate counter (see Section 18.15.5.6, "Cascading Counters").

### 18.15.5.4 Reading a Performance Counter's Count

Performance counters can be read using either the RDPMC or RDMSR instructions. The enhanced functions of the RDPMC instruction (including fast read) are described in Section 18.15.2, "Performance Counters." These instructions can be used to read a performance counter while it is counting or when it is stopped.

The following procedural step shows how to read the event counter. This step is a continuation of the setup procedure introduced in Section 18.15.5.3, "Starting Event Counting."

10. To read a performance counters current event count, execute the RDPMC instruction with the counter number obtained from Table 18-63 used as an operand.

This setup procedure is continued in the next section, Section 18.15.5.5, "Halting Event Counting."

### 18.15.5.5 Halting Event Counting

After a performance counter has been started (enabled), it continues counting indefinitely. If the counter overflows (goes one count past its maximum count), it wraps around and continues counting. When the counter wraps around, it sets its OVF flag to indicate that the counter has overflowed. The OVF flag is a sticky flag that indicates that the counter has overflowed at least once since the OVF bit was last cleared.

To halt counting, the CCCR enable flag for the counter must be cleared.

The following procedural step shows how to stop event counting. This step is a continuation of the setup procedure introduced in Section 18.15.5.4, "Reading a Performance Counter's Count."

11. To stop event counting, execute a WRMSR instruction to clear the CCCR enable flag for the performance counter.

To halt a cascaded counter (a counter that was started when its alternate counter overflowed), either clear the Cascade flag in the cascaded counter's CCCR MSR or clear the OVF flag in the alternate counter's CCCR MSR.

### 18.15.5.6 Cascading Counters

As described in Section 18.15.2, “Performance Counters,” eighteen performance counters are implemented in pairs. Nine pairs of counters and associated CCCRs are further organized as four blocks: BPU, MS, FLAME, and IQ (see Table 18-63). The first three blocks contain two pairs each. The IQ block contains three pairs of counters (12 through 17) with associated CCCRs (MSR\_IQ\_CCCR0 through MSR\_IQ\_CCCR5).

The first 8 counter pairs (0 through 15) can be programmed using ESCRs to detect performance monitoring events. Pairs of ESCRs in each of the four blocks allow many different types of events to be counted. The cascade flag in the CCCR MSR allows nested monitoring of events to be performed by cascading one counter to a second counter located in another pair in the same block (see Figure 18-45 for the location of the flag).

Counters 0 and 1 form the first pair in the BPU block. Either counter 0 or 1 can be programmed to detect an event via MSR\_MO B\_ESCR0. Counters 0 and 2 can be cascaded in any order, as can counters 1 and 3. It’s possible to set up 4 counters in the same block to cascade on two pairs of independent events. The pairing described also applies to subsequent blocks. Since the IQ PUB has two extra counters, cascading operates somewhat differently if 16 and 17 are involved. In the IQ block, counter 16 can only be cascaded from counter 14 (not from 12); counter 14 cannot be cascaded from counter 16 using the CCCR cascade bit mechanism. Similar restrictions apply to counter 17.

#### Example 18-1. Counting Events

Assume a scenario where counter X is set up to count 200 occurrences of event A; then counter Y is set up to count 400 occurrences of event B. Each counter is set up to count a specific event and overflow to the next counter. In the above example, counter X is preset for a count of -200 and counter Y for a count of -400; this setup causes the counters to overflow on the 200th and 400th counts respectively.

Continuing this scenario, counter X is set up to count indefinitely and wraparound on overflow. This is described in the basic performance counter setup procedure that begins in Section 18.15.5.1, “Selecting Events to Count.” Counter Y is set up with the cascade flag in its associated CCCR MSR set to 1 and its enable flag set to 0.

To begin the nested counting, the enable bit for the counter X is set. Once enabled, counter X counts until it overflows. At this point, counter Y is automatically enabled and begins counting. Thus counter X overflows after 200 occurrences of event A. Counter Y then starts, counting 400 occurrences of event B before overflowing. When performance counters are cascaded, the counter Y would typically be set up to generate an interrupt on overflow. This is described in Section 18.15.5.8, “Generating an Interrupt on Overflow.”

The cascading counters mechanism can be used to count a single event. The counting begins on one counter then continues on the second counter after the first counter overflows. This technique doubles the number of event counts that can be recorded, since the contents of the two counters can be added together.

### 18.15.5.7 EXTENDED CASCADING

Extended cascading is a model-specific feature in the Intel NetBurst microarchitecture with CPUID DisplayFamily\_DisplayModel 0F\_02, 0F\_03, 0F\_04, 0F\_06. This feature uses bit 11 in CCCRs associated with the IQ block. See Table 18-65.

**Table 18-65. CCR Names and Bit Positions**

CCCR Name:Bit Position	Bit Name	Description
MSR_IQ_CCCR1 2:11	Reserved	
MSR_IQ_CCCR0:11	CASCNT4INT00	Allow counter 4 to cascade into counter 0
MSR_IQ_CCCR3:11	CASCNT5INT03	Allow counter 5 to cascade into counter 3
MSR_IQ_CCCR4:11	CASCNT5INT04	Allow counter 5 to cascade into counter 4
MSR_IQ_CCCR5:11	CASCNT4INT05	Allow counter 4 to cascade into counter 5

The extended cascading feature can be adapted to the Interrupt based sampling usage model for performance monitoring. However, it is known that performance counters do not generate PMI in cascade mode or extended cascade mode due to an erratum. This erratum applies to processors with CPUID DisplayFamily\_DisplayModel signature of 0F\_02. For processors with CPUID DisplayFamily\_DisplayModel signature of 0F\_00 and 0F\_01, the erratum applies to processors with stepping encoding greater than 09H.

Counters 16 and 17 in the IQ block are frequently used in processor event-based sampling or at-retirement counting of events indicating a stalled condition in the pipeline. Neither counter 16 or 17 can initiate the cascading of counter pairs using the cascade bit in a CCCR.

Extended cascading permits performance monitoring tools to use counters 16 and 17 to initiate cascading of two counters in the IQ block. Extended cascading from counter 16 and 17 is conceptually similar to cascading other counters, but instead of using CASCADE bit of a CCCR, one of the four CASCNTxINT0y bits is used.

#### Example 18-2. Scenario for Extended Cascading

A usage scenario for extended cascading is to sample instructions retired on logical processor 1 after the first 4096 instructions retired on logical processor 0. A procedure to program extended cascading in this scenario is outlined below:

1. Write the value 0 to counter 12.
2. Write the value 04000603H to MSR\_CRU\_ESCR0 (corresponding to selecting the NBOGNTAG and NBOGNTAG event masks with qualification restricted to logical processor 1).
3. Write the value 04038800H to MSR\_IQ\_CCCR0. This enables CASCNT4INT00 and OVF\_PMI. An ISR can sample on instruction addresses in this case (do not set ENABLE, or CASCADE).
4. Write the value FFFF000H into counter 16.1.
5. Write the value 0400060CH to MSR\_CRU\_ESCR2 (corresponding to selecting the NBOGNTAG and NBOGNTAG event masks with qualification restricted to logical processor 0).
6. Write the value 00039000H to MSR\_IQ\_CCCR4 (set ENABLE bit, but not OVF\_PMI).

Another use for cascading is to locate stalled execution in a multithreaded application. Assume MOB replays in thread B cause thread A to stall. Getting a sample of the stalled execution in this scenario could be accomplished by:

1. Set up counter B to count MOB replays on thread B.
2. Set up counter A to count resource stalls on thread A; set its force overflow bit and the appropriate CASCNTx-INT0y bit.
3. Use the performance monitoring interrupt to capture the program execution data of the stalled thread.

### 18.15.5.8 Generating an Interrupt on Overflow

Any performance counter can be configured to generate a performance monitor interrupt (PMI) if the counter overflows. The PMI interrupt service routine can then collect information about the state of the processor or program when overflow occurred. This information can then be used with a tool like the Intel® VTune™ Performance Analyzer to analyze and tune program performance.

To enable an interrupt on counter overflow, the OVR\_PMI flag in the counter's associated CCCR MSR must be set. When overflow occurs, a PMI is generated through the local APIC. (Here, the performance counter entry in the local vector table [LVT] is set up to deliver the interrupt generated by the PMI to the processor.)

The PMI service routine can use the OVF flag to determine which counter overflowed when multiple counters have been configured to generate PMIs. Also, note that these processors mask PMIs upon receiving an interrupt. Clear this condition before leaving the interrupt handler.

When generating interrupts on overflow, the performance counter being used should be preset to value that will cause an overflow after a specified number of events are counted plus 1. The simplest way to select the preset value is to write a negative number into the counter, as described in Section 18.15.5.6, "Cascading Counters." Here, however, if an interrupt is to be generated after 100 event counts, the counter should be preset to minus 100 plus 1 (-100 + 1), or -99. The counter will then overflow after it counts 99 events and generate an interrupt on the next (100th) event counted. The difference of 1 for this count enables the interrupt to be generated immediately after the selected event count has been reached, instead of waiting for the overflow to be propagation through the counter.

Because of latency in the microarchitecture between the generation of events and the generation of interrupts on overflow, it is sometimes difficult to generate an interrupt close to an event that caused it. In these situations, the FORCE\_OVF flag in the CCCR can be used to improve reporting. Setting this flag causes the counter to overflow on every counter increment, which in turn triggers an interrupt after every counter increment.

### 18.15.5.9 Counter Usage Guideline

There are some instances where the user must take care to configure counting logic properly, so that it is not powered down. To use any ESCR, even when it is being used just for tagging, (any) one of the counters that the particular ESCR (or its paired ESCR) can be connected to should be enabled. If this is not done, 0 counts may result. Likewise, to use any counter, there must be some event selected in a corresponding ESCR (other than no\_event, which generally has a select value of 0).

## 18.15.6 At-Retirement Counting

At-retirement counting provides a means counting only events that represent work committed to architectural state and ignoring work that was performed speculatively and later discarded.

One example of this speculative activity is branch prediction. When a branch misprediction occurs, the results of instructions that were decoded and executed down the mispredicted path are canceled. If a performance counter was set up to count all executed instructions, the count would include instructions whose results were canceled as well as those whose results committed to architectural state.

To provide finer granularity in event counting in these situations, the performance monitoring facilities provided in the Pentium 4 and Intel Xeon processors provide a mechanism for tagging events and then counting only those tagged events that represent committed results. This mechanism is called "at-retirement counting."

Tables 19-29 through 19-33 list predefined at-retirement events and event metrics that can be used to for tagging events when using at retirement counting. The following terminology is used in describing at-retirement counting:

- **Bogus, non-bogus, retire** — In at-retirement event descriptions, the term "bogus" refers to instructions or  $\mu$ ops that must be canceled because they are on a path taken from a mispredicted branch. The terms "retired" and "non-bogus" refer to instructions or  $\mu$ ops along the path that results in committed architectural state changes as required by the program being executed. Thus instructions and  $\mu$ ops are either bogus or non-bogus, but not both. Several of the Pentium 4 and Intel Xeon processors' performance monitoring events (such as, Instruction\_Retired and Uops\_Retired in Table 19-29) can count instructions or  $\mu$ ops that are retired based on the characterization of bogus" versus non-bogus.

- **Tagging** — Tagging is a means of marking  $\mu$ ops that have encountered a particular performance event so they can be counted at retirement. During the course of execution, the same event can happen more than once per  $\mu$ op and a direct count of the event would not provide an indication of how many  $\mu$ ops encountered that event. The tagging mechanisms allow a  $\mu$ op to be tagged once during its lifetime and thus counted once at retirement. The retired suffix is used for performance metrics that increment a count once per  $\mu$ op, rather than once per event. For example, a  $\mu$ op may encounter a cache miss more than once during its life time, but a “Miss Retired” metric (that counts the number of retired  $\mu$ ops that encountered a cache miss) will increment only once for that  $\mu$ op. A “Miss Retired” metric would be useful for characterizing the performance of the cache hierarchy for a particular instruction sequence. Details of various performance metrics and how these can be constructed using the Pentium 4 and Intel Xeon processors performance events are provided in the *Intel Pentium 4 Processor Optimization Reference Manual* (see Section 1.4, “Related Literature”).
- **Replay** — To maximize performance for the common case, the Intel NetBurst microarchitecture aggressively schedules  $\mu$ ops for execution before all the conditions for correct execution are guaranteed to be satisfied. In the event that all of these conditions are not satisfied,  $\mu$ ops must be reissued. The mechanism that the Pentium 4 and Intel Xeon processors use for this reissuing of  $\mu$ ops is called replay. Some examples of replay causes are cache misses, dependence violations, and unforeseen resource constraints. In normal operation, some number of replays is common and unavoidable. An excessive number of replays is an indication of a performance problem.
- **Assist** — When the hardware needs the assistance of microcode to deal with some event, the machine takes an assist. One example of this is an underflow condition in the input operands of a floating-point operation. The hardware must internally modify the format of the operands in order to perform the computation. Assists clear the entire machine of  $\mu$ ops before they begin and are costly.

### 18.15.6.1 Using At-Retirement Counting

Processors based on Intel NetBurst microarchitecture allow counting both events and  $\mu$ ops that encountered a specified event. For a subset of the at-retirement events listed in Table 19-29, a  $\mu$ op may be tagged when it encounters that event. The tagging mechanisms can be used in Interrupt-based event sampling, and a subset of these mechanisms can be used in PEBS. There are four independent tagging mechanisms, and each mechanism uses a different event to count  $\mu$ ops tagged with that mechanism:

- **Front-end tagging** — This mechanism pertains to the tagging of  $\mu$ ops that encountered front-end events (for example, trace cache and instruction counts) and are counted with the `Front_end_event` event
- **Execution tagging** — This mechanism pertains to the tagging of  $\mu$ ops that encountered execution events (for example, instruction types) and are counted with the `Execution_Event` event.
- **Replay tagging** — This mechanism pertains to tagging of  $\mu$ ops whose retirement is replayed (for example, a cache miss) and are counted with the `Replay_event` event. Branch mispredictions are also tagged with this mechanism.
- **No tags** — This mechanism does not use tags. It uses the `Instr_retired` and the `Uops_retired` events.

Each tagging mechanism is independent from all others; that is, a  $\mu$ op that has been tagged using one mechanism will not be detected with another mechanism’s tagged- $\mu$ op detector. For example, if  $\mu$ ops are tagged using the front-end tagging mechanisms, the `Replay_event` will not count those as tagged  $\mu$ ops unless they are also tagged using the replay tagging mechanism. However, execution tags allow up to four different types of  $\mu$ ops to be counted at retirement through execution tagging.

The independence of tagging mechanisms does not hold when using PEBS. When using PEBS, only one tagging mechanism should be used at a time.

Certain kinds of  $\mu$ ops that cannot be tagged, including I/O, uncacheable and locked accesses, returns, and far transfers.

Table 19-29 lists the performance monitoring events that support at-retirement counting: specifically the `Front_end_event`, `Execution_event`, `Replay_event`, `Inst_retired` and `Uops_retired` events. The following sections describe the tagging mechanisms for using these events to tag  $\mu$ op and count tagged  $\mu$ ops.

### 18.15.6.2 Tagging Mechanism for Front\_end\_event

The Front\_end\_event counts  $\mu$ ops that have been tagged as encountering any of the following events:

- **$\mu$ op decode events** — Tagging  $\mu$ ops for  $\mu$ op decode events requires specifying bits in the ESCR associated with the performance-monitoring event, Uop\_type.
- **Trace cache events** — Tagging  $\mu$ ops for trace cache events may require specifying certain bits in the MSR\_TC\_PRECISE\_EVENT MSR (see Table 19-31).

Table 19-29 describes the Front\_end\_event and Table 19-31 describes metrics that are used to set up a Front\_end\_event count.

The MSRs specified in the Table 19-29 that are supported by the front-end tagging mechanism must be set and one or both of the NBOGUS and BOGUS bits in the Front\_end\_event event mask must be set to count events. None of the events currently supported requires the use of the MSR\_TC\_PRECISE\_EVENT MSR.

### 18.15.6.3 Tagging Mechanism For Execution\_event

Table 19-29 describes the Execution\_event and Table 19-32 describes metrics that are used to set up an Execution\_event count.

The execution tagging mechanism differs from other tagging mechanisms in how it causes tagging. One *upstream* ESCR is used to specify an event to detect and to specify a tag value (bits 5 through 8) to identify that event. A second *downstream* ESCR is used to detect  $\mu$ ops that have been tagged with that tag value identifier using Execution\_event for the event selection.

The upstream ESCR that counts the event must have its tag enable flag (bit 4) set and must have an appropriate tag value mask entered in its tag value field. The 4-bit tag value mask specifies which of tag bits should be set for a particular  $\mu$ op. The value selected for the tag value should coincide with the event mask selected in the downstream ESCR. For example, if a tag value of 1 is set, then the event mask of NBOGUS0 should be enabled, correspondingly in the downstream ESCR. The downstream ESCR detects and counts tagged  $\mu$ ops. The normal (not tag value) mask bits in the downstream ESCR specify which tag bits to count. If any one of the tag bits selected by the mask is set, the related counter is incremented by one. This mechanism is summarized in the Table 19-32 metrics that are supported by the execution tagging mechanism. The tag enable and tag value bits are irrelevant for the downstream ESCR used to select the Execution\_event.

The four separate tag bits allow the user to simultaneously but distinctly count up to four execution events at retirement. (This applies for interrupt-based event sampling. There are additional restrictions for PEBS as noted in Section 18.15.7.3, "Setting Up the PEBS Buffer.") It is also possible to detect or count combinations of events by setting multiple tag value bits in the upstream ESCR or multiple mask bits in the downstream ESCR. For example, use a tag value of 3H in the upstream ESCR and use NBOGUS0/NBOGUS1 in the downstream ESCR event mask.

### 18.15.6.4 Tagging Mechanism for Replay\_event

Table 19-29 describes the Replay\_event and Table 19-33 describes metrics that are used to set up an Replay\_event count.

The replay mechanism enables tagging of  $\mu$ ops for a subset of all replays before retirement. Use of the replay mechanism requires selecting the type of  $\mu$ op that may experience the replay in the MSR\_PEBS\_MATRIX\_VERT MSR and selecting the type of event in the MSR\_PEBS\_ENABLE MSR. Replay tagging must also be enabled with the UOP\_Tag flag (bit 24) in the MSR\_PEBS\_ENABLE MSR.

The Table 19-33 lists the metrics that are support the replay tagging mechanism and the at-retirement events that use the replay tagging mechanism, and specifies how the appropriate MSRs need to be configured. The replay tags defined in Table A-5 also enable Processor Event-Based Sampling (PEBS, see Section 17.4.9). Each of these replay tags can also be used in normal sampling by not setting Bit 24 nor Bit 25 in IA\_32\_PEBS\_ENABLE\_MSR. Each of these metrics requires that the Replay\_Event (see Table 19-29) be used to count the tagged  $\mu$ ops.

## 18.15.7 Processor Event-Based Sampling (PEBS)

The debug store (DS) mechanism in processors based on Intel NetBurst microarchitecture allow two types of information to be collected for use in debugging and tuning programs: PEBS records and BTS records. See Section 17.4.5, “Branch Trace Store (BTS),” for a description of the BTS mechanism.

PEBS permits the saving of precise architectural information associated with one or more performance events in the precise event records buffer, which is part of the DS save area (see Section 17.4.9, “BTS and DS Save Area”). To use this mechanism, a counter is configured to overflow after it has counted a preset number of events. After the counter overflows, the processor copies the current state of the general-purpose and EFLAGS registers and instruction pointer into a record in the precise event records buffer. The processor then resets the count in the performance counter and restarts the counter. When the precise event records buffer is nearly full, an interrupt is generated, allowing the precise event records to be saved. A circular buffer is not supported for precise event records.

PEBS is supported only for a subset of the at-retirement events: Execution\_event, Front\_end\_event, and Replay\_event. Also, PEBS can only be carried out using the one performance counter, the MSR\_IQ\_COUNTER4 MSR.

In processors based on Intel Core microarchitecture, a similar PEBS mechanism is also supported using IA32\_PMC0 and IA32\_PERFVTSEL0 MSRs (See Section 18.4.4).

### 18.15.7.1 Detection of the Availability of the PEBS Facilities

The DS feature flag (bit 21) returned by the CPUID instruction indicates (when set) the availability of the DS mechanism in the processor, which supports the PEBS (and BTS) facilities. When this bit is set, the following PEBS facilities are available:

- The PEBS\_UNAVAILABLE flag in the IA32\_MISC\_ENABLE MSR indicates (when clear) the availability of the PEBS facilities, including the MSR\_PEBS\_ENABLE MSR.
- The enable PEBS flag (bit 24) in the MSR\_PEBS\_ENABLE MSR allows PEBS to be enabled (set) or disabled (clear).
- The IA32\_DS\_AREA MSR can be programmed to point to the DS save area.

### 18.15.7.2 Setting Up the DS Save Area

Section 17.4.9.2, “Setting Up the DS Save Area,” describes how to set up and enable the DS save area. This procedure is common for PEBS and BTS.

### 18.15.7.3 Setting Up the PEBS Buffer

Only the MSR\_IQ\_COUNTER4 performance counter can be used for PEBS. Use the following procedure to set up the processor and this counter for PEBS:

1. Set up the precise event buffering facilities. Place values in the precise event buffer base, precise event index, precise event absolute maximum, and precise event interrupt threshold, and precise event counter reset fields of the DS buffer management area (see Figure 17-5) to set up the precise event records buffer in memory.
2. Enable PEBS. Set the Enable PEBS flag (bit 24) in MSR\_PEBS\_ENABLE MSR.
3. Set up the MSR\_IQ\_COUNTER4 performance counter and its associated CCCR and one or more ESCRs for PEBS as described in Tables 19-29 through 19-33.

### 18.15.7.4 Writing a PEBS Interrupt Service Routine

The PEBS facilities share the same interrupt vector and interrupt service routine (called the DS ISR) with the non-precise event-based sampling and BTS facilities. To handle PEBS interrupts, PEBS handler code must be included in the DS ISR. See Section 17.4.9.5, “Writing the DS Interrupt Service Routine,” for guidelines for writing the DS ISR.



### 18.15.7.5 Other DS Mechanism Implications

The DS mechanism is not available in the SMM. It is disabled on transition to the SMM mode. Similarly the DS mechanism is disabled on the generation of a machine check exception and is cleared on processor RESET and INIT.

The DS mechanism is available in real address mode.

### 18.15.8 Operating System Implications

The DS mechanism can be used by the operating system as a debugging extension to facilitate failure analysis. When using this facility, a 25 to 30 times slowdown can be expected due to the effects of the trace store occurring on every taken branch.

Depending upon intended usage, the instruction pointers that are part of the branch records or the PEBS records need to have an association with the corresponding process. One solution requires the ability for the DS specific operating system module to be chained to the context switch. A separate buffer can then be maintained for each process of interest and the MSR pointing to the configuration area saved and setup appropriately on each context switch.

If the BTS facility has been enabled, then it must be disabled and state stored on transition of the system to a sleep state in which processor context is lost. The state must be restored on return from the sleep state.

It is required that an interrupt gate be used for the DS interrupt as opposed to a trap gate to prevent the generation of an endless interrupt loop.

Pages that contain buffers must have mappings to the same physical address for all processes/logical processors, such that any change to CR3 will not change DS addresses. If this requirement cannot be satisfied (that is, the feature is enabled on a per thread/process basis), then the operating system must ensure that the feature is enabled/disabled appropriately in the context switch code.

## 18.16 PERFORMANCE MONITORING AND INTEL HYPER-THREADING TECHNOLOGY IN PROCESSORS BASED ON INTEL NETBURST® MICROARCHITECTURE

The performance monitoring capability of processors based on Intel NetBurst microarchitecture and supporting Intel Hyper-Threading Technology is similar to that described in Section 18.15. However, the capability is extended so that:

- Performance counters can be programmed to select events qualified by logical processor IDs.
- Performance monitoring interrupts can be directed to a specific logical processor within the physical processor.

The sections below describe performance counters, event qualification by logical processor ID, and special purpose bits in ESCRs/CCCRs. They also describe MSR\_PEBS\_ENABLE, MSR\_PEBS\_MATRIX\_VERT, and MSR\_TC\_PRECISE\_EVENT.

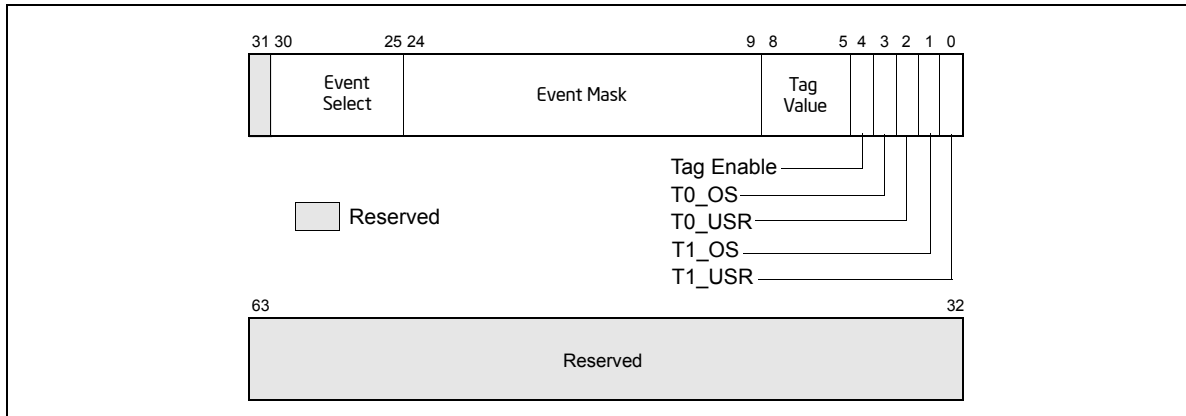
### 18.16.1 ESCR MSRs

Figure 18-47 shows the layout of an ESCR MSR in processors supporting Intel Hyper-Threading Technology.

The functions of the flags and fields are as follows:

- **T1\_USR flag, bit 0** — When set, events are counted when thread 1 (logical processor 1) is executing at a current privilege level (CPL) of 1, 2, or 3. These privilege levels are generally used by application code and unprotected operating system code.





**Figure 18-47. Event Selection Control Register (ESCR) for the Pentium 4 Processor, Intel Xeon Processor and Intel Xeon Processor MP Supporting Hyper-Threading Technology**

- **T1\_OS flag, bit 1** — When set, events are counted when thread 1 (logical processor 1) is executing at CPL of 0. This privilege level is generally reserved for protected operating system code. (When both the T1\_OS and T1\_USR flags are set, thread 1 events are counted at all privilege levels.)
- **T0\_USR flag, bit 2** — When set, events are counted when thread 0 (logical processor 0) is executing at a CPL of 1, 2, or 3.
- **T0\_OS flag, bit 3** — When set, events are counted when thread 0 (logical processor 0) is executing at CPL of 0. (When both the T0\_OS and T0\_USR flags are set, thread 0 events are counted at all privilege levels.)
- **Tag enable, bit 4** — When set, enables tagging of  $\mu$ ops to assist in at-retirement event counting; when clear, disables tagging. See Section 18.15.6, “At-Retirement Counting.”
- **Tag value field, bits 5 through 8** — Selects a tag value to associate with a  $\mu$ op to assist in at-retirement event counting.
- **Event mask field, bits 9 through 24** — Selects events to be counted from the event class selected with the event select field.
- **Event select field, bits 25 through 30** — Selects a class of events to be counted. The events within this class that are counted are selected with the event mask field.

The T0\_OS and T0\_USR flags and the T1\_OS and T1\_USR flags allow event counting and sampling to be specified for a specific logical processor (0 or 1) within an Intel Xeon processor MP (See also: Section 8.4.5, “Identifying Logical Processors in an MP System,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*).

Not all performance monitoring events can be detected within an Intel Xeon processor MP on a per logical processor basis (see Section 18.16.4, “Performance Monitoring Events”). Some sub-events (specified by an event mask bits) are counted or sampled without regard to which logical processor is associated with the detected event.

## 18.16.2 CCCR MSRs

Figure 18-48 shows the layout of a CCCR MSR in processors supporting Intel Hyper-Threading Technology. The functions of the flags and fields are as follows:

- **Enable flag, bit 12** — When set, enables counting; when clear, the counter is disabled. This flag is cleared on reset
- **ESCR select field, bits 13 through 15** — Identifies the ESCR to be used to select events to be counted with the counter associated with the CCCR.
- **Active thread field, bits 16 and 17** — Enables counting depending on which logical processors are active (executing a thread). This field enables filtering of events based on the state (active or inactive) of the logical processors. The encodings of this field are as follows:

**00** — None. Count only when neither logical processor is active.

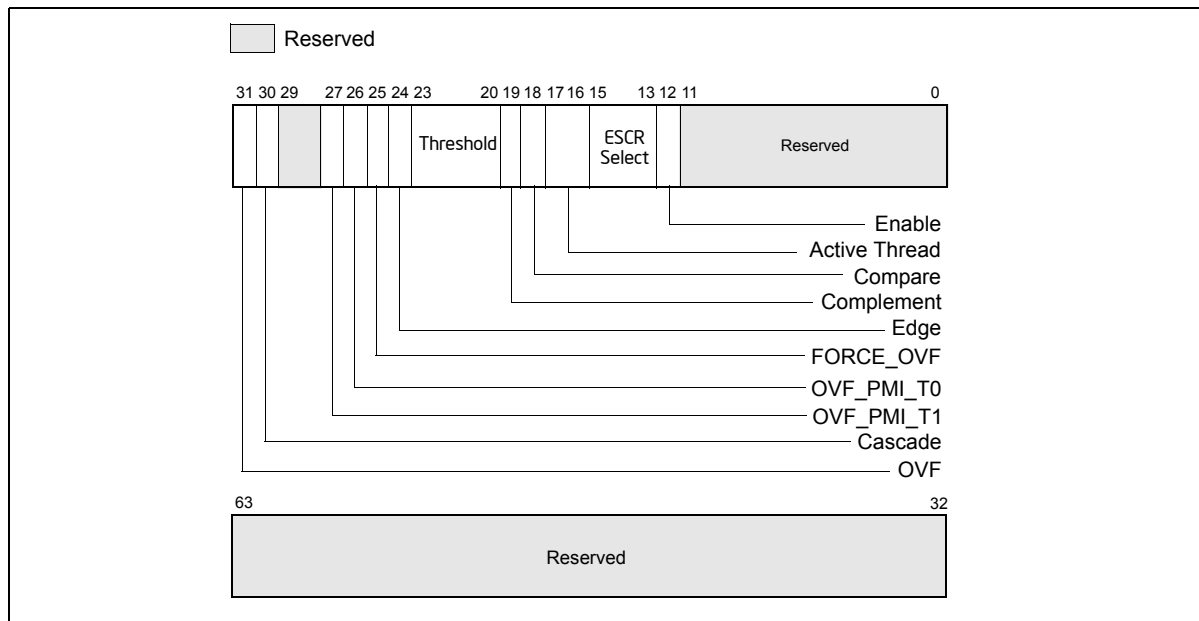
**01** — Single. Count only when one logical processor is active (either 0 or 1).

**10** — Both. Count only when both logical processors are active.

**11** — Any. Count when either logical processor is active.

A halted logical processor or a logical processor in the “wait for SIPI” state is considered inactive.

- **Compare flag, bit 18** — When set, enables filtering of the event count; when clear, disables filtering. The filtering method is selected with the threshold, complement, and edge flags.



**Figure 18-48. Counter Configuration Control Register (CCCR)**

- **Complement flag, bit 19** — Selects how the incoming event count is compared with the threshold value. When set, event counts that are less than or equal to the threshold value result in a single count being delivered to the performance counter; when clear, counts greater than the threshold value result in a count being delivered to the performance counter (see Section 18.15.5.2, “Filtering Events”). The compare flag is not active unless the compare flag is set.
- **Threshold field, bits 20 through 23** — Selects the threshold value to be used for comparisons. The processor examines this field only when the compare flag is set, and uses the complement flag setting to determine the type of threshold comparison to be made. The useful range of values that can be entered in this field depend on the type of event being counted (see Section 18.15.5.2, “Filtering Events”).
- **Edge flag, bit 24** — When set, enables rising edge (false-to-true) edge detection of the threshold comparison output for filtering event counts; when clear, rising edge detection is disabled. This flag is active only when the compare flag is set.
- **FORCE\_OVF flag, bit 25** — When set, forces a counter overflow on every counter increment; when clear, overflow only occurs when the counter actually overflows.
- **OVF\_PMI\_T0 flag, bit 26** — When set, causes a performance monitor interrupt (PMI) to be sent to logical processor 0 when the counter overflows occurs; when clear, disables PMI generation for logical processor 0. Note that the PMI is generate on the next event count after the counter has overflowed.
- **OVF\_PMI\_T1 flag, bit 27** — When set, causes a performance monitor interrupt (PMI) to be sent to logical processor 1 when the counter overflows occurs; when clear, disables PMI generation for logical processor 1. Note that the PMI is generate on the next event count after the counter has overflowed.

- **Cascade flag, bit 30** — When set, enables counting on one counter of a counter pair when its alternate counter in the other the counter pair in the same counter group overflows (see Section 18.15.2, “Performance Counters,” for further details); when clear, disables cascading of counters.
- **OVF flag, bit 31** — Indicates that the counter has overflowed when set. This flag is a sticky flag that must be explicitly cleared by software.

### 18.16.3 IA32\_PEBS\_ENABLE MSR

In a processor supporting Intel Hyper-Threading Technology and based on the Intel NetBurst microarchitecture, PEBS is enabled and qualified with two bits in the MSR\_PEBS\_ENABLE MSR: bit 25 (ENABLE\_PEBS\_MY\_THR) and 26 (ENABLE\_PEBS\_OTH\_THR) respectively. These bits do not explicitly identify a specific logical processor by logic processor ID(T0 or T1); instead, they allow a software agent to enable PEBS for subsequent threads of execution on the same logical processor on which the agent is running (“my thread”) or for the other logical processor in the physical package on which the agent is not running (“other thread”).

PEBS is supported for only a subset of the at-retirement events: Execution\_event, Front\_end\_event, and Replay\_event. Also, PEBS can be carried out only with two performance counters: MSR\_IQ\_CCCR4 (MSR address 370H) for logical processor 0 and MSR\_IQ\_CCCR5 (MSR address 371H) for logical processor 1.

Performance monitoring tools should use a processor affinity mask to bind the kernel mode components that need to modify the ENABLE\_PEBS\_MY\_THR and ENABLE\_PEBS\_OTH\_THR bits in the MSR\_PEBS\_ENABLE MSR to a specific logical processor. This is to prevent these kernel mode components from migrating between different logical processors due to OS scheduling.

### 18.16.4 Performance Monitoring Events

All of the events listed in Table 19-28 and 19-29 are available in an Intel Xeon processor MP. When Intel Hyper-Threading Technology is active, many performance monitoring events can be can be qualified by the logical processor ID, which corresponds to bit 0 of the initial APIC ID. This allows for counting an event in any or all of the logical processors. However, not all the events have this logic processor specificity, or thread specificity.

Here, each event falls into one of two categories:

- **Thread specific (TS)** — The event can be qualified as occurring on a specific logical processor.
- **Thread independent (TI)** — The event cannot be qualified as being associated with a specific logical processor.

Table 19-34 gives logical processor specific information (TS or TI) for each of the events described in Tables 19-28 and 19-29. If for example, a TS event occurred in logical processor T0, the counting of the event (as shown in Table 18-66) depends only on the setting of the T0\_USR and T0\_OS flags in the ESCR being used to set up the event counter. The T1\_USR and T1\_OS flags have no effect on the count.

**Table 18-66. Effect of Logical Processor and CPL Qualification for Logical-Processor-Specific (TS) Events**

	T1_OS/T1_USR = 00	T1_OS/T1_USR = 01	T1_OS/T1_USR = 11	T1_OS/T1_USR = 10
T0_OS/T0_USR = 00	Zero count	Counts while T1 in USR	Counts while T1 in OS or USR	Counts while T1 in OS
T0_OS/T0_USR = 01	Counts while T0 in USR	Counts while T0 in USR or T1 in USR	Counts while (a) T0 in USR or (b) T1 in OS or (c) T1 in USR	Counts while (a) T0 in OS or (b) T1 in OS
T0_OS/T0_USR = 11	Counts while T0 in OS or USR	Counts while (a) T0 in OS or (b) T0 in USR or (c) T1 in USR	Counts irrespective of CPL, T0, T1	Counts while (a) T0 in OS or (b) or T0 in USR or (c) T1 in OS
T0_OS/T0_USR = 10	Counts T0 in OS	Counts T0 in OS or T1 in USR	Counts while (a)T0 in Os or (b) T1 in OS or (c) T1 in USR	Counts while (a) T0 in OS or (b) T1 in OS

When a bit in the event mask field is TI, the effect of specifying bit-0-3 of the associated ESCR are described in Table 15-6. For events that are marked as TI in Chapter 19, the effect of selectively specifying T0\_USR, T0\_OS, T1\_USR, T1\_OS bits is shown in Table 18-67.

**Table 18-67. Effect of Logical Processor and CPL Qualification for Non-logical-Processor-specific (TI) Events**

	T1_OS/T1_USR = 00	T1_OS/T1_USR = 01	T1_OS/T1_USR = 11	T1_OS/T1_USR = 10
T0_OS/T0_USR = 00	Zero count	Counts while (a) T0 in USR or (b) T1 in USR	Counts irrespective of CPL, T0, T1	Counts while (a) T0 in OS or (b) T1 in OS
T0_OS/T0_USR = 01	Counts while (a) T0 in USR or (b) T1 in USR	Counts while (a) T0 in USR or (b) T1 in USR	Counts irrespective of CPL, T0, T1	Counts irrespective of CPL, T0, T1
T0_OS/T0_USR = 11	Counts irrespective of CPL, T0, T1	Counts irrespective of CPL, T0, T1	Counts irrespective of CPL, T0, T1	Counts irrespective of CPL, T0, T1
T0_OS/T0_USR = 0	Counts while (a) T0 in OS or (b) T1 in OS	Counts irrespective of CPL, T0, T1	Counts irrespective of CPL, T0, T1	Counts while (a) T0 in OS or (b) T1 in OS

## 18.17 COUNTING CLOCKS

The count of cycles, also known as clockticks, forms a the basis for measuring how long a program takes to execute. Clockticks are also used as part of efficiency ratios like cycles per instruction (CPI). Processor clocks may stop ticking under circumstances like the following:

- The processor is halted when there is nothing for the CPU to do. For example, the processor may halt to save power while the computer is servicing an I/O request. When Intel Hyper-Threading Technology is enabled, both logical processors must be halted for performance-monitoring counters to be powered down.
- The processor is asleep as a result of being halted or because of a power-management scheme. There are different levels of sleep. In the some deep sleep levels, the time-stamp counter stops counting.

In addition, processor core clocks may undergo transitions at different ratios relative to the processor's bus clock frequency. Some of the situations that can cause processor core clock to undergo frequency transitions include:

- TM2 transitions
- Enhanced Intel SpeedStep Technology transitions (P-state transitions)

For Intel processors that support Intel Dynamic Acceleration or XE operation, the processor core clocks may operate at a frequency that differs from the Processor Base frequency (as indicated by processor frequency information reported by CPUID instruction). See Section 18.17.5 for more detail.

There are several ways to count processor clock cycles to monitor performance. These are:

- **Non-halted clockticks** — Measures clock cycles in which the specified logical processor is not halted and is not in any power-saving state. When Intel Hyper-Threading Technology is enabled, ticks can be measured on a per-logical-processor basis. There are also performance events on dual-core processors that measure clockticks per logical processor when the processor is not halted.
- **Non-sleep clockticks** — Measures clock cycles in which the specified physical processor is not in a sleep mode or in a power-saving state. These ticks cannot be measured on a logical-processor basis.
- **Time-stamp counter** — Measures clock cycles in which the physical processor is not in deep sleep. These ticks cannot be measured on a logical-processor basis.
- **Reference clockticks** — TM2 or Enhanced Intel SpeedStep technology are two examples of processor features that can cause processor core clockticks to represent non-uniform tick intervals due to change of bus ratios. Performance events that counts clockticks of a constant reference frequency was introduced Intel Core Duo and Intel Core Solo processors. The mechanism is further enhanced on processors based on Intel Core microarchitecture.

Some processor models permit clock cycles to be measured when the physical processor is not in deep sleep (by using the time-stamp counter and the RDTSC instruction). Note that such ticks cannot be measured on a per-logical-processor basis. See Section 17.15, "Time-Stamp Counter," for detail on processor capabilities.

The first two methods use performance counters and can be set up to cause an interrupt upon overflow (for sampling). They may also be useful where it is easier for a tool to read a performance counter than to use a time stamp counter (the timestamp counter is accessed using the RDTSC instruction).

For applications with a significant amount of I/O, there are two ratios of interest:

- **Non-halted CPI** — Non-halted clockticks/instructions retired measures the CPI for phases where the CPU was being used. This ratio can be measured on a logical-processor basis when Intel Hyper-Threading Technology is enabled.
- **Nominal CPI** — Time-stamp counter ticks/instructions retired measures the CPI over the duration of a program, including those periods when the machine halts while waiting for I/O.

### 18.17.1 Non-Halted Clockticks

Use the following procedure to program ESCRs and CCCRs to obtain non-halted clockticks on processors based on Intel NetBurst microarchitecture:

1. Select an ESCR for the `global_power_events` and specify the `RUNNING` sub-event mask and the desired `T0_OS/T0_USR/T1_OS/T1_USR` bits for the targeted processor.
2. Select an appropriate counter.
3. Enable counting in the CCCR for that counter by setting the enable bit.

### 18.17.2 Non-Sleep Clockticks

Performance monitoring counters can be configured to count clockticks whenever the performance monitoring hardware is not powered-down. To count Non-sleep Clockticks with a performance-monitoring counter, do the following:

1. Select one of the 18 counters.
2. Select any of the ESCRs whose events the selected counter can count. Set its event select to anything other than `no_event`. This may not seem necessary, but the counter may be disabled if this is not done.
3. Turn threshold comparison on in the CCCR by setting the compare bit to 1.
4. Set the threshold to 15 and the complement to 1 in the CCCR. Since no event can exceed this threshold, the threshold condition is met every cycle and the counter counts every cycle. Note that this overrides any qualification (e.g. by CPL) specified in the ESCR.
5. Enable counting in the CCCR for the counter by setting the enable bit.

In most cases, the counts produced by the non-halted and non-sleep metrics are equivalent if the physical package supports one logical processor and is not placed in a power-saving state. Operating systems may execute an HLT instruction and place a physical processor in a power-saving state.

On processors that support Intel Hyper-Threading Technology (Intel HT Technology), each physical package can support two or more logical processors. Current implementation of Intel HT Technology provides two logical processors for each physical processor. While both logical processors can execute two threads simultaneously, one logical processor may halt to allow the other logical processor to execute without sharing execution resources between two logical processors.

Non-halted Clockticks can be set up to count the number of processor clock cycles for each logical processor whenever the logical processor is not halted (the count may include some portion of the clock cycles for that logical processor to complete a transition to a halted state). Physical processors that support Intel HT Technology enter into a power-saving state if all logical processors halt.

The Non-sleep Clockticks mechanism uses a filtering mechanism in CCCRs. The mechanism will continue to increment as long as one logical processor is not halted or in a power-saving state. Applications may cause a processor to enter into a power-saving state by using an OS service that transfers control to an OS's idle loop. The idle loop then may place the processor into a power-saving state after an implementation-dependent period if there is no work for the processor.

### 18.17.3 Incrementing the Time-Stamp Counter

The time-stamp counter increments when the clock signal on the system bus is active and when the sleep pin is not asserted. The counter value can be read with the RDTSC instruction.

The time-stamp counter and the non-sleep clockticks count may not agree in all cases and for all processors. See Section 17.15, “Time-Stamp Counter,” for more information on counter operation.

### 18.17.4 Non-Halted Reference Clockticks

Software can use either processor-specific performance monitor events (for example: CPU\_CLK\_UNHALTED.BUS on processors based on the Intel Core microarchitecture, and equivalent event specifications on the Intel Core Duo and Intel Core Solo processors) to count non-halted reference clockticks.

These events count reference clock cycles whenever the specified processor is not halted. The counter counts reference cycles associated with a fixed-frequency clock source irrespective of P-state, TM2, or frequency transitions that may occur to the processor.

### 18.17.5 Cycle Counting and Opportunistic Processor Operation

As a result of the state transitions due to opportunistic processor performance operation (see Chapter 14, “Power and Thermal Management”), a logical processor or a processor core can operate at frequency different from the Processor Base frequency.

The following items are expected to hold true irrespective of when opportunistic processor operation causes state transitions:

- The time stamp counter operates at a fixed-rate frequency of the processor.
- The IA32\_MPERF counter increments at a fixed frequency irrespective of any transitions caused by opportunistic processor operation.
- The IA32\_FIXED\_CTR2 counter increments at the same TSC frequency irrespective of any transitions caused by opportunistic processor operation.
- The Local APIC timer operation is unaffected by opportunistic processor operation.
- The TSC, IA32\_MPERF, and IA32\_FIXED\_CTR2 operate at close to the maximum non-turbo frequency, which is equal to the product of scalable bus frequency and maximum non-turbo ratio.

For processors based on Intel Core microarchitecture, the scalable bus frequency is encoded in the bit field MSR\_FSB\_FREQ[2:0] at (0CDH), see Chapter 35, “Model-Specific Registers (MSRs)”. The maximum resolved bus ratio can be read from the following bit field:

- If XE operation is disabled, the maximum resolved bus ratio can be read in MSR\_PLATFORM\_ID[12:8]. It corresponds to the Processor Base frequency.
- If XE operation is enabled, the maximum resolved bus ratio is given in MSR\_PERF\_STAT[44:40], it corresponds to the maximum XE operation frequency configured by BIOS.

XE operation of an Intel 64 processor is implementation specific. XE operation can be enabled only by BIOS. If MSR\_PERF\_STAT[31] is set, XE operation is enabled. The MSR\_PERF\_STAT[31] field is read-only.

## 18.18 IA32\_PERF\_CAPABILITIES MSR ENUMERATION

The layout of IA32\_PERF\_CAPABILITIES MSR is shown in Figure 18-49, it provides enumeration of a variety of interfaces:

- IA32\_PERF\_CAPABILITIES.LBR\_FMT[bits 5:0]: encodes the LBR format, details are described in Section 17.4.8.1.
- IA32\_PERF\_CAPABILITIES.PEBSTrap[6]: Trap/Fault-like indicator of PEBS recording assist, see Section 18.4.4.2.

- IA32\_PERF\_CAPABILITIES.PEBSArchRegs[7]: Indicator of PEBS assist save architectural registers, see Section 18.4.4.2.
- IA32\_PERF\_CAPABILITIES.PEBS\_FMT[bits 11:8]: Specifies the encoding of the layout of PEBS records, see Section 18.4.4.2.
- IA32\_PERF\_CAPABILITIES.SMM\_FRZ[12]: Indicates IA32\_DEBUGCTL.FREEZE\_WHILE\_SMM is supported if 1, see Section 18.18.1.
- IA32\_PERF\_CAPABILITIES.FULL\_WRITE[13]: Indicates the processor supports IA32\_A\_PMCx interface for updating bits 32 and above of IA32\_PMCx, see Section 18.2.5.

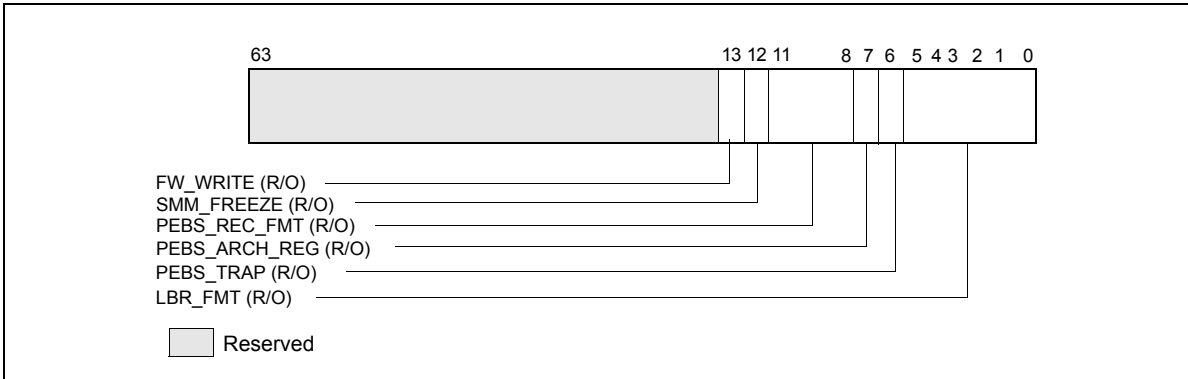


Figure 18-49. Layout of IA32\_PERF\_CAPABILITIES MSR

### 18.18.1 Filtering of SMM Handler Overhead

When performance monitoring facilities and/or branch profiling facilities (see Section 17.5, “Last Branch, Interrupt, and Exception Recording (Intel® Core™ 2 Duo and Intel® Atom™ Processors)”) are enabled, these facilities capture event counts, branch records and branch trace messages occurring in a logical processor. The occurrence of interrupts, instruction streams due to various interrupt handlers all contribute to the results recorded by these facilities.

If CPUID.01H:ECX.PDCM[bit 15] is 1, the processor supports the IA32\_PERF\_CAPABILITIES MSR. If IA32\_PERF\_CAPABILITIES.FREEZE\_WHILE\_SMM[Bit 12] is 1, the processor supports the ability for system software using performance monitoring and/or branch profiling facilities to filter out the effects of servicing system management interrupts.

If the FREEZE\_WHILE\_SMM capability is enabled on a logical processor and after an SMI is delivered, the processor will clear all the enable bits of IA32\_PERF\_GLOBAL\_CTRL, save a copy of the content of IA32\_DEBUGCTL and disable LBR, BTF, TR, and BTS fields of IA32\_DEBUGCTL before transferring control to the SMI handler.

The enable bits of IA32\_PERF\_GLOBAL\_CTRL will be set to 1, the saved copy of IA32\_DEBUGCTL prior to SMI delivery will be restored, after the SMI handler issues RSM to complete its servicing.

It is the responsibility of the SMM code to ensure the state of the performance monitoring and branch profiling facilities are preserved upon entry or until prior to exiting the SMM. If any of this state is modified due to actions by the SMM code, the SMM code is required to restore such state to the values present at entry to the SMM handler.

System software is allowed to set IA32\_DEBUGCTL.FREEZE\_WHILE\_SMM\_EN[bit 14] to 1 only supported as indicated by IA32\_PERF\_CAPABILITIES.FREEZE\_WHILE\_SMM[Bit 12] reporting 1.

## 18.19 PERFORMANCE MONITORING AND DUAL-CORE TECHNOLOGY

The performance monitoring capability of dual-core processors duplicates the microarchitectural resources of a single-core processor implementation. Each processor core has dedicated performance monitoring resources.

In the case of Pentium D processor, each logical processor is associated with dedicated resources for performance monitoring. In the case of Pentium processor Extreme edition, each processor core has dedicated resources, but



two logical processors in the same core share performance monitoring resources (see Section 18.16, “Performance Monitoring and Intel Hyper-Threading Technology in Processors Based on Intel NetBurst® Microarchitecture”).

## 18.20 PERFORMANCE MONITORING ON 64-BIT INTEL XEON PROCESSOR MP WITH UP TO 8-MBYTE L3 CACHE

The 64-bit Intel Xeon processor MP with up to 8-MByte L3 cache has a CPUID signature of family [0FH], model [03H or 04H]. Performance monitoring capabilities available to Pentium 4 and Intel Xeon processors with the same values (see Section 18.1 and Section 18.16) apply to the 64-bit Intel Xeon processor MP with an L3 cache.

The level 3 cache is connected between the system bus and IOQ through additional control logic. See Figure 18-50.

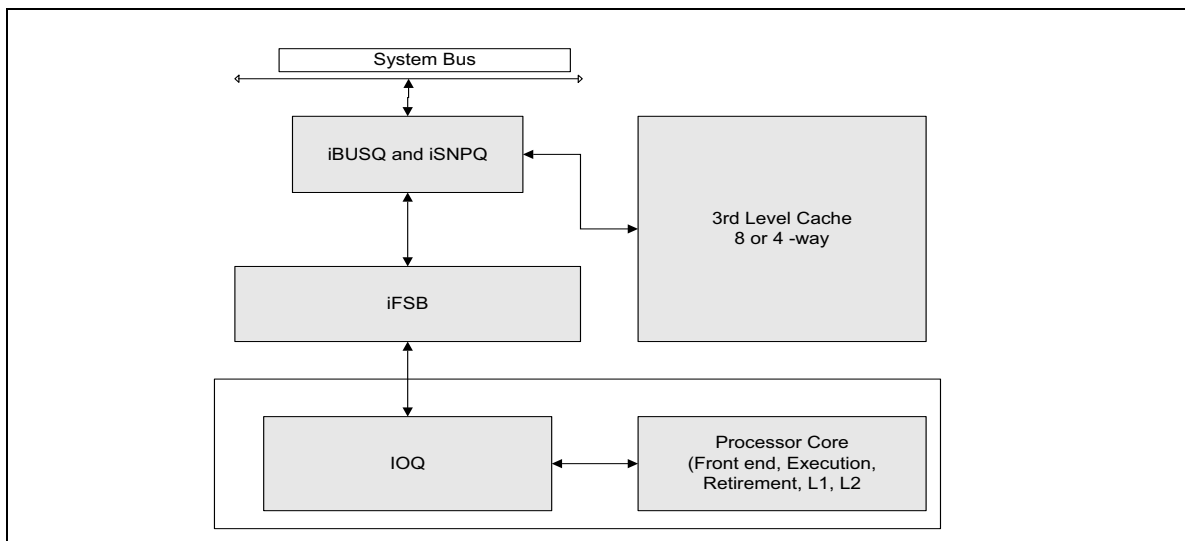


Figure 18-50. Block Diagram of 64-bit Intel Xeon Processor MP with 8-MByte L3

Additional performance monitoring capabilities and facilities unique to 64-bit Intel Xeon processor MP with an L3 cache are described in this section. The facility for monitoring events consists of a set of dedicated model-specific registers (MSRs), each dedicated to a specific event. Programming of these MSRs requires using RDMSR/WRMSR instructions with 64-bit values.

The lower 32-bits of the MSRs at addresses 107CC through 107D3 are treated as 32 bit performance counter registers. These performance counters can be accessed using RDPNC instruction with the index starting from 18 through 25. The EDX register returns zero when reading these 8 PMCs.

The performance monitoring capabilities consist of four events. These are:

- **IBUSQ event** — This event detects the occurrence of micro-architectural conditions related to the iBUSQ unit. It provides two MSRs: MSR\_IFSB\_IBUSQ0 and MSR\_IFSB\_IBUSQ1. Configure sub-event qualification and enable/disable functions using the high 32 bits of these MSRs. The low 32 bits act as a 32-bit event counter. Counting starts after software writes a non-zero value to one or more of the upper 32 bits. See Figure 18-51.



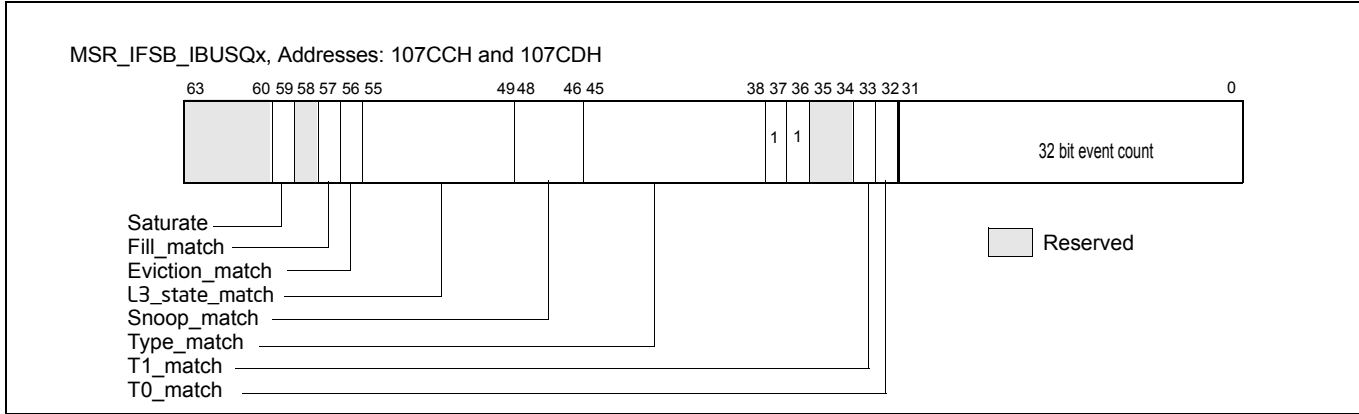


Figure 18-51. MSR\_IFSB\_IBUSQx, Addresses: 107CCH and 107CDH

- ISNPQ event** — This event detects the occurrence of microarchitectural conditions related to the iSNPQ unit. It provides two MSRs: MSR\_IFSB\_ISNPQ0 and MSR\_IFSB\_ISNPQ1. Configure sub-event qualifications and enable/disable functions using the high 32 bits of the MSRs. The low 32-bits act as a 32-bit event counter. Counting starts after software writes a non-zero value to one or more of the upper 32-bits. See Figure 18-52.

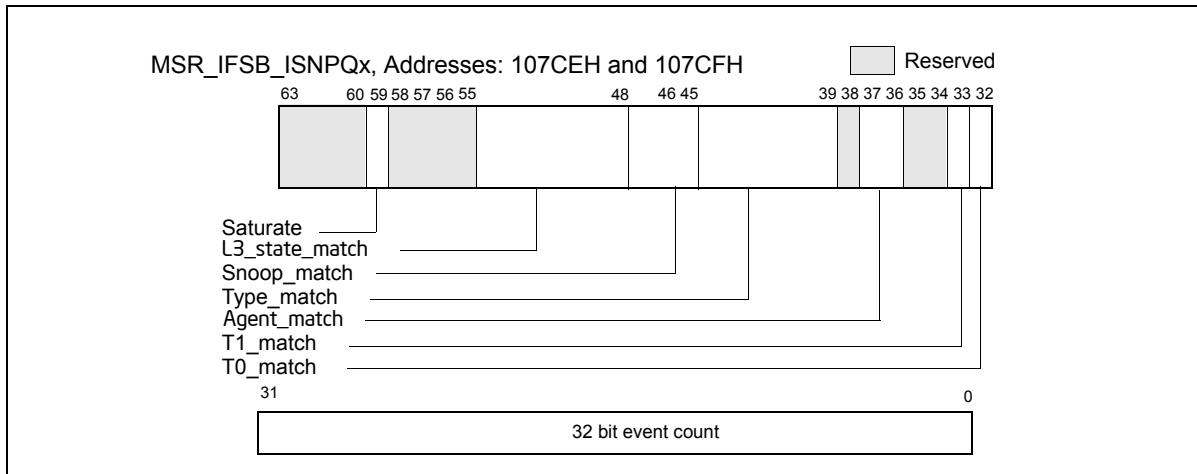


Figure 18-52. MSR\_IFSB\_ISNPQx, Addresses: 107CEH and 107CFH

- EFSB event** — This event can detect the occurrence of micro-architectural conditions related to the iFSB unit or system bus. It provides two MSRs: MSR\_EFSB\_DRDY0 and MSR\_EFSB\_DRDY1. Configure sub-event qualifications and enable/disable functions using the high 32 bits of the 64-bit MSR. The low 32-bit act as a 32-bit event counter. Counting starts after software writes a non-zero value to one or more of the qualification bits in the upper 32-bits of the MSR. See Figure 18-53.

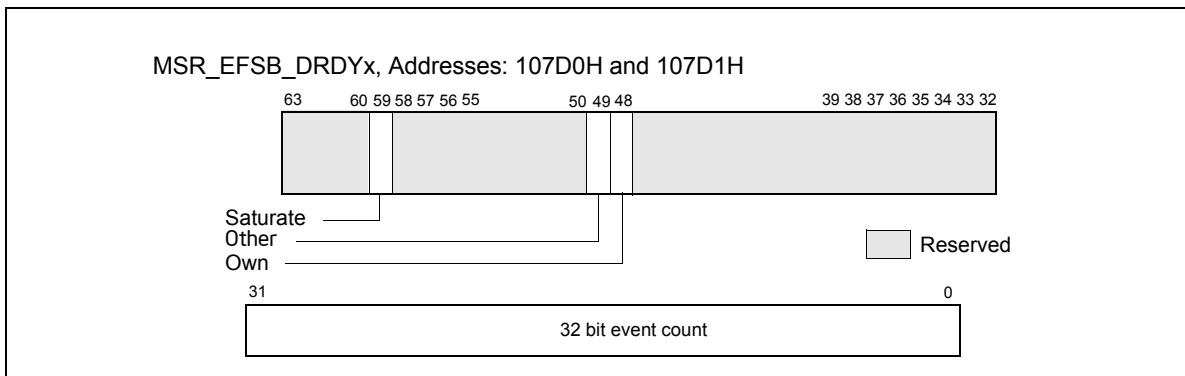


Figure 18-53. MSR\_EFSB\_DRDYx, Addresses: 107D0H and 107D1H

- IBUSQ Latency event** — This event accumulates weighted cycle counts for latency measurement of transactions in the iBUSQ unit. The count is enabled by setting MSR\_IFSB\_CTRL6[bit 26] to 1; the count freezes after software sets MSR\_IFSB\_CTRL6[bit 26] to 0. MSR\_IFSB\_CNTR7 acts as a 64-bit event counter for this event. See Figure 18-54.

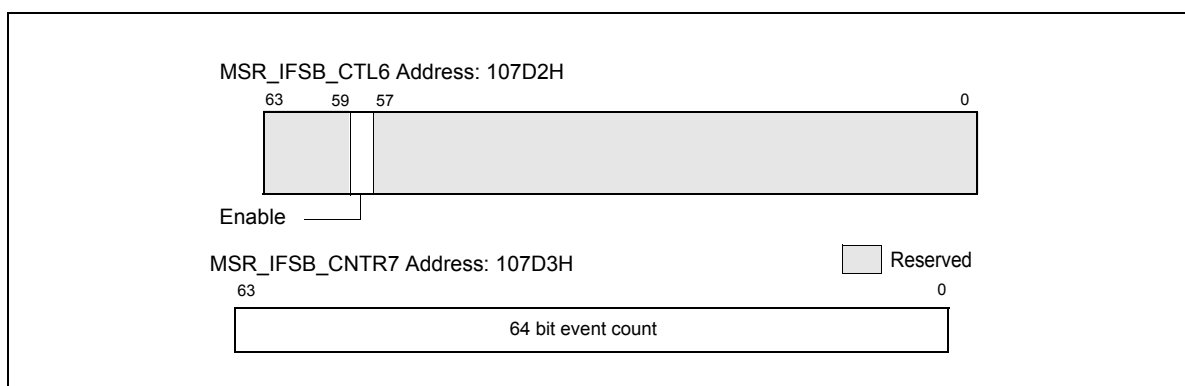


Figure 18-54. MSR\_IFSB\_CTL6, Address: 107D2H;  
 MSR\_IFSB\_CNTR7, Address: 107D3H

## 18.21 PERFORMANCE MONITORING ON L3 AND CACHING BUS CONTROLLER SUB-SYSTEMS

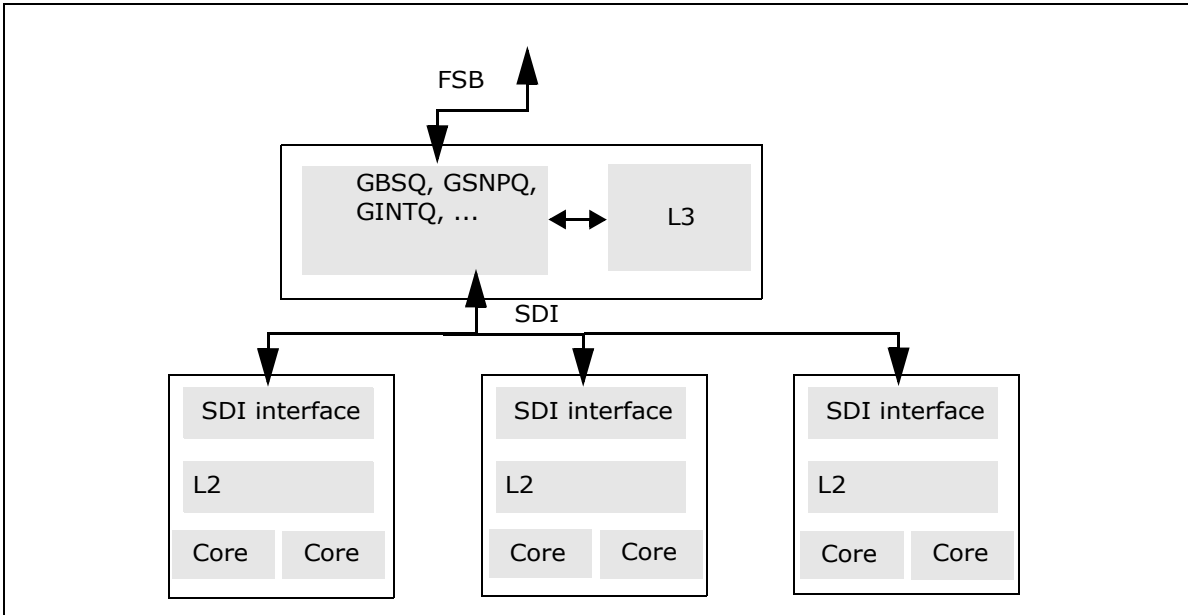
The Intel Xeon processor 7400 series and Dual-Core Intel Xeon processor 7100 series employ a distinct L3/caching bus controller sub-system. These sub-system have a unique set of performance monitoring capability and programming interfaces that are largely common between these two processor families.

Intel Xeon processor 7400 series are based on 45 nm enhanced Intel Core microarchitecture. The CPUID signature is indicated by DisplayFamily\_DisplayModel value of 06\_1DH (see CPUID instruction in Chapter 3, "Instruction Set Reference, A-L" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*). Intel Xeon processor 7400 series have six processor cores that share an L3 cache.

Dual-Core Intel Xeon processor 7100 series are based on Intel NetBurst microarchitecture, have a CPUID signature of family [0FH], model [06H] and a unified L3 cache shared between two cores. Each core in an Intel Xeon processor 7100 series supports Intel Hyper-Threading Technology, providing two logical processors per core.

Both Intel Xeon processor 7400 series and Intel Xeon processor 7100 series support multi-processor configurations using system bus interfaces. In Intel Xeon processor 7400 series, the L3/caching bus controller sub-system

provides three Simple Direct Interface (SDI) to service transactions originated the XQ-replacement SDI logic in each dual-core modules. In Intel Xeon processor 7100 series, the IOQ logic in each processor core is replaced with a Simple Direct Interface (SDI) logic. The L3 cache is connected between the system bus and the SDI through additional control logic. See Figure 18-55 for the block configuration of six processor cores and the L3/Caching bus controller sub-system in Intel Xeon processor 7400 series. Figure 18-55 shows the block configuration of two processor cores (four logical processors) and the L3/Caching bus controller sub-system in Intel Xeon processor 7100 series.



**Figure 18-55. Block Diagram of Intel Xeon Processor 7400 Series**

Almost all of the performance monitoring capabilities available to processor cores with the same CPUID signatures (see Section 18.1 and Section 18.16) apply to Intel Xeon processor 7100 series. The MSRs used by performance monitoring interface are shared between two logical processors in the same processor core.

The performance monitoring capabilities available to processor with DisplayFamily\_DisplayModel signature 06\_17H also apply to Intel Xeon processor 7400 series. Each processor core provides its own set of MSRs for performance monitoring interface.

The IOQ\_allocation and IOQ\_active\_entries events are not supported in Intel Xeon processor 7100 series and 7400 series. Additional performance monitoring capabilities applicable to the L3/caching bus controller sub-system are described in this section.

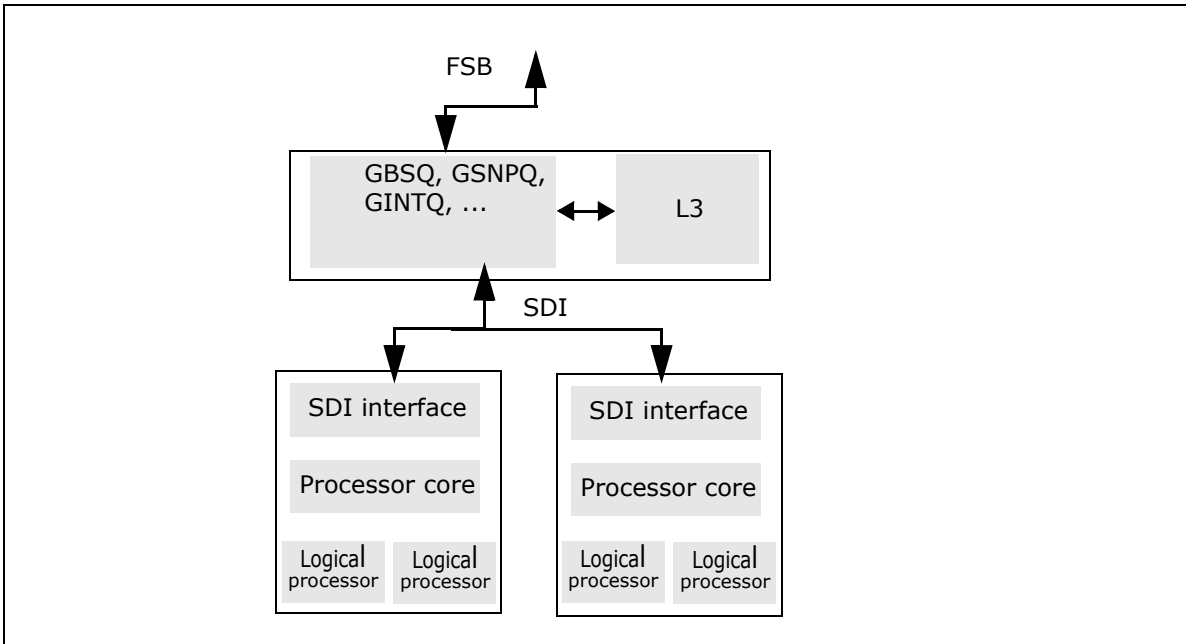


Figure 18-56. Block Diagram of Intel Xeon Processor 7100 Series

### 18.21.1 Overview of Performance Monitoring with L3/Caching Bus Controller

The facility for monitoring events consists of a set of dedicated model-specific registers (MSRs). There are eight event select/counting MSRs that are dedicated to counting events associated with specified microarchitectural conditions. Programming of these MSRs requires using RDMSR/WRMSR instructions with 64-bit values. In addition, an MSR MSR\_EMON\_L3\_GL\_CTL provides simplified interface to control freezing, resetting, re-enabling operation of any combination of these event select/counting MSRs.

The eight MSRs dedicated to count occurrences of specific conditions are further divided to count three sub-classes of microarchitectural conditions:

- Two MSRs (MSR\_EMON\_L3\_CTR\_CTL0 and MSR\_EMON\_L3\_CTR\_CTL1) are dedicated to counting GBSQ events. Up to two GBSQ events can be programmed and counted simultaneously.
- Two MSRs (MSR\_EMON\_L3\_CTR\_CTL2 and MSR\_EMON\_L3\_CTR\_CTL3) are dedicated to counting GSNPQ events. Up to two GSNPQ events can be programmed and counted simultaneously.
- Four MSRs (MSR\_EMON\_L3\_CTR\_CTL4, MSR\_EMON\_L3\_CTR\_CTL5, MSR\_EMON\_L3\_CTR\_CTL6, and MSR\_EMON\_L3\_CTR\_CTL7) are dedicated to counting external bus operations.

The bit fields in each of eight MSRs share the following common characteristics:

- Bits 63:32 is the event control field that includes an event mask and other bit fields that control counter operation. The event mask field specifies details of the microarchitectural condition, and its definition differs across GBSQ, GSNPQ, FSB.
- Bits 31:0 is the event count field. If the specified condition is met during each relevant clock domain of the event logic, the matched condition signals the counter logic to increment the associated event count field. The lower 32-bits of these 8 MSRs at addresses 107CC through 107D3 are treated as 32 bit performance counter registers.

In Dual-Core Intel Xeon processor 7100 series, the uncore performance counters can be accessed using RDPMC instruction with the index starting from 18 through 25. The EDX register returns zero when reading these 8 PMCs.

In Intel Xeon processor 7400 series, RDPMC with ECX between 2 and 9 can be used to access the eight uncore performance counter/control registers.

### 18.21.2 GBSQ Event Interface

The layout of MSR\_EMON\_L3\_CTR\_CTL0 and MSR\_EMON\_L3\_CTR\_CTL1 is given in Figure 18-57. Counting starts after software writes a non-zero value to one or more of the upper 32 bits.

The event mask field (bits 58:32) consists of the following eight attributes:

- Agent\_Select (bits 35:32): The definition of this field differs slightly between Intel Xeon processor 7100 and 7400.

For Intel Xeon processor 7100 series, each bit specifies a logical processor in the physical package. The lower two bits corresponds to two logical processors in the first processor core, the upper two bits corresponds to two logical processors in the second processor core. 0FH encoding matches transactions from any logical processor.

For Intel Xeon processor 7400 series, each bit of [34:32] specifies the SDI logic of a dual-core module as the originator of the transaction. A value of 0111B in bits [35:32] specifies transaction from any processor core.

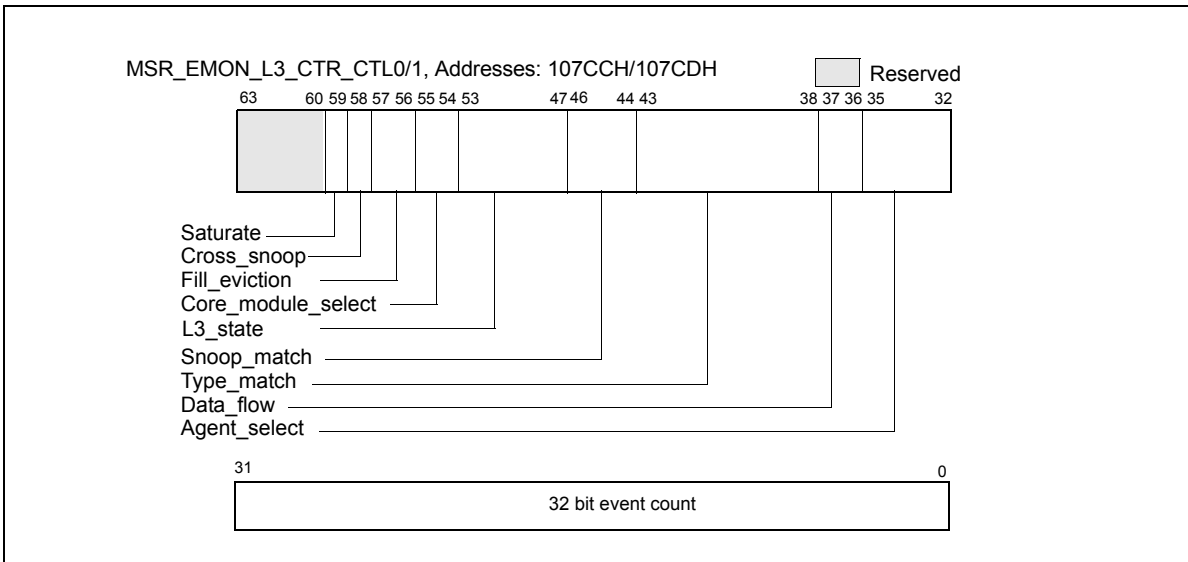


Figure 18-57. MSR\_EMON\_L3\_CTR\_CTL0/1, Addresses: 107CCH/107CDH

- Data\_Flow (bits 37:36): Bit 36 specifies demand transactions, bit 37 specifies prefetch transactions.
- Type\_Match (bits 43:38): Specifies transaction types. If all six bits are set, event count will include all transaction types.
- Snoop\_Match (bits 46:44): The three bits specify (in ascending bit position) clean snoop result, HIT snoop result, and HITM snoop results respectively.
- L3\_State (bits 53:47): Each bit specifies an L2 coherency state.
- Core\_Module\_Select (bits 55:54): The valid encodings for L3 lookup differ slightly between Intel Xeon processor 7100 and 7400.

For Intel Xeon processor 7100 series,

- 00B: Match transactions from any core in the physical package
- 01B: Match transactions from this core only
- 10B: Match transactions from the other core in the physical package
- 11B: Match transaction from both cores in the physical package

For Intel Xeon processor 7400 series,

- 00B: Match transactions from any dual-core module in the physical package
- 01B: Match transactions from this dual-core module only
- 10B: Match transactions from either one of the other two dual-core modules in the physical package

- 11B: Match transaction from more than one dual-core modules in the physical package
- Fill\_Eviction (bits 57:56): The valid encodings are
  - 00B: Match any transactions
  - 01B: Match transactions that fill L3
  - 10B: Match transactions that fill L3 without an eviction
  - 11B: Match transaction fill L3 with an eviction
- Cross\_Snoop (bit 58): The encodings are
  - 0B: Match any transactions
  - 1B: Match cross snoop transactions

For each counting clock domain, if all eight attributes match, event logic signals to increment the event count field.

### 18.21.3 GSNPQ Event Interface

The layout of MSR\_EMON\_L3\_CTR\_CTL2 and MSR\_EMON\_L3\_CTR\_CTL3 is given in Figure 18-58. Counting starts after software writes a non-zero value to one or more of the upper 32 bits.

The event mask field (bits 58:32) consists of the following six attributes:

- Agent\_Select (bits 37:32): The definition of this field differs slightly between Intel Xeon processor 7100 and 7400.
  - For Intel Xeon processor 7100 series, each of the lowest 4 bits specifies a logical processor in the physical package. The lowest two bits corresponds to two logical processors in the first processor core, the next two bits corresponds to two logical processors in the second processor core. Bit 36 specifies other symmetric agent transactions. Bit 37 specifies central agent transactions. 3FH encoding matches transactions from any logical processor.
  - For Intel Xeon processor 7400 series, each of the lowest 3 bits specifies a dual-core module in the physical package. Bit 37 specifies central agent transactions.
- Type\_Match (bits 43:38): Specifies transaction types. If all six bits are set, event count will include any transaction types.
- Snoop\_Match: (bits 46:44): The three bits specify (in ascending bit position) clean snoop result, HIT snoop result, and HITM snoop results respectively.
- L2\_State (bits 53:47): Each bit specifies an L3 coherency state.
- Core\_Module\_Select (bits 56:54): Bit 56 enables Core\_Module\_Select matching. If bit 56 is clear, Core\_Module\_Select encoding is ignored. The valid encodings for the lower two bits (bit 55, 54) differ slightly between Intel Xeon processor 7100 and 7400.

For Intel Xeon processor 7100 series, if bit 56 is set, the valid encodings for the lower two bits (bit 55, 54) are

- 00B: Match transactions from only one core (irrespective which core) in the physical package
- 01B: Match transactions from this core and not the other core
- 10B: Match transactions from the other core in the physical package, but not this core
- 11B: Match transaction from both cores in the physical package

For Intel Xeon processor 7400 series, if bit 56 is set, the valid encodings for the lower two bits (bit 55, 54) are

- 00B: Match transactions from only one dual-core module (irrespective which module) in the physical package.
- 01B: Match transactions from one or more dual-core modules.
- 10B: Match transactions from two or more dual-core modules.
- 11B: Match transaction from all three dual-core modules in the physical package.

- Block\_Snoop (bit 57): specifies blocked snoop.

For each counting clock domain, if all six attributes match, event logic signals to increment the event count field.

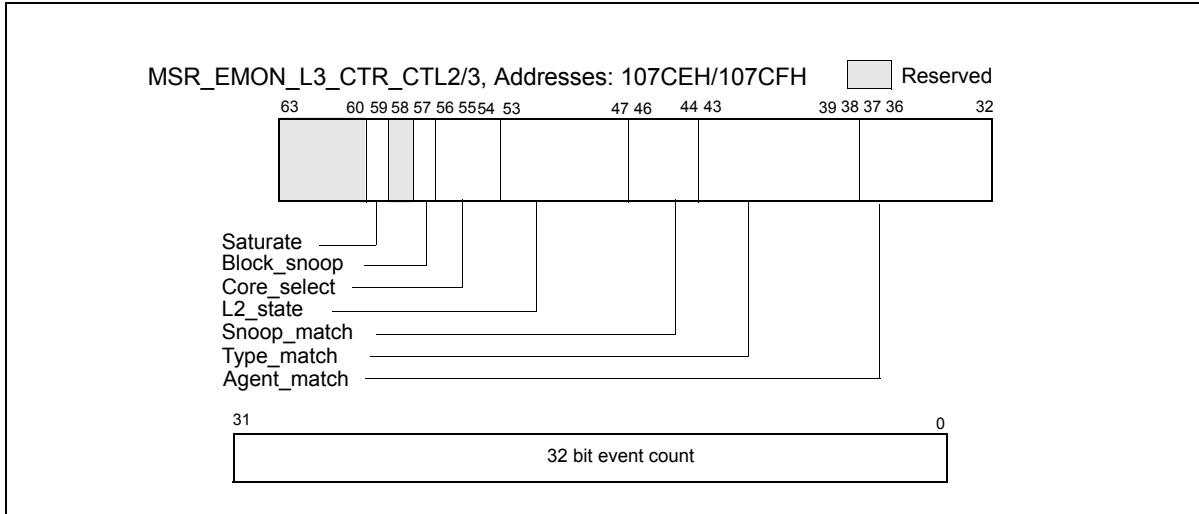


Figure 18-58. MSR\_EMON\_L3\_CTR\_CTL2/3, Addresses: 107CEH/107CFH

### 18.21.4 FSB Event Interface

The layout of MSR\_EMON\_L3\_CTR\_CTL4 through MSR\_EMON\_L3\_CTR\_CTL7 is given in Figure 18-59. Counting starts after software writes a non-zero value to one or more of the upper 32 bits.

The event mask field (bits 58:32) is organized as follows:

- Bit 58: must set to 1.
- FSB\_Submask (bits 57:32): Specifies FSB-specific sub-event mask.

The FSB sub-event mask defines a set of independent attributes. The event logic signals to increment the associated event count field if one of the attribute matches. Some of the sub-event mask bit counts durations. A duration event increments at most once per cycle.

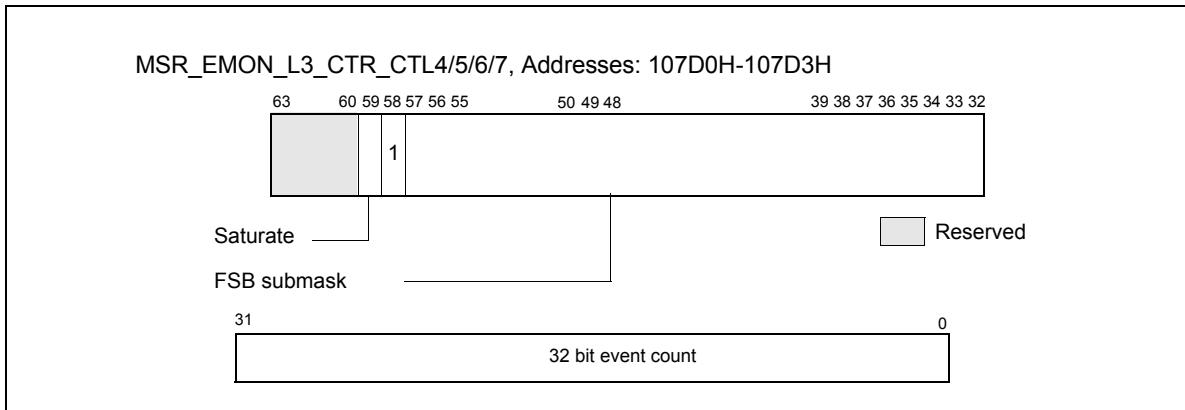


Figure 18-59. MSR\_EMON\_L3\_CTR\_CTL4/5/6/7, Addresses: 107D0H-107D3H

#### 18.21.4.1 FSB Sub-Event Mask Interface

- FSB\_type (bit 37:32): Specifies different FSB transaction types originated from this physical package
- FSB\_L\_clear (bit 38): Count clean snoop results from any source for transaction originated from this physical package
- FSB\_L\_hit (bit 39): Count HIT snoop results from any source for transaction originated from this physical package

- FSB\_L\_hitm (bit 40): Count HITM snoop results from any source for transaction originated from this physical package
- FSB\_L\_defer (bit 41): Count DEFER responses to this processor's transactions
- FSB\_L\_retry (bit 42): Count RETRY responses to this processor's transactions
- FSB\_L\_snoop\_stall (bit 43): Count snoop stalls to this processor's transactions
- FSB\_DBSY (bit 44): Count DBSY assertions by this processor (without a concurrent DRDY)
- FSB\_DRDY (bit 45): Count DRDY assertions by this processor
- FSB\_BNR (bit 46): Count BNR assertions by this processor
- FSB\_IOQ\_empty (bit 47): Counts each bus clocks when the IOQ is empty
- FSB\_IOQ\_full (bit 48): Counts each bus clocks when the IOQ is full
- FSB\_IOQ\_active (bit 49): Counts each bus clocks when there is at least one entry in the IOQ
- FSB\_WW\_data (bit 50): Counts back-to-back write transaction's data phase.
- FSB\_WW\_issue (bit 51): Counts back-to-back write transaction request pairs issued by this processor.
- FSB\_WR\_issue (bit 52): Counts back-to-back write-read transaction request pairs issued by this processor.
- FSB\_RW\_issue (bit 53): Counts back-to-back read-write transaction request pairs issued by this processor.
- FSB\_other\_DBSY (bit 54): Count DBSY assertions by another agent (without a concurrent DRDY)
- FSB\_other\_DRDY (bit 55): Count DRDY assertions by another agent
- FSB\_other\_snoop\_stall (bit 56): Count snoop stalls on the FSB due to another agent
- FSB\_other\_BNR (bit 57): Count BNR assertions from another agent

### 18.21.5 Common Event Control Interface

The MSR\_EMON\_L3\_GL\_CTL MSR provides simplified access to query overflow status of the GBSQ, GSNPQ, FSB event counters. It also provides control bit fields to freeze, unfreeze, or reset those counters. The following bit fields are supported:

- GL\_freeze\_cmd (bit 0): Freeze the event counters specified by the GL\_event\_select field.
- GL\_unfreeze\_cmd (bit 1): Unfreeze the event counters specified by the GL\_event\_select field.
- GL\_reset\_cmd (bit 2): Clear the event count field of the event counters specified by the GL\_event\_select field. The event select field is not affected.
- GL\_event\_select (bit 23:16): Selects one or more event counters to subject to specified command operations indicated by bits 2:0. Bit 16 corresponds to MSR\_EMON\_L3\_CTR\_CTL0, bit 23 corresponds to MSR\_EMON\_L3\_CTR\_CTL7.
- GL\_event\_status (bit 55:48): Indicates the overflow status of each event counters. Bit 48 corresponds to MSR\_EMON\_L3\_CTR\_CTL0, bit 55 corresponds to MSR\_EMON\_L3\_CTR\_CTL7.

In the event control field (bits 63:32) of each MSR, if the saturate control (bit 59, see Figure 18-57 for example) is set, the event logic forces the value FFFF\_FFFFH into the event count field instead of incrementing it.

## 18.22 PERFORMANCE MONITORING (P6 FAMILY PROCESSOR)

The P6 family processors provide two 40-bit performance counters, allowing two types of events to be monitored simultaneously. These can either count events or measure duration. When counting events, a counter increments each time a specified event takes place or a specified number of events takes place. When measuring duration, it counts the number of processor clocks that occur while a specified condition is true. The counters can count events or measure durations that occur at any privilege level.

Table 19-37, Chapter 19, lists the events that can be counted with the P6 family performance monitoring counters.



**NOTE**

The performance-monitoring events listed in Chapter 19 are intended to be used as guides for performance tuning. Counter values reported are not guaranteed to be accurate and should be used as a relative guide for tuning. Known discrepancies are documented where applicable.

The performance-monitoring counters are supported by four MSR: the performance event select MSRs (PerfEvtSel0 and PerfEvtSel1) and the performance counter MSRs (PerfCtr0 and PerfCtr1). These registers can be read from and written to using the RDMSR and WRMSR instructions, respectively. They can be accessed using these instructions only when operating at privilege level 0. The PerfCtr0 and PerfCtr1 MSRs can be read from any privilege level using the RDPMC (read performance-monitoring counters) instruction.

**NOTE**

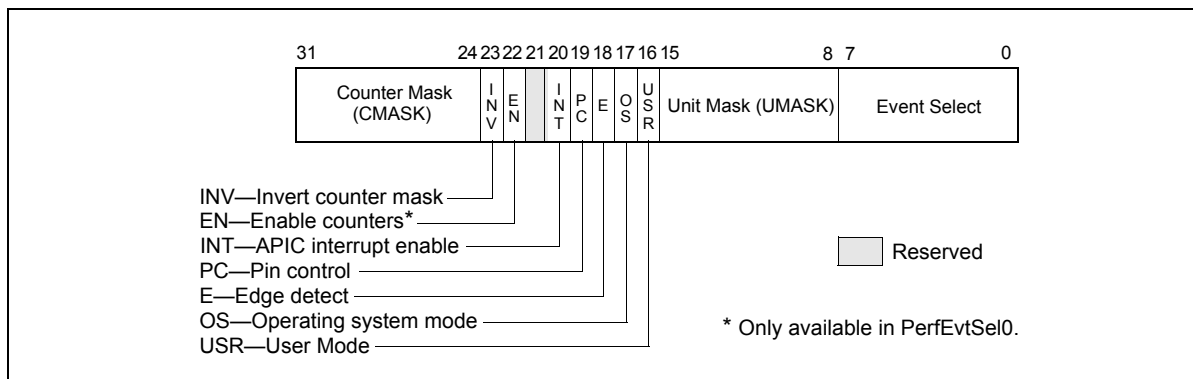
The PerfEvtSel0, PerfEvtSel1, PerfCtr0, and PerfCtr1 MSRs and the events listed in Table 19-37 are model-specific for P6 family processors. They are not guaranteed to be available in other IA-32 processors.

**18.22.1 PerfEvtSel0 and PerfEvtSel1 MSRs**

The PerfEvtSel0 and PerfEvtSel1 MSRs control the operation of the performance-monitoring counters, with one register used to set up each counter. They specify the events to be counted, how they should be counted, and the privilege levels at which counting should take place. Figure 18-60 shows the flags and fields in these MSRs.

The functions of the flags and fields in the PerfEvtSel0 and PerfEvtSel1 MSRs are as follows:

- **Event select field (bits 0 through 7)** — Selects the event logic unit to detect certain microarchitectural conditions (see Table 19-37, for a list of events and their 8-bit codes).
- **Unit mask (UMASK) field (bits 8 through 15)** — Further qualifies the event logic unit selected in the event select field to detect a specific microarchitectural condition. For example, for some cache events, the mask is used as a MESI-protocol qualifier of cache states (see Table 19-37).



**Figure 18-60. PerfEvtSel0 and PerfEvtSel1 MSRs**

- **USR (user mode) flag (bit 16)** — Specifies that events are counted only when the processor is operating at privilege levels 1, 2 or 3. This flag can be used in conjunction with the OS flag.
- **OS (operating system mode) flag (bit 17)** — Specifies that events are counted only when the processor is operating at privilege level 0. This flag can be used in conjunction with the USR flag.
- **E (edge detect) flag (bit 18)** — Enables (when set) edge detection of events. The processor counts the number of deasserted to asserted transitions of any condition that can be expressed by the other fields. The mechanism is limited in that it does not permit back-to-back assertions to be distinguished. This mechanism allows software to measure not only the fraction of time spent in a particular state, but also the average length of time spent in such a state (for example, the time spent waiting for an interrupt to be serviced).

- **PC (pin control) flag (bit 19)** — When set, the processor toggles the PM*i* pins and increments the counter when performance-monitoring events occur; when clear, the processor toggles the PM*i* pins when the counter overflows. The toggling of a pin is defined as assertion of the pin for a single bus clock followed by deassertion.
- **INT (APIC interrupt enable) flag (bit 20)** — When set, the processor generates an exception through its local APIC on counter overflow.
- **EN (Enable Counters) Flag (bit 22)** — This flag is only present in the PerfEvtSel0 MSR. When set, performance counting is enabled in both performance-monitoring counters; when clear, both counters are disabled.
- **INV (invert) flag (bit 23)** — When set, inverts the counter-mask (CMASK) comparison, so that both greater than or equal to and less than comparisons can be made (0: greater than or equal; 1: less than). Note if counter-mask is programmed to zero, INV flag is ignored.
- **Counter mask (CMASK) field (bits 24 through 31)** — When nonzero, the processor compares this mask to the number of events counted during a single cycle. If the event count is greater than or equal to this mask, the counter is incremented by one. Otherwise the counter is not incremented. This mask can be used to count events only if multiple occurrences happen per clock (for example, two or more instructions retired per clock). If the counter-mask field is 0, then the counter is incremented each cycle by the number of events that occurred that cycle.

### 18.22.2 PerfCtr0 and PerfCtr1 MSRs

The performance-counter MSRs (PerfCtr0 and PerfCtr1) contain the event or duration counts for the selected events being counted. The RDPMC instruction can be used by programs or procedures running at any privilege level and in virtual-8086 mode to read these counters. The PCE flag in control register CR4 (bit 8) allows the use of this instruction to be restricted to only programs and procedures running at privilege level 0.

The RDPMC instruction is not serializing or ordered with other instructions. Thus, it does not necessarily wait until all previous instructions have been executed before reading the counter. Similarly, subsequent instructions may begin execution before the RDPMC instruction operation is performed.

Only the operating system, executing at privilege level 0, can directly manipulate the performance counters, using the RDMSR and WRMSR instructions. A secure operating system would clear the PCE flag during system initialization to disable direct user access to the performance-monitoring counters, but provide a user-accessible programming interface that emulates the RDPMC instruction.

The WRMSR instruction cannot arbitrarily write to the performance-monitoring counter MSRs (PerfCtr0 and PerfCtr1). Instead, the lower-order 32 bits of each MSR may be written with any value, and the high-order 8 bits are sign-extended according to the value of bit 31. This operation allows writing both positive and negative values to the performance counters.

### 18.22.3 Starting and Stopping the Performance-Monitoring Counters

The performance-monitoring counters are started by writing valid setup information in the PerfEvtSel0 and/or PerfEvtSel1 MSRs and setting the enable counters flag in the PerfEvtSel0 MSR. If the setup is valid, the counters begin counting following the execution of a WRMSR instruction that sets the enable counter flag. The counters can be stopped by clearing the enable counters flag or by clearing all the bits in the PerfEvtSel0 and PerfEvtSel1 MSRs. Counter 1 alone can be stopped by clearing the PerfEvtSel1 MSR.

### 18.22.4 Event and Time-Stamp Monitoring Software

To use the performance-monitoring counters and time-stamp counter, the operating system needs to provide an event-monitoring device driver. This driver should include procedures for handling the following operations:

- Feature checking
- Initialize and start counters
- Stop counters
- Read the event counters
- Read the time-stamp counter

The event monitor feature determination procedure must check whether the current processor supports the performance-monitoring counters and time-stamp counter. This procedure compares the family and model of the processor returned by the CPUID instruction with those of processors known to support performance monitoring. (The Pentium and P6 family processors support performance counters.) The procedure also checks the MSR and TSC flags returned to register EDX by the CPUID instruction to determine if the MSRs and the RDTSC instruction are supported.

The initialize and start counters procedure sets the PerfEvtSel0 and/or PerfEvtSel1 MSRs for the events to be counted and the method used to count them and initializes the counter MSRs (PerfCtr0 and PerfCtr1) to starting counts. The stop counters procedure stops the performance counters (see Section 18.22.3, "Starting and Stopping the Performance-Monitoring Counters").

The read counters procedure reads the values in the PerfCtr0 and PerfCtr1 MSRs, and a read time-stamp counter procedure reads the time-stamp counter. These procedures would be provided in lieu of enabling the RDTSC and RDPMC instructions that allow application code to read the counters.

### 18.22.5 Monitoring Counter Overflow

The P6 family processors provide the option of generating a local APIC interrupt when a performance-monitoring counter overflows. This mechanism is enabled by setting the interrupt enable flag in either the PerfEvtSel0 or the PerfEvtSel1 MSR. The primary use of this option is for statistical performance sampling.

To use this option, the operating system should do the following things on the processor for which performance events are required to be monitored:

- Provide an interrupt vector for handling the counter-overflow interrupt.
- Initialize the APIC PERF local vector entry to enable handling of performance-monitor counter overflow events.
- Provide an entry in the IDT that points to a stub exception handler that returns without executing any instructions.
- Provide an event monitor driver that provides the actual interrupt handler and modifies the reserved IDT entry to point to its interrupt routine.

When interrupted by a counter overflow, the interrupt handler needs to perform the following actions:

- Save the instruction pointer (EIP register), code-segment selector, TSS segment selector, counter values and other relevant information at the time of the interrupt.
- Reset the counter to its initial setting and return from the interrupt.

An event monitor application utility or another application program can read the information collected for analysis of the performance of the profiled application.

## 18.23 PERFORMANCE MONITORING (PENTIUM PROCESSORS)

The Pentium processor provides two 40-bit performance counters, which can be used to count events or measure duration. The counters are supported by three MSRs: the control and event select MSR (CESR) and the performance counter MSRs (CTR0 and CTR1). These can be read from and written to using the RDMSR and WRMSR instructions, respectively. They can be accessed using these instructions only when operating at privilege level 0.

Each counter has an associated external pin (PM0/BP0 and PM1/BP1), which can be used to indicate the state of the counter to external hardware.

## NOTES

The CESR, CTR0, and CTR1 MSRs and the events listed in Table 19-38 are model-specific for the Pentium processor.

The performance-monitoring events listed in Chapter 19 are intended to be used as guides for performance tuning. Counter values reported are not guaranteed to be accurate and should be used as a relative guide for tuning. Known discrepancies are documented where applicable.

### 18.23.1 Control and Event Select Register (CESR)

The 32-bit control and event select MSR (CESR) controls the operation of performance-monitoring counters CTR0 and CTR1 and the associated pins (see Figure 18-61). To control each counter, the CESR register contains a 6-bit event select field (ES0 and ES1), a pin control flag (PC0 and PC1), and a 3-bit counter control field (CC0 and CC1). The functions of these fields are as follows:

- **ES0 and ES1 (event select) fields (bits 0-5, bits 16-21)** — Selects (by entering an event code in the field) up to two events to be monitored. See Table 19-38 for a list of available event codes.

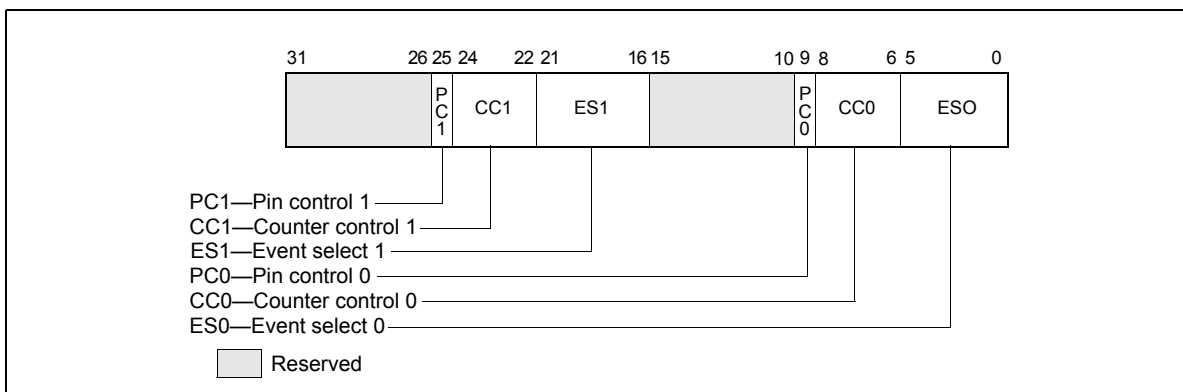


Figure 18-61. CESR MSR (Pentium Processor Only)

- **CC0 and CC1 (counter control) fields (bits 6-8, bits 22-24)** — Controls the operation of the counter. Control codes are as follows:

- 000 — Count nothing (counter disabled)
- 001 — Count the selected event while CPL is 0, 1, or 2
- 010 — Count the selected event while CPL is 3
- 011 — Count the selected event regardless of CPL
- 100 — Count nothing (counter disabled)
- 101 — Count clocks (duration) while CPL is 0, 1, or 2
- 110 — Count clocks (duration) while CPL is 3
- 111 — Count clocks (duration) regardless of CPL

The highest order bit selects between counting events and counting clocks (duration); the middle bit enables counting when the CPL is 3; and the low-order bit enables counting when the CPL is 0, 1, or 2.

- **PC0 and PC1 (pin control) flags (bits 9, 25)** — Selects the function of the external performance-monitoring counter pin (PM0/BP0 and PM1/BP1). Setting one of these flags to 1 causes the processor to assert its associated pin when the counter has overflowed; setting the flag to 0 causes the pin to be asserted when the counter has been incremented. These flags permit the pins to be individually programmed to indicate the

overflow or incremented condition. The external signalling of the event on the pins will lag the internal event by a few clocks as the signals are latched and buffered.

While a counter need not be stopped to sample its contents, it must be stopped and cleared or preset before switching to a new event. It is not possible to set one counter separately. If only one event needs to be changed, the CESR register must be read, the appropriate bits modified, and all bits must then be written back to CESR. At reset, all bits in the CESR register are cleared.

### 18.23.2 Use of the Performance-Monitoring Pins

When performance-monitor pins PM0/BP0 and/or PM1/BP1 are configured to indicate when the performance-monitor counter has incremented and an “occurrence event” is being counted, the associated pin is asserted (high) each time the event occurs. When a “duration event” is being counted, the associated PM pin is asserted for the entire duration of the event. When the performance-monitor pins are configured to indicate when the counter has overflowed, the associated PM pin is asserted when the counter has overflowed.

When the PM0/BP0 and/or PM1/BP1 pins are configured to signal that a counter has incremented, it should be noted that although the counters may increment by 1 or 2 in a single clock, the pins can only indicate that the event occurred. Moreover, since the internal clock frequency may be higher than the external clock frequency, a single external clock may correspond to multiple internal clocks.

A “count up to” function may be provided when the event pin is programmed to signal an overflow of the counter. Because the counters are 40 bits, a carry out of bit 39 indicates an overflow. A counter may be preset to a specific value less than  $2^{40} - 1$ . After the counter has been enabled and the prescribed number of events has transpired, the counter will overflow.

Approximately 5 clocks later, the overflow is indicated externally and appropriate action, such as signaling an interrupt, may then be taken.

The PM0/BP0 and PM1/BP1 pins also serve to indicate breakpoint matches during in-circuit emulation, during which time the counter increment or overflow function of these pins is not available. After RESET, the PM0/BP0 and PM1/BP1 pins are configured for performance monitoring, however a hardware debugger may reconfigure these pins to indicate breakpoint matches.

### 18.23.3 Events Counted

Events that performance-monitoring counters can be set to count and record (using CTR0 and CTR1) are divided in two categories: occurrence and duration:

- **Occurrence events** — Counts are incremented each time an event takes place. If PM0/BP0 or PM1/BP1 pins are used to indicate when a counter increments, the pins are asserted each clock counters increment. But if an event happens twice in one clock, the counter increments by 2 (the pins are asserted only once).
- **Duration events** — Counters increment the total number of clocks that the condition is true. When used to indicate when counters increment, PM0/BP0 and/or PM1/BP1 pins are asserted for the duration.

## 16. Updates to Chapter 22, Volume 3B

Change bars show changes to Chapter 22 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B: System Programming Guide, Part 2*.

-----  
Changes include the addition of section 22.39 covering the initial state of Pentium, Pentium Pro and Pentium 4 processors (this used to be table 9-1 in the previous manual, revision 058).

Intel 64 and IA-32 processors are binary compatible. Compatibility means that, within limited constraints, programs that execute on previous generations of processors will produce identical results when executed on later processors. The compatibility constraints and any implementation differences between the Intel 64 and IA-32 processors are described in this chapter.

Each new processor has enhanced the software visible architecture from that found in earlier Intel 64 and IA-32 processors. Those enhancements have been defined with consideration for compatibility with previous and future processors. This chapter also summarizes the compatibility considerations for those extensions.

## 22.1 PROCESSOR FAMILIES AND CATEGORIES

IA-32 processors are referred to in several different ways in this chapter, depending on the type of compatibility information being related, as described in the following:

- **IA-32 Processors** — All the Intel processors based on the Intel IA-32 Architecture, which include the 8086/88, Intel 286, Intel386, Intel486, Pentium, Pentium Pro, Pentium II, Pentium III, Pentium 4, and Intel Xeon processors.
- **32-bit Processors** — All the IA-32 processors that use a 32-bit architecture, which include the Intel386, Intel486, Pentium, Pentium Pro, Pentium II, Pentium III, Pentium 4, and Intel Xeon processors.
- **16-bit Processors** — All the IA-32 processors that use a 16-bit architecture, which include the 8086/88 and Intel 286 processors.
- **P6 Family Processors** — All the IA-32 processors that are based on the P6 microarchitecture, which include the Pentium Pro, Pentium II, and Pentium III processors.
- **Pentium® 4 Processors** — A family of IA-32 and Intel 64 processors that are based on the Intel NetBurst® microarchitecture.
- **Intel® Pentium® M Processors** — A family of IA-32 processors that are based on the Intel Pentium M processor microarchitecture.
- **Intel® Core™ Duo and Solo Processors** — Families of IA-32 processors that are based on an improved Intel Pentium M processor microarchitecture.
- **Intel® Xeon® Processors** — A family of IA-32 and Intel 64 processors that are based on the Intel NetBurst microarchitecture. This family includes the Intel Xeon processor and the Intel Xeon processor MP based on the Intel NetBurst microarchitecture. Intel Xeon processors 3000, 3100, 3200, 3300, 3200, 5100, 5200, 5300, 5400, 7200, 7300 series are based on Intel Core microarchitectures and support Intel 64 architecture.
- **Pentium® D Processors** — A family of dual-core Intel 64 processors that provides two processor cores in a physical package. Each core is based on the Intel NetBurst microarchitecture.
- **Pentium® Processor Extreme Editions** — A family of dual-core Intel 64 processors that provides two processor cores in a physical package. Each core is based on the Intel NetBurst microarchitecture and supports Intel Hyper-Threading Technology.
- **Intel® Core™ 2 Processor family**— A family of Intel 64 processors that are based on the Intel Core microarchitecture. Intel Pentium Dual-Core processors are also based on the Intel Core microarchitecture.
- **Intel® Atom™ Processors** — A family of IA-32 and Intel 64 processors. 45 nm Intel Atom processors are based on the Intel Atom microarchitecture. 32 nm Intel Atom processors are based on newer microarchitectures including the Silvermont microarchitecture and the Airmont microarchitecture. Each generation of Intel Atom processors can be identified by the CPUID's DisplayFamily\_DisplayModel signature; see Table 35-1 "CPUID Signature Values of DisplayFamily\_DisplayModel".

## 22.2 RESERVED BITS

Throughout this manual, certain bits are marked as reserved in many register and memory layout descriptions. When bits are marked as undefined or reserved, it is essential for compatibility with future processors that software treat these bits as having a future, though unknown effect. Software should follow these guidelines in dealing with reserved bits:

- Do not depend on the states of any reserved bits when testing the values of registers or memory locations that contain such bits. Mask out the reserved bits before testing.
- Do not depend on the states of any reserved bits when storing them to memory or to a register.
- Do not depend on the ability to retain information written into any reserved bits.
- When loading a register, always load the reserved bits with the values indicated in the documentation, if any, or reload them with values previously read from the same register.

Software written for existing IA-32 processor that handles reserved bits correctly will port to future IA-32 processors without generating protection exceptions.

## 22.3 ENABLING NEW FUNCTIONS AND MODES

Most of the new control functions defined for the P6 family and Pentium processors are enabled by new mode flags in the control registers (primarily register CR4). This register is undefined for IA-32 processors earlier than the Pentium processor. Attempting to access this register with an Intel486 or earlier IA-32 processor results in an invalid-opcode exception (#UD). Consequently, programs that execute correctly on the Intel486 or earlier IA-32 processor cannot erroneously enable these functions. Attempting to set a reserved bit in register CR4 to a value other than its original value results in a general-protection exception (#GP). So, programs that execute on the P6 family and Pentium processors cannot erroneously enable functions that may be implemented in future IA-32 processors.

The P6 family and Pentium processors do not check for attempts to set reserved bits in model-specific registers; however these bits may be checked on more recent processors. It is the obligation of the software writer to enforce this discipline. These reserved bits may be used in future Intel processors.

## 22.4 DETECTING THE PRESENCE OF NEW FEATURES THROUGH SOFTWARE

Software can check for the presence of new architectural features and extensions in either of two ways:

1. Test for the presence of the feature or extension. Software can test for the presence of new flags in the EFLAGS register and control registers. If these flags are reserved (meaning not present in the processor executing the test), an exception is generated. Likewise, software can attempt to execute a new instruction, which results in an invalid-opcode exception (#UD) being generated if it is not supported.
2. Execute the CPUID instruction. The CPUID instruction (added to the IA-32 in the Pentium processor) indicates the presence of new features directly.

See Chapter 19, "Processor Identification and Feature Determination," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for detailed information on detecting new processor features and extensions.

## 22.5 INTEL MMX TECHNOLOGY

The Pentium processor with MMX technology introduced the MMX technology and a set of MMX instructions to the IA-32. The MMX instructions are described in Chapter 9, "Programming with Intel® MMX™ Technology," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, and in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volumes 2A, 2B, 2C & 2D*. The MMX technology and MMX instructions are also included in the Pentium II, Pentium III, Pentium 4, and Intel Xeon processors.



## 22.6 STREAMING SIMD EXTENSIONS (SSE)

The Streaming SIMD Extensions (SSE) were introduced in the Pentium III processor. The SSE extensions consist of a new set of instructions and a new set of registers. The new registers include the eight 128-bit XMM registers and the 32-bit MXCSR control and status register. These instructions and registers are designed to allow SIMD computations to be made on single-precision floating-point numbers. Several of these new instructions also operate in the MMX registers. SSE instructions and registers are described in Section 10, "Programming with Streaming SIMD Extensions (SSE)," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, and in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volumes 2A, 2B, 2C & 2D*.

## 22.7 STREAMING SIMD EXTENSIONS 2 (SSE2)

The Streaming SIMD Extensions 2 (SSE2) were introduced in the Pentium 4 and Intel Xeon processors. They consist of a new set of instructions that operate on the XMM and MXCSR registers and perform SIMD operations on double-precision floating-point values and on integer values. Several of these new instructions also operate in the MMX registers. SSE2 instructions and registers are described in Chapter 11, "Programming with Streaming SIMD Extensions 2 (SSE2)," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, and in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volumes 2A, 2B, 2C & 2D*.

## 22.8 STREAMING SIMD EXTENSIONS 3 (SSE3)

The Streaming SIMD Extensions 3 (SSE3) were introduced in Pentium 4 processors supporting Intel Hyper-Threading Technology and Intel Xeon processors. SSE3 extensions include 13 instructions. Ten of these 13 instructions support the single instruction multiple data (SIMD) execution model used with SSE/SSE2 extensions. One SSE3 instruction accelerates x87 style programming for conversion to integer. The remaining two instructions (MONITOR and MWAIT) accelerate synchronization of threads. SSE3 instructions are described in Chapter 12, "Programming with SSE3, SSSE3 and SSE4," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, and in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volumes 2A, 2B, 2C & 2D*.

## 22.9 ADDITIONAL STREAMING SIMD EXTENSIONS

The Supplemental Streaming SIMD Extensions 3 (SSSE3) were introduced in the Intel Core 2 processor and Intel Xeon processor 5100 series. Streaming SIMD Extensions 4 provided 54 new instructions introduced in 45 nm Intel Xeon processors and Intel Core 2 processors. SSSE3, SSE4.1 and SSE4.2 instructions are described in Chapter 12, "Programming with SSE3, SSSE3 and SSE4," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, and in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volumes 2A, 2B, 2C & 2D*.

## 22.10 INTEL HYPER-THREADING TECHNOLOGY

Intel Hyper-Threading Technology provides two logical processors that can execute two separate code streams (called *threads*) concurrently by using shared resources in a single processor core or in a physical package.

This feature was introduced in the Intel Xeon processor MP and later steppings of the Intel Xeon processor, and Pentium 4 processors supporting Intel Hyper-Threading Technology. The feature is also found in the Pentium processor Extreme Edition. See also: Section 8.7, "Intel® Hyper-Threading Technology Architecture."

45 nm and 32 nm Intel Atom processors support Intel Hyper-Threading Technology.

Intel Atom processors based on Silvermont and Airmont microarchitectures do not support Intel Hyper-Threading Technology

## 22.11 MULTI-CORE TECHNOLOGY

The Pentium D processor and Pentium processor Extreme Edition provide two processor cores in each physical processor package. See also: Section 8.5, “Intel® Hyper-Threading Technology and Intel® Multi-Core Technology,” and Section 8.8, “Multi-Core Architecture.” Intel Core 2 Duo, Intel Pentium Dual-Core processors, Intel Xeon processors 3000, 3100, 5100, 5200 series provide two processor cores in each physical processor package. Intel Core 2 Extreme, Intel Core 2 Quad processors, Intel Xeon processors 3200, 3300, 5300, 5400, 7300 series provide two processor cores in each physical processor package.

## 22.12 SPECIFIC FEATURES OF DUAL-CORE PROCESSOR

Dual-core processors may have some processor-specific features. Use CPUID feature flags to detect the availability features. Note the following:

- **CPUID Brand String** — On Pentium processor Extreme Edition, the process will report the correct brand string only after the correct microcode updates are loaded.
- **Enhanced Intel SpeedStep Technology** — This feature is supported in Pentium D processor but not in Pentium processor Extreme Edition.

## 22.13 NEW INSTRUCTIONS IN THE PENTIUM AND LATER IA-32 PROCESSORS

Table 22-1 identifies the instructions introduced into the IA-32 in the Pentium processor and later IA-32 processors.

### 22.13.1 Instructions Added Prior to the Pentium Processor

The following instructions were added in the Intel486 processor:

- BSWAP (byte swap) instruction.
- XADD (exchange and add) instruction.
- CMPXCHG (compare and exchange) instruction.
- INVD (invalidate cache) instruction.
- WBINVD (write-back and invalidate cache) instruction.
- INVLPG (invalidate TLB entry) instruction.

**Table 22-1. New Instruction in the Pentium Processor and Later IA-32 Processors**

Instruction	CPUID Identification Bits	Introduced In
CMOVcc (conditional move)	EDX, Bit 15	Pentium Pro processor
FCMOVcc (floating-point conditional move)	EDX, Bits 0 and 15	
FCOMI (floating-point compare and set EFLAGS)	EDX, Bits 0 and 15	
RDPMC (read performance monitoring counters)	EAX, Bits 8-11, set to 6H; see Note 1	
UD2 (undefined)	EAX, Bits 8-11, set to 6H	
CMPXCHG8B (compare and exchange 8 bytes)	EDX, Bit 8	Pentium processor
CPUID (CPU identification)	None; see Note 2	
RDTSC (read time-stamp counter)	EDX, Bit 4	
RDMSR (read model-specific register)	EDX, Bit 5	
WRMSR (write model-specific register)	EDX, Bit 5	
MMX Instructions	EDX, Bit 23	

**Table 22-1. New Instruction in the Pentium Processor and Later IA-32 Processors (Contd.)**

Instruction	CUID Identification Bits	Introduced In
-------------	--------------------------	---------------

**NOTES:**

1. The RDPNC instruction was introduced in the P6 family of processors and added to later model Pentium processors. This instruction is model specific in nature and not architectural.
2. The CUID instruction is available in all Pentium and P6 family processors and in later models of the Intel486 processors. The ability to set and clear the ID flag (bit 21) in the EFLAGS register indicates the availability of the CUID instruction.

The following instructions were added in the Intel386 processor:

- LSS, LFS, and LGS (load SS, FS, and GS registers).
- Long-displacement conditional jumps.
- Single-bit instructions.
- Bit scan instructions.
- Double-shift instructions.
- Byte set on condition instruction.
- Move with sign/zero extension.
- Generalized multiply instruction.
- MOV to and from control registers.
- MOV to and from test registers (now obsolete).
- MOV to and from debug registers.
- RSM (resume from SMM). This instruction was introduced in the Intel386 SL and Intel486 SL processors.

The following instructions were added in the Intel 387 math coprocessor:

- FPREM1.
- FUCOM, FUCOMP, and FUCOMPP.

## 22.14 OBSOLETE INSTRUCTIONS

The MOV to and from test registers instructions were removed from the Pentium processor and future IA-32 processors. Execution of these instructions generates an invalid-opcode exception (#UD).

## 22.15 UNDEFINED OPCODES

All new instructions defined for IA-32 processors use binary encodings that were reserved on earlier-generation processors. Attempting to execute a reserved opcode always results in an invalid-opcode (#UD) exception being generated. Consequently, programs that execute correctly on earlier-generation processors cannot erroneously execute these instructions and thereby produce unexpected results when executed on later IA-32 processors.

## 22.16 NEW FLAGS IN THE EFLAGS REGISTER

The section titled "EFLAGS Register" in Chapter 3, "Basic Execution Environment," of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, shows the configuration of flags in the EFLAGS register for the P6 family processors. No new flags have been added to this register in the P6 family processors. The flags added to this register in the Pentium and Intel486 processors are described in the following sections.

The following flags were added to the EFLAGS register in the Pentium processor:

- VIF (virtual interrupt flag), bit 19.
- VIP (virtual interrupt pending), bit 20.

- ID (identification flag), bit 21.

The AC flag (bit 18) was added to the EFLAGS register in the Intel486 processor.

### 22.16.1 Using EFLAGS Flags to Distinguish Between 32-Bit IA-32 Processors

The following bits in the EFLAGS register that can be used to differentiate between the 32-bit IA-32 processors:

- Bit 18 (the AC flag) can be used to distinguish an Intel386 processor from the P6 family, Pentium, and Intel486 processors. Since it is not implemented on the Intel386 processor, it will always be clear.
- Bit 21 (the ID flag) indicates whether an application can execute the CPUID instruction. The ability to set and clear this bit indicates that the processor is a P6 family or Pentium processor. The CPUID instruction can then be used to determine which processor.
- Bits 19 (the VIF flag) and 20 (the VIP flag) will always be zero on processors that do not support virtual mode extensions, which includes all 32-bit processors prior to the Pentium processor.

See Chapter 19, "Processor Identification and Feature Determination," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for more information on identifying processors.

## 22.17 STACK OPERATIONS

This section identifies the differences in stack implementation between the various IA-32 processors.

### 22.17.1 PUSH SP

The P6 family, Pentium, Intel486, Intel386, and Intel 286 processors push a different value on the stack for a PUSH SP instruction than the 8086 processor. The 32-bit processors push the value of the SP register before it is decremented as part of the push operation; the 8086 processor pushes the value of the SP register after it is decremented. If the value pushed is important, replace PUSH SP instructions with the following three instructions:

```
PUSH BP
MOV BP, SP
XCHG BP, [BP]
```

This code functions as the 8086 processor PUSH SP instruction on the P6 family, Pentium, Intel486, Intel386, and Intel 286 processors.

### 22.17.2 EFLAGS Pushed on the Stack

The setting of the stored values of bits 12 through 15 (which includes the IOPL field and the NT flag) in the EFLAGS register by the PUSHF instruction, by interrupts, and by exceptions is different with the 32-bit IA-32 processors than with the 8086 and Intel 286 processors. The differences are as follows:

- 8086 processor—bits 12 through 15 are always set.
- Intel 286 processor—bits 12 through 15 are always cleared in real-address mode.
- 32-bit processors in real-address mode—bit 15 (reserved) is always cleared, and bits 12 through 14 have the last value loaded into them.

## 22.18 X87 FPU

This section addresses the issues that must be faced when porting floating-point software designed to run on earlier IA-32 processors and math coprocessors to a Pentium 4, Intel Xeon, P6 family, or Pentium processor with integrated x87 FPU. To software, a Pentium 4, Intel Xeon, or P6 family processor looks very much like a Pentium processor. Floating-point software which runs on a Pentium or Intel486 DX processor, or on an Intel486 SX

processor/Intel 487 SX math coprocessor system or an Intel386 processor/Intel 387 math coprocessor system, will run with at most minor modifications on a Pentium 4, Intel Xeon, or P6 family processor. To port code directly from an Intel 286 processor/Intel 287 math coprocessor system or an Intel 8086 processor/8087 math coprocessor system to a Pentium 4, Intel Xeon, P6 family, or Pentium processor, certain additional issues must be addressed.

In the following sections, the term “32-bit x87 FPUs” refers to the P6 family, Pentium, and Intel486 DX processors, and to the Intel 487 SX and Intel 387 math coprocessors; the term “16-bit IA-32 math coprocessors” refers to the Intel 287 and 8087 math coprocessors.

## 22.18.1 Control Register CR0 Flags

The ET, NE, and MP flags in control register CR0 control the interface between the integer unit of an IA-32 processor and either its internal x87 FPU or an external math coprocessor. The effect of these flags in the various IA-32 processors are described in the following paragraphs.

The ET (extension type) flag (bit 4 of the CR0 register) is used in the Intel386 processor to indicate whether the math coprocessor in the system is an Intel 287 math coprocessor (flag is clear) or an Intel 387 DX math coprocessor (flag is set). This bit is hardwired to 1 in the P6 family, Pentium, and Intel486 processors.

The NE (Numeric Exception) flag (bit 5 of the CR0 register) is used in the P6 family, Pentium, and Intel486 processors to determine whether unmasked floating-point exceptions are reported internally through interrupt vector 16 (flag is set) or externally through an external interrupt (flag is clear). On a hardware reset, the NE flag is initialized to 0, so software using the automatic internal error-reporting mechanism must set this flag to 1. This flag is nonexistent on the Intel386 processor.

As on the Intel 286 and Intel386 processors, the MP (monitor coprocessor) flag (bit 1 of register CR0) determines whether the WAIT/FWAIT instructions or waiting-type floating-point instructions trap when the context of the x87 FPU is different from that of the currently-executing task. If the MP and TS flag are set, then a WAIT/FWAIT instruction and waiting instructions will cause a device-not-available exception (interrupt vector 7). The MP flag is used on the Intel 286 and Intel386 processors to support the use of a WAIT/FWAIT instruction to wait on a device other than a math coprocessor. The device reports its status through the BUSY# pin. Since the P6 family, Pentium, and Intel486 processors do not have such a pin, the MP flag has no relevant use and should be set to 1 for normal operation.

## 22.18.2 x87 FPU Status Word

This section identifies differences to the x87 FPU status word for the different IA-32 processors and math coprocessors, the reason for the differences, and their impact on software.

### 22.18.2.1 Condition Code Flags (C0 through C3)

The following information pertains to differences in the use of the condition code flags (C0 through C3) located in bits 8, 9, 10, and 14 of the x87 FPU status word.

After execution of an FINIT instruction or a hardware reset on a 32-bit x87 FPU, the condition code flags are set to 0. The same operations on a 16-bit IA-32 math coprocessor leave these flags intact (they contain their prior value). This difference in operation has no impact on software and provides a consistent state after reset.

Transcendental instruction results in the core range of the P6 family and Pentium processors may differ from the Intel486 DX processor and Intel 487 SX math coprocessor by 2 to 3 units in the last place (ulps)—(see “Transcendental Instruction Accuracy” in Chapter 8, “Programming with the x87 FPU,” of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*). As a result, the value saved in the C1 flag may also differ.

After an incomplete FPREM/FPREM1 instruction, the C0, C1, and C3 flags are set to 0 on the 32-bit x87 FPUs. After the same operation on a 16-bit IA-32 math coprocessor, these flags are left intact.

On the 32-bit x87 FPUs, the C2 flag serves as an incomplete flag for the FTAN instruction. On the 16-bit IA-32 math coprocessors, the C2 flag is undefined for the FPTAN instruction. This difference has no impact on software, because Intel 287 or 8087 programs do not check C2 after an FPTAN instruction. The use of this flag on later processors allows fast checking of operand range.

### 22.18.2.2 Stack Fault Flag

When unmasked stack overflow or underflow occurs on a 32-bit x87 FPU, the IE flag (bit 0) and the SF flag (bit 6) of the x87 FPU status word are set to indicate a stack fault and condition code flag C1 is set or cleared to indicate overflow or underflow, respectively. When unmasked stack overflow or underflow occurs on a 16-bit IA-32 math coprocessor, only the IE flag is set. Bit 6 is reserved on these processors. The addition of the SF flag on a 32-bit x87 FPU has no impact on software. Existing exception handlers need not change, but may be upgraded to take advantage of the additional information.

### 22.18.3 x87 FPU Control Word

Only affine closure is supported for infinity control on a 32-bit x87 FPU. The infinity control flag (bit 12 of the x87 FPU control word) remains programmable on these processors, but has no effect. This change was made to conform to the IEEE Standard 754 for Binary Floating-Point Arithmetic. On a 16-bit IA-32 math coprocessor, both affine and projective closures are supported, as determined by the setting of bit 12. After a hardware reset, the default value of bit 12 is projective. Software that requires projective infinity arithmetic may give different results.

### 22.18.4 x87 FPU Tag Word

When loading the tag word of a 32-bit x87 FPU, using an FLDENV, FRSTOR, or FXRSTOR (Pentium III processor only) instruction, the processor examines the incoming tag and classifies the location only as empty or non-empty. Thus, tag values of 00, 01, and 10 are interpreted by the processor to indicate a non-empty location. The tag value of 11 is interpreted by the processor to indicate an empty location. Subsequent operations on a non-empty register always examine the value in the register, not the value in its tag. The FSTENV, FSAVE, and FXSAVE (Pentium III processor only) instructions examine the non-empty registers and put the correct values in the tags before storing the tag word.

The corresponding tag for a 16-bit IA-32 math coprocessor is checked before each register access to determine the class of operand in the register; the tag is updated after every change to a register so that the tag always reflects the most recent status of the register. Software can load a tag with a value that disagrees with the contents of a register (for example, the register contains a valid value, but the tag says special). Here, the 16-bit IA-32 math coprocessors honor the tag and do not examine the register.

Software written to run on a 16-bit IA-32 math coprocessor may not operate correctly on a 16-bit x87 FPU, if it uses the FLDENV, FRSTOR, or FXRSTOR instructions to change tags to values (other than to empty) that are different from actual register contents.

The encoding in the tag word for the 32-bit x87 FPUs for unsupported data formats (including pseudo-zero and unnormal) is special (10B), to comply with IEEE Standard 754. The encoding in the 16-bit IA-32 math coprocessors for pseudo-zero and unnormal is valid (00B) and the encoding for other unsupported data formats is special (10B). Code that recognizes the pseudo-zero or unnormal format as valid must therefore be changed if it is ported to a 32-bit x87 FPU.

### 22.18.5 Data Types

This section discusses the differences of data types for the various x87 FPUs and math coprocessors.

#### 22.18.5.1 NaNs

The 32-bit x87 FPUs distinguish between signaling NaNs (SNaNs) and quiet NaNs (QNaNs). These x87 FPUs only generate QNaNs and normally do not generate an exception upon encountering a QNaN. An invalid-operation exception (#I) is generated only upon encountering a SNaN, except for the FCOM, FIST, and FBSTP instructions, which also generates an invalid-operation exceptions for a QNaNs. This behavior matches IEEE Standard 754.

The 16-bit IA-32 math coprocessors only generate one kind of NaN (the equivalent of a QNaN), but the raise an invalid-operation exception upon encountering any kind of NaN.

When porting software written to run on a 16-bit IA-32 math coprocessor to a 32-bit x87 FPU, uninitialized memory locations that contain QNaNs should be changed to SNaNs to cause the x87 FPU or math coprocessor to fault when uninitialized memory locations are referenced.

### 22.18.5.2 Pseudo-zero, Pseudo-NaN, Pseudo-infinity, and Unnormal Formats

The 32-bit x87 FPUs neither generate nor support the pseudo-zero, pseudo-NaN, pseudo-infinity, and unnormal formats. Whenever they encounter them in an arithmetic operation, they raise an invalid-operation exception. The 16-bit IA-32 math coprocessors define and support special handling for these formats. Support for these formats was dropped to conform with IEEE Standard 754 for Binary Floating-Point Arithmetic.

This change should not impact software ported from 16-bit IA-32 math coprocessors to 32-bit x87 FPUs. The 32-bit x87 FPUs do not generate these formats, and therefore will not encounter them unless software explicitly loads them in the data registers. The only affect may be in how software handles the tags in the tag word (see also: Section 22.18.4, "x87 FPU Tag Word").

## 22.18.6 Floating-Point Exceptions

This section identifies the implementation differences in exception handling for floating-point instructions in the various x87 FPUs and math coprocessors.

### 22.18.6.1 Denormal Operand Exception (#D)

When the denormal operand exception is masked, the 32-bit x87 FPUs automatically normalize denormalized numbers when possible; whereas, the 16-bit IA-32 math coprocessors return a denormal result. A program written to run on a 16-bit IA-32 math coprocessor that uses the denormal exception solely to normalize denormalized operands is redundant when run on the 32-bit x87 FPUs. If such a program is run on 32-bit x87 FPUs, performance can be improved by masking the denormal exception. Floating-point programs run faster when the FPU performs normalization of denormalized operands.

The denormal operand exception is not raised for transcendental instructions and the FEXTRACT instruction on the 16-bit IA-32 math coprocessors. This exception is raised for these instructions on the 32-bit x87 FPUs. The exception handlers ported to these latter processors need to be changed only if the handlers gives special treatment to different opcodes.

### 22.18.6.2 Numeric Overflow Exception (#O)

On the 32-bit x87 FPUs, when the numeric overflow exception is masked and the rounding mode is set to chop (toward 0), the result is the largest positive or smallest negative number. The 16-bit IA-32 math coprocessors do not signal the overflow exception when the masked response is not  $\infty$ ; that is, they signal overflow only when the rounding control is not set to round to 0. If rounding is set to chop (toward 0), the result is positive or negative  $\infty$ . Under the most common rounding modes, this difference has no impact on existing software.

If rounding is toward 0 (chop), a program on a 32-bit x87 FPU produces, under overflow conditions, a result that is different in the least significant bit of the significand, compared to the result on a 16-bit IA-32 math coprocessor. The reason for this difference is IEEE Standard 754 compatibility.

When the overflow exception is not masked, the precision exception is flagged on the 32-bit x87 FPUs. When the result is stored in the stack, the significand is rounded according to the precision control (PC) field of the FPU control word or according to the opcode. On the 16-bit IA-32 math coprocessors, the precision exception is not flagged and the significand is not rounded. The impact on existing software is that if the result is stored on the stack, a program running on a 32-bit x87 FPU produces a different result under overflow conditions than on a 16-bit IA-32 math coprocessor. The difference is apparent only to the exception handler. This difference is for IEEE Standard 754 compatibility.



### 22.18.6.3 Numeric Underflow Exception (#U)

When the underflow exception is masked on the 32-bit x87 FPUs, the underflow exception is signaled when the result is tiny and inexact (see Section 4.9.1.5, "Numeric Underflow Exception (#U)" in *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*). When the underflow exception is unmasked and the instruction is supposed to store the result on the stack, the significand is rounded to the appropriate precision (according to the PC flag in the FPU control word, for those instructions controlled by PC, otherwise to extended precision), after adjusting the exponent.

### 22.18.6.4 Exception Precedence

There is no difference in the precedence of the denormal-operand exception on the 32-bit x87 FPUs, whether it be masked or not. When the denormal-operand exception is not masked on the 16-bit IA-32 math coprocessors, it takes precedence over all other exceptions. This difference causes no impact on existing software, but some unneeded normalization of denormalized operands is prevented on the Intel486 processor and Intel 387 math coprocessor.

### 22.18.6.5 CS and EIP For FPU Exceptions

On the Intel 32-bit x87 FPUs, the values from the CS and EIP registers saved for floating-point exceptions point to any prefixes that come before the floating-point instruction. On the 8087 math coprocessor, the saved CS and IP registers points to the floating-point instruction.

### 22.18.6.6 FPU Error Signals

The floating-point error signals to the P6 family, Pentium, and Intel486 processors do not pass through an interrupt controller; an INT# signal from an Intel 387, Intel 287 or 8087 math coprocessors does. If an 8086 processor uses another exception for the 8087 interrupt, both exception vectors should call the floating-point-error exception handler. Some instructions in a floating-point-error exception handler may need to be deleted if they use the interrupt controller. The P6 family, Pentium, and Intel486 processors have signals that, with the addition of external logic, support reporting for emulation of the interrupt mechanism used in many personal computers.

On the P6 family, Pentium, and Intel486 processors, an undefined floating-point opcode will cause an invalid-opcode exception (#UD, interrupt vector 6). Undefined floating-point opcodes, like legal floating-point opcodes, cause a device not available exception (#NM, interrupt vector 7) when either the TS or EM flag in control register CR0 is set. The P6 family, Pentium, and Intel486 processors do not check for floating-point error conditions on encountering an undefined floating-point opcode.

### 22.18.6.7 Assertion of the FERR# Pin

When using the MS-DOS compatibility mode for handing floating-point exceptions, the FERR# pin must be connected to an input to an external interrupt controller. An external interrupt is then generated when the FERR# output drives the input to the interrupt controller and the interrupt controller in turn drives the INTR pin on the processor.

For the P6 family and Intel386 processors, an unmasked floating-point exception always causes the FERR# pin to be asserted upon completion of the instruction that caused the exception. For the Pentium and Intel486 processors, an unmasked floating-point exception may cause the FERR# pin to be asserted either at the end of the instruction causing the exception or immediately before execution of the next floating-point instruction. (Note that the next floating-point instruction would not be executed until the pending unmasked exception has been handled.) See Appendix D, "Guidelines for Writing x87 FPU Extension Handlers," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for a complete description of the required mechanism for handling floating-point exceptions using the MS-DOS compatibility mode.

Using FERR# and IGNNE# to handle floating-point exception is deprecated by modern operating systems; this approach also limits newer processors to operate with one logical processor active.



### 22.18.6.8 Invalid Operation Exception On Denormals

An invalid-operation exception is not generated on the 32-bit x87 FPUs upon encountering a denormal value when executing a FSQRT, FDIV, or FPREM instruction or upon conversion to BCD or to integer. The operation proceeds by first normalizing the value. On the 16-bit IA-32 math coprocessors, upon encountering this situation, the invalid-operation exception is generated. This difference has no impact on existing software. Software running on the 32-bit x87 FPUs continues to execute in cases where the 16-bit IA-32 math coprocessors trap. The reason for this change was to eliminate an exception from being raised.

### 22.18.6.9 Alignment Check Exceptions (#AC)

If alignment checking is enabled, a misaligned data operand on the P6 family, Pentium, and Intel486 processors causes an alignment check exception (#AC) when a program or procedure is running at privilege-level 3, except for the stack portion of the FSAVE/FNSAVE, FXSAVE, FRSTOR, and FXRSTOR instructions.

### 22.18.6.10 Segment Not Present Exception During FLDENV

On the Intel486 processor, when a segment not present exception (#NP) occurs in the middle of an FLDENV instruction, it can happen that part of the environment is loaded and part not. In such cases, the FPU control word is left with a value of 007FH. The P6 family and Pentium processors ensure the internal state is correct at all times by attempting to read the first and last bytes of the environment before updating the internal state.

### 22.18.6.11 Device Not Available Exception (#NM)

The device-not-available exception (#NM, interrupt 7) will occur in the P6 family, Pentium, and Intel486 processors as described in Section 2.5, "Control Registers," Table 2-2, and Chapter 6, "Interrupt 7—Device Not Available Exception (#NM)."

### 22.18.6.12 Coprocessor Segment Overrun Exception

The coprocessor segment overrun exception (interrupt 9) does not occur in the P6 family, Pentium, and Intel486 processors. In situations where the Intel 387 math coprocessor would cause an interrupt 9, the P6 family, Pentium, and Intel486 processors simply abort the instruction. To avoid undetected segment overruns, it is recommended that the floating-point save area be placed in the same page as the TSS. This placement will prevent the FPU environment from being lost if a page fault occurs during the execution of an FLDENV, FRSTOR, or FXRSTOR instruction while the operating system is performing a task switch.

### 22.18.6.13 General Protection Exception (#GP)

A general-protection exception (#GP, interrupt 13) occurs if the starting address of a floating-point operand falls outside a segment's size. An exception handler should be included to report these programming errors.

### 22.18.6.14 Floating-Point Error Exception (#MF)

In real mode and protected mode (not including virtual-8086 mode), interrupt vector 16 must point to the floating-point exception handler. In virtual-8086 mode, the virtual-8086 monitor can be programmed to accommodate a different location of the interrupt vector for floating-point exceptions.

## 22.18.7 Changes to Floating-Point Instructions

This section identifies the differences in floating-point instructions for the various Intel FPU and math coprocessor architectures, the reason for the differences, and their impact on software.

### 22.18.7.1 FDIV, FPREM, and FSQRT Instructions

The 32-bit x87 FPUs support operations on denormalized operands and, when detected, an underflow exception can occur, for compatibility with the IEEE Standard 754. The 16-bit IA-32 math coprocessors do not operate on denormalized operands or return underflow results. Instead, they generate an invalid-operation exception when they detect an underflow condition. An existing underflow exception handler will require change only if it gives different treatment to different opcodes. Also, it is possible that fewer invalid-operation exceptions will occur.

### 22.18.7.2 FSCALE Instruction

With the 32-bit x87 FPUs, the range of the scaling operand is not restricted. If  $(0 < |ST(1)| < 1)$ , the scaling factor is 0; therefore,  $ST(0)$  remains unchanged. If the rounded result is not exact or if there was a loss of accuracy (masked underflow), the precision exception is signaled. With the 16-bit IA-32 math coprocessors, the range of the scaling operand is restricted. If  $(0 < |ST(1)| < 1)$ , the result is undefined and no exception is signaled. The impact of this difference on exiting software is that different results are delivered on the 32-bit and 16-bit FPUs and math coprocessors when  $(0 < |ST(1)| < 1)$ .

### 22.18.7.3 FPREM1 Instruction

The 32-bit x87 FPUs compute a partial remainder according to IEEE Standard 754. This instruction does not exist on the 16-bit IA-32 math coprocessors. The availability of the FPREM1 instruction has no impact on existing software.

### 22.18.7.4 FPREM Instruction

On the 32-bit x87 FPUs, the condition code flags C0, C3, C1 in the status word correctly reflect the three low-order bits of the quotient following execution of the FPREM instruction. On the 16-bit IA-32 math coprocessors, the quotient bits are incorrect when performing a reduction of  $(64^N + M)$  when  $(N \geq 1)$  and  $M$  is 1 or 2. This difference does not affect existing software; software that works around the bug should not be affected.

### 22.18.7.5 FUCOM, FUCOMP, and FUCOMPP Instructions

When executing the FUCOM, FUCOMP, and FUCOMPP instructions, the 32-bit x87 FPUs perform unordered compare according to IEEE Standard 754. These instructions do not exist on the 16-bit IA-32 math coprocessors. The availability of these new instructions has no impact on existing software.

### 22.18.7.6 FPTAN Instruction

On the 32-bit x87 FPUs, the range of the operand for the FPTAN instruction is much less restricted ( $|ST(0)| < 2^{63}$ ) than on earlier math coprocessors. The instruction reduces the operand internally using an internal  $\pi/4$  constant that is more accurate. The range of the operand is restricted to  $(|ST(0)| < \pi/4)$  on the 16-bit IA-32 math coprocessors; the operand must be reduced to this range using FPREM. This change has no impact on existing software. See also sections 8.3.8 and section 8.3.10 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1* for more information on the accuracy of the FPTAN instruction.

### 22.18.7.7 Stack Overflow

On the 32-bit x87 FPUs, if an FPU stack overflow occurs when the invalid-operation exception is masked, the FPU returns the real, integer, or BCD-integer indefinite value to the destination operand, depending on the instruction being executed. On the 16-bit IA-32 math coprocessors, the original operand remains unchanged following a stack overflow, but it is loaded into register  $ST(1)$ . This difference has no impact on existing software.

### 22.18.7.8 FSIN, FCOS, and FSINCOS Instructions

On the 32-bit x87 FPUs, these instructions perform three common trigonometric functions. These instructions do not exist on the 16-bit IA-32 math coprocessors. The availability of these instructions has no impact on existing

software, but using them provides a performance upgrade. See also sections 8.3.8 and section 8.3.10 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1* for more information on the accuracy of the FSIN, FCOS, and FSINCOS instructions.

### 22.18.7.9 FPATAN Instruction

On the 32-bit x87 FPUs, the range of operands for the FPATAN instruction is unrestricted. On the 16-bit IA-32 math coprocessors, the absolute value of the operand in register ST(0) must be smaller than the absolute value of the operand in register ST(1). This difference has impact on existing software.

### 22.18.7.10 F2XM1 Instruction

The 32-bit x87 FPUs support a wider range of operands ( $-1 < ST(0) < +1$ ) for the F2XM1 instruction. The supported operand range for the 16-bit IA-32 math coprocessors is ( $0 \leq ST(0) \leq 0.5$ ). This difference has no impact on existing software.

### 22.18.7.11 FLD Instruction

On the 32-bit x87 FPUs, when using the FLD instruction to load an extended-real value, a denormal-operand exception is not generated because the instruction is not arithmetic. The 16-bit IA-32 math coprocessors do report a denormal-operand exception in this situation. This difference does not affect existing software.

On the 32-bit x87 FPUs, loading a denormal value that is in single- or double-real format causes the value to be converted to extended-real format. Loading a denormal value on the 16-bit IA-32 math coprocessors causes the value to be converted to an unnormal. If the next instruction is FXTRACT or FXAM, the 32-bit x87 FPUs will give a different result than the 16-bit IA-32 math coprocessors. This change was made for IEEE Standard 754 compatibility.

On the 32-bit x87 FPUs, loading an SNaN that is in single- or double-real format causes the FPU to generate an invalid-operation exception. The 16-bit IA-32 math coprocessors do not raise an exception when loading a signaling NaN. The invalid-operation exception handler for 16-bit math coprocessor software needs to be updated to handle this condition when porting software to 32-bit FPUs. This change was made for IEEE Standard 754 compatibility.

### 22.18.7.12 FXTRACT Instruction

On the 32-bit x87 FPUs, if the operand is 0 for the FXTRACT instruction, the divide-by-zero exception is reported and  $-\infty$  is delivered to register ST(1). If the operand is  $+\infty$ , no exception is reported. If the operand is 0 on the 16-bit IA-32 math coprocessors, 0 is delivered to register ST(1) and no exception is reported. If the operand is  $+\infty$ , the invalid-operation exception is reported. These differences have no impact on existing software. Software usually bypasses 0 and  $\infty$ . This change is due to the IEEE Standard 754 recommendation to fully support the "logb" function.

### 22.18.7.13 Load Constant Instructions

On 32-bit x87 FPUs, rounding control is in effect for the load constant instructions. Rounding control is not in effect for the 16-bit IA-32 math coprocessors. Results for the FLDPI, FLDLN2, FLDLG2, and FLDL2E instructions are the same as for the 16-bit IA-32 math coprocessors when rounding control is set to round to nearest or round to  $+\infty$ . They are the same for the FLDL2T instruction when rounding control is set to round to nearest, round to  $-\infty$ , or round to zero. Results are different from the 16-bit IA-32 math coprocessors in the least significant bit of the mantissa if rounding control is set to round to  $-\infty$  or round to 0 for the FLDPI, FLDLN2, FLDLG2, and FLDL2E instructions; they are different for the FLDL2T instruction if round to  $+\infty$  is specified. These changes were implemented for compatibility with IEEE Standard 754 for Floating-Point Arithmetic recommendations.

### 22.18.7.14 FSETPM Instruction

With the 32-bit x87 FPUs, the FSETPM instruction is treated as NOP (no operation). This instruction informs the Intel 287 math coprocessor that the processor is in protected mode. This change has no impact on existing software. The 32-bit x87 FPUs handle all addressing and exception-pointer information, whether in protected mode or not.

### 22.18.7.15 FXAM Instruction

With the 32-bit x87 FPUs, if the FPU encounters an empty register when executing the FXAM instruction, it not generate combinations of C0 through C3 equal to 1101 or 1111. The 16-bit IA-32 math coprocessors may generate these combinations, among others. This difference has no impact on existing software; it provides a performance upgrade to provide repeatable results.

### 22.18.7.16 FSAVE and FSTENV Instructions

With the 32-bit x87 FPUs, the address of a memory operand pointer stored by FSAVE or FSTENV is undefined if the previous floating-point instruction did not refer to memory

## 22.18.8 Transcendental Instructions

The floating-point results of the P6 family and Pentium processors for transcendental instructions in the core range may differ from the Intel486 processors by about 2 or 3 ulps (see "Transcendental Instruction Accuracy" in Chapter 8, "Programming with the x87 FPU," of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*). Condition code flag C1 of the status word may differ as a result. The exact threshold for underflow and overflow will vary by a few ulps. The P6 family and Pentium processors' results will have a worst case error of less than 1 ulp when rounding to the nearest-even and less than 1.5 ulps when rounding in other modes. The transcendental instructions are guaranteed to be monotonic, with respect to the input operands, throughout the domain supported by the instruction.

Transcendental instructions may generate different results in the round-up flag (C1) on the 32-bit x87 FPUs. The round-up flag is undefined for these instructions on the 16-bit IA-32 math coprocessors. This difference has no impact on existing software.

## 22.18.9 Obsolete Instructions

The 8087 math coprocessor instructions FENI and FDISI and the Intel 287 math coprocessor instruction FSETPM are treated as integer NOP instructions in the 32-bit x87 FPUs. If these opcodes are detected in the instruction stream, no specific operation is performed and no internal states are affected.

## 22.18.10 WAIT/FWAIT Prefix Differences

On the Intel486 processor, when a WAIT/FWAIT instruction precedes a floating-point instruction (one which itself automatically synchronizes with the previous floating-point instruction), the WAIT/FWAIT instruction is treated as a no-op. Pending floating-point exceptions from a previous floating-point instruction are processed not on the WAIT/FWAIT instruction but on the floating-point instruction following the WAIT/FWAIT instruction. In such a case, the report of a floating-point exception may appear one instruction later on the Intel486 processor than on a P6 family or Pentium FPU, or on Intel 387 math coprocessor.

## 22.18.11 Operands Split Across Segments and/or Pages

On the P6 family, Pentium, and Intel486 processor FPUs, when the first half of an operand to be written is inside a page or segment and the second half is outside, a memory fault can cause the first half to be stored but not the second half. In this situation, the Intel 387 math coprocessor stores nothing.

### 22.18.12 FPU Instruction Synchronization

On the 32-bit x87 FPUs, all floating-point instructions are automatically synchronized; that is, the processor automatically waits until the previous floating-point instruction has completed before completing the next floating-point instruction. No explicit WAIT/FWAIT instructions are required to assure this synchronization. For the 8087 math coprocessors, explicit waits are required before each floating-point instruction to ensure synchronization. Although 8087 programs having explicit WAIT instructions execute perfectly on the 32-bit IA-32 processors without reassembly, these WAIT instructions are unnecessary.

## 22.19 SERIALIZING INSTRUCTIONS

Certain instructions have been defined to serialize instruction execution to ensure that modifications to flags, registers and memory are completed before the next instruction is executed (or in P6 family processor terminology “committed to machine state”). Because the P6 family processors use branch-prediction and out-of-order execution techniques to improve performance, instruction execution is not generally serialized until the results of an executed instruction are committed to machine state (see Chapter 2, “Intel® 64 and IA-32 Architectures,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*).

As a result, at places in a program or task where it is critical to have execution completed for all previous instructions before executing the next instruction (for example, at a branch, at the end of a procedure, or in multiprocessor dependent code), it is useful to add a serializing instruction. See Section 8.3, “Serializing Instructions,” for more information on serializing instructions.

## 22.20 FPU AND MATH COPROCESSOR INITIALIZATION

Table 9-1 shows the states of the FPUs in the P6 family, Pentium, Intel486 processors and of the Intel 387 math coprocessor and Intel 287 coprocessor following a power-up, reset, or INIT, or following the execution of an FINIT/FNINIT instruction. The following is some additional compatibility information concerning the initialization of x87 FPUs and math coprocessors.

### 22.20.1 Intel® 387 and Intel® 287 Math Coprocessor Initialization

Following an Intel386 processor reset, the processor identifies its coprocessor type (Intel® 287 or Intel® 387 DX math coprocessor) by sampling its ERROR# input some time after the falling edge of RESET# signal and before execution of the first floating-point instruction. The Intel 287 coprocessor keeps its ERROR# output in inactive state after hardware reset; the Intel 387 coprocessor keeps its ERROR# output in active state after hardware reset.

Upon hardware reset or execution of the FINIT/FNINIT instruction, the Intel 387 math coprocessor signals an error condition. The P6 family, Pentium, and Intel486 processors, like the Intel 287 coprocessor, do not.

### 22.20.2 Intel486 SX Processor and Intel 487 SX Math Coprocessor Initialization

When initializing an Intel486 SX processor and an Intel 487 SX math coprocessor, the initialization routine should check the presence of the math coprocessor and should set the FPU related flags (EM, MP, and NE) in control register CR0 accordingly (see Section 2.5, “Control Registers,” for a complete description of these flags). Table 22-2 gives the recommended settings for these flags when the math coprocessor is present. The FSTCW instruction will give a value of FFFFH for the Intel486 SX microprocessor and 037FH for the Intel 487 SX math coprocessor.

**Table 22-2. Recommended Values of the EM, MP, and NE Flags for Intel486 SX Microprocessor/Intel 487 SX Math Coprocessor System**

CRO Flags	Intel486 SX Processor Only	Intel 487 SX Math Coprocessor Present
EM	1	0
MP	0	1
NE	1	0, for MS-DOS* systems 1, for user-defined exception handler

The EM and MP flags in register CR0 are interpreted as shown in Table 22-3.

**Table 22-3. EM and MP Flag Interpretation**

EM	MP	Interpretation
0	0	Floating-point instructions are passed to FPU; WAIT/FWAIT and other waiting-type instructions ignore TS.
0	1	Floating-point instructions are passed to FPU; WAIT/FWAIT and other waiting-type instructions test TS.
1	0	Floating-point instructions trap to emulator; WAIT/FWAIT and other waiting-type instructions ignore TS.
1	1	Floating-point instructions trap to emulator; WAIT/FWAIT and other waiting-type instructions test TS.

Following is an example code sequence to initialize the system and check for the presence of Intel486 SX processor/Intel 487 SX math coprocessor.

```
fninit
fstcw mem_loc
mov ax, mem_loc
cmp ax, 037fh
jz Intel487_SX_Math_CoProcessor_present ;ax=037fh
jmp Intel486_SX_microprocessor_present ;ax=ffffh
```

If the Intel 487 SX math coprocessor is not present, the following code can be run to set the CR0 register for the Intel486 SX processor.

```
mov eax, cr0
and eax, ffffffffh ;make MP=0
or eax, 0024h ;make EM=1, NE=1
mov cr0, eax
```

This initialization will cause any floating-point instruction to generate a device not available exception (#NH), interrupt 7. The software emulation will then take control to execute these instructions. This code is not required if an Intel 487 SX math coprocessor is present in the system. In that case, the typical initialization routine for the Intel486 SX microprocessor will be adequate.

Also, when designing an Intel486 SX processor based system with an Intel 487 SX math coprocessor, timing loops should be independent of frequency and clocks per instruction. One way to attain this is to implement these loops in hardware and not in software (for example, BIOS).

## 22.21 CONTROL REGISTERS

The following sections identify the new control registers and control register flags and fields that were introduced to the 32-bit IA-32 in various processor families. See Figure 2-7 for the location of these flags and fields in the control registers.

The Pentium III processor introduced one new control flag in control register CR4:

- OSXMMEXCPT (bit 10) — The OS will set this bit if it supports unmasked SIMD floating-point exceptions.

The Pentium II processor introduced one new control flag in control register CR4:

- OSFXSR (bit 9) — The OS supports saving and restoring the Pentium III processor state during context switches.

The Pentium Pro processor introduced three new control flags in control register CR4:

- PAE (bit 5) — Physical address extension. Enables paging mechanism to reference extended physical addresses when set; restricts physical addresses to 32 bits when clear (see also: Section 22.22.1.1, “Physical Memory Addressing Extension”).
- PGE (bit 7) — Page global enable. Inhibits flushing of frequently-used or shared pages on CR3 writes (see also: Section 22.22.1.2, “Global Pages”).
- PCE (bit 8) — Performance-monitoring counter enable. Enables execution of the RDPMC instruction at any protection level.

The content of CR4 is 0H following a hardware reset.

Control register CR4 was introduced in the Pentium processor. This register contains flags that enable certain new extensions provided in the Pentium processor:

- VME — Virtual-8086 mode extensions. Enables support for a virtual interrupt flag in virtual-8086 mode (see Section 20.3, “Interrupt and Exception Handling in Virtual-8086 Mode”).
- PVI — Protected-mode virtual interrupts. Enables support for a virtual interrupt flag in protected mode (see Section 20.4, “Protected-Mode Virtual Interrupts”).
- TSD — Time-stamp disable. Restricts the execution of the RDTSC instruction to procedures running at privileged level 0.
- DE — Debugging extensions. Causes an undefined opcode (#UD) exception to be generated when debug registers DR4 and DR5 are references for improved performance (see Section 22.23.3, “Debug Registers DR4 and DR5”).
- PSE — Page size extensions. Enables 4-MByte pages with 32-bit paging when set (see Section 4.3, “32-Bit Paging”).
- MCE — Machine-check enable. Enables the machine-check exception, allowing exception handling for certain hardware error conditions (see Chapter 15, “Machine-Check Architecture”).

The Intel486 processor introduced five new flags in control register CR0:

- NE — Numeric error. Enables the normal mechanism for reporting floating-point numeric errors.
- WP — Write protect. Write-protects read-only pages against supervisor-mode accesses.
- AM — Alignment mask. Controls whether alignment checking is performed. Operates in conjunction with the AC (Alignment Check) flag.
- NW — Not write-through. Enables write-throughs and cache invalidation cycles when clear and disables invalidation cycles and write-throughs that hit in the cache when set.
- CD — Cache disable. Enables the internal cache when clear and disables the cache when set.

The Intel486 processor introduced two new flags in control register CR3:

- PCD — Page-level cache disable. The state of this flag is driven on the PCD# pin during bus cycles that are not paged, such as interrupt acknowledge cycles, when paging is enabled. The PCD# pin is used to control caching in an external cache on a cycle-by-cycle basis.
- PWT — Page-level write-through. The state of this flag is driven on the PWT# pin during bus cycles that are not paged, such as interrupt acknowledge cycles, when paging is enabled. The PWT# pin is used to control write through in an external cache on a cycle-by-cycle basis.



## 22.22 MEMORY MANAGEMENT FACILITIES

The following sections describe the new memory management facilities available in the various IA-32 processors and some compatibility differences.

### 22.22.1 New Memory Management Control Flags

The Pentium Pro processor introduced three new memory management features: physical memory addressing extension, the global bit in page-table entries, and general support for larger page sizes. These features are only available when operating in protected mode.

#### 22.22.1.1 Physical Memory Addressing Extension

The new PAE (physical address extension) flag in control register CR4, bit 5, may enable additional address lines on the processor, allowing extended physical addresses. This option can only be used when paging is enabled, using a new page-table mechanism provided to support the larger physical address range (see Section 4.1, "Paging Modes and Control Bits").

#### 22.22.1.2 Global Pages

The new PGE (page global enable) flag in control register CR4, bit 7, provides a mechanism for preventing frequently used pages from being flushed from the translation lookaside buffer (TLB). When this flag is set, frequently used pages (such as pages containing kernel procedures or common data tables) can be marked global by setting the global flag in a page-directory or page-table entry.

On a task switch or a write to control register CR3 (which normally causes the TLBs to be flushed), the entries in the TLB marked global are not flushed. Marking pages global in this manner prevents unnecessary reloading of the TLB due to TLB misses on frequently used pages. See Section 4.10, "Caching Translation Information" for a detailed description of this mechanism.

#### 22.22.1.3 Larger Page Sizes

The P6 family processors support large page sizes. For 32-bit paging, this facility is enabled with the PSE (page size extension) flag in control register CR4, bit 4. When this flag is set, the processor supports either 4-KByte or 4-MByte page sizes. PAE paging and IA-32e paging support 2-MByte pages regardless of the value of CR4.PSE (see Section 4.4, "PAE Paging" and Section 4.5, "IA-32e Paging"). See Chapter 4, "Paging," for more information about large page sizes.

### 22.22.2 CD and NW Cache Control Flags

The CD and NW flags in control register CR0 were introduced in the Intel486 processor. In the P6 family and Pentium processors, these flags are used to implement a writeback strategy for the data cache; in the Intel486 processor, they implement a write-through strategy. See Table 11-5 for a comparison of these bits on the P6 family, Pentium, and Intel486 processors. For complete information on caching, see Chapter 11, "Memory Cache Control."

### 22.22.3 Descriptor Types and Contents

Operating-system code that manages space in descriptor tables often contains an invalid value in the access-rights field of descriptor-table entries to identify unused entries. Access rights values of 80H and 00H remain invalid for the P6 family, Pentium, Intel486, Intel386, and Intel 286 processors. Other values that were invalid on the Intel 286 processor may be valid on the 32-bit processors because uses for these bits have been defined.



## 22.22.4 Changes in Segment Descriptor Loads

On the Intel386 processor, loading a segment descriptor always causes a locked read and write to set the accessed bit of the descriptor. On the P6 family, Pentium, and Intel486 processors, the locked read and write occur only if the bit is not already set.

## 22.23 DEBUG FACILITIES

The P6 family and Pentium processors include extensions to the Intel486 processor debugging support for breakpoints. To use the new breakpoint features, it is necessary to set the DE flag in control register CR4.

### 22.23.1 Differences in Debug Register DR6

It is not possible to write a 1 to reserved bit 12 in debug status register DR6 on the P6 family and Pentium processors; however, it is possible to write a 1 in this bit on the Intel486 processor. See Table 9-1 for the different setting of this register following a power-up or hardware reset.

### 22.23.2 Differences in Debug Register DR7

The P6 family and Pentium processors determines the type of breakpoint access by the R/W0 through R/W3 fields in debug control register DR7 as follows:

- 00 Break on instruction execution only.
- 01 Break on data writes only.
- 10 Undefined if the DE flag in control register CR4 is cleared; break on I/O reads or writes but not instruction fetches if the DE flag in control register CR4 is set.
- 11 Break on data reads or writes but not instruction fetches.

On the P6 family and Pentium processors, reserved bits 11, 12, 14 and 15 are hard-wired to 0. On the Intel486 processor, however, bit 12 can be set. See Table 9-1 for the different settings of this register following a power-up or hardware reset.

### 22.23.3 Debug Registers DR4 and DR5

Although the DR4 and DR5 registers are documented as reserved, previous generations of processors aliased references to these registers to debug registers DR6 and DR7, respectively. When debug extensions are not enabled (the DE flag in control register CR4 is cleared), the P6 family and Pentium processors remain compatible with existing software by allowing these aliased references. When debug extensions are enabled (the DE flag is set), attempts to reference registers DR4 or DR5 will result in an invalid-opcode exception (#UD).

## 22.24 RECOGNITION OF BREAKPOINTS

For the Pentium processor, it is recommended that debuggers execute the LGDT instruction before returning to the program being debugged to ensure that breakpoints are detected. This operation does not need to be performed on the P6 family, Intel486, or Intel386 processors.

The implementation of test registers on the Intel486 processor used for testing the cache and TLB has been redesigned using MSRs on the P6 family and Pentium processors. (Note that MSRs used for this function are different on the P6 family and Pentium processors.) The MOV to and from test register instructions generate invalid-opcode exceptions (#UD) on the P6 family processors.

## 22.25 EXCEPTIONS AND/OR EXCEPTION CONDITIONS

This section describes the new exceptions and exception conditions added to the 32-bit IA-32 processors and implementation differences in existing exception handling. See Chapter 6, "Interrupt and Exception Handling," for a detailed description of the IA-32 exceptions.

The Pentium III processor introduced new state with the XMM registers. Computations involving data in these registers can produce exceptions. A new MXCSR control/status register is used to determine which exception or exceptions have occurred. When an exception associated with the XMM registers occurs, an interrupt is generated.

- SIMD floating-point exception (#XM, interrupt 19) — New exceptions associated with the SIMD floating-point registers and resulting computations.

No new exceptions were added with the Pentium Pro and Pentium II processors. The set of available exceptions is the same as for the Pentium processor. However, the following exception condition was added to the IA-32 with the Pentium Pro processor:

- Machine-check exception (#MC, interrupt 18) — New exception conditions. Many exception conditions have been added to the machine-check exception and a new architecture has been added for handling and reporting on hardware errors. See Chapter 15, "Machine-Check Architecture," for a detailed description of the new conditions.

The following exceptions and/or exception conditions were added to the IA-32 with the Pentium processor:

- Machine-check exception (#MC, interrupt 18) — New exception. This exception reports parity and other hardware errors. It is a model-specific exception and may not be implemented or implemented differently in future processors. The MCE flag in control register CR4 enables the machine-check exception. When this bit is clear (which it is at reset), the processor inhibits generation of the machine-check exception.
- General-protection exception (#GP, interrupt 13) — New exception condition added. An attempt to write a 1 to a reserved bit position of a special register causes a general-protection exception to be generated.
- Page-fault exception (#PF, interrupt 14) — New exception condition added. When a 1 is detected in any of the reserved bit positions of a page-table entry, page-directory entry, or page-directory pointer during address translation, a page-fault exception is generated.

The following exception was added to the Intel486 processor:

- Alignment-check exception (#AC, interrupt 17) — New exception. Reports unaligned memory references when alignment checking is being performed.

The following exceptions and/or exception conditions were added to the Intel386 processor:

- Divide-error exception (#DE, interrupt 0)
  - Change in exception handling. Divide-error exceptions on the Intel386 processors always leave the saved CS:IP value pointing to the instruction that failed. On the 8086 processor, the CS:IP value points to the next instruction.
  - Change in exception handling. The Intel386 processors can generate the largest negative number as a quotient for the IDIV instruction (80H and 8000H). The 8086 processor generates a divide-error exception instead.
- Invalid-opcode exception (#UD, interrupt 6) — New exception condition added. Improper use of the LOCK instruction prefix can generate an invalid-opcode exception.
- Page-fault exception (#PF, interrupt 14) — New exception condition added. If paging is enabled in a 16-bit program, a page-fault exception can be generated as follows. Paging can be used in a system with 16-bit tasks if all tasks use the same page directory. Because there is no place in a 16-bit TSS to store the PDBR register, switching to a 16-bit task does not change the value of the PDBR register. Tasks ported from the Intel 286 processor should be given 32-bit TSSs so they can make full use of paging.
- General-protection exception (#GP, interrupt 13) — New exception condition added. The Intel386 processor sets a limit of 15 bytes on instruction length. The only way to violate this limit is by putting redundant prefixes before an instruction. A general-protection exception is generated if the limit on instruction length is violated. The 8086 processor has no instruction length limit.

## 22.25.1 Machine-Check Architecture

The Pentium Pro processor introduced a new architecture to the IA-32 for handling and reporting on machine-check exceptions. This machine-check architecture (described in detail in Chapter 15, “Machine-Check Architecture”) greatly expands the ability of the processor to report on internal hardware errors.

## 22.25.2 Priority of Exceptions

The priority of exceptions are broken down into several major categories:

1. Traps on the previous instruction
2. External interrupts
3. Faults on fetching the next instruction
4. Faults in decoding the next instruction
5. Faults on executing an instruction

There are no changes in the priority of these major categories between the different processors, however, exceptions within these categories are implementation dependent and may change from processor to processor.

## 22.25.3 Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers

MMX instructions and a subset of SSE, SSE2, SSSE3 instructions operate on MMX registers. The exception conditions of these instructions are described in the following tables.

**Table 22-4. Exception Conditions for Legacy SIMD/MMX Instructions with FP Exception and 16-Byte Alignment**

Exception	Real	Virtual-8086	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X	X	X	If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0.
	X	X	X	X	If CRO.EM[bit 2] = 1. If CR4.OSFXSR[bit 9] = 0.
	X	X	X	X	If preceded by a LOCK prefix (FOH)
	X	X	X	X	If any corresponding CPUID feature flag is '0'
#MF	X	X	X	X	If there is a pending X87 FPU exception
#NM	X	X	X	X	If CRO.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	X	X	X	Legacy SSE: Memory operand is not 16-byte aligned
			X		For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If the memory address is in a non-canonical form.
	X	X			If any part of the operand lies outside the effective address space from 0 to FFFFH
#PF(fault-code)		X	X	X	For a page fault
#XM	X	X	X	X	If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1
Applicable Instructions	CVTPD2PI, CVTTPD2PI				

**Table 22-5. Exception Conditions for Legacy SIMD/MMX Instructions with XMM and FP Exception**

Exception	Real	Virtual-8086	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X	X	X	If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0.
	X	X	X	X	If CR0.EM[bit 2] = 1. If CR4.OSFXSR[bit 9] = 0.
	X	X	X	X	If preceded by a LOCK prefix (FOH)
	X	X	X	X	If any corresponding CPUID feature flag is '0'
#MF	X	X	X	X	If there is a pending X87 FPU exception
#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.
				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
#PF(fault-code)		X	X	X	For a page fault
Alignment Check #AC(0)		X	X	X	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
SIMD Floating-point Exception, #XM	X	X	X	X	If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1
Applicable Instructions	CVTPI2PS, CVTPS2PI, CVTTPS2PI				

Table 22-6. Exception Conditions for Legacy SIMD/MMX Instructions with XMM and without FP Exception

Exception	Real	Virtual-8086	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X	X	X	If CRO.EM[bit 2] = 1. If CR4.OSFXSR[bit 9] = 0.
	X	X	X	X	If preceded by a LOCK prefix (F0H)
	X	X	X	X	If any corresponding CPUID feature flag is '0'
#MF <sup>1</sup>	X	X	X	X	If there is a pending X87 FPU exception
#NM	X	X	X	X	If CRO.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.
				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
#PF(fault-code)		X	X	X	For a page fault
Alignment Check #AC(0)		X	X	X	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
Applicable Instructions	CVTPI2PD				

**NOTES:**

1. Applies to "CVTPI2PD xmm, mm" but not "CVTPI2PD xmm, m64".

**Table 22-7. Exception Conditions for SIMD/MMX Instructions with Memory Reference**

Exception	Real	Virtual-8086	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X	X	X	If CR0.EM[bit 2] = 1.
	X	X	X	X	If preceded by a LOCK prefix (FOH)
	X	X	X	X	If any corresponding CPUID feature flag is '0'
#MF	X	X	X	X	If there is a pending X87 FPU exception
#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.
				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
#PF(fault-code)		X	X	X	For a page fault
Alignment Check #AC(0)		X	X	X	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
Applicable Instructions	PABSB, PABSD, PABSW, PACKSSWB, PACKSSDW, PACKUSWB, PADDB, PADDD, PADDQ, PADDW, PADDSB, PADDSD, PADDUSB, PADDUSW, PALIGNR, PAND, PANDN, PAVGB, PAVGW, PCMPEQB, PCMPEQD, PCMPEQW, PCMPGTB, PCMPGTD, PCMPGTW, PHADDD, PHADDW, PHADDSW, PHSUBD, PHSUBW, PHSUBSW, PINSRW, PMADDUBSW, PMADDWD, PMAXS, PMAXSUB, PMINSW, PMINUB, PMULHRW, PMULHUW, PMULHW, PMULLW, PMULUDQ, PSADBW, PSHUFB, PSHUFW, PSIGNB, PSIGND, PSIGNW, PSLW, PSLD, PSLQ, PSRAD, PSRAW, PSRLW, PSRLD, PSRLQ, PSUBB, PSUBD, PSUBQ, PSUBW, PSUBSB, PSUBSW, PSUBUSB, PSUBUSW, PUNPCKHBW, PUNPCKHWD, PUNPCKHDQ, PUNPCKLBW, PUNPCKLWD, PUNPCKLDQ, PXOR				

Table 22-8. Exception Conditions for Legacy SIMD/MMX Instructions without FP Exception

Exception	Real	Virtual-8086	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X	X	X	If CRO.EM[bit 2] = 1. If ModR/M.mod ≠ 11b <sup>1</sup>
	X	X	X	X	If preceded by a LOCK prefix (FOH)
	X	X	X	X	If any corresponding CPUID feature flag is '0'
#MF	X	X	X	X	If there is a pending X87 FPU exception
#NM	X	X	X	X	If CRO.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
#GP(0)			X		For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments. If the destination operand is in a non-writable segment. <sup>2</sup> If the DS, ES, FS, or GS register contains a NULL segment selector. <sup>3</sup>
				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
#PF(fault-code)		X	X	X	For a page fault
#AC(0)		X	X	X	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
Applicable Instructions	MASKMOVQ, MOVNTQ, "MOVQ (mmreg)"				

**NOTES:**

1. Applies to MASKMOVQ only.
2. Applies to MASKMOVQ and MOVQ (mmreg) only.
3. Applies to MASKMOVQ only.

Table 22-9. Exception Conditions for Legacy SIMD/MMX Instructions without Memory Reference

Exception	Real	Virtual-8086	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X	X	X	If CR0.EM[bit 2] = 1.
	X	X	X	X	If preceded by a LOCK prefix (FOH)
	X	X	X	X	If any corresponding CPUID feature flag is '0'
#MF	X	X	X	X	If there is a pending X87 FPU exception
#NM			X	X	If CR0.TS[bit 3]=1
Applicable Instructions	PEXTRW, PMOVMASKB				

## 22.26 INTERRUPTS

The following differences in handling interrupts are found among the IA-32 processors.

### 22.26.1 Interrupt Propagation Delay

External hardware interrupts may be recognized on different instruction boundaries on the P6 family, Pentium, Intel486, and Intel386 processors, due to the superscaler designs of the P6 family and Pentium processors. Therefore, the EIP pushed onto the stack when servicing an interrupt may be different for the P6 family, Pentium, Intel486, and Intel386 processors.

### 22.26.2 NMI Interrupts

After an NMI interrupt is recognized by the P6 family, Pentium, Intel486, Intel386, and Intel 286 processors, the NMI interrupt is masked until the first IRET instruction is executed, unlike the 8086 processor.

### 22.26.3 IDT Limit

The LIDT instruction can be used to set a limit on the size of the IDT. A double-fault exception (#DF) is generated if an interrupt or exception attempts to read a vector beyond the limit. Shutdown then occurs on the 32-bit IA-32 processors if the double-fault handler vector is beyond the limit. (The 8086 processor does not have a shutdown mode nor a limit.)

## 22.27 ADVANCED PROGRAMMABLE INTERRUPT CONTROLLER (APIC)

The Advanced Programmable Interrupt Controller (APIC), referred to in this book as the **local APIC**, was introduced into the IA-32 processors with the Pentium processor (beginning with the 735/90 and 815/100 models) and is included in the Pentium 4, Intel Xeon, and P6 family processors. The features and functions of the local APIC are derived from the Intel 82489DX external APIC, which was used with the Intel486 and early Pentium processors. Additional refinements of the local APIC architecture were incorporated in the Pentium 4 and Intel Xeon processors.



### 22.27.1 Software Visible Differences Between the Local APIC and the 82489DX

The following features in the local APIC features differ from those found in the 82489DX external APIC:

- When the local APIC is disabled by clearing the APIC software enable/disable flag in the spurious-interrupt vector MSR, the state of its internal registers are unaffected, except that the mask bits in the LVT are all set to block local interrupts to the processor. Also, the local APIC ceases accepting IPIs except for INIT, SMI, NMI, and start-up IPIs. In the 82489DX, when the local unit is disabled, all the internal registers including the IRR, ISR and TMR are cleared and the mask bits in the LVT are set. In this state, the 82489DX local unit will accept only the reset deassert message.
- In the local APIC, NMI and INIT (except for INIT deassert) are always treated as edge triggered interrupts, even if programmed otherwise. In the 82489DX, these interrupts are always level triggered.
- In the local APIC, IPIs generated through the ICR are always treated as edge triggered (except INIT Deassert). In the 82489DX, the ICR can be used to generate either edge or level triggered IPIs.
- In the local APIC, the logical destination register supports 8 bits; in the 82489DX, it supports 32 bits.
- In the local APIC, the APIC ID register is 4 bits wide; in the 82489DX, it is 8 bits wide.
- The remote read delivery mode provided in the 82489DX and local APIC for Pentium processors is not supported in the local APIC in the Pentium 4, Intel Xeon, and P6 family processors.
- For the 82489DX, in the lowest priority delivery mode, all the target local APICs specified by the destination field participate in the lowest priority arbitration. For the local APIC, only those local APICs which have free interrupt slots will participate in the lowest priority arbitration.

### 22.27.2 New Features Incorporated in the Local APIC for the P6 Family and Pentium Processors

The local APIC in the Pentium and P6 family processors have the following new features not found in the 82489DX external APIC.

- Cluster addressing is supported in logical destination mode.
- Focus processor checking can be enabled/disabled.
- Interrupt input signal polarity can be programmed for the LINT0 and LINT1 pins.
- An SMI IPI is supported through the ICR and I/O redirection table.
- An error status register is incorporated into the LVT to log and report APIC errors.

In the P6 family processors, the local APIC incorporates an additional LVT register to handle performance monitoring counter interrupts.

### 22.27.3 New Features Incorporated in the Local APIC of the Pentium 4 and Intel Xeon Processors

The local APIC in the Pentium 4 and Intel Xeon processors has the following new features not found in the P6 family and Pentium processors and in the 82489DX.

- The local APIC ID is extended to 8 bits.
- An thermal sensor register is incorporated into the LVT to handle thermal sensor interrupts.
- The the ability to deliver lowest-priority interrupts to a focus processor is no longer supported.
- The flat cluster logical destination mode is not supported.

## 22.28 TASK SWITCHING AND TSS

This section identifies the implementation differences of task switching, additions to the TSS and the handling of TSSs and TSS segment selectors.

## 22.28.1 P6 Family and Pentium Processor TSS

When the virtual mode extensions are enabled (by setting the VME flag in control register CR4), the TSS in the P6 family and Pentium processors contain an interrupt redirection bit map, which is used in virtual-8086 mode to redirect interrupts back to an 8086 program.

## 22.28.2 TSS Selector Writes

During task state saves, the Intel486 processor writes 2-byte segment selectors into a 32-bit TSS, leaving the upper 16 bits undefined. For performance reasons, the P6 family and Pentium processors write 4-byte segment selectors into the TSS, with the upper 2 bytes being 0. For compatibility reasons, code should not depend on the value of the upper 16 bits of the selector in the TSS.

## 22.28.3 Order of Reads/Writes to the TSS

The order of reads and writes into the TSS is processor dependent. The P6 family and Pentium processors may generate different page-fault addresses in control register CR2 in the same TSS area than the Intel486 and Intel386 processors, if a TSS crosses a page boundary (which is not recommended).

## 22.28.4 Using A 16-Bit TSS with 32-Bit Constructs

Task switches using 16-bit TSSs should be used only for pure 16-bit code. Any new code written using 32-bit constructs (operands, addressing, or the upper word of the EFLAGS register) should use only 32-bit TSSs. This is due to the fact that the 32-bit processors do not save the upper 16 bits of EFLAGS to a 16-bit TSS. A task switch back to a 16-bit task that was executing in virtual mode will never re-enable the virtual mode, as this flag was not saved in the upper half of the EFLAGS value in the TSS. Therefore, it is strongly recommended that any code using 32-bit constructs use a 32-bit TSS to ensure correct behavior in a multitasking environment.

## 22.28.5 Differences in I/O Map Base Addresses

The Intel486 processor considers the TSS segment to be a 16-bit segment and wraps around the 64K boundary. Any I/O accesses check for permission to access this I/O address at the I/O base address plus the I/O offset. If the I/O map base address exceeds the specified limit of 0DFFFH, an I/O access will wrap around and obtain the permission for the I/O address at an incorrect location within the TSS. A TSS limit violation does not occur in this situation on the Intel486 processor. However, the P6 family and Pentium processors consider the TSS to be a 32-bit segment and a limit violation occurs when the I/O base address plus the I/O offset is greater than the TSS limit. By following the recommended specification for the I/O base address to be less than 0DFFFH, the Intel486 processor will not wrap around and access incorrect locations within the TSS for I/O port validation and the P6 family and Pentium processors will not experience general-protection exceptions (#GP). Figure 22-1 demonstrates the different areas accessed by the Intel486 and the P6 family and Pentium processors.

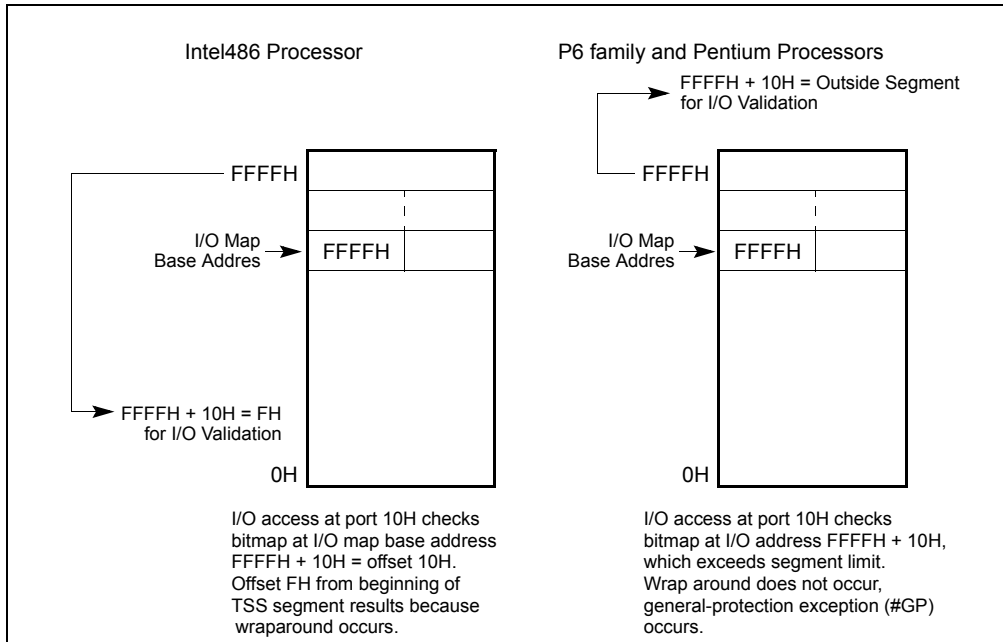


Figure 22-1. I/O Map Base Address Differences

## 22.29 CACHE MANAGEMENT

The P6 family processors include two levels of internal caches: L1 (level 1) and L2 (level 2). The L1 cache is divided into an instruction cache and a data cache; the L2 cache is a general-purpose cache. See Section 11.1, "Internal Caches, TLBs, and Buffers," for a description of these caches. (Note that although the Pentium II processor L2 cache is physically located on a separate chip in the cassette, it is considered an internal cache.)

The Pentium processor includes separate level 1 instruction and data caches. The data cache supports a writeback (or alternatively write-through, on a line by line basis) policy for memory updates.

The Intel486 processor includes a single level 1 cache for both instructions and data.

The meaning of the CD and NW flags in control register CR0 have been redefined for the P6 family and Pentium processors. For these processors, the recommended value (00B) enables writeback for the data cache of the Pentium processor and for the L1 data cache and L2 cache of the P6 family processors. In the Intel486 processor, setting these flags to (00B) enables write-through for the cache.

External system hardware can force the Pentium processor to disable caching or to use the write-through cache policy should that be required. In the P6 family processors, the MTRRs can be used to override the CD and NW flags (see Table 11-6).

The P6 family and Pentium processors support page-level cache management in the same manner as the Intel486 processor by using the PCD and PWT flags in control register CR3, the page-directory entries, and the page-table entries. The Intel486 processor, however, is not affected by the state of the PWT flag since the internal cache of the Intel486 processor is a write-through cache.

### 22.29.1 Self-Modifying Code with Cache Enabled

On the Intel486 processor, a write to an instruction in the cache will modify it in both the cache and memory. If the instruction was prefetched before the write, however, the old version of the instruction could be the one executed. To prevent this problem, it is necessary to flush the instruction prefetch unit of the Intel486 processor by coding a jump instruction immediately after any write that modifies an instruction. The P6 family and Pentium processors, however, check whether a write may modify an instruction that has been prefetched for execution. This check is based on the linear address of the instruction. If the linear address of an instruction is found to be present in the

prefetch queue, the P6 family and Pentium processors flush the prefetch queue, eliminating the need to code a jump instruction after any writes that modify an instruction.

Because the linear address of the write is checked against the linear address of the instructions that have been prefetched, special care must be taken for self-modifying code to work correctly when the physical addresses of the instruction and the written data are the same, but the linear addresses differ. In such cases, it is necessary to execute a serializing operation to flush the prefetch queue after the write and before executing the modified instruction. See Section 8.3, “Serializing Instructions,” for more information on serializing instructions.

### NOTE

The check on linear addresses described above is not in practice a concern for compatibility. Applications that include self-modifying code use the same linear address for modifying and fetching the instruction. System software, such as a debugger, that might possibly modify an instruction using a different linear address than that used to fetch the instruction must execute a serializing operation, such as IRET, before the modified instruction is executed.

## 22.29.2 Disabling the L3 Cache

A unified third-level (L3) cache in processors based on Intel NetBurst microarchitecture (see Section 11.1, “Internal Caches, TLBs, and Buffers”) provides the third-level cache disable flag, bit 6 of the IA32\_MISC\_ENABLE MSR. The third-level cache disable flag allows the L3 cache to be disabled and enabled, independently of the L1 and L2 caches (see Section 11.5.4, “Disabling and Enabling the L3 Cache”). The third-level cache disable flag applies only to processors based on Intel NetBurst microarchitecture. Processors with L3 and based on other microarchitectures do not support the third-level cache disable flag.

## 22.30 PAGING

This section identifies enhancements made to the paging mechanism and implementation differences in the paging mechanism for various IA-32 processors.

### 22.30.1 Large Pages

The Pentium processor extended the memory management/paging facilities of the IA-32 to allow large (4 MBytes) pages sizes (see Section 4.3, “32-Bit Paging”). The first P6 family processor (the Pentium Pro processor) added a 2 MByte page size to the IA-32 in conjunction with the physical address extension (PAE) feature (see Section 4.4, “PAE Paging”).

The availability of large pages with 32-bit paging on any IA-32 processor can be determined via feature bit 3 (PSE) of register EDX after the CPUID instruction has been execution with an argument of 1. (Large pages are always available with PAE paging and IA-32e paging.) Intel processors that do not support the CPUID instruction support only 32-bit paging and do not support page size enhancements. (See “CPUID—CPU Identification” in Chapter 3, “Instruction Set Reference, A-L,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* for more information on the CPUID instruction.)

### 22.30.2 PCD and PWT Flags

The PCD and PWT flags were introduced to the IA-32 in the Intel486 processor to control the caching of pages:

- PCD (page-level cache disable) flag—Controls caching on a page-by-page basis.
- PWT (page-level write-through) flag—Controls the write-through/writeback caching policy on a page-by-page basis. Since the internal cache of the Intel486 processor is a write-through cache, it is not affected by the state of the PWT flag.

### 22.30.3 Enabling and Disabling Paging

Paging is enabled and disabled by loading a value into control register CR0 that modifies the PG flag. For backward and forward compatibility with all IA-32 processors, Intel recommends that the following operations be performed when enabling or disabling paging:

1. Execute a MOV CR0, REG instruction to either set (enable paging) or clear (disable paging) the PG flag.
2. Execute a near JMP instruction.

The sequence bounded by the MOV and JMP instructions should be identity mapped (that is, the instructions should reside on a page whose linear and physical addresses are identical).

For the P6 family processors, the MOV CR0, REG instruction is serializing, so the jump operation is not required. However, for backwards compatibility, the JMP instruction should still be included.

## 22.31 STACK OPERATIONS

This section identifies the differences in the stack mechanism for the various IA-32 processors.

### 22.31.1 Selector Pushes and Pops

When pushing a segment selector onto the stack, the Pentium 4, Intel Xeon, P6 family, and Intel486 processors decrement the ESP register by the operand size and then write 2 bytes. If the operand size is 32-bits, the upper two bytes of the write are not modified. The Pentium processor decrements the ESP register by the operand size and determines the size of the write by the operand size. If the operand size is 32-bits, the upper two bytes are written as 0s.

When popping a segment selector from the stack, the Pentium 4, Intel Xeon, P6 family, and Intel486 processors read 2 bytes and increment the ESP register by the operand size of the instruction. The Pentium processor determines the size of the read from the operand size and increments the ESP register by the operand size.

It is possible to align a 32-bit selector push or pop such that the operation generates an exception on a Pentium processor and not on an Pentium 4, Intel Xeon, P6 family, or Intel486 processor. This could occur if the third and/or fourth byte of the operation lies beyond the limit of the segment or if the third and/or fourth byte of the operation is located on a non-present or inaccessible page.

For a POP-to-memory instruction that meets the following conditions:

- The stack segment size is 16-bit.
- Any 32-bit addressing form with the SIB byte specifying ESP as the base register.
- The initial stack pointer is FFFCH (32-bit operand) or FFFEh (16-bit operand) and will wrap around to 0H as a result of the POP operation.

The result of the memory write is implementation-specific. For example, in P6 family processors, the result of the memory write is SS:0H plus any scaled index and displacement. In Pentium processors, the result of the memory write may be either a stack fault (real mode or protected mode with stack segment size of 64 KByte), or write to SS:10000H plus any scaled index and displacement (protected mode and stack segment size exceeds 64 KByte).

### 22.31.2 Error Code Pushes

The Intel486 processor implements the error code pushed on the stack as a 16-bit value. When pushed onto a 32-bit stack, the Intel486 processor only pushes 2 bytes and updates ESP by 4. The P6 family and Pentium processors' error code is a full 32 bits with the upper 16 bits set to zero. The P6 family and Pentium processors, therefore, push 4 bytes and update ESP by 4. Any code that relies on the state of the upper 16 bits may produce inconsistent results.

### 22.31.3 Fault Handling Effects on the Stack

During the handling of certain instructions, such as CALL and PUSH, faults may occur in different sequences for the different processors. For example, during far calls, the Intel486 processor pushes the old CS and EIP before a possible branch fault is resolved. A branch fault is a fault from a branch instruction occurring from a segment limit or access rights violation. If a branch fault is taken, the Intel486 and P6 family processors will have corrupted memory below the stack pointer. However, the ESP register is backed up to make the instruction restartable. The P6 family processors issue the branch before the pushes. Therefore, if a branch fault does occur, these processors do not corrupt memory below the stack pointer. This implementation difference, however, does not constitute a compatibility problem, as only values at or above the stack pointer are considered to be valid. Other operations that encounter faults may also corrupt memory below the stack pointer and this behavior may vary on different implementations.

### 22.31.4 Interlevel RET/IRET From a 16-Bit Interrupt or Call Gate

If a call or interrupt is made from a 32-bit stack environment through a 16-bit gate, only 16 bits of the old ESP can be pushed onto the stack. On the subsequent RET/IRET, the 16-bit ESP is popped but the full 32-bit ESP is updated since control is being resumed in a 32-bit stack environment. The Intel486 processor writes the SS selector into the upper 16 bits of ESP. The P6 family and Pentium processors write zeros into the upper 16 bits.

## 22.32 MIXING 16- AND 32-BIT SEGMENTS

The features of the 16-bit Intel 286 processor are an object-code compatible subset of those of the 32-bit IA-32 processors. The D (default operation size) flag in segment descriptors indicates whether the processor treats a code or data segment as a 16-bit or 32-bit segment; the B (default stack size) flag in segment descriptors indicates whether the processor treats a stack segment as a 16-bit or 32-bit segment.

The segment descriptors used by the Intel 286 processor are supported by the 32-bit IA-32 processors if the Intel-reserved word (highest word) of the descriptor is clear. On the 32-bit IA-32 processors, this word includes the upper bits of the base address and the segment limit.

The segment descriptors for data segments, code segments, local descriptor tables (there are no descriptors for global descriptor tables), and task gates are the same for the 16- and 32-bit processors. Other 16-bit descriptors (TSS segment, call gate, interrupt gate, and trap gate) are supported by the 32-bit processors.

The 32-bit processors also have descriptors for TSS segments, call gates, interrupt gates, and trap gates that support the 32-bit architecture. Both kinds of descriptors can be used in the same system.

For those segment descriptors common to both 16- and 32-bit processors, clear bits in the reserved word cause the 32-bit processors to interpret these descriptors exactly as an Intel 286 processor does, that is:

- Base Address — The upper 8 bits of the 32-bit base address are clear, which limits base addresses to 24 bits.
- Limit — The upper 4 bits of the limit field are clear, restricting the value of the limit field to 64 KBytes.
- Granularity bit — The G (granularity) flag is clear, indicating the value of the 16-bit limit is interpreted in units of 1 byte.
- Big bit — In a data-segment descriptor, the B flag is clear in the segment descriptor used by the 32-bit processors, indicating the segment is no larger than 64 KBytes.
- Default bit — In a code-segment descriptor, the D flag is clear, indicating 16-bit addressing and operands are the default. In a stack-segment descriptor, the D flag is clear, indicating use of the SP register (instead of the ESP register) and a 64-KByte maximum segment limit.

For information on mixing 16- and 32-bit code in applications, see Chapter 21, "Mixing 16-Bit and 32-Bit Code."

## 22.33 SEGMENT AND ADDRESS WRAPAROUND

This section discusses differences in segment and address wraparound between the P6 family, Pentium, Intel486, Intel386, Intel 286, and 8086 processors.

### 22.33.1 Segment Wraparound

On the 8086 processor, an attempt to access a memory operand that crosses offset 65,535 or 0FFFFH or offset 0 (for example, moving a word to offset 65,535 or pushing a word when the stack pointer is set to 1) causes the offset to wrap around modulo 65,536 or 010000H. With the Intel 286 processor, any base and offset combination that addresses beyond 16 MBytes wraps around to the 1 MByte of the address space. The P6 family, Pentium, Intel486, and Intel386 processors in real-address mode generate an exception in these cases:

- A general-protection exception (#GP) if the segment is a data segment (that is, if the CS, DS, ES, FS, or GS register is being used to address the segment).
- A stack-fault exception (#SS) if the segment is a stack segment (that is, if the SS register is being used).

An exception to this behavior occurs when a stack access is data aligned, and the stack pointer is pointing to the last aligned piece of data that size at the top of the stack (ESP is FFFFFFFCH). When this data is popped, no segment limit violation occurs and the stack pointer will wrap around to 0.

The address space of the P6 family, Pentium, and Intel486 processors may wraparound at 1 MByte in real-address mode. An external A20M# pin forces wraparound if enabled. On Intel 8086 processors, it is possible to specify addresses greater than 1 MByte. For example, with a selector value FFFFH and an offset of FFFFH, the effective address would be 10FFE7H (1 MByte plus 65519 bytes). The 8086 processor, which can form addresses up to 20 bits long, truncates the uppermost bit, which “wraps” this address to FFE7H. However, the P6 family, Pentium, and Intel486 processors do not truncate this bit if A20M# is not enabled.

If a stack operation wraps around the address limit, shutdown occurs. (The 8086 processor does not have a shutdown mode or a limit.)

The behavior when executing near the limit of a 4-GByte selector (limit = FFFFFFFFH) is different between the Pentium Pro and the Pentium 4 family of processors. On the Pentium Pro, instructions which cross the limit -- for example, a two byte instruction such as INC EAX that is encoded as FFH C0H starting exactly at the limit faults for a segment violation (a one byte instruction at FFFFFFFFH does not cause an exception). Using the Pentium 4 microprocessor family, neither of these situations causes a fault.

Segment wraparound and the functionality of A20M# is used primarily by older operating systems and not used by modern operating systems. On newer Intel 64 processors, A20M# may be absent.

## 22.34 STORE BUFFERS AND MEMORY ORDERING

The Pentium 4, Intel Xeon, and P6 family processors provide a store buffer for temporary storage of writes (stores) to memory (see Section 11.10, “Store Buffer”). Writes stored in the store buffer(s) are always written to memory in program order, with the exception of “fast string” store operations (see Section 8.2.4, “Fast-String Operation and Out-of-Order Stores”).

The Pentium processor has two store buffers, one corresponding to each of the pipelines. Writes in these buffers are always written to memory in the order they were generated by the processor core.

It should be noted that only memory writes are buffered and I/O writes are not. The Pentium 4, Intel Xeon, P6 family, Pentium, and Intel486 processors do not synchronize the completion of memory writes on the bus and instruction execution after a write. An I/O, locked, or serializing instruction needs to be executed to synchronize writes with the next instruction (see Section 8.3, “Serializing Instructions”).

The Pentium 4, Intel Xeon, and P6 family processors use processor ordering to maintain consistency in the order that data is read (loaded) and written (stored) in a program and the order the processor actually carries out the reads and writes. With this type of ordering, reads can be carried out speculatively and in any order, reads can pass buffered writes, and writes to memory are always carried out in program order. (See Section 8.2, “Memory Ordering,” for more information about processor ordering.) The Pentium III processor introduced a new instruction to serialize writes and make them globally visible. Memory ordering issues can arise between a producer and a consumer of data. The SFENCE instruction provides a performance-efficient way of ensuring ordering between routines that produce weakly-ordered results and routines that consume this data.

No re-ordering of reads occurs on the Pentium processor, except under the condition noted in Section 8.2.1, “Memory Ordering in the Intel® Pentium® and Intel486™ Processors,” and in the following paragraph describing the Intel486 processor.



Specifically, the store buffers are flushed before the IN instruction is executed. No reads (as a result of cache miss) are reordered around previously generated writes sitting in the store buffers. The implication of this is that the store buffers will be flushed or emptied before a subsequent bus cycle is run on the external bus.

On both the Intel486 and Pentium processors, under certain conditions, a memory read will go onto the external bus before the pending memory writes in the buffer even though the writes occurred earlier in the program execution. A memory read will only be reordered in front of all writes pending in the buffers if all writes pending in the buffers are cache hits and the read is a cache miss. Under these conditions, the Intel486 and Pentium processors will not read from an external memory location that needs to be updated by one of the pending writes.

During a locked bus cycle, the Intel486 processor will always access external memory, it will never look for the location in the on-chip cache. All data pending in the Intel486 processor's store buffers will be written to memory before a locked cycle is allowed to proceed to the external bus. Thus, the locked bus cycle can be used for eliminating the possibility of reordering read cycles on the Intel486 processor. The Pentium processor does check its cache on a read-modify-write access and, if the cache line has been modified, writes the contents back to memory before locking the bus. The P6 family processors write to their cache on a read-modify-write operation (if the access does not split across a cache line) and does not write back to system memory. If the access does split across a cache line, it locks the bus and accesses system memory.

I/O reads are never reordered in front of buffered memory writes on an IA-32 processor. This ensures an update of all memory locations before reading the status from an I/O device.

## 22.35 BUS LOCKING

The Intel 286 processor performs the bus locking differently than the Intel P6 family, Pentium, Intel486, and Intel386 processors. Programs that use forms of memory locking specific to the Intel 286 processor may not run properly when run on later processors.

A locked instruction is guaranteed to lock only the area of memory defined by the destination operand, but may lock a larger memory area. For example, typical 8086 and Intel 286 configurations lock the entire physical memory space. Programmers should not depend on this.

On the Intel 286 processor, the LOCK prefix is sensitive to IOPL. If the CPL is greater than the IOPL, a general-protection exception (#GP) is generated. On the Intel386 DX, Intel486, and Pentium, and P6 family processors, no check against IOPL is performed.

The Pentium processor automatically asserts the LOCK# signal when acknowledging external interrupts. After signaling an interrupt request, an external interrupt controller may use the data bus to send the interrupt vector to the processor. After receiving the interrupt request signal, the processor asserts LOCK# to insure that no other data appears on the data bus until the interrupt vector is received. This bus locking does not occur on the P6 family processors.

## 22.36 BUS HOLD

Unlike the 8086 and Intel 286 processors, but like the Intel386 and Intel486 processors, the P6 family and Pentium processors respond to requests for control of the bus from other potential bus masters, such as DMA controllers, between transfers of parts of an unaligned operand, such as two words which form a doubleword. Unlike the Intel386 processor, the P6 family, Pentium and Intel486 processors respond to bus hold during reset initialization.

## 22.37 MODEL-SPECIFIC EXTENSIONS TO THE IA-32

Certain extensions to the IA-32 are specific to a processor or family of IA-32 processors and may not be implemented or implemented in the same way in future processors. The following sections describe these model-specific extensions. The CPUID instruction indicates the availability of some of the model-specific features.



### 22.37.1 Model-Specific Registers

The Pentium processor introduced a set of model-specific registers (MSRs) for use in controlling hardware functions and performance monitoring. To access these MSRs, two new instructions were added to the IA-32 architecture: read MSR (RDMSR) and write MSR (WRMSR). The MSRs in the Pentium processor are not guaranteed to be duplicated or provided in the next generation IA-32 processors.

The P6 family processors greatly increased the number of MSRs available to software. See Chapter 35, “Model-Specific Registers (MSRs),” for a complete list of the available MSRs. The new registers control the debug extensions, the performance counters, the machine-check exception capability, the machine-check architecture, and the MTRRs. These registers are accessible using the RDMSR and WRMSR instructions. Specific information on some of these new MSRs is provided in the following sections. As with the Pentium processor MSR, the P6 family processor MSRs are not guaranteed to be duplicated or provided in the next generation IA-32 processors.

### 22.37.2 RDMSR and WRMSR Instructions

The RDMSR (read model-specific register) and WRMSR (write model-specific register) instructions recognize a much larger number of model-specific registers in the P6 family processors. (See “RDMSR—Read from Model Specific Register” and “WRMSR—Write to Model Specific Register” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volumes 2A, 2B, 2C & 2D* for more information.)

### 22.37.3 Memory Type Range Registers

Memory type range registers (MTRRs) are a new feature introduced into the IA-32 in the Pentium Pro processor. MTRRs allow the processor to optimize memory operations for different types of memory, such as RAM, ROM, frame buffer memory, and memory-mapped I/O.

MTRRs are MSRs that contain an internal map of how physical address ranges are mapped to various types of memory. The processor uses this internal memory map to determine the cacheability of various physical memory locations and the optimal method of accessing memory locations. For example, if a memory location is specified in an MTRR as write-through memory, the processor handles accesses to this location as follows. It reads data from that location in lines and caches the read data or maps all writes to that location to the bus and updates the cache to maintain cache coherency. In mapping the physical address space with MTRRs, the processor recognizes five types of memory: uncacheable (UC), uncacheable, speculatable, write-combining (WC), write-through (WT), write-protected (WP), and writeback (WB).

Earlier IA-32 processors (such as the Intel486 and Pentium processors) used the KEN# (cache enable) pin and external logic to maintain an external memory map and signal cacheable accesses to the processor. The MTRR mechanism simplifies hardware designs by eliminating the KEN# pin and the external logic required to drive it.

See Chapter 9, “Processor Management and Initialization,” and Chapter 35, “Model-Specific Registers (MSRs),” for more information on the MTRRs.

### 22.37.4 Machine-Check Exception and Architecture

The Pentium processor introduced a new exception called the machine-check exception (#MC, interrupt 18). This exception is used to detect hardware-related errors, such as a parity error on a read cycle.

The P6 family processors extend the types of errors that can be detected and that generate a machine-check exception. It also provides a new machine-check architecture for recording information about a machine-check error and provides extended recovery capability.

The machine-check architecture provides several banks of reporting registers for recording machine-check errors. Each bank of registers is associated with a specific hardware unit in the processor. The primary focus of the machine checks is on bus and interconnect operations; however, checks are also made of translation lookaside buffer (TLB) and cache operations.

The machine-check architecture can correct some errors automatically and allow for reliable restart of instruction execution. It also collects sufficient information for software to use in correcting other machine errors not corrected by hardware.

See Chapter 15, “Machine-Check Architecture,” for more information on the machine-check exception and the machine-check architecture.

### 22.37.5 Performance-Monitoring Counters

The P6 family and Pentium processors provide two performance-monitoring counters for use in monitoring internal hardware operations. The number of performance monitoring counters and associated programming interfaces may be implementation specific for Pentium 4 processors, Pentium M processors. Later processors may have implemented these as part of an architectural performance monitoring feature. The architectural and non-architectural performance monitoring interfaces for different processor families are described in Chapter 18, “Performance Monitoring,”. Chapter 19, “Performance Monitoring Events.” lists all the events that can be counted for architectural performance monitoring events and non-architectural events. The counters are set up, started, and stopped using two MSRs and the RDMSR and WRMSR instructions. For the P6 family processors, the current count for a particular counter can be read using the new RDPMC instruction.

The performance-monitoring counters are useful for debugging programs, optimizing code, diagnosing system failures, or refining hardware designs. See Chapter 18, “Performance Monitoring,” for more information on these counters.

## 22.38 TWO WAYS TO RUN INTEL 286 PROCESSOR TASKS

When porting 16-bit programs to run on 32-bit IA-32 processors, there are two approaches to consider:

- Porting an entire 16-bit software system to a 32-bit processor, complete with the old operating system, loader, and system builder. Here, all tasks will have 16-bit TSSs. The 32-bit processor is being used as if it were a faster version of the 16-bit processor.
- Porting selected 16-bit applications to run in a 32-bit processor environment with a 32-bit operating system, loader, and system builder. Here, the TSSs used to represent 286 tasks should be changed to 32-bit TSSs. It is possible to mix 16 and 32-bit TSSs, but the benefits are small and the problems are great. All tasks in a 32-bit software system should have 32-bit TSSs. It is not necessary to change the 16-bit object modules themselves; TSSs are usually constructed by the operating system, by the loader, or by the system builder. See Chapter 21, “Mixing 16-Bit and 32-Bit Code,” for more detailed information about mixing 16-bit and 32-bit code.

Because the 32-bit processors use the contents of the reserved word of 16-bit segment descriptors, 16-bit programs that place values in this word may not run correctly on the 32-bit processors.

## 22.39 INITIAL STATE OF PENTIUM, PENTIUM PRO AND PENTIUM 4 PROCESSORS

Table 22-10 shows the state of the flags and other registers following power-up for the Pentium, Pentium Pro and Pentium 4 processors. The state of control register CR0 is 60000010H (see Figure 9-1 “Contents of CR0 Register after Reset” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*). This places the processor in real-address mode with paging disabled.

**Table 22-10. Processor State Following Power-up/Reset/INIT for Pentium, Pentium Pro and Pentium 4 Processors**

Register	Pentium 4 Processor	Pentium Pro Processor	Pentium Processor
EFLAGS <sup>1</sup>	00000002H	00000002H	00000002H
EIP	0000FFF0H	0000FFF0H	0000FFF0H
CR0	60000010H <sup>2</sup>	60000010H <sup>2</sup>	60000010H <sup>2</sup>
CR2, CR3, CR4	00000000H	00000000H	00000000H

Table 22-10. Processor State Following Power-up/Reset/INIT for Pentium, Pentium Pro and Pentium 4 Processors

Register	Pentium 4 Processor	Pentium Pro Processor	Pentium Processor
CS	Selector = F000H Base = FFFF0000H Limit = FFFFH AR = Present, R/W, Accessed	Selector = F000H Base = FFFF0000H Limit = FFFFH AR = Present, R/W, Accessed	Selector = F000H Base = FFFF0000H Limit = FFFFH AR = Present, R/W, Accessed
SS, DS, ES, FS, GS	Selector = 0000H Base = 00000000H Limit = FFFFH AR = Present, R/W, Accessed	Selector = 0000H Base = 00000000H Limit = FFFFH AR = Present, R/W, Accessed	Selector = 0000H Base = 00000000H Limit = FFFFH AR = Present, R/W, Accessed
EDX	00000FxxH	000n06xxH <sup>3</sup>	000005xxH
EAX	0 <sup>4</sup>	0 <sup>4</sup>	0 <sup>4</sup>
EBX, ECX, ESI, EDI, EBP, ESP	00000000H	00000000H	00000000H
ST0 through ST7 <sup>5</sup>	Pwr up or Reset: +0.0 FINIT/FNINIT: Unchanged	Pwr up or Reset: +0.0 FINIT/FNINIT: Unchanged	Pwr up or Reset: +0.0 FINIT/FNINIT: Unchanged
x87 FPU Control Word <sup>5</sup>	Pwr up or Reset: 0040H FINIT/FNINIT: 037FH	Pwr up or Reset: 0040H FINIT/FNINIT: 037FH	Pwr up or Reset: 0040H FINIT/FNINIT: 037FH
x87 FPU Status Word <sup>5</sup>	Pwr up or Reset: 0000H FINIT/FNINIT: 0000H	Pwr up or Reset: 0000H FINIT/FNINIT: 0000H	Pwr up or Reset: 0000H FINIT/FNINIT: 0000H
x87 FPU Tag Word <sup>5</sup>	Pwr up or Reset: 5555H FINIT/FNINIT: FFFFH	Pwr up or Reset: 5555H FINIT/FNINIT: FFFFH	Pwr up or Reset: 5555H FINIT/FNINIT: FFFFH
x87 FPU Data Operand and CS Seg. Selectors <sup>5</sup>	Pwr up or Reset: 0000H FINIT/FNINIT: 0000H	Pwr up or Reset: 0000H FINIT/FNINIT: 0000H	Pwr up or Reset: 0000H FINIT/FNINIT: 0000H
x87 FPU Data Operand and Inst. Pointers <sup>5</sup>	Pwr up or Reset: 00000000H FINIT/FNINIT: 00000000H	Pwr up or Reset: 00000000H FINIT/FNINIT: 00000000H	Pwr up or Reset: 00000000H FINIT/FNINIT: 00000000H
MM0 through MM7 <sup>5</sup>	Pwr up or Reset: 0000000000000000H INIT or FINIT/FNINIT: Unchanged	Pentium II and Pentium III Processors Only— Pwr up or Reset: 0000000000000000H INIT or FINIT/FNINIT: Unchanged	Pentium with MMX Technology Only— Pwr up or Reset: 0000000000000000H INIT or FINIT/FNINIT: Unchanged
XMM0 through XMM7	Pwr up or Reset: 0H INIT: Unchanged	If CPUID.01H:SSE is 1 — Pwr up or Reset: 0H INIT: Unchanged	NA
MXCSR	Pwr up or Reset: 1F80H INIT: Unchanged	Pentium III processor only- Pwr up or Reset: 1F80H INIT: Unchanged	NA
GDTR, IDTR	Base = 00000000H Limit = FFFFH AR = Present, R/W	Base = 00000000H Limit = FFFFH AR = Present, R/W	Base = 00000000H Limit = FFFFH AR = Present, R/W
LDTR, Task Register	Selector = 0000H Base = 00000000H Limit = FFFFH AR = Present, R/W	Selector = 0000H Base = 00000000H Limit = FFFFH AR = Present, R/W	Selector = 0000H Base = 00000000H Limit = FFFFH AR = Present, R/W
DR0, DR1, DR2, DR3	00000000H	00000000H	00000000H
DR6	FFFF0FF0H	FFFF0FF0H	FFFF0FF0H

**Table 22-10. Processor State Following Power-up/Reset/INIT for Pentium, Pentium Pro and Pentium 4 Processors**

Register	Pentium 4 Processor	Pentium Pro Processor	Pentium Processor
DR7	00000400H	00000400H	00000400H
Time-Stamp Counter	Power up or Reset: 0H INIT: Unchanged	Power up or Reset: 0H INIT: Unchanged	Power up or Reset: 0H INIT: Unchanged
Perf. Counters and Event Select	Power up or Reset: 0H INIT: Unchanged	Power up or Reset: 0H INIT: Unchanged	Power up or Reset: 0H INIT: Unchanged
All Other MSRs	Pwr up or Reset: Undefined INIT: Unchanged	Pwr up or Reset: Undefined INIT: Unchanged	Pwr up or Reset: Undefined INIT: Unchanged
Data and Code Cache, TLBs	Invalid <sup>6</sup>	Invalid <sup>6</sup>	Invalid <sup>6</sup>
Fixed MTRRs	Pwr up or Reset: Disabled INIT: Unchanged	Pwr up or Reset: Disabled INIT: Unchanged	Not Implemented
Variable MTRRs	Pwr up or Reset: Disabled INIT: Unchanged	Pwr up or Reset: Disabled INIT: Unchanged	Not Implemented
Machine-Check Architecture	Pwr up or Reset: Undefined INIT: Unchanged	Pwr up or Reset: Undefined INIT: Unchanged	Not Implemented
APIC	Pwr up or Reset: Enabled INIT: Unchanged	Pwr up or Reset: Enabled INIT: Unchanged	Pwr up or Reset: Enabled INIT: Unchanged
R8-R15 <sup>7</sup>	0000000000000000H	0000000000000000H	N.A.
XMM8-XMM15 <sup>7</sup>	Pwr up or Reset: 0H INIT: Unchanged	Pwr up or Reset: 0H INIT: Unchanged	N.A.

**NOTES:**

1. The 10 most-significant bits of the EFLAGS register are undefined following a reset. Software should not depend on the states of any of these bits.
2. The CD and NW flags are unchanged, bit 4 is set to 1, all other bits are cleared.
3. Where “n” is the Extended Model Value for the respective processor.
4. If Built-In Self-Test (BIST) is invoked on power up or reset, EAX is 0 only if all tests passed. (BIST cannot be invoked during an INIT.)
5. The state of the x87 FPU and MMX registers is not changed by the execution of an INIT.
6. Internal caches are invalid after power-up and RESET, but left unchanged with an INIT.
7. If the processor supports IA-32e mode.

## 17. Updates to Chapter 29, Volume 3C

Change bars show changes to Chapter 1 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A: System Programming Guide, Part 1*.

-----  
Changes include...

...  
...  
...

# CHAPTER 29

## APIC VIRTUALIZATION AND VIRTUAL INTERRUPTS

---

The VMCS includes controls that enable the virtualization of interrupts and the Advanced Programmable Interrupt Controller (APIC).

When these controls are used, the processor will emulate many accesses to the APIC, track the state of the virtual APIC, and deliver virtual interrupts — all in VMX non-root operation with out a VM exit.<sup>1</sup>

The processor tracks the state of the virtual APIC using a virtual-APIC page identified by the virtual-machine monitor (VMM). Section 29.1 discusses the virtual-APIC page and how the processor uses it to track the state of the virtual APIC.

The following are the VM-execution controls relevant to APIC virtualization and virtual interrupts (see Section 24.6 for information about the locations of these controls):

- **Virtual-interrupt delivery.** This control enables the evaluation and delivery of pending virtual interrupts (Section 29.2). It also enables the emulation of writes (memory-mapped or MSR-based, as enabled) to the APIC registers that control interrupt prioritization.
- **Use TPR shadow.** This control enables emulation of accesses to the APIC's task-priority register (TPR) via CR8 (Section 29.3) and, if enabled, via the memory-mapped or MSR-based interfaces.
- **Virtualize APIC accesses.** This control enables virtualization of memory-mapped accesses to the APIC (Section 29.4) by causing VM exits on accesses to a VMM-specified APIC-access page. Some of the other controls, if set, may cause some of these accesses to be emulated rather than causing VM exits.
- **Virtualize x2APIC mode.** This control enables virtualization of MSR-based accesses to the APIC (Section 29.5).
- **APIC-register virtualization.** This control allows memory-mapped and MSR-based reads of most APIC registers (as enabled) by satisfying them from the virtual-APIC page. It directs memory-mapped writes to the APIC-access page to the virtual-APIC page, following them by VM exits for VMM emulation.
- **Process posted interrupts.** This control allows software to post virtual interrupts in a data structure and send a notification to another logical processor; upon receipt of the notification, the target processor will process the posted interrupts by copying them into the virtual-APIC page (Section 29.6).

"Virtualize APIC accesses", "virtualize x2APIC mode", "virtual-interrupt delivery", and "APIC-register virtualization" are all secondary processor-based VM-execution controls. If bit 31 of the primary processor-based VM-execution controls is 0, the processor operates as if these controls were all 0. See Section 24.6.2.

## 29.1 VIRTUAL APIC STATE

The **virtual-APIC page** is a 4-KByte region of memory that the processor uses to virtualize certain accesses to APIC registers and to manage virtual interrupts. The physical address of the virtual-APIC page is the **virtual-APIC address**, a 64-bit VM-execution control field in the VMCS (see Section 24.6.8).

Depending on the settings of certain VM-execution controls, the processor may virtualize certain fields on the virtual-APIC page with functionality analogous to that performed by the local APIC. Section 29.1.1 identifies and defines these fields. Section 29.1.2, Section 29.1.3, Section 29.1.4, and Section 29.1.5 detail the actions taken to virtualize updates to some of these fields.

### 29.1.1 Virtualized APIC Registers

Depending on the setting of certain VM-execution controls, a logical processor may virtualize certain accesses to APIC registers using the following fields on the virtual-APIC page:

- **Virtual task-priority register (VTPR):** the 32-bit field located at offset 080H on the virtual-APIC page.

---

1. In most cases, it is not necessary for a virtual-machine monitor (VMM) to inject virtual interrupts as part of VM entry.

- **Virtual processor-priority register (VPPR)**: the 32-bit field located at offset 0A0H on the virtual-APIC page.
- **Virtual end-of-interrupt register (VEOI)**: the 32-bit field located at offset 0B0H on the virtual-APIC page.
- **Virtual interrupt-service register (VISR)**: the 256-bit value comprising eight non-contiguous 32-bit fields at offsets 100H, 110H, 120H, 130H, 140H, 150H, 160H, and 170H on the virtual-APIC page. Bit  $x$  of the VISR is at bit position  $(x \& 1FH)$  at offset  $(100H | ((x \& E0H) \gg 1))$ . The processor uses only the low 4 bytes of each of the 16-byte fields at offsets 100H, 110H, 120H, 130H, 140H, 150H, 160H, and 170H.
- **Virtual interrupt-request register (VIRR)**: the 256-bit value comprising eight non-contiguous 32-bit fields at offsets 200H, 210H, 220H, 230H, 240H, 250H, 260H, and 270H on the virtual-APIC page. Bit  $x$  of the VIRR is at bit position  $(x \& 1FH)$  at offset  $(200H | ((x \& E0H) \gg 1))$ . The processor uses only the low 4 bytes of each of the 16-Byte fields at offsets 200H, 210H, 220H, 230H, 240H, 250H, 260H, and 270H.
- **Virtual interrupt-command register (VICR\_LO)**: the 32-bit field located at offset 300H on the virtual-APIC page
- **Virtual interrupt-command register (VICR\_HI)**: the 32-bit field located at offset 310H on the virtual-APIC page.

### 29.1.2 TPR Virtualization

The processor performs **TPR virtualization** in response to the following operations: (1) virtualization of the MOV to CR8 instruction; (2) virtualization of a write to offset 080H on the APIC-access page; and (3) virtualization of the WRMSR instruction with ECX = 808H. See Section 29.3, Section 29.4.3, and Section 29.5 for details of when TPR virtualization is performed.

The following pseudocode details the behavior of TPR virtualization:

```

IF "virtual-interrupt delivery" is 0
    THEN
        IF VTPR[7:4] < TPR threshold (see Section 24.6.8)
            THEN cause VM exit due to TPR below threshold;
        FI;
    ELSE
        perform PPR virtualization (see Section 29.1.3);
        evaluate pending virtual interrupts (see Section 29.2.1);
    FI;

```

Any VM exit caused by TPR virtualization is trap-like: the instruction causing TPR virtualization completes before the VM exit occurs (for example, the value of CS:RIP saved in the guest-state area of the VMCS references the next instruction).

### 29.1.3 PPR Virtualization

The processor performs **PPR virtualization** in response to the following operations: (1) VM entry; (2) TPR virtualization; and (3) EOI virtualization. See Section 26.3.2.5, Section 29.1.2, and Section 29.1.4 for details of when PPR virtualization is performed.

PPR virtualization uses the guest interrupt status (specifically, SVI; see Section 24.4.2) and VTPR. The following pseudocode details the behavior of PPR virtualization:

```

IF VTPR[7:4] ≥ SVI[7:4]
    THEN VPPR ← VTPR & FFH;
    ELSE VPPR ← SVI & FOH;
FI;

```

PPR virtualization always clears bytes 3:1 of VPPR.

PPR virtualization is caused only by TPR virtualization, EOI virtualization, and VM entry. Delivery of a virtual interrupt also modifies VPPR, but in a different way (see Section 29.2.2). No other operations modify VPPR, even if they modify SVI, VISR, or VTPR.

## 29.1.4 EOI Virtualization

The processor performs **EOI virtualization** in response to the following operations: (1) virtualization of a write to offset 0B0H on the APIC-access page; and (2) virtualization of the WRMSR instruction with ECX = 80BH. See Section 29.4.3 and Section 29.5 for details of when EOI virtualization is performed. EOI virtualization occurs only if the “virtual-interrupt delivery” VM-execution control is 1.

EOI virtualization uses and updates the guest interrupt status (specifically, SVI; see Section 24.4.2). The following pseudocode details the behavior of EOI virtualization:

```

Vector ← SVI;
VISR[Vector] ← 0; (see Section 29.1.1 for definition of VISR)
IF any bits set in VISR
    THEN SVI ← highest index of bit set in VISR
    ELSE SVI ← 0;
FI;
perform PPR virtualiation (see Section 29.1.3);
IF EOI_exit_bitmap[Vector] = 1 (see Section 24.6.8 for definition of EOI_exit_bitmap)
    THEN cause EOI-induced VM exit with Vector as exit qualification;
    ELSE evaluate pending virtual interrupts; (see Section 29.2.1)
FI;

```

Any VM exit caused by EOI virtualization is trap-like: the instruction causing EOI virtualization completes before the VM exit occurs (for example, the value of CS:RIP saved in the guest-state area of the VMCS references the next instruction).

## 29.1.5 Self-IPI Virtualization

The processor performs **self-IPI virtualization** in response to the following operations: (1) virtualization of a write to offset 300H on the APIC-access page; and (2) virtualization of the WRMSR instruction with ECX = 83FH. See Section 29.4.3 and Section 29.5 for details of when self-IPI virtualization is performed. Self-IPI virtualization occurs only if the “virtual-interrupt delivery” VM-execution control is 1.

Each operation that leads to self-IPI virtualization provides an 8-bit vector (see Section 29.4.3 and Section 29.5). Self-IPI virtualization updates the guest interrupt status (specifically, RVI; see Section 24.4.2). The following pseudocode details the behavior of self-IPI virtualization:

```

VIRR[Vector] ← 1; (see Section 29.1.1 for definition of VIRR)
RVI ← max[RVI,Vector];
evaluate pending virtual interrupts; (see Section 29.2.1)

```

## 29.2 EVALUATION AND DELIVERY OF VIRTUAL INTERRUPTS

If the “virtual-interrupt delivery” VM-execution control is 1, certain actions in VMX non-root operation or during VM entry cause the processor to evaluate and deliver virtual interrupts.

Evaluation of virtual interrupts is triggered by certain actions change the state of the virtual-APIC page and is described in Section 29.2.1. This evaluation may result in recognition of a virtual interrupt. Once a virtual interrupt is recognized, the processor may deliver it within VMX non-root operation without a VM exit. Virtual-interrupt delivery is described in Section 29.2.2.

### 29.2.1 Evaluation of Pending Virtual Interrupts

If the “virtual-interrupt delivery” VM-execution control is 1, certain actions cause a logical processor to **evaluate pending virtual interrupts**.

The following actions cause the evaluation of pending virtual interrupts: VM entry; TPR virtualization; EOI virtualization; self-IPI virtualization; and posted-interrupt processing. See Section 26.3.2.5, Section 29.1.2, Section



29.1.4, Section 29.1.5, and Section 29.6 for details of when evaluation of pending virtual interrupts is performed. No other operations cause the evaluation of pending virtual interrupts, even if they modify RVI or VPPR.

Evaluation of pending virtual interrupts uses the guest interrupt status (specifically, RVI; see Section 24.4.2). The following pseudocode details the evaluation of pending virtual interrupts:

```

IF "interrupt-window exiting" is 0 AND
RVI[7:4] > VPPR[7:4] (see Section 29.1.1 for definition of VPPR)
    THEN recognize a pending virtual interrupt;
ELSE
    do not recognize a pending virtual interrupt;
FI;

```

Once recognized, a virtual interrupt may be delivered in VMX non-root operation; see Section 29.2.2.

Evaluation of pending virtual interrupts is caused only by VM entry, TPR virtualization, EOI virtualization, self-IPI virtualization, and posted-interrupt processing. No other operations do so, even if they modify RVI or VPPR. The logical processor ceases recognition of a pending virtual interrupt following the delivery of a virtual interrupt.

## 29.2.2 Virtual-Interrupt Delivery

If a virtual interrupt has been recognized (see Section 29.2.1), it is delivered at an instruction boundary when the following conditions all hold: (1) RFLAGS.IF = 1; (2) there is no blocking by STI; (3) there is no blocking by MOV SS or by POP SS; and (4) the "interrupt-window exiting" VM-execution control is 0.

Virtual-interrupt delivery has the same priority as that of VM exits due to the 1-setting of the "interrupt-window exiting" VM-execution control.<sup>2</sup> Thus, non-maskable interrupts (NMIs) and higher priority events take priority over delivery of a virtual interrupt; delivery of a virtual interrupt takes priority over external interrupts and lower priority events.

Virtual-interrupt delivery wakes a logical processor from the same inactive activity states as would an external interrupt. Specifically, it wakes a logical processor from the states entered using the HLT and MWAIT instructions. It does not wake a logical processor in the shutdown state or in the wait-for-SIPI state.

Virtual-interrupt delivery updates the guest interrupt status (both RVI and SVI; see Section 24.4.2) and delivers an event within VMX non-root operation without a VM exit. The following pseudocode details the behavior of virtual-interrupt delivery (see Section 29.1.1 for definition of VISR, VIRR, and VPPR):

```

Vector ← RVI;
VISR[Vector] ← 1;
SVI ← Vector;
VPPR ← Vector & FOH;
VIRR[Vector] ← 0;
IF any bits set in VIRR
    THEN RVI ← highest index of bit set in VIRR
    ELSE RVI ← 0;
FI;
deliver interrupt with Vector through IDT;
cease recognition of any pending virtual interrupt;

```

If a logical processor is in enclave mode, an Asynchronous Enclave Exit (AEX) occurs before delivery of a virtual interrupt (see Chapter 40, "Enclave Exiting Events").

---

2. A logical processor never recognizes or delivers a virtual interrupt if the "interrupt-window exiting" VM-execution control is 1. Because of this, the relative priority of virtual-interrupt delivery and VM exits due to the 1-setting of that control is not defined.

## 29.3 VIRTUALIZING CR8-BASED TPR ACCESSES

In 64-bit mode, software can access the local APIC's task-priority register (TPR) through CR8. Specifically, software uses the MOV from CR8 and MOV to CR8 instructions (see Section 10.8.6, "Task Priority in IA-32e Mode"). This section describes how these accesses can be virtualized.

A virtual-machine monitor can virtualize these CR8-based APIC accesses by setting the "CR8-load exiting" and "CR8-store exiting" VM-execution controls, ensuring that the accesses cause VM exits (see Section 25.1.3). Alternatively, there are methods for virtualizing some CR8-based APIC accesses without VM exits.

Normally, an execution of MOV from CR8 or MOV to CR8 that does not fault or cause a VM exit accesses the APIC's TPR. However, such an execution are treated specially if the "use TPR shadow" VM-execution control is 1. The following items provide details:

- **MOV from CR8.** The instruction loads bits 3:0 of its destination operand with bits 7:4 of VTPR (see Section 29.1.1). Bits 63:4 of the destination operand are cleared.
- **MOV to CR8.** The instruction stores bits 3:0 of its source operand into bits 7:4 of VTPR; the remainder of VTPR (bits 3:0 and bits 31:8) are cleared. Following this, the processor performs TPR virtualization (see Section 29.1.2).

## 29.4 VIRTUALIZING MEMORY-MAPPED APIC ACCESSES

When the local APIC is in xAPIC mode, software accesses the local APIC's control registers using a memory-mapped interface. Specifically, software uses linear addresses that translate to physical addresses on page frame indicated by the base address in the IA32\_APIC\_BASE MSR (see Section 10.4.4, "Local APIC Status and Location"). This section describes how these accesses can be virtualized.

A virtual-machine monitor (VMM) can virtualize these memory-mapped APIC accesses by ensuring that any access to a linear address that would access the local APIC instead causes a VM exit. This could be done using paging or the extended page-table mechanism (EPT). Another way is by using the 1-setting of the "virtualize APIC accesses" VM-execution control.

If the "virtualize APIC accesses" VM-execution control is 1, the logical processor treats specially memory accesses using linear addresses that translate to physical addresses in the 4-KByte **APIC-access page**.<sup>3</sup> (The APIC-access page is identified by the **APIC-access address**, a field in the VMCS; see Section 24.6.8.)

In general, an access to the APIC-access page causes an **APIC-access VM exit**. APIC-access VM exits provide a VMM with information about the access causing the VM exit. Section 29.4.1 discusses the priority of APIC-access VM exits.

Certain VM-execution controls enable the processor to virtualize certain accesses to the APIC-access page without a VM exit. In general, this virtualization causes these accesses to be made to the virtual-APIC page instead of the APIC-access page.

### NOTES

Unless stated otherwise, this section characterizes only linear accesses to the APIC-access page; an access to the APIC-access page is a linear access if (1) it results from a memory access using a linear address; and (2) the access's physical address is the translation of that linear address. Section 29.4.6 discusses accesses to the APIC-access page that are not linear accesses.

The distinction between the APIC-access page and the virtual-APIC page allows a VMM to share paging structures or EPT paging structures among the virtual processors of a virtual machine (the shared paging structures referencing the same APIC-access address, which appears in the VMCS of

---

3. Even when addresses are translated using EPT (see Section 28.2), the determination of whether an APIC-access VM exit occurs depends on an access's physical address, not its guest-physical address. Even when CR0.PG = 0, ordinary memory accesses by software use linear addresses; the fact that CR0.PG = 0 means only that the identity translation is used to convert linear addresses to physical (or guest-physical) addresses.

all the virtual processors) while giving each virtual processor its own virtual APIC (the VMCS of each virtual processor will have a unique virtual-APIC address).

Section 29.4.2 discusses when and how the processor may virtualize read accesses from the APIC-access page. Section 29.4.3 does the same for write accesses. When virtualizing a write to the APIC-access page, the processor typically takes actions in addition to passing the write through to the virtual-APIC page.

The discussion in those sections uses the concept of an **operation** within which these memory accesses may occur. For those discussions, an “operation” can be an iteration of a REP-prefixed string instruction, an execution of any other instruction, or delivery of an event through the IDT.

The 1-setting of the “virtualize APIC accesses” VM-execution control may also affect accesses to the APIC-access page that do not result directly from linear addresses. This is discussed in Section 29.4.6.

Special treatment may apply to Intel SGX instructions or if the logical processor is in enclave mode. See Section 42.5.3 for details.

### 29.4.1 Priority of APIC-Access VM Exits

The following items specify the priority of APIC-access VM exits relative to other events.

- The priority of an APIC-access VM exit due to a memory access is below that of any page fault or EPT violation that that access may incur. That is, an access does not cause an APIC-access VM exit if it would cause a page fault or an EPT violation.
- A memory access does not cause an APIC-access VM exit until after the accessed flags are set in the paging structures (including EPT paging structures, if enabled).
- A write access does not cause an APIC-access VM exit until after the dirty flags are set in the appropriate paging structure and EPT paging structure (if enabled).
- With respect to all other events, any APIC-access VM exit due to a memory access has the same priority as any page fault or EPT violation that the access could cause. (This item applies to other events that the access may generate as well as events that may be generated by other accesses by the same operation.)

These principles imply, among other things, that an APIC-access VM exit may occur during the execution of a repeated string instruction (including INS and OUTS). Suppose, for example, that the first  $n$  iterations ( $n$  may be 0) of such an instruction do not access the APIC-access page and that the next iteration does access that page. As a result, the first  $n$  iterations may complete and be followed by an APIC-access VM exit. The instruction pointer saved in the VMCS references the repeated string instruction and the values of the general-purpose registers reflect the completion of  $n$  iterations.

### 29.4.2 Virtualizing Reads from the APIC-Access Page

A read access from the APIC-access page causes an APIC-access VM exit if any of the following are true:

- The “use TPR shadow” VM-execution control is 0.
- The access is for an instruction fetch.
- The access is more than 32 bits in size.
- The access is part of an operation for which the processor has already virtualized a write to the APIC-access page.
- The access is not entirely contained within the low 4 bytes of a naturally aligned 16-byte region. That is, bits 3:2 of the access’s address are 0, and the same is true of the address of the highest byte accessed.

If none of the above are true, whether a read access is virtualized depends on the setting of the “APIC-register virtualization” VM-execution control:

- If “APIC-register virtualization” is 0, a read access is virtualized if its page offset is 080H (task priority); otherwise, the access causes an APIC-access VM exit.
- If “APIC-register virtualization” is 1, a read access is virtualized if it is entirely within one of the following ranges of offsets:

- 020H–023H (local APIC ID);
- 030H–033H (local APIC version);
- 080H–083H (task priority);
- 0B0H–0B3H (end of interrupt);
- 0D0H–0D3H (logical destination);
- 0E0H–0E3H (destination format);
- 0F0H–0F3H (spurious-interrupt vector);
- 100H–103H, 110H–113H, 120H–123H, 130H–133H, 140H–143H, 150H–153H, 160H–163H, or 170H–173H (in-service);
- 180H–183H, 190H–193H, 1A0H–1A3H, 1B0H–1B3H, 1C0H–1C3H, 1D0H–1D3H, 1E0H–1E3H, or 1F0H–1F3H (trigger mode);
- 200H–203H, 210H–213H, 220H–223H, 230H–233H, 240H–243H, 250H–253H, 260H–263H, or 270H–273H (interrupt request);
- 280H–283H (error status);
- 300H–303H or 310H–313H (interrupt command);
- 320H–323H, 330H–333H, 340H–343H, 350H–353H, 360H–363H, or 370H–373H (LVT entries);
- 380H–383H (initial count); or
- 3E0H–3E3H (divide configuration).

In all other cases, the access causes an APIC-access VM exit.

A read access from the APIC-access page that is virtualized returns data from the corresponding page offset on the virtual-APIC page.<sup>4</sup>

### 29.4.3 Virtualizing Writes to the APIC-Access Page

Whether a write access to the APIC-access page is virtualized depends on the settings of the VM-execution controls and the page offset of the access. Section 29.4.3.1 details when APIC-write virtualization occurs.

Unlike reads, writes to the local APIC have side effects; because of this, virtualization of writes to the APIC-access page may require emulation specific to the access's page offset (which identifies the APIC register being accessed). Section 29.4.3.2 describes this **APIC-write emulation**.

For some page offsets, it is necessary for software to complete the virtualization after a write completes. In these cases, the processor causes an **APIC-write VM exit** to invoke VMM software. Section 29.4.3.3 discusses APIC-write VM exits.

#### 29.4.3.1 Determining Whether a Write Access is Virtualized

A write access to the APIC-access page causes an APIC-access VM exit if any of the following are true:

- The "use TPR shadow" VM-execution control is 0.
- The access is more than 32 bits in size.
- The access is part of an operation for which the processor has already virtualized a write (with a different page offset or a different size) to the APIC-access page.
- The access is not entirely contained within the low 4 bytes of a naturally aligned 16-byte region. That is, bits 3:2 of the access's address are 0, and the same is true of the address of the highest byte accessed.

If none of the above are true, whether a write access is virtualized depends on the settings of the "APIC-register virtualization" and "virtual-interrupt delivery" VM-execution controls:

---

4. The memory type used for accesses that read from the virtual-APIC page is reported in bits 53:50 of the IA32\_VMX\_BASIC MSR (see Appendix A.1).

- If the “APIC-register virtualization” and “virtual-interrupt delivery” VM-execution controls are both 0, a write access is virtualized if its page offset is 080H; otherwise, the access causes an APIC-access VM exit.
- If the “APIC-register virtualization” VM-execution control is 0 and the “virtual-interrupt delivery” VM-execution control is 1, a write access is virtualized if its page offset is 080H (task priority), 0B0H (end of interrupt), and 300H (interrupt command — low); otherwise, the access causes an APIC-access VM exit.
- If “APIC-register virtualization is 1, a write access is virtualized if it is entirely within one the following ranges of offsets:
  - 020H–023H (local APIC ID);
  - 080H–083H (task priority);
  - 0B0H–0B3H (end of interrupt);
  - 0D0H–0D3H (logical destination);
  - 0E0H–0E3H (destination format);
  - 0F0H–0F3H (spurious-interrupt vector);
  - 280H–283H (error status);
  - 300H–303H or 310H–313H (interrupt command);
  - 320H–323H, 330H–333H, 340H–343H, 350H–353H, 360H–363H, or 370H–373H (LVT entries);
  - 380H–383H (initial count); or
  - 3E0H–3E3H (divide configuration).

In all other cases, the access causes an APIC-access VM exit.

The processor virtualizes a write access to the APIC-access page by writing data to the corresponding page offset on the virtual-APIC page.<sup>5</sup> Following this, the processor performs certain actions after completion of the operation of which the access was a part.<sup>6</sup> APIC-write emulation is described in Section 29.4.3.2.

### 29.4.3.2 APIC-Write Emulation

If the processor virtualizes a write access to the APIC-access page, it performs additional actions after completion of an operation of which the access was a part. These actions are called **APIC-write emulation**.

The details of APIC-write emulation depend upon the page offset of the virtualized write access:<sup>7</sup>

- 080H (task priority). The processor clears bytes 3:1 of VTPR and then causes TPR virtualization (Section 29.1.2).
- 0B0H (end of interrupt). If the “virtual-interrupt delivery” VM-execution control is 1, the processor clears VEOI and then causes EOI virtualization (Section 29.1.4); otherwise, the processor causes an APIC-write VM exit (Section 29.4.3.3).
- 300H (interrupt command — low). If the “virtual-interrupt delivery” VM-execution control is 1, the processor checks the value of VICR\_LO to determine whether the following are all true:
  - Reserved bits (31:20, 17:16, 13) and bit 12 (delivery status) are all 0.
  - Bits 19:18 (destination shorthand) are 01B (self).
  - Bit 15 (trigger mode) is 0 (edge).
  - Bits 10:8 (delivery mode) are 000B (fixed).
  - Bits 7:4 (the upper half of the vector) are **not** 0000B.

---

5. The memory type used for accesses that write to the virtual-APIC page is reported in bits 53:50 of the IA32\_VMX\_BASIC MSR (see Appendix A.1).

6. Recall that, for the purposes of this discussion, an operation is an iteration of a REP-prefixed string instruction, an execution of any other instruction, or delivery of an event through the IDT.

7. For any operation, there can be only one page offset for which a write access was virtualized. This is because a write access is not virtualized if the processor has already virtualized a write access for the same operation with a different page offset.

If all of the items above are true, the processor performs self-IPI virtualization using the 8-bit vector in byte 0 of VICR\_LO (Section 29.1.5).

If the “virtual-interrupt delivery” VM-execution control is 0, or if any of the items above are false, the processor causes an APIC-write VM exit (Section 29.4.3.3).

- 310H–313H (interrupt command — high). The processor clears bytes 2:0 of VICR\_HI. No other virtualization or VM exit occurs.
- Any other page offset. The processor causes an APIC-write VM exit (Section 29.4.3.3).

APIC-write emulation takes priority over system-management interrupts (SMIs), INIT signals, and lower priority events. APIC-write emulation is not blocked if RFLAGS.IF = 0 or by the MOV SS, POP SS, or STI instructions.

If an operation causes a fault after a write access to the APIC-access page and before APIC-write emulation, and that fault is delivered without a VM exit, APIC-write emulation occurs after the fault is delivered and before the fault handler can execute. If an operation causes a VM exit (perhaps due to a fault) after a write access to the APIC-access page and before APIC-write emulation, the APIC-write emulation does not occur.

### 29.4.3.3 APIC-Write VM Exits

In certain cases, VMM software must be invoked to complete the virtualization of a write access to the APIC-access page. In this case, APIC-write emulation causes an **APIC-write VM exit**. (Section 29.4.3.2 details the cases that causes APIC-write VM exits.)

APIC-write VM exits are invoked by APIC-write emulation, and APIC-write emulation occurs after an operation that performs a write access to the APIC-access page. Because of this, every APIC-write VM exit is trap-like: it occurs after completion of the operation containing the write access that caused the VM exit (for example, the value of CS:RIP saved in the guest-state area of the VMCS references the next instruction).

The basic exit reason for an APIC-write VM exit is “APIC write.” The exit qualification is the page offset of the write access that led to the VM exit.

As noted in Section 29.5, execution of WRMSR with ECX = 83FH (self-IPI MSR) can lead to an APIC-write VM exit if the “virtual-interrupt delivery” VM-execution control is 1. The exit qualification for such an APIC-write VM exit is 3F0H.

## 29.4.4 Instruction-Specific Considerations

Certain instructions that use linear address may cause page faults even though they do not use those addresses to access memory. The APIC-virtualization features may affect these instructions as well:

- **CLFLUSH, CLFLUSHOPT**. With regard to faulting, the processor operates as if each of these instructions reads from the linear address in its source operand. If that address translates to one on the APIC-access page, the instruction may cause an APIC-access VM exit. If it does not, it will flush the corresponding cache line on the virtual-APIC page instead of the APIC-access page.
- **ENTER**. With regard to faulting, the processor operates if ENTER writes to the byte referenced by the final value of the stack pointer (even though it does not if its size operand is non-zero). If that value translates to an address on the APIC-access page, the instruction may cause an APIC-access VM exit. If it does not, it will cause the APIC-write emulation appropriate to the address’s page offset.
- **MASKMOVQ and MAKSMOVDQU**. Even if the instruction’s mask is zero, the processor may operate with regard to faulting as if MASKMOVQ or MASKMOVDQU writes to memory (the behavior is implementation-specific). In such a situation, an APIC-access VM exit may occur.
- **MONITOR**. With regard to faulting, the processor operates as if MONITOR reads from the effective address in RAX. If the resulting linear address translates to one on the APIC-access page, the instruction may cause an APIC-access VM exit.<sup>8</sup> If it does not, it will monitor the corresponding address on the virtual-APIC page instead of the APIC-access page.

8. This chapter uses the notation RAX, RIP, RSP, RFLAGS, etc. for processor registers because most processors that support VMX operation also support Intel 64 architecture. For IA-32 processors, this notation refers to the 32-bit forms of those registers (EAX, EIP, ESP, EFLAGS, etc.). In a few places, notation such as EAX is used to refer specifically to lower 32 bits of the indicated register.



- **PREFETCH.** An execution of the PREFETCH instruction that would result in an access to the APIC-access page does not cause an APIC-access VM exit. Such an access may prefetch data; if so, it is from the corresponding address on the virtual-APIC page.

Virtualization of accesses to the APIC-access page is principally intended for basic instructions such as AND, MOV, OR, TEST, XCHG, and XOR. Use of an instruction that normally operates on floating-point, SSE, AVX, or AVX-512 registers may cause an APIC-access VM exit unconditionally regardless of the page offset it accesses on the APIC-access page.

## 29.4.5 Issues Pertaining to Page Size and TLB Management

The 1-setting of the “virtualize APIC accesses” VM-execution is guaranteed to apply only if translations to the APIC-access address use a 4-KByte page. The following items provide details:

- If EPT is not in use, any linear address that translates to an address on the APIC-access page should use a 4-KByte page. Any access to a linear address that translates to the APIC-access page using a larger page may operate as if the “virtualize APIC accesses” VM-execution control were 0.
- If EPT is in use, any guest-physical address that translates to an address on the APIC-access page should use a 4-KByte page. Any access to a linear address that translates to a guest-physical address that in turn translates to the APIC-access page using a larger page may operate as if the “virtualize APIC accesses” VM-execution control were 0. (This is true also for guest-physical accesses to the APIC-access page; see Section 29.4.6.1.)

In addition, software should perform appropriate TLB invalidation when making changes that may affect APIC-virtualization. The specifics depend on whether VPIDs or EPT is being used:

- **VPIDs being used but EPT not being used.** Suppose that there is a VPID that has been used before and that software has since made either of the following changes: (1) set the “virtualize APIC accesses” VM-execution control when it had previously been 0; or (2) changed the paging structures so that some linear address translates to the APIC-access address when it previously did not. In that case, software should execute INVVPID (see “INVVPID— Invalidate Translations Based on VPID” in Section 30.3) before performing on the same logical processor and with the same VPID.<sup>9</sup>
- **EPT being used.** Suppose that there is an EPTP value that has been used before and that software has since made either of the following changes: (1) set the “virtualize APIC accesses” VM-execution control when it had previously been 0; or (2) changed the EPT paging structures so that some guest-physical address translates to the APIC-access address when it previously did not. In that case, software should execute INVEPT (see “INVEPT— Invalidate Translations Derived from EPT” in Section 30.3) before performing on the same logical processor and with the same EPTP value.<sup>10</sup>
- **Neither VPIDs nor EPT being used.** No invalidation is required.

Failure to perform the appropriate TLB invalidation may result in the logical processor operating as if the “virtualize APIC accesses” VM-execution control were 0 in responses to accesses to the affected address. (No invalidation is necessary if neither VPIDs nor EPT is being used.)

## 29.4.6 APIC Accesses Not Directly Resulting From Linear Addresses

Section 29.4 has described the treatment of accesses that use linear addresses that translate to addresses on the APIC-access page. This section considers memory accesses that do not result directly from linear addresses.

- An access is called a **guest-physical access** if (1) CR0.PG = 1;<sup>11</sup> (2) the “enable EPT” VM-execution control is 1;<sup>12</sup> (3) the access’s physical address is the result of an EPT translation; and (4) either (a) the access was not generated by a linear address; or (b) the access’s guest-physical address is not the translation of the

9. INVVPID should use either (1) the all-contexts INVVPID type; (2) the single-context INVVPID type with the VPID in the INVVPID descriptor; or (3) the individual-address INVVPID type with the linear address and the VPID in the INVVPID descriptor.

10. INVEPT should use either (1) the global INVEPT type; or (2) the single-context INVEPT type with the EPTP value in the INVEPT descriptor.

11. If the capability MSR IA32\_VMX\_CR0\_FIXED0 reports that CR0.PG must be 1 in VMX operation, CR0.PG must be 1 unless the “unrestricted guest” VM-execution control and bit 31 of the primary processor-based VM-execution controls are both 1.

access's linear address. Section 29.4.6.1 discusses the treatment of guest-physical accesses to the APIC-access page.

- An access is called a **physical access** if (1) either (a) the "enable EPT" VM-execution control is 0; or (b) the access's physical address is not the result of a translation through the EPT paging structures; and (2) either (a) the access is not generated by a linear address; or (b) the access's physical address is not the translation of its linear address. Section 29.4.6.2 discusses the treatment of physical accesses to the APIC-access page.

### 29.4.6.1 Guest-Physical Accesses to the APIC-Access Page

Guest-physical accesses include the following when guest-physical addresses are being translated using EPT:

- Reads from the guest paging structures when translating a linear address (such an access uses a guest-physical address that is not the translation of that linear address).
- Loads of the page-directory-pointer-table entries by MOV to CR when the logical processor is using (or that causes the logical processor to use) PAE paging (see Section 4.4).
- Updates to the accessed and dirty flags in the guest paging structures when using a linear address (such an access uses a guest-physical address that is not the translation of that linear address).

Every guest-physical access to an address on the APIC-access page causes an APIC-access VM exit. Such accesses are never virtualized regardless of the page offset.

The following items specify the priority relative to other events of APIC-access VM exits caused by guest-physical accesses to the APIC-access page.

- The priority of an APIC-access VM exit caused by a guest-physical access to memory is below that of any EPT violation that that access may incur. That is, a guest-physical access does not cause an APIC-access VM exit if it would cause an EPT violation.
- With respect to all other events, any APIC-access VM exit caused by a guest-physical access has the same priority as any EPT violation that the guest-physical access could cause.

### 29.4.6.2 Physical Accesses to the APIC-Access Page

Physical accesses include the following:

- If the "enable EPT" VM-execution control is 0:
  - Reads from the paging structures when translating a linear address.
  - Loads of the page-directory-pointer-table entries by MOV to CR when the logical processor is using (or that causes the logical processor to use) PAE paging (see Section 4.4).
  - Updates to the accessed and dirty flags in the paging structures.
- If the "enable EPT" VM-execution control is 1, accesses to the EPT paging structures (including updates to the accessed and dirty flags for EPT).
- Any of the following accesses made by the processor to support VMX non-root operation:
  - Accesses to the VMCS region.
  - Accesses to data structures referenced (directly or indirectly) by physical addresses in VM-execution control fields in the VMCS. These include the I/O bitmaps, the MSR bitmaps, and the virtual-APIC page.
- Accesses that effect transitions into and out of SMM.<sup>13</sup> These include the following:
  - Accesses to SMRAM during SMI delivery and during execution of RSM.
  - Accesses during SMM VM exits (including accesses to MSEG) and during VM entries that return from SMM.

12. "Enable EPT" is a secondary processor-based VM-execution control. If bit 31 of the primary processor-based VM-execution controls is 0, VMX non-root operation functions as if the "enable EPT" VM-execution control were 0. See Section 24.6.2.

13. Technically, these accesses do not occur in VMX non-root operation. They are included here for clarity.



A physical access to the APIC-access page may or may not cause an APIC-access VM exit. If it does not cause an APIC-access VM exit, it may access the APIC-access page or the virtual-APIC page. Physical write accesses to the APIC-access page may or may not cause APIC-write emulation or APIC-write VM exits.

The priority of an APIC-access VM exit caused by physical access is not defined relative to other events that the access may cause.

It is recommended that software not set the APIC-access address to any of the addresses used by physical memory accesses (identified above). For example, it should not set the APIC-access address to the physical address of any of the active paging structures if the “enable EPT” VM-execution control is 0.

## 29.5 VIRTUALIZING MSR-BASED APIC ACCESSES

When the local APIC is in x2APIC mode, software accesses the local APIC’s control registers using the MSR interface. Specifically, software uses the RDMSR and WRMSR instructions, setting ECX (identifying the MSR being accessed) to values in the range 800H–8FFH (see Section 10.12, “Extended XAPIC (x2APIC)”). This section describes how these accesses can be virtualized.

A virtual-machine monitor can virtualize these MSR-based APIC accesses by configuring the MSR bitmaps (see Section 24.6.9) to ensure that the accesses cause VM exits (see Section 25.1.3). Alternatively, there are methods for virtualizing some MSR-based APIC accesses without VM exits.

Normally, an execution of RDMSR or WRMSR that does not fault or cause a VM exit accesses the MSR indicated in ECX. However, such an execution treats some values of ECX in the range 800H–8FFH specially if the “virtualize x2APIC mode” VM-execution control is 1. The following items provide details:

- **RDMSR.** The instruction’s behavior depends on the setting of the “APIC-register virtualization” VM-execution control.
  - If the “APIC-register virtualization” VM-execution control is 0, behavior depends upon the value of ECX.
    - If ECX contains 808H (indicating the TPR MSR), the instruction reads the 8 bytes from offset 080H on the virtual-APIC page (VTPR and the 4 bytes above it) into EDX:EAX. This occurs even if the local APIC is not in x2APIC mode (no general-protection fault occurs because the local APIC is not x2APIC mode).
    - If ECX contains any other value in the range 800H–8FFH, the instruction operates normally. If the local APIC is in x2APIC mode and ECX indicates a readable APIC register, EDX and EAX are loaded with the value of that register. If the local APIC is not in x2APIC mode or ECX does not indicate a readable APIC register, a general-protection fault occurs.
  - If “APIC-register virtualization” is 1 and ECX contains a value in the range 800H–8FFH, the instruction reads the 8 bytes from offset X on the virtual-APIC page into EDX:EAX, where  $X = (ECX \& FFH) \ll 4$ . This occurs even if the local APIC is not in x2APIC mode (no general-protection fault occurs because the local APIC is not in x2APIC mode).
- **WRMSR.** The instruction’s behavior depends on the value of ECX and the setting of the “virtual-interrupt delivery” VM-execution control.

Special processing applies in the following cases: (1) ECX contains 808H (indicating the TPR MSR); (2) ECX contains 80BH (indicating the EOI MSR) and the “virtual-interrupt delivery” VM-execution control is 1; and (3) ECX contains 83FH (indicating the self-IPI MSR) and the “virtual-interrupt delivery” VM-execution control is 1.

If special processing applies, no general-protection exception is produced due to the fact that the local APIC is in xAPIC mode. However, WRMSR does perform the normal reserved-bit checking:

- If ECX contains 808H or 83FH, a general-protection fault occurs if either EDX or EAX[31:8] is non-zero.
- If ECX contains 80BH, a general-protection fault occurs if either EDX or EAX is non-zero.

If there is no fault, WRMSR stores EDX:EAX at offset X on the virtual-APIC page, where  $X = (ECX \& FFH) \ll 4$ . Following this, the processor performs an operation depending on the value of ECX:

- If ECX contains 808H, the processor performs TPR virtualization (see Section 29.1.2).
- If ECX contains 80BH, the processor performs EOI virtualization (see Section 29.1.4).
- If ECX contains 83FH, the processor then checks the value of EAX[7:4] and proceeds as follows:

- If the value is non-zero, the logical processor performs self-IPI virtualization with the 8-bit vector in EAX[7:0] (see Section 29.1.5).
- If the value is zero, the logical processor causes an APIC-write VM exit as if there had been a write access to page offset 3F0H on the APIC-access page (see Section 29.4.3.3).

If special processing does not apply, the instruction operates normally. If the local APIC is in x2APIC mode and ECX indicates a writable APIC register, the value in EDX:EAX is written to that register. If the local APIC is not in x2APIC mode or ECX does not indicate a writable APIC register, a general-protection fault occurs.

## 29.6 POSTED-INTERRUPT PROCESSING

Posted-interrupt processing is a feature by which a processor processes the virtual interrupts by recording them as pending on the virtual-APIC page.

Posted-interrupt processing is enabled by setting the “process posted interrupts” VM-execution control. The processing is performed in response to the arrival of an interrupt with the **posted-interrupt notification vector**. In response to such an interrupt, the processor processes virtual interrupts recorded in a data structure called a **posted-interrupt descriptor**. The posted-interrupt notification vector and the address of the posted-interrupt descriptor are fields in the VMCS; see Section 24.6.8.

If the “process posted interrupts” VM-execution control is 1, a logical processor uses a 64-byte posted-interrupt descriptor located at the posted-interrupt descriptor address. The posted-interrupt descriptor has the following format:

**Table 29-1. Format of Posted-Interrupt Descriptor**

Bit Position(s)	Name	Description
255:0	Posted-interrupt requests	One bit for each interrupt vector. There is a posted-interrupt request for a vector if the corresponding bit is 1
256	Outstanding notification	If this bit is set, there is a notification outstanding for one or more posted interrupts in bits 255:0
511:257	Reserved for software and other agents	These bits may be used by software and by other agents in the system (e.g., chipset). The processor does not modify these bits.

The notation **PIR** (posted-interrupt requests) refers to the 256 posted-interrupt bits in the posted-interrupt descriptor.

Use of the posted-interrupt descriptor differs from that of other data structures that are referenced by pointers in a VMCS. There is a general requirement that software ensure that each such data structure is modified only when no logical processor with a current VMCS that references it is in VMX non-root operation. That requirement does not apply to the posted-interrupt descriptor. There is a requirement, however, that such modifications be done using locked read-modify-write instructions.

If the “external-interrupt exiting” VM-execution control is 1, any unmasked external interrupt causes a VM exit (see Section 25.2). If the “process posted interrupts” VM-execution control is also 1, this behavior is changed and the processor handles an external interrupt as follows:<sup>14</sup>

1. The local APIC is acknowledged; this provides the processor core with an interrupt vector, called here the **physical vector**.
2. If the physical vector equals the posted-interrupt notification vector, the logical processor continues to the next step. Otherwise, a VM exit occurs as it would normally due to an external interrupt; the vector is saved in the VM-exit interruption-information field.
3. The processor clears the outstanding-notification bit in the posted-interrupt descriptor. This is done atomically so as to leave the remainder of the descriptor unmodified (e.g., with a locked AND operation).

14. VM entry ensures that the “process posted interrupts” VM-execution control is 1 only if the “external-interrupt exiting” VM-execution control is also 1. See Section 26.2.1.1.

4. The processor writes zero to the EOI register in the local APIC; this dismisses the interrupt with the posted-interrupt notification vector from the local APIC.
5. The logical processor performs a logical-OR of PIR into VIRR and clears PIR. No other agent can read or write a PIR bit (or group of bits) between the time it is read (to determine what to OR into VIRR) and when it is cleared.
6. The logical processor sets RVI to be the maximum of the old value of RVI and the highest index of all bits that were set in PIR; if no bit was set in PIR, RVI is left unmodified.
7. The logical processor evaluates pending virtual interrupts as described in Section 29.2.1.

The logical processor performs the steps above in an uninterruptible manner. If step #7 leads to recognition of a virtual interrupt, the processor may deliver that interrupt immediately.

Steps #1 to #7 above occur when the interrupt controller delivers an unmasked external interrupt to the CPU core. The following items consider certain cases of interrupt delivery:

- Interrupt delivery can occur between iterations of a REP-prefixed instruction (after at least one iteration has completed but before all iterations have completed). If this occurs, the following items characterize processor state after posted-interrupt processing completes and before guest execution resumes:
  - RIP references the REP-prefixed instruction;
  - RCX, RSI, and RDI are updated to reflect the iterations completed; and
  - RFLAGS.RF = 1.
- Interrupt delivery can occur when the logical processor is in the active, HLT, or MWAIT states. If the logical processor had been in the active or MWAIT state before the arrival of the interrupt, it is in the active state following completion of step #7; if it had been in the HLT state, it returns to the HLT state after step #7 (if a pending virtual interrupt was recognized, the logical processor may immediately wake from the HLT state).
- Interrupt delivery can occur while the logical processor is in enclave mode. If the logical processor had been in enclave mode before the arrival of the interrupt, an Asynchronous Enclave Exit (AEX) may occur before the steps #1 to #7 (see Chapter 40, "Enclave Exiting Events"). If no AEX occurs before step #1 and a VM exit occurs at step #2, an AEX occurs before the VM exit is delivered.

## 18. Updates to Chapter 34, Volume 3C

Change bars show changes to Chapter 34 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C: System Programming Guide, Part 3*.

-----  
Changes include an update to the Enabling Dual-Monitor Treatment section; corrected typo regarding IA32\_SMM\_MONITOR\_CTL MSR format. Minor updates due to [Intel® 64 and IA-32 Architectures Software Developer's Manual, Volumes 2A, 2B, 2C & 2D](#) changes.

This chapter describes aspects of IA-64 and IA-32 architecture used in system management mode (SMM).

SMM provides an alternate operating environment that can be used to monitor and manage various system resources for more efficient energy usage, to control system hardware, and/or to run proprietary code. It was introduced into the IA-32 architecture in the Intel386 SL processor (a mobile specialized version of the Intel386 processor). It is also available in the Pentium M, Pentium 4, Intel Xeon, P6 family, and Pentium and Intel486 processors (beginning with the enhanced versions of the Intel486 SL and Intel486 processors).

### 34.1 SYSTEM MANAGEMENT MODE OVERVIEW

SMM is a special-purpose operating mode provided for handling system-wide functions like power management, system hardware control, or proprietary OEM-designed code. It is intended for use only by system firmware, not by applications software or general-purpose systems software. The main benefit of SMM is that it offers a distinct and easily isolated processor environment that operates transparently to the operating system or executive and software applications.

When SMM is invoked through a system management interrupt (SMI), the processor saves the current state of the processor (the processor's context), then switches to a separate operating environment defined by a new address space. The system management software executive (SMI handler) starts execution in that environment, and the critical code and data of the SMI handler reside in a physical memory region (SMRAM) within that address space. While in SMM, the processor executes SMI handler code to perform operations such as powering down unused disk drives or monitors, executing proprietary code, or placing the whole system in a suspended state. When the SMI handler has completed its operations, it executes a resume (RSM) instruction. This instruction causes the processor to reload the saved context of the processor, switch back to protected or real mode, and resume executing the interrupted application or operating-system program or task.

The following SMM mechanisms make it transparent to applications programs and operating systems:

- The only way to enter SMM is by means of an SMI.
- The processor executes SMM code in a separate address space that can be made inaccessible from the other operating modes.
- Upon entering SMM, the processor saves the context of the interrupted program or task.
- All interrupts normally handled by the operating system are disabled upon entry into SMM.
- The RSM instruction can be executed only in SMM.

Section 34.3 describes transitions into and out of SMM. The execution environment after entering SMM is in real-address mode with paging disabled ( $CR0.PE = CR0.PG = 0$ ). In this initial execution environment, the SMI handler can address up to 4 GBytes of memory and can execute all I/O and system instructions. Section 34.5 describes in detail the initial SMM execution environment for an SMI handler and operation within that environment. The SMI handler may subsequently switch to other operating modes while remaining in SMM.

#### NOTES

Software developers should be aware that, even if a logical processor was using the physical-address extension (PAE) mechanism (introduced in the P6 family processors) or was in IA-32e mode before an SMI, this will not be the case after the SMI is delivered. This is because delivery of an SMI disables paging (see Table 34-4). (This does not apply if the dual-monitor treatment of SMIs and SMM is active; see Section 34.15.)

#### 34.1.1 System Management Mode and VMX Operation

Traditionally, SMM services system management interrupts and then resumes program execution (back to the software stack consisting of executive and application software; see Section 34.2 through Section 34.13).

A virtual machine monitor (VMM) using VMX can act as a host to multiple virtual machines and each virtual machine can support its own software stack of executive and application software. On processors that support VMX, virtual-machine extensions may use system-management interrupts (SMIs) and system-management mode (SMM) in one of two ways:

- **Default treatment.** System firmware handles SMIs. The processor saves architectural states and critical states relevant to VMX operation upon entering SMM. When the firmware completes servicing SMIs, it uses RSM to resume VMX operation.
- **Dual-monitor treatment.** Two VM monitors collaborate to control the servicing of SMIs: one VMM operates outside of SMM to provide basic virtualization in support for guests; the other VMM operates inside SMM (while in VMX operation) to support system-management functions. The former is referred to as **executive monitor**, the latter **SMM-transfer monitor (STM)**.<sup>1</sup>

The default treatment is described in Section 34.14, “Default Treatment of SMIs and SMM with VMX Operation and SMX Operation”. Dual-monitor treatment of SMM is described in Section 34.15, “Dual-Monitor Treatment of SMIs and SMM”.

## 34.2 SYSTEM MANAGEMENT INTERRUPT (SMI)

The only way to enter SMM is by signaling an SMI through the SMI# pin on the processor or through an SMI message received through the APIC bus. The SMI is a nonmaskable external interrupt that operates independently from the processor’s interrupt- and exception-handling mechanism and the local APIC. The SMI takes precedence over an NMI and a maskable interrupt. SMM is non-reentrant; that is, the SMI is disabled while the processor is in SMM.

### NOTES

In the Pentium 4, Intel Xeon, and P6 family processors, when a processor that is designated as an application processor during an MP initialization sequence is waiting for a startup IPI (SIPI), it is in a mode where SMIs are masked. However if a SMI is received while an application processor is in the wait for SIPI mode, the SMI will be pended. The processor then responds on receipt of a SIPI by immediately servicing the pended SMI and going into SMM before handling the SIPI.

An SMI may be blocked for one instruction following execution of STI, MOV to SS, or POP into SS.

## 34.3 SWITCHING BETWEEN SMM AND THE OTHER PROCESSOR OPERATING MODES

Figure 2-3 shows how the processor moves between SMM and the other processor operating modes (protected, real-address, and virtual-8086). Signaling an SMI while the processor is in real-address, protected, or virtual-8086 modes always causes the processor to switch to SMM. Upon execution of the RSM instruction, the processor always returns to the mode it was in when the SMI occurred.

### 34.3.1 Entering SMM

The processor always handles an SMI on an architecturally defined “interruptible” point in program execution (which is commonly at an IA-32 architecture instruction boundary). When the processor receives an SMI, it waits for all instructions to retire and for all stores to complete. The processor then saves its current context in SMRAM (see Section 34.4), enters SMM, and begins to execute the SMI handler.

Upon entering SMM, the processor signals external hardware that SMI handling has begun. The signaling mechanism used is implementation dependent. For the P6 family processors, an SMI acknowledge transaction is gener-

1. The dual-monitor treatment may not be supported by all processors. Software should consult the VMX capability MSR IA32\_VMX\_BASIC (see Appendix A.1) to determine whether it is supported.

ated on the system bus and the multiplexed status signal EXF4 is asserted each time a bus transaction is generated while the processor is in SMM. For the Pentium and Intel486 processors, the SMIACK# pin is asserted.

An SMI has a greater priority than debug exceptions and external interrupts. Thus, if an NMI, maskable hardware interrupt, or a debug exception occurs at an instruction boundary along with an SMI, only the SMI is handled. Subsequent SMI requests are not acknowledged while the processor is in SMM. The first SMI interrupt request that occurs while the processor is in SMM (that is, after SMM has been acknowledged to external hardware) is latched and serviced when the processor exits SMM with the RSM instruction. The processor will latch only one SMI while in SMM.

See Section 34.5 for a detailed description of the execution environment when in SMM.

### 34.3.2 Exiting From SMM

The only way to exit SMM is to execute the RSM instruction. The RSM instruction is only available to the SMI handler; if the processor is not in SMM, attempts to execute the RSM instruction result in an invalid-opcode exception (#UD) being generated.

The RSM instruction restores the processor's context by loading the state save image from SMRAM back into the processor's registers. The processor then returns an SMIACK transaction on the system bus and returns program control back to the interrupted program.

Upon successful completion of the RSM instruction, the processor signals external hardware that SMM has been exited. For the P6 family processors, an SMI acknowledge transaction is generated on the system bus and the multiplexed status signal EXF4 is no longer generated on bus cycles. For the Pentium and Intel486 processors, the SMIACK# pin is deserted.

If the processor detects invalid state information saved in the SMRAM, it enters the shutdown state and generates a special bus cycle to indicate it has entered shutdown state. Shutdown happens only in the following situations:

- A reserved bit in control register CR4 is set to 1 on a write to CR4. This error should not happen unless SMI handler code modifies reserved areas of the SMRAM saved state map (see Section 34.4.1). CR4 is saved in the state map in a reserved location and cannot be read or modified in its saved state.
- An illegal combination of bits is written to control register CR0, in particular PG set to 1 and PE set to 0, or NW set to 1 and CD set to 0.
- CR4.PCIDE would be set to 1 and IA32\_EFER.LMA to 0.
- (For the Pentium and Intel486 processors only.) If the address stored in the SMBASE register when an RSM instruction is executed is not aligned on a 32-KByte boundary. This restriction does not apply to the P6 family processors.

In the shutdown state, Intel processors stop executing instructions until a RESET#, INIT# or NMI# is asserted. While Pentium family processors recognize the SMI# signal in shutdown state, P6 family and Intel486 processors do not. Intel does not support using SMI# to recover from shutdown states for any processor family; the response of processors in this circumstance is not well defined. On Pentium 4 and later processors, shutdown will inhibit INTR and A20M but will not change any of the other inhibits. On these processors, NMIs will be inhibited if no action is taken in the SMI handler to uninhibit them (see Section 34.8).

If the processor is in the HALT state when the SMI is received, the processor handles the return from SMM slightly differently (see Section 34.10). Also, the SMBASE address can be changed on a return from SMM (see Section 34.11).

## 34.4 SMRAM

Upon entering SMM, the processor switches to a new address space. Because paging is disabled upon entering SMM, this initial address space maps all memory accesses to the low 4 GBytes of the processor's physical address space. The SMI handler's critical code and data reside in a memory region referred to as system-management RAM (SMRAM). The processor uses a pre-defined region within SMRAM to save the processor's pre-SMI context. SMRAM can also be used to store system management information (such as the system configuration and specific information about powered-down devices) and OEM-specific information.



The default SMRAM size is 64 KBytes beginning at a base physical address in physical memory called the SMBASE (see Figure 34-1). The SMBASE default value following a hardware reset is 30000H. The processor looks for the first instruction of the SMI handler at the address [SMBASE + 8000H]. It stores the processor's state in the area from [SMBASE + FE00H] to [SMBASE + FFFFH]. See Section 34.4.1 for a description of the mapping of the state save area.

The system logic is minimally required to decode the physical address range for the SMRAM from [SMBASE + 8000H] to [SMBASE + FFFFH]. A larger area can be decoded if needed. The size of this SMRAM can be between 32 KBytes and 4 GBytes.

The location of the SMRAM can be changed by changing the SMBASE value (see Section 34.11). It should be noted that all processors in a multiple-processor system are initialized with the same SMBASE value (30000H). Initialization software must sequentially place each processor in SMM and change its SMBASE so that it does not overlap those of other processors.

The actual physical location of the SMRAM can be in system memory or in a separate RAM memory. The processor generates an SMI acknowledge transaction (P6 family processors) or asserts the SMIACT# pin (Pentium and Intel486 processors) when the processor receives an SMI (see Section 34.3.1).

System logic can use the SMI acknowledge transaction or the assertion of the SMIACT# pin to decode accesses to the SMRAM and redirect them (if desired) to specific SMRAM memory. If a separate RAM memory is used for SMRAM, system logic should provide a programmable method of mapping the SMRAM into system memory space when the processor is not in SMM. This mechanism will enable start-up procedures to initialize the SMRAM space (that is, load the SMI handler) before executing the SMI handler during SMM.

### 34.4.1 SMRAM State Save Map

When an IA-32 processor that does not support Intel 64 architecture initially enters SMM, it writes its state to the state save area of the SMRAM. The state save area begins at [SMBASE + 8000H + 7FFFH] and extends down to [SMBASE + 8000H + 7E00H]. Table 34-1 shows the state save map. The offset in column 1 is relative to the SMBASE value plus 8000H. Reserved spaces should not be used by software.

Some of the registers in the SMRAM state save area (marked YES in column 3) may be read and changed by the SMI handler, with the changed values restored to the processor registers by the RSM instruction. Some register images are read-only, and must not be modified (modifying these registers will result in unpredictable behavior). An SMI handler should not rely on any values stored in an area that is marked as reserved.

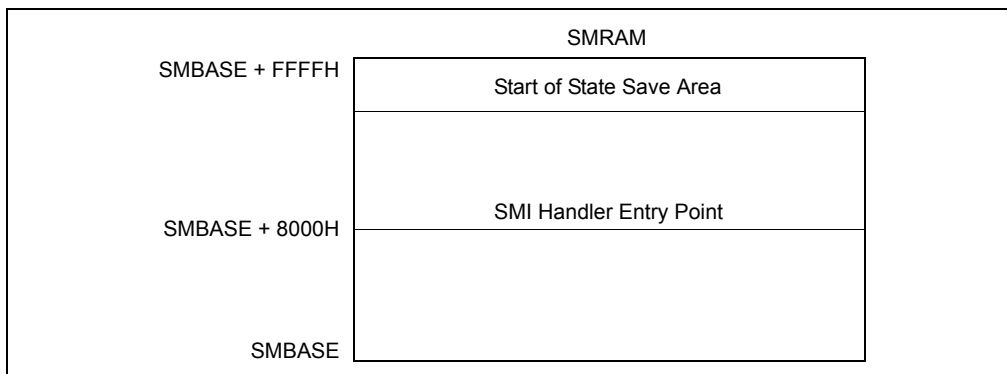


Figure 34-1. SMRAM Usage



Table 34-1. SMRAM State Save Map

Offset (Added to SMBASE + 8000H)	Register	Writable?
7FFCH	CR0	No
7FF8H	CR3	No
7FF4H	EFLAGS	Yes
7FF0H	EIP	Yes
7FECH	EDI	Yes
7FE8H	ESI	Yes
7FE4H	EBP	Yes
7FE0H	ESP	Yes
7FDCH	EBX	Yes
7FD8H	EDX	Yes
7FD4H	ECX	Yes
7FD0H	EAX	Yes
7FCCH	DR6	No
7FC8H	DR7	No
7FC4H	TR <sup>1</sup>	No
7FC0H	Reserved	No
7FBCH	GS <sup>1</sup>	No
7FB8H	FS <sup>1</sup>	No
7FB4H	DS <sup>1</sup>	No
7FB0H	SS <sup>1</sup>	No
7FACH	CS <sup>1</sup>	No
7FA8H	ES <sup>1</sup>	No
7FA4H	I/O State Field, see Section 34.7	No
7FA0H	I/O Memory Address Field, see Section 34.7	No
7F9FH-7F03H	Reserved	No
7F02H	Auto HALT Restart Field (Word)	Yes
7F00H	I/O Instruction Restart Field (Word)	Yes
7EFCH	SMM Revision Identifier Field (Doubleword)	No
7EF8H	SMBASE Field (Doubleword)	Yes
7EF7H - 7E00H	Reserved	No

**NOTE:**

1. The two most significant bytes are reserved.

The following registers are saved (but not readable) and restored upon exiting SMM:

- Control register CR4. (This register is cleared to all 0s when entering SMM).
- The hidden segment descriptor information stored in segment registers CS, DS, ES, FS, GS, and SS.

If an SMI request is issued for the purpose of powering down the processor, the values of all reserved locations in the SMM state save must be saved to nonvolatile memory.

The following state is not automatically saved and restored following an SMI and the RSM instruction, respectively:

- Debug registers DR0 through DR3.
- The x87 FPU registers.
- The MTRRs.
- Control register CR2.
- The model-specific registers (for the P6 family and Pentium processors) or test registers TR3 through TR7 (for the Pentium and Intel486 processors).
- The state of the trap controller.
- The machine-check architecture registers.
- The APIC internal interrupt state (ISR, IRR, etc.).
- The microcode update state.

If an SMI is used to power down the processor, a power-on reset will be required before returning to SMM, which will reset much of this state back to its default values. So an SMI handler that is going to trigger power down should first read these registers listed above directly, and save them (along with the rest of RAM) to nonvolatile storage. After the power-on reset, the continuation of the SMI handler should restore these values, along with the rest of the system's state. Anytime the SMI handler changes these registers in the processor, it must also save and restore them.

### NOTES

A small subset of the MSRs (such as, the time-stamp counter and performance-monitoring counters) are not arbitrarily writable and therefore cannot be saved and restored. SMM-based power-down and restoration should only be performed with operating systems that do not use or rely on the values of these registers.

Operating system developers should be aware of this fact and insure that their operating-system assisted power-down and restoration software is immune to unexpected changes in these register values.

#### 34.4.1.1 SMRAM State Save Map and Intel 64 Architecture

When the processor initially enters SMM, it writes its state to the state save area of the SMRAM. The state save area on an Intel 64 processor at [SMBASE + 8000H + 7FFFH] and extends to [SMBASE + 8000H + 7C00H].

Support for Intel 64 architecture is reported by CPUID.80000001:EDX[29] = 1. The layout of the SMRAM state save map is shown in Table 34-3.

Additionally, the SMRAM state save map shown in Table 34-3 also applies to processors with the following CPUID signatures listed in Table 34-2, irrespective of the value in CPUID.80000001:EDX[29].

**Table 34-2. Processor Signatures and 64-bit SMRAM State Save Map Format**

DisplayFamily_DisplayModel	Processor Families/Processor Number Series
06_17H	Intel Xeon Processor 5200, 5400 series, Intel Core 2 Quad processor Q9xxx, Intel Core 2 Duo processors E8000, T9000,
06_0FH	Intel Xeon Processor 3000, 3200, 5100, 5300, 7300 series, Intel Core 2 Quad, Intel Core 2 Extreme, Intel Core 2 Duo processors, Intel Pentium dual-core processors
06_1CH	45 nm Intel® Atom™ processors

Table 34-3. SMRAM State Save Map for Intel 64 Architecture

Offset (Added to SMBASE + 8000H)	Register	Writable?
7FF8H	CR0	No
7FF0H	CR3	No
7FE8H	RFLAGS	Yes
7FE0H	IA32_EFER	Yes
7FD8H	RIP	Yes
7FD0H	DR6	No
7FC8H	DR7	No
7FC4H	TR SEL <sup>1</sup>	No
7FC0H	LDTR SEL <sup>1</sup>	No
7FBCH	GS SEL <sup>1</sup>	No
7FB8H	FS SEL <sup>1</sup>	No
7FB4H	DS SEL <sup>1</sup>	No
7FB0H	SS SEL <sup>1</sup>	No
7FACH	CS SEL <sup>1</sup>	No
7FA8H	ES SEL <sup>1</sup>	No
7FA4H	IO_MISC	No
7F9CH	IO_MEM_ADDR	No
7F94H	RDI	Yes
7F8CH	RSI	Yes
7F84H	RBP	Yes
7F7CH	RSP	Yes
7F74H	RBX	Yes
7F6CH	RDX	Yes
7F64H	RCX	Yes
7F5CH	RAX	Yes
7F54H	R8	Yes
7F4CH	R9	Yes
7F44H	R10	Yes
7F3CH	R11	Yes
7F34H	R12	Yes
7F2CH	R13	Yes
7F24H	R14	Yes
7F1CH	R15	Yes
7F1BH-7F04H	Reserved	No
7F02H	Auto HALT Restart Field (Word)	Yes
7F00H	I/O Instruction Restart Field (Word)	Yes
7EFCH	SMM Revision Identifier Field (Doubleword)	No
7EF8H	SMBASE Field (Doubleword)	Yes

Table 34-3. SMRAM State Save Map for Intel 64 Architecture (Contd.)

Offset (Added to SMBASE + 8000H)	Register	Writable?
7EF7H - 7EE4H	Reserved	No
7EE0H	Setting of "enable EPT" VM-execution control	No
7ED8H	Value of EPTP VM-execution control field	No
7ED7H - 7EA0H	Reserved	No
7E9CH	LDT Base (lower 32 bits)	No
7E98H	Reserved	No
7E94H	IDT Base (lower 32 bits)	No
7E90H	Reserved	No
7E8CH	GDT Base (lower 32 bits)	No
7E8BH - 7E44H	Reserved	No
7E40H	CR4	No
7E3FH - 7DF0H	Reserved	No
7DE8H	IO_RIP	Yes
7DE7H - 7DDCH	Reserved	No
7DD8H	IDT Base (Upper 32 bits)	No
7DD4H	LDT Base (Upper 32 bits)	No
7DD0H	GDT Base (Upper 32 bits)	No
7DCFH - 7C00H	Reserved	No

**NOTE:**

1. The two most significant bytes are reserved.

### 34.4.2 SMRAM Caching

An IA-32 processor does not automatically write back and invalidate its caches before entering SMM or before exiting SMM. Because of this behavior, care must be taken in the placement of the SMRAM in system memory and in the caching of the SMRAM to prevent cache incoherence when switching back and forth between SMM and protected mode operation. Either of the following three methods of locating the SMRAM in system memory will guarantee cache coherency:

- Place the SRAM in a dedicated section of system memory that the operating system and applications are prevented from accessing. Here, the SRAM can be designated as cacheable (WB, WT, or WC) for optimum processor performance, without risking cache incoherence when entering or exiting SMM.
- Place the SRAM in a section of memory that overlaps an area used by the operating system (such as the video memory), but designate the SMRAM as uncacheable (UC). This method prevents cache access when in SMM to maintain cache coherency, but the use of uncacheable memory reduces the performance of SMM code.
- Place the SRAM in a section of system memory that overlaps an area used by the operating system and/or application code, but explicitly flush (write back and invalidate) the caches upon entering and exiting SMM mode. This method maintains cache coherency, but incurs the overhead of two complete cache flushes.

For Pentium 4, Intel Xeon, and P6 family processors, a combination of the first two methods of locating the SMRAM is recommended. Here the SMRAM is split between an overlapping and a dedicated region of memory. Upon entering SMM, the SMRAM space that is accessed overlaps video memory (typically located in low memory). This SMRAM section is designated as UC memory. The initial SMM code then jumps to a second SMRAM section that is located in a dedicated region of system memory (typically in high memory). This SMRAM section can be cached for optimum processor performance.

For systems that explicitly flush the caches upon entering SMM (the third method described above), the cache flush can be accomplished by asserting the FLUSH# pin at the same time as the request to enter SMM (generally initiated by asserting the SMI# pin). The priorities of the FLUSH# and SMI# pins are such that the FLUSH# is serviced first. To guarantee this behavior, the processor requires that the following constraints on the interaction of FLUSH# and SMI# be met. In a system where the FLUSH# and SMI# pins are synchronous and the set up and hold times are met, then the FLUSH# and SMI# pins may be asserted in the same clock. In asynchronous systems, the FLUSH# pin must be asserted at least one clock before the SMI# pin to guarantee that the FLUSH# pin is serviced first.

Upon leaving SMM (for systems that explicitly flush the caches), the WBINVD instruction should be executed prior to leaving SMM to flush the caches.

## NOTES

In systems based on the Pentium processor that use the FLUSH# pin to write back and invalidate cache contents before entering SMM, the processor will prefetch at least one cache line in between when the Flush Acknowledge cycle is run and the subsequent recognition of SMI# and the assertion of SMIACK#.

It is the obligation of the system to ensure that these lines are not cached by returning KEN# inactive to the Pentium processor.

### 34.4.2.1 System Management Range Registers (SMRR)

SMI handler code and data stored by SMM code resides in SMRAM. The SMRR interface is an enhancement in Intel 64 architecture to limit cacheable reference of addresses in SMRAM to code running in SMM. The SMRR interface can be configured only by code running in SMM. Details of SMRR is described in Section 11.11.2.4.

## 34.5 SMI HANDLER EXECUTION ENVIRONMENT

Section 34.5.1 describes the initial execution environment for an SMI handler. An SMI handler may re-configure its execution environment to other supported operating modes. Section 34.5.2 discusses modifications an SMI handler can make to its execution environment.

### 34.5.1 Initial SMM Execution Environment

After saving the current context of the processor, the processor initializes its core registers to the values shown in Table 34-4. Upon entering SMM, the PE and PG flags in control register CR0 are cleared, which places the processor in an environment similar to real-address mode. The differences between the SMM execution environment and the real-address mode execution environment are as follows:

- The addressable address space ranges from 0 to FFFFFFFFH (4 GBytes).
- The normal 64-KByte segment limit for real-address mode is increased to 4 GBytes.
- The default operand and address sizes are set to 16 bits, which restricts the addressable SMRAM address space to the 1-MByte real-address mode limit for native real-address-mode code. However, operand-size and address-size override prefixes can be used to access the address space beyond the 1-MByte.

**Table 34-4. Processor Register Initialization in SMM**

Register	Contents
General-purpose registers	Undefined
EFLAGS	00000002H
EIP	00008000H
CS selector	SMM Base shifted right 4 bits (default 3000H)
CS base	SMM Base (default 30000H)
DS, ES, FS, GS, SS Selectors	0000H

**Table 34-4. Processor Register Initialization in SMM**

DS, ES, FS, GS, SS Bases	000000000H
DS, ES, FS, GS, SS Limits	0FFFFFFFH
CR0	PE, EM, TS, and PG flags set to 0; others unmodified
CR4	Cleared to zero
DR6	Undefined
DR7	00000400H

- Near jumps and calls can be made to anywhere in the 4-GByte address space if a 32-bit operand-size override prefix is used. Due to the real-address-mode style of base-address formation, a far call or jump cannot transfer control to a segment with a base address of more than 20 bits (1 MByte). However, since the segment limit in SMM is 4 GBytes, offsets into a segment that go beyond the 1-MByte limit are allowed when using 32-bit operand-size override prefixes. Any program control transfer that does not have a 32-bit operand-size override prefix truncates the EIP value to the 16 low-order bits.
- Data and the stack can be located anywhere in the 4-GByte address space, but can be accessed only with a 32-bit address-size override if they are located above 1 MByte. As with the code segment, the base address for a data or stack segment cannot be more than 20 bits.

The value in segment register CS is automatically set to the default of 30000H for the SMBASE shifted 4 bits to the right; that is, 3000H. The EIP register is set to 8000H. When the EIP value is added to shifted CS value (the SMBASE), the resulting linear address points to the first instruction of the SMI handler.

The other segment registers (DS, SS, ES, FS, and GS) are cleared to 0 and their segment limits are set to 4 GBytes. In this state, the SMRAM address space may be treated as a single flat 4-GByte linear address space. If a segment register is loaded with a 16-bit value, that value is then shifted left by 4 bits and loaded into the segment base (hidden part of the segment register). The limits and attributes are not modified.

Maskable hardware interrupts, exceptions, NMI interrupts, SMI interrupts, A20M interrupts, single-step traps, breakpoint traps, and INIT operations are inhibited when the processor enters SMM. Maskable hardware interrupts, exceptions, single-step traps, and breakpoint traps can be enabled in SMM if the SMM execution environment provides and initializes an interrupt table and the necessary interrupt and exception handlers (see Section 34.6).

### 34.5.2 SMI Handler Operating Mode Switching

Within SMM, an SMI handler may change the processor's operating mode (e.g., to enable PAE paging, enter 64-bit mode, etc.) after it has made proper preparation and initialization to do so. For example, if switching to 32-bit protected mode, the SMI handler should follow the guidelines provided in Chapter 9, "Processor Management and Initialization". If the SMI handler does wish to change operating mode, it is responsible for executing the appropriate mode-transition code after each SMI.

It is recommended that the SMI handler make use of all means available to protect the integrity of its critical code and data. In particular, it should use the system-management range register (SMRR) interface if it is available (see Section 11.11.2.4). The SMRR interface can protect only the first 4 GBytes of the physical address space. The SMI handler should take that fact into account if it uses operating modes that allow access to physical addresses beyond that 4-GByte limit (e.g. PAE paging or 64-bit mode).

Execution of the RSM instruction restores the pre-SMI processor state from the SMRAM state-state map (see Section 34.4.1) into which it was stored when the processor entered SMM. (The SMBASE field in the SMRAM state-state map does not determine the state following RSM but rather the initial environment following the next entry to SMM.) Any required change to operating mode is performed by the RSM instruction; there is no need for the SMI handler to change modes explicitly prior to executing RSM.

## 34.6 EXCEPTIONS AND INTERRUPTS WITHIN SMM

When the processor enters SMM, all hardware interrupts are disabled in the following manner:

- The IF flag in the EFLAGS register is cleared, which inhibits maskable hardware interrupts from being generated.
- The TF flag in the EFLAGS register is cleared, which disables single-step traps.
- Debug register DR7 is cleared, which disables breakpoint traps. (This action prevents a debugger from accidentally breaking into an SMI handler if a debug breakpoint is set in normal address space that overlays code or data in SMRAM.)
- NMI, SMI, and A20M interrupts are blocked by internal SMM logic. (See Section 34.8 for more information about how NMIs are handled in SMM.)

Software-invoked interrupts and exceptions can still occur, and maskable hardware interrupts can be enabled by setting the IF flag. Intel recommends that SMM code be written in so that it does not invoke software interrupts (with the INT *n*, INTO, INT 3, or BOUND instructions) or generate exceptions.

If the SMI handler requires interrupt and exception handling, an SMM interrupt table and the necessary exception and interrupt handlers must be created and initialized from within SMM. Until the interrupt table is correctly initialized (using the LIDT instruction), exceptions and software interrupts will result in unpredictable processor behavior.

The following restrictions apply when designing SMM interrupt and exception-handling facilities:

- The interrupt table should be located at linear address 0 and must contain real-address mode style interrupt vectors (4 bytes containing CS and IP).
- Due to the real-address mode style of base address formation, an interrupt or exception cannot transfer control to a segment with a base address of more than 20 bits.
- An interrupt or exception cannot transfer control to a segment offset of more than 16 bits (64 KBytes).
- When an exception or interrupt occurs, only the 16 least-significant bits of the return address (EIP) are pushed onto the stack. If the offset of the interrupted procedure is greater than 64 KBytes, it is not possible for the interrupt/exception handler to return control to that procedure. (One solution to this problem is for a handler to adjust the return address on the stack.)
- The SMBASE relocation feature affects the way the processor will return from an interrupt or exception generated while the SMI handler is executing. For example, if the SMBASE is relocated to above 1 MByte, but the exception handlers are below 1 MByte, a normal return to the SMI handler is not possible. One solution is to provide the exception handler with a mechanism for calculating a return address above 1 MByte from the 16-bit return address on the stack, then use a 32-bit far call to return to the interrupted procedure.
- If an SMI handler needs access to the debug trap facilities, it must insure that an SMM accessible debug handler is available and save the current contents of debug registers DR0 through DR3 (for later restoration). Debug registers DR0 through DR3 and DR7 must then be initialized with the appropriate values.
- If an SMI handler needs access to the single-step mechanism, it must insure that an SMM accessible single-step handler is available, and then set the TF flag in the EFLAGS register.
- If the SMI design requires the processor to respond to maskable hardware interrupts or software-generated interrupts while in SMM, it must ensure that SMM accessible interrupt handlers are available and then set the IF flag in the EFLAGS register (using the STI instruction). Software interrupts are not blocked upon entry to SMM, so they do not need to be enabled.

## 34.7 MANAGING SYNCHRONOUS AND ASYNCHRONOUS SYSTEM MANAGEMENT INTERRUPTS

When coding for a multiprocessor system or a system with Intel HT Technology, it was not always possible for an SMI handler to distinguish between a synchronous SMI (triggered during an I/O instruction) and an asynchronous SMI. To facilitate the discrimination of these two events, incremental state information has been added to the SMM state save map.

Processors that have an SMM revision ID of 30004H or higher have the incremental state information described below.

### 34.7.1 I/O State Implementation

Within the extended SMM state save map, a bit (IO\_SMI) is provided that is set only when an SMI is either taken immediately after a *successful* I/O instruction or is taken after a *successful* iteration of a REP I/O instruction (the *successful* notion pertains to the processor point of view; not necessarily to the corresponding platform function). When set, the IO\_SMI bit provides a strong indication that the corresponding SMI was synchronous. In this case, the SMM State Save Map also supplies the port address of the I/O operation. The IO\_SMI bit and the I/O Port Address may be used in conjunction with the information logged by the platform to confirm that the SMI was indeed synchronous.

The IO\_SMI bit by itself is a strong indication, not a guarantee, that the SMI is synchronous. This is because an asynchronous SMI might coincidentally be taken after an I/O instruction. In such a case, the IO\_SMI bit would still be set in the SMM state save map.

Information characterizing the I/O instruction is saved in two locations in the SMM State Save Map (Table 34-5). The IO\_SMI bit also serves as a valid bit for the rest of the I/O information fields. The contents of these I/O information fields are not defined when the IO\_SMI bit is not set.

**Table 34-5. I/O Instruction Information in the SMM State Save Map**

State (SMM Rev. ID: 30004H or higher)	Format								
	31	16	15	8	7	4	3	1	0
I/O State Field SMRAM offset 7FA4		I/O Port		Reserved		I/O Type		I/O Length	IO_SMI
	31								0
I/O Memory Address Field SMRAM offset 7FA0	I/O Memory Address								

When IO\_SMI is set, the other fields may be interpreted as follows:

- I/O length:
  - 001 – Byte
  - 010 – Word
  - 100 – Dword
- I/O instruction type (Table 34-6)

**Table 34-6. I/O Instruction Type Encodings**

Instruction	Encoding
IN Immediate	1001
IN DX	0001
OUT Immediate	1000
OUT DX	0000
INS	0011
OUTS	0010
REP INS	0111
REP OUTS	0110



## 34.8 NMI HANDLING WHILE IN SMM

NMI interrupts are blocked upon entry to the SMI handler. If an NMI request occurs during the SMI handler, it is latched and serviced after the processor exits SMM. Only one NMI request will be latched during the SMI handler. If an NMI request is pending when the processor executes the RSM instruction, the NMI is serviced before the next instruction of the interrupted code sequence. This assumes that NMIs were not blocked before the SMI occurred. If NMIs were blocked before the SMI occurred, they are blocked after execution of RSM.

Although NMI requests are blocked when the processor enters SMM, they may be enabled through software by executing an IRET instruction. If the SMI handler requires the use of NMI interrupts, it should invoke a dummy interrupt service routine for the purpose of executing an IRET instruction. Once an IRET instruction is executed, NMI interrupt requests are serviced in the same “real mode” manner in which they are handled outside of SMM.

A special case can occur if an SMI handler nests inside an NMI handler and then another NMI occurs. During NMI interrupt handling, NMI interrupts are disabled, so normally NMI interrupts are serviced and completed with an IRET instruction one at a time. When the processor enters SMM while executing an NMI handler, the processor saves the SMRAM state save map but does not save the attribute to keep NMI interrupts disabled. Potentially, an NMI could be latched (while in SMM or upon exit) and serviced upon exit of SMM even though the previous NMI handler has still not completed. One or more NMIs could thus be nested inside the first NMI handler. The NMI interrupt handler should take this possibility into consideration.

Also, for the Pentium processor, exceptions that invoke a trap or fault handler will enable NMI interrupts from inside of SMM. This behavior is implementation specific for the Pentium processor and is not part of the IA-32 architecture.

## 34.9 SMM REVISION IDENTIFIER

The SMM revision identifier field is used to indicate the version of SMM and the SMM extensions that are supported by the processor (see Figure 34-2). The SMM revision identifier is written during SMM entry and can be examined in SMRAM space at offset 7EFCH. The lower word of the SMM revision identifier refers to the version of the base SMM architecture.

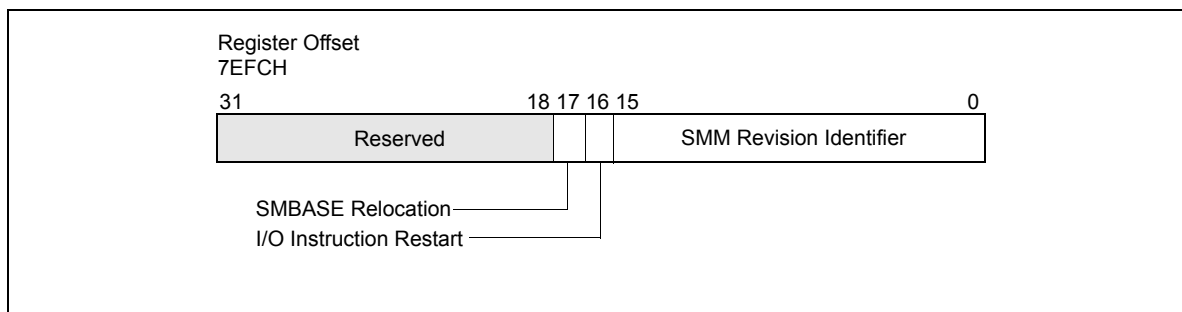


Figure 34-2. SMM Revision Identifier

The upper word of the SMM revision identifier refers to the extensions available. If the I/O instruction restart flag (bit 16) is set, the processor supports the I/O instruction restart (see Section 34.12); if the SMBASE relocation flag (bit 17) is set, SMRAM base address relocation is supported (see Section 34.11).

## 34.10 AUTO HALT RESTART

If the processor is in a HALT state (due to the prior execution of a HLT instruction) when it receives an SMI, the processor records the fact in the auto HALT restart flag in the saved processor state (see Figure 34-3). (This flag is located at offset 7F02H and bit 0 in the state save area of the SMRAM.)

If the processor sets the auto HALT restart flag upon entering SMM (indicating that the SMI occurred when the processor was in the HALT state), the SMI handler has two options:

- It can leave the auto HALT restart flag set, which instructs the RSM instruction to return program control to the HLT instruction. This option in effect causes the processor to re-enter the HALT state after handling the SMI. (This is the default operation.)
- It can clear the auto HALT restart flag, which instructs the RSM instruction to return program control to the instruction following the HLT instruction.

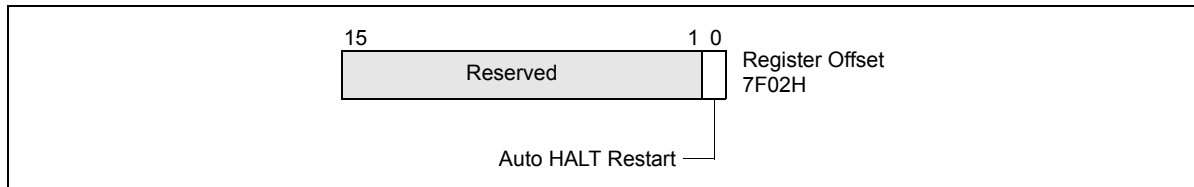


Figure 34-3. Auto HALT Restart Field

These options are summarized in Table 34-7. If the processor was not in a HALT state when the SMI was received (the auto HALT restart flag is cleared), setting the flag to 1 will cause unpredictable behavior when the RSM instruction is executed.

Table 34-7. Auto HALT Restart Flag Values

Value of Flag After Entry to SMM	Value of Flag When Exiting SMM	Action of Processor When Exiting SMM
0	0	Returns to next instruction in interrupted program or task.
0	1	Unpredictable.
1	0	Returns to next instruction after HLT instruction.
1	1	Returns to HALT state.

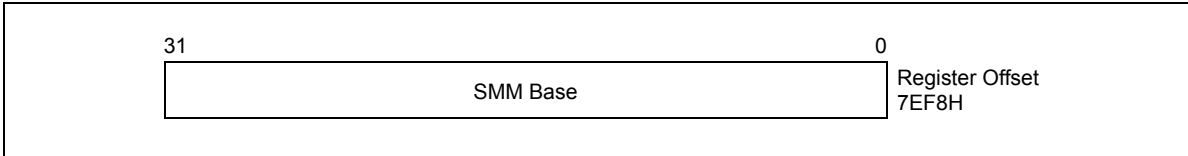
If the HLT instruction is restarted, the processor will generate a memory access to fetch the HLT instruction (if it is not in the internal cache), and execute a HLT bus transaction. This behavior results in multiple HLT bus transactions for the same HLT instruction.

### 34.10.1 Executing the HLT Instruction in SMM

The HLT instruction should not be executed during SMM, unless interrupts have been enabled by setting the IF flag in the EFLAGS register. If the processor is halted in SMM, the only event that can remove the processor from this state is a maskable hardware interrupt or a hardware reset.

## 34.11 SMBASE RELOCATION

The default base address for the SMRAM is 30000H. This value is contained in an internal processor register called the SMBASE register. The operating system or executive can relocate the SMRAM by setting the SMBASE field in the saved state map (at offset 7EF8H) to a new value (see Figure 34-4). The RSM instruction reloads the internal SMBASE register with the value in the SMBASE field each time it exits SMM. All subsequent SMI requests will use the new SMBASE value to find the starting address for the SMI handler (at SMBASE + 8000H) and the SMRAM state save area (from SMBASE + FE00H to SMBASE + FFFFH). (The processor resets the value in its internal SMBASE register to 30000H on a RESET, but does not change it on an INIT.)



**Figure 34-4. SMBASE Relocation Field**

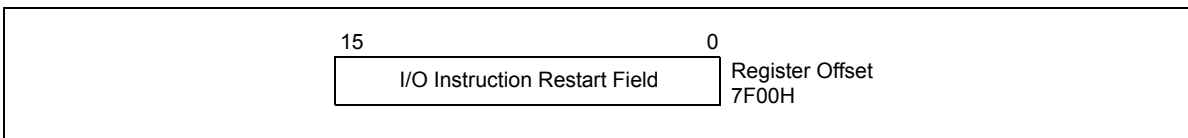
In multiple-processor systems, initialization software must adjust the SMBASE value for each processor so that the SMRAM state save areas for each processor do not overlap. (For Pentium and Intel486 processors, the SMBASE values must be aligned on a 32-KByte boundary or the processor will enter shutdown state during the execution of a RSM instruction.)

If the SMBASE relocation flag in the SMM revision identifier field is set, it indicates the ability to relocate the SMBASE (see Section 34.9).

## 34.12 I/O INSTRUCTION RESTART

If the I/O instruction restart flag in the SMM revision identifier field is set (see Section 34.9), the I/O instruction restart mechanism is present on the processor. This mechanism allows an interrupted I/O instruction to be re-executed upon returning from SMM mode. For example, if an I/O instruction is used to access a powered-down I/O device, a chip set supporting this device can intercept the access and respond by asserting SMI#. This action invokes the SMI handler to power-up the device. Upon returning from the SMI handler, the I/O instruction restart mechanism can be used to re-execute the I/O instruction that caused the SMI.

The I/O instruction restart field (at offset 7F00H in the SMM state-save area, see Figure 34-5) controls I/O instruction restart. When an RSM instruction is executed, if this field contains the value FFH, then the EIP register is modified to point to the I/O instruction that received the SMI request. The processor will then automatically re-execute the I/O instruction that the SMI trapped. (The processor saves the necessary machine state to insure that re-execution of the instruction is handled coherently.)



**Figure 34-5. I/O Instruction Restart Field**

If the I/O instruction restart field contains the value 00H when the RSM instruction is executed, then the processor begins program execution with the instruction following the I/O instruction. (When a repeat prefix is being used, the next instruction may be the next I/O instruction in the repeat loop.) Not re-executing the interrupted I/O instruction is the default behavior; the processor automatically initializes the I/O instruction restart field to 00H upon entering SMM. Table 34-8 summarizes the states of the I/O instruction restart field.

**Table 34-8. I/O Instruction Restart Field Values**

Value of Flag After Entry to SMM	Value of Flag When Exiting SMM	Action of Processor When Exiting SMM
00H	00H	Does not re-execute trapped I/O instruction.
00H	FFH	Re-executes trapped I/O instruction.

The I/O instruction restart mechanism does not indicate the cause of the SMI. It is the responsibility of the SMI handler to examine the state of the processor to determine the cause of the SMI and to determine if an I/O instruction was interrupted and should be restarted upon exiting SMM. If an SMI interrupt is signaled on a non-I/O instruction boundary, setting the I/O instruction restart field to FFH prior to executing the RSM instruction will likely result in a program error.

### 34.12.1 Back-to-Back SMI Interrupts When I/O Instruction Restart Is Being Used

If an SMI interrupt is signaled while the processor is servicing an SMI interrupt that occurred on an I/O instruction boundary, the processor will service the new SMI request before restarting the originally interrupted I/O instruction. If the I/O instruction restart field is set to FFH prior to returning from the second SMI handler, the EIP will point to an address different from the originally interrupted I/O instruction, which will likely lead to a program error. To avoid this situation, the SMI handler must be able to recognize the occurrence of back-to-back SMI interrupts when I/O instruction restart is being used and insure that the handler sets the I/O instruction restart field to 00H prior to returning from the second invocation of the SMI handler.

## 34.13 SMM MULTIPLE-PROCESSOR CONSIDERATIONS

The following should be noted when designing multiple-processor systems:

- Any processor in a multiprocessor system can respond to an SMM.
- Each processor needs its own SMRAM space. This space can be in system memory or in a separate RAM.
- The SMRAMs for different processors can be overlapped in the same memory space. The only stipulation is that each processor needs its own state save area and its own dynamic data storage area. (Also, for the Pentium and Intel486 processors, the SMBASE address must be located on a 32-KByte boundary.) Code and static data can be shared among processors. Overlapping SMRAM spaces can be done more efficiently with the P6 family processors because they do not require that the SMBASE address be on a 32-KByte boundary.
- The SMI handler will need to initialize the SMBASE for each processor.
- Processors can respond to local SMIs through their SMI# pins or to SMIs received through the APIC interface. The APIC interface can distribute SMIs to different processors.
- Two or more processors can be executing in SMM at the same time.
- When operating Pentium processors in dual processing (DP) mode, the SMIACT# pin is driven only by the MRM processor and should be sampled with ADS#. For additional details, see Chapter 14 of the *Pentium Processor Family User's Manual, Volume 1*.

SMM is not re-entrant, because the SMRAM State Save Map is fixed relative to the SMBASE. If there is a need to support two or more processors in SMM mode at the same time then each processor should have dedicated SMRAM spaces. This can be done by using the SMBASE Relocation feature (see Section 34.11).

## 34.14 DEFAULT TREATMENT OF SMIS AND SMM WITH VMX OPERATION AND SMX OPERATION

Under the default treatment, the interactions of SMIs and SMM with VMX operation are few. This section details those interactions. It also explains how this treatment affects SMX operation.

### 34.14.1 Default Treatment of SMI Delivery

Ordinary SMI delivery saves processor state into SMRAM and then loads state based on architectural definitions. Under the default treatment, processors that support VMX operation perform SMI delivery as follows:

```

enter SMM;
save the following internal to the processor:
    CR4.VMXE
    an indication of whether the logical processor was in VMX operation (root or non-root)
IF the logical processor is in VMX operation
    THEN
        save current VMCS pointer internal to the processor;
        leave VMX operation;
        save VMX-critical state defined below;

```

```

FI;
IF the logical processor supports SMX operation
  THEN
    save internal to the logical processor an indication of whether the Intel® TXT private space is locked;
    IF the TXT private space is unlocked
      THEN lock the TXT private space;
    FI;
FI;
CR4.VMXE ← 0;
perform ordinary SMI delivery:
  save processor state in SMRAM;
  set processor state to standard SMM values;1
  invalidate linear mappings and combined mappings associated with VPID 0000H (for all PCIDs); combined mappings for VPID 0000H
  are invalidated for all EP4TA values (EP4TA is the value of bits 51:12 of EPTP; see Section 28.3);

```

The pseudocode above makes reference to the saving of **VMX-critical state**. This state consists of the following: (1) SS.DPL (the current privilege level); (2) RFLAGS.VM<sup>2</sup>; (3) the state of blocking by STI and by MOV SS (see Table 24-3 in Section 24.4.2); (4) the state of virtual-NMI blocking (only if the processor is in VMX non-root operation and the “virtual NMIs” VM-execution control is 1); and (5) an indication of whether an MTF VM exit is pending (see Section 25.5.2). These data may be saved internal to the processor or in the VMCS region of the current VMCS. Processors that do not support SMI recognition while there is blocking by STI or by MOV SS need not save the state of such blocking.

If the logical processor supports the 1-setting of the “enable EPT” VM-execution control and the logical processor was in VMX non-root operation at the time of an SMI, it saves the value of that control into bit 0 of the 32-bit field at offset SMBASE + 8000H + 7EE0H (SMBASE + FEE0H; see Table 34-3).<sup>3</sup> If the logical processor was not in VMX non-root operation at the time of the SMI, it saves 0 into that bit. If the logical processor saves 1 into that bit (it was in VMX non-root operation and the “enable EPT” VM-execution control was 1), it saves the value of the EPT pointer (EPTP) into the 64-bit field at offset SMBASE + 8000H + 7ED8H (SMBASE + FED8H).

Because SMI delivery causes a logical processor to leave VMX operation, all the controls associated with VMX non-root operation are disabled in SMM and thus cannot cause VM exits while the logical processor in SMM.

### 34.14.2 Default Treatment of RSM

Ordinary execution of RSM restores processor state from SMRAM. Under the default treatment, processors that support VMX operation perform RSM as follows:

```

IF VMXE = 1 in CR4 image in SMRAM
  THEN fail and enter shutdown state;
  ELSE
    restore state normally from SMRAM;
    invalidate linear mappings and combined mappings associated with all VPIDs and all PCIDs; combined mappings are invalidated
    for all EP4TA values (EP4TA is the value of bits 51:12 of EPTP; see Section 28.3);
    IF the logical processor supports SMX operation and the Intel® TXT private space was unlocked at the time of the last SMI (as
    saved)
      THEN unlock the TXT private space;
    FI;
    CR4.VMXE ← value stored internally;

```

1. This causes the logical processor to block INIT signals, NMIs, and SMIs.
2. Section 34.14 and Section 34.15 use the notation RAX, RIP, RSP, RFLAGS, etc. for processor registers because most processors that support VMX operation also support Intel 64 architecture. For processors that do not support Intel 64 architecture, this notation refers to the 32-bit forms of these registers (EAX, EIP, ESP, EFLAGS, etc.). In a few places, notation such as EAX is used to refer specifically to the lower 32 bits of the register.
3. “Enable EPT” is a secondary processor-based VM-execution control. If bit 31 of the primary processor-based VM-execution controls is 0, SMI functions as the “enable EPT” VM-execution control were 0. See Section 24.6.2.

IF internal storage indicates that the logical processor  
had been in VMX operation (root or non-root)

THEN

enter VMX operation (root or non-root);

restore VMX-critical state as defined in Section 34.14.1;

set to their fixed values any bits in CR0 and CR4 whose values must be fixed in VMX operation (see Section 23.8);<sup>1</sup>

IF RFLAGS.VM = 0 AND (in VMX root operation OR the “unrestricted guest” VM-execution control is 0)<sup>2</sup>

THEN

CS.RPL ← SS.DPL;

SS.RPL ← SS.DPL;

FI;

restore current VMCS pointer;

FI;

leave SMM;

IF logical processor will be in VMX operation or in SMX operation after RSM

THEN block A20M and leave A20M mode;

FI;

FI;

RSM unblocks SMIs. It restores the state of blocking by NMI (see Table 24-3 in Section 24.4.2) as follows:

- If the RSM is not to VMX non-root operation or if the “virtual NMIs” VM-execution control will be 0, the state of NMI blocking is restored normally.
- If the RSM is to VMX non-root operation and the “virtual NMIs” VM-execution control will be 1, NMIs are not blocked after RSM. The state of virtual-NMI blocking is restored as part of VMX-critical state.

INIT signals are blocked after RSM if and only if the logical processor will be in VMX root operation.

If RSM returns a logical processor to VMX non-root operation, it re-establishes the controls associated with the current VMCS. If the “interrupt-window exiting” VM-execution control is 1, a VM exit occurs immediately after RSM if the enabling conditions apply. The same is true for the “NMI-window exiting” VM-execution control. Such VM exits occur with their normal priority. See Section 25.2.

If an MTF VM exit was pending at the time of the previous SMI, an MTF VM exit is pending on the instruction boundary following execution of RSM. The following items detail the treatment of MTF VM exits that may be pending following RSM:

- System-management interrupts (SMIs), INIT signals, and higher priority events take priority over these MTF VM exits. These MTF VM exits take priority over debug-trap exceptions and lower priority events.
- These MTF VM exits wake the logical processor if RSM caused the logical processor to enter the HLT state (see Section 34.10). They do not occur if the logical processor just entered the shutdown state.

### 34.14.3 Protection of CR4.VMXE in SMM

Under the default treatment, CR4.VMXE is treated as a reserved bit while a logical processor is in SMM. Any attempt by software running in SMM to set this bit causes a general-protection exception. In addition, software cannot use VMX instructions or enter VMX operation while in SMM.

### 34.14.4 VMXOFF and SMI Unblocking

The VMXOFF instruction can be executed only with the default treatment (see Section 34.15.1) and only outside SMM. If SMIs are blocked when VMXOFF is executed, VMXOFF unblocks them unless

1. If the RSM is to VMX non-root operation and both the “unrestricted guest” VM-execution control and bit 31 of the primary processor-based VM-execution controls will be 1, CR0.PE and CR0.PG retain the values that were loaded from SMRAM regardless of what is reported in the capability MSR IA32\_VMX\_CRO\_FIXED0.
2. “Unrestricted guest” is a secondary processor-based VM-execution control. If bit 31 of the primary processor-based VM-execution controls is 0, VM entry functions as if the “unrestricted guest” VM-execution control were 0. See Section 24.6.2.

IA32\_SMM\_MONITOR\_CTL[bit 2] is 1 (see Section 34.15.5 for details regarding this MSR).<sup>1</sup> Section 34.15.7 identifies a case in which SMIs may be blocked when VMXOFF is executed.

Not all processors allow this bit to be set to 1. Software should consult the VMX capability MSR IA32\_VMX\_MISC (see Appendix A.6) to determine whether this is allowed.

## 34.15 DUAL-MONITOR TREATMENT OF SMIs AND SMM

Dual-monitor treatment is activated through the cooperation of the **executive monitor** (the VMM that operates outside of SMM to provide basic virtualization) and the **SMM-transfer monitor (STM)** (the VMM that operates inside SMM—while in VMX operation—to support system-management functions). Control is transferred to the STM through VM exits; VM entries are used to return from SMM.

The dual-monitor treatment may not be supported by all processors. Software should consult the VMX capability MSR IA32\_VMX\_BASIC (see Appendix A.1) to determine whether it is supported.

### 34.15.1 Dual-Monitor Treatment Overview

The dual-monitor treatment uses an executive monitor and an SMM-transfer monitor (STM). Transitions from the executive monitor or its guests to the STM are called **SMM VM exits** and are discussed in Section 34.15.2. SMM VM exits are caused by SMIs as well as executions of VMCALL in VMX root operation. The latter allow the executive monitor to call the STM for service.

The STM runs in VMX root operation and uses VMX instructions to establish a VMCS and perform VM entries to its own guests. This is done all inside SMM (see Section 34.15.3). The STM returns from SMM, not by using the RSM instruction, but by using a VM entry that returns from SMM. Such VM entries are described in Section 34.15.4.

Initially, there is no STM and the default treatment (Section 34.14) is used. The dual-monitor treatment is not used until it is enabled and activated. The steps to do this are described in Section 34.15.5 and Section 34.15.6.

It is not possible to leave VMX operation under the dual-monitor treatment; VMXOFF will fail if executed. The dual-monitor treatment must be deactivated first. The STM deactivates dual-monitor treatment using a VM entry that returns from SMM with the “deactivate dual-monitor treatment” VM-entry control set to 1 (see Section 34.15.7).

The executive monitor configures any VMCS that it uses for VM exits to the executive monitor. SMM VM exits, which transfer control to the STM, use a different VMCS. Under the dual-monitor treatment, each logical processor uses a separate VMCS called the **SMM-transfer VMCS**. When the dual-monitor treatment is active, the logical processor maintains another VMCS pointer called the **SMM-transfer VMCS pointer**. The SMM-transfer VMCS pointer is established when the dual-monitor treatment is activated.

### 34.15.2 SMM VM Exits

An SMM VM exit is a VM exit that begins outside SMM and that ends in SMM.

Unlike other VM exits, SMM VM exits can begin in VMX root operation. SMM VM exits result from the arrival of an SMI outside SMM or from execution of VMCALL in VMX root operation outside SMM. Execution of VMCALL in VMX root operation causes an SMM VM exit only if the valid bit is set in the IA32\_SMM\_MONITOR\_CTL MSR (see Section 34.15.5).

Execution of VMCALL in VMX root operation causes an SMM VM exit even under the default treatment. This SMM VM exit activates the dual-monitor treatment (see Section 34.15.6).

Differences between SMM VM exits and other VM exits are detailed in Sections 34.15.2.1 through 34.15.2.5. Differences between SMM VM exits that activate the dual-monitor treatment and other SMM VM exits are described in Section 34.15.6.

---

1. Setting IA32\_SMM\_MONITOR\_CTL[bit 2] to 1 prevents VMXOFF from unblocking SMIs regardless of the value of the register’s valid bit (bit 0).



### 34.15.2.1 Architectural State Before a VM Exit

System-management interrupts (SMIs) that cause SMM VM exits always do so directly. They do not save state to SMRAM as they do under the default treatment.

### 34.15.2.2 Updating the Current-VMCS and Executive-VMCS Pointers

SMM VM exits begin by performing the following steps:

1. The executive-VMCS pointer field in the SMM-transfer VMCS is loaded as follows:
  - If the SMM VM exit commenced in VMX non-root operation, it receives the current-VMCS pointer.
  - If the SMM VM exit commenced in VMX root operation, it receives the VMXON pointer.
2. The current-VMCS pointer is loaded with the value of the SMM-transfer VMCS pointer.

The last step ensures that the current VMCS is the SMM-transfer VMCS. VM-exit information is recorded in that VMCS, and VM-entry control fields in that VMCS are updated. State is saved into the guest-state area of that VMCS. The VM-exit controls and host-state area of that VMCS determine how the VM exit operates.

### 34.15.2.3 Recording VM-Exit Information

SMM VM exits differ from other VM exit with regard to the way they record VM-exit information. The differences follow.

- **Exit reason.**
  - Bits 15:0 of this field contain the basic exit reason. The field is loaded with the reason for the SMM VM exit: I/O SMI (an SMI arrived immediately after retirement of an I/O instruction), other SMI, or VMCALL. See Appendix C, “VMX Basic Exit Reasons”.
  - SMM VM exits are the only VM exits that may occur in VMX root operation. Because the SMM-transfer monitor may need to know whether it was invoked from VMX root or VMX non-root operation, this information is stored in bit 29 of the exit-reason field (see Table 24-14 in Section 24.9.1). The bit is set by SMM VM exits from VMX root operation.
  - If the SMM VM exit occurred in VMX non-root operation and an MTF VM exit was pending, bit 28 of the exit-reason field is set; otherwise, it is cleared.
  - Bits 27:16 and bits 31:30 are cleared.
- **Exit qualification.** For an SMM VM exit due an SMI that arrives immediately after the retirement of an I/O instruction, the exit qualification contains information about the I/O instruction that retired immediately before the SMI. It has the format given in Table 34-9.

**Table 34-9. Exit Qualification for SMIs That Arrive Immediately After the Retirement of an I/O Instruction**

Bit Position(s)	Contents
2:0	Size of access: 0 = 1-byte 1 = 2-byte 3 = 4-byte  Other values not used.
3	Direction of the attempted access (0 = OUT, 1 = IN)
4	String instruction (0 = not string; 1 = string)
5	REP prefixed (0 = not REP; 1 = REP)
6	Operand encoding (0 = DX, 1 = immediate)



**Table 34-9. Exit Qualification for SMIs That Arrive Immediately After the Retirement of an I/O Instruction (Contd.)**

Bit Position(s)	Contents
15:7	Reserved (cleared to 0)
31:16	Port number (as specified in the I/O instruction)
63:32	Reserved (cleared to 0). These bits exist only on processors that support Intel 64 architecture.

- **Guest linear address.** This field is used for VM exits due to SMIs that arrive immediately after the retirement of an INS or OUTS instruction for which the relevant segment (ES for INS; DS for OUTS unless overridden by an instruction prefix) is usable. The field receives the value of the linear address generated by ES:(E)DI (for INS) or segment:(E)SI (for OUTS; the default segment is DS but can be overridden by a segment override prefix) at the time the instruction started. If the relevant segment is not usable, the value is undefined. On processors that support Intel 64 architecture, bits 63:32 are clear if the logical processor was not in 64-bit mode before the VM exit.
- **I/O RCX, I/O RSI, I/O RDI, and I/O RIP.** For an SMM VM exit due an SMI that arrives immediately after the retirement of an I/O instruction, these fields receive the values that were in RCX, RSI, RDI, and RIP, respectively, before the I/O instruction executed. Thus, the value saved for I/O RIP addresses the I/O instruction.

#### 34.15.2.4 Saving Guest State

SMM VM exits save the contents of the SMBASE register into the corresponding field in the guest-state area.

The value of the VMX-preemption timer is saved into the corresponding field in the guest-state area if the “save VMX-preemption timer value” VM-exit control is 1. That field becomes undefined if, in addition, either the SMM VM exit is from VMX root operation or the SMM VM exit is from VMX non-root operation and the “activate VMX-preemption timer” VM-execution control is 0.

#### 34.15.2.5 Updating Non-Register State

SMM VM exits affect the non-register state of a logical processor as follows:

- SMM VM exits cause non-maskable interrupts (NMIs) to be blocked; they may be unblocked through execution of IRET or through a VM entry (depending on the value loaded for the interruptibility state and the setting of the “virtual NMIs” VM-execution control).
- SMM VM exits cause SMIs to be blocked; they may be unblocked by a VM entry that returns from SMM (see Section 34.15.4).

SMM VM exits invalidate linear mappings and combined mappings associated with VPID 0000H for all PCIDs. Combined mappings for VPID 0000H are invalidated for all EP4TA values (EP4TA is the value of bits 51:12 of EPTP; see Section 28.3). (Ordinary VM exits are not required to perform such invalidation if the “enable VPID” VM-execution control is 1; see Section 27.5.5.)

### 34.15.3 Operation of the SMM-Transfer Monitor

Once invoked, the SMM-transfer monitor (STM) is in VMX root operation and can use VMX instructions to configure VMCSs and to cause VM entries to virtual machines supported by those structures. As noted in Section 34.15.1, the VMXOFF instruction cannot be used under the dual-monitor treatment and thus cannot be used by the STM.

The RSM instruction also cannot be used under the dual-monitor treatment. As noted in Section 25.1.3, it causes a VM exit if executed in SMM in VMX non-root operation. If executed in VMX root operation, it causes an invalid-opcode exception. The STM uses VM entries to return from SMM (see Section 34.15.4).

### 34.15.4 VM Entries that Return from SMM

The SMM-transfer monitor (STM) returns from SMM using a VM entry with the “entry to SMM” VM-entry control clear. VM entries that return from SMM reverse the effects of an SMM VM exit (see Section 34.15.2).

VM entries that return from SMM may differ from other VM entries in that they do not necessarily enter VMX non-root operation. If the executive-VMCS pointer field in the current VMCS contains the VMXON pointer, the logical processor remains in VMX root operation after VM entry.

For differences between VM entries that return from SMM and other VM entries see Sections 34.15.4.1 through 34.15.4.10.

#### 34.15.4.1 Checks on the Executive-VMCS Pointer Field

VM entries that return from SMM perform the following checks on the executive-VMCS pointer field in the current VMCS:

- Bits 11:0 must be 0.
- The pointer must not set any bits beyond the processor’s physical-address width.<sup>1,2</sup>
- The 32 bits located in memory referenced by the physical address in the pointer must contain the processor’s VMCS revision identifier (see Section 24.2).

The checks above are performed before the checks described in Section 34.15.4.2 and before any of the following checks:

- If the “deactivate dual-monitor treatment” VM-entry control is 0 and the executive-VMCS pointer field does not contain the VMXON pointer, the launch state of the executive VMCS (the VMCS referenced by the executive-VMCS pointer field) must be launched (see Section 24.11.3).
- If the “deactivate dual-monitor treatment” VM-entry control is 1, the executive-VMCS pointer field must contain the VMXON pointer (see Section 34.15.7).<sup>3</sup>

#### 34.15.4.2 Checks on VM-Execution Control Fields

VM entries that return from SMM differ from other VM entries with regard to the checks performed on the VM-execution control fields specified in Section 26.2.1.1. They do not apply the checks to the current VMCS. Instead, VM-entry behavior depends on whether the executive-VMCS pointer field contains the VMXON pointer:

- If the executive-VMCS pointer field contains the VMXON pointer (the VM entry remains in VMX root operation), the checks are not performed at all.
- If the executive-VMCS pointer field does not contain the VMXON pointer (the VM entry enters VMX non-root operation), the checks are performed on the VM-execution control fields in the executive VMCS (the VMCS referenced by the executive-VMCS pointer field in the current VMCS). These checks are performed after checking the executive-VMCS pointer field itself (for proper alignment).

Other VM entries ensure that, if “activate VMX-preemption timer” VM-execution control is 0, the “save VMX-preemption timer value” VM-exit control is also 0. This check is not performed by VM entries that return from SMM.

#### 34.15.4.3 Checks on VM-Entry Control Fields

VM entries that return from SMM differ from other VM entries with regard to the checks performed on the VM-entry control fields specified in Section 26.2.1.3.

Specifically, if the executive-VMCS pointer field contains the VMXON pointer (the VM entry remains in VMX root operation), the following must **not** all hold for the VM-entry interruption-information field:

- 
1. Software can determine a processor’s physical-address width by executing CPUID with 80000008H in EAX. The physical-address width is returned in bits 7:0 of EAX.
  2. If IA32\_VMX\_BASIC[48] is read as 1, this pointer must not set any bits in the range 63:32; see Appendix A.1.
  3. The STM can determine the VMXON pointer by reading the executive-VMCS pointer field in the current VMCS after the SMM VM exit that activates the dual-monitor treatment.

- the valid bit (bit 31) in the VM-entry interruption-information field is 1
- the interruption type (bits 10:8) is not 7 (other event); and
- the vector (bits 7:0) is not 0 (pending MTF VM exit).

#### 34.15.4.4 Checks on the Guest State Area

Section 26.3.1 specifies checks performed on fields in the guest-state area of the VMCS. Some of these checks are conditioned on the settings of certain VM-execution controls (e.g., “virtual NMIs” or “unrestricted guest”).

VM entries that return from SMM modify these checks based on whether the executive-VMCS pointer field contains the VMXON pointer:<sup>1</sup>

- If the executive-VMCS pointer field contains the VMXON pointer (the VM entry remains in VMX root operation), the checks are performed as all relevant VM-execution controls were 0. (As a result, some checks may not be performed at all.)
- If the executive-VMCS pointer field does not contain the VMXON pointer (the VM entry enters VMX non-root operation), this check is performed based on the settings of the VM-execution controls in the executive VMCS (the VMCS referenced by the executive-VMCS pointer field in the current VMCS).

For VM entries that return from SMM, the activity-state field must not indicate the wait-for-SIPI state if the executive-VMCS pointer field contains the VMXON pointer (the VM entry is to VMX root operation).

#### 34.15.4.5 Loading Guest State

VM entries that return from SMM load the SMBASE register from the SMBASE field.

VM entries that return from SMM invalidate linear mappings and combined mappings associated with all VPIDs. Combined mappings are invalidated for all EP4TA values (EP4TA is the value of bits 51:12 of EPTP; see Section 28.3). (Ordinary VM entries are required to perform such invalidation only for VPID 0000H and are not required to do even that if the “enable VPID” VM-execution control is 1; see Section 26.3.2.5.)

#### 34.15.4.6 VMX-Preemption Timer

A VM entry that returns from SMM activates the VMX-preemption timer only if the executive-VMCS pointer field does not contain the VMXON pointer (the VM entry enters VMX non-root operation) and the “activate VMX-preemption timer” VM-execution control is 1 in the executive VMCS (the VMCS referenced by the executive-VMCS pointer field). In this case, VM entry starts the VMX-preemption timer with the value in the VMX-preemption timer-value field in the current VMCS.

#### 34.15.4.7 Updating the Current-VMCS and SMM-Transfer VMCS Pointers

Successful VM entries (returning from SMM) load the SMM-transfer VMCS pointer with the current-VMCS pointer. Following this, they load the current-VMCS pointer from a field in the current VMCS:

- If the executive-VMCS pointer field contains the VMXON pointer (the VM entry remains in VMX root operation), the current-VMCS pointer is loaded from the VMCS-link pointer field.
- If the executive-VMCS pointer field does not contain the VMXON pointer (the VM entry enters VMX non-root operation), the current-VMCS pointer is loaded with the value of the executive-VMCS pointer field.

If the VM entry successfully enters VMX non-root operation, the VM-execution controls in effect after the VM entry are those from the new current VMCS. This includes any structures external to the VMCS referenced by VM-execution control fields.

The updating of these VMCS pointers occurs before event injection. Event injection is determined, however, by the VM-entry control fields in the VMCS that was current when the VM entry commenced.

---

1. The STM can determine the VMXON pointer by reading the executive-VMCS pointer field in the current VMCS after the SMM VM exit that activates the dual-monitor treatment.

### 34.15.4.8 VM Exits Induced by VM Entry

Section 26.5.1.2 describes how the event-delivery process invoked by event injection may lead to a VM exit. Section 26.6.3 to Section 26.6.7 describe other situations that may cause a VM exit to occur immediately after a VM entry.

Whether these VM exits occur is determined by the VM-execution control fields in the current VMCS. For VM entries that return from SMM, they can occur only if the executive-VMCS pointer field does not contain the VMXON pointer (the VM entry enters VMX non-root operation).

In this case, determination is based on the VM-execution control fields in the VMCS that is current after the VM entry. This is the VMCS referenced by the value of the executive-VMCS pointer field at the time of the VM entry (see Section 34.15.4.7). This VMCS also controls the delivery of such VM exits. Thus, VM exits induced by a VM entry returning from SMM are to the executive monitor and not to the STM.

### 34.15.4.9 SMI Blocking

VM entries that return from SMM determine the blocking of system-management interrupts (SMIs) as follows:

- If the “deactivate dual-monitor treatment” VM-entry control is 0, SMIs are blocked after VM entry if and only if the bit 2 in the interruptibility-state field is 1.
- If the “deactivate dual-monitor treatment” VM-entry control is 1, the blocking of SMIs depends on whether the logical processor is in SMX operation:<sup>1</sup>
  - If the logical processor is in SMX operation, SMIs are blocked after VM entry.
  - If the logical processor is outside SMX operation, SMIs are unblocked after VM entry.

VM entries that return from SMM and that do not deactivate the dual-monitor treatment may leave SMIs blocked. This feature exists to allow the STM to invoke functionality outside of SMM without unblocking SMIs.

### 34.15.4.10 Failures of VM Entries That Return from SMM

Section 26.7 describes the treatment of VM entries that fail during or after loading guest state. Such failures record information in the VM-exit information fields and load processor state as would be done on a VM exit. The VMCS used is the one that was current before the VM entry commenced. Control is thus transferred to the STM and the logical processor remains in SMM.

## 34.15.5 Enabling the Dual-Monitor Treatment

Code and data for the SMM-transfer monitor (STM) reside in a region of SMRAM called the **monitor segment** (MSEG). Code running in SMM determines the location of MSEG and establishes its content. This code is also responsible for enabling the dual-monitor treatment.

SMM code enables the dual-monitor treatment and specifies the location of MSEG by writing to the IA32\_SMM\_MONITOR\_CTL MSR (index 9BH). The MSR has the following format:

- Bit 0 is the register’s valid bit. The STM may be invoked using VMCALL only if this bit is 1. Because VMCALL is used to activate the dual-monitor treatment (see Section 34.15.6), the dual-monitor treatment cannot be activated if the bit is 0. This bit is cleared when the logical processor is reset.
- Bit 1 is reserved.
- Bit 2 determines whether executions of VMXOFF unblock SMIs under the default treatment of SMIs and SMM. Executions of VMXOFF unblock SMIs unless bit 2 is 1 (the value of bit 0 is irrelevant). See Section 34.14.4.

Certain leaf functions of the GETSEC instruction clear this bit (see Chapter 6, “Safer Mode Extensions Reference,” in *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2D*).

1. A logical processor is in SMX operation if GETSEC[SEXIT] has not been executed since the last execution of GETSEC[SENDER]. A logical processor is outside SMX operation if GETSEC[SENDER] has not been executed or if GETSEC[SEXIT] was executed after the last execution of GETSEC[SENDER]. See Chapter 6, “Safer Mode Extensions Reference,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2D*.

- Bits 11:3 are reserved.
- Bits 31:12 contain a value that, when shifted left 12 bits, is the physical address of MSEG (the MSEG base address).
- Bits 63:32 are reserved.

The following items detail use of this MSR:

- The IA32\_SMM\_MONITOR\_CTL MSR is supported only on processors that support the dual-monitor treatment.<sup>1</sup> On other processors, accesses to the MSR using RDMSR or WRMSR generate a general-protection fault (#GP(0)).
- A write to the IA32\_SMM\_MONITOR\_CTL MSR using WRMSR generates a general-protection fault (#GP(0)) if executed outside of SMM or if an attempt is made to set any reserved bit. An attempt to write to the IA32\_SMM\_MONITOR\_CTL MSR fails if made as part of a VM exit that does not end in SMM or part of a VM entry that does not begin in SMM.
- Reads from the IA32\_SMM\_MONITOR\_CTL MSR using RDMSR are allowed any time RDMSR is allowed. The MSR may be read as part of any VM exit.
- The dual-monitor treatment can be activated only if the valid bit in the MSR is set to 1.

The 32 bytes located at the MSEG base address are called the **MSEG header**. The format of the MSEG header is given in Table 34-10 (each field is 32 bits).

**Table 34-10. Format of MSEG Header**

Byte Offset	Field
0	MSEG-header revision identifier
4	SMM-transfer monitor features
8	GDTR limit
12	GDTR base offset
16	CS selector
20	EIP offset
24	ESP offset
28	CR3 offset

To ensure proper behavior in VMX operation, software should maintain the MSEG header in writeback cacheable memory. Future implementations may allow or require a different memory type.<sup>2</sup> Software should consult the VMX capability MSR IA32\_VMX\_BASIC (see Appendix A.1).

SMM code should enable the dual-monitor treatment (by setting the valid bit in IA32\_SMM\_MONITOR\_CTL MSR) only after establishing the content of the MSEG header as follows:

- Bytes 3:0 contain the **MSEG revision identifier**. Different processors may use different MSEG revision identifiers. These identifiers enable software to avoid using an MSEG header formatted for one processor on a processor that uses a different format. Software can discover the MSEG revision identifier that a processor uses by reading the VMX capability MSR IA32\_VMX\_MISC (see Appendix A.6).

1. Software should consult the VMX capability MSR IA32\_VMX\_BASIC (see Appendix A.1) to determine whether the dual-monitor treatment is supported.

2. Alternatively, software may map the MSEG header with the UC memory type; this may be necessary, depending on how memory is organized. Doing so is strongly discouraged unless necessary as it will cause the performance of transitions using those structures to suffer significantly. In addition, the processor will continue to use the memory type reported in the VMX capability MSR IA32\_VMX\_BASIC with exceptions noted in Appendix A.1.

- Bytes 7:4 contain the **SMM-transfer monitor features** field. Bits 31:1 of this field are reserved and must be zero. Bit 0 of the field is the **IA-32e mode SMM feature bit**. It indicates whether the logical processor will be in IA-32e mode after the STM is activated (see Section 34.15.6).
- Bytes 31:8 contain fields that determine how processor state is loaded when the STM is activated (see Section 34.15.6.6). SMM code should establish these fields so that activating of the STM invokes the STM's initialization code.

### 34.15.6 Activating the Dual-Monitor Treatment

The dual-monitor treatment may be enabled by SMM code as described in Section 34.15.5. The dual-monitor treatment is activated only if it is enabled and only by the executive monitor. The executive monitor activates the dual-monitor treatment by executing VMCALL in VMX root operation.

When VMCALL activates the dual-monitor treatment, it causes an SMM VM exit. Differences between this SMM VM exit and other SMM VM exits are discussed in Sections 34.15.6.1 through 34.15.6.7. See also “VMCALL—Call to VM Monitor” in Chapter 30.

#### 34.15.6.1 Initial Checks

An execution of VMCALL attempts to activate the dual-monitor treatment if (1) the processor supports the dual-monitor treatment;<sup>1</sup> (2) the logical processor is in VMX root operation; (3) the logical processor is outside SMM and the valid bit is set in the IA32\_SMM\_MONITOR\_CTL MSR; (4) the logical processor is not in virtual-8086 mode and not in compatibility mode; (5) CPL = 0; and (6) the dual-monitor treatment is not active.

The VMCS that manages SMM VM exit caused by this VMCALL is the current VMCS established by the executive monitor. The VMCALL performs the following checks on the current VMCS in the order indicated:

1. There must be a current VMCS pointer.
2. The launch state of the current VMCS must be clear.
3. Reserved bits in the VM-exit controls in the current VMCS must be set properly. Software may consult the VMX capability MSR IA32\_VMX\_EXIT\_CTLS to determine the proper settings (see Appendix A.4).

If any of these checks fail, subsequent checks are skipped and VMCALL fails. If all these checks succeed, the logical processor uses the IA32\_SMM\_MONITOR\_CTL MSR to determine the base address of MSEG. The following checks are performed in the order indicated:

1. The logical processor reads the 32 bits at the base of MSEG and compares them to the processor's MSEG revision identifier.
2. The logical processor reads the SMM-transfer monitor features field:
  - Bit 0 of the field is the IA-32e mode SMM feature bit, and it indicates whether the logical processor will be in IA-32e mode after the SMM-transfer monitor (STM) is activated.
    - If the VMCALL is executed on a processor that does not support Intel 64 architecture, the IA-32e mode SMM feature bit must be 0.
    - If the VMCALL is executed in 64-bit mode, the IA-32e mode SMM feature bit must be 1.
  - Bits 31:1 of this field are currently reserved and must be zero.

If any of these checks fail, subsequent checks are skipped and the VMCALL fails.

#### 34.15.6.2 MSEG Checking

SMM VM exits that activate the dual-monitor treatment check the following before updating the current-VMCS pointer and the executive-VMCS pointer field (see Section 34.15.2.2):

- 
1. Software should consult the VMX capability MSR IA32\_VMX\_BASIC (see Appendix A.1) to determine whether the dual-monitor treatment is supported.

- The 32 bits at the MSEG base address (used as a physical address) must contain the processor's MSEG revision identifier.
- Bits 31:1 of the SMM-transfer monitor features field in the MSEG header (see Table 34-10) must be 0. Bit 0 of the field (the IA-32e mode SMM feature bit) must be 0 if the processor does not support Intel 64 architecture.

If either of these checks fail, execution of VMCALL fails.

### 34.15.6.3 Updating the Current-VMCS and Executive-VMCS Pointers

Before performing the steps in Section 34.15.2.2, SMM VM exits that activate the dual-monitor treatment begin by loading the SMM-transfer VMCS pointer with the value of the current-VMCS pointer.

### 34.15.6.4 Saving Guest State

As noted in Section 34.15.2.4, SMM VM exits save the contents of the SMBASE register into the corresponding field in the guest-state area. While this is true also for SMM VM exits that activate the dual-monitor treatment, the VMCS used for those VM exits exists outside SMRAM.

The SMM-transfer monitor (STM) can also discover the current value of the SMBASE register by using the RDMSR instruction to read the IA32\_SMBASE MSR (MSR address 9EH). The following items detail use of this MSR:

- The MSR is supported only if IA32\_VMX\_MISC[15] = 1 (see Appendix A.6).
- A write to the IA32\_SMBASE MSR using WRMSR generates a general-protection fault (#GP(0)). An attempt to write to the IA32\_SMBASE MSR fails if made as part of a VM exit or part of a VM entry.
- A read from the IA32\_SMBASE MSR using RDMSR generates a general-protection fault (#GP(0)) if executed outside of SMM. An attempt to read from the IA32\_SMBASE MSR fails if made as part of a VM exit that does not end in SMM.

### 34.15.6.5 Saving MSRs

The VM-exit MSR-store area is not used by SMM VM exits that activate the dual-monitor treatment. No MSRs are saved into that area.

### 34.15.6.6 Loading Host State

The VMCS that is current during an SMM VM exit that activates the dual-monitor treatment was established by the executive monitor. It does not contain the VM-exit controls and host state required to initialize the STM. For this reason, such SMM VM exits do not load processor state as described in Section 27.5. Instead, state is set to fixed values or loaded based on the content of the MSEG header (see Table 34-10):

- CR0 is set to as follows:
  - PG, NE, ET, MP, and PE are all set to 1.
  - CD and NW are left unchanged.
  - All other bits are cleared to 0.
- CR3 is set as follows:
  - Bits 63:32 are cleared on processors that support IA-32e mode.
  - Bits 31:12 are set to bits 31:12 of the sum of the MSEG base address and the CR3-offset field in the MSEG header.
  - Bits 11:5 and bits 2:0 are cleared (the corresponding bits in the CR3-offset field in the MSEG header are ignored).
  - Bits 4:3 are set to bits 4:3 of the CR3-offset field in the MSEG header.
- CR4 is set as follows:
  - MCE and PGE are cleared.
  - PAE is set to the value of the IA-32e mode SMM feature bit.



- If the IA-32e mode SMM feature bit is clear, PSE is set to 1 if supported by the processor; if the bit is set, PSE is cleared.
- All other bits are unchanged.
- DR7 is set to 400H.
- The IA32\_DEBUGCTL MSR is cleared to 00000000\_00000000H.
- The registers CS, SS, DS, ES, FS, and GS are loaded as follows:
  - All registers are usable.
  - CS.selector is loaded from the corresponding field in the MSEG header (the high 16 bits are ignored), with bits 2:0 cleared to 0. If the result is 0000H, CS.selector is set to 0008H.
  - The selectors for SS, DS, ES, FS, and GS are set to CS.selector+0008H. If the result is 0000H (if the CS selector was FFF8H), these selectors are instead set to 0008H.
  - The base addresses of all registers are cleared to zero.
  - The segment limits for all registers are set to FFFFFFFFH.
  - The AR bytes for the registers are set as follows:
    - CS.Type is set to 11 (execute/read, accessed, non-conforming code segment).
    - For SS, DS, FS, and GS, the Type is set to 3 (read/write, accessed, expand-up data segment).
    - The S bits for all registers are set to 1.
    - The DPL for each register is set to 0.
    - The P bits for all registers are set to 1.
    - On processors that support Intel 64 architecture, CS.L is loaded with the value of the IA-32e mode SMM feature bit.
    - CS.D is loaded with the inverse of the value of the IA-32e mode SMM feature bit.
    - For each of SS, DS, FS, and GS, the D/B bit is set to 1.
    - The G bits for all registers are set to 1.
- LDTR is unusable. The LDTR selector is cleared to 0000H, and the register is otherwise undefined (although the base address is always canonical)
- GDTR.base is set to the sum of the MSEG base address and the GDTR base-offset field in the MSEG header (bits 63:32 are always cleared on processors that support IA-32e mode). GDTR.limit is set to the corresponding field in the MSEG header (the high 16 bits are ignored).
- IDTR.base is unchanged. IDTR.limit is cleared to 0000H.
- RIP is set to the sum of the MSEG base address and the value of the RIP-offset field in the MSEG header (bits 63:32 are always cleared on logical processors that support IA-32e mode).
- RSP is set to the sum of the MSEG base address and the value of the RSP-offset field in the MSEG header (bits 63:32 are always cleared on logical processor that supports IA-32e mode).
- RFLAGS is cleared, except bit 1, which is always set.
- The logical processor is left in the active state.
- Event blocking after the SMM VM exit is as follows:
  - There is no blocking by STI or by MOV SS.
  - There is blocking by non-maskable interrupts (NMIs) and by SMIs.
- There are no pending debug exceptions after the SMM VM exit.
- For processors that support IA-32e mode, the IA32\_EFER MSR is modified so that LME and LMA both contain the value of the IA-32e mode SMM feature bit.

If any of CR3[63:5], CR4.PAE, CR4.PSE, or IA32\_EFER.LMA is changing, the TLBs are updated so that, after VM exit, the logical processor does not use translations that were cached before the transition. This is not neces-



sary for changes that would not affect paging due to the settings of other bits (for example, changes to CR4.PSE if IA32\_EFER.LMA was 1 before and after the transition).

### 34.15.6.7 Loading MSRs

The VM-exit MSR-load area is not used by SMM VM exits that activate the dual-monitor treatment. No MSRs are loaded from that area.

## 34.15.7 Deactivating the Dual-Monitor Treatment

The SMM-transfer monitor may deactivate the dual-monitor treatment and return the processor to default treatment of SMIs and SMM (see Section 34.14). It does this by executing a VM entry with the “deactivate dual-monitor treatment” VM-entry control set to 1.

As noted in Section 26.2.1.3 and Section 34.15.4.1, an attempt to deactivate the dual-monitor treatment fails in the following situations: (1) the processor is not in SMM; (2) the “entry to SMM” VM-entry control is 1; or (3) the executive-VMCS pointer does not contain the VMXON pointer (the VM entry is to VMX non-root operation).

As noted in Section 34.15.4.9, VM entries that deactivate the dual-monitor treatment ignore the SMI bit in the interruptibility-state field of the guest-state area. Instead, the blocking of SMIs following such a VM entry depends on whether the logical processor is in SMX operation:<sup>1</sup>

- If the logical processor is in SMX operation, SMIs are blocked after VM entry. SMIs may later be unblocked by the VMXOFF instruction (see Section 34.14.4) or by certain leaf functions of the GETSEC instruction (see Chapter 6, “Safer Mode Extensions Reference,” in *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2D*).
- If the logical processor is outside SMX operation, SMIs are unblocked after VM entry.

## 34.16 SMI AND PROCESSOR EXTENDED STATE MANAGEMENT

On processors that support processor extended states using XSAVE/XRSTOR (see Chapter 13, “Managing State Using the XSAVE Feature Set” of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*), the processor does not save any XSAVE/XRSTOR related state on an SMI. It is the responsibility of the SMI handler code to properly preserve the state information (including CR4.OSXSAVE, XCR0, and possibly processor extended states using XSAVE/XRSTOR). Therefore, the SMI handler must follow the rules described in Chapter 13, “Managing State Using the XSAVE Feature Set” of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*.

## 34.17 MODEL-SPECIFIC SYSTEM MANAGEMENT ENHANCEMENT

This section describes enhancement of system management features that apply only to the 4th generation Intel Core processors. These features are model-specific. BIOS and SMM handler must use CPUID to enumerate DisplayFamily\_DisplayModel signature when programming with these interfaces.

### 34.17.1 SMM Handler Code Access Control

The BIOS may choose to restrict the address ranges of code that SMM handler executes. When SMM handler code execution check is enabled, an attempt by the SMM handler to execute outside the ranges specified by SMRR (see Section 34.4.2.1) will cause the assertion of an unrecoverable machine check exception (MCE).

---

1. A logical processor is in SMX operation if GETSEC[SEXIT] has not been executed since the last execution of GETSEC[SENDER]. A logical processor is outside SMX operation if GETSEC[SENDER] has not been executed or if GETSEC[SEXIT] was executed after the last execution of GETSEC[SENDER]. See Chapter 6, “Safer Mode Extensions Reference,” in *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2B*.

The interface to enable SMM handler code access check resides in a per-package scope model-specific register MSR\_SMM\_FEATURE\_CONTROL at address 4E0H. An attempt to access MSR\_SMM\_FEATURE\_CONTROL outside of SMM will cause a #GP. Writes to MSR\_SMM\_FEATURE\_CONTROL is further protected by configuration interface of MSR\_SMM\_MCA\_CAP at address 17DH.

Details of the interface of MSR\_SMM\_FEATURE\_CONTROL and MSR\_SMM\_MCA\_CAP are described in Table 35-27.

### 34.17.2 SMI Delivery Delay Reporting

Entry into the system management mode occurs at instruction boundary. In situations where a logical processor is executing an instruction involving a long flow of internal operations, servicing an SMI by that logical processor will be delayed. Delayed servicing of SMI of each logical processor due to executing long flows of internal operation in a physical processor can be queried via a package-scope register MSR\_SMM\_DELAYED at address 4E2H.

The interface to enable reporting of SMI delivery delay due to long internal flows resides in a per-package scope model-specific register MSR\_SMM\_DELAYED. An attempt to access MSR\_SMM\_DELAYED outside of SMM will cause a #GP. Availability to MSR\_SMM\_DELAYED is protected by configuration interface of MSR\_SMM\_MCA\_CAP at address 17DH.

Details of the interface of MSR\_SMM\_DELAYED and MSR\_SMM\_MCA\_CAP are described in Table 35-27.

### 34.17.3 Blocked SMI Reporting

A logical processor may have entered into a state and blocked from servicing other interrupts (including SMI). Logical processors in a physical processor that are blocked in serving SMI can be queried in a package-scope register MSR\_SMM\_BLOCKED at address 4E3H. An attempt to access MSR\_SMM\_BLOCKED outside of SMM will cause a #GP.

Details of the interface of MSR\_SMM\_BLOCKED is described in Table 35-27.

## 19. Updates to Chapter 35, Volume 3C

Change bars show changes to Chapter 35 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C: System Programming Guide, Part 3*.

-----  
Changes include updates to Haswell/Broadwell RAPL\_ENERGY\_STATUS behavior for PP0 and DRAM, miscellaneous Haswell/Broadwell MSR updates, Skylake perf\_limit MSR update, minor updates to Knights Landing MSRs.

## CHAPTER 35 MODEL-SPECIFIC REGISTERS (MSRS)

This chapter lists MSRs across Intel processor families. All MSRs listed can be read with the RDMSR and written with the WRMSR instructions.

Register addresses are given in both hexadecimal and decimal. The register name is the mnemonic register name and the bit description describes individual bits in registers.

Model specific registers and its bit-fields may be supported for a finite range of processor families/models. To distinguish between different processor family and/or models, software must use CPUID.01H leaf function to query the combination of DisplayFamily and DisplayModel to determine model-specific availability of MSRs (see CPUID instruction in Chapter 3, "Instruction Set Reference, A-L" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*). Table 35-1 lists the signature values of DisplayFamily and DisplayModel for various processor families or processor number series.

**Table 35-1. CPUID Signature Values of DisplayFamily\_DisplayModel**

DisplayFamily_DisplayModel	Processor Families/Processor Number Series
06_57H	Intel® Xeon Phi™ Processor 3200, 5200, 7200 Series
06_8EH, 06_9EH	Future Intel® Core Processors
06_55H	Future Intel® Xeon Processors
06_4EH, 06_5EH	6th generation Intel Core processors and Intel Xeon processor E3-1500m v5 product family and E3-1200 v5 product family based on Skylake microarchitecture
06_56H	Intel Xeon processor D-1500 product family based on Broadwell microarchitecture
06_4FH	Intel Xeon processor E5 v4 Family based on Broadwell microarchitecture, Intel Xeon processor E7 v4 Family, Intel Core i7-69xx Processor Extreme Edition
06_47H	5th generation Intel Core processors, Intel Xeon processor E3-1200 v4 product family based on Broadwell microarchitecture
06_3DH	Intel Core M-5xxx Processor, 5th generation Intel Core processors based on Broadwell microarchitecture
06_3FH	Intel Xeon processor E5-4600/2600/1600 v3 product families, Intel Xeon processor E7 v3 product families based on Haswell-E microarchitecture, Intel Core i7-59xx Processor Extreme Edition
06_3CH, 06_45H, 06_46H	4th Generation Intel Core processor and Intel Xeon processor E3-1200 v3 product family based on Haswell microarchitecture
06_3EH	Intel Xeon processor E7-8800/4800/2800 v2 product families based on Ivy Bridge-E microarchitecture
06_3EH	Intel Xeon processor E5-2600/1600 v2 product families and Intel Xeon processor E5-2400 v2 product family based on Ivy Bridge-E microarchitecture, Intel Core i7-49xx Processor Extreme Edition
06_3AH	3rd Generation Intel Core Processor and Intel Xeon processor E3-1200 v2 product family based on Ivy Bridge microarchitecture
06_2DH	Intel Xeon processor E5 Family based on Intel microarchitecture code name Sandy Bridge, Intel Core i7-39xx Processor Extreme Edition
06_2FH	Intel Xeon Processor E7 Family
06_2AH	Intel Xeon processor E3-1200 product family; 2nd Generation Intel Core i7, i5, i3 Processors 2xxx Series
06_2EH	Intel Xeon processor 7500, 6500 series
06_25H, 06_2CH	Intel Xeon processors 3600, 5600 series, Intel Core i7, i5 and i3 Processors
06_1EH, 06_1FH	Intel Core i7 and i5 Processors

**Table 35-1. CPUID Signature (Contd.)Values of DisplayFamily\_DisplayModel (Contd.)**

DisplayFamily_DisplayModel	Processor Families/Processor Number Series
06_1AH	Intel Core i7 Processor, Intel Xeon processor 3400, 3500, 5500 series
06_1DH	Intel Xeon processor MP 7400 series
06_17H	Intel Xeon processor 3100, 3300, 5200, 5400 series, Intel Core 2 Quad processors 8000, 9000 series
06_0FH	Intel Xeon processor 3000, 3200, 5100, 5300, 7300 series, Intel Core 2 Quad processor 6000 series, Intel Core 2 Extreme 6000 series, Intel Core 2 Duo 4000, 5000, 6000, 7000 series processors, Intel Pentium dual-core processors
06_0EH	Intel Core Duo, Intel Core Solo processors
06_0DH	Intel Pentium M processor
06_5FH	Future Intel® Atom™ processors based on Goldmont Microarchitecture
06_5CH	Next Generation Intel® Atom™ processors based on Goldmont Microarchitecture
06_4CH	Intel® Atom™ processor X7-Z8000 and X5-Z8000 series based on Airmont Microarchitecture
06_5DH	Intel® Atom™ processor X3-C3000 based on Silvermont Microarchitecture
06_5AH	Intel Atom processor Z3500 series
06_4AH	Intel Atom processor Z3400 series
06_37H	Intel Atom processor E3000 series, Z3600 series, Z3700 series
06_4DH	Intel Atom processor C2000 series
06_36H	Intel Atom processor S1000 Series
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H	Intel Atom processor family, Intel Atom processor D2000, N2000, E2000, Z2000, C1000 series
0F_06H	Intel Xeon processor 7100, 5000 Series, Intel Xeon Processor MP, Intel Pentium 4, Pentium D processors
0F_03H, 0F_04H	Intel Xeon processor, Intel Xeon processor MP, Intel Pentium 4, Pentium D processors
06_09H	Intel Pentium M processor
0F_02H	Intel Xeon Processor, Intel Xeon processor MP, Intel Pentium 4 processors
0F_0H, 0F_01H	Intel Xeon Processor, Intel Xeon processor MP, Intel Pentium 4 processors
06_7H, 06_08H, 06_0AH, 06_0BH	Intel Pentium III Xeon processor, Intel Pentium III processor
06_03H, 06_05H	Intel Pentium II Xeon processor, Intel Pentium II processor
06_01H	Intel Pentium Pro processor
05_01H, 05_02H, 05_04H	Intel Pentium processor, Intel Pentium processor with MMX Technology

The Intel® Quark™ SoC X1000 processor can be identified by the signature of DisplayFamily\_DisplayModel = 05\_09H and SteppingID = 0

### 35.1 ARCHITECTURAL MSRS

Many MSRs have carried over from one generation of IA-32 processors to the next and to Intel 64 processors. A subset of MSRs and associated bit fields, which do not change on future processor generations, are now considered architectural MSRs. For historical reasons (beginning with the Pentium 4 processor), these “architectural MSRs” were given the prefix “IA32\_”. Table 35-2 lists the architectural MSRs, their addresses, their current names, their names in previous IA-32 processors, and bit fields that are considered architectural. MSR addresses outside Table 35-2 and certain bit fields in an MSR address that may overlap with architectural MSR addresses are model-specific. Code that accesses a machine specified MSR and that is executed on a processor that does not support that MSR will generate an exception.

Architectural MSR or individual bit fields in an architectural MSR may be introduced or transitioned at the granularity of certain processor family/model or the presence of certain CPUID feature flags. The right-most column of Table 35-2 provides information on the introduction of each architectural MSR or its individual fields. This information is expressed either as signature values of "DF\_DM" (see Table 35-1) or via CPUID flags.

Certain bit field position may be related to the maximum physical address width, the value of which is expressed as "MAXPHYADDR" in Table 35-2. "MAXPHYADDR" is reported by CPUID.8000\_0008H leaf.

MSR address range between 40000000H - 400000FFH is marked as a specially reserved range. All existing and future processors will not implement any features using any MSR in this range.

**Table 35-2. IA-32 Architectural MSRs**

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
0H	0	IA32_P5_MC_ADDR (P5_MC_ADDR)	See Section 35.22, "MSRs in Pentium Processors."	<b>Pentium Processor (05_01H)</b>
1H	1	IA32_P5_MC_TYPE (P5_MC_TYPE)	See Section 35.22, "MSRs in Pentium Processors."	DF_DM = 05_01H
6H	6	IA32_MONITOR_FILTER_SIZE	See Section 8.10.5, "Monitor/Mwait Address Range Determination."	0F_03H
10H	16	IA32_TIME_STAMP_COUNTER (TSC)	See Section 17.15, "Time-Stamp Counter."	05_01H
17H	23	IA32_PLATFORM_ID (MSR_PLATFORM_ID)	<b>Platform ID (RO)</b> The operating system can use this MSR to determine "slot" information for the processor and the proper microcode update to load.	06_01H
		49:0	Reserved.	
		52:50	<b>Platform Id (RO)</b> Contains information concerning the intended platform for the processor. 52 51 50 0 0 0 Processor Flag 0 0 0 1 Processor Flag 1 0 1 0 Processor Flag 2 0 1 1 Processor Flag 3 1 0 0 Processor Flag 4 1 0 1 Processor Flag 5 1 1 0 Processor Flag 6 1 1 1 Processor Flag 7	
		63:53	Reserved.	
1BH	27	IA32_APIC_BASE (APIC_BASE)		06_01H
		7:0	Reserved	
		8	BSP flag (R/W)	
		9	Reserved	
		10	Enable x2APIC mode	06_1AH
		11	APIC Global Enable (R/W)	
		(MAXPHYADDR - 1):12	APIC Base (R/W)	
		63: MAXPHYADDR	Reserved	

**Table 35-2. IA-32 Architectural MSRs (Contd.)**

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
3AH	58	IA32_FEATURE_CONTROL	<b>Control Features in Intel 64 Processor (R/W)</b>	If any one enumeration condition for defined bit field holds
		0	Lock bit (R/WO): (1 = locked). When set, locks this MSR from being written, writes to this bit will result in GP(0). Note: Once the Lock bit is set, the contents of this register cannot be modified. Therefore the lock bit must be set after configuring support for Intel Virtualization Technology and prior to transferring control to an option ROM or the OS. Hence, once the Lock bit is set, the entire IA32_FEATURE_CONTROL contents are preserved across RESET when PWRGOOD is not deasserted.	If any one enumeration condition for defined bit field position greater than bit 0 holds
		1	Enable VMX inside SMX operation (R/WL): This bit enables a system executive to use VMX in conjunction with SMX to support Intel® Trusted Execution Technology. BIOS must set this bit only when the CPUID function 1 returns VMX feature flag and SMX feature flag set (ECX bits 5 and 6 respectively).	If CPUID.01H:ECX[5] = 1 && CPUID.01H:ECX[6] = 1
		2	Enable VMX outside SMX operation (R/WL): This bit enables VMX for system executive that do not require SMX. BIOS must set this bit only when the CPUID function 1 returns VMX feature flag set (ECX bit 5).	If CPUID.01H:ECX[5] = 1
		7:3	Reserved	
		14:8	SENTER Local Function Enables (R/WL): When set, each bit in the field represents an enable control for a corresponding SENTER function. This bit is supported only if CPUID.1:ECX.[bit 6] is set	If CPUID.01H:ECX[6] = 1
		15	SENTER Global Enable (R/WL): This bit must be set to enable SENTER leaf functions. This bit is supported only if CPUID.1:ECX.[bit 6] is set	If CPUID.01H:ECX[6] = 1
		16	Reserved	
		17	SGX Launch Control Enable (R/WL): This bit must be set to enable runtime reconfiguration of SGX Launch Control via IA32_SGXLEPUBKEYHASHn MSR.	If CPUID.(EAX=07H, ECX=0H): ECX[30] = 1
		18	SGX Global Enable (R/WL): This bit must be set to enable SGX leaf functions.	If CPUID.(EAX=07H, ECX=0H): EBX[2] = 1
		19	Reserved	

Table 35-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
		20	LMCE On (R/WL): When set, system software can program the MSRs associated with LMCE to configure delivery of some machine check exceptions to a single logical processor.	If IA32_MCG_CAP[27] = 1
		63:21	Reserved	
3BH	59	IA32_TSC_ADJUST	Per Logical Processor TSC Adjust (R/Write to clear)	If CPUID.(EAX=07H, ECX=0H): EBX[1] = 1
		63:0	<b>THREAD_ADJUST:</b> Local offset value of the IA32_TSC for a logical processor. Reset value is Zero. A write to IA32_TSC will modify the local offset in IA32_TSC_ADJUST and the content of IA32_TSC, but does not affect the internal invariant TSC hardware.	
79H	121	IA32_BIOS_UPDT_TRIG (BIOS_UPDT_TRIG)	BIOS Update Trigger (W) Executing a WRMSR instruction to this MSR causes a microcode update to be loaded into the processor. See Section 9.11.6, "Microcode Update Loader." A processor may prevent writing to this MSR when loading guest states on VM entries or saving guest states on VM exits.	06_01H
8BH	139	IA32_BIOS_SIGN_ID (BIOS_SIGN/BBL_CR_D3)	BIOS Update Signature (RO) Returns the microcode update signature following the execution of CPUID.01H. A processor may prevent writing to this MSR when loading guest states on VM entries or saving guest states on VM exits.	06_01H
		31:0	Reserved	
		63:32	It is recommended that this field be pre-loaded with 0 prior to executing CPUID. If the field remains 0 following the execution of CPUID; this indicates that no microcode update is loaded. Any non-zero value is the microcode update signature.	
8CH	140	IA32_SGXLEPUBKEYHASH0	IA32_SGXLEPUBKEYHASH[63:0] (R/W) Bits 63:0 of the SHA256 digest of the SIGSTRUCT.MODULUS for SGX Launch Enclave. On reset, the default value is the digest of Intel's signing key.	Read permitted if CPUID.(EAX=12H,ECX=0H): EAX[0]=1, Write permitted if CPUID.(EAX=12H,ECX=0H): EAX[0]=1 && IA32_FEATURE_CONTROL[17] = 1 && IA32_FEATURE_CONTROL[0] = 1



**Table 35-2. IA-32 Architectural MSRs (Contd.)**

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
8DH	141	IA32_SGXLEPUBKEYHASH1	IA32_SGXLEPUBKEYHASH[127:64] (R/W) Bits 127:64 of the SHA256 digest of the SIGSTRUCT.MODULUS for SGX Launch Enclave. On reset, the default value is the digest of Intel's signing key.	Read permitted If CPUID.(EAX=12H,ECX=0H): EAX[0]=1, Write permitted if CPUID.(EAX=12H,ECX=0H): EAX[0]=1 && IA32_FEATURE_CONTROL[17] = 1 && IA32_FEATURE_CONTROL[0] = 1
8EH	142	IA32_SGXLEPUBKEYHASH2	IA32_SGXLEPUBKEYHASH[191:128] (R/W) Bits 191:128 of the SHA256 digest of the SIGSTRUCT.MODULUS for SGX Launch Enclave. On reset, the default value is the digest of Intel's signing key.	Read permitted If CPUID.(EAX=12H,ECX=0H): EAX[0]=1, Write permitted if CPUID.(EAX=12H,ECX=0H): EAX[0]=1 && IA32_FEATURE_CONTROL[17] = 1 && IA32_FEATURE_CONTROL[0] = 1
8FH	143	IA32_SGXLEPUBKEYHASH3	IA32_SGXLEPUBKEYHASH[255:192] (R/W) Bits 255:192 of the SHA256 digest of the SIGSTRUCT.MODULUS for SGX Launch Enclave. On reset, the default value is the digest of Intel's signing key.	Read permitted If CPUID.(EAX=12H,ECX=0H): EAX[0]=1 && IA32_FEATURE_CONTROL[17] = 1 && IA32_FEATURE_CONTROL[0] = 1
9BH	155	IA32_SMM_MONITOR_CTL	SMM Monitor Configuration (R/W)	If CPUID.01H: ECX[5]=1    CPUID.01H: ECX[6] = 1
		0	Valid (R/W)	
		1	Reserved	
		2	Controls SMI unblocking by VMXOFF (see Section 34.14.4)	If IA32_VMX_MISC[28]
		11:3	Reserved	
		31:12	MSEG Base (R/W)	
		63:32	Reserved	
9EH	158	IA32_SMBASE	Base address of the logical processor's SMRAM image (RO, SMM only)	If IA32_VMX_MISC[15]
C1H	193	IA32_PMC0 (PERFCTR0)	General Performance Counter 0 (R/W)	If CPUID.0AH: EAX[15:8] > 0
C2H	194	IA32_PMC1 (PERFCTR1)	General Performance Counter 1 (R/W)	If CPUID.0AH: EAX[15:8] > 1
C3H	195	IA32_PMC2	General Performance Counter 2 (R/W)	If CPUID.0AH: EAX[15:8] > 2
C4H	196	IA32_PMC3	General Performance Counter 3 (R/W)	If CPUID.0AH: EAX[15:8] > 3
C5H	197	IA32_PMC4	General Performance Counter 4 (R/W)	If CPUID.0AH: EAX[15:8] > 4
C6H	198	IA32_PMC5	General Performance Counter 5 (R/W)	If CPUID.0AH: EAX[15:8] > 5
C7H	199	IA32_PMC6	General Performance Counter 6 (R/W)	If CPUID.0AH: EAX[15:8] > 6

Table 35-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
C8H	200	IA32_PMC7	General Performance Counter 7 (R/W)	If CPUID.0AH: EAX[15:8] > 7
E7H	231	IA32_MPERF	TSC Frequency Clock Counter (R/Write to clear)	If CPUID.06H: ECX[0] = 1
		63:0	<b>CO_MCNT: CO TSC Frequency Clock Count</b> Increments at fixed interval (relative to TSC freq.) when the logical processor is in CO. Cleared upon overflow / wrap-around of IA32_APERF.	
E8H	232	IA32_APERF	Actual Performance Clock Counter (R/Write to clear).	If CPUID.06H: ECX[0] = 1
		63:0	<b>CO_ACNT: CO Actual Frequency Clock Count</b> Accumulates core clock counts at the coordinated clock frequency, when the logical processor is in CO. Cleared upon overflow / wrap-around of IA32_MPERF.	
FEH	254	IA32_MTRRCAP (MTRRcap)	MTRR Capability (RO) Section 11.11.2.1, "IA32_MTRR_DEF_TYPE MSR."	06_01H
		7:0	VCNT: The number of variable memory type ranges in the processor.	
		8	Fixed range MTRRs are supported when set.	
		9	Reserved.	
		10	WC Supported when set.	
		11	SMRR Supported when set.	
		63:12	Reserved.	
174H	372	IA32_SYSENTER_CS	SYSENTER_CS_MSR (R/W)	06_01H
		15:0	CS Selector	
		63:16	Reserved.	
175H	373	IA32_SYSENTER_ESP	SYSENTER_ESP_MSR (R/W)	06_01H
176H	374	IA32_SYSENTER_EIP	SYSENTER_EIP_MSR (R/W)	06_01H
179H	377	IA32_MCG_CAP (MCG_CAP)	Global Machine Check Capability (RO)	06_01H
		7:0	Count: Number of reporting banks.	
		8	MCG_CTL_P: IA32_MCG_CTL is present if this bit is set	
		9	MCG_EXT_P: Extended machine check state registers are present if this bit is set	
		10	MCP_CMCI_P: Support for corrected MC error event is present.	06_01H

**Table 35-2. IA-32 Architectural MSRs (Contd.)**

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
		11	MCG_TES_P: Threshold-based error status register are present if this bit is set.	
		15:12	Reserved	
		23:16	MCG_EXT_CNT: Number of extended machine check state registers present.	
		24	MCG_SER_P: The processor supports software error recovery if this bit is set.	
		25	Reserved.	
		26	MCG_ELOG_P: Indicates that the processor allows platform firmware to be invoked when an error is detected so that it may provide additional platform specific information in an ACPI format "Generic Error Data Entry" that augments the data included in machine check bank registers.	06_3EH
		27	MCG_LMCE_P: Indicates that the processor support extended state in IA32_MCG_STATUS and associated MSR necessary to configure Local Machine Check Exception (LMCE).	06_3EH
		63:28	Reserved.	
17AH	378	IA32_MCG_STATUS (MCG_STATUS)	Global Machine Check Status (R/W0)	06_01H
		0	RIPV. Restart IP valid	06_01H
		1	EIPV. Error IP valid	06_01H
		2	MCIP. Machine check in progress	06_01H
		3	LMCE_S.	If IA32_MCG_CAP.LMCE_P[2:7] = 1
		63:4	Reserved.	
17BH	379	IA32_MCG_CTL (MCG_CTL)	Global Machine Check Control (R/W)	If IA32_MCG_CAP.CTL_P[8] = 1
180H-185H	384-389	Reserved		06_0EH <sup>1</sup>
186H	390	IA32_PERFEVTSELO (PERFEVTSELO)	Performance Event Select Register 0 (R/W)	If CPUID.0AH: EAX[15:8] > 0
		7:0	Event Select: Selects a performance event logic unit.	
		15:8	UMask: Qualifies the microarchitectural condition to detect on the selected event logic.	
		16	USR: Counts while in privilege level is not ring 0.	
		17	OS: Counts while in privilege level is ring 0.	

Table 35-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
		18	Edge: Enables edge detection if set.	
		19	PC: enables pin control.	
		20	INT: enables interrupt on counter overflow.	
		21	AnyThread: When set to 1, it enables counting the associated event conditions occurring across all logical processors sharing a processor core. When set to 0, the counter only increments the associated event conditions occurring in the logical processor which programmed the MSR.	
		22	EN: enables the corresponding performance counter to commence counting when this bit is set.	
		23	INV: invert the CMASK.	
		31:24	CMASK: When CMASK is not zero, the corresponding performance counter increments each cycle if the event count is greater than or equal to the CMASK.	
		63:32	Reserved.	
187H	391	IA32_PERFEVTSEL1 (PERFEVTSEL1)	Performance Event Select Register 1 (R/W)	If CPUID.0AH: EAX[15:8] > 1
188H	392	IA32_PERFEVTSEL2	Performance Event Select Register 2 (R/W)	If CPUID.0AH: EAX[15:8] > 2
189H	393	IA32_PERFEVTSEL3	Performance Event Select Register 3 (R/W)	If CPUID.0AH: EAX[15:8] > 3
18AH-197H	394-407	Reserved		06_0EH <sup>2</sup>
198H	408	IA32_PERF_STATUS	(RO)	0F_03H
		15:0	Current performance State Value	
		63:16	Reserved.	
199H	409	IA32_PERF_CTL	(R/W)	0F_03H
		15:0	Target performance State Value	
		31:16	Reserved.	
		32	IDA Engage. (R/W) When set to 1: disengages IDA	06_0FH (Mobile only)
		63:33	Reserved.	
19AH	410	IA32_CLOCK_MODULATION	Clock Modulation Control (R/W) See Section 14.7.3, "Software Controlled Clock Modulation."	If CPUID.01H:EDX[22] = 1
		0	Extended On-Demand Clock Modulation Duty Cycle:	If CPUID.06H:EAX[5] = 1

**Table 35-2. IA-32 Architectural MSRs (Contd.)**

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
		3:1	On-Demand Clock Modulation Duty Cycle: Specific encoded values for target duty cycle modulation.	If CPUID.01H:EDX[22] = 1
		4	On-Demand Clock Modulation Enable: Set 1 to enable modulation.	If CPUID.01H:EDX[22] = 1
		63:5	Reserved.	
19BH	411	IA32_THERM_INTERRUPT	Thermal Interrupt Control (R/W) Enables and disables the generation of an interrupt on temperature transitions detected with the processor's thermal sensors and thermal monitor. See Section 14.7.2, "Thermal Monitor."	If CPUID.01H:EDX[22] = 1
		0	High-Temperature Interrupt Enable	If CPUID.01H:EDX[22] = 1
		1	Low-Temperature Interrupt Enable	If CPUID.01H:EDX[22] = 1
		2	PROCHOT# Interrupt Enable	If CPUID.01H:EDX[22] = 1
		3	FORCEPR# Interrupt Enable	If CPUID.01H:EDX[22] = 1
		4	Critical Temperature Interrupt Enable	If CPUID.01H:EDX[22] = 1
		7:5	Reserved.	
		14:8	Threshold #1 Value	If CPUID.01H:EDX[22] = 1
		15	Threshold #1 Interrupt Enable	If CPUID.01H:EDX[22] = 1
		22:16	Threshold #2 Value	If CPUID.01H:EDX[22] = 1
		23	Threshold #2 Interrupt Enable	If CPUID.01H:EDX[22] = 1
		24	Power Limit Notification Enable	If CPUID.06H:EAX[4] = 1
		63:25	Reserved.	
19CH	412	IA32_THERM_STATUS	Thermal Status Information (RO) Contains status information about the processor's thermal sensor and automatic thermal monitoring facilities. See Section 14.7.2, "Thermal Monitor"	If CPUID.01H:EDX[22] = 1
		0	Thermal Status (RO):	If CPUID.01H:EDX[22] = 1
		1	Thermal Status Log (R/W):	If CPUID.01H:EDX[22] = 1
		2	PROCHOT # or FORCEPR# event (RO)	If CPUID.01H:EDX[22] = 1
		3	PROCHOT # or FORCEPR# log (R/WCO)	If CPUID.01H:EDX[22] = 1
		4	Critical Temperature Status (RO)	If CPUID.01H:EDX[22] = 1
		5	Critical Temperature Status log (R/WCO)	If CPUID.01H:EDX[22] = 1
		6	Thermal Threshold #1 Status (RO)	If CPUID.01H:ECX[8] = 1
		7	Thermal Threshold #1 log (R/WCO)	If CPUID.01H:ECX[8] = 1
		8	Thermal Threshold #2 Status (RO)	If CPUID.01H:ECX[8] = 1
9	Thermal Threshold #2 log (R/WCO)	If CPUID.01H:ECX[8] = 1		

Table 35-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
		10	Power Limitation Status (RO)	If CPUID.06H:EAX[4] = 1
		11	Power Limitation log (R/WCO)	If CPUID.06H:EAX[4] = 1
		12	Current Limit Status (RO)	If CPUID.06H:EAX[7] = 1
		13	Current Limit log (R/WCO)	If CPUID.06H:EAX[7] = 1
		14	Cross Domain Limit Status (RO)	If CPUID.06H:EAX[7] = 1
		15	Cross Domain Limit log (R/WCO)	If CPUID.06H:EAX[7] = 1
		22:16	Digital Readout (RO)	If CPUID.06H:EAX[0] = 1
		26:23	Reserved.	
		30:27	Resolution in Degrees Celsius (RO)	If CPUID.06H:EAX[0] = 1
		31	Reading Valid (RO)	If CPUID.06H:EAX[0] = 1
		63:32	Reserved.	
1A0H	416	IA32_MISC_ENABLE	<b>Enable Misc. Processor Features (R/W)</b> Allows a variety of processor functions to be enabled and disabled.	
		0	<b>Fast-Strings Enable</b> When set, the fast-strings feature (for REP MOVS and REP STORS) is enabled (default); when clear, fast-strings are disabled.	OF_OH
		2:1	Reserved.	
		3	<b>Automatic Thermal Control Circuit Enable (R/W)</b> 1 = Setting this bit enables the thermal control circuit (TCC) portion of the Intel Thermal Monitor feature. This allows the processor to automatically reduce power consumption in response to TCC activation. 0 = Disabled. Note: In some products clearing this bit might be ignored in critical thermal conditions, and TM1, TM2 and adaptive thermal throttling will still be activated. The default value of this field varies with product . See respective tables where default value is listed.	OF_OH
		6:4	Reserved	
		7	<b>Performance Monitoring Available (R)</b> 1 = Performance monitoring enabled 0 = Performance monitoring disabled	OF_OH
		10:8	Reserved.	

**Table 35-2. IA-32 Architectural MSRs (Contd.)**

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
		11	<b>Branch Trace Storage Unavailable (RO)</b> 1 = Processor doesn't support branch trace storage (BTS) 0 = BTS is supported	0F_0H
		12	<b>Processor Event Based Sampling (PEBS) Unavailable (RO)</b> 1 = PEBS is not supported; 0 = PEBS is supported.	06_0FH
		15:13	Reserved.	
		16	<b>Enhanced Intel SpeedStep Technology Enable (R/W)</b> 0= Enhanced Intel SpeedStep Technology disabled 1 = Enhanced Intel SpeedStep Technology enabled	If CPUID.01H: ECX[7] = 1
		17	Reserved.	
		18	<b>ENABLE MONITOR FSM (R/W)</b> When this bit is set to 0, the MONITOR feature flag is not set (CPUID.01H:ECX[bit 3] = 0). This indicates that MONITOR/MWAIT are not supported. Software attempts to execute MONITOR/MWAIT will cause #UD when this bit is 0. When this bit is set to 1 (default), MONITOR/MWAIT are supported (CPUID.01H:ECX[bit 3] = 1). If the SSE3 feature flag ECX[0] is not set (CPUID.01H:ECX[bit 0] = 0), the OS must not attempt to alter this bit. BIOS must leave it in the default state. Writing this bit when the SSE3 feature flag is set to 0 may generate a #GP exception.	0F_03H
		21:19	Reserved.	

Table 35-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
		22	<p><b>Limit CPUID Maxval (R/W)</b></p> <p>When this bit is set to 1, CPUID.00H returns a maximum value in EAX[7:0] of 2.</p> <p>BIOS should contain a setup question that allows users to specify when the installed OS does not support CPUID functions greater than 2.</p> <p>Before setting this bit, BIOS must execute the CPUID.0H and examine the maximum value returned in EAX[7:0]. If the maximum value is greater than 2, this bit is supported.</p> <p>Otherwise, this bit is not supported. Setting this bit when the maximum value is not greater than 2 may generate a #GP exception.</p> <p>Setting this bit may cause unexpected behavior in software that depends on the availability of CPUID leaves greater than 2.</p>	0F_03H
		23	<p><b>xTPR Message Disable (R/W)</b></p> <p>When set to 1, xTPR messages are disabled. xTPR messages are optional messages that allow the processor to inform the chipset of its priority.</p>	if CPUID.01H:ECX[14] = 1
		33:24	Reserved.	
		34	<p><b>XD Bit Disable (R/W)</b></p> <p>When set to 1, the Execute Disable Bit feature (XD Bit) is disabled and the XD Bit extended feature flag will be clear (CPUID.80000001H: EDX[20]=0).</p> <p>When set to a 0 (default), the Execute Disable Bit feature (if available) allows the OS to enable PAE paging and take advantage of data only pages.</p> <p>BIOS must not alter the contents of this bit location, if XD bit is not supported. Writing this bit to 1 when the XD Bit extended feature flag is set to 0 may generate a #GP exception.</p>	if CPUID.80000001H:EDX[20] = 1
		63:35	Reserved.	
1B0H	432	IA32_ENERGY_PERF_BIAS	Performance Energy Bias Hint (R/W)	if CPUID.6H:ECX[3] = 1
		3:0	<p>Power Policy Preference:</p> <p>0 indicates preference to highest performance.</p> <p>15 indicates preference to maximize energy saving.</p>	
		63:4	Reserved.	



**Table 35-2. IA-32 Architectural MSRs (Contd.)**

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
1B1H	433	IA32_PACKAGE_THERM_STATUS	Package Thermal Status Information (RO) Contains status information about the package's thermal sensor. See Section 14.8, "Package Level Thermal Management."	If CPUID.06H: EAX[6] = 1
		0	Pkg Thermal Status (RO):	
		1	Pkg Thermal Status Log (R/W):	
		2	Pkg PROCHOT # event (RO)	
		3	Pkg PROCHOT # log (R/WCO)	
		4	Pkg Critical Temperature Status (RO)	
		5	Pkg Critical Temperature Status log (R/WCO)	
		6	Pkg Thermal Threshold #1 Status (RO)	
		7	Pkg Thermal Threshold #1 log (R/WCO)	
		8	Pkg Thermal Threshold #2 Status (RO)	
		9	Pkg Thermal Threshold #1 log (R/WCO)	
		10	Pkg Power Limitation Status (RO)	
		11	Pkg Power Limitation log (R/WCO)	
		15:12	Reserved.	
		22:16	Pkg Digital Readout (RO)	
63:23	Reserved.			
1B2H	434	IA32_PACKAGE_THERM_INTERRUPT	Pkg Thermal Interrupt Control (R/W) Enables and disables the generation of an interrupt on temperature transitions detected with the package's thermal sensor. See Section 14.8, "Package Level Thermal Management."	If CPUID.06H: EAX[6] = 1
		0	Pkg High-Temperature Interrupt Enable	
		1	Pkg Low-Temperature Interrupt Enable	
		2	Pkg PROCHOT# Interrupt Enable	
		3	Reserved.	
		4	Pkg Overheat Interrupt Enable	
		7:5	Reserved.	
		14:8	Pkg Threshold #1 Value	
		15	Pkg Threshold #1 Interrupt Enable	
		22:16	Pkg Threshold #2 Value	
		23	Pkg Threshold #2 Interrupt Enable	
		24	Pkg Power Limit Notification Enable	

Table 35-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
		63:25	Reserved.	
1D9H	473	IA32_DEBUGCTL (MSR_DEBUGCTLA, MSR_DEBUGCTLB)	Trace/Profile Resource Control (R/W)	06_0EH
		0	LBR: Setting this bit to 1 enables the processor to record a running trace of the most recent branches taken by the processor in the LBR stack.	06_01H
		1	BTF: Setting this bit to 1 enables the processor to treat EFLAGS.TF as single-step on branches instead of single-step on instructions.	06_01H
		5:2	Reserved.	
		6	TR: Setting this bit to 1 enables branch trace messages to be sent.	06_0EH
		7	BTS: Setting this bit enables branch trace messages (BTMs) to be logged in a BTS buffer.	06_0EH
		8	BTINT: When clear, BTMs are logged in a BTS buffer in circular fashion. When this bit is set, an interrupt is generated by the BTS facility when the BTS buffer is full.	06_0EH
		9	1: BTS_OFF_OS: When set, BTS or BTM is skipped if CPL = 0.	06_0FH
		10	BTS_OFF_USR: When set, BTS or BTM is skipped if CPL > 0.	06_0FH
		11	FREEZE_LBRS_ON_PMI: When set, the LBR stack is frozen on a PMI request.	If CPUID.01H: ECX[15] = 1 && CPUID.0AH: EAX[7:0] > 1
		12	FREEZE_PERFMON_ON_PMI: When set, each ENABLE bit of the global counter control MSR are frozen (address 38FH) on a PMI request	If CPUID.01H: ECX[15] = 1 && CPUID.0AH: EAX[7:0] > 1
		13	ENABLE_UNCORE_PMI: When set, enables the logical processor to receive and generate PMI on behalf of the uncore.	06_1AH
		14	FREEZE_WHILE_SMM: When set, freezes perfmon and trace messages while in SMM.	If IA32_PERF_CAPABILITIES[12] = 1
		15	RTM_DEBUG: When set, enables DR7 debug bit on XBEGIN	If (CPUID.(EAX=07H, ECX=0):EBX[11] = 1)
		63:16	Reserved.	
1F2H	498	IA32_SMRR_PHYSBASE	<b>SMRR Base Address (Writeable only in SMM)</b> Base address of SMM memory range.	If IA32_MTRRCAP.SMRR[11] = 1
		7:0	Type. Specifies memory type of the range.	

**Table 35-2. IA-32 Architectural MSRs (Contd.)**

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
		11:8	Reserved.	
		31:12	<b>PhysBase.</b> SMRR physical Base Address.	
		63:32	Reserved.	
1F3H	499	IA32_SMRR_PHYSMASK	<b>SMRR Range Mask. (Writeable only in SMM)</b> Range Mask of SMM memory range.	If IA32_MTRRCAP[SMRR] = 1
		10:0	Reserved.	
		11	<b>Valid</b> Enable range mask.	
		31:12	<b>PhysMask</b> SMRR address range mask.	
		63:32	Reserved.	
1F8H	504	IA32_PLATFORM_DCA_CAP	DCA Capability (R)	If CPUID.01H: ECX[18] = 1
1F9H	505	IA32_CPU_DCA_CAP	If set, CPU supports Prefetch-Hint type.	If CPUID.01H: ECX[18] = 1
1FAH	506	IA32_DCA_0_CAP	DCA type 0 Status and Control register.	If CPUID.01H: ECX[18] = 1
		0	DCA_ACTIVE: Set by HW when DCA is fuse-enabled and no defeatures are set.	
		2:1	TRANSACTION	
		6:3	DCA_TYPE	
		10:7	DCA_QUEUE_SIZE	
		12:11	Reserved.	
		16:13	DCA_DELAY: Writes will update the register but have no HW side-effect.	
		23:17	Reserved.	
		24	Sw_BLOCK: SW can request DCA block by setting this bit.	
		25	Reserved.	
		26	HW_BLOCK: Set when DCA is blocked by HW (e.g. CRO.CD = 1).	
		31:27	Reserved.	
200H	512	IA32_MTRR_PHYSBASE0 (MTRRphysBase0)	See Section 11.11.2.3, "Variable Range MTRRs."	If CPUID.01H: EDX.MTRR[12] = 1
201H	513	IA32_MTRR_PHYSMASK0	MTRRphysMask0	If CPUID.01H: EDX.MTRR[12] = 1
202H	514	IA32_MTRR_PHYSBASE1	MTRRphysBase1	If CPUID.01H: EDX.MTRR[12] = 1
203H	515	IA32_MTRR_PHYSMASK1	MTRRphysMask1	If CPUID.01H: EDX.MTRR[12] = 1

Table 35-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
204H	516	IA32_MTRR_PHYSBASE2	MTRRphysBase2	If CPUID.01H: EDX.MTRR[12] = 1
205H	517	IA32_MTRR_PHYSMASK2	MTRRphysMask2	If CPUID.01H: EDX.MTRR[12] = 1
206H	518	IA32_MTRR_PHYSBASE3	MTRRphysBase3	If CPUID.01H: EDX.MTRR[12] = 1
207H	519	IA32_MTRR_PHYSMASK3	MTRRphysMask3	If CPUID.01H: EDX.MTRR[12] = 1
208H	520	IA32_MTRR_PHYSBASE4	MTRRphysBase4	If CPUID.01H: EDX.MTRR[12] = 1
209H	521	IA32_MTRR_PHYSMASK4	MTRRphysMask4	If CPUID.01H: EDX.MTRR[12] = 1
20AH	522	IA32_MTRR_PHYSBASE5	MTRRphysBase5	If CPUID.01H: EDX.MTRR[12] = 1
20BH	523	IA32_MTRR_PHYSMASK5	MTRRphysMask5	If CPUID.01H: EDX.MTRR[12] = 1
20CH	524	IA32_MTRR_PHYSBASE6	MTRRphysBase6	If CPUID.01H: EDX.MTRR[12] = 1
20DH	525	IA32_MTRR_PHYSMASK6	MTRRphysMask6	If CPUID.01H: EDX.MTRR[12] = 1
20EH	526	IA32_MTRR_PHYSBASE7	MTRRphysBase7	If CPUID.01H: EDX.MTRR[12] = 1
20FH	527	IA32_MTRR_PHYSMASK7	MTRRphysMask7	If CPUID.01H: EDX.MTRR[12] = 1
210H	528	IA32_MTRR_PHYSBASE8	MTRRphysBase8	if IA32_MTRRCAP[7:0] > 8
211H	529	IA32_MTRR_PHYSMASK8	MTRRphysMask8	if IA32_MTRRCAP[7:0] > 8
212H	530	IA32_MTRR_PHYSBASE9	MTRRphysBase9	if IA32_MTRRCAP[7:0] > 9
213H	531	IA32_MTRR_PHYSMASK9	MTRRphysMask9	if IA32_MTRRCAP[7:0] > 9
250H	592	IA32_MTRR_FIX64K_00000	MTRRfix64K_00000	If CPUID.01H: EDX.MTRR[12] = 1
258H	600	IA32_MTRR_FIX16K_80000	MTRRfix16K_80000	If CPUID.01H: EDX.MTRR[12] = 1
259H	601	IA32_MTRR_FIX16K_A0000	MTRRfix16K_A0000	If CPUID.01H: EDX.MTRR[12] = 1
268H	616	IA32_MTRR_FIX4K_C0000 (MTRRfix4K_C0000)	See Section 11.11.2.2, "Fixed Range MTRRs."	If CPUID.01H: EDX.MTRR[12] = 1
269H	617	IA32_MTRR_FIX4K_C8000	MTRRfix4K_C8000	If CPUID.01H: EDX.MTRR[12] = 1
26AH	618	IA32_MTRR_FIX4K_D0000	MTRRfix4K_D0000	If CPUID.01H: EDX.MTRR[12] = 1
26BH	619	IA32_MTRR_FIX4K_D8000	MTRRfix4K_D8000	If CPUID.01H: EDX.MTRR[12] = 1

**Table 35-2. IA-32 Architectural MSRs (Contd.)**

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
26CH	620	IA32_MTRR_FIX4K_E0000	MTRRfix4K_E0000	If CPUID.01H: EDX.MTRR[12] = 1
26DH	621	IA32_MTRR_FIX4K_E8000	MTRRfix4K_E8000	If CPUID.01H: EDX.MTRR[12] = 1
26EH	622	IA32_MTRR_FIX4K_F0000	MTRRfix4K_F0000	If CPUID.01H: EDX.MTRR[12] = 1
26FH	623	IA32_MTRR_FIX4K_F8000	MTRRfix4K_F8000	If CPUID.01H: EDX.MTRR[12] = 1
277H	631	IA32_PAT	IA32_PAT (R/W)	If CPUID.01H: EDX.MTRR[16] = 1
		2:0	PA0	
		7:3	Reserved.	
		10:8	PA1	
		15:11	Reserved.	
		18:16	PA2	
		23:19	Reserved.	
		26:24	PA3	
		31:27	Reserved.	
		34:32	PA4	
		39:35	Reserved.	
		42:40	PA5	
		47:43	Reserved.	
		50:48	PA6	
		55:51	Reserved.	
58:56	PA7			
63:59	Reserved.			
280H	640	IA32_MCO_CTL2	(R/W)	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 0
		14:0	Corrected error count threshold.	
		29:15	Reserved.	
		30	CMCI_EN	
		63:31	Reserved.	
281H	641	IA32_MC1_CTL2	(R/W) same fields as IA32_MCO_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 1
282H	642	IA32_MC2_CTL2	(R/W) same fields as IA32_MCO_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 2

Table 35-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
283H	643	IA32_MC3_CTL2	(R/W) same fields as IA32_MCO_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 3
284H	644	IA32_MC4_CTL2	(R/W) same fields as IA32_MCO_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 4
285H	645	IA32_MC5_CTL2	(R/W) same fields as IA32_MCO_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 5
286H	646	IA32_MC6_CTL2	(R/W) same fields as IA32_MCO_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 6
287H	647	IA32_MC7_CTL2	(R/W) same fields as IA32_MCO_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 7
288H	648	IA32_MC8_CTL2	(R/W) same fields as IA32_MCO_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 8
289H	649	IA32_MC9_CTL2	(R/W) same fields as IA32_MCO_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 9
28AH	650	IA32_MC10_CTL2	(R/W) same fields as IA32_MCO_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 10
28BH	651	IA32_MC11_CTL2	(R/W) same fields as IA32_MCO_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 11
28CH	652	IA32_MC12_CTL2	(R/W) same fields as IA32_MCO_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 12
28DH	653	IA32_MC13_CTL2	(R/W) same fields as IA32_MCO_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 13
28EH	654	IA32_MC14_CTL2	(R/W) same fields as IA32_MCO_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 14
28FH	655	IA32_MC15_CTL2	(R/W) same fields as IA32_MCO_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 15
290H	656	IA32_MC16_CTL2	(R/W) same fields as IA32_MCO_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 16
291H	657	IA32_MC17_CTL2	(R/W) same fields as IA32_MCO_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 17

**Table 35-2. IA-32 Architectural MSRs (Contd.)**

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
292H	658	IA32_MC18_CTL2	(R/W) same fields as IA32_MCO_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 18
293H	659	IA32_MC19_CTL2	(R/W) same fields as IA32_MCO_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 19
294H	660	IA32_MC20_CTL2	(R/W) same fields as IA32_MCO_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 20
295H	661	IA32_MC21_CTL2	(R/W) same fields as IA32_MCO_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 21
296H	662	IA32_MC22_CTL2	(R/W) same fields as IA32_MCO_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 22
297H	663	IA32_MC23_CTL2	(R/W) same fields as IA32_MCO_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 23
298H	664	IA32_MC24_CTL2	(R/W) same fields as IA32_MCO_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 24
299H	665	IA32_MC25_CTL2	(R/W) same fields as IA32_MCO_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 25
29AH	666	IA32_MC26_CTL2	(R/W) same fields as IA32_MCO_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 26
29BH	667	IA32_MC27_CTL2	(R/W) same fields as IA32_MCO_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 27
29CH	668	IA32_MC28_CTL2	(R/W) same fields as IA32_MCO_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 28
29DH	669	IA32_MC29_CTL2	(R/W) same fields as IA32_MCO_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 29
29EH	670	IA32_MC30_CTL2	(R/W) same fields as IA32_MCO_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 30
29FH	671	IA32_MC31_CTL2	(R/W) same fields as IA32_MCO_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 31
2FFH	767	IA32_MTRR_DEF_TYPE	MTRRdefType (R/W)	If CPUID.01H: EDX.MTRR[12] = 1
		2:0	Default Memory Type	
		9:3	Reserved.	

Table 35-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
		10	Fixed Range MTRR Enable	
		11	MTRR Enable	
		63:12	Reserved.	
309H	777	IA32_FIXED_CTR0 (MSR_PERF_FIXED_CTR0)	Fixed-Function Performance Counter 0 (R/W): Counts Instr_Retired.Any.	If CPUID.0AH: EDX[4:0] > 0
30AH	778	IA32_FIXED_CTR1 (MSR_PERF_FIXED_CTR1)	Fixed-Function Performance Counter 1 (R/W): Counts CPU_CLK_Unhalted.Core	If CPUID.0AH: EDX[4:0] > 1
30BH	779	IA32_FIXED_CTR2 (MSR_PERF_FIXED_CTR2)	Fixed-Function Performance Counter 2 (R/W): Counts CPU_CLK_Unhalted.Ref	If CPUID.0AH: EDX[4:0] > 2
345H	837	IA32_PERF_CAPABILITIES	RO	If CPUID.01H: ECX[15] = 1
		5:0	LBR format	
		6	PEBS Trap	
		7	PEBSSaveArchRegs	
		11:8	PEBS Record Format	
		12	1: Freeze while SMM is supported.	
		13	1: Full width of counter writable via IA32_A_PMCx.	
		63:14	Reserved.	
38DH	909	IA32_FIXED_CTR_CTRL	Fixed-Function Performance Counter Control (R/W)  Counter increments while the results of ANDing respective enable bit in IA32_PERF_GLOBAL_CTRL with the corresponding OS or USR bits in this MSR is true.	If CPUID.0AH: EAX[7:0] > 1
		0	ENO_OS: Enable Fixed Counter 0 to count while CPL = 0.	
		1	ENO_Usr: Enable Fixed Counter 0 to count while CPL > 0.	
		2	AnyThread: When set to 1, it enables counting the associated event conditions occurring across all logical processors sharing a processor core. When set to 0, the counter only increments the associated event conditions occurring in the logical processor which programmed the MSR.	If CPUID.0AH: EAX[7:0] > 2
		3	ENO_PMI: Enable PMI when fixed counter 0 overflows.	
		4	EN1_OS: Enable Fixed Counter 1 to count while CPL = 0.	
		5	EN1_Usr: Enable Fixed Counter 1 to count while CPL > 0.	



Table 35-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
		6	AnyThread: When set to 1, it enables counting the associated event conditions occurring across all logical processors sharing a processor core. When set to 0, the counter only increments the associated event conditions occurring in the logical processor which programmed the MSR.	If CPUID.0AH: EAX[7:0] > 2
		7	EN1_PMI: Enable PMI when fixed counter 1 overflows.	
		8	EN2_OS: Enable Fixed Counter 2 to count while CPL = 0.	
		9	EN2_Usr: Enable Fixed Counter 2 to count while CPL > 0.	
		10	AnyThread: When set to 1, it enables counting the associated event conditions occurring across all logical processors sharing a processor core. When set to 0, the counter only increments the associated event conditions occurring in the logical processor which programmed the MSR.	If CPUID.0AH: EAX[7:0] > 2
		11	EN2_PMI: Enable PMI when fixed counter 2 overflows.	
		63:12	Reserved.	
38EH	910	IA32_PERF_GLOBAL_STATUS	Global Performance Counter Status (RO)	If CPUID.0AH: EAX[7:0] > 0
		0	Ovf_PMC0: Overflow status of IA32_PMC0.	If CPUID.0AH: EAX[15:8] > 0
		1	Ovf_PMC1: Overflow status of IA32_PMC1.	If CPUID.0AH: EAX[15:8] > 1
		2	Ovf_PMC2: Overflow status of IA32_PMC2.	If CPUID.0AH: EAX[15:8] > 2
		3	Ovf_PMC3: Overflow status of IA32_PMC3.	If CPUID.0AH: EAX[15:8] > 3
		31:4	Reserved.	
		32	Ovf_FixedCtr0: Overflow status of IA32_FIXED_CTR0.	If CPUID.0AH: EAX[7:0] > 1
		33	Ovf_FixedCtr1: Overflow status of IA32_FIXED_CTR1.	If CPUID.0AH: EAX[7:0] > 1
		34	Ovf_FixedCtr2: Overflow status of IA32_FIXED_CTR2.	If CPUID.0AH: EAX[7:0] > 1
		54:35	Reserved.	
		55	Trace_ToPA_PMI: A PMI occurred due to a ToPA entry memory buffer was completely filled.	If (CPUID.(EAX=07H, ECX=0):EBX[25] = 1) && IA32_RTIT_CTL.ToPA = 1
		57:56	Reserved.	

Table 35-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
		58	LBR_Frz: LBRs are frozen due to <ul style="list-style-type: none"> <li>IA32_DEBUGCTL.FREEZE_LBR_ON_PMI=1,</li> <li>The LBR stack overflowed</li> </ul>	If CPUID.OAH: EAX[7:0] > 3
		59	CTR_Frz: Performance counters in the core PMU are frozen due to <ul style="list-style-type: none"> <li>IA32_DEBUGCTL.FREEZE_PERFMON_ON_PMI=1,</li> <li>one or more core PMU counters overflowed.</li> </ul>	If CPUID.OAH: EAX[7:0] > 3
		60	ASCI: Data in the performance counters in the core PMU may include contributions from the direct or indirect operation intel SGX to protect an enclave.	If CPUID.(EAX=07H, ECX=0):EBX[2] = 1
		61	Ovf_Uncore: Uncore counter overflow status.	If CPUID.OAH: EAX[7:0] > 2
		62	OvfBuf: DS SAVE area Buffer overflow status.	If CPUID.OAH: EAX[7:0] > 0
		63	CondChgd: status bits of this register has changed.	If CPUID.OAH: EAX[7:0] > 0
38FH	911	IA32_PERF_GLOBAL_CTRL	Global Performance Counter Control (R/W) Counter increments while the result of ANDing respective enable bit in this MSR with the corresponding OS or USR bits in the general-purpose or fixed counter control MSR is true.	If CPUID.OAH: EAX[7:0] > 0
		0	EN_PMC0	If CPUID.OAH: EAX[15:8] > 0
		1	EN_PMC1	If CPUID.OAH: EAX[15:8] > 1
		2	EN_PMC2	If CPUID.OAH: EAX[15:8] > 2
		n	EN_PMCn	If CPUID.OAH: EAX[15:8] > n
		31:n+1	Reserved.	
		32	EN_FIXED_CTR0	If CPUID.OAH: EDX[4:0] > 0
		33	EN_FIXED_CTR1	If CPUID.OAH: EDX[4:0] > 1
		34	EN_FIXED_CTR2	If CPUID.OAH: EDX[4:0] > 2
		63:35	Reserved.	
390H	912	IA32_PERF_GLOBAL_OVF_CTRL	Global Performance Counter Overflow Control (R/W)	If CPUID.OAH: EAX[7:0] > 0 && CPUID.OAH: EAX[7:0] <= 3
		0	Set 1 to Clear Ovf_PMC0 bit.	If CPUID.OAH: EAX[15:8] > 0
		1	Set 1 to Clear Ovf_PMC1 bit.	If CPUID.OAH: EAX[15:8] > 1

**Table 35-2. IA-32 Architectural MSRs (Contd.)**

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
		2	Set 1 to Clear Ovf_PMC2 bit.	If CPUID.0AH: EAX[15:8] > 2
		n	Set 1 to Clear Ovf_PMCn bit.	If CPUID.0AH: EAX[15:8] > n
		31:n	Reserved.	
		32	Set 1 to Clear Ovf_FIXED_CTR0 bit.	If CPUID.0AH: EDX[4:0] > 0
		33	Set 1 to Clear Ovf_FIXED_CTR1 bit.	If CPUID.0AH: EDX[4:0] > 1
		34	Set 1 to Clear Ovf_FIXED_CTR2 bit.	If CPUID.0AH: EDX[4:0] > 2
		54:35	Reserved.	
		55	Set 1 to Clear Trace_ToPA_PMI bit.	If (CPUID.(EAX=07H, ECX=0):EBX[25] = 1) && IA32_RTIT_CTL.ToPA = 1
		60:56	Reserved.	
		61	Set 1 to Clear Ovf_Uncore bit.	06_2EH
		62	Set 1 to Clear OvfBuf: bit.	If CPUID.0AH: EAX[7:0] > 0
		63	Set to 1 to clear CondChgd: bit.	If CPUID.0AH: EAX[7:0] > 0
390H	912	IA32_PERF_GLOBAL_STATUS_RESET	Global Performance Counter Overflow Reset Control (R/W)	If CPUID.0AH: EAX[7:0] > 3
		0	Set 1 to Clear Ovf_PMC0 bit.	If CPUID.0AH: EAX[15:8] > 0
		1	Set 1 to Clear Ovf_PMC1 bit.	If CPUID.0AH: EAX[15:8] > 1
		2	Set 1 to Clear Ovf_PMC2 bit.	If CPUID.0AH: EAX[15:8] > 2
		n	Set 1 to Clear Ovf_PMCn bit.	If CPUID.0AH: EAX[15:8] > n
		31:n	Reserved.	
		32	Set 1 to Clear Ovf_FIXED_CTR0 bit.	If CPUID.0AH: EDX[4:0] > 0
		33	Set 1 to Clear Ovf_FIXED_CTR1 bit.	If CPUID.0AH: EDX[4:0] > 1
		34	Set 1 to Clear Ovf_FIXED_CTR2 bit.	If CPUID.0AH: EDX[4:0] > 2
		54:35	Reserved.	
		55	Set 1 to Clear Trace_ToPA_PMI bit.	If (CPUID.(EAX=07H, ECX=0):EBX[25] = 1) && IA32_RTIT_CTL.ToPA[8] = 1
		57:56	Reserved.	
		58	Set 1 to Clear LBR_Frz bit.	If CPUID.0AH: EAX[7:0] > 3
		59	Set 1 to Clear CTR_Frz bit.	If CPUID.0AH: EAX[7:0] > 3
		58	Set 1 to Clear ASCII bit.	If CPUID.0AH: EAX[7:0] > 3
61	Set 1 to Clear Ovf_Uncore bit.	06_2EH		
62	Set 1 to Clear OvfBuf: bit.	If CPUID.0AH: EAX[7:0] > 0		

Table 35-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
		63	Set to 1 to clear CondChgd: bit.	If CPUID.OAH: EAX[7:0] > 0
391H	913	IA32_PERF_GLOBAL_STATUS_SET	Global Performance Counter Overflow Set Control (R/W)	If CPUID.OAH: EAX[7:0] > 3
		0	Set 1 to cause Ovf_PMC0 = 1.	If CPUID.OAH: EAX[7:0] > 3
		1	Set 1 to cause Ovf_PMC1 = 1	If CPUID.OAH: EAX[15:8] > 1
		2	Set 1 to cause Ovf_PMC2 = 1	If CPUID.OAH: EAX[15:8] > 2
		n	Set 1 to cause Ovf_PMCn = 1	If CPUID.OAH: EAX[15:8] > n
		31:n	Reserved.	
		32	Set 1 to cause Ovf_FIXED_CTR0 = 1.	If CPUID.OAH: EAX[7:0] > 3
		33	Set 1 to cause Ovf_FIXED_CTR1 = 1.	If CPUID.OAH: EAX[7:0] > 3
		34	Set 1 to cause Ovf_FIXED_CTR2 = 1.	If CPUID.OAH: EAX[7:0] > 3
		54:35	Reserved.	
		55	Set 1 to cause Trace_ToPA_PMI = 1.	If CPUID.OAH: EAX[7:0] > 3
		57:56	Reserved.	
		58	Set 1 to cause LBR_Frz = 1.	If CPUID.OAH: EAX[7:0] > 3
		59	Set 1 to cause CTR_Frz = 1.	If CPUID.OAH: EAX[7:0] > 3
		58	Set 1 to cause ASCI = 1.	If CPUID.OAH: EAX[7:0] > 3
		61	Set 1 to cause Ovf_Uncore = 1.	If CPUID.OAH: EAX[7:0] > 3
		62	Set 1 to cause OvfBuf = 1.	If CPUID.OAH: EAX[7:0] > 3
63	Reserved			
392H	914	IA32_PERF_GLOBAL_INUSE	Indicator of core perfmon interface is in use (RO)	If CPUID.OAH: EAX[7:0] > 3
		0	IA32_PERFEVTSELO in use	
		1	IA32_PERFEVTSEL1 in use	If CPUID.OAH: EAX[15:8] > 1
		2	IA32_PERFEVTSEL2 in use	If CPUID.OAH: EAX[15:8] > 2
		n	IA32_PERFEVTSELn in use	If CPUID.OAH: EAX[15:8] > n
		31:n	Reserved.	
		32	IA32_FIXED_CTR0 in use	
		33	IA32_FIXED_CTR1 in use	
		34	IA32_FIXED_CTR2 in use	
		62:35	Reserved or Model specific.	
		63	PMI in use.	
3F1H	1009	IA32_PEBS_ENABLE	PEBS Control (R/W)	

Table 35-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
		0	Enable PEBS on IA32_PMC0.	06_0FH
		3:1	Reserved or Model specific.	
		31:4	Reserved.	
		35:32	Reserved or Model specific.	
		63:36	Reserved.	
400H	1024	IA32_MCO_CTL	MCO_CTL	If IA32_MCG_CAP.CNT >0
401H	1025	IA32_MCO_STATUS	MCO_STATUS	If IA32_MCG_CAP.CNT >0
402H	1026	IA32_MCO_ADDR <sup>1</sup>	MCO_ADDR	If IA32_MCG_CAP.CNT >0
403H	1027	IA32_MCO_MISC	MCO_MISC	If IA32_MCG_CAP.CNT >0
404H	1028	IA32_MC1_CTL	MC1_CTL	If IA32_MCG_CAP.CNT >1
405H	1029	IA32_MC1_STATUS	MC1_STATUS	If IA32_MCG_CAP.CNT >1
406H	1030	IA32_MC1_ADDR <sup>2</sup>	MC1_ADDR	If IA32_MCG_CAP.CNT >1
407H	1031	IA32_MC1_MISC	MC1_MISC	If IA32_MCG_CAP.CNT >1
408H	1032	IA32_MC2_CTL	MC2_CTL	If IA32_MCG_CAP.CNT >2
409H	1033	IA32_MC2_STATUS	MC2_STATUS	If IA32_MCG_CAP.CNT >2
40AH	1034	IA32_MC2_ADDR <sup>1</sup>	MC2_ADDR	If IA32_MCG_CAP.CNT >2
40BH	1035	IA32_MC2_MISC	MC2_MISC	If IA32_MCG_CAP.CNT >2
40CH	1036	IA32_MC3_CTL	MC3_CTL	If IA32_MCG_CAP.CNT >3
40DH	1037	IA32_MC3_STATUS	MC3_STATUS	If IA32_MCG_CAP.CNT >3
40EH	1038	IA32_MC3_ADDR <sup>1</sup>	MC3_ADDR	If IA32_MCG_CAP.CNT >3
40FH	1039	IA32_MC3_MISC	MC3_MISC	If IA32_MCG_CAP.CNT >3
410H	1040	IA32_MC4_CTL	MC4_CTL	If IA32_MCG_CAP.CNT >4
411H	1041	IA32_MC4_STATUS	MC4_STATUS	If IA32_MCG_CAP.CNT >4
412H	1042	IA32_MC4_ADDR <sup>1</sup>	MC4_ADDR	If IA32_MCG_CAP.CNT >4
413H	1043	IA32_MC4_MISC	MC4_MISC	If IA32_MCG_CAP.CNT >4
414H	1044	IA32_MC5_CTL	MC5_CTL	If IA32_MCG_CAP.CNT >5
415H	1045	IA32_MC5_STATUS	MC5_STATUS	If IA32_MCG_CAP.CNT >5
416H	1046	IA32_MC5_ADDR <sup>1</sup>	MC5_ADDR	If IA32_MCG_CAP.CNT >5
417H	1047	IA32_MC5_MISC	MC5_MISC	If IA32_MCG_CAP.CNT >5
418H	1048	IA32_MC6_CTL	MC6_CTL	If IA32_MCG_CAP.CNT >6
419H	1049	IA32_MC6_STATUS	MC6_STATUS	If IA32_MCG_CAP.CNT >6
41AH	1050	IA32_MC6_ADDR <sup>1</sup>	MC6_ADDR	If IA32_MCG_CAP.CNT >6
41BH	1051	IA32_MC6_MISC	MC6_MISC	If IA32_MCG_CAP.CNT >6
41CH	1052	IA32_MC7_CTL	MC7_CTL	If IA32_MCG_CAP.CNT >7
41DH	1053	IA32_MC7_STATUS	MC7_STATUS	If IA32_MCG_CAP.CNT >7
41EH	1054	IA32_MC7_ADDR <sup>1</sup>	MC7_ADDR	If IA32_MCG_CAP.CNT >7
41FH	1055	IA32_MC7_MISC	MC7_MISC	If IA32_MCG_CAP.CNT >7

Table 35-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
420H	1056	IA32_MC8_CTL	MC8_CTL	If IA32_MCG_CAP.CNT >8
421H	1057	IA32_MC8_STATUS	MC8_STATUS	If IA32_MCG_CAP.CNT >8
422H	1058	IA32_MC8_ADDR <sup>7</sup>	MC8_ADDR	If IA32_MCG_CAP.CNT >8
423H	1059	IA32_MC8_MISC	MC8_MISC	If IA32_MCG_CAP.CNT >8
424H	1060	IA32_MC9_CTL	MC9_CTL	If IA32_MCG_CAP.CNT >9
425H	1061	IA32_MC9_STATUS	MC9_STATUS	If IA32_MCG_CAP.CNT >9
426H	1062	IA32_MC9_ADDR <sup>7</sup>	MC9_ADDR	If IA32_MCG_CAP.CNT >9
427H	1063	IA32_MC9_MISC	MC9_MISC	If IA32_MCG_CAP.CNT >9
428H	1064	IA32_MC10_CTL	MC10_CTL	If IA32_MCG_CAP.CNT >10
429H	1065	IA32_MC10_STATUS	MC10_STATUS	If IA32_MCG_CAP.CNT >10
42AH	1066	IA32_MC10_ADDR <sup>7</sup>	MC10_ADDR	If IA32_MCG_CAP.CNT >10
42BH	1067	IA32_MC10_MISC	MC10_MISC	If IA32_MCG_CAP.CNT >10
42CH	1068	IA32_MC11_CTL	MC11_CTL	If IA32_MCG_CAP.CNT >11
42DH	1069	IA32_MC11_STATUS	MC11_STATUS	If IA32_MCG_CAP.CNT >11
42EH	1070	IA32_MC11_ADDR <sup>7</sup>	MC11_ADDR	If IA32_MCG_CAP.CNT >11
42FH	1071	IA32_MC11_MISC	MC11_MISC	If IA32_MCG_CAP.CNT >11
430H	1072	IA32_MC12_CTL	MC12_CTL	If IA32_MCG_CAP.CNT >12
431H	1073	IA32_MC12_STATUS	MC12_STATUS	If IA32_MCG_CAP.CNT >12
432H	1074	IA32_MC12_ADDR <sup>7</sup>	MC12_ADDR	If IA32_MCG_CAP.CNT >12
433H	1075	IA32_MC12_MISC	MC12_MISC	If IA32_MCG_CAP.CNT >12
434H	1076	IA32_MC13_CTL	MC13_CTL	If IA32_MCG_CAP.CNT >13
435H	1077	IA32_MC13_STATUS	MC13_STATUS	If IA32_MCG_CAP.CNT >13
436H	1078	IA32_MC13_ADDR <sup>7</sup>	MC13_ADDR	If IA32_MCG_CAP.CNT >13
437H	1079	IA32_MC13_MISC	MC13_MISC	If IA32_MCG_CAP.CNT >13
438H	1080	IA32_MC14_CTL	MC14_CTL	If IA32_MCG_CAP.CNT >14
439H	1081	IA32_MC14_STATUS	MC14_STATUS	If IA32_MCG_CAP.CNT >14
43AH	1082	IA32_MC14_ADDR <sup>7</sup>	MC14_ADDR	If IA32_MCG_CAP.CNT >14
43BH	1083	IA32_MC14_MISC	MC14_MISC	If IA32_MCG_CAP.CNT >14
43CH	1084	IA32_MC15_CTL	MC15_CTL	If IA32_MCG_CAP.CNT >15
43DH	1085	IA32_MC15_STATUS	MC15_STATUS	If IA32_MCG_CAP.CNT >15
43EH	1086	IA32_MC15_ADDR <sup>7</sup>	MC15_ADDR	If IA32_MCG_CAP.CNT >15
43FH	1087	IA32_MC15_MISC	MC15_MISC	If IA32_MCG_CAP.CNT >15
440H	1088	IA32_MC16_CTL	MC16_CTL	If IA32_MCG_CAP.CNT >16
441H	1089	IA32_MC16_STATUS	MC16_STATUS	If IA32_MCG_CAP.CNT >16
442H	1090	IA32_MC16_ADDR <sup>7</sup>	MC16_ADDR	If IA32_MCG_CAP.CNT >16
443H	1091	IA32_MC16_MISC	MC16_MISC	If IA32_MCG_CAP.CNT >16
444H	1092	IA32_MC17_CTL	MC17_CTL	If IA32_MCG_CAP.CNT >17

**Table 35-2. IA-32 Architectural MSRs (Contd.)**

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
445H	1093	IA32_MC17_STATUS	MC17_STATUS	If IA32_MCG_CAP.CNT >17
446H	1094	IA32_MC17_ADDR <sup>7</sup>	MC17_ADDR	If IA32_MCG_CAP.CNT >17
447H	1095	IA32_MC17_MISC	MC17_MISC	If IA32_MCG_CAP.CNT >17
448H	1096	IA32_MC18_CTL	MC18_CTL	If IA32_MCG_CAP.CNT >18
449H	1097	IA32_MC18_STATUS	MC18_STATUS	If IA32_MCG_CAP.CNT >18
44AH	1098	IA32_MC18_ADDR <sup>7</sup>	MC18_ADDR	If IA32_MCG_CAP.CNT >18
44BH	1099	IA32_MC18_MISC	MC18_MISC	If IA32_MCG_CAP.CNT >18
44CH	1100	IA32_MC19_CTL	MC19_CTL	If IA32_MCG_CAP.CNT >19
44DH	1101	IA32_MC19_STATUS	MC19_STATUS	If IA32_MCG_CAP.CNT >19
44EH	1102	IA32_MC19_ADDR <sup>7</sup>	MC19_ADDR	If IA32_MCG_CAP.CNT >19
44FH	1103	IA32_MC19_MISC	MC19_MISC	If IA32_MCG_CAP.CNT >19
450H	1104	IA32_MC20_CTL	MC20_CTL	If IA32_MCG_CAP.CNT >20
451H	1105	IA32_MC20_STATUS	MC20_STATUS	If IA32_MCG_CAP.CNT >20
452H	1106	IA32_MC20_ADDR <sup>7</sup>	MC20_ADDR	If IA32_MCG_CAP.CNT >20
453H	1107	IA32_MC20_MISC	MC20_MISC	If IA32_MCG_CAP.CNT >20
454H	1108	IA32_MC21_CTL	MC21_CTL	If IA32_MCG_CAP.CNT >21
455H	1109	IA32_MC21_STATUS	MC21_STATUS	If IA32_MCG_CAP.CNT >21
456H	1110	IA32_MC21_ADDR <sup>7</sup>	MC21_ADDR	If IA32_MCG_CAP.CNT >21
457H	1111	IA32_MC21_MISC	MC21_MISC	If IA32_MCG_CAP.CNT >21
458H		IA32_MC22_CTL	MC22_CTL	If IA32_MCG_CAP.CNT >22
459H		IA32_MC22_STATUS	MC22_STATUS	If IA32_MCG_CAP.CNT >22
45AH		IA32_MC22_ADDR <sup>7</sup>	MC22_ADDR	If IA32_MCG_CAP.CNT >22
45BH		IA32_MC22_MISC	MC22_MISC	If IA32_MCG_CAP.CNT >22
45CH		IA32_MC23_CTL	MC23_CTL	If IA32_MCG_CAP.CNT >23
45DH		IA32_MC23_STATUS	MC23_STATUS	If IA32_MCG_CAP.CNT >23
45EH		IA32_MC23_ADDR <sup>7</sup>	MC23_ADDR	If IA32_MCG_CAP.CNT >23
45FH		IA32_MC23_MISC	MC23_MISC	If IA32_MCG_CAP.CNT >23
460H		IA32_MC24_CTL	MC24_CTL	If IA32_MCG_CAP.CNT >24
461H		IA32_MC24_STATUS	MC24_STATUS	If IA32_MCG_CAP.CNT >24
462H		IA32_MC24_ADDR <sup>7</sup>	MC24_ADDR	If IA32_MCG_CAP.CNT >24
463H		IA32_MC24_MISC	MC24_MISC	If IA32_MCG_CAP.CNT >24
464H		IA32_MC25_CTL	MC25_CTL	If IA32_MCG_CAP.CNT >25
465H		IA32_MC25_STATUS	MC25_STATUS	If IA32_MCG_CAP.CNT >25
466H		IA32_MC25_ADDR <sup>7</sup>	MC25_ADDR	If IA32_MCG_CAP.CNT >25
467H		IA32_MC25_MISC	MC25_MISC	If IA32_MCG_CAP.CNT >25
468H		IA32_MC26_CTL	MC26_CTL	If IA32_MCG_CAP.CNT >26
469H		IA32_MC26_STATUS	MC26_STATUS	If IA32_MCG_CAP.CNT >26

Table 35-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
46AH		IA32_MC26_ADDR <sup>1</sup>	MC26_ADDR	If IA32_MCG_CAP.CNT > 26
46BH		IA32_MC26_MISC	MC26_MISC	If IA32_MCG_CAP.CNT > 26
46CH		IA32_MC27_CTL	MC27_CTL	If IA32_MCG_CAP.CNT > 27
46DH		IA32_MC27_STATUS	MC27_STATUS	If IA32_MCG_CAP.CNT > 27
46EH		IA32_MC27_ADDR <sup>1</sup>	MC27_ADDR	If IA32_MCG_CAP.CNT > 27
46FH		IA32_MC27_MISC	MC27_MISC	If IA32_MCG_CAP.CNT > 27
470H		IA32_MC28_CTL	MC28_CTL	If IA32_MCG_CAP.CNT > 28
471H		IA32_MC28_STATUS	MC28_STATUS	If IA32_MCG_CAP.CNT > 28
472H		IA32_MC28_ADDR <sup>1</sup>	MC28_ADDR	If IA32_MCG_CAP.CNT > 28
473H		IA32_MC28_MISC	MC28_MISC	If IA32_MCG_CAP.CNT > 28
480H	1152	IA32_VMX_BASIC	<b>Reporting Register of Basic VMX Capabilities (R/O)</b> See Appendix A.1, "Basic VMX Information."	If CPUID.01H:ECX.[5] = 1
481H	1153	IA32_VMX_PINBASED_CTL	<b>Capability Reporting Register of Pin-based VM-execution Controls (R/O)</b> See Appendix A.3.1, "Pin-Based VM-Execution Controls."	If CPUID.01H:ECX.[5] = 1
482H	1154	IA32_VMX_PROCBASED_CTL	<b>Capability Reporting Register of Primary Processor-based VM-execution Controls (R/O)</b> See Appendix A.3.2, "Primary Processor-Based VM-Execution Controls."	If CPUID.01H:ECX.[5] = 1
483H	1155	IA32_VMX_EXIT_CTL	<b>Capability Reporting Register of VM-exit Controls (R/O)</b> See Appendix A.4, "VM-Exit Controls."	If CPUID.01H:ECX.[5] = 1
484H	1156	IA32_VMX_ENTRY_CTL	<b>Capability Reporting Register of VM-entry Controls (R/O)</b> See Appendix A.5, "VM-Entry Controls."	If CPUID.01H:ECX.[5] = 1
485H	1157	IA32_VMX_MISC	<b>Reporting Register of Miscellaneous VMX Capabilities (R/O)</b> See Appendix A.6, "Miscellaneous Data."	If CPUID.01H:ECX.[5] = 1
486H	1158	IA32_VMX_CR0_FIXED0	<b>Capability Reporting Register of CR0 Bits Fixed to 0 (R/O)</b> See Appendix A.7, "VMX-Fixed Bits in CR0."	If CPUID.01H:ECX.[5] = 1
487H	1159	IA32_VMX_CR0_FIXED1	<b>Capability Reporting Register of CR0 Bits Fixed to 1 (R/O)</b> See Appendix A.7, "VMX-Fixed Bits in CR0."	If CPUID.01H:ECX.[5] = 1
488H	1160	IA32_VMX_CR4_FIXED0	<b>Capability Reporting Register of CR4 Bits Fixed to 0 (R/O)</b> See Appendix A.8, "VMX-Fixed Bits in CR4."	If CPUID.01H:ECX.[5] = 1



**Table 35-2. IA-32 Architectural MSRs (Contd.)**

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
489H	1161	IA32_VMX_CR4_FIXED1	<b>Capability Reporting Register of CR4 Bits Fixed to 1 (R/O)</b> See Appendix A.8, "VMX-Fixed Bits in CR4."	If CPUID.01H:ECX.[5] = 1
48AH	1162	IA32_VMX_VMCS_ENUM	<b>Capability Reporting Register of VMCS Field Enumeration (R/O)</b> See Appendix A.9, "VMCS Enumeration."	If CPUID.01H:ECX.[5] = 1
48BH	1163	IA32_VMX_PROCBASED_CTL2	<b>Capability Reporting Register of Secondary Processor-based VM-execution Controls (R/O)</b> See Appendix A.3.3, "Secondary Processor-Based VM-Execution Controls."	If ( CPUID.01H:ECX.[5] && IA32_VMX_PROCBASED_CTL2[63] )
48CH	1164	IA32_VMX_EPT_VPID_CAP	<b>Capability Reporting Register of EPT and VPID (R/O)</b> See Appendix A.10, "VPID and EPT Capabilities."	If ( CPUID.01H:ECX.[5] && IA32_VMX_PROCBASED_CTL2[63] && ( IA32_VMX_PROCBASED_CTL2[33]    IA32_VMX_PROCBASED_CTL2[37] ) )
48DH	1165	IA32_VMX_TRUE_PINBASED_CTL2	<b>Capability Reporting Register of Pin-based VM-execution Flex Controls (R/O)</b> See Appendix A.3.1, "Pin-Based VM-Execution Controls."	If ( CPUID.01H:ECX.[5] = 1 && IA32_VMX_BASIC[55] )
48EH	1166	IA32_VMX_TRUE_PROCBASED_CTL2	<b>Capability Reporting Register of Primary Processor-based VM-execution Flex Controls (R/O)</b> See Appendix A.3.2, "Primary Processor-Based VM-Execution Controls."	If ( CPUID.01H:ECX.[5] = 1 && IA32_VMX_BASIC[55] )
48FH	1167	IA32_VMX_TRUE_EXIT_CTL2	<b>Capability Reporting Register of VM-exit Flex Controls (R/O)</b> See Appendix A.4, "VM-Exit Controls."	If ( CPUID.01H:ECX.[5] = 1 && IA32_VMX_BASIC[55] )
490H	1168	IA32_VMX_TRUE_ENTRY_CTL2	<b>Capability Reporting Register of VM-entry Flex Controls (R/O)</b> See Appendix A.5, "VM-Entry Controls."	If ( CPUID.01H:ECX.[5] = 1 && IA32_VMX_BASIC[55] )
491H	1169	IA32_VMX_VMFUNC	<b>Capability Reporting Register of VM-function Controls (R/O)</b>	If ( CPUID.01H:ECX.[5] = 1 && IA32_VMX_BASIC[55] )
4C1H	1217	IA32_A_PMC0	Full Width Writable IA32_PMC0 Alias (R/W)	(If CPUID.0AH: EAX[15:8] > 0) && IA32_PERF_CAPABILITIES[13] = 1
4C2H	1218	IA32_A_PMC1	Full Width Writable IA32_PMC1 Alias (R/W)	(If CPUID.0AH: EAX[15:8] > 1) && IA32_PERF_CAPABILITIES[13] = 1

Table 35-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
4C3H	1219	IA32_A_PMC2	Full Width Writable IA32_PMC2 Alias (R/W)	(If CPUID.0AH: EAX[15:8] > 2) && IA32_PERF_CAPABILITIES[13] = 1
4C4H	1220	IA32_A_PMC3	Full Width Writable IA32_PMC3 Alias (R/W)	(If CPUID.0AH: EAX[15:8] > 3) && IA32_PERF_CAPABILITIES[13] = 1
4C5H	1221	IA32_A_PMC4	Full Width Writable IA32_PMC4 Alias (R/W)	(If CPUID.0AH: EAX[15:8] > 4) && IA32_PERF_CAPABILITIES[13] = 1
4C6H	1222	IA32_A_PMC5	Full Width Writable IA32_PMC5 Alias (R/W)	(If CPUID.0AH: EAX[15:8] > 5) && IA32_PERF_CAPABILITIES[13] = 1
4C7H	1223	IA32_A_PMC6	Full Width Writable IA32_PMC6 Alias (R/W)	(If CPUID.0AH: EAX[15:8] > 6) && IA32_PERF_CAPABILITIES[13] = 1
4C8H	1224	IA32_A_PMC7	Full Width Writable IA32_PMC7 Alias (R/W)	(If CPUID.0AH: EAX[15:8] > 7) && IA32_PERF_CAPABILITIES[13] = 1
4D0H	1232	IA32_MCG_EXT_CTL	(R/W)	If IA32_MCG_CAP.LMCE_P = 1
		0	LMCE_EN.	
		63:1	Reserved.	
500H	1280	IA32_SGX_SVN_STATUS	Status and SVN Threshold of SGX Support for ACM (RO).	If CPUID.(EAX=07H, ECX=0H): EBX[2] = 1
		0	Lock.	See Section 42.11.3, "Interactions with Authenticated Code Modules (ACMs)".
		15:1	Reserved.	
		23:16	SGX_SVN_SINIT.	See Section 42.11.3, "Interactions with Authenticated Code Modules (ACMs)".
		63:24	Reserved.	

**Table 35-2. IA-32 Architectural MSRs (Contd.)**

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
560H	1376	IA32_RTIT_OUTPUT_BASE	<b>Trace Output Base Register (R/W)</b>	If ((CPUID.(EAX=07H, ECX=0);EBX[25] = 1) && (CPUID.(EAX=14H,ECX=0); ECX[0] = 1)    (CPUID.(EAX=14H,ECX=0); ECX[2] = 1) ) )
		6:0	Reserved	
		MAXPHYADDR <sup>3</sup> -1:7	Base physical address	
		63:MAXPHYADDR	<b>Reserved.</b>	
561H	1377	IA32_RTIT_OUTPUT_MASK_PTRS	<b>Trace Output Mask Pointers Register (R/W)</b>	If ((CPUID.(EAX=07H, ECX=0);EBX[25] = 1) && (CPUID.(EAX=14H,ECX=0); ECX[0] = 1)    (CPUID.(EAX=14H,ECX=0); ECX[2] = 1) ) )
		6:0	Reserved	
		31:7	MaskOrTableOffset	
		63:32	<b>Output Offset.</b>	
570H	1392	IA32_RTIT_CTL	<b>Trace Control Register (R/W)</b>	If (CPUID.(EAX=07H, ECX=0);EBX[25] = 1)
		0	<b>TraceEn</b>	
		1	<b>CYCEn</b>	If (CPUID.(EAX=07H, ECX=0);EBX[1] = 1)
		2	<b>OS</b>	
		3	<b>User</b>	
		5:4	Reserved,	
		6	<b>FabricEn</b>	If (CPUID.(EAX=07H, ECX=0);ECX[3] = 1)
		7	<b>CR3 filter</b>	
		8	<b>ToPA</b>	
		9	<b>MTCEn</b>	If (CPUID.(EAX=07H, ECX=0);EBX[3] = 1)
		10	<b>TSCEn</b>	
		11	<b>DisRETC</b>	
		12	Reserved, MBZ	
		13	<b>BranchEn</b>	
		17:14	<b>MTCFreq</b>	If (CPUID.(EAX=07H, ECX=0);EBX[3] = 1)
18	Reserved, MBZ			
22:19	<b>CYCThresh</b>	If (CPUID.(EAX=07H, ECX=0);EBX[1] = 1)		
23	Reserved, MBZ			

Table 35-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
		27:24	PSBFreq	If (CPUID.(EAX=07H, ECX=0):EBX[1] = 1)
		31:28	Reserved, MBZ	
		35:32	ADDR0_CFG	If (CPUID.(EAX=07H, ECX=1):EAX[2:0] > 0)
		39:36	ADDR1_CFG	If (CPUID.(EAX=07H, ECX=1):EAX[2:0] > 1)
		43:40	ADDR2_CFG	If (CPUID.(EAX=07H, ECX=1):EAX[2:0] > 2)
		47:44	ADDR3_CFG	If (CPUID.(EAX=07H, ECX=1):EAX[2:0] > 3)
		63:48	Reserved, MBZ.	
571H	1393	IA32_RTIT_STATUS	Tracing Status Register (R/W)	If (CPUID.(EAX=07H, ECX=0):EBX[25] = 1)
		0	FilterEn, (writes ignored)	If (CPUID.(EAX=07H, ECX=0):EBX[2] = 1)
		1	ContexEn, (writes ignored)	
		2	TriggerEn, (writes ignored)	
		3	Reserved	
		4	Error	
		5	Stopped	
		31:6	Reserved, MBZ	
		48:32	PacketByteCnt	If (CPUID.(EAX=07H, ECX=0):EBX[1] > 3)
		63:49	Reserved.	
572H	1394	IA32_RTIT_CR3_MATCH	Trace Filter CR3 Match Register (R/W)	If (CPUID.(EAX=07H, ECX=0):EBX[25] = 1)
		4:0	Reserved	
		63:5	CR3[63:5] value to match	
580H	1408	IA32_RTIT_ADDR0_A	Region 0 Start Address (R/W)	If (CPUID.(EAX=07H, ECX=1):EAX[2:0] > 0)
		47:0	Virtual Address	
		63:48	SignExt_VA	
581H	1409	IA32_RTIT_ADDR0_B	Region 0 End Address (R/W)	If (CPUID.(EAX=07H, ECX=1):EAX[2:0] > 0)
		47:0	Virtual Address	
		63:48	SignExt_VA	
582H	1410	IA32_RTIT_ADDR1_A	Region 1 Start Address (R/W)	If (CPUID.(EAX=07H, ECX=1):EAX[2:0] > 1)
		47:0	Virtual Address	
		63:48	SignExt_VA	

**Table 35-2. IA-32 Architectural MSRs (Contd.)**

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
583H	1411	IA32_RTIT_ADDR1_B	<b>Region 1 End Address (R/W)</b>	If (CPUID.(EAX=07H, ECX=1):EAX[2:0] > 1)
		47:0	Virtual Address	
		63:48	SignExt_VA	
584H	1412	IA32_RTIT_ADDR2_A	<b>Region 2 Start Address (R/W)</b>	If (CPUID.(EAX=07H, ECX=1):EAX[2:0] > 2)
		47:0	Virtual Address	
		63:48	SignExt_VA	
585H	1413	IA32_RTIT_ADDR2_B	<b>Region 2 End Address (R/W)</b>	If (CPUID.(EAX=07H, ECX=1):EAX[2:0] > 2)
		47:0	Virtual Address	
		63:48	SignExt_VA	
586H	1414	IA32_RTIT_ADDR3_A	<b>Region 3 Start Address (R/W)</b>	If (CPUID.(EAX=07H, ECX=1):EAX[2:0] > 3)
		47:0	Virtual Address	
		63:48	SignExt_VA	
587H	1415	IA32_RTIT_ADDR3_B	<b>Region 3 End Address (R/W)</b>	If (CPUID.(EAX=07H, ECX=1):EAX[2:0] > 3)
		47:0	Virtual Address	
		63:48	SignExt_VA	
600H	1536	IA32_DS_AREA	<b>DS Save Area (R/W)</b> Points to the linear address of the first byte of the DS buffer management area, which is used to manage the BTS and PEBS buffers. See Section 18.12.4, "Debug Store (DS) Mechanism."	If (CPUID.01H:EDX.DS[21] = 1)
		63:0	The linear address of the first byte of the DS buffer management area, if IA-32e mode is active.	
		31:0	The linear address of the first byte of the DS buffer management area, if not in IA-32e mode.	
		63:32	Reserved if not in IA-32e mode.	
6E0H	1760	IA32_TSC_DEADLINE	<b>TSC Target of Local APIC's TSC Deadline Mode (R/W)</b>	If CPUID.01H:ECX.[24] = 1
770H	1904	IA32_PM_ENABLE	<b>Enable/disable HWP (R/W)</b>	If CPUID.06H:EAX.[7] = 1
		0	<b>HWP_ENABLE (R/W1-Once).</b> See Section 14.4.2, "Enabling HWP"	If CPUID.06H:EAX.[7] = 1
		63:1	Reserved.	
771H	1905	IA32_HWP_CAPABILITIES	<b>HWP Performance Range Enumeration (RO)</b>	If CPUID.06H:EAX.[7] = 1

Table 35-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
		7:0	<b>Highest_Performance</b> See Section 14.4.3, "HWP Performance Range and Dynamic Capabilities"	If CPUID.06H:EAX.[7] = 1
		15:8	<b>Guaranteed_Performance</b> See Section 14.4.3, "HWP Performance Range and Dynamic Capabilities"	If CPUID.06H:EAX.[7] = 1
		23:16	<b>Most_Efficient_Performance</b> See Section 14.4.3, "HWP Performance Range and Dynamic Capabilities"	If CPUID.06H:EAX.[7] = 1
		31:24	<b>Lowest_Performance</b> See Section 14.4.3, "HWP Performance Range and Dynamic Capabilities"	If CPUID.06H:EAX.[7] = 1
		63:32	Reserved.	
772H	1906	IA32_HWP_REQUEST_PKG	<b>Power Management Control Hints for All Logical Processors in a Package (R/W)</b>	If CPUID.06H:EAX.[11] = 1
		7:0	<b>Minimum_Performance</b> See Section 14.4.4, "Managing HWP"	If CPUID.06H:EAX.[11] = 1
		15:8	<b>Maximum_Performance</b> See Section 14.4.4, "Managing HWP"	If CPUID.06H:EAX.[11] = 1
		23:16	<b>Desired_Performance</b> See Section 14.4.4, "Managing HWP"	If CPUID.06H:EAX.[11] = 1
		31:24	<b>Energy_Performance_Preference</b> See Section 14.4.4, "Managing HWP"	If CPUID.06H:EAX.[11] = 1 && CPUID.06H:EAX.[10] = 1
		41:32	<b>Activity_Window</b> See Section 14.4.4, "Managing HWP"	If CPUID.06H:EAX.[11] = 1 && CPUID.06H:EAX.[9] = 1
		63:42	Reserved.	
773H	1907	IA32_HWP_INTERRUPT	<b>Control HWP Native Interrupts (R/W)</b>	If CPUID.06H:EAX.[8] = 1
		0	<b>EN_Guaranteed_Performance_Change.</b> See Section 14.4.6, "HWP Notifications"	If CPUID.06H:EAX.[8] = 1
		1	<b>EN_Excursion_Minimum.</b> See Section 14.4.6, "HWP Notifications"	If CPUID.06H:EAX.[8] = 1
		63:2	Reserved.	
774H	1908	IA32_HWP_REQUEST	<b>Power Management Control Hints to a Logical Processor (R/W)</b>	If CPUID.06H:EAX.[7] = 1
		7:0	<b>Minimum_Performance</b> See Section 14.4.4, "Managing HWP"	If CPUID.06H:EAX.[7] = 1
		15:8	<b>Maximum_Performance</b> See Section 14.4.4, "Managing HWP"	If CPUID.06H:EAX.[7] = 1

**Table 35-2. IA-32 Architectural MSRs (Contd.)**

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
		23:16	<b>Desired_Performance</b> See Section 14.4.4, "Managing HWP"	If CPUID.06H:EAX.[7] = 1
		31:24	<b>Energy_Performance_Preference</b> See Section 14.4.4, "Managing HWP"	If CPUID.06H:EAX.[7] = 1 && CPUID.06H:EAX.[10] = 1
		41:32	<b>Activity_Window</b> See Section 14.4.4, "Managing HWP"	If CPUID.06H:EAX.[7] = 1 && CPUID.06H:EAX.[9] = 1
		42	<b>Package_Control</b> See Section 14.4.4, "Managing HWP"	If CPUID.06H:EAX.[7] = 1 && CPUID.06H:EAX.[11] = 1
		63:43	Reserved.	
777H	1911	IA32_HWP_STATUS	<b>Log bits indicating changes to Guaranteed &amp; excursions to Minimum (R/W)</b>	If CPUID.06H:EAX.[7] = 1
		0	<b>Guaranteed_Performance_Change (R/WCO).</b> See Section 14.4.5, "HWP Feedback"	If CPUID.06H:EAX.[7] = 1
		1	Reserved.	
		2	<b>Excursion_To_Minimum (R/WCO).</b> See Section 14.4.5, "HWP Feedback"	If CPUID.06H:EAX.[7] = 1
		63:3	Reserved.	
802H	2050	IA32_X2APIC_APICID	<b>x2APIC ID Register (R/O)</b> See x2APIC Specification	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
803H	2051	IA32_X2APIC_VERSION	<b>x2APIC Version Register (R/O)</b>	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
808H	2056	IA32_X2APIC_TPR	<b>x2APIC Task Priority Register (R/W)</b>	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
80AH	2058	IA32_X2APIC_PPR	<b>x2APIC Processor Priority Register (R/O)</b>	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
80BH	2059	IA32_X2APIC_EOI	<b>x2APIC EOI Register (W/O)</b>	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
80DH	2061	IA32_X2APIC_LDR	<b>x2APIC Logical Destination Register (R/O)</b>	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
80FH	2063	IA32_X2APIC_SIVR	<b>x2APIC Spurious Interrupt Vector Register (R/W)</b>	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1

Table 35-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
810H	2064	IA32_X2APIC_ISR0	x2APIC In-Service Register Bits 31:0 (R/O)	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
811H	2065	IA32_X2APIC_ISR1	x2APIC In-Service Register Bits 63:32 (R/O)	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
812H	2066	IA32_X2APIC_ISR2	x2APIC In-Service Register Bits 95:64 (R/O)	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
813H	2067	IA32_X2APIC_ISR3	x2APIC In-Service Register Bits 127:96 (R/O)	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
814H	2068	IA32_X2APIC_ISR4	x2APIC In-Service Register Bits 159:128 (R/O)	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
815H	2069	IA32_X2APIC_ISR5	x2APIC In-Service Register Bits 191:160 (R/O)	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
816H	2070	IA32_X2APIC_ISR6	x2APIC In-Service Register Bits 223:192 (R/O)	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
817H	2071	IA32_X2APIC_ISR7	x2APIC In-Service Register Bits 255:224 (R/O)	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
818H	2072	IA32_X2APIC_TMR0	x2APIC Trigger Mode Register Bits 31:0 (R/O)	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
819H	2073	IA32_X2APIC_TMR1	x2APIC Trigger Mode Register Bits 63:32 (R/O)	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
81AH	2074	IA32_X2APIC_TMR2	x2APIC Trigger Mode Register Bits 95:64 (R/O)	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
81BH	2075	IA32_X2APIC_TMR3	x2APIC Trigger Mode Register Bits 127:96 (R/O)	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
81CH	2076	IA32_X2APIC_TMR4	x2APIC Trigger Mode Register Bits 159:128 (R/O)	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
81DH	2077	IA32_X2APIC_TMR5	x2APIC Trigger Mode Register Bits 191:160 (R/O)	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
81EH	2078	IA32_X2APIC_TMR6	x2APIC Trigger Mode Register Bits 223:192 (R/O)	If (CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1)



Table 35-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
81FH	2079	IA32_X2APIC_TMR7	x2APIC Trigger Mode Register Bits 255:224 (R/O)	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
820H	2080	IA32_X2APIC_IRR0	x2APIC Interrupt Request Register Bits 31:0 (R/O)	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
821H	2081	IA32_X2APIC_IRR1	x2APIC Interrupt Request Register Bits 63:32 (R/O)	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
822H	2082	IA32_X2APIC_IRR2	x2APIC Interrupt Request Register Bits 95:64 (R/O)	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
823H	2083	IA32_X2APIC_IRR3	x2APIC Interrupt Request Register Bits 127:96 (R/O)	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
824H	2084	IA32_X2APIC_IRR4	x2APIC Interrupt Request Register Bits 159:128 (R/O)	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
825H	2085	IA32_X2APIC_IRR5	x2APIC Interrupt Request Register Bits 191:160 (R/O)	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
826H	2086	IA32_X2APIC_IRR6	x2APIC Interrupt Request Register Bits 223:192 (R/O)	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
827H	2087	IA32_X2APIC_IRR7	x2APIC Interrupt Request Register Bits 255:224 (R/O)	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
828H	2088	IA32_X2APIC_ESR	x2APIC Error Status Register (R/W)	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
82FH	2095	IA32_X2APIC_LVT_CMCI	x2APIC LVT Corrected Machine Check Interrupt Register (R/W)	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
830H	2096	IA32_X2APIC_ICR	x2APIC Interrupt Command Register (R/W)	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
832H	2098	IA32_X2APIC_LVT_TIMER	x2APIC LVT Timer Interrupt Register (R/W)	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
833H	2099	IA32_X2APIC_LVT_THERMAL	x2APIC LVT Thermal Sensor Interrupt Register (R/W)	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
834H	2100	IA32_X2APIC_LVT_PMI	x2APIC LVT Performance Monitor Interrupt Register (R/W)	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1

Table 35-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
835H	2101	IA32_X2APIC_LVT_LINT0	<b>x2APIC LVT LINT0 Register (R/W)</b>	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
836H	2102	IA32_X2APIC_LVT_LINT1	<b>x2APIC LVT LINT1 Register (R/W)</b>	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
837H	2103	IA32_X2APIC_LVT_ERROR	<b>x2APIC LVT Error Register (R/W)</b>	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
838H	2104	IA32_X2APIC_INIT_COUNT	<b>x2APIC Initial Count Register (R/W)</b>	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
839H	2105	IA32_X2APIC_CUR_COUNT	<b>x2APIC Current Count Register (R/O)</b>	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
83EH	2110	IA32_X2APIC_DIV_CONF	<b>x2APIC Divide Configuration Register (R/W)</b>	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
83FH	2111	IA32_X2APIC_SELF_IPI	<b>x2APIC Self IPI Register (W/O)</b>	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
C80H	3200	IA32_DEBUG_INTERFACE	<b>Silicon Debug Feature Control (R/W)</b>	If CPUID.01H:ECX.[11] = 1
		0	<b>Enable (R/W)</b> BIOS set 1 to enable Silicon debug features. Default is 0	If CPUID.01H:ECX.[11] = 1
		29:1	Reserved.	
		30	<b>Lock (R/W):</b> If 1, locks any further change to the MSR. The lock bit is set automatically on the first SMI assertion even if not explicitly set by BIOS. Default is 0.	If CPUID.01H:ECX.[11] = 1
		31	<b>Debug Occurred (R/O):</b> This “sticky bit” is set by hardware to indicate the status of bit 0. Default is 0.	If CPUID.01H:ECX.[11] = 1
		63:32	Reserved.	
C81H	3201	IA32_L3_QOS_CFG	<b>L3 QOS Configuration (R/W)</b>	If ( CPUID.(EAX=10H, ECX=1);ECX.[2] = 1 )
		0	<b>Enable (R/W)</b> Set 1 to enable L3 CAT masks and COS to operate in Code and Data Prioritization (CDP) mode	
		63:1	Reserved.	
C8DH	3213	IA32_QM_EVTSEL	<b>Monitoring Event Select Register (R/W)</b>	If ( CPUID.(EAX=07H, ECX=0);EBX.[12] = 1 )
		7:0	<b>Event ID:</b> ID of a supported monitoring event to report via IA32_QM_CTR.	

**Table 35-2. IA-32 Architectural MSRs (Contd.)**

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
		31:8	Reserved.	
		N+31:32	<b>Resource Monitoring ID:</b> ID for monitoring hardware to report monitored data via IA32_QM_CTR.	N = Ceil (Log <sub>2</sub> ( CPUID.(EAX= 0FH, ECX=0H).EBX[31:0] +1))
		63:N+32	Reserved.	
C8EH	3214	IA32_QM_CTR	<b>Monitoring Counter Register (R/O)</b>	If ( CPUID.(EAX=07H, ECX=0);EBX.[12] = 1 )
		61:0	<b>Resource Monitored Data</b>	
		62	<b>Unavailable:</b> If 1, indicates data for this RMID is not available or not monitored for this resource or RMID.	
		63	<b>Error:</b> If 1, indicates and unsupported RMID or event type was written to IA32_PQR_QM_EVTSEL.	
C8FH	3215	IA32_PQR_ASSOC	<b>Resource Association Register (R/W)</b>	If ( ( CPUID.(EAX=07H, ECX=0);EBX[12] = 1) or ( CPUID.(EAX=07H, ECX=0);EBX[15] = 1 ) )
		N-1:0	<b>Resource Monitoring ID (R/W):</b> ID for monitoring hardware to track internal operation, e.g. memory access.	N = Ceil (Log <sub>2</sub> ( CPUID.(EAX= 0FH, ECX=0H).EBX[31:0] +1))
		31:N	Reserved	
		63:32	<b>COS (R/W).</b> The class of service (COS) to enforce (on writes); returns the current COS when read.	If ( CPUID.(EAX=07H, ECX=0);EBX.[15] = 1 )
C90H - D8FH		Reserved MSR Address Space for CAT Mask Registers	<b>See Section 17.17.3.1, “Enumeration and Detection Support of Cache Allocation Technology”</b>	
C90H	3216	IA32_L3_MASK_0	<b>L3 CAT Mask for COS0 (R/w)</b>	If ( CPUID.(EAX=10H, ECX=0H);EBX[1] != 0)
		31:0	<b>Capacity Bit Mask (R/w)</b>	
		63:32	Reserved.	
C90H+ n	3216+n	IA32_L3_MASK_n	<b>L3 CAT Mask for COSn (R/W)</b>	n = CPUID.(EAX=10H, ECX=1H);EDX[15:0]
		31:0	<b>Capacity Bit Mask (R/w)</b>	
		63:32	Reserved.	
D10H - D4FH		Reserved MSR Address Space for L2 CAT Mask Registers	<b>See Section 17.17.3.1, “Enumeration and Detection Support of Cache Allocation Technology”</b>	

Table 35-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
D10H	3344	IA32_L2_MASK_0	L2 CAT Mask for COS0 (R/W)	If (CPUID.(EAX=10H, ECX=0H):EBX[2] != 0)
		31:0	Capacity Bit Mask (R/W)	
		63:32	Reserved.	
D10H+n	3344+n	IA32_L2_MASK_n	L2 CAT Mask for COSn (R/W)	n = CPUID.(EAX=10H, ECX=2H):EDX[15:0]
		31:0	Capacity Bit Mask (R/W)	
		63:32	Reserved.	
D90H	3472	IA32_BNDCFGS	Supervisor State of MPX Configuration. (R/W)	If (CPUID.(EAX=07H, ECX=0H):EBX[14] = 1)
		0	EN: Enable Intel MPX in supervisor mode	
		1	BNDPRESERVE: Preserve the bounds registers for near branch instructions in the absence of the BND prefix	
		11:2	Reserved, must be 0	
		63:12	Base Address of Bound Directory.	
DA0H	3488	IA32_XSS	Extended Supervisor State Mask (R/W)	If (CPUID.(0DH, 1):EAX.[3] = 1)
		7:0	Reserved	
		8	Trace Packet Configuration State (R/W)	
		63:9	Reserved.	
DB0H	3504	IA32_PKG_HDC_CTL	Package Level Enable/disable HDC (R/W)	If CPUID.06H:EAX.[13] = 1
		0	HDC_Pkg_Enable (R/W) Force HDC idling or wake up HDC-idled logical processors in the package. See Section 14.5.2, "Package level Enabling HDC"	If CPUID.06H:EAX.[13] = 1
		63:1	Reserved.	
DB1H	3505	IA32_PM_CTL1	Enable/disable HWP (R/W)	If CPUID.06H:EAX.[13] = 1
		0	HDC_Allow_Block (R/W) Allow/Block this logical processor for package level HDC control. See Section 14.5.3	If CPUID.06H:EAX.[13] = 1
		63:1	Reserved.	
DB2H	3506	IA32_THREAD_STALL	Per-Logical_Processor HDC Idle Residency (R/O)	If CPUID.06H:EAX.[13] = 1
		63:0	Stall_Cycle_Cnt (R/W) Stalled cycles due to HDC forced idle on this logical processor. See Section 14.5.4.1	If CPUID.06H:EAX.[13] = 1

**Table 35-2. IA-32 Architectural MSRs (Contd.)**

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
4000_0000H - 4000_00FFH		Reserved MSR Address Space	All existing and future processors will not implement MSR in this range.	
C000_0080H		IA32_EFER	<b>Extended Feature Enables</b>	If (CPUID.80000001H:EDX.[20])    CPUID.80000001H:EDX.[29])
	0		<b>SYSCALL Enable: IA32_EFER.SCE (R/W)</b> Enables SYSCALL/SYSRET instructions in 64-bit mode.	
	7:1		Reserved.	
	8		<b>IA-32e Mode Enable: IA32_EFER.LME (R/W)</b> Enables IA-32e mode operation.	
	9		Reserved.	
	10		<b>IA-32e Mode Active: IA32_EFER.LMA (R)</b> Indicates IA-32e mode is active when set.	
	11		<b>Execute Disable Bit Enable: IA32_EFER.NXE (R/W)</b>	
	63:12		Reserved.	
C000_0081H		IA32_STAR	<b>System Call Target Address (R/W)</b>	If CPUID.80000001:EDX.[29] = 1
C000_0082H		IA32_LSTAR	<b>IA-32e Mode System Call Target Address (R/W)</b>	If CPUID.80000001:EDX.[29] = 1
C000_0084H		IA32_FMASK	<b>System Call Flag Mask (R/W)</b>	If CPUID.80000001:EDX.[29] = 1
C000_0100H		IA32_FS_BASE	<b>Map of BASE Address of FS (R/W)</b>	If CPUID.80000001:EDX.[29] = 1
C000_0101H		IA32_GS_BASE	<b>Map of BASE Address of GS (R/W)</b>	If CPUID.80000001:EDX.[29] = 1
C000_0102H		IA32_KERNEL_GS_BASE	<b>Swap Target of BASE Address of GS (R/W)</b>	If CPUID.80000001:EDX.[29] = 1
C000_0103H		IA32_TSC_AUX	Auxiliary TSC (Rw)	If CPUID.80000001H:EDX[27] = 1
	31:0		AUX: Auxiliary signature of TSC	
	63:32		Reserved.	

**NOTES:**

1. In processors based on Intel NetBurst® microarchitecture, MSR addresses 180H-197H are supported, software must treat them as model-specific. Starting with Intel Core Duo processors, MSR addresses 180H-185H, 188H-197H are reserved.
2. The \*\_ADDR MSRs may or may not be present; this depends on flag settings in IA32\_MCI\_STATUS. See Section 15.3.2.3 and Section 15.3.2.4 for more information.
3. MAXPHYADDR is reported by CPUID.80000008H:EAX[7:0].

## 35.2 MSRS IN THE INTEL® CORE™ 2 PROCESSOR FAMILY

Table 35-3 lists model-specific registers (MSRs) for Intel Core 2 processor family and for Intel Xeon processors based on Intel Core microarchitecture, architectural MSR addresses are also included in Table 35-3. These processors have a CPUID signature with DisplayFamily\_DisplayModel of 06\_0FH, see Table 35-1.

MSRs listed in Table 35-2 and Table 35-3 are also supported by processors based on the Enhanced Intel Core microarchitecture. Processors based on the Enhanced Intel Core microarchitecture have the CPUID signature DisplayFamily\_DisplayModel of 06\_17H.

The column “Shared/Unique” applies to multi-core processors based on Intel Core microarchitecture. “Unique” means each processor core has a separate MSR, or a bit field in an MSR governs only a core independently. “Shared” means the MSR or the bit field in an MSR address governs the operation of both processor cores.

**Table 35-3. MSRs in Processors Based on Intel® Core™ Microarchitecture**

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
0H	0	IA32_P5_MC_ADDR	Unique	See Section 35.22, “MSRs in Pentium Processors.”
1H	1	IA32_P5_MC_TYPE	Unique	See Section 35.22, “MSRs in Pentium Processors.”
6H	6	IA32_MONITOR_FILTER_SIZE	Unique	See Section 8.10.5, “Monitor/Mwait Address Range Determination,” and Table 35-2.
10H	16	IA32_TIME_STAMP_COUNTER	Unique	See Section 17.15, “Time-Stamp Counter,” and see Table 35-2.
17H	23	IA32_PLATFORM_ID	Shared	<b>Platform ID (R)</b> See Table 35-2.
17H	23	MSR_PLATFORM_ID	Shared	<b>Model Specific Platform ID (R)</b>
		7:0		Reserved.
		12:8		<b>Maximum Qualified Ratio (R)</b> The maximum allowed bus ratio.
		49:13		Reserved.
		52:50		See Table 35-2.
		63:53		Reserved.
1BH	27	IA32_APIC_BASE	Unique	See Section 10.4.4, “Local APIC Status and Location,” and Table 35-2.
2AH	42	MSR_EBL_CR_POWERON	Shared	<b>Processor Hard Power-On Configuration (R/W)</b> Enables and disables processor features; (R) indicates current processor configuration.
		0		Reserved.

**Table 35-3. MSRs in Processors Based on Intel® Core™ Microarchitecture (Contd.)**

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
		1		<b>Data Error Checking Enable (R/W)</b> 1 = Enabled; 0 = Disabled Note: Not all processor implements R/W.
		2		<b>Response Error Checking Enable (R/W)</b> 1 = Enabled; 0 = Disabled Note: Not all processor implements R/W.
		3		<b>MCERR# Drive Enable (R/W)</b> 1 = Enabled; 0 = Disabled Note: Not all processor implements R/W.
		4		<b>Address Parity Enable (R/W)</b> 1 = Enabled; 0 = Disabled Note: Not all processor implements R/W.
		5		Reserved.
		6		Reserved.
		7		<b>BINIT# Driver Enable (R/W)</b> 1 = Enabled; 0 = Disabled Note: Not all processor implements R/W.
		8		<b>Output Tri-state Enabled (R/O)</b> 1 = Enabled; 0 = Disabled
		9		<b>Execute BIST (R/O)</b> 1 = Enabled; 0 = Disabled
		10		<b>MCERR# Observation Enabled (R/O)</b> 1 = Enabled; 0 = Disabled
		11		Intel TXT Capable Chipset. (R/O) 1 = Present; 0 = Not Present
		12		<b>BINIT# Observation Enabled (R/O)</b> 1 = Enabled; 0 = Disabled
		13		Reserved.
		14		<b>1 MByte Power on Reset Vector (R/O)</b> 1 = 1 MByte; 0 = 4 GBytes
		15		Reserved.
		17:16		<b>APIC Cluster ID (R/O)</b>
		18		<b>N/2 Non-Integer Bus Ratio (R/O)</b> 0 = Integer ratio; 1 = Non-integer ratio
		19		Reserved.
		21:20		<b>Symmetric Arbitration ID (R/O)</b>
		26:22		<b>Integer Bus Frequency Ratio (R/O)</b>

Table 35-3. MSRs in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
3AH	58	MSR_FEATURE_CONTROL	Unique	<b>Control Features in Intel 64 Processor (R/W)</b> See Table 35-2.
		3	Unique	<b>SMRR Enable (R/WL)</b> When this bit is set and the lock bit is set makes the SMRR_PHYS_BASE and SMRR_PHYS_MASK registers read visible and writeable while in SMM.
40H	64	MSR_LASTBRANCH_0_FROM_IP	Unique	<b>Last Branch Record 0 From IP (R/W)</b> One of four pairs of last branch record registers on the last branch record stack. The From_IP part of the stack contains pointers to the <b>source instruction</b> . See also: <ul style="list-style-type: none"> <li>Last Branch Record Stack TOS at 1C9H</li> <li>Section 17.5</li> </ul>
41H	65	MSR_LASTBRANCH_1_FROM_IP	Unique	<b>Last Branch Record 1 From IP (R/W)</b> See description of MSR_LASTBRANCH_0_FROM_IP.
42H	66	MSR_LASTBRANCH_2_FROM_IP	Unique	<b>Last Branch Record 2 From IP (R/W)</b> See description of MSR_LASTBRANCH_0_FROM_IP.
43H	67	MSR_LASTBRANCH_3_FROM_IP	Unique	<b>Last Branch Record 3 From IP (R/W)</b> See description of MSR_LASTBRANCH_0_FROM_IP.
60H	96	MSR_LASTBRANCH_0_TO_IP	Unique	<b>Last Branch Record 0 To IP (R/W)</b> One of four pairs of last branch record registers on the last branch record stack. This To_IP part of the stack contains pointers to the destination instruction.
61H	97	MSR_LASTBRANCH_1_TO_IP	Unique	<b>Last Branch Record 1 To IP (R/W)</b> See description of MSR_LASTBRANCH_0_TO_IP.
62H	98	MSR_LASTBRANCH_2_TO_IP	Unique	<b>Last Branch Record 2 To IP (R/W)</b> See description of MSR_LASTBRANCH_0_TO_IP.
63H	99	MSR_LASTBRANCH_3_TO_IP	Unique	<b>Last Branch Record 3 To IP (R/W)</b> See description of MSR_LASTBRANCH_0_TO_IP.
79H	121	IA32_BIOS_UPDT_TRIG	Unique	<b>BIOS Update Trigger Register (w)</b> See Table 35-2.
8BH	139	IA32_BIOS_SIGN_ID	Unique	<b>BIOS Update Signature ID (RO)</b> See Table 35-2.
A0H	160	MSR_SMRR_PHYSBASE	Unique	<b>System Management Mode Base Address register (wO in SMM)</b> Model-specific implementation of SMRR-like interface, read visible and write only in SMM.
		11:0		Reserved.
		31:12		PhysBase. SMRR physical Base Address.
		63:32		Reserved.



**Table 35-3. MSRs in Processors Based on Intel® Core™ Microarchitecture (Contd.)**

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
A1H	161	MSR_SMRR_PHYSMASK	Unique	<b>System Management Mode Physical Address Mask register (WO in SMM)</b> Model-specific implementation of SMRR-like interface, read visible and write only in SMM.
		10:0		Reserved.
		11		Valid. Physical address base and range mask are valid.
		31:12		PhysMask. SMRR physical address range mask.
		63:32		Reserved.
C1H	193	IA32_PMC0	Unique	<b>Performance Counter Register</b> See Table 35-2.
C2H	194	IA32_PMC1	Unique	<b>Performance Counter Register</b> See Table 35-2.
CDH	205	MSR_FSB_FREQ	Shared	<b>Scaleable Bus Speed(R0)</b> This field indicates the intended scaleable bus clock speed for processors based on Intel Core microarchitecture:
		2:0		<ul style="list-style-type: none"> <li>▪ 101B: 100 MHz (FSB 400)</li> <li>▪ 001B: 133 MHz (FSB 533)</li> <li>▪ 011B: 167 MHz (FSB 667)</li> <li>▪ 010B: 200 MHz (FSB 800)</li> <li>▪ 000B: 267 MHz (FSB 1067)</li> <li>▪ 100B: 333 MHz (FSB 1333)</li> </ul>
				133.33 MHz should be utilized if performing calculation with System Bus Speed when encoding is 001B. 166.67 MHz should be utilized if performing calculation with System Bus Speed when encoding is 011B.
				266.67 MHz should be utilized if performing calculation with System Bus Speed when encoding is 000B. 333.33 MHz should be utilized if performing calculation with System Bus Speed when encoding is 100B.
		63:3		Reserved.
CDH	205	MSR_FSB_FREQ	Shared	<b>Scaleable Bus Speed(R0)</b> This field indicates the intended scaleable bus clock speed for processors based on Enhanced Intel Core microarchitecture:
		2:0		<ul style="list-style-type: none"> <li>▪ 101B: 100 MHz (FSB 400)</li> <li>▪ 001B: 133 MHz (FSB 533)</li> <li>▪ 011B: 167 MHz (FSB 667)</li> <li>▪ 010B: 200 MHz (FSB 800)</li> <li>▪ 000B: 267 MHz (FSB 1067)</li> <li>▪ 100B: 333 MHz (FSB 1333)</li> <li>▪ 110B: 400 MHz (FSB 1600)</li> </ul>

Table 35-3. MSRs in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
				133.33 MHz should be utilized if performing calculation with System Bus Speed when encoding is 001B. 166.67 MHz should be utilized if performing calculation with System Bus Speed when encoding is 011B.
				266.67 MHz should be utilized if performing calculation with System Bus Speed when encoding is 110B. 333.33 MHz should be utilized if performing calculation with System Bus Speed when encoding is 111B.
		63:3		Reserved.
E7H	231	IA32_MPERF	Unique	<b>Maximum Performance Frequency Clock Count (RW)</b> See Table 35-2.
E8H	232	IA32_APERF	Unique	<b>Actual Performance Frequency Clock Count (RW)</b> See Table 35-2.
FEH	254	IA32_MTRRCAP	Unique	See Table 35-2.
		11	Unique	<b>SMRR Capability Using MSR OAOH and OA1H (R)</b>
11EH	281	MSR_BBL_CR_CTL3	Shared	
		0		<b>L2 Hardware Enabled (RO)</b> 1 = If the L2 is hardware-enabled 0 = Indicates if the L2 is hardware-disabled
		7:1		Reserved.
		8		<b>L2 Enabled (R/W)</b> 1 = L2 cache has been initialized 0 = Disabled (default) Until this bit is set the processor will not respond to the WBINVD instruction or the assertion of the FLUSH# input.
		22:9		Reserved.
		23		<b>L2 Not Present (RO)</b> 0 = L2 Present 1 = L2 Not Present
		63:24		Reserved.
174H	372	IA32_SYSENTER_CS	Unique	See Table 35-2.
175H	373	IA32_SYSENTER_ESP	Unique	See Table 35-2.
176H	374	IA32_SYSENTER_EIP	Unique	See Table 35-2.
179H	377	IA32_MCG_CAP	Unique	See Table 35-2.
17AH	378	IA32_MCG_STATUS	Unique	

**Table 35-3. MSRs in Processors Based on Intel® Core™ Microarchitecture (Contd.)**

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
		0		<b>RIPV</b> When set, bit indicates that the instruction addressed by the instruction pointer pushed on the stack (when the machine check was generated) can be used to restart the program. If cleared, the program cannot be reliably restarted.
		1		<b>EIPV</b> When set, bit indicates that the instruction addressed by the instruction pointer pushed on the stack (when the machine check was generated) is directly associated with the error.
		2		<b>MCIP</b> When set, bit indicates that a machine check has been generated. If a second machine check is detected while this bit is still set, the processor enters a shutdown state. Software should write this bit to 0 after processing a machine check exception.
		63:3		Reserved.
186H	390	IA32_PERFEVTSELO	Unique	See Table 35-2.
187H	391	IA32_PERFEVTSEL1	Unique	See Table 35-2.
198H	408	IA32_PERF_STATUS	Shared	See Table 35-2.
198H	408	MSR_PERF_STATUS	Shared	
		15:0		Current Performance State Value.
		30:16		Reserved.
		31		XE Operation (R/O). If set, XE operation is enabled. Default is cleared.
		39:32		Reserved.
		44:40		Maximum Bus Ratio (R/O) Indicates maximum bus ratio configured for the processor.
		45		Reserved.
		46		Non-Integer Bus Ratio (R/O) Indicates non-integer bus ratio is enabled. Applies processors based on Enhanced Intel Core microarchitecture.
63:47		Reserved.		
199H	409	IA32_PERF_CTL	Unique	See Table 35-2.
19AH	410	IA32_CLOCK_MODULATION	Unique	<b>Clock Modulation (R/W)</b> See Table 35-2. IA32_CLOCK_MODULATION MSR was originally named IA32_THERM_CONTROL MSR.
19BH	411	IA32_THERM_INTERRUPT	Unique	<b>Thermal Interrupt Control (R/W)</b> See Table 35-2.
19CH	412	IA32_THERM_STATUS	Unique	<b>Thermal Monitor Status (R/W)</b> See Table 35-2.

Table 35-3. MSRs in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
19DH	413	MSR_THERM2_CTL	Unique	
		15:0		Reserved.
		16		<b>TM_SELECT (R/W)</b> Mode of automatic thermal monitor: 0 = Thermal Monitor 1 (thermally-initiated on-die modulation of the stop-clock duty cycle) 1 = Thermal Monitor 2 (thermally-initiated frequency transitions) If bit 3 of the IA32_MISC_ENABLE register is cleared, TM_SELECT has no effect. Neither TM1 nor TM2 are enabled.
		63:16		Reserved.
1A0H	416	IA32_MISC_ENABLE		<b>Enable Misc. Processor Features (R/W)</b> Allows a variety of processor functions to be enabled and disabled.
		0		<b>Fast-Strings Enable</b> See Table 35-2.
		2:1		Reserved.
		3	Unique	<b>Automatic Thermal Control Circuit Enable (R/W)</b> See Table 35-2.
		6:4		Reserved.
		7	Shared	<b>Performance Monitoring Available (R)</b> See Table 35-2.
		8		Reserved.
		9		<b>Hardware Prefetcher Disable (R/W)</b> When set, disables the hardware prefetcher operation on streams of data. When clear (default), enables the prefetch queue. Disabling of the hardware prefetcher may impact processor performance.
		10	Shared	<b>FERR# Multiplexing Enable (R/W)</b> 1 = FERR# asserted by the processor to indicate a pending break event within the processor 0 = Indicates compatible FERR# signaling behavior This bit must be set to 1 to support XAPIC interrupt model usage.
		11	Shared	<b>Branch Trace Storage Unavailable (RO)</b> See Table 35-2.
12	Shared	<b>Processor Event Based Sampling Unavailable (RO)</b> See Table 35-2.		

**Table 35-3. MSRs in Processors Based on Intel® Core™ Microarchitecture (Contd.)**

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
		13	Shared	<p><b>TM2 Enable (R/W)</b></p> <p>When this bit is set (1) and the thermal sensor indicates that the die temperature is at the pre-determined threshold, the Thermal Monitor 2 mechanism is engaged. TM2 will reduce the bus to core ratio and voltage according to the value last written to MSR_THERM2_CTL bits 15:0.</p>
				<p>When this bit is clear (0, default), the processor does not change the VID signals or the bus to core ratio when the processor enters a thermally managed state.</p> <p>The BIOS must enable this feature if the TM2 feature flag (CPUID.1:ECX[8]) is set; if the TM2 feature flag is not set, this feature is not supported and BIOS must not alter the contents of the TM2 bit location.</p> <p>The processor is operating out of specification if both this bit and the TM1 bit are set to 0.</p>
		15:14		Reserved.
		16	Shared	<p><b>Enhanced Intel SpeedStep Technology Enable (R/W)</b></p> <p>See Table 35-2.</p>
		18	Shared	<p><b>ENABLE MONITOR FSM (R/W)</b></p> <p>See Table 35-2.</p>
		19	Shared	<p><b>Adjacent Cache Line Prefetch Disable (R/W)</b></p> <p>When set to 1, the processor fetches the cache line that contains data currently required by the processor. When set to 0, the processor fetches cache lines that comprise a cache line pair (128 bytes).</p> <p>Single processor platforms should not set this bit. Server platforms should set or clear this bit based on platform performance observed in validation and testing.</p> <p>BIOS may contain a setup option that controls the setting of this bit.</p>
		20	Shared	<p><b>Enhanced Intel SpeedStep Technology Select Lock (R/W0)</b></p> <p>When set, this bit causes the following bits to become read-only:</p> <ul style="list-style-type: none"> <li>▪ Enhanced Intel SpeedStep Technology Select Lock (this bit),</li> <li>▪ Enhanced Intel SpeedStep Technology Enable bit.</li> </ul> <p>The bit must be set before an Enhanced Intel SpeedStep Technology transition is requested. This bit is cleared on reset.</p>
		21		Reserved.
		22	Shared	<p><b>Limit CPUID Maxval (R/W)</b></p> <p>See Table 35-2.</p>
		23	Shared	<p><b>xTPR Message Disable (R/W)</b></p> <p>See Table 35-2.</p>
		33:24		Reserved.

Table 35-3. MSRs in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
		34	Unique	<b>XD Bit Disable (R/W)</b> See Table 35-2.
		36:35		Reserved.
		37	Unique	<b>DCU Prefetcher Disable (R/W)</b> When set to 1, The DCU L1 data cache prefetcher is disabled. The default value after reset is 0. BIOS may write '1' to disable this feature.  The DCU prefetcher is an L1 data cache prefetcher. When the DCU prefetcher detects multiple loads from the same line done within a time limit, the DCU prefetcher assumes the next line will be required. The next line is prefetched in to the L1 data cache from memory or L2.
		38	Shared	<b>IDA Disable (R/W)</b> When set to 1 on processors that support IDA, the Intel Dynamic Acceleration feature (IDA) is disabled and the IDA_Enable feature flag will be clear (CPUID.06H: EAX[1]=0). When set to a 0 on processors that support IDA, CPUID.06H: EAX[1] reports the processor's support of IDA is enabled. <b>Note:</b> the power-on default value is used by BIOS to detect hardware support of IDA. If power-on default value is 1, IDA is available in the processor. If power-on default value is 0, IDA is not available.
		39	Unique	<b>IP Prefetcher Disable (R/W)</b> When set to 1, The IP prefetcher is disabled. The default value after reset is 0. BIOS may write '1' to disable this feature.  The IP prefetcher is an L1 data cache prefetcher. The IP prefetcher looks for sequential load history to determine whether to prefetch the next expected data into the L1 cache from memory or L2.
		63:40		Reserved.
1C9H	457	MSR_LASTBRANCH_TOS	Unique	<b>Last Branch Record Stack TOS (R/W)</b> Contains an index (bits 0-3) that points to the MSR containing the most recent branch record. See MSR_LASTBRANCH_0_FROM_IP (at 40H).
1D9H	473	IA32_DEBUGCTL	Unique	<b>Debug Control (R/W)</b> See Table 35-2
1DDH	477	MSR_LER_FROM_LIP	Unique	<b>Last Exception Record From Linear IP (R)</b> Contains a pointer to the last branch instruction that the processor executed prior to the last exception that was generated or the last interrupt that was handled.
1DEH	478	MSR_LER_TO_LIP	Unique	<b>Last Exception Record To Linear IP (R)</b> This area contains a pointer to the target of the last branch instruction that the processor executed prior to the last exception that was generated or the last interrupt that was handled.

**Table 35-3. MSRs in Processors Based on Intel® Core™ Microarchitecture (Contd.)**

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
200H	512	IA32_MTRR_PHYSBASE0	Unique	See Table 35-2.
201H	513	IA32_MTRR_PHYSMASK0	Unique	See Table 35-2.
202H	514	IA32_MTRR_PHYSBASE1	Unique	See Table 35-2.
203H	515	IA32_MTRR_PHYSMASK1	Unique	See Table 35-2.
204H	516	IA32_MTRR_PHYSBASE2	Unique	See Table 35-2.
205H	517	IA32_MTRR_PHYSMASK2	Unique	See Table 35-2.
206H	518	IA32_MTRR_PHYSBASE3	Unique	See Table 35-2.
207H	519	IA32_MTRR_PHYSMASK3	Unique	See Table 35-2.
208H	520	IA32_MTRR_PHYSBASE4	Unique	See Table 35-2.
209H	521	IA32_MTRR_PHYSMASK4	Unique	See Table 35-2.
20AH	522	IA32_MTRR_PHYSBASE5	Unique	See Table 35-2.
20BH	523	IA32_MTRR_PHYSMASK5	Unique	See Table 35-2.
20CH	524	IA32_MTRR_PHYSBASE6	Unique	See Table 35-2.
20DH	525	IA32_MTRR_PHYSMASK6	Unique	See Table 35-2.
20EH	526	IA32_MTRR_PHYSBASE7	Unique	See Table 35-2.
20FH	527	IA32_MTRR_PHYSMASK7	Unique	See Table 35-2.
250H	592	IA32_MTRR_FIX64K_00000	Unique	See Table 35-2.
258H	600	IA32_MTRR_FIX16K_80000	Unique	See Table 35-2.
259H	601	IA32_MTRR_FIX16K_A0000	Unique	See Table 35-2.
268H	616	IA32_MTRR_FIX4K_C0000	Unique	See Table 35-2.
269H	617	IA32_MTRR_FIX4K_C8000	Unique	See Table 35-2.
26AH	618	IA32_MTRR_FIX4K_D0000	Unique	See Table 35-2.
26BH	619	IA32_MTRR_FIX4K_D8000	Unique	See Table 35-2.
26CH	620	IA32_MTRR_FIX4K_E0000	Unique	See Table 35-2.
26DH	621	IA32_MTRR_FIX4K_E8000	Unique	See Table 35-2.
26EH	622	IA32_MTRR_FIX4K_F0000	Unique	See Table 35-2.
26FH	623	IA32_MTRR_FIX4K_F8000	Unique	See Table 35-2.
277H	631	IA32_PAT	Unique	See Table 35-2.
2FFH	767	IA32_MTRR_DEF_TYPE	Unique	<b>Default Memory Types (R/W)</b> See Table 35-2.
309H	777	IA32_FIXED_CTR0	Unique	<b>Fixed-Function Performance Counter Register 0 (R/W)</b> See Table 35-2.
309H	777	MSR_PERF_FIXED_CTR0	Unique	<b>Fixed-Function Performance Counter Register 0 (R/W)</b>

Table 35-3. MSRs in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
30AH	778	IA32_FIXED_CTR1	Unique	<b>Fixed-Function Performance Counter Register 1 (R/W)</b> See Table 35-2.
30AH	778	MSR_PERF_FIXED_CTR1	Unique	<b>Fixed-Function Performance Counter Register 1 (R/W)</b>
30BH	779	IA32_FIXED_CTR2	Unique	<b>Fixed-Function Performance Counter Register 2 (R/W)</b> See Table 35-2.
30BH	779	MSR_PERF_FIXED_CTR2	Unique	<b>Fixed-Function Performance Counter Register 2 (R/W)</b>
345H	837	IA32_PERF_CAPABILITIES	Unique	See Table 35-2. See Section 17.4.1, "IA32_DEBUGCTL MSR."
345H	837	MSR_PERF_CAPABILITIES	Unique	RO. This applies to processors that do not support architectural perfmon version 2.
		5:0		LBR Format. See Table 35-2.
		6		PEBS Record Format.
		7		PEBSSaveArchRegs. See Table 35-2.
63:8		Reserved.		
38DH	909	IA32_FIXED_CTR_CTRL	Unique	<b>Fixed-Function-Counter Control Register (R/W)</b> See Table 35-2.
38DH	909	MSR_PERF_FIXED_CTR_CTRL	Unique	<b>Fixed-Function-Counter Control Register (R/W)</b>
38EH	910	IA32_PERF_GLOBAL_STATUS	Unique	See Table 35-2. See Section 18.4.2, "Global Counter Control Facilities."
38EH	910	MSR_PERF_GLOBAL_STATUS	Unique	See Section 18.4.2, "Global Counter Control Facilities."
38FH	911	IA32_PERF_GLOBAL_CTRL	Unique	See Table 35-2. See Section 18.4.2, "Global Counter Control Facilities."
38FH	911	MSR_PERF_GLOBAL_CTRL	Unique	See Section 18.4.2, "Global Counter Control Facilities."
390H	912	IA32_PERF_GLOBAL_OVF_CTRL	Unique	See Table 35-2. See Section 18.4.2, "Global Counter Control Facilities."
390H	912	MSR_PERF_GLOBAL_OVF_CTRL	Unique	See Section 18.4.2, "Global Counter Control Facilities."
3F1H	1009	MSR_PEBS_ENABLE	Unique	See Table 35-2. See Section 18.4.4, "Precise Event Based Sampling (PEBS)."
		0		Enable PEBS on IA32_PMC0. (R/W)
400H	1024	IA32_MCO_CTL	Unique	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
401H	1025	IA32_MCO_STATUS	Unique	See Section 15.3.2.2, "IA32_MCi_STATUS MSRs."
402H	1026	IA32_MCO_ADDR	Unique	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The IA32_MCO_ADDR register is either not implemented or contains no address if the ADDR_V flag in the IA32_MCO_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.



**Table 35-3. MSRs in Processors Based on Intel® Core™ Microarchitecture (Contd.)**

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
404H	1028	IA32_MC1_CTL	Unique	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
405H	1029	IA32_MC1_STATUS	Unique	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
406H	1030	IA32_MC1_ADDR	Unique	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The IA32_MC1_ADDR register is either not implemented or contains no address if the ADDR_V flag in the IA32_MC1_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
408H	1032	IA32_MC2_CTL	Unique	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
409H	1033	IA32_MC2_STATUS	Unique	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
40AH	1034	IA32_MC2_ADDR	Unique	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The IA32_MC2_ADDR register is either not implemented or contains no address if the ADDR_V flag in the IA32_MC2_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
40CH	1036	IA32_MC4_CTL	Unique	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
40DH	1037	IA32_MC4_STATUS	Unique	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
40EH	1038	IA32_MC4_ADDR	Unique	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The MSR_MC4_ADDR register is either not implemented or contains no address if the ADDR_V flag in the MSR_MC4_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
410H	1040	IA32_MC3_CTL		See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
411H	1041	IA32_MC3_STATUS		See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
412H	1042	IA32_MC3_ADDR	Unique	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The MSR_MC3_ADDR register is either not implemented or contains no address if the ADDR_V flag in the MSR_MC3_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
413H	1043	IA32_MC3_MISC	Unique	
414H	1044	IA32_MC5_CTL	Unique	
415H	1045	IA32_MC5_STATUS	Unique	
416H	1046	IA32_MC5_ADDR	Unique	
417H	1047	IA32_MC5_MISC	Unique	
419H	1045	IA32_MC6_STATUS	Unique	Apply to Intel Xeon processor 7400 series (processor signature 06_1D) only. See Section 15.3.2.2, "IA32_MCi_STATUS MSRS." and Chapter 23.

Table 35-3. MSRs in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
480H	1152	IA32_VMX_BASIC	Unique	<b>Reporting Register of Basic VMX Capabilities (R/O)</b> See Table 35-2. See Appendix A.1, “Basic VMX Information.”
481H	1153	IA32_VMX_PINBASED_ CTLS	Unique	<b>Capability Reporting Register of Pin-based VM-execution Controls (R/O)</b> See Table 35-2. See Appendix A.3, “VM-Execution Controls.”
482H	1154	IA32_VMX_PROCBASED_ CTLS	Unique	<b>Capability Reporting Register of Primary Processor-based VM-execution Controls (R/O)</b> See Appendix A.3, “VM-Execution Controls.”
483H	1155	IA32_VMX_EXIT_ CTLS	Unique	<b>Capability Reporting Register of VM-exit Controls (R/O)</b> See Table 35-2. See Appendix A.4, “VM-Exit Controls.”
484H	1156	IA32_VMX_ENTRY_ CTLS	Unique	<b>Capability Reporting Register of VM-entry Controls (R/O)</b> See Table 35-2. See Appendix A.5, “VM-Entry Controls.”
485H	1157	IA32_VMX_MISC	Unique	<b>Reporting Register of Miscellaneous VMX Capabilities (R/O)</b> See Table 35-2. See Appendix A.6, “Miscellaneous Data.”
486H	1158	IA32_VMX_CRO_FIXED0	Unique	<b>Capability Reporting Register of CRO Bits Fixed to 0 (R/O)</b> See Table 35-2. See Appendix A.7, “VMX-Fixed Bits in CRO.”
487H	1159	IA32_VMX_CRO_FIXED1	Unique	<b>Capability Reporting Register of CRO Bits Fixed to 1 (R/O)</b> See Table 35-2. See Appendix A.7, “VMX-Fixed Bits in CRO.”
488H	1160	IA32_VMX_CR4_FIXED0	Unique	<b>Capability Reporting Register of CR4 Bits Fixed to 0 (R/O)</b> See Table 35-2. See Appendix A.8, “VMX-Fixed Bits in CR4.”
489H	1161	IA32_VMX_CR4_FIXED1	Unique	<b>Capability Reporting Register of CR4 Bits Fixed to 1 (R/O)</b> See Table 35-2. See Appendix A.8, “VMX-Fixed Bits in CR4.”
48AH	1162	IA32_VMX_VMCS_ENUM	Unique	<b>Capability Reporting Register of VMCS Field Enumeration (R/O)</b> See Table 35-2. See Appendix A.9, “VMCS Enumeration.”
48BH	1163	IA32_VMX_PROCBASED_ CTLS2	Unique	<b>Capability Reporting Register of Secondary Processor-based VM-execution Controls (R/O)</b> See Appendix A.3, “VM-Execution Controls.”

**Table 35-3. MSRs in Processors Based on Intel® Core™ Microarchitecture (Contd.)**

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
600H	1536	IA32_DS_AREA	Unique	<b>DS Save Area (R/W)</b> See Table 35-2. See Section 18.12.4, “Debug Store (DS) Mechanism.”
107CC H		MSR_EMON_L3_CTR_CTL0	Unique	<b>GBUSQ Event Control/Counter Register (R/W)</b> Apply to Intel Xeon processor 7400 series (processor signature 06_1D) only. See Section 17.2.2
107CD H		MSR_EMON_L3_CTR_CTL1	Unique	<b>GBUSQ Event Control/Counter Register (R/W)</b> Apply to Intel Xeon processor 7400 series (processor signature 06_1D) only. See Section 17.2.2
107CE H		MSR_EMON_L3_CTR_CTL2	Unique	<b>GSNPQ Event Control/Counter Register (R/W)</b> Apply to Intel Xeon processor 7400 series (processor signature 06_1D) only. See Section 17.2.2
107CF H		MSR_EMON_L3_CTR_CTL3	Unique	<b>GSNPQ Event Control/Counter Register (R/W)</b> Apply to Intel Xeon processor 7400 series (processor signature 06_1D) only. See Section 17.2.2
107D0 H		MSR_EMON_L3_CTR_CTL4	Unique	<b>FSB Event Control/Counter Register (R/W)</b> Apply to Intel Xeon processor 7400 series (processor signature 06_1D) only. See Section 17.2.2
107D1 H		MSR_EMON_L3_CTR_CTL5	Unique	<b>FSB Event Control/Counter Register (R/W)</b> Apply to Intel Xeon processor 7400 series (processor signature 06_1D) only. See Section 17.2.2
107D2 H		MSR_EMON_L3_CTR_CTL6	Unique	<b>FSB Event Control/Counter Register (R/W)</b> Apply to Intel Xeon processor 7400 series (processor signature 06_1D) only. See Section 17.2.2
107D3 H		MSR_EMON_L3_CTR_CTL7	Unique	<b>FSB Event Control/Counter Register (R/W)</b> Apply to Intel Xeon processor 7400 series (processor signature 06_1D) only. See Section 17.2.2
107D8 H		MSR_EMON_L3_GL_CTL	Unique	<b>L3/FSB Common Control Register (R/W)</b> Apply to Intel Xeon processor 7400 series (processor signature 06_1D) only. See Section 17.2.2
C000_ 0080H		IA32_EFER	Unique	<b>Extended Feature Enables</b> See Table 35-2.
C000_ 0081H		IA32_STAR	Unique	<b>System Call Target Address (R/W)</b> See Table 35-2.
C000_ 0082H		IA32_LSTAR	Unique	<b>IA-32e Mode System Call Target Address (R/W)</b> See Table 35-2.
C000_ 0084H		IA32_FMASK	Unique	<b>System Call Flag Mask (R/W)</b> See Table 35-2.
C000_ 0100H		IA32_FS_BASE	Unique	<b>Map of BASE Address of FS (R/W)</b> See Table 35-2.

**Table 35-3. MSRs in Processors Based on Intel® Core™ Microarchitecture (Contd.)**

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
C000_0101H		IA32_GS_BASE	Unique	<b>Map of BASE Address of GS (R/W)</b> See Table 35-2.
C000_0102H		IA32_KERNEL_GSBASE	Unique	<b>Swap Target of BASE Address of GS (R/W)</b> See Table 35-2.

### 35.3 MSRS IN THE 45 NM AND 32 NM INTEL® ATOM™ PROCESSOR FAMILY

Table 35-4 lists model-specific registers (MSRs) for 45 nm and 32 nm Intel Atom processors, architectural MSR addresses are also included in Table 35-4. These processors have a CPUID signature with DisplayFamily\_DisplayModel of 06\_1CH, 06\_26H, 06\_27H, 06\_35H and 06\_36H; see Table 35-1.

The column “Shared/Unique” applies to logical processors sharing the same core in processors based on the Intel Atom microarchitecture. “Unique” means each logical processor has a separate MSR, or a bit field in an MSR governs only a logical processor. “Shared” means the MSR or the bit field in an MSR address governs the operation of both logical processors in the same core.

**Table 35-4. MSRs in 45 nm and 32 nm Intel® Atom™ Processor Family**

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
0H	0	IA32_P5_MC_ADDR	Shared	See Section 35.22, “MSRs in Pentium Processors.”
1H	1	IA32_P5_MC_TYPE	Shared	See Section 35.22, “MSRs in Pentium Processors.”
6H	6	IA32_MONITOR_FILTER_SIZE	Unique	See Section 8.10.5, “Monitor/Mwait Address Range Determination,” and Table 35-2
10H	16	IA32_TIME_STAMP_COUNTER	Unique	See Section 17.15, “Time-Stamp Counter,” and see Table 35-2.
17H	23	IA32_PLATFORM_ID	Shared	<b>Platform ID (R)</b> See Table 35-2.
17H	23	MSR_PLATFORM_ID	Shared	<b>Model Specific Platform ID (R)</b>
		7:0		Reserved.
		12:8		<b>Maximum Qualified Ratio (R)</b> The maximum allowed bus ratio.
		63:13		Reserved.
1BH	27	IA32_APIC_BASE	Unique	See Section 10.4.4, “Local APIC Status and Location,” and Table 35-2.
2AH	42	MSR_EBL_CR_POWERON	Shared	<b>Processor Hard Power-On Configuration (R/W)</b> Enables and disables processor features; <b>(R)</b> indicates current processor configuration.
		0		Reserved.

**Table 35-4. MSRs in 45 nm and 32 nm Intel® Atom™ Processor Family (Contd.)**

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
		1		<b>Data Error Checking Enable (R/W)</b> 1 = Enabled; 0 = Disabled Always 0.
		2		<b>Response Error Checking Enable (R/W)</b> 1 = Enabled; 0 = Disabled Always 0.
		3		<b>AERR# Drive Enable (R/W)</b> 1 = Enabled; 0 = Disabled Always 0.
		4		<b>BERR# Enable for initiator bus requests (R/W)</b> 1 = Enabled; 0 = Disabled Always 0.
		5		Reserved.
		6		Reserved.
		7		<b>BINIT# Driver Enable (R/W)</b> 1 = Enabled; 0 = Disabled Always 0.
		8		Reserved.
		9		<b>Execute BIST (R/O)</b> 1 = Enabled; 0 = Disabled
		10		<b>AERR# Observation Enabled (R/O)</b> 1 = Enabled; 0 = Disabled Always 0.
		11		Reserved.
		12		<b>BINIT# Observation Enabled (R/O)</b> 1 = Enabled; 0 = Disabled Always 0.
		13		<b>Reserved.</b>
		14		<b>1 MByte Power on Reset Vector (R/O)</b> 1 = 1 MByte; 0 = 4 GBytes
		15		Reserved
		17:16		<b>APIC Cluster ID (R/O)</b> Always 00B.
		19: 18		Reserved.
		21: 20		<b>Symmetric Arbitration ID (R/O)</b> Always 00B.
		26:22		<b>Integer Bus Frequency Ratio (R/O)</b>
3AH	58	IA32_FEATURE_CONTROL	Unique	<b>Control Features in Intel 64Processor (R/W)</b> See Table 35-2.

Table 35-4. MSRs in 45 nm and 32 nm Intel® Atom™ Processor Family (Contd.)

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
40H	64	MSR_LASTBRANCH_0_FROM_IP	Unique	<b>Last Branch Record 0 From IP (R/W)</b> One of eight pairs of last branch record registers on the last branch record stack. The From_IP part of the stack contains pointers to the <b>source instruction</b> . See also: <ul style="list-style-type: none"> <li>Last Branch Record Stack TOS at 1C9H</li> <li>Section 17.5</li> </ul>
41H	65	MSR_LASTBRANCH_1_FROM_IP	Unique	<b>Last Branch Record 1 From IP (R/W)</b> See description of MSR_LASTBRANCH_0_FROM_IP.
42H	66	MSR_LASTBRANCH_2_FROM_IP	Unique	<b>Last Branch Record 2 From IP (R/W)</b> See description of MSR_LASTBRANCH_0_FROM_IP.
43H	67	MSR_LASTBRANCH_3_FROM_IP	Unique	<b>Last Branch Record 3 From IP (R/W)</b> See description of MSR_LASTBRANCH_0_FROM_IP.
44H	68	MSR_LASTBRANCH_4_FROM_IP	Unique	<b>Last Branch Record 4 From IP (R/W)</b> See description of MSR_LASTBRANCH_0_FROM_IP.
45H	69	MSR_LASTBRANCH_5_FROM_IP	Unique	<b>Last Branch Record 5 From IP (R/W)</b> See description of MSR_LASTBRANCH_0_FROM_IP.
46H	70	MSR_LASTBRANCH_6_FROM_IP	Unique	<b>Last Branch Record 6 From IP (R/W)</b> See description of MSR_LASTBRANCH_0_FROM_IP.
47H	71	MSR_LASTBRANCH_7_FROM_IP	Unique	<b>Last Branch Record 7 From IP (R/W)</b> See description of MSR_LASTBRANCH_0_FROM_IP.
60H	96	MSR_LASTBRANCH_0_TO_IP	Unique	<b>Last Branch Record 0 To IP (R/W)</b> One of eight pairs of last branch record registers on the last branch record stack. The To_IP part of the stack contains pointers to the destination instruction.
61H	97	MSR_LASTBRANCH_1_TO_IP	Unique	<b>Last Branch Record 1 To IP (R/W)</b> See description of MSR_LASTBRANCH_0_TO_IP.
62H	98	MSR_LASTBRANCH_2_TO_IP	Unique	<b>Last Branch Record 2 To IP (R/W)</b> See description of MSR_LASTBRANCH_0_TO_IP.
63H	99	MSR_LASTBRANCH_3_TO_IP	Unique	<b>Last Branch Record 3 To IP (R/W)</b> See description of MSR_LASTBRANCH_0_TO_IP.
64H	100	MSR_LASTBRANCH_4_TO_IP	Unique	<b>Last Branch Record 4 To IP (R/W)</b> See description of MSR_LASTBRANCH_0_TO_IP.
65H	101	MSR_LASTBRANCH_5_TO_IP	Unique	<b>Last Branch Record 5 To IP (R/W)</b> See description of MSR_LASTBRANCH_0_TO_IP.
66H	102	MSR_LASTBRANCH_6_TO_IP	Unique	<b>Last Branch Record 6 To IP (R/W)</b> See description of MSR_LASTBRANCH_0_TO_IP.
67H	103	MSR_LASTBRANCH_7_TO_IP	Unique	<b>Last Branch Record 7 To IP (R/W)</b> See description of MSR_LASTBRANCH_0_TO_IP.
79H	121	IA32_BIOS_UPDT_TRIG	Shared	<b>BIOS Update Trigger Register (W)</b> See Table 35-2.

**Table 35-4. MSRs in 45 nm and 32 nm Intel® Atom™ Processor Family (Contd.)**

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
8BH	139	IA32_BIOS_SIGN_ID	Unique	<b>BIOS Update Signature ID (RO)</b> See Table 35-2.
C1H	193	IA32_PMC0	Unique	<b>Performance counter register</b> See Table 35-2.
C2H	194	IA32_PMC1	Unique	<b>Performance Counter Register</b> See Table 35-2.
CDH	205	MSR_FSB_FREQ	Shared	<b>Scaleable Bus Speed(RO)</b> This field indicates the intended scaleable bus clock speed for processors based on Intel Atom microarchitecture:
		2:0		<ul style="list-style-type: none"> <li>▪ 111B: 083 MHz (FSB 333)</li> <li>▪ 101B: 100 MHz (FSB 400)</li> <li>▪ 001B: 133 MHz (FSB 533)</li> <li>▪ 011B: 167 MHz (FSB 667)</li> </ul> 133.33 MHz should be utilized if performing calculation with System Bus Speed when encoding is 001B. 166.67 MHz should be utilized if performing calculation with System Bus Speed when encoding is 011B.
		63:3		Reserved.
E7H	231	IA32_MPERF	Unique	<b>Maximum Performance Frequency Clock Count (RW)</b> See Table 35-2.
E8H	232	IA32_APERF	Unique	<b>Actual Performance Frequency Clock Count (RW)</b> See Table 35-2.
FEH	254	IA32_MTRRCAP	Shared	<b>Memory Type Range Register (R)</b> See Table 35-2.
11EH	281	MSR_BBL_CR_CTL3	Shared	
		0		<b>L2 Hardware Enabled (RO)</b> 1 = If the L2 is hardware-enabled 0 = Indicates if the L2 is hardware-disabled
		7:1		Reserved.
		8		<b>L2 Enabled. (R/W)</b> 1 = L2 cache has been initialized 0 = Disabled (default) Until this bit is set the processor will not respond to the WBINVD instruction or the assertion of the FLUSH# input.
		22:9		Reserved.
		23		<b>L2 Not Present (RO)</b> 0 = L2 Present 1 = L2 Not Present
		63:24		Reserved.
174H	372	IA32_SYSENTER_CS	Unique	See Table 35-2.
175H	373	IA32_SYSENTER_ESP	Unique	See Table 35-2.

Table 35-4. MSRs in 45 nm and 32 nm Intel® Atom™ Processor Family (Contd.)

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
176H	374	IA32_SYSENTER_EIP	Unique	See Table 35-2.
179H	377	IA32_MCG_CAP	Unique	See Table 35-2.
17AH	378	IA32_MCG_STATUS	Unique	
		0		<b>RIPV</b> When set, bit indicates that the instruction addressed by the instruction pointer pushed on the stack (when the machine check was generated) can be used to restart the program. If cleared, the program cannot be reliably restarted
		1		<b>EIPV</b> When set, bit indicates that the instruction addressed by the instruction pointer pushed on the stack (when the machine check was generated) is directly associated with the error.
		2		<b>MCIP</b> When set, bit indicates that a machine check has been generated. If a second machine check is detected while this bit is still set, the processor enters a shutdown state. Software should write this bit to 0 after processing a machine check exception.
		63:3		Reserved.
186H	390	IA32_PERFEVTSELO	Unique	See Table 35-2.
187H	391	IA32_PERFEVTSEL1	Unique	See Table 35-2.
198H	408	IA32_PERF_STATUS	Shared	See Table 35-2.
198H	408	MSR_PERF_STATUS	Shared	
		15:0		Current Performance State Value.
		39:16		Reserved.
		44:40		Maximum Bus Ratio (R/O) Indicates maximum bus ratio configured for the processor.
		63:45		Reserved.
199H	409	IA32_PERF_CTL	Unique	See Table 35-2.
19AH	410	IA32_CLOCK_MODULATION	Unique	<b>Clock Modulation (R/W)</b> See Table 35-2. IA32_CLOCK_MODULATION MSR was originally named IA32_THERM_CONTROL MSR.
19BH	411	IA32_THERM_INTERRUPT	Unique	<b>Thermal Interrupt Control (R/W)</b> See Table 35-2.
19CH	412	IA32_THERM_STATUS	Unique	<b>Thermal Monitor Status (R/W)</b> See Table 35-2.
19DH	413	MSR_THERM2_CTL	Shared	
		15:0		Reserved.



**Table 35-4. MSRs in 45 nm and 32 nm Intel® Atom™ Processor Family (Contd.)**

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
		16		<b>TM_SELECT (R/W)</b> Mode of automatic thermal monitor: 0 = Thermal Monitor 1 (thermally-initiated on-die modulation of the stop-clock duty cycle) 1 = Thermal Monitor 2 (thermally-initiated frequency transitions) If bit 3 of the IA32_MISC_ENABLE register is cleared, TM_SELECT has no effect. Neither TM1 nor TM2 are enabled.
		63:17		Reserved.
1A0H	416	IA32_MISC_ENABLE	Unique	<b>Enable Misc. Processor Features (R/W)</b> Allows a variety of processor functions to be enabled and disabled.
		0		<b>Fast-Strings Enable</b> See Table 35-2.
		2:1		Reserved.
		3	Unique	<b>Automatic Thermal Control Circuit Enable (R/W)</b> See Table 35-2. Default value is 0.
		6:4		Reserved.
		7	Shared	<b>Performance Monitoring Available (R)</b> See Table 35-2.
		8		Reserved.
		9		Reserved.
		10	Shared	<b>FERR# Multiplexing Enable (R/W)</b> 1 = FERR# asserted by the processor to indicate a pending break event within the processor 0 = Indicates compatible FERR# signaling behavior This bit must be set to 1 to support XAPIC interrupt model usage.
		11	Shared	<b>Branch Trace Storage Unavailable (RO)</b> See Table 35-2.
12	Shared	<b>Processor Event Based Sampling Unavailable (RO)</b> See Table 35-2.		
13	Shared	<b>TM2 Enable (R/W)</b> When this bit is set (1) and the thermal sensor indicates that the die temperature is at the pre-determined threshold, the Thermal Monitor 2 mechanism is engaged. TM2 will reduce the bus to core ratio and voltage according to the value last written to MSR_THERM2_CTL bits 15:0.		

Table 35-4. MSRs in 45 nm and 32 nm Intel® Atom™ Processor Family (Contd.)

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
				When this bit is clear (0, default), the processor does not change the VID signals or the bus to core ratio when the processor enters a thermally managed state.  The BIOS must enable this feature if the TM2 feature flag (CPUID.1:ECX[8]) is set; if the TM2 feature flag is not set, this feature is not supported and BIOS must not alter the contents of the TM2 bit location.  The processor is operating out of specification if both this bit and the TM1 bit are set to 0.
		15:14		Reserved.
		16	Shared	<b>Enhanced Intel SpeedStep Technology Enable (R/W)</b> See Table 35-2.
		18	Shared	<b>ENABLE MONITOR FSM (R/W)</b> See Table 35-2.
		19		Reserved.
		20	Shared	<b>Enhanced Intel SpeedStep Technology Select Lock (R/WO)</b> When set, this bit causes the following bits to become read-only: <ul style="list-style-type: none"> <li>▪ Enhanced Intel SpeedStep Technology Select Lock (this bit),</li> <li>▪ Enhanced Intel SpeedStep Technology Enable bit.</li> </ul> The bit must be set before an Enhanced Intel SpeedStep Technology transition is requested. This bit is cleared on reset.
		21		Reserved.
		22	Unique	<b>Limit CPUID Maxval (R/W)</b> See Table 35-2.
		23	Shared	<b>xTPR Message Disable (R/W)</b> See Table 35-2.
		33:24		Reserved.
		34	Unique	<b>XD Bit Disable (R/W)</b> See Table 35-2.
		63:35		Reserved.
1C9H	457	MSR_LASTBRANCH_TOS	Unique	<b>Last Branch Record Stack TOS (R/W)</b> Contains an index (bits 0-2) that points to the MSR containing the most recent branch record. See MSR_LASTBRANCH_0_FROM_IP (at 40H).
1D9H	473	IA32_DEBUGCTL	Unique	<b>Debug Control (R/W)</b> See Table 35-2.
1DDH	477	MSR_LER_FROM_LIP	Unique	<b>Last Exception Record From Linear IP (R)</b> Contains a pointer to the last branch instruction that the processor executed prior to the last exception that was generated or the last interrupt that was handled.

Table 35-4. MSRs in 45 nm and 32 nm Intel® Atom™ Processor Family (Contd.)

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
1DEH	478	MSR_LER_TO_LIP	Unique	<b>Last Exception Record To Linear IP (R)</b> This area contains a pointer to the target of the last branch instruction that the processor executed prior to the last exception that was generated or the last interrupt that was handled.
200H	512	IA32_MTRR_PHYSBASE0	Shared	See Table 35-2.
201H	513	IA32_MTRR_PHYSMASK0	Shared	See Table 35-2.
202H	514	IA32_MTRR_PHYSBASE1	Shared	See Table 35-2.
203H	515	IA32_MTRR_PHYSMASK1	Shared	See Table 35-2.
204H	516	IA32_MTRR_PHYSBASE2	Shared	See Table 35-2.
205H	517	IA32_MTRR_PHYSMASK2	Shared	See Table 35-2.
206H	518	IA32_MTRR_PHYSBASE3	Shared	See Table 35-2.
207H	519	IA32_MTRR_PHYSMASK3	Shared	See Table 35-2.
208H	520	IA32_MTRR_PHYSBASE4	Shared	See Table 35-2.
209H	521	IA32_MTRR_PHYSMASK4	Shared	See Table 35-2.
20AH	522	IA32_MTRR_PHYSBASE5	Shared	See Table 35-2.
20BH	523	IA32_MTRR_PHYSMASK5	Shared	See Table 35-2.
20CH	524	IA32_MTRR_PHYSBASE6	Shared	See Table 35-2.
20DH	525	IA32_MTRR_PHYSMASK6	Shared	See Table 35-2.
20EH	526	IA32_MTRR_PHYSBASE7	Shared	See Table 35-2.
20FH	527	IA32_MTRR_PHYSMASK7	Shared	See Table 35-2.
250H	592	IA32_MTRR_FIX64K_00000	Shared	See Table 35-2.
258H	600	IA32_MTRR_FIX16K_80000	Shared	See Table 35-2.
259H	601	IA32_MTRR_FIX16K_A0000	Shared	See Table 35-2.
268H	616	IA32_MTRR_FIX4K_C0000	Shared	See Table 35-2.
269H	617	IA32_MTRR_FIX4K_C8000	Shared	See Table 35-2.
26AH	618	IA32_MTRR_FIX4K_D0000	Shared	See Table 35-2.
26BH	619	IA32_MTRR_FIX4K_D8000	Shared	See Table 35-2.
26CH	620	IA32_MTRR_FIX4K_E0000	Shared	See Table 35-2.
26DH	621	IA32_MTRR_FIX4K_E8000	Shared	See Table 35-2.
26EH	622	IA32_MTRR_FIX4K_F0000	Shared	See Table 35-2.
26FH	623	IA32_MTRR_FIX4K_F8000	Shared	See Table 35-2.
277H	631	IA32_PAT	Unique	See Table 35-2.
309H	777	IA32_FIXED_CTR0	Unique	<b>Fixed-Function Performance Counter Register 0 (R/W)</b> See Table 35-2.
30AH	778	IA32_FIXED_CTR1	Unique	<b>Fixed-Function Performance Counter Register 1 (R/W)</b> See Table 35-2.

Table 35-4. MSRs in 45 nm and 32 nm Intel® Atom™ Processor Family (Contd.)

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
30BH	779	IA32_FIXED_CTR2	Unique	<b>Fixed-Function Performance Counter Register 2 (R/W)</b> See Table 35-2.
345H	837	IA32_PERF_CAPABILITIES	Shared	See Table 35-2. See Section 17.4.1, "IA32_DEBUGCTL MSR."
38DH	909	IA32_FIXED_CTR_CTRL	Unique	<b>Fixed-Function-Counter Control Register (R/W)</b> See Table 35-2.
38EH	910	IA32_PERF_GLOBAL_STATUS	Unique	See Table 35-2. See Section 18.4.2, "Global Counter Control Facilities."
38FH	911	IA32_PERF_GLOBAL_CTRL	Unique	See Table 35-2. See Section 18.4.2, "Global Counter Control Facilities."
390H	912	IA32_PERF_GLOBAL_OVF_CTRL	Unique	See Table 35-2. See Section 18.4.2, "Global Counter Control Facilities."
3F1H	1009	MSR_PEBS_ENABLE	Unique	See Table 35-2. See Section 18.4.4, "Precise Event Based Sampling (PEBS)."
		0		Enable PEBS on IA32_PMC0. (R/W)
400H	1024	IA32_MCO_CTL	Shared	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
401H	1025	IA32_MCO_STATUS	Shared	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
402H	1026	IA32_MCO_ADDR	Shared	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The IA32_MCO_ADDR register is either not implemented or contains no address if the ADDR_V flag in the IA32_MCO_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
404H	1028	IA32_MC1_CTL	Shared	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
405H	1029	IA32_MC1_STATUS	Shared	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
408H	1032	IA32_MC2_CTL	Shared	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
409H	1033	IA32_MC2_STATUS	Shared	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
40AH	1034	IA32_MC2_ADDR	Shared	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The IA32_MC2_ADDR register is either not implemented or contains no address if the ADDR_V flag in the IA32_MC2_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
40CH	1036	IA32_MC3_CTL	Shared	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
40DH	1037	IA32_MC3_STATUS	Shared	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
40EH	1038	IA32_MC3_ADDR	Shared	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The MSR_MC3_ADDR register is either not implemented or contains no address if the ADDR_V flag in the MSR_MC3_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
410H	1040	IA32_MC4_CTL	Shared	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
411H	1041	IA32_MC4_STATUS	Shared	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."

**Table 35-4. MSRs in 45 nm and 32 nm Intel® Atom™ Processor Family (Contd.)**

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
412H	1042	IA32_MC4_ADDR	Shared	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The MSR_MC4_ADDR register is either not implemented or contains no address if the ADDR_V flag in the MSR_MC4_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
480H	1152	IA32_VMX_BASIC	Unique	<b>Reporting Register of Basic VMX Capabilities (R/O)</b> See Table 35-2. See Appendix A.1, "Basic VMX Information."
481H	1153	IA32_VMX_PINBASED_CTL	Unique	<b>Capability Reporting Register of Pin-based VM-execution Controls (R/O)</b> See Table 35-2. See Appendix A.3, "VM-Execution Controls."
482H	1154	IA32_VMX_PROCBASED_CTL	Unique	<b>Capability Reporting Register of Primary Processor-based VM-execution Controls (R/O)</b> See Appendix A.3, "VM-Execution Controls."
483H	1155	IA32_VMX_EXIT_CTL	Unique	<b>Capability Reporting Register of VM-exit Controls (R/O)</b> See Table 35-2. See Appendix A.4, "VM-Exit Controls."
484H	1156	IA32_VMX_ENTRY_CTL	Unique	<b>Capability Reporting Register of VM-entry Controls (R/O)</b> See Table 35-2. See Appendix A.5, "VM-Entry Controls."
485H	1157	IA32_VMX_MISC	Unique	<b>Reporting Register of Miscellaneous VMX Capabilities (R/O)</b> See Table 35-2. See Appendix A.6, "Miscellaneous Data."
486H	1158	IA32_VMX_CR0_FIXED0	Unique	<b>Capability Reporting Register of CR0 Bits Fixed to 0 (R/O)</b> See Table 35-2. See Appendix A.7, "VMX-Fixed Bits in CR0."
487H	1159	IA32_VMX_CR0_FIXED1	Unique	<b>Capability Reporting Register of CR0 Bits Fixed to 1 (R/O)</b> See Table 35-2. See Appendix A.7, "VMX-Fixed Bits in CR0."
488H	1160	IA32_VMX_CR4_FIXED0	Unique	<b>Capability Reporting Register of CR4 Bits Fixed to 0 (R/O)</b> See Table 35-2. See Appendix A.8, "VMX-Fixed Bits in CR4."
489H	1161	IA32_VMX_CR4_FIXED1	Unique	<b>Capability Reporting Register of CR4 Bits Fixed to 1 (R/O)</b> See Table 35-2. See Appendix A.8, "VMX-Fixed Bits in CR4."
48AH	1162	IA32_VMX_VMCS_ENUM	Unique	<b>Capability Reporting Register of VMCS Field Enumeration (R/O)</b> See Table 35-2. See Appendix A.9, "VMCS Enumeration."

**Table 35-4. MSRs in 45 nm and 32 nm Intel® Atom™ Processor Family (Contd.)**

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
48BH	1163	IA32_VMX_PROCBASED_CTLDS2	Unique	<b>Capability Reporting Register of Secondary Processor-based VM-execution Controls (R/O)</b> See Appendix A.3, “VM-Execution Controls.”
600H	1536	IA32_DS_AREA	Unique	<b>DS Save Area (R/W)</b> See Table 35-2. See Section 18.12.4, “Debug Store (DS) Mechanism.”
C000_0080H		IA32_EFER	Unique	<b>Extended Feature Enables</b> See Table 35-2.
C000_0081H		IA32_STAR	Unique	<b>System Call Target Address (R/W)</b> See Table 35-2.
C000_0082H		IA32_LSTAR	Unique	<b>IA-32e Mode System Call Target Address (R/W)</b> See Table 35-2.
C000_0084H		IA32_FMASK	Unique	<b>System Call Flag Mask (R/W)</b> See Table 35-2.
C000_0100H		IA32_FS_BASE	Unique	<b>Map of BASE Address of FS (R/W)</b> See Table 35-2.
C000_0101H		IA32_GS_BASE	Unique	<b>Map of BASE Address of GS (R/W)</b> See Table 35-2.
C000_0102H		IA32_KERNEL_GSBASE	Unique	<b>Swap Target of BASE Address of GS (R/W)</b> See Table 35-2.

Table 35-5 lists model-specific registers (MSRs) that are specific to Intel® Atom™ processor with the CPUID signature with DisplayFamily\_DisplayModel of 06\_27H.

**Table 35-5. MSRs Supported by Intel® Atom™ Processors with CPUID Signature 06\_27H**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
3F8H	1016	MSR_PKG_C2_RESIDENCY	Package	<b>Package C2 Residency</b> Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States
		63:0	Package	Package C2 Residency Counter. (R/O) Time that this package is in processor-specific C2 states since last reset. Counts at 1 Mhz frequency.
3F9H	1017	MSR_PKG_C4_RESIDENCY	Package	<b>Package C4 Residency</b> Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States
		63:0	Package	Package C4 Residency Counter. (R/O) Time that this package is in processor-specific C4 states since last reset. Counts at 1 Mhz frequency.

**Table 35-5. MSRs Supported by Intel® Atom™ Processors (Contd.)with CPUID Signature 06\_27H**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
3FAH	1018	MSR_PKG_C6_RESIDENCY	Package	<b>Package C6 Residency</b> Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States
		63:0	Package	Package C6 Residency Counter. (R/O) Time that this package is in processor-specific C6 states since last reset. Counts at 1 Mhz frequency.

### 35.4 MSRS IN INTEL PROCESSORS BASED ON SILVERMONT MICROARCHITECTURE

Table 35-6 lists model-specific registers (MSRs) common to Intel processors based on the Silvermont microarchitecture. These processors have a CPUID signature with DisplayFamily\_DisplayModel of 06\_37H, 06\_4AH, 06\_4DH, 06\_5AH, and 06\_5DH; see Table 35-1. The MSRs listed in Table 35-6 are also common to processors based on the Airmont microarchitecture and newer microarchitectures for next generation Intel Atom processors.

Table 35-7 lists MSRs common to processors based on the Silvermont and Airmont microarchitectures, but not newer microarchitectures.

Table 35-8, Table 35-9, and Table 35-10 lists MSRs that are model-specific across processors based on the Silvermont microarchitecture.

In the Silvermont microarchitecture, the scope column indicates the following: “Core” means each processor core has a separate MSR, or a bit field not shared with another processor core. “Module” means the MSR or the bit field is shared by a pair of processor cores in the physical package. “Package” means all processor cores in the physical package share the same MSR or bit interface.

**Table 35-6. MSRs Common to the Silvermont Microarchitecture and Newer Microarchitectures for Intel Atom Processors**

Address		Register Name	Scope	Bit Description
Hex	Dec			
0H	0	IA32_P5_MC_ADDR	Module	See Section 35.22, “MSRs in Pentium Processors.”
1H	1	IA32_P5_MC_TYPE	Module	See Section 35.22, “MSRs in Pentium Processors.”
6H	6	IA32_MONITOR_FILTER_SIZE	Core	See Section 8.10.5, “Monitor/Mwait Address Range Determination,” and Table 35-2
10H	16	IA32_TIME_STAMP_COUNTER	Core	See Section 17.15, “Time-Stamp Counter,” and see Table 35-2.
1BH	27	IA32_APIC_BASE	Core	See Section 10.4.4, “Local APIC Status and Location,” and Table 35-2.
2AH	42	MSR_EBL_CR_POWERON	Module	<b>Processor Hard Power-On Configuration (R/W)</b> Writes ignored.
		63:0		Reserved (R/O)
34H	52	MSR_SMI_COUNT	Core	<b>SMI Counter (R/O)</b>
		31:0		<b>SMI Count (R/O)</b> Running count of SMI events since last RESET.
		63:32		Reserved.

**Table 35-6. MSRs Common to the Silvermont Microarchitecture and Newer Microarchitectures for Intel Atom Processors**

Address		Register Name	Scope	Bit Description
Hex	Dec			
79H	121	IA32_BIOS_UPDT_TRIG	Core	<b>BIOS Update Trigger Register (W)</b> See Table 35-2.
8BH	139	IA32_BIOS_SIGN_ID	Core	<b>BIOS Update Signature ID (RO)</b> See Table 35-2.
C1H	193	IA32_PMC0	Core	<b>Performance counter register</b> See Table 35-2.
C2H	194	IA32_PMC1	Core	<b>Performance Counter Register</b> See Table 35-2.
E4H	228	MSR_PMG_IO_CAPTURE_BASE	Module	<b>Power Management IO Redirection in C-state (R/W)</b> See <a href="http://biosbits.org">http://biosbits.org</a> .
		15:0		<b>LVL_2 Base Address (R/W)</b> Specifies the base address visible to software for IO redirection. If IO MWAIT Redirection is enabled, reads to this address will be consumed by the power management logic and decoded to MWAIT instructions. When IO port address redirection is enabled, this is the IO port address reported to the OS/software.
		18:16		<b>C-state Range (R/W)</b> Specifies the encoding value of the maximum C-State code name to be included when IO read to MWAIT redirection is enabled by MSR_PKG_CST_CONFIG_CONTROL[bit10]: 100b - C4 is the max C-State to include 110b - C6 is the max C-State to include 111b - C7 is the max C-State to include
		63:19		Reserved.
E7H	231	IA32_MPERF	Core	<b>Maximum Performance Frequency Clock Count (RW)</b> See Table 35-2.
E8H	232	IA32_APERF	Core	<b>Actual Performance Frequency Clock Count (RW)</b> See Table 35-2.
FEH	254	IA32_MTRRCAP	Core	<b>Memory Type Range Register (R)</b> See Table 35-2.
13CH	52	MSR_FEATURE_CONFIG	Core	<b>AES Configuration (RW-L)</b> Privileged post-BIOS agent must provide a #GP handler to handle unsuccessful read of this MSR.
		1:0		<b>AES Configuration (RW-L)</b> Upon a successful read of this MSR, the configuration of AES instruction set availability is as follows: 11b: AES instructions are not available until next RESET. otherwise, AES instructions are available. Note, AES instruction set is not available if read is unsuccessful. If the configuration is not 01b, AES instruction can be mis-configured if a privileged agent unintentionally writes 11b.
		63:2		Reserved.



**Table 35-6. MSRs Common to the Silvermont Microarchitecture and Newer Microarchitectures for Intel Atom Processors**

Address		Register Name	Scope	Bit Description
Hex	Dec			
174H	372	IA32_SYSENTER_CS	Core	See Table 35-2.
175H	373	IA32_SYSENTER_ESP	Core	See Table 35-2.
176H	374	IA32_SYSENTER_EIP	Core	See Table 35-2.
179H	377	IA32_MCG_CAP	Core	See Table 35-2.
17AH	378	IA32_MCG_STATUS	Core	
		0		<b>RIPV</b> When set, bit indicates that the instruction addressed by the instruction pointer pushed on the stack (when the machine check was generated) can be used to restart the program. If cleared, the program cannot be reliably restarted
		1		<b>EIPV</b> When set, bit indicates that the instruction addressed by the instruction pointer pushed on the stack (when the machine check was generated) is directly associated with the error.
		2		<b>MCIP</b> When set, bit indicates that a machine check has been generated. If a second machine check is detected while this bit is still set, the processor enters a shutdown state. Software should write this bit to 0 after processing a machine check exception.
		63:3		Reserved.
186H	390	IA32_PERFEVTSELO	Core	See Table 35-2.
		7:0		<b>Event Select</b>
		15:8		<b>UMask</b>
		16		<b>USR</b>
		17		<b>OS</b>
		18		<b>Edge</b>
		19		<b>PC</b>
		20		<b>INT</b>
		21		<b>Reserved</b>
		22		<b>EN</b>
		23		<b>INV</b>
		31:24		<b>CMASK</b>
		63:32		Reserved.
187H	391	IA32_PERFEVTSEL1	Core	See Table 35-2.
198H	408	IA32_PERF_STATUS	Module	See Table 35-2.
199H	409	IA32_PERF_CTL	Core	See Table 35-2.
19AH	410	IA32_CLOCK_MODULATION	Core	<b>Clock Modulation (R/W)</b> See Table 35-2. IA32_CLOCK_MODULATION MSR was originally named IA32_THERM_CONTROL MSR.

**Table 35-6. MSRs Common to the Silvermont Microarchitecture and Newer Microarchitectures for Intel Atom Processors**

Address		Register Name	Scope	Bit Description
Hex	Dec			
19BH	411	IA32_THERM_INTERRUPT	Core	<b>Thermal Interrupt Control (R/W)</b> See Table 35-2.
19CH	412	IA32_THERM_STATUS	Core	<b>Thermal Monitor Status (R/W)</b> See Table 35-2.
1A2H	418	MSR_TEMPERATURE_TARGET	Package	
		15:0		Reserved.
		23:16		<b>Temperature Target (R)</b> The default thermal throttling or PROCHOT# activation temperature in degree C, The effective temperature for thermal throttling or PROCHOT# activation is "Temperature Target" + "Target Offset"
		29:24		<b>Target Offset (R/W)</b> Specifies an offset in degrees C to adjust the throttling and PROCHOT# activation temperature from the default target specified in TEMPERATURE_TARGET (bits 23:16).
		63:30		Reserved.
1A6H	422	MSR_OFFCORE_RSP_0	Module	<b>Offcore Response Event Select Register (R/W)</b>
1A7H	423	MSR_OFFCORE_RSP_1	Module	<b>Offcore Response Event Select Register (R/W)</b>
1B0H	432	IA32_ENERGY_PERF_BIAS	Core	See Table 35-2.
1D9H	473	IA32_DEBUGCTL	Core	<b>Debug Control (R/W)</b> See Table 35-2.
1DDH	477	MSR_LER_FROM_LIP	Core	<b>Last Exception Record From Linear IP (R)</b> Contains a pointer to the last branch instruction that the processor executed prior to the last exception that was generated or the last interrupt that was handled.
1DEH	478	MSR_LER_TO_LIP	Core	<b>Last Exception Record To Linear IP (R)</b> This area contains a pointer to the target of the last branch instruction that the processor executed prior to the last exception that was generated or the last interrupt that was handled.
1F2H	498	IA32_SMRR_PHYSBASE	Core	See Table 35-2.
1F3H	499	IA32_SMRR_PHYSMASK	Core	See Table 35-2.
200H	512	IA32_MTRR_PHYSBASE0	Core	See Table 35-2.
201H	513	IA32_MTRR_PHYSMASK0	Core	See Table 35-2.
202H	514	IA32_MTRR_PHYSBASE1	Core	See Table 35-2.
203H	515	IA32_MTRR_PHYSMASK1	Core	See Table 35-2.
204H	516	IA32_MTRR_PHYSBASE2	Core	See Table 35-2.
205H	517	IA32_MTRR_PHYSMASK2	Core	See Table 35-2.
206H	518	IA32_MTRR_PHYSBASE3	Core	See Table 35-2.
207H	519	IA32_MTRR_PHYSMASK3	Core	See Table 35-2.
208H	520	IA32_MTRR_PHYSBASE4	Core	See Table 35-2.

**Table 35-6. MSRs Common to the Silvermont Microarchitecture and Newer Microarchitectures for Intel Atom Processors**

Address		Register Name	Scope	Bit Description
Hex	Dec			
209H	521	IA32_MTRR_PHYSMASK4	Core	See Table 35-2.
20AH	522	IA32_MTRR_PHYSBASE5	Core	See Table 35-2.
20BH	523	IA32_MTRR_PHYSMASK5	Core	See Table 35-2.
20CH	524	IA32_MTRR_PHYSBASE6	Core	See Table 35-2.
20DH	525	IA32_MTRR_PHYSMASK6	Core	See Table 35-2.
20EH	526	IA32_MTRR_PHYSBASE7	Core	See Table 35-2.
20FH	527	IA32_MTRR_PHYSMASK7	Core	See Table 35-2.
250H	592	IA32_MTRR_FIX64K_00000	Core	See Table 35-2.
258H	600	IA32_MTRR_FIX16K_80000	Core	See Table 35-2.
259H	601	IA32_MTRR_FIX16K_A0000	Core	See Table 35-2.
268H	616	IA32_MTRR_FIX4K_C0000	Core	See Table 35-2.
269H	617	IA32_MTRR_FIX4K_C8000	Core	See Table 35-2.
26AH	618	IA32_MTRR_FIX4K_D0000	Core	See Table 35-2.
26BH	619	IA32_MTRR_FIX4K_D8000	Core	See Table 35-2.
26CH	620	IA32_MTRR_FIX4K_E0000	Core	See Table 35-2.
26DH	621	IA32_MTRR_FIX4K_E8000	Core	See Table 35-2.
26EH	622	IA32_MTRR_FIX4K_F0000	Core	See Table 35-2.
26FH	623	IA32_MTRR_FIX4K_F8000	Core	See Table 35-2.
277H	631	IA32_PAT	Core	See Table 35-2.
2FFH	767	IA32_MTRR_DEF_TYPE	Core	<b>Default Memory Types (R/W)</b> See Table 35-2.
309H	777	IA32_FIXED_CTR0	Core	<b>Fixed-Function Performance Counter Register 0 (R/W)</b> See Table 35-2.
30AH	778	IA32_FIXED_CTR1	Core	<b>Fixed-Function Performance Counter Register 1 (R/W)</b> See Table 35-2.
30BH	779	IA32_FIXED_CTR2	Core	<b>Fixed-Function Performance Counter Register 2 (R/W)</b> See Table 35-2.
345H	837	IA32_PERF_CAPABILITIES	Core	See Table 35-2. See Section 17.4.1, "IA32_DEBUGCTL MSR."
38DH	909	IA32_FIXED_CTR_CTRL	Core	<b>Fixed-Function-Counter Control Register (R/W)</b> See Table 35-2.
38FH	911	IA32_PERF_GLOBAL_CTRL	Core	See Table 35-2. See Section 18.4.2, "Global Counter Control Facilities."
3FDH	1021	MSR_CORE_C6_RESIDENCY	Core	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.

**Table 35-6. MSRs Common to the Silvermont Microarchitecture and Newer Microarchitectures for Intel Atom Processors**

Address		Register Name	Scope	Bit Description
Hex	Dec			
		63:0		CORE C6 Residency Counter. (R/O) Value since last reset that this core is in processor-specific C6 states. Counts at the TSC Frequency.
400H	1024	IA32_MCO_CTL	Module	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
401H	1025	IA32_MCO_STATUS	Module	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
402H	1026	IA32_MCO_ADDR	Module	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The IA32_MCO_ADDR register is either not implemented or contains no address if the ADDR_V flag in the IA32_MCO_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
404H	1028	IA32_MC1_CTL	Module	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
405H	1029	IA32_MC1_STATUS	Module	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
408H	1032	IA32_MC2_CTL	Module	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
409H	1033	IA32_MC2_STATUS	Module	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
40AH	1034	IA32_MC2_ADDR	Module	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The IA32_MC2_ADDR register is either not implemented or contains no address if the ADDR_V flag in the IA32_MC2_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
40CH	1036	IA32_MC3_CTL	Core	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
40DH	1037	IA32_MC3_STATUS	Core	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
40EH	1038	IA32_MC3_ADDR	Core	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The MSR_MC3_ADDR register is either not implemented or contains no address if the ADDR_V flag in the MSR_MC3_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
410H	1040	IA32_MC4_CTL	Core	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
411H	1041	IA32_MC4_STATUS	Core	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
412H	1042	IA32_MC4_ADDR	Core	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The MSR_MC4_ADDR register is either not implemented or contains no address if the ADDR_V flag in the MSR_MC4_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
414H	1044	IA32_MC5_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
415H	1045	IA32_MC5_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."

**Table 35-6. MSRs Common to the Silvermont Microarchitecture and Newer Microarchitectures for Intel Atom Processors**

Address		Register Name	Scope	Bit Description
Hex	Dec			
416H	1046	IA32_MC5_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The MSR_MC4_ADDR register is either not implemented or contains no address if the ADDR_V flag in the MSR_MC4_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
480H	1152	IA32_VMX_BASIC	Core	<b>Reporting Register of Basic VMX Capabilities (R/O)</b> See Table 35-2. See Appendix A.1, "Basic VMX Information."
481H	1153	IA32_VMX_PINBASED_CTL	Core	<b>Capability Reporting Register of Pin-based VM-execution Controls (R/O)</b> See Table 35-2. See Appendix A.3, "VM-Execution Controls."
482H	1154	IA32_VMX_PROCBASED_CTL	Core	<b>Capability Reporting Register of Primary Processor-based VM-execution Controls (R/O)</b> See Appendix A.3, "VM-Execution Controls."
483H	1155	IA32_VMX_EXIT_CTL	Core	<b>Capability Reporting Register of VM-exit Controls (R/O)</b> See Table 35-2. See Appendix A.4, "VM-Exit Controls."
484H	1156	IA32_VMX_ENTRY_CTL	Core	<b>Capability Reporting Register of VM-entry Controls (R/O)</b> See Table 35-2. See Appendix A.5, "VM-Entry Controls."
485H	1157	IA32_VMX_MISC	Core	<b>Reporting Register of Miscellaneous VMX Capabilities (R/O)</b> See Table 35-2. See Appendix A.6, "Miscellaneous Data."
486H	1158	IA32_VMX_CR0_FIXED0	Core	<b>Capability Reporting Register of CR0 Bits Fixed to 0 (R/O)</b> See Table 35-2. See Appendix A.7, "VMX-Fixed Bits in CR0."
487H	1159	IA32_VMX_CR0_FIXED1	Core	<b>Capability Reporting Register of CR0 Bits Fixed to 1 (R/O)</b> See Table 35-2. See Appendix A.7, "VMX-Fixed Bits in CR0."
488H	1160	IA32_VMX_CR4_FIXED0	Core	<b>Capability Reporting Register of CR4 Bits Fixed to 0 (R/O)</b> See Table 35-2. See Appendix A.8, "VMX-Fixed Bits in CR4."
489H	1161	IA32_VMX_CR4_FIXED1	Core	<b>Capability Reporting Register of CR4 Bits Fixed to 1 (R/O)</b> See Table 35-2. See Appendix A.8, "VMX-Fixed Bits in CR4."
48AH	1162	IA32_VMX_VMCS_ENUM	Core	<b>Capability Reporting Register of VMCS Field Enumeration (R/O)</b> See Table 35-2. See Appendix A.9, "VMCS Enumeration."

**Table 35-6. MSRs Common to the Silvermont Microarchitecture and Newer Microarchitectures for Intel Atom Processors**

Address		Register Name	Scope	Bit Description
Hex	Dec			
48BH	1163	IA32_VMX_PROCBASED_CTLSS2	Core	<b>Capability Reporting Register of Secondary Processor-based VM-execution Controls (R/O)</b> See Appendix A.3, "VM-Execution Controls."
48CH	1164	IA32_VMX_EPT_VPID_ENUM	Core	<b>Capability Reporting Register of EPT and VPID (R/O)</b> See Table 35-2
48DH	1165	IA32_VMX_TRUE_PINBASED_CTLSS	Core	<b>Capability Reporting Register of Pin-based VM-execution Flex Controls (R/O)</b> See Table 35-2
48EH	1166	IA32_VMX_TRUE_PROCBASED_CTLSS	Core	<b>Capability Reporting Register of Primary Processor-based VM-execution Flex Controls (R/O)</b> See Table 35-2
48FH	1167	IA32_VMX_TRUE_EXIT_CTLSS	Core	<b>Capability Reporting Register of VM-exit Flex Controls (R/O)</b> See Table 35-2
490H	1168	IA32_VMX_TRUE_ENTRY_CTLSS	Core	<b>Capability Reporting Register of VM-entry Flex Controls (R/O)</b> See Table 35-2
491H	1169	IA32_VMX_FMFUNC	Core	<b>Capability Reporting Register of VM-function Controls (R/O)</b> See Table 35-2
4C1H	1217	IA32_A_PMC0	Core	See Table 35-2.
4C2H	1218	IA32_A_PMC1	Core	See Table 35-2.
600H	1536	IA32_DS_AREA	Core	<b>DS Save Area (R/W)</b> See Table 35-2. See Section 18.12.4, "Debug Store (DS) Mechanism."
660H	1632	MSR_CORE_C1_RESIDENCY	Core	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		63:0		CORE C1 Residency Counter. (R/O) Value since last reset that this core is in processor-specific C1 states. Counts at the TSC frequency.
6E0H	1760	IA32_TSC_DEADLINE	Core	<b>TSC Target of Local APIC's TSC Deadline Mode (R/W)</b> See Table 35-2
C000_0080H		IA32_EFER	Core	<b>Extended Feature Enables</b> See Table 35-2.
C000_0081H		IA32_STAR	Core	<b>System Call Target Address (R/W)</b> See Table 35-2.
C000_0082H		IA32_LSTAR	Core	<b>IA-32e Mode System Call Target Address (R/W)</b> See Table 35-2.
C000_0084H		IA32_FMASK	Core	<b>System Call Flag Mask (R/W)</b> See Table 35-2.
C000_0100H		IA32_FS_BASE	Core	<b>Map of BASE Address of FS (R/W)</b> See Table 35-2.

**Table 35-6. MSRs Common to the Silvermont Microarchitecture and Newer Microarchitectures for Intel Atom Processors**

Address		Register Name	Scope	Bit Description
Hex	Dec			
C000_0101H		IA32_GS_BASE	Core	<b>Map of BASE Address of GS (R/W)</b> See Table 35-2.
C000_0102H		IA32_KERNEL_GSBASE	Core	<b>Swap Target of BASE Address of GS (R/W)</b> See Table 35-2.
C000_0103H		IA32_TSC_AUX	Core	<b>AUXILIARY TSC Signature. (R/W)</b> See Table 35-2

Table 35-7 lists model-specific registers (MSRs) that are common to Intel® Atom™ processors based on the Silvermont and Airmont microarchitectures but not newer microarchitectures.

**Table 35-7. MSRs Common to the Silvermont and Airmont Microarchitectures**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
17H	23	MSR_PLATFORM_ID	Module	<b>Model Specific Platform ID (R)</b>
		7:0		Reserved.
		12:8		<b>Maximum Qualified Ratio (R)</b> The maximum allowed bus ratio.
		49:13		Reserved.
		52:50		See Table 35-2
		63:33		Reserved.
3AH	58	IA32_FEATURE_CONTROL	Core	<b>Control Features in Intel 64Processor (R/W)</b> See Table 35-2.
		0		<b>Lock (R/WL)</b>
		1		<b>Reserved</b>
		2		<b>Enable VMX outside SMX operation (R/WL)</b>
40H	64	MSR_LASTBRANCH_0_FROM_IP	Core	<b>Last Branch Record 0 From IP (R/W)</b> One of eight pairs of last branch record registers on the last branch record stack. The From_IP part of the stack contains pointers to the <b>source instruction</b> . See also: <ul style="list-style-type: none"> <li>▪ Last Branch Record Stack TOS at 1C9H</li> <li>▪ Section 17.5 and record format in Section 17.4.8.1</li> </ul>
41H	65	MSR_LASTBRANCH_1_FROM_IP	Core	<b>Last Branch Record 1 From IP (R/W)</b> See description of MSR_LASTBRANCH_0_FROM_IP.
42H	66	MSR_LASTBRANCH_2_FROM_IP	Core	<b>Last Branch Record 2 From IP (R/W)</b> See description of MSR_LASTBRANCH_0_FROM_IP.
43H	67	MSR_LASTBRANCH_3_FROM_IP	Core	<b>Last Branch Record 3 From IP (R/W)</b> See description of MSR_LASTBRANCH_0_FROM_IP.
44H	68	MSR_LASTBRANCH_4_FROM_IP	Core	<b>Last Branch Record 4 From IP (R/W)</b> See description of MSR_LASTBRANCH_0_FROM_IP.

Table 35-7. MSRs Common to the Silvermont and Airmont Microarchitectures

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
45H	69	MSR_LASTBRANCH_5_FROM_IP	Core	<b>Last Branch Record 5 From IP (R/W)</b> See description of MSR_LASTBRANCH_0_FROM_IP.
46H	70	MSR_LASTBRANCH_6_FROM_IP	Core	<b>Last Branch Record 6 From IP (R/W)</b> See description of MSR_LASTBRANCH_0_FROM_IP.
47H	71	MSR_LASTBRANCH_7_FROM_IP	Core	<b>Last Branch Record 7 From IP (R/W)</b> See description of MSR_LASTBRANCH_0_FROM_IP.
60H	96	MSR_LASTBRANCH_0_TO_IP	Core	<b>Last Branch Record 0 To IP (R/W)</b> One of eight pairs of last branch record registers on the last branch record stack. The To_IP part of the stack contains pointers to the destination instruction.
61H	97	MSR_LASTBRANCH_1_TO_IP	Core	<b>Last Branch Record 1 To IP (R/W)</b> See description of MSR_LASTBRANCH_0_TO_IP.
62H	98	MSR_LASTBRANCH_2_TO_IP	Core	<b>Last Branch Record 2 To IP (R/W)</b> See description of MSR_LASTBRANCH_0_TO_IP.
63H	99	MSR_LASTBRANCH_3_TO_IP	Core	<b>Last Branch Record 3 To IP (R/W)</b> See description of MSR_LASTBRANCH_0_TO_IP.
64H	100	MSR_LASTBRANCH_4_TO_IP	Core	<b>Last Branch Record 4 To IP (R/W)</b> See description of MSR_LASTBRANCH_0_TO_IP.
65H	101	MSR_LASTBRANCH_5_TO_IP	Core	<b>Last Branch Record 5 To IP (R/W)</b> See description of MSR_LASTBRANCH_0_TO_IP.
66H	102	MSR_LASTBRANCH_6_TO_IP	Core	<b>Last Branch Record 6 To IP (R/W)</b> See description of MSR_LASTBRANCH_0_TO_IP.
67H	103	MSR_LASTBRANCH_7_TO_IP	Core	<b>Last Branch Record 7 To IP (R/W)</b> See description of MSR_LASTBRANCH_0_TO_IP.
CDH	205	MSR_FSB_FREQ	Module	<b>Scaleable Bus Speed(R0)</b> This field indicates the intended scaleable bus clock speed for processors based on Silvermont microarchitecture:
		2:0		<ul style="list-style-type: none"> <li>▪ 100B: 080.0 MHz</li> <li>▪ 000B: 083.3 MHz</li> <li>▪ 001B: 100.0 MHz</li> <li>▪ 010B: 133.3 MHz</li> <li>▪ 011B: 116.7 MHz</li> </ul>
		63:3		Reserved.
E2H	226	MSR_PKG_CST_CONFIG_CONTROL	Module	<b>C-State Configuration Control (R/W)</b> Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States. See <a href="http://biosbits.org">http://biosbits.org</a> .



**Table 35-7. MSRs Common to the Silvermont and Airmont Microarchitectures**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		2:0		<b>Package C-State Limit (R/W)</b> Specifies the lowest processor-specific C-state code name (consuming the least power), for the package. The default is set as factory-configured package C-state limit. The following C-state code name encodings are supported: 000b: C0 (no package C-state support) 001b: C1 (Behavior is the same as 000b) 100b: C4 110b: C6 111b: C7 (Silvermont only).
		9:3		Reserved.
		10		<b>I/O MWAIT Redirection Enable (R/W)</b> When set, will map IO_read instructions sent to IO register specified by MSR_PMG_IO_CAPTURE_BASE to MWAIT instructions
		14:11		Reserved.
		15		<b>CFG Lock (R/WO)</b> When set, lock bits 15:0 of this register until next reset.
		63:16		Reserved.
11EH	281	MSR_BBL_CR_CTL3	Module	
		0		<b>L2 Hardware Enabled (RO)</b> 1 = If the L2 is hardware-enabled 0 = Indicates if the L2 is hardware-disabled
		7:1		Reserved.
		8		<b>L2 Enabled. (R/W)</b> 1 = L2 cache has been initialized 0 = Disabled (default) Until this bit is set the processor will not respond to the WBINVD instruction or the assertion of the FLUSH# input.
		22:9		Reserved.
		23		<b>L2 Not Present (RO)</b> 0 = L2 Present 1 = L2 Not Present
		63:24		Reserved.
1A0H	416	IA32_MISC_ENABLE		<b>Enable Misc. Processor Features (R/W)</b> Allows a variety of processor functions to be enabled and disabled.
		0	Core	<b>Fast-Strings Enable</b> See Table 35-2.
		2:1		Reserved.
		3	Module	<b>Automatic Thermal Control Circuit Enable (R/W)</b> See Table 35-2. Default value is 0.
		6:4		Reserved.

Table 35-7. MSRs Common to the Silvermont and Airmont Microarchitectures

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		7	Core	<b>Performance Monitoring Available (R)</b> See Table 35-2.
		10:8		Reserved.
		11	Core	<b>Branch Trace Storage Unavailable (RO)</b> See Table 35-2.
		12	Core	<b>Processor Event Based Sampling Unavailable (RO)</b> See Table 35-2.
		15:13		Reserved.
		16	Module	<b>Enhanced Intel SpeedStep Technology Enable (R/W)</b> See Table 35-2.
		18	Core	<b>ENABLE MONITOR FSM (R/W)</b> See Table 35-2.
		21:19		Reserved.
		22	Core	<b>Limit CPUID Maxval (R/W)</b> See Table 35-2.
		23	Module	<b>xTPR Message Disable (R/W)</b> See Table 35-2.
		33:24		Reserved.
		34	Core	<b>XD Bit Disable (R/W)</b> See Table 35-2.
		37:35		Reserved.
		38	Module	<b>Turbo Mode Disable (R/W)</b> When set to 1 on processors that support Intel Turbo Boost Technology, the turbo mode feature is disabled and the IDA_Enable feature flag will be clear (CPUID.06H: EAX[1]=0). When set to a 0 on processors that support IDA, CPUID.06H: EAX[1] reports the processor's support of turbo mode is enabled. <b>Note:</b> the power-on default value is used by BIOS to detect hardware support of turbo mode. If power-on default value is 1, turbo mode is available in the processor. If power-on default value is 0, turbo mode is not available.
		63:39		Reserved.
1C8H	456	MSR_LBR_SELECT	Core	<b>Last Branch Record Filtering Select Register (R/W)</b> See Section 17.7.2, "Filtering of Last Branch Records."
		0		<b>CPL_EQ_0</b>
		1		<b>CPL_NEQ_0</b>
		2		<b>JCC</b>
		3		<b>NEAR_REL_CALL</b>
		4		<b>NEAR_IND_CALL</b>
		5		<b>NEAR_RET</b>

**Table 35-7. MSRs Common to the Silvermont and Airmont Microarchitectures**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		6		<b>NEAR_IND_JMP</b>
		7		<b>NEAR_REL_JMP</b>
		8		<b>FAR_BRANCH</b>
		63:9		Reserved.
1C9H	457	MSR_LASTBRANCH_TOS	Core	<b>Last Branch Record Stack TOS (R/W)</b> Contains an index (bits 0-2) that points to the MSR containing the most recent branch record. See MSR_LASTBRANCH_0_FROM_IP.
38EH	910	IA32_PERF_GLOBAL_STATUS	Core	See Table 35-2. See Section 18.4.2, "Global Counter Control Facilities."
390H	912	IA32_PERF_GLOBAL_OVF_CTRL	Core	See Table 35-2. See Section 18.4.2, "Global Counter Control Facilities."
3F1H	1009	MSR_PEBS_ENABLE	Core	See Table 35-2. See Section 18.4.4, "Precise Event Based Sampling (PEBS)."
		0		Enable PEBS for precise event on IA32_PMC0. (R/W)
3FAH	1018	MSR_PKG_C6_RESIDENCY	Package	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		63:0		Package C6 Residency Counter. (R/O) Value since last reset that this package is in processor-specific C6 states. Counts at the TSC Frequency.
664H	1636	MSR_MC6_RESIDENCY_COUNTER	Module	<b>Module C6 Residency Counter (R/O)</b> Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		63:0		Time that this module is in module-specific C6 states since last reset. Counts at 1 Mhz frequency.

### 35.4.1 MSRs with Model-Specific Behavior in the Silvermont Microarchitecture

Table 35-8 lists model-specific registers (MSRs) that are specific to Intel® Atom™ processor E3000 Series (CPUID signature with DisplayFamily\_DisplayModel of 06\_37H) and Intel Atom processors (CPUID signatures with DisplayFamily\_DisplayModel of 06\_4AH, 06\_5AH, 06\_5DH).

**Table 35-8. Specific MSRs Supported by Intel® Atom™ Processors with CPUID Signature 06\_37H, 06\_4AH, 06\_5AH, 06\_5DH**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
606H	1542	MSR_RAPL_POWER_UNIT	Package	<b>Unit Multipliers used in RAPL Interfaces (R/O)</b> See Section 14.9.1, "RAPL Interfaces."

**Table 35-8. Specific MSRs Supported by Intel® Atom™ Processors with CPUID Signature 06\_37H, 06\_4AH, 06\_5AH, 06\_5DH**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		3:0		Power Units. Power related information (in milliwatts) is based on the multiplier, $2^{\text{PU}}$ ; where PU is an unsigned integer represented by bits 3:0. Default value is 0101b, indicating power unit is in 32 milliwatts increment.
		7:4		Reserved
		12:8		Energy Status Units. Energy related information (in microjoules) is based on the multiplier, $2^{\text{ESU}}$ ; where ESU is an unsigned integer represented by bits 12:8. Default value is 00101b, indicating energy unit is in 32 microjoules increment.
		15:13		Reserved
		19:16		Time Unit. The value is 0000b, indicating time unit is in one second.
		63:20		Reserved
610H	1552	MSR_PKG_POWER_LIMIT	Package	<b>PKG RAPL Power Limit Control (R/W)</b>
		14:0		Package Power Limit #1. (R/W) See Section 14.9.3, "Package RAPL Domain." and MSR_RAPL_POWER_UNIT in Table 35-8.
		15		Enable Power Limit #1. (R/W) See Section 14.9.3, "Package RAPL Domain."
		16		Package Clamping Limitation #1. (R/W) See Section 14.9.3, "Package RAPL Domain."
		23:17		Time Window for Power Limit #1. (R/W) in unit of second. If 0 is specified in bits [23:17], defaults to 1 second window.
		63:24		Reserved
611H	1553	MSR_PKG_ENERGY_STATUS	Package	<b>PKG Energy Status (R/O)</b> See Section 14.9.3, "Package RAPL Domain." and MSR_RAPL_POWER_UNIT in Table 35-8
639H	1593	MSR_PPO_ENERGY_STATUS	Package	<b>PPO Energy Status (R/O)</b> See Section 14.9.4, "PPO/PP1 RAPL Domains." and MSR_RAPL_POWER_UNIT in Table 35-8

Table 35-9 lists model-specific registers (MSRs) that are specific to Intel® Atom™ processor E3000 Series (CPUID signature with DisplayFamily\_DisplayModel of 06\_37H).

**Table 35-9. Specific MSRs Supported by Intel® Atom™ Processor E3000 Series with CPUID Signature 06\_37H**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
668H	1640	MSR_CC6_DEMOTION_POLICY_CONFIG	Package	<b>Core C6 demotion policy config MSR</b>
		63:0		Controls per-core C6 demotion policy. Writing a value of 0 disables core level HW demotion policy.
669H	1641	MSR_MC6_DEMOTION_POLICY_CONFIG	Package	<b>Module C6 demotion policy config MSR</b>
		63:0		Controls module (i.e. two cores sharing the second-level cache) C6 demotion policy. Writing a value of 0 disables module level HW demotion policy.
664H	1636	MSR_MC6_RESIDENCY_COUNTER	Module	<b>Module C6 Residency Counter (R/O)</b> Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		63:0		Time that this module is in module-specific C6 states since last reset. Counts at 1 Mhz frequency.

Table 35-10 lists model-specific registers (MSRs) that are specific to Intel® Atom™ processor C2000 Series (CPUID signature with DisplayFamily\_DisplayModel of 06\_4DH).

**Table 35-10. Specific MSRs Supported by Intel® Atom™ Processor C2000 Series with CPUID Signature 06\_4DH**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
1A4H	420	MSR_MISC_FEATURE_CONTROL		<b>Miscellaneous Feature Control (R/W)</b>
		0	Core	<b>L2 Hardware Prefetcher Disable (R/W)</b> If 1, disables the L2 hardware prefetcher, which fetches additional lines of code or data into the L2 cache.
		1		<b>Reserved</b>
		2	Core	<b>DCU Hardware Prefetcher Disable (R/W)</b> If 1, disables the L1 data cache prefetcher, which fetches the next cache line into L1 data cache.
		63:3		Reserved.
1ADH	429	MSR_TURBO_RATIO_LIMIT	Package	<b>Maximum Ratio Limit of Turbo Mode (RW)</b>
		7:0	Package	<b>Maximum Ratio Limit for 1C</b> Maximum turbo ratio limit of 1 core active.
		15:8	Package	<b>Maximum Ratio Limit for 2C</b> Maximum turbo ratio limit of 2 core active.
		23:16	Package	<b>Maximum Ratio Limit for 3C</b> Maximum turbo ratio limit of 3 core active.
		31:24	Package	<b>Maximum Ratio Limit for 4C</b> Maximum turbo ratio limit of 4 core active.

**Table 35-10. Specific MSRs Supported by Intel® Atom™ Processor C2000 Series (Contd.)with CPUID Signature**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		39:32	Package	<b>Maximum Ratio Limit for 5C</b> Maximum turbo ratio limit of 5 core active.
		47:40	Package	<b>Maximum Ratio Limit for 6C</b> Maximum turbo ratio limit of 6 core active.
		55:48	Package	<b>Maximum Ratio Limit for 7C</b> Maximum turbo ratio limit of 7 core active.
		63:56	Package	<b>Maximum Ratio Limit for 8C</b> Maximum turbo ratio limit of 8 core active.
606H	1542	MSR_RAPL_POWER_UNIT	Package	<b>Unit Multipliers used in RAPL Interfaces (R/O)</b> See Section 14.9.1, "RAPL Interfaces."
		3:0		Power Units. Power related information (in milliWatts) is based on the multiplier, $2^{\text{PU}}$ ; where PU is an unsigned integer represented by bits 3:0. Default value is 0101b, indicating power unit is in 32 milliWatts increment.
		7:4		Reserved
		12:8		Energy Status Units. Energy related information (in microjoules) is based on the multiplier, $2^{\text{ESU}}$ ; where ESU is an unsigned integer represented by bits 12:8. Default value is 00101b, indicating energy unit is in 32 microjoules increment.
		15:13		Reserved
		19:16		Time Unit. The value is 0000b, indicating time unit is in one second.
		63:20		Reserved
		610H	1552	MSR_PKG_POWER_LIMIT
66EH	1646	MSR_PKG_POWER_INFO	Package	<b>PKG RAPL Parameter (R/O)</b>
		14:0		Thermal Spec Power. (R/O) The unsigned integer value is the equivalent of thermal specification power of the package domain. The unit of this field is specified by the "Power Units" field of MSR_RAPL_POWER_UNIT
		63:15		Reserved

### 35.4.2 MSRs In Intel Atom Processors Based on Airmont Microarchitecture

Intel Atom processor X7-Z8000 and X5-Z8000 series are based on the Airmont microarchitecture. These processors support MSRs listed in Table 35-6, Table 35-7, Table 35-8, and Table 35-11. These processors have a CPUID signature with DisplayFamily\_DisplayModel including 06\_4CH; see Table 35-1.

**Table 35-11. MSRs in Intel Atom Processors Based on the Airmont Microarchitecture**

Address		Register Name	Scope	Bit Description
Hex	Dec			
CDH	205	MSR_FSB_FREQ	Module	<b>Scaleable Bus Speed(R0)</b> This field indicates the intended scaleable bus clock speed for processors based on Airmont microarchitecture:
		3:0		<ul style="list-style-type: none"> <li>▪ 0000B: 083.3 MHz</li> <li>▪ 0001B: 100.0 MHz</li> <li>▪ 0010B: 133.3 MHz</li> <li>▪ 0011B: 116.7 MHz</li> <li>▪ 0100B: 080.0 MHz</li> <li>▪ 0101B: 093.3 MHz</li> <li>▪ 0110B: 090.0 MHz</li> <li>▪ 0111B: 088.9 MHz</li> <li>▪ 10sure00B: 087.5 MHz</li> </ul>
		63:5		Reserved.
E2H	226	MSR_PKG_CST_CONFIG_CONTROL	Module	<b>C-State Configuration Control (R/W)</b> Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States. See <a href="http://biosbits.org">http://biosbits.org</a> .
		2:0		<b>Package C-State Limit (R/W)</b> Specifies the lowest processor-specific C-state code name (consuming the least power), for the package. The default is set as factory-configured package C-state limit. The following C-state code name encodings are supported: 000b: No limit 001b: C1 010b: C2 110b: C6 111b: C7
		9:3		Reserved.
		10		<b>I/O MWAIT Redirection Enable (R/W)</b> When set, will map IO_read instructions sent to IO register specified by MSR_PMG_IO_CAPTURE_BASE to MWAIT instructions
		14:11		Reserved.
		15		<b>CFG Lock (R/W0)</b> When set, lock bits 15:0 of this register until next reset.
		63:16		Reserved.
E4H	228	MSR_PMG_IO_CAPTURE_BASE	Module	<b>Power Management IO Redirection in C-state (R/W)</b> See <a href="http://biosbits.org">http://biosbits.org</a> .
		15:0		<b>LVL_2 Base Address (R/W)</b> Specifies the base address visible to software for IO redirection. If IO MWAIT Redirection is enabled, reads to this address will be consumed by the power management logic and decoded to MWAIT instructions. When IO port address redirection is enabled, this is the IO port address reported to the OS/software.

Table 35-11. MSRs in Intel Atom Processors Based on the Airmont Microarchitecture (Contd.)

Address		Register Name	Scope	Bit Description
Hex	Dec			
		18:16		<b>C-state Range (R/W)</b> Specifies the encoding value of the maximum C-State code name to be included when IO read to MWAIT redirection is enabled by MSR_PKG_CST_CONFIG_CONTROL[bit10]: 000b - C3 is the max C-State to include 001b - Deep Power Down Technology is the max C-State 010b - C7 is the max C-State to include
		63:19		Reserved.
638H	1592	MSR_PP0_POWER_LIMIT	Package	<b>PP0 RAPL Power Limit Control (R/W)</b>
		14:0		PP0 Power Limit #1. (R/W) See Section 14.9.4, "PP0/PP1 RAPL Domains." and MSR_RAPL_POWER_UNIT in Table 35-8.
		15		Enable Power Limit #1. (R/W) See Section 14.9.4, "PP0/PP1 RAPL Domains."
		16		Reserved
		23:17		Time Window for Power Limit #1. (R/W) Specifies the time duration over which the average power must remain below PP0_POWER_LIMIT #1(14:0). Supported Encodings: 0x0: 1 second time duration. 0x1: 5 second time duration (Default). 0x2: 10 second time duration. 0x3: 15 second time duration. 0x4: 20 second time duration. 0x5: 25 second time duration. 0x6: 30 second time duration. 0x7: 35 second time duration. 0x8: 40 second time duration. 0x9: 45 second time duration. 0xA: 50 second time duration. 0xB-0x7F - reserved.
		63:24		Reserved

## 35.5 MSRS IN NEXT GENERATION INTEL ATOM PROCESSORS

Next Generation Intel Atom processors are based on the Goldmont microarchitecture. These processors support MSRs listed in Table 35-6 and Table 35-12. These processors have a CPUID signature with DisplayFamily\_DisplayModel including 06\_5CH; see Table 35-1.

In the Goldmont microarchitecture, the scope column indicates the following: "Core" means each processor core has a separate MSR, or a bit field not shared with another processor core. "Module" means the MSR or the bit field is shared by a pair of processor cores in the physical package. "Package" means all processor cores in the physical package share the same MSR or bit interface.



**Table 35-12. MSRs in Next Generation Intel Atom Processors Based on the Goldmont Microarchitecture**

Address		Register Name	Scope	Bit Description
Hex	Dec			
17H	23	MSR_PLATFORM_ID	Module	<b>Model Specific Platform ID (R)</b>
		49:0		Reserved.
		52:50		See Table 35-2.
		63:33		Reserved.
3AH	58	IA32_FEATURE_CONTROL	Core	<b>Control Features in Intel 64Processor (R/W)</b> See Table 35-2.
		0		<b>Lock (R/WL)</b>
		1		<b>Enable VMX inside SMX operation (R/WL)</b>
		2		<b>Enable VMX outside SMX operation (R/WL)</b>
		14:8		<b>SENTER local functions enables (R/WL)</b>
		15		<b>SENTER global functions enable (R/WL)</b>
		18		<b>SGX global functions enable (R/WL)</b>
		63:19		Reserved.
3BH	59	IA32_TSC_ADJUST	Core	<b>Per-Core TSC ADJUST (R/W)</b> See Table 35-2.
C3H	195	IA32_PMC2	Core	<b>Performance Counter Register</b> See Table 35-2.
C4H	196	IA32_PMC3	Core	<b>Performance Counter Register</b> See Table 35-2.
CEH	206	MSR_PLATFORM_INFO	Package	See <a href="http://biosbits.org">http://biosbits.org</a> .
		7:0		Reserved.
		15:8	Package	<b>Maximum Non-Turbo Ratio (R/O)</b> The is the ratio of the frequency that invariant TSC runs at. Frequency = ratio * 100 MHz.
		27:16		Reserved.
		28	Package	<b>Programmable Ratio Limit for Turbo Mode (R/O)</b> When set to 1, indicates that Programmable Ratio Limits for Turbo mode is enabled, and when set to 0, indicates Programmable Ratio Limits for Turbo mode is disabled.
		29	Package	<b>Programmable TDP Limit for Turbo Mode (R/O)</b> When set to 1, indicates that TDP Limits for Turbo mode are programmable, and when set to 0, indicates TDP Limit for Turbo mode is not programmable.
		30	Package	<b>Programmable TJ OFFSET (R/O)</b> When set to 1, indicates that MSR_TEMPERATURE_TARGET.[27:24] is valid and writable to specify an temperature offset.
39:31		Reserved.		

**Table 35-12. MSRs in Next Generation Intel Atom Processors Based on the Goldmont Microarchitecture (Contd.)**

Address		Register Name	Scope	Bit Description
Hex	Dec			
		47:40	Package	<b>Maximum Efficiency Ratio (R/O)</b> The is the minimum ratio (maximum efficiency) that the processor can operates, in units of 100MHz.
		63:48		Reserved.
E2H	226	MSR_PKG_CST_CONFIG_CONTROL	Core	<b>C-State Configuration Control (R/W)</b> Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States. See <a href="http://biosbits.org">http://biosbits.org</a> .
		3:0		<b>Package C-State Limit (R/W)</b> Specifies the lowest processor-specific C-state code name (consuming the least power), for the package. The default is set as factory-configured package C-state limit. The following C-state code name encodings are supported: 0000b: No limit 0001b: C1 0010b: C3 0011b: C6 0100b: C7 0101b: C7S 0110b: C8 0111b: C9 1000b: C10
		9:3		Reserved.
		10		<b>I/O MWAIT Redirection Enable (R/W)</b> When set, will map IO_read instructions sent to IO register specified by MSR_PMG_IO_CAPTURE_BASE to MWAIT instructions
		14:11		Reserved.
		15		<b>CFG Lock (R/WO)</b> When set, lock bits 15:0 of this register until next reset.
		63:16		Reserved.
17DH	381	MSR_SMM_MCA_CAP	Core	<b>Enhanced SMM Capabilities (SMM-RO)</b> Reports SMM capability Enhancement. Accessible only while in SMM.
		57:0		<b>Reserved</b>
		58		<b>SMM_Code_Access_Chk (SMM-RO)</b> If set to 1 indicates that the SMM code access restriction is supported and the MSR_SMM_FEATURE_CONTROL is supported.
		59		<b>Long_Flow_Indication (SMM-RO)</b> If set to 1 indicates that the SMM long flow indicator is supported and the MSR_SMM_DELAYED is supported.
		63:60		Reserved
188H	392	IA32_PERFEVTSEL2	Core	See Table 35-2.

**Table 35-12. MSRs in Next Generation Intel Atom Processors Based on the Goldmont Microarchitecture (Contd.)**

Address		Register Name	Scope	Bit Description
Hex	Dec			
189H	393	IA32_PERFEVTSEL3	Core	See Table 35-2.
1A0H	416	IA32_MISC_ENABLE		<b>Enable Misc. Processor Features (R/W)</b> Allows a variety of processor functions to be enabled and disabled.
		0	Core	<b>Fast-Strings Enable</b> See Table 35-2.
		2:1		Reserved.
		3	Package	<b>Automatic Thermal Control Circuit Enable (R/W)</b> See Table 35-2. Default value is 1.
		6:4		Reserved.
		7	Core	<b>Performance Monitoring Available (R)</b> See Table 35-2.
		10:8		Reserved.
		11	Core	<b>Branch Trace Storage Unavailable (RO)</b> See Table 35-2.
		12	Core	<b>Processor Event Based Sampling Unavailable (RO)</b> See Table 35-2.
		15:13		Reserved.
		16	Package	<b>Enhanced Intel SpeedStep Technology Enable (R/W)</b> See Table 35-2.
		18	Core	<b>ENABLE MONITOR FSM (R/W)</b> See Table 35-2.
		21:19		Reserved.
		22	Core	<b>Limit CPUID Maxval (R/W)</b> See Table 35-2.
		23	Package	<b>xTPR Message Disable (R/W)</b> See Table 35-2.
		33:24		Reserved.
		34	Core	<b>XD Bit Disable (R/W)</b> See Table 35-2.
37:35		Reserved.		
38	Package	<b>Turbo Mode Disable (R/W)</b> When set to 1 on processors that support Intel Turbo Boost Technology, the turbo mode feature is disabled and the IDA_Enable feature flag will be clear (CPUID.06H: EAX[1]=0). When set to a 0 on processors that support IDA, CPUID.06H: EAX[1] reports the processor's support of turbo mode is enabled. <b>Note:</b> the power-on default value is used by BIOS to detect hardware support of turbo mode. If power-on default value is 1, turbo mode is available in the processor. If power-on default value is 0, turbo mode is not available.		
63:39		Reserved.		

**Table 35-12. MSRs in Next Generation Intel Atom Processors Based on the Goldmont Microarchitecture (Contd.)**

Address		Register Name	Scope	Bit Description
Hex	Dec			
1A4H	420	MSR_MISC_FEATURE_CONTROL		<b>Miscellaneous Feature Control (R/W)</b>
		0	Core	<b>L2 Hardware Prefetcher Disable (R/W)</b> If 1, disables the L2 hardware prefetcher, which fetches additional lines of code or data into the L2 cache.
		1		<b>Reserved</b>
		2	Core	<b>DCU Hardware Prefetcher Disable (R/W)</b> If 1, disables the L1 data cache prefetcher, which fetches the next cache line into L1 data cache.
		63:3		Reserved.
1AAH	426	MSR_MISC_PWR_MGMT	Package	See <a href="http://biosbits.org">http://biosbits.org</a> .
		0		<b>EIST Hardware Coordination Disable (R/W)</b> When 0, enables hardware coordination of Enhanced Intel Speedstep Technology request from processor cores; When 1, disables hardware coordination of Enhanced Intel Speedstep Technology requests.
		21:1		Reserved.
		22		<b>Thermal Interrupt Coordination Enable (R/W)</b> If set, then thermal interrupt on one core is routed to all cores.
		63:23		Reserved.
1ADH	429	MSR_TURBO_RATIO_LIMIT	Package	<b>Maximum Ratio Limit of Turbo Mode by Core Groups (RW)</b> <b>Specifies Maximum Ratio Limit for each Core Group. Max ratio for groups with more cores must decrease monotonically.</b> <b>For groups with less than 4 cores, the max ratio must be 32 or less. For groups with 4-5 cores, the max ratio must be 22 or less. For groups with more than 5 cores, the max ratio must be 16 or less.</b>
		7:0	Package	<b>Maximum Ratio Limit for Active cores in Group 0</b> Maximum turbo ratio limit when number of active cores is less or equal to Group 0 threshold.
		15:8	Package	<b>Maximum Ratio Limit for Active cores in Group 1</b> Maximum turbo ratio limit when number of active cores is less or equal to Group 1 threshold and greater than Group 0 threshold.
		23:16	Package	<b>Maximum Ratio Limit for Active cores in Group 2</b> Maximum turbo ratio limit when number of active cores is less or equal to Group 2 threshold and greater than Group 1 threshold.
		31:24	Package	<b>Maximum Ratio Limit for Active cores in Group 3</b> Maximum turbo ratio limit when number of active cores is less or equal to Group 3 threshold and greater than Group 2 threshold.
		39:32	Package	<b>Maximum Ratio Limit for Active cores in Group 4</b> Maximum turbo ratio limit when number of active cores is less or equal to Group 4 threshold and greater than Group 3 threshold.

**Table 35-12. MSRs in Next Generation Intel Atom Processors Based on the Goldmont Microarchitecture (Contd.)**

Address		Register Name	Scope	Bit Description
Hex	Dec			
		47:40	Package	<b>Maximum Ratio Limit for Active cores in Group 5</b> Maximum turbo ratio limit when number of active cores is less or equal to Group 5 threshold and greater than Group 4 threshold.
		55:48	Package	<b>Maximum Ratio Limit for Active cores in Group 6</b> Maximum turbo ratio limit when number of active cores is less or equal to Group 6 threshold and greater than Group 5 threshold.
		63:56	Package	<b>Maximum Ratio Limit for Active cores in Group 7</b> Maximum turbo ratio limit when number of active cores is less or equal to Group 7 threshold and greater than Group 6 threshold.
1AEH	430	MSR_TURBO_GROUP_CORE CNT	Package	<b>Group Size of Active Cores for Turbo Mode Operation (RW)</b> <b>Writes of 0 threshold is ignored</b>
		7:0	Package	<b>Group 0 Core Count Threshold</b> Maximum number of active cores to operate under Group 0 Max Turbo Ratio limit.
		15:8	Package	<b>Group 1 Core Count Threshold</b> Maximum number of active cores to operate under Group 1 Max Turbo Ratio limit. Must be greater than Group 0 Core Count.
		23:16	Package	<b>Group 2 Core Count Threshold</b> Maximum number of active cores to operate under Group 2 Max Turbo Ratio limit. Must be greater than Group 1 Core Count.
		31:24	Package	<b>Group 3 Core Count Threshold</b> Maximum number of active cores to operate under Group 3 Max Turbo Ratio limit. Must be greater than Group 2 Core Count.
		39:32	Package	<b>Group 4 Core Count Threshold</b> Maximum number of active cores to operate under Group 4 Max Turbo Ratio limit. Must be greater than Group 3 Core Count.
		47:40	Package	<b>Group 5 Core Count Threshold</b> Maximum number of active cores to operate under Group 5 Max Turbo Ratio limit. Must be greater than Group 4 Core Count.
		55:48	Package	<b>Group 6 Core Count Threshold</b> Maximum number of active cores to operate under Group 6 Max Turbo Ratio limit. Must be greater than Group 5 Core Count.
		63:56	Package	<b>Group 7 Core Count Threshold</b> Maximum number of active cores to operate under Group 7 Max Turbo Ratio limit. Must be greater than Group 6 Core Count and not less than the total number of processor cores in the package. E.g. specify 255.
1C8H	456	MSR_LBR_SELECT	Core	<b>Last Branch Record Filtering Select Register (R/W)</b> See Section 17.7.2, "Filtering of Last Branch Records."
		0		<b>CPL_EQ_0</b>
		1		<b>CPL_NEQ_0</b>
		2		<b>JCC</b>
		3		<b>NEAR_REL_CALL</b>

**Table 35-12. MSRs in Next Generation Intel Atom Processors Based on the Goldmont Microarchitecture (Contd.)**

Address		Register Name	Scope	Bit Description
Hex	Dec			
		4		<b>NEAR_IND_CALL</b>
		5		<b>NEAR_RET</b>
		6		<b>NEAR_IND_JMP</b>
		7		<b>NEAR_REL_JMP</b>
		8		<b>FAR_BRANCH</b>
		9		<b>EN_CALL_STACK</b>
		63:10		Reserved.
1C9H	457	MSR_LASTBRANCH_TOS	Core	<b>Last Branch Record Stack TOS (R/W)</b> Contains an index (bits 0-4) that points to the MSR containing the most recent branch record. See MSR_LASTBRANCH_0_FROM_IP.
1FCH	508	MSR_POWER_CTL	Core	Power Control Register. See <a href="http://biosbits.org">http://biosbits.org</a> .
		0		Reserved.
		1	Package	<b>C1E Enable (R/W)</b> When set to '1', will enable the CPU to switch to the Minimum Enhanced Intel SpeedStep Technology operating point when all execution cores enter MWAIT (C1).
		63:2		Reserved.
210H	528	IA32_MTRR_PHYSBASE8	Core	See Table 35-2.
211H	529	IA32_MTRR_PHYSMASK8	Core	See Table 35-2.
212H	530	IA32_MTRR_PHYSBASE9	Core	See Table 35-2.
213H	531	IA32_MTRR_PHYSMASK9	Core	See Table 35-2.
280H	640	IA32_MC0_CTL2	Module	See Table 35-2.
281H	641	IA32_MC1_CTL2	Module	See Table 35-2.
282H	642	IA32_MC2_CTL2	Core	See Table 35-2.
283H	643	IA32_MC3_CTL2	Module	See Table 35-2.
284H	644	IA32_MC4_CTL2	Package	See Table 35-2.
285H	645	IA32_MC5_CTL2	Package	See Table 35-2.
286H	646	IA32_MC6_CTL2	Package	See Table 35-2.
300H	768	MSR_SGXOWNER0	Package	<b>Lower 64 Bit OwnerEpoch Component of SGX Key (RO).</b>
		63:0		<b>Low 64 bits of an 128-bit external entropy value for key derivation of an enclave.</b>
301H	769	MSR_SGXOWNER1	Package	<b>Upper 64 Bit OwnerEpoch Component of SGX Key (RO).</b>
		63:0		<b>Upper 64 bits of an 128-bit external entropy value for key derivation of an enclave.</b>
38EH	910	IA32_PERF_GLOBAL_STATUS	Core	See Table 35-2. See Section 18.2.2.3, "Full-Width Writes to Performance Counter Registers."
		0		<b>Ovf_PMC0</b>
		1		<b>Ovf_PMC1</b>
		2		<b>Ovf_PMC2</b>

**Table 35-12. MSRs in Next Generation Intel Atom Processors Based on the Goldmont Microarchitecture (Contd.)**

Address		Register Name	Scope	Bit Description
Hex	Dec			
		3		<b>Ovf_PMC3</b>
		31:4		Reserved.
		32		<b>Ovf_FixedCtr0</b>
		33		<b>Ovf_FixedCtr1</b>
		34		<b>Ovf_FixedCtr2</b>
		54:35		Reserved.
		55		<b>Trace_ToPA_PMI.</b>
		57:56		Reserved.
		58		<b>LBR_Frz.</b>
		59		<b>CTR_Frz.</b>
		60		<b>ASCI.</b>
		61		<b>Ovf_Uncore</b>
		62		<b>Ovf_BufDSSAVE</b>
		63		<b>CondChgd</b>
390H	912	IA32_PERF_GLOBAL_STAT US_RESET	Core	See Table 35-2. See Section 18.2.2.3, "Full-Width Writes to Performance Counter Registers."
		0		<b>Set 1 to clear Ovf_PMC0</b>
		1		<b>Set 1 to clear Ovf_PMC1</b>
		2		<b>Set 1 to clear Ovf_PMC2</b>
		3		<b>Set 1 to clear Ovf_PMC3</b>
		31:4		Reserved.
		32		<b>Set 1 to clear Ovf_FixedCtr0</b>
		33		<b>Set 1 to clear Ovf_FixedCtr1</b>
		34		<b>Set 1 to clear Ovf_FixedCtr2</b>
		54:35		Reserved.
		55		<b>Set 1 to clear Trace_ToPA_PMI.</b>
		57:56		Reserved.
		58		<b>Set 1 to clear LBR_Frz.</b>
		59		<b>Set 1 to clear CTR_Frz.</b>
		60		<b>Set 1 to clear ASCII.</b>
		61		<b>Set 1 to clear Ovf_Uncore</b>
		62		<b>Set 1 to clear Ovf_BufDSSAVE</b>
		63		<b>Set 1 to clear CondChgd</b>
391H	913	IA32_PERF_GLOBAL_STAT US_SET	Core	See Table 35-2. See Section 18.2.2.3, "Full-Width Writes to Performance Counter Registers."
		0		<b>Set 1 to cause Ovf_PMC0 = 1</b>
		1		<b>Set 1 to cause Ovf_PMC1 = 1</b>
		2		<b>Set 1 to cause Ovf_PMC2 = 1</b>

**Table 35-12. MSRs in Next Generation Intel Atom Processors Based on the Goldmont Microarchitecture (Contd.)**

Address		Register Name	Scope	Bit Description
Hex	Dec			
		3		Set 1 to cause Ovf_PMC3 = 1
		31:4		Reserved.
		32		Set 1 to cause Ovf_FixedCtr0 = 1
		33		Set 1 to cause Ovf_FixedCtr1 = 1
		34		Set 1 to cause Ovf_FixedCtr2 = 1
		54:35		Reserved.
		55		Set 1 to cause Trace_ToPA_PMI = 1
		57:56		Reserved.
		58		Set 1 to cause LBR_Frz = 1
		59		Set 1 to cause CTR_Frz = 1
		60		Set 1 to cause ASCI = 1
		61		Set 1 to cause Ovf_Uncore
		62		Set 1 to cause Ovf_BufDSSAVE
		63		Reserved.
392H	914	IA32_PERF_GLOBAL_INUSE		See Table 35-2.
3F1H	1009	MSR_PEBS_ENABLE	Core	See Table 35-2. See Section 18.4.4, "Precise Event Based Sampling (PEBS)."
		0		Enable PEBS trigger and recording for the programmed event (precise or otherwise) on IA32_PMC0. (R/W)
3F8H	1016	MSR_PKG_C3_RESIDENCY	Package	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		63:0		Package C3 Residency Counter. (R/O) Value since last reset that this package is in processor-specific C3 states. Count at the same frequency as the TSC.
3F9H	1017	MSR_PKG_C6_RESIDENCY	Package	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		63:0		Package C6 Residency Counter. (R/O) Value since last reset that this package is in processor-specific C6 states. Count at the same frequency as the TSC.
3FCH	1020	MSR_CORE_C3_RESIDENCY	Core	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		63:0		CORE C3 Residency Counter. (R/O) Value since last reset that this core is in processor-specific C3 states. Count at the same frequency as the TSC.



**Table 35-12. MSRs in Next Generation Intel Atom Processors Based on the Goldmont Microarchitecture (Contd.)**

Address		Register Name	Scope	Bit Description
Hex	Dec			
406H	1030	IA32_MC1_ADDR	Module	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The IA32_MC2_ADDR register is either not implemented or contains no address if the ADDR_V flag in the IA32_MC2_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
418H	1048	IA32_MC6_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
419H	1049	IA32_MC6_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 16.
41AH	1050	IA32_MC6_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
4C3H	1219	IA32_A_PMC2	Core	See Table 35-2.
4C4H	1220	IA32_A_PMC3	Core	See Table 35-2.
4E0H	1248	MSR_SMM_FEATURE_CTRL	Package	<b>Enhanced SMM Feature Control (SMM-RW)</b> Reports SMM capability Enhancement. Accessible only while in SMM.
		0		<b>Lock (SMM-RWO)</b> When set to '1' locks this register from further changes
		1		Reserved
		2		<b>SMM_Code_Chk_En (SMM-RW)</b> This control bit is available only if MSR_SMM_MCA_CAP[58] == 1. When set to '0' (default) none of the logical processors are prevented from executing SMM code outside the ranges defined by the SMRR. When set to '1' any logical processor in the package that attempts to execute SMM code not within the ranges defined by the SMRR will assert an unrecoverable MCE.
	63:3		Reserved	
4E2H	1250	MSR_SMM_DELAYED	Package	<b>SMM Delayed (SMM-RO)</b> Reports the interruptible state of all logical processors in the package. Available only while in SMM and MSR_SMM_MCA_CAP[LONG_FLOW_INDICATION] == 1.
		N-1:0		<b>LOG_PROC_STATE (SMM-RO)</b> Each bit represents a processor core of its state in a long flow of internal operation which delays servicing an interrupt. The corresponding bit will be set at the start of long events such as: Microcode Update Load, C6, WBINVD, Ratio Change, Throttle. The bit is automatically cleared at the end of each long event. The reset value of this field is 0. Only bit positions below N = CPUID.(EAX=0BH, ECX=PKG_LVL):EBX[15:0] can be updated.
		63:N		Reserved
4E3H	1251	MSR_SMM_BLOCKED	Package	<b>SMM Blocked (SMM-RO)</b> Reports the blocked state of all logical processors in the package. Available only while in SMM.

**Table 35-12. MSRs in Next Generation Intel Atom Processors Based on the Goldmont Microarchitecture (Contd.)**

Address		Register Name	Scope	Bit Description
Hex	Dec			
		N-1:0		<b>LOG_PROC_STATE (SMM-RO)</b> Each bit represents a processor core of its blocked state to service an SMI. The corresponding bit will be set if the logical processor is in one of the following states: Wait For SIPI or SENTER Sleep. The reset value of this field is 0FFFH. Only bit positions below N = CPUID.(EAX=0BH, ECX=PKG_LVL):EBX[15:0] can be updated.
		63:N		Reserved
500H	1280	IA32_SGX_SVN_STATUS	Core	<b>Status and SVN Threshold of SGX Support for ACM (RO).</b>
		0		<b>Lock.</b> See Section 42.11.3, "Interactions with Authenticated Code Modules (ACMs)"
		15:1		Reserved.
		23:16		<b>SGX_SVN_SINIT.</b> See Section 42.11.3, "Interactions with Authenticated Code Modules (ACMs)"
		63:24		Reserved.
560H	1376	IA32_RTIT_OUTPUT_BASE	Core	<b>Trace Output Base Register (R/W).</b> See Table 35-2.
561H	1377	IA32_RTIT_OUTPUT_MASK_PTRS	Core	<b>Trace Output Mask Pointers Register (R/W).</b> See Table 35-2.
570H	1392	IA32_RTIT_CTL	Core	<b>Trace Control Register (R/W)</b>
		0		<b>TraceEn</b>
		1		<b>CYCEn</b>
		2		<b>OS</b>
		3		<b>User</b>
		6:4		Reserved, MBZ
		7		<b>CR3 filter</b>
		8		<b>ToPA; writing 0 will #GP if also setting TraceEn</b>
		9		<b>MTCEn</b>
		10		<b>TSCEn</b>
		11		<b>DisRETC</b>
		12		Reserved, MBZ
		13		<b>BranchEn</b>
		17:14		<b>MTCFreq</b>
		18		Reserved, MBZ
		22:19		<b>CYCThresh</b>
		23		Reserved, MBZ
		27:24		<b>PSBFreq</b>
		31:28		Reserved, MBZ
35:32		<b>ADDR0_CFG</b>		
39:36		<b>ADDR1_CFG</b>		
63:40		Reserved, MBZ.		

**Table 35-12. MSRs in Next Generation Intel Atom Processors Based on the Goldmont Microarchitecture (Contd.)**

Address		Register Name	Scope	Bit Description
Hex	Dec			
571H	1393	IA32_RTIT_STATUS	Core	<b>Tracing Status Register (R/W)</b>
		0		<b>FilterEn</b> , writes ignored.
		1		<b>ContexEn</b> , writes ignored.
		2		<b>TriggerEn</b> , writes ignored.
		3		Reserved
		4		<b>Error (R/W)</b>
		5		<b>Stopped</b>
		31:6		Reserved. MBZ
		48:32		<b>PacketByteCnt</b>
63:49		Reserved, MBZ.		
572H	1394	IA32_RTIT_CR3_MATCH	Core	<b>Trace Filter CR3 Match Register (R/W)</b>
		4:0		Reserved
		63:5		CR3[63:5] value to match
580H	1408	IA32_RTIT_ADDRO_A	Core	<b>Region 0 Start Address (R/W)</b>
		63:0		See Table 35-2.
581H	1409	IA32_RTIT_ADDRO_B	Core	<b>Region 0 End Address (R/W)</b>
		63:0		See Table 35-2.
582H	1410	IA32_RTIT_ADDR1_A	Core	<b>Region 1 Start Address (R/W)</b>
		63:0		See Table 35-2.
583H	1411	IA32_RTIT_ADDR1_B	Core	<b>Region 1 End Address (R/W)</b>
		63:0		See Table 35-2.
606H	1542	MSR_RAPL_POWER_UNIT	Package	<b>Unit Multipliers used in RAPL Interfaces (R/O)</b> See Section 14.9.1, "RAPL Interfaces."
		3:0		Power Units. Power related information (in Watts) is in unit of, $1W/2^{PU}$ ; where PU is an unsigned integer represented by bits 3:0. Default value is 1000b, indicating power unit is in 3.9 milliWatts increment.
		7:4		Reserved
		12:8		Energy Status Units. Energy related information (in Joules) is in unit of, $1Joule/(2^{ESU})$ ; where ESU is an unsigned integer represented by bits 12:8. Default value is 01110b, indicating energy unit is in 61 microJoules.
		15:13		Reserved
		19:16		Time Unit. Time related information (in seconds) is in unit of, $1S/2^{TU}$ ; where TU is an unsigned integer represented by bits 19:16. Default value is 1010b, indicating power unit is in 0.977 millisecond.
		63:20		Reserved

**Table 35-12. MSRs in Next Generation Intel Atom Processors Based on the Goldmont Microarchitecture (Contd.)**

Address		Register Name	Scope	Bit Description
Hex	Dec			
60AH	1546	MSR_PKGC3_IRTL	Package	<b>Package C3 Interrupt Response Limit (R/W)</b> Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		9:0		<b>Interrupt response time limit (R/W)</b> Specifies the limit that should be used to decide if the package should be put into a package C3 state.
		12:10		<b>Time Unit (R/W)</b> Specifies the encoding value of time unit of the interrupt response time limit. See Table 35-18 for supported time unit encodings.
		14:13		Reserved.
		15		<b>Valid (R/W)</b> Indicates whether the values in bits 12:0 are valid and can be used by the processor for package C-state management.
		63:16		Reserved.
60BH	1547	MSR_PKGC_IRTL1	Package	<b>Package C6/C7S Interrupt Response Limit 1 (R/W)</b> This MSR defines the interrupt response time limit used by the processor to manage transition to package C6 or C7S state. Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		9:0		<b>Interrupt response time limit (R/W)</b> Specifies the limit that should be used to decide if the package should be put into a package C6 or C7S state.
		12:10		<b>Time Unit (R/W)</b> Specifies the encoding value of time unit of the interrupt response time limit. See Table 35-18 for supported time unit encodings
		14:13		Reserved.
		15		<b>Valid (R/W)</b> Indicates whether the values in bits 12:0 are valid and can be used by the processor for package C-state management.
		63:16		Reserved.
60CH	1548	MSR_PKGC_IRTL2	Package	<b>Package C7 Interrupt Response Limit 2 (R/W)</b> This MSR defines the interrupt response time limit used by the processor to manage transition to package C7 state. Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		9:0		<b>Interrupt response time limit (R/W)</b> Specifies the limit that should be used to decide if the package should be put into a package C7 state.
		12:10		<b>Time Unit (R/W)</b> Specifies the encoding value of time unit of the interrupt response time limit. See Table 35-18 for supported time unit encodings

**Table 35-12. MSRs in Next Generation Intel Atom Processors Based on the Goldmont Microarchitecture (Contd.)**

Address		Register Name	Scope	Bit Description
Hex	Dec			
		14:13		Reserved.
		15		<b>Valid (R/W)</b> Indicates whether the values in bits 12:0 are valid and can be used by the processor for package C-state management.
		63:16		Reserved.
60DH	1549	MSR_PKG_C2_RESIDENCY	Package	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		63:0		<b>Package C2 Residency Counter. (R/O)</b> Value since last reset that this package is in processor-specific C2 states. Count at the same frequency as the TSC.
610H	1552	MSR_PKG_POWER_LIMIT	Package	<b>PKG RAPL Power Limit Control (R/W)</b> See Section 14.9.3, "Package RAPL Domain."
611H	1553	MSR_PKG_ENERGY_STATUS	Package	<b>PKG Energy Status (R/O)</b> See Section 14.9.3, "Package RAPL Domain."
613H	1555	MSR_PKG_PERF_STATUS	Package	<b>PKG Perf Status (R/O)</b> See Section 14.9.3, "Package RAPL Domain."
614H	1556	MSR_PKG_POWER_INFO	Package	<b>PKG RAPL Parameters (R/W)</b>
		14:0		<b>Thermal Spec Power (R/W)</b> See Section 14.9.3, "Package RAPL Domain."
		15		Reserved.
		30:16		<b>Minimum Power (R/W)</b> See Section 14.9.3, "Package RAPL Domain."
		31		Reserved.
		46:32		<b>Maximum Power (R/W)</b> See Section 14.9.3, "Package RAPL Domain."
		47		Reserved.
		54:48		<b>Maximum Time Window (R/W)</b> Specified by $2^Y * (1.0 + Z/4.0) * \text{Time\_Unit}$ , where "Y" is the unsigned integer value represented by bits 52:48, "Z" is an unsigned integer represented by bits 54:53. "Time_Unit" is specified by the "Time Units" field of MSR_RAPL_POWER_UNIT
		63:55		Reserved.
618H	1560	MSR_DRAM_POWER_LIMIT	Package	<b>DRAM RAPL Power Limit Control (R/W)</b> See Section 14.9.5, "DRAM RAPL Domain."
619H	1561	MSR_DRAM_ENERGY_STATUS	Package	<b>DRAM Energy Status (R/O)</b> See Section 14.9.5, "DRAM RAPL Domain."
61BH	1563	MSR_DRAM_PERF_STATUS	Package	<b>DRAM Performance Throttling Status (R/O)</b> See Section 14.9.5, "DRAM RAPL Domain."
61CH	1564	MSR_DRAM_POWER_INFO	Package	<b>DRAM RAPL Parameters (R/W)</b> See Section 14.9.5, "DRAM RAPL Domain."

**Table 35-12. MSRs in Next Generation Intel Atom Processors Based on the Goldmont Microarchitecture (Contd.)**

Address		Register Name	Scope	Bit Description
Hex	Dec			
632H	1586	MSR_PKG_C10_RESIDENCY	Package	Note: C-state values are processor specific C-state code names,
		63:0		<b>Package C10 Residency Counter. (R/O)</b> Value since last reset that the entire SOC is in an S0i3 state. Count at the same frequency as the TSC.
639H	1593	MSR_PPO_ENERGY_STATUS	Package	<b>PPO Energy Status (R/O)</b> See Section 14.9.4, "PPO/PP1 RAPL Domains."
641H	1601	MSR_PP1_ENERGY_STATUS	Package	<b>PP1 Energy Status (R/O)</b> See Section 14.9.4, "PPO/PP1 RAPL Domains."
64CH	1612	MSR_TURBO_ACTIVATION_RATIO	Package	<b>ConfigTDP Control (R/W)</b>
		7:0		<b>MAX_NON_TURBO_RATIO (Rw/L)</b> System BIOS can program this field.
		30:8		Reserved.
		31		<b>TURBO_ACTIVATION_RATIO_Lock (Rw/L)</b> When this bit is set, the content of this register is locked until a reset.
		63:32		Reserved.
64FH	1615	MSR_CORE_PERF_LIMIT_REASONS	Package	<b>Indicator of Frequency Clipping in Processor Cores (R/W) (frequency refers to processor core frequency)</b>
		0		<b>PROCHOT Status (R0)</b> When set, processor core frequency is reduced below the operating system request due to assertion of external PROCHOT.
		1		<b>Thermal Status (R0)</b> When set, frequency is reduced below the operating system request due to a thermal event.
		2		<b>Package-Level Power Limiting PL1 Status (R0)</b> When set, frequency is reduced below the operating system request due to package-level power limiting PL1.
		3		<b>Package-Level PL2 Power Limiting Status (R0)</b> When set, frequency is reduced below the operating system request due to package-level power limiting PL2.
		8:4		Reserved.
		9		<b>Core Power Limiting Status (R0)</b> When set, frequency is reduced below the operating system request due to domain-level power limiting.
		10		<b>VR Therm Alert Status (R0)</b> When set, frequency is reduced below the operating system request due to a thermal alert from the Voltage Regulator.
		11		<b>Max Turbo Limit Status (R0)</b> When set, frequency is reduced below the operating system request due to multi-core turbo limits.

**Table 35-12. MSRs in Next Generation Intel Atom Processors Based on the Goldmont Microarchitecture (Contd.)**

Address		Register Name	Scope	Bit Description
Hex	Dec			
		12		<b>Electrical Design Point Status (R0)</b> When set, frequency is reduced below the operating system request due to electrical design point constraints (e.g. maximum electrical current consumption).
		13		<b>Turbo Transition Attenuation Status (R0)</b> When set, frequency is reduced below the operating system request due to Turbo transition attenuation. This prevents performance degradation due to frequent operating ratio changes.
		14		<b>Maximum Efficiency Frequency Status (R0)</b> When set, frequency is reduced below the maximum efficiency frequency.
		15		<b>Reserved</b>
		16		<b>PROCHOT Log</b> When set, indicates that the PROCHOT Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		17		<b>Thermal Log</b> When set, indicates that the Thermal Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		18		<b>Package-Level PL1 Power Limiting Log</b> When set, indicates that the Package Level PL1 Power Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		19		<b>Package-Level PL2 Power Limiting Log</b> When set, indicates that the Package Level PL2 Power Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		24:20		Reserved.
		25		<b>Core Power Limiting Log</b> When set, indicates that the Core Power Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		26		<b>VR Therm Alert Log</b> When set, indicates that the VR Therm Alert Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		27		<b>Max Turbo Limit Log</b> When set, indicates that the Max Turbo Limit Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.

**Table 35-12. MSRs in Next Generation Intel Atom Processors Based on the Goldmont Microarchitecture (Contd.)**

Address		Register Name	Scope	Bit Description
Hex	Dec			
		28		<b>Electrical Design Point Log</b> When set, indicates that the EDP Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		29		<b>Turbo Transition Attenuation Log</b> When set, indicates that the Turbo Transition Attenuation Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		30		<b>Maximum Efficiency Frequency Log</b> When set, indicates that the Maximum Efficiency Frequency Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		63:31		Reserved.
680H	1664	MSR_LASTBRANCH_0_FROM_IP	Core	<b>Last Branch Record 0 From IP (R/W)</b> One of 32 pairs of last branch record registers on the last branch record stack. The From_IP part of the stack contains pointers to the <b>source instruction</b> . See also: <ul style="list-style-type: none"> <li>▪ Last Branch Record Stack TOS at 1C9H</li> <li>▪ Section 17.6 and record format in Section 17.4.8.1</li> </ul>
		0:47		<b>From Linear Address (R/W)</b>
		62:48		Signed extension of bits 47:0.
		63		Mispred
681H	1665	MSR_LASTBRANCH_1_FROM_IP	Core	<b>Last Branch Record 1 From IP (R/W)</b> See description of MSR_LASTBRANCH_0_FROM_IP.
682H	1666	MSR_LASTBRANCH_2_FROM_IP	Core	<b>Last Branch Record 2 From IP (R/W)</b> See description of MSR_LASTBRANCH_0_FROM_IP.
683H	1667	MSR_LASTBRANCH_3_FROM_IP	Core	<b>Last Branch Record 3 From IP (R/W)</b> See description of MSR_LASTBRANCH_0_FROM_IP.
684H	1668	MSR_LASTBRANCH_4_FROM_IP	Core	<b>Last Branch Record 4 From IP (R/W)</b> See description of MSR_LASTBRANCH_0_FROM_IP.
685H	1669	MSR_LASTBRANCH_5_FROM_IP	Core	<b>Last Branch Record 5 From IP (R/W)</b> See description of MSR_LASTBRANCH_0_FROM_IP.
686H	1670	MSR_LASTBRANCH_6_FROM_IP	Core	<b>Last Branch Record 6 From IP (R/W)</b> See description of MSR_LASTBRANCH_0_FROM_IP.
687H	1671	MSR_LASTBRANCH_7_FROM_IP	Core	<b>Last Branch Record 7 From IP (R/W)</b> See description of MSR_LASTBRANCH_0_FROM_IP.
688H	1672	MSR_LASTBRANCH_8_FROM_IP	Core	<b>Last Branch Record 8 From IP (R/W)</b> See description of MSR_LASTBRANCH_0_FROM_IP.
689H	1673	MSR_LASTBRANCH_9_FROM_IP	Core	<b>Last Branch Record 9 From IP (R/W)</b> See description of MSR_LASTBRANCH_0_FROM_IP.
68AH	1674	MSR_LASTBRANCH_10_FROM_IP	Core	<b>Last Branch Record 10 From IP (R/W)</b> See description of MSR_LASTBRANCH_0_FROM_IP.



**Table 35-12. MSRs in Next Generation Intel Atom Processors Based on the Goldmont Microarchitecture (Contd.)**

Address		Register Name	Scope	Bit Description
Hex	Dec			
68BH	1675	MSR_LASTBRANCH_11_FROM_IP	Core	<b>Last Branch Record 11 From IP (R/w)</b> See description of MSR_LASTBRANCH_0_FROM_IP.
68CH	1676	MSR_LASTBRANCH_12_FROM_IP	Core	<b>Last Branch Record 12 From IP (R/w)</b> See description of MSR_LASTBRANCH_0_FROM_IP.
68DH	1677	MSR_LASTBRANCH_13_FROM_IP	Core	<b>Last Branch Record 13 From IP (R/w)</b> See description of MSR_LASTBRANCH_0_FROM_IP.
68EH	1678	MSR_LASTBRANCH_14_FROM_IP	Core	<b>Last Branch Record 14 From IP (R/w)</b> See description of MSR_LASTBRANCH_0_FROM_IP.
68FH	1679	MSR_LASTBRANCH_15_FROM_IP	Core	<b>Last Branch Record 15 From IP (R/w)</b> See description of MSR_LASTBRANCH_0_FROM_IP.
690H	1680	MSR_LASTBRANCH_16_FROM_IP	Core	<b>Last Branch Record 16 From IP (R/w)</b> See description of MSR_LASTBRANCH_0_FROM_IP.
691H	1681	MSR_LASTBRANCH_17_FROM_IP	Core	<b>Last Branch Record 17 From IP (R/w)</b> See description of MSR_LASTBRANCH_0_FROM_IP.
692H	1682	MSR_LASTBRANCH_18_FROM_IP	Core	<b>Last Branch Record 18 From IP (R/w)</b> See description of MSR_LASTBRANCH_0_FROM_IP.
693H	1683	MSR_LASTBRANCH_19_FROM_IP	Core	<b>Last Branch Record 19 From IP (R/w)</b> See description of MSR_LASTBRANCH_0_FROM_IP.
694H	1684	MSR_LASTBRANCH_20_FROM_IP	Core	<b>Last Branch Record 20 From IP (R/w)</b> See description of MSR_LASTBRANCH_0_FROM_IP.
695H	1685	MSR_LASTBRANCH_21_FROM_IP	Core	<b>Last Branch Record 21 From IP (R/w)</b> See description of MSR_LASTBRANCH_0_FROM_IP.
696H	1686	MSR_LASTBRANCH_22_FROM_IP	Core	<b>Last Branch Record 22 From IP (R/w)</b> See description of MSR_LASTBRANCH_0_FROM_IP.
697H	1687	MSR_LASTBRANCH_23_FROM_IP	Core	<b>Last Branch Record 23 From IP (R/w)</b> See description of MSR_LASTBRANCH_0_FROM_IP.
698H	1688	MSR_LASTBRANCH_24_FROM_IP	Core	<b>Last Branch Record 24 From IP (R/w)</b> See description of MSR_LASTBRANCH_0_FROM_IP.
699H	1689	MSR_LASTBRANCH_25_FROM_IP	Core	<b>Last Branch Record 25 From IP (R/w)</b> See description of MSR_LASTBRANCH_0_FROM_IP.
69AH	1690	MSR_LASTBRANCH_26_FROM_IP	Core	<b>Last Branch Record 26 From IP (R/w)</b> See description of MSR_LASTBRANCH_0_FROM_IP.
69BH	1691	MSR_LASTBRANCH_27_FROM_IP	Core	<b>Last Branch Record 27 From IP (R/w)</b> See description of MSR_LASTBRANCH_0_FROM_IP.
69CH	1692	MSR_LASTBRANCH_28_FROM_IP	Core	<b>Last Branch Record 28 From IP (R/w)</b> See description of MSR_LASTBRANCH_0_FROM_IP.
69DH	1693	MSR_LASTBRANCH_29_FROM_IP	Core	<b>Last Branch Record 29 From IP (R/w)</b> See description of MSR_LASTBRANCH_0_FROM_IP.
69EH	1694	MSR_LASTBRANCH_30_FROM_IP	Core	<b>Last Branch Record 30 From IP (R/w)</b> See description of MSR_LASTBRANCH_0_FROM_IP.

**Table 35-12. MSRs in Next Generation Intel Atom Processors Based on the Goldmont Microarchitecture (Contd.)**

Address		Register Name	Scope	Bit Description	
Hex	Dec				
69FH	1695	MSR_LASTBRANCH_31_FROM_IP	Core	<b>Last Branch Record 31 From IP (R/W)</b> See description of MSR_LASTBRANCH_0_FROM_IP.	
6C0H	1728	MSR_LASTBRANCH_0_TO_IP	Core	<b>Last Branch Record 0 To IP (R/W)</b> One of 32 pairs of last branch record registers on the last branch record stack. The To_IP part of the stack contains pointers to the <b>Destination instruction</b> and elapsed cycles from last LBR update. See also: ▪ Section 17.6	
				0:47	<b>Target Linear Address (R/W)</b>
				63:48	Elapsed cycles from last update to the LBR.
6C1H	1729	MSR_LASTBRANCH_1_TO_IP	Core	<b>Last Branch Record 1 To IP (R/W)</b> See description of MSR_LASTBRANCH_0_TO_IP.	
6C2H	1730	MSR_LASTBRANCH_2_TO_IP	Core	<b>Last Branch Record 2 To IP (R/W)</b> See description of MSR_LASTBRANCH_0_TO_IP.	
6C3H	1731	MSR_LASTBRANCH_3_TO_IP	Core	<b>Last Branch Record 3 To IP (R/W)</b> See description of MSR_LASTBRANCH_0_TO_IP.	
6C4H	1732	MSR_LASTBRANCH_4_TO_IP	Core	<b>Last Branch Record 4 To IP (R/W)</b> See description of MSR_LASTBRANCH_0_TO_IP.	
6C5H	1733	MSR_LASTBRANCH_5_TO_IP	Core	<b>Last Branch Record 5 To IP (R/W)</b> See description of MSR_LASTBRANCH_0_TO_IP.	
6C6H	1734	MSR_LASTBRANCH_6_TO_IP	Core	<b>Last Branch Record 6 To IP (R/W)</b> See description of MSR_LASTBRANCH_0_TO_IP.	
6C7H	1735	MSR_LASTBRANCH_7_TO_IP	Core	<b>Last Branch Record 7 To IP (R/W)</b> See description of MSR_LASTBRANCH_0_TO_IP.	
6C8H	1736	MSR_LASTBRANCH_8_TO_IP	Core	<b>Last Branch Record 8 To IP (R/W)</b> See description of MSR_LASTBRANCH_0_TO_IP.	
6C9H	1737	MSR_LASTBRANCH_9_TO_IP	Core	<b>Last Branch Record 9 To IP (R/W)</b> See description of MSR_LASTBRANCH_0_TO_IP.	
6CAH	1738	MSR_LASTBRANCH_10_TO_IP	Core	<b>Last Branch Record 10 To IP (R/W)</b> See description of MSR_LASTBRANCH_0_TO_IP.	
6CBH	1739	MSR_LASTBRANCH_11_TO_IP	Core	<b>Last Branch Record 11 To IP (R/W)</b> See description of MSR_LASTBRANCH_0_TO_IP.	
6CCH	1740	MSR_LASTBRANCH_12_TO_IP	Core	<b>Last Branch Record 12 To IP (R/W)</b> See description of MSR_LASTBRANCH_0_TO_IP.	
6CDH	1741	MSR_LASTBRANCH_13_TO_IP	Core	<b>Last Branch Record 13 To IP (R/W)</b> See description of MSR_LASTBRANCH_0_TO_IP.	
6CEH	1742	MSR_LASTBRANCH_14_TO_IP	Core	<b>Last Branch Record 14 To IP (R/W)</b> See description of MSR_LASTBRANCH_0_TO_IP.	
6CFH	1743	MSR_LASTBRANCH_15_TO_IP	Core	<b>Last Branch Record 15 To IP (R/W)</b> See description of MSR_LASTBRANCH_0_TO_IP.	

**Table 35-12. MSRs in Next Generation Intel Atom Processors Based on the Goldmont Microarchitecture (Contd.)**

Address		Register Name	Scope	Bit Description
Hex	Dec			
6D0H	1744	MSR_LASTBRANCH_16_TO_IP	Core	<b>Last Branch Record 16 To IP (R/W)</b> See description of MSR_LASTBRANCH_0_TO_IP.
6D1H	1745	MSR_LASTBRANCH_17_TO_IP	Core	<b>Last Branch Record 17 To IP (R/W)</b> See description of MSR_LASTBRANCH_0_TO_IP.
6D2H	1746	MSR_LASTBRANCH_18_TO_IP	Core	<b>Last Branch Record 18 To IP (R/W)</b> See description of MSR_LASTBRANCH_0_TO_IP.
6D3H	1747	MSR_LASTBRANCH_19_TO_IP	Core	<b>Last Branch Record 19 To IP (R/W)</b> See description of MSR_LASTBRANCH_0_TO_IP.
6D4H	1748	MSR_LASTBRANCH_20_TO_IP	Core	<b>Last Branch Record 20 To IP (R/W)</b> See description of MSR_LASTBRANCH_0_TO_IP.
6D5H	1749	MSR_LASTBRANCH_21_TO_IP	Core	<b>Last Branch Record 21 To IP (R/W)</b> See description of MSR_LASTBRANCH_0_TO_IP.
6D6H	1750	MSR_LASTBRANCH_22_TO_IP	Core	<b>Last Branch Record 22 To IP (R/W)</b> See description of MSR_LASTBRANCH_0_TO_IP.
6D7H	1751	MSR_LASTBRANCH_23_TO_IP	Core	<b>Last Branch Record 23 To IP (R/W)</b> See description of MSR_LASTBRANCH_0_TO_IP.
6D8H	1752	MSR_LASTBRANCH_24_TO_IP	Core	<b>Last Branch Record 24 To IP (R/W)</b> See description of MSR_LASTBRANCH_0_TO_IP.
6D9H	1753	MSR_LASTBRANCH_25_TO_IP	Core	<b>Last Branch Record 25 To IP (R/W)</b> See description of MSR_LASTBRANCH_0_TO_IP.
6DAH	1754	MSR_LASTBRANCH_26_TO_IP	Core	<b>Last Branch Record 26 To IP (R/W)</b> See description of MSR_LASTBRANCH_0_TO_IP.
6DBH	1755	MSR_LASTBRANCH_27_TO_IP	Core	<b>Last Branch Record 27 To IP (R/W)</b> See description of MSR_LASTBRANCH_0_TO_IP.
6DCH	1756	MSR_LASTBRANCH_28_TO_IP	Core	<b>Last Branch Record 28 To IP (R/W)</b> See description of MSR_LASTBRANCH_0_TO_IP.
6DDH	1757	MSR_LASTBRANCH_29_TO_IP	Core	<b>Last Branch Record 29 To IP (R/W)</b> See description of MSR_LASTBRANCH_0_TO_IP.
6DEH	1758	MSR_LASTBRANCH_30_TO_IP	Core	<b>Last Branch Record 30 To IP (R/W)</b> See description of MSR_LASTBRANCH_0_TO_IP.
6DFH	1759	MSR_LASTBRANCH_31_TO_IP	Core	<b>Last Branch Record 31 To IP (R/W)</b> See description of MSR_LASTBRANCH_0_TO_IP.
802H	2050	IA32_X2APIC_APICID	Core	x2APIC ID register (R/O) See x2APIC Specification.
803H	2051	IA32_X2APIC_VERSION	Core	x2APIC Version register (R/O)
808H	2056	IA32_X2APIC_TPR	Core	x2APIC Task Priority register (R/W)
80AH	2058	IA32_X2APIC_PPR	Core	x2APIC Processor Priority register (R/O)
80BH	2059	IA32_X2APIC_EOI	Core	x2APIC EOI register (W/O)
80DH	2061	IA32_X2APIC_LDR	Core	x2APIC Logical Destination register (R/O)
80FH	2063	IA32_X2APIC_SIVR	Core	x2APIC Spurious Interrupt Vector register (R/W)

**Table 35-12. MSRs in Next Generation Intel Atom Processors Based on the Goldmont Microarchitecture (Contd.)**

Address		Register Name	Scope	Bit Description
Hex	Dec			
810H	2064	IA32_X2APIC_ISR0	Core	x2APIC In-Service register bits [31:0] (R/O)
811H	2065	IA32_X2APIC_ISR1	Core	x2APIC In-Service register bits [63:32] (R/O)
812H	2066	IA32_X2APIC_ISR2	Core	x2APIC In-Service register bits [95:64] (R/O)
813H	2067	IA32_X2APIC_ISR3	Core	x2APIC In-Service register bits [127:96] (R/O)
814H	2068	IA32_X2APIC_ISR4	Core	x2APIC In-Service register bits [159:128] (R/O)
815H	2069	IA32_X2APIC_ISR5	Core	x2APIC In-Service register bits [191:160] (R/O)
816H	2070	IA32_X2APIC_ISR6	Core	x2APIC In-Service register bits [223:192] (R/O)
817H	2071	IA32_X2APIC_ISR7	Core	x2APIC In-Service register bits [255:224] (R/O)
818H	2072	IA32_X2APIC_TMR0	Core	x2APIC Trigger Mode register bits [31:0] (R/O)
819H	2073	IA32_X2APIC_TMR1	Core	x2APIC Trigger Mode register bits [63:32] (R/O)
81AH	2074	IA32_X2APIC_TMR2	Core	x2APIC Trigger Mode register bits [95:64] (R/O)
81BH	2075	IA32_X2APIC_TMR3	Core	x2APIC Trigger Mode register bits [127:96] (R/O)
81CH	2076	IA32_X2APIC_TMR4	Core	x2APIC Trigger Mode register bits [159:128] (R/O)
81DH	2077	IA32_X2APIC_TMR5	Core	x2APIC Trigger Mode register bits [191:160] (R/O)
81EH	2078	IA32_X2APIC_TMR6	Core	x2APIC Trigger Mode register bits [223:192] (R/O)
81FH	2079	IA32_X2APIC_TMR7	Core	x2APIC Trigger Mode register bits [255:224] (R/O)
820H	2080	IA32_X2APIC_IRR0	Core	x2APIC Interrupt Request register bits [31:0] (R/O)
821H	2081	IA32_X2APIC_IRR1	Core	x2APIC Interrupt Request register bits [63:32] (R/O)
822H	2082	IA32_X2APIC_IRR2	Core	x2APIC Interrupt Request register bits [95:64] (R/O)
823H	2083	IA32_X2APIC_IRR3	Core	x2APIC Interrupt Request register bits [127:96] (R/O)
824H	2084	IA32_X2APIC_IRR4	Core	x2APIC Interrupt Request register bits [159:128] (R/O)
825H	2085	IA32_X2APIC_IRR5	Core	x2APIC Interrupt Request register bits [191:160] (R/O)
826H	2086	IA32_X2APIC_IRR6	Core	x2APIC Interrupt Request register bits [223:192] (R/O)
827H	2087	IA32_X2APIC_IRR7	Core	x2APIC Interrupt Request register bits [255:224] (R/O)
828H	2088	IA32_X2APIC_ESR	Core	x2APIC Error Status register (R/W)
82FH	2095	IA32_X2APIC_LVT_CMCI	Core	x2APIC LVT Corrected Machine Check Interrupt register (R/W)
830H	2096	IA32_X2APIC_ICR	Core	x2APIC Interrupt Command register (R/W)
832H	2098	IA32_X2APIC_LVT_TIMER	Core	x2APIC LVT Timer Interrupt register (R/W)
833H	2099	IA32_X2APIC_LVT_THERMAL	Core	x2APIC LVT Thermal Sensor Interrupt register (R/W)
834H	2100	IA32_X2APIC_LVT_PMI	Core	x2APIC LVT Performance Monitor register (R/W)
835H	2101	IA32_X2APIC_LVT_LINT0	Core	x2APIC LVT LINT0 register (R/W)
836H	2102	IA32_X2APIC_LVT_LINT1	Core	x2APIC LVT LINT1 register (R/W)
837H	2103	IA32_X2APIC_LVT_ERROR	Core	x2APIC LVT Error register (R/W)
838H	2104	IA32_X2APIC_INIT_COUNT	Core	x2APIC Initial Count register (R/W)
839H	2105	IA32_X2APIC_CUR_COUNT	Core	x2APIC Current Count register (R/O)
83EH	2110	IA32_X2APIC_DIV_CONF	Core	x2APIC Divide Configuration register (R/W)
83FH	2111	IA32_X2APIC_SELF_IPI	Core	x2APIC Self IPI register (W/O)

**Table 35-12. MSRs in Next Generation Intel Atom Processors Based on the Goldmont Microarchitecture (Contd.)**

Address		Register Name	Scope	Bit Description
Hex	Dec			
C8FH	3215	IA32_PQR_ASSOC	Core	<b>Resource Association Register (R/W)</b>
		31:0		<b>Reserved</b>
		33:32		<b>COS (R/W).</b>
		63:34		<b>Reserved</b>
D10H	3344	IA32_L2_QOS_MASK_0	Module	<b>L2 Class Of Service Mask - COS 0 (R/W)</b> if CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=0
		0:7		<b>CBM: Bit vector of available L2 ways for COS 0 enforcement</b>
		63:8		<b>Reserved</b>
D11H	3345	IA32_L2_QOS_MASK_1	Module	<b>L2 Class Of Service Mask - COS 1 (R/W)</b> if CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=1
		0:7		<b>CBM: Bit vector of available L2 ways for COS 0 enforcement</b>
		63:8		<b>Reserved</b>
D12H	3346	IA32_L2_QOS_MASK_2	Module	<b>L2 Class Of Service Mask - COS 2 (R/W)</b> if CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=2
		0:7		<b>CBM: Bit vector of available L2 ways for COS 0 enforcement</b>
		63:8		<b>Reserved</b>
D13H	3347	IA32_L2_QOS_MASK_3	Package	<b>L2 Class Of Service Mask - COS 3 (R/W).</b> if CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=3
		0:19		<b>CBM: Bit vector of available L2 ways for COS 3 enforcement</b>
		63:20		<b>Reserved</b>
D90H	3472	IA32_BNDCFGS	Core	See Table 35-2.
DA0H	3488	IA32_XSS	Core	See Table 35-2.
See Table 35-6, and Table 35-12 for MSR definitions applicable to processors with CPUID signature 06_5CH.				

### 35.6 MSRS IN THE INTEL® MICROARCHITECTURE CODE NAME NEHALEM

Table 35-13 lists model-specific registers (MSRs) that are common for Intel® microarchitecture code name Nehalem. These include Intel Core i7 and i5 processor family. These processors have a CPUID signature with DisplayFamily\_DisplayModel of 06\_1AH, 06\_1EH, 06\_1FH, 06\_2EH, see Table 35-1. Additional MSRs specific to 06\_1AH, 06\_1EH, 06\_1FH are listed in Table 35-14. Some MSRs listed in these tables are used by BIOS. More information about these MSR can be found at <http://biosbits.org>.

The column “Scope” represents the package/core/thread scope of individual bit field of an MSR. “Thread” means this bit field must be programmed on each logical processor independently. “Core” means the bit field must be programmed on each processor core independently, logical processors in the same core will be affected by change of this bit on the other logical processor in the same core. “Package” means the bit field must be programmed once for each physical package. Change of a bit filed with a package scope will affect all logical processors in that physical package.

Table 35-13. MSRs in Processors Based on Intel® Microarchitecture Code Name Nehalem

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
0H	0	IA32_P5_MC_ADDR	Thread	See Section 35.22, "MSRs in Pentium Processors."
1H	1	IA32_P5_MC_TYPE	Thread	See Section 35.22, "MSRs in Pentium Processors."
6H	6	IA32_MONITOR_FILTER_SIZE	Thread	See Section 8.10.5, "Monitor/Mwait Address Range Determination," and Table 35-2.
10H	16	IA32_TIME_STAMP_COUNTER	Thread	See Section 17.15, "Time-Stamp Counter," and see Table 35-2.
17H	23	IA32_PLATFORM_ID	Package	<b>Platform ID (R)</b> See Table 35-2.
17H	23	MSR_PLATFORM_ID	Package	<b>Model Specific Platform ID (R)</b>
		49:0		Reserved.
		52:50		See Table 35-2.
		63:53		Reserved.
1BH	27	IA32_APIC_BASE	Thread	See Section 10.4.4, "Local APIC Status and Location," and Table 35-2.
34H	52	MSR_SMI_COUNT	Thread	<b>SMI Counter (R/O)</b>
		31:0		<b>SMI Count (R/O)</b> Running count of SMI events since last RESET.
		63:32		Reserved.
3AH	58	IA32_FEATURE_CONTROL	Thread	<b>Control Features in Intel 64 Processor (R/W)</b> See Table 35-2.
79H	121	IA32_BIOS_UPDT_TRIG	Core	<b>BIOS Update Trigger Register (W)</b> See Table 35-2.
8BH	139	IA32_BIOS_SIGN_ID	Thread	<b>BIOS Update Signature ID (RO)</b> See Table 35-2.
C1H	193	IA32_PMC0	Thread	<b>Performance Counter Register</b> See Table 35-2.
C2H	194	IA32_PMC1	Thread	<b>Performance Counter Register</b> See Table 35-2.
C3H	195	IA32_PMC2	Thread	<b>Performance Counter Register</b> See Table 35-2.
C4H	196	IA32_PMC3	Thread	<b>Performance Counter Register</b> See Table 35-2.
CEH	206	MSR_PLATFORM_INFO	Package	see <a href="http://biosbits.org">http://biosbits.org</a> .
		7:0		Reserved.

**Table 35-13. MSRs in Processors Based on Intel® Microarchitecture Code Name Nehalem (Contd.)**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		15:8	Package	<b>Maximum Non-Turbo Ratio (R/O)</b> The is the ratio of the frequency that invariant TSC runs at. The invariant TSC frequency can be computed by multiplying this ratio by 133.33 MHz.
		27:16		Reserved.
		28	Package	<b>Programmable Ratio Limit for Turbo Mode (R/O)</b> When set to 1, indicates that Programmable Ratio Limits for Turbo mode is enabled, and when set to 0, indicates Programmable Ratio Limits for Turbo mode is disabled.
		29	Package	<b>Programmable TDC-TDP Limit for Turbo Mode (R/O)</b> When set to 1, indicates that TDC/TDP Limits for Turbo mode are programmable, and when set to 0, indicates TDC and TDP Limits for Turbo mode are not programmable.
		39:30		Reserved.
		47:40	Package	<b>Maximum Efficiency Ratio (R/O)</b> The is the minimum ratio (maximum efficiency) that the processor can operates, in units of 133.33MHz.
		63:48		Reserved.
E2H	226	MSR_PKG_CST_CONFIG_CONTROL	Core	<b>C-State Configuration Control (R/W)</b> Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States. See <a href="http://biosbits.org">http://biosbits.org</a> .
		2:0		<b>Package C-State Limit (R/W)</b> Specifies the lowest processor-specific C-state code name (consuming the least power), for the package. The default is set as factory-configured package C-state limit. The following C-state code name encodings are supported: 000b: C0 (no package C-sate support) 001b: C1 (Behavior is the same as 000b) 010b: C3 011b: C6 100b: C7 101b and 110b: Reserved 111: No package C-state limit. Note: This field cannot be used to limit package C-state to C3.
		9:3		Reserved.
		10		<b>I/O MWAIT Redirection Enable (R/W)</b> When set, will map IO_read instructions sent to IO register specified by MSR_PMG_IO_CAPTURE_BASE to MWAIT instructions.
		14:11		Reserved.
		15		<b>CFG Lock (R/WO)</b> When set, lock bits 15:0 of this register until next reset.

Table 35-13. MSRs in Processors Based on Intel® Microarchitecture Code Name Nehalem (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		23:16		Reserved.
		24		<b>Interrupt filtering enable (R/W)</b> When set, processor cores in a deep C-State will wake only when the event message is destined for that core. When 0, all processor cores in a deep C-State will wake for an event message.
		25		<b>C3 state auto demotion enable (R/W)</b> When set, the processor will conditionally demote C6/C7 requests to C3 based on uncore auto-demote information.
		26		<b>C1 state auto demotion enable (R/W)</b> When set, the processor will conditionally demote C3/C6/C7 requests to C1 based on uncore auto-demote information.
		27		<b>Enable C3 Undemotion (R/W)</b>
		28		<b>Enable C1 Undemotion (R/W)</b>
		29		<b>Package C State Demotion Enable (R/W)</b>
		30		<b>Package C State UnDemotion Enable (R/W)</b>
		63:31		Reserved.
E4H	228	MSR_PMG_IO_CAPTURE_BASE	Core	<b>Power Management IO Redirection in C-state (R/W)</b> See <a href="http://biosbits.org">http://biosbits.org</a> .
		15:0		<b>LVL_2 Base Address (R/W)</b> Specifies the base address visible to software for IO redirection. If IO MWait Redirection is enabled, reads to this address will be consumed by the power management logic and decoded to MWait instructions. When IO port address redirection is enabled, this is the IO port address reported to the OS/software.
		18:16		<b>C-state Range (R/W)</b> Specifies the encoding value of the maximum C-State code name to be included when IO read to MWait redirection is enabled by MSR_PKG_CST_CONFIG_CONTROL[bit10]: 000b - C3 is the max C-State to include 001b - C6 is the max C-State to include 010b - C7 is the max C-State to include
		63:19		Reserved.
E7H	231	IA32_MPERF	Thread	<b>Maximum Performance Frequency Clock Count (RW)</b> See Table 35-2.
E8H	232	IA32_APERF	Thread	<b>Actual Performance Frequency Clock Count (RW)</b> See Table 35-2.
FEH	254	IA32_MTRRCAP	Thread	See Table 35-2.
174H	372	IA32_SYSENTER_CS	Thread	See Table 35-2.
175H	373	IA32_SYSENTER_ESP	Thread	See Table 35-2.
176H	374	IA32_SYSENTER_EIP	Thread	See Table 35-2.



**Table 35-13. MSRs in Processors Based on Intel® Microarchitecture Code Name Nehalem (Contd.)**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
179H	377	IA32_MCG_CAP	Thread	See Table 35-2.
17AH	378	IA32_MCG_STATUS	Thread	
		0		<b>RIPV</b> When set, bit indicates that the instruction addressed by the instruction pointer pushed on the stack (when the machine check was generated) can be used to restart the program. If cleared, the program cannot be reliably restarted.
		1		<b>EIPV</b> When set, bit indicates that the instruction addressed by the instruction pointer pushed on the stack (when the machine check was generated) is directly associated with the error.
		2		<b>MCIP</b> When set, bit indicates that a machine check has been generated. If a second machine check is detected while this bit is still set, the processor enters a shutdown state. Software should write this bit to 0 after processing a machine check exception.
		63:3		Reserved.
186H	390	IA32_PERFEVTSELO	Thread	See Table 35-2.
		7:0		<b>Event Select</b>
		15:8		<b>UMask</b>
		16		<b>USR</b>
		17		<b>OS</b>
		18		<b>Edge</b>
		19		<b>PC</b>
		20		<b>INT</b>
		21		<b>AnyThread</b>
		22		<b>EN</b>
		23		<b>INV</b>
		31:24		<b>CMASK</b>
		63:32		Reserved.
187H	391	IA32_PERFEVTSEL1	Thread	See Table 35-2.
188H	392	IA32_PERFEVTSEL2	Thread	See Table 35-2.
189H	393	IA32_PERFEVTSEL3	Thread	See Table 35-2.
198H	408	IA32_PERF_STATUS	Core	See Table 35-2.
		15:0		Current Performance State Value.
		63:16		Reserved.
199H	409	IA32_PERF_CTL	Thread	See Table 35-2.

Table 35-13. MSRs in Processors Based on Intel® Microarchitecture Code Name Nehalem (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
19AH	410	IA32_CLOCK_MODULATION	Thread	<b>Clock Modulation (R/W)</b> See Table 35-2. IA32_CLOCK_MODULATION MSR was originally named IA32_THERM_CONTROL MSR.
		0		Reserved.
		3:1		<b>On demand Clock Modulation Duty Cycle (R/W)</b>
		4		<b>On demand Clock Modulation Enable (R/W)</b>
		63:5		Reserved.
19BH	411	IA32_THERM_INTERRUPT	Core	<b>Thermal Interrupt Control (R/W)</b> See Table 35-2.
19CH	412	IA32_THERM_STATUS	Core	<b>Thermal Monitor Status (R/W)</b> See Table 35-2.
1A0H	416	IA32_MISC_ENABLE		<b>Enable Misc. Processor Features (R/W)</b> Allows a variety of processor functions to be enabled and disabled.
		0	Thread	<b>Fast-Strings Enable</b> See Table 35-2.
		2:1		Reserved.
		3	Thread	<b>Automatic Thermal Control Circuit Enable (R/W)</b> See Table 35-2. Default value is 1.
		6:4		Reserved.
		7	Thread	<b>Performance Monitoring Available (R)</b> See Table 35-2.
		10:8		Reserved.
		11	Thread	<b>Branch Trace Storage Unavailable (RO)</b> See Table 35-2.
		12	Thread	<b>Processor Event Based Sampling Unavailable (RO)</b> See Table 35-2.
		15:13		Reserved.
		16	Package	<b>Enhanced Intel SpeedStep Technology Enable (R/W)</b> See Table 35-2.
		18	Thread	ENABLE MONITOR FSM. (R/W) See Table 35-2.
		21:19		Reserved.
		22	Thread	<b>Limit CPUID Maxval (R/W)</b> See Table 35-2.
23	Thread	<b>xTPR Message Disable (R/W)</b> See Table 35-2.		
33:24		Reserved.		

**Table 35-13. MSRs in Processors Based on Intel® Microarchitecture Code Name Nehalem (Contd.)**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		34	Thread	<b>XD Bit Disable (R/W)</b> See Table 35-2.
		37:35		Reserved.
		38	Package	<b>Turbo Mode Disable (R/W)</b> When set to 1 on processors that support Intel Turbo Boost Technology, the turbo mode feature is disabled and the IDA_Enable feature flag will be clear (CPUID.06H: EAX[1]=0). When set to a 0 on processors that support IDA, CPUID.06H: EAX[1] reports the processor's support of turbo mode is enabled. <b>Note:</b> the power-on default value is used by BIOS to detect hardware support of turbo mode. If power-on default value is 1, turbo mode is available in the processor. If power-on default value is 0, turbo mode is not available.
		63:39		Reserved.
1A2H	418	MSR_TEMPERATURE_TARGET	Thread	
		15:0		Reserved.
		23:16		<b>Temperature Target (R)</b> The minimum temperature at which PROCHOT# will be asserted. The value is degree C.
		63:24		Reserved.
1A4H	420	MSR_MISC_FEATURE_CONTROL		<b>Miscellaneous Feature Control (R/W)</b>
		0	Core	<b>L2 Hardware Prefetcher Disable (R/W)</b> If 1, disables the L2 hardware prefetcher, which fetches additional lines of code or data into the L2 cache.
		1	Core	<b>L2 Adjacent Cache Line Prefetcher Disable (R/W)</b> If 1, disables the adjacent cache line prefetcher, which fetches the cache line that comprises a cache line pair (128 bytes).
		2	Core	<b>DCU Hardware Prefetcher Disable (R/W)</b> If 1, disables the L1 data cache prefetcher, which fetches the next cache line into L1 data cache.
		3	Core	<b>DCU IP Prefetcher Disable (R/W)</b> If 1, disables the L1 data cache IP prefetcher, which uses sequential load history (based on instruction Pointer of previous loads) to determine whether to prefetch additional lines.
		63:4		Reserved.
1A6H	422	MSR_OFFCORE_RSP_0	Thread	<b>Offcore Response Event Select Register (R/W)</b>
1AAH	426	MSR_MISC_PWR_MGMT		See <a href="http://biosbits.org">http://biosbits.org</a> .

Table 35-13. MSRs in Processors Based on Intel® Microarchitecture Code Name Nehalem (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		0	Package	<b>EIST Hardware Coordination Disable (R/W)</b> When 0, enables hardware coordination of Enhanced Intel Speedstep Technology request from processor cores; When 1, disables hardware coordination of Enhanced Intel Speedstep Technology requests.
		1	Thread	<b>Energy/Performance Bias Enable (R/W)</b> This bit makes the IA32_ENERGY_PERF_BIAS register (MSR 1B0h) visible to software with Ring 0 privileges. This bit's status (1 or 0) is also reflected by CPUID.(EAX=06h):ECX[3].
		63:2		Reserved.
1ACH	428	MSR_TURBO_POWER_CURRENT_LIMIT		See <a href="http://biosbits.org">http://biosbits.org</a> .
		14:0	Package	<b>TDP Limit (R/W)</b> TDP limit in 1/8 Watt granularity.
		15	Package	<b>TDP Limit Override Enable (R/W)</b> A value = 0 indicates override is not active, and a value = 1 indicates active.
		30:16	Package	<b>TDC Limit (R/W)</b> TDC limit in 1/8 Amp granularity.
		31	Package	<b>TDC Limit Override Enable (R/W)</b> A value = 0 indicates override is not active, and a value = 1 indicates active.
		63:32		Reserved.
1ADH	429	MSR_TURBO_RATIO_LIMIT	Package	<b>Maximum Ratio Limit of Turbo Mode</b> <b>RO</b> if MSR_PLATFORM_INFO.[28] = 0, <b>RW</b> if MSR_PLATFORM_INFO.[28] = 1
		7:0	Package	<b>Maximum Ratio Limit for 1C</b> Maximum turbo ratio limit of 1 core active.
		15:8	Package	<b>Maximum Ratio Limit for 2C</b> Maximum turbo ratio limit of 2 core active.
		23:16	Package	<b>Maximum Ratio Limit for 3C</b> Maximum turbo ratio limit of 3 core active.
		31:24	Package	<b>Maximum Ratio Limit for 4C</b> Maximum turbo ratio limit of 4 core active.
		63:32		Reserved.
1C8H	456	MSR_LBR_SELECT	Core	<b>Last Branch Record Filtering Select Register (R/W)</b> See Section 17.7.2, "Filtering of Last Branch Records."
		0		<b>CPL_EQ_0</b>
		1		<b>CPL_NEQ_0</b>
		2		<b>JCC</b>

Table 35-13. MSRs in Processors Based on Intel® Microarchitecture Code Name Nehalem (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		3		<b>NEAR_REL_CALL</b>
		4		<b>NEAR_IND_CALL</b>
		5		<b>NEAR_RET</b>
		6		<b>NEAR_IND_JMP</b>
		7		<b>NEAR_REL_JMP</b>
		8		<b>FAR_BRANCH</b>
		63:9		Reserved.
1C9H	457	MSR_LASTBRANCH_TOS	Thread	<b>Last Branch Record Stack TOS (R/W)</b> Contains an index (bits 0-3) that points to the MSR containing the most recent branch record. See MSR_LASTBRANCH_O_FROM_IP (at 680H).
1D9H	473	IA32_DEBUGCTL	Thread	<b>Debug Control (R/W)</b> See Table 35-2.
1DDH	477	MSR_LER_FROM_LIP	Thread	<b>Last Exception Record From Linear IP (R)</b> Contains a pointer to the last branch instruction that the processor executed prior to the last exception that was generated or the last interrupt that was handled.
1DEH	478	MSR_LER_TO_LIP	Thread	<b>Last Exception Record To Linear IP (R)</b> This area contains a pointer to the target of the last branch instruction that the processor executed prior to the last exception that was generated or the last interrupt that was handled.
1F2H	498	IA32_SMRR_PHYSBASE	Core	See Table 35-2.
1F3H	499	IA32_SMRR_PHYSMASK	Core	See Table 35-2.
1FCH	508	MSR_POWER_CTL	Core	Power Control Register. See <a href="http://biosbits.org">http://biosbits.org</a> .
		0		Reserved.
		1	Package	<b>C1E Enable (R/W)</b> When set to '1', will enable the CPU to switch to the Minimum Enhanced Intel SpeedStep Technology operating point when all execution cores enter MWAIT (C1).
		63:2		Reserved.
200H	512	IA32_MTRR_PHYSBASE0	Thread	See Table 35-2.
201H	513	IA32_MTRR_PHYSMASK0	Thread	See Table 35-2.
202H	514	IA32_MTRR_PHYSBASE1	Thread	See Table 35-2.
203H	515	IA32_MTRR_PHYSMASK1	Thread	See Table 35-2.
204H	516	IA32_MTRR_PHYSBASE2	Thread	See Table 35-2.
205H	517	IA32_MTRR_PHYSMASK2	Thread	See Table 35-2.
206H	518	IA32_MTRR_PHYSBASE3	Thread	See Table 35-2.

Table 35-13. MSRs in Processors Based on Intel® Microarchitecture Code Name Nehalem (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
207H	519	IA32_MTRR_PHYSMASK3	Thread	See Table 35-2.
208H	520	IA32_MTRR_PHYSBASE4	Thread	See Table 35-2.
209H	521	IA32_MTRR_PHYSMASK4	Thread	See Table 35-2.
20AH	522	IA32_MTRR_PHYSBASE5	Thread	See Table 35-2.
20BH	523	IA32_MTRR_PHYSMASK5	Thread	See Table 35-2.
20CH	524	IA32_MTRR_PHYSBASE6	Thread	See Table 35-2.
20DH	525	IA32_MTRR_PHYSMASK6	Thread	See Table 35-2.
20EH	526	IA32_MTRR_PHYSBASE7	Thread	See Table 35-2.
20FH	527	IA32_MTRR_PHYSMASK7	Thread	See Table 35-2.
210H	528	IA32_MTRR_PHYSBASE8	Thread	See Table 35-2.
211H	529	IA32_MTRR_PHYSMASK8	Thread	See Table 35-2.
212H	530	IA32_MTRR_PHYSBASE9	Thread	See Table 35-2.
213H	531	IA32_MTRR_PHYSMASK9	Thread	See Table 35-2.
250H	592	IA32_MTRR_FIX64K_00000	Thread	See Table 35-2.
258H	600	IA32_MTRR_FIX16K_80000	Thread	See Table 35-2.
259H	601	IA32_MTRR_FIX16K_A0000	Thread	See Table 35-2.
268H	616	IA32_MTRR_FIX4K_C0000	Thread	See Table 35-2.
269H	617	IA32_MTRR_FIX4K_C8000	Thread	See Table 35-2.
26AH	618	IA32_MTRR_FIX4K_D0000	Thread	See Table 35-2.
26BH	619	IA32_MTRR_FIX4K_D8000	Thread	See Table 35-2.
26CH	620	IA32_MTRR_FIX4K_E0000	Thread	See Table 35-2.
26DH	621	IA32_MTRR_FIX4K_E8000	Thread	See Table 35-2.
26EH	622	IA32_MTRR_FIX4K_F0000	Thread	See Table 35-2.
26FH	623	IA32_MTRR_FIX4K_F8000	Thread	See Table 35-2.
277H	631	IA32_PAT	Thread	See Table 35-2.
280H	640	IA32_MC0_CTL2	Package	See Table 35-2.
281H	641	IA32_MC1_CTL2	Package	See Table 35-2.
282H	642	IA32_MC2_CTL2	Core	See Table 35-2.
283H	643	IA32_MC3_CTL2	Core	See Table 35-2.
284H	644	IA32_MC4_CTL2	Core	See Table 35-2.
285H	645	IA32_MC5_CTL2	Core	See Table 35-2.
286H	646	IA32_MC6_CTL2	Package	See Table 35-2.
287H	647	IA32_MC7_CTL2	Package	See Table 35-2.
288H	648	IA32_MC8_CTL2	Package	See Table 35-2.

**Table 35-13. MSRs in Processors Based on Intel® Microarchitecture Code Name Nehalem (Contd.)**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
2FFH	767	IA32_MTRR_DEF_TYPE	Thread	<b>Default Memory Types (R/W)</b> See Table 35-2.
309H	777	IA32_FIXED_CTR0	Thread	<b>Fixed-Function Performance Counter Register 0 (R/W)</b> See Table 35-2.
30AH	778	IA32_FIXED_CTR1	Thread	<b>Fixed-Function Performance Counter Register 1 (R/W)</b> See Table 35-2.
30BH	779	IA32_FIXED_CTR2	Thread	<b>Fixed-Function Performance Counter Register 2 (R/W)</b> See Table 35-2.
345H	837	IA32_PERF_CAPABILITIES	Thread	See Table 35-2. See Section 17.4.1, "IA32_DEBUGCTL MSR."
		5:0		LBR Format. See Table 35-2.
		6		PEBS Record Format.
		7		PEBSSaveArchRegs. See Table 35-2.
		11:8		PEBS_REC_FORMAT. See Table 35-2.
		12		SMM_FREEZE. See Table 35-2.
		63:13		Reserved.
38DH	909	IA32_FIXED_CTR_CTRL	Thread	<b>Fixed-Function-Counter Control Register (R/W)</b> See Table 35-2.
38EH	910	IA32_PERF_GLOBAL_STATUS	Thread	See Table 35-2. See Section 18.4.2, "Global Counter Control Facilities."
38EH	910	MSR_PERF_GLOBAL_STATUS	Thread	<b>(RO)</b>
		61		<b>UNC_Ovf</b> Uncore overflowed if 1.
38FH	911	IA32_PERF_GLOBAL_CTRL	Thread	See Table 35-2. See Section 18.4.2, "Global Counter Control Facilities."
390H	912	IA32_PERF_GLOBAL_OVF_CTRL	Thread	See Table 35-2. See Section 18.4.2, "Global Counter Control Facilities."
390H	912	MSR_PERF_GLOBAL_OVF_CTRL	Thread	<b>(R/W)</b>
		61		<b>CLR_UNC_Ovf</b> Set 1 to clear UNC_Ovf.
3F1H	1009	MSR_PEBS_ENABLE	Thread	See Section 18.7.1.1, "Precise Event Based Sampling (PEBS)."
		0		Enable PEBS on IA32_PMC0. (R/W)
		1		Enable PEBS on IA32_PMC1. (R/W)
		2		Enable PEBS on IA32_PMC2. (R/W)
		3		Enable PEBS on IA32_PMC3. (R/W)
		31:4		Reserved.
		32		Enable Load Latency on IA32_PMC0. (R/W)

Table 35-13. MSRs in Processors Based on Intel® Microarchitecture Code Name Nehalem (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		33		Enable Load Latency on IA32_PMC1. (R/W)
		34		Enable Load Latency on IA32_PMC2. (R/W)
		35		Enable Load Latency on IA32_PMC3. (R/W)
		63:36		Reserved.
3F6H	1014	MSR_PEBS_LD_LAT	Thread	See Section 18.7.1.2, "Load Latency Performance Monitoring Facility."
		15:0		Minimum threshold latency value of tagged load operation that will be counted. (R/W)
		63:36		Reserved.
3F8H	1016	MSR_PKG_C3_RESIDENCY	Package	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		63:0		Package C3 Residency Counter. (R/O) Value since last reset that this package is in processor-specific C3 states. Count at the same frequency as the TSC.
3F9H	1017	MSR_PKG_C6_RESIDENCY	Package	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		63:0		Package C6 Residency Counter. (R/O) Value since last reset that this package is in processor-specific C6 states. Count at the same frequency as the TSC.
3FAH	1018	MSR_PKG_C7_RESIDENCY	Package	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		63:0		Package C7 Residency Counter. (R/O) Value since last reset that this package is in processor-specific C7 states. Count at the same frequency as the TSC.
3FCH	1020	MSR_CORE_C3_RESIDENCY	Core	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		63:0		CORE C3 Residency Counter. (R/O) Value since last reset that this core is in processor-specific C3 states. Count at the same frequency as the TSC.
3FDH	1021	MSR_CORE_C6_RESIDENCY	Core	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		63:0		CORE C6 Residency Counter. (R/O) Value since last reset that this core is in processor-specific C6 states. Count at the same frequency as the TSC.
400H	1024	IA32_MCO_CTL	Package	See Section 15.3.2.1, "IA32_MCI_CTL MSRs."
401H	1025	IA32_MCO_STATUS	Package	See Section 15.3.2.2, "IA32_MCI_STATUS MSRs."



Table 35-13. MSRs in Processors Based on Intel® Microarchitecture Code Name Nehalem (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
402H	1026	IA32_MCO_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The IA32_MCO_ADDR register is either not implemented or contains no address if the ADDR_V flag in the IA32_MCO_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
403H	1027	IA32_MCO_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
404H	1028	IA32_MC1_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
405H	1029	IA32_MC1_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
406H	1030	IA32_MC1_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The IA32_MC1_ADDR register is either not implemented or contains no address if the ADDR_V flag in the IA32_MC1_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
407H	1031	IA32_MC1_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
408H	1032	IA32_MC2_CTL	Core	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
409H	1033	IA32_MC2_STATUS	Core	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
40AH	1034	IA32_MC2_ADDR	Core	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The IA32_MC2_ADDR register is either not implemented or contains no address if the ADDR_V flag in the IA32_MC2_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
40BH	1035	IA32_MC2_MISC	Core	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
40CH	1036	IA32_MC3_CTL	Core	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
40DH	1037	IA32_MC3_STATUS	Core	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
40EH	1038	IA32_MC3_ADDR	Core	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The MSR_MC4_ADDR register is either not implemented or contains no address if the ADDR_V flag in the MSR_MC4_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
40FH	1039	IA32_MC3_MISC	Core	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
410H	1040	IA32_MC4_CTL	Core	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
411H	1041	IA32_MC4_STATUS	Core	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
412H	1042	IA32_MC4_ADDR	Core	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The MSR_MC3_ADDR register is either not implemented or contains no address if the ADDR_V flag in the MSR_MC3_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.

Table 35-13. MSRs in Processors Based on Intel® Microarchitecture Code Name Nehalem (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
413H	1043	IA32_MC4_MISC	Core	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
414H	1044	IA32_MC5_CTL	Core	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
415H	1045	IA32_MC5_STATUS	Core	See Section 15.3.2.2, "IA32_MCi_STATUS MSRs."
416H	1046	IA32_MC5_ADDR	Core	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
417H	1047	IA32_MC5_MISC	Core	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
418H	1048	IA32_MC6_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
419H	1049	IA32_MC6_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRs," and Chapter 16.
41AH	1050	IA32_MC6_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
41BH	1051	IA32_MC6_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
41CH	1052	IA32_MC7_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
41DH	1053	IA32_MC7_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRs," and Chapter 16.
41EH	1054	IA32_MC7_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
41FH	1055	IA32_MC7_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
420H	1056	IA32_MC8_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
421H	1057	IA32_MC8_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRs," and Chapter 16.
422H	1058	IA32_MC8_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
423H	1059	IA32_MC8_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
480H	1152	IA32_VMX_BASIC	Thread	<b>Reporting Register of Basic VMX Capabilities (R/O)</b> See Table 35-2. See Appendix A.1, "Basic VMX Information."
481H	1153	IA32_VMX_PINBASED_CTLS	Thread	<b>Capability Reporting Register of Pin-based VM-execution Controls (R/O)</b> See Table 35-2. See Appendix A.3, "VM-Execution Controls."
482H	1154	IA32_VMX_PROCBASED_CTLS	Thread	<b>Capability Reporting Register of Primary Processor-based VM-execution Controls (R/O)</b> See Appendix A.3, "VM-Execution Controls."
483H	1155	IA32_VMX_EXIT_CTLS	Thread	<b>Capability Reporting Register of VM-exit Controls (R/O)</b> See Table 35-2. See Appendix A.4, "VM-Exit Controls."
484H	1156	IA32_VMX_ENTRY_CTLS	Thread	<b>Capability Reporting Register of VM-entry Controls (R/O)</b> See Table 35-2. See Appendix A.5, "VM-Entry Controls."
485H	1157	IA32_VMX_MISC	Thread	<b>Reporting Register of Miscellaneous VMX Capabilities (R/O)</b> See Table 35-2. See Appendix A.6, "Miscellaneous Data."

Table 35-13. MSRs in Processors Based on Intel® Microarchitecture Code Name Nehalem (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
486H	1158	IA32_VMX_CRO_FIXED0	Thread	<b>Capability Reporting Register of CRO Bits Fixed to 0 (R/O)</b> See Table 35-2. See Appendix A.7, "VMX-Fixed Bits in CRO."
487H	1159	IA32_VMX_CRO_FIXED1	Thread	<b>Capability Reporting Register of CRO Bits Fixed to 1 (R/O)</b> See Table 35-2. See Appendix A.7, "VMX-Fixed Bits in CRO."
488H	1160	IA32_VMX_CR4_FIXED0	Thread	<b>Capability Reporting Register of CR4 Bits Fixed to 0 (R/O)</b> See Table 35-2. See Appendix A.8, "VMX-Fixed Bits in CR4."
489H	1161	IA32_VMX_CR4_FIXED1	Thread	<b>Capability Reporting Register of CR4 Bits Fixed to 1 (R/O)</b> See Table 35-2. See Appendix A.8, "VMX-Fixed Bits in CR4."
48AH	1162	IA32_VMX_VMCS_ENUM	Thread	<b>Capability Reporting Register of VMCS Field Enumeration (R/O).</b> See Table 35-2. See Appendix A.9, "VMCS Enumeration."
48BH	1163	IA32_VMX_PROCBASED_CTLDS2	Thread	<b>Capability Reporting Register of Secondary Processor-based VM-execution Controls (R/O)</b> See Appendix A.3, "VM-Execution Controls."
600H	1536	IA32_DS_AREA	Thread	<b>DS Save Area (R/W)</b> See Table 35-2. See Section 18.1.2.4, "Debug Store (DS) Mechanism."
680H	1664	MSR_LASTBRANCH_0_FROM_IP	Thread	<b>Last Branch Record 0 From IP (R/W)</b> One of sixteen pairs of last branch record registers on the last branch record stack. The From_IP part of the stack contains pointers to the <b>source instruction</b> . See also: <ul style="list-style-type: none"> <li>▪ Last Branch Record Stack TOS at 1C9H</li> <li>▪ Section 17.7.1 and record format in Section 17.4.8.1</li> </ul>
681H	1665	MSR_LASTBRANCH_1_FROM_IP	Thread	<b>Last Branch Record 1 From IP (R/W)</b> See description of MSR_LASTBRANCH_0_FROM_IP.
682H	1666	MSR_LASTBRANCH_2_FROM_IP	Thread	<b>Last Branch Record 2 From IP (R/W)</b> See description of MSR_LASTBRANCH_0_FROM_IP.
683H	1667	MSR_LASTBRANCH_3_FROM_IP	Thread	<b>Last Branch Record 3 From IP (R/W)</b> See description of MSR_LASTBRANCH_0_FROM_IP.
684H	1668	MSR_LASTBRANCH_4_FROM_IP	Thread	<b>Last Branch Record 4 From IP (R/W)</b> See description of MSR_LASTBRANCH_0_FROM_IP.
685H	1669	MSR_LASTBRANCH_5_FROM_IP	Thread	<b>Last Branch Record 5 From IP (R/W)</b> See description of MSR_LASTBRANCH_0_FROM_IP.
686H	1670	MSR_LASTBRANCH_6_FROM_IP	Thread	<b>Last Branch Record 6 From IP (R/W)</b> See description of MSR_LASTBRANCH_0_FROM_IP.

Table 35-13. MSRs in Processors Based on Intel® Microarchitecture Code Name Nehalem (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
687H	1671	MSR_LASTBRANCH_7_FROM_IP	Thread	<b>Last Branch Record 7 From IP (R/W)</b> See description of MSR_LASTBRANCH_0_FROM_IP.
688H	1672	MSR_LASTBRANCH_8_FROM_IP	Thread	<b>Last Branch Record 8 From IP (R/W)</b> See description of MSR_LASTBRANCH_0_FROM_IP.
689H	1673	MSR_LASTBRANCH_9_FROM_IP	Thread	<b>Last Branch Record 9 From IP (R/W)</b> See description of MSR_LASTBRANCH_0_FROM_IP.
68AH	1674	MSR_LASTBRANCH_10_FROM_IP	Thread	<b>Last Branch Record 10 From IP (R/W)</b> See description of MSR_LASTBRANCH_0_FROM_IP.
68BH	1675	MSR_LASTBRANCH_11_FROM_IP	Thread	<b>Last Branch Record 11 From IP (R/W)</b> See description of MSR_LASTBRANCH_0_FROM_IP.
68CH	1676	MSR_LASTBRANCH_12_FROM_IP	Thread	<b>Last Branch Record 12 From IP (R/W)</b> See description of MSR_LASTBRANCH_0_FROM_IP.
68DH	1677	MSR_LASTBRANCH_13_FROM_IP	Thread	<b>Last Branch Record 13 From IP (R/W)</b> See description of MSR_LASTBRANCH_0_FROM_IP.
68EH	1678	MSR_LASTBRANCH_14_FROM_IP	Thread	<b>Last Branch Record 14 From IP (R/W)</b> See description of MSR_LASTBRANCH_0_FROM_IP.
68FH	1679	MSR_LASTBRANCH_15_FROM_IP	Thread	<b>Last Branch Record 15 From IP (R/W)</b> See description of MSR_LASTBRANCH_0_FROM_IP.
6C0H	1728	MSR_LASTBRANCH_0_TO_IP	Thread	<b>Last Branch Record 0 To IP (R/W)</b> One of sixteen pairs of last branch record registers on the last branch record stack. This part of the stack contains pointers to the destination instruction.
6C1H	1729	MSR_LASTBRANCH_1_TO_IP	Thread	<b>Last Branch Record 1 To IP (R/W)</b> See description of MSR_LASTBRANCH_0_TO_IP.
6C2H	1730	MSR_LASTBRANCH_2_TO_IP	Thread	<b>Last Branch Record 2 To IP (R/W)</b> See description of MSR_LASTBRANCH_0_TO_IP.
6C3H	1731	MSR_LASTBRANCH_3_TO_IP	Thread	<b>Last Branch Record 3 To IP (R/W)</b> See description of MSR_LASTBRANCH_0_TO_IP.
6C4H	1732	MSR_LASTBRANCH_4_TO_IP	Thread	<b>Last Branch Record 4 To IP (R/W)</b> See description of MSR_LASTBRANCH_0_TO_IP.
6C5H	1733	MSR_LASTBRANCH_5_TO_IP	Thread	<b>Last Branch Record 5 To IP (R/W)</b> See description of MSR_LASTBRANCH_0_TO_IP.
6C6H	1734	MSR_LASTBRANCH_6_TO_IP	Thread	<b>Last Branch Record 6 To IP (R/W)</b> See description of MSR_LASTBRANCH_0_TO_IP.
6C7H	1735	MSR_LASTBRANCH_7_TO_IP	Thread	<b>Last Branch Record 7 To IP (R/W)</b> See description of MSR_LASTBRANCH_0_TO_IP.
6C8H	1736	MSR_LASTBRANCH_8_TO_IP	Thread	<b>Last Branch Record 8 To IP (R/W)</b> See description of MSR_LASTBRANCH_0_TO_IP.

Table 35-13. MSRs in Processors Based on Intel® Microarchitecture Code Name Nehalem (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
6C9H	1737	MSR_LASTBRANCH_9_TO_IP	Thread	<b>Last Branch Record 9 To IP (R/W)</b> See description of MSR_LASTBRANCH_0_TO_IP.
6CAH	1738	MSR_LASTBRANCH_10_TO_IP	Thread	<b>Last Branch Record 10 To IP (R/W)</b> See description of MSR_LASTBRANCH_0_TO_IP.
6CBH	1739	MSR_LASTBRANCH_11_TO_IP	Thread	<b>Last Branch Record 11 To IP (R/W)</b> See description of MSR_LASTBRANCH_0_TO_IP.
6CCH	1740	MSR_LASTBRANCH_12_TO_IP	Thread	<b>Last Branch Record 12 To IP (R/W)</b> See description of MSR_LASTBRANCH_0_TO_IP.
6CDH	1741	MSR_LASTBRANCH_13_TO_IP	Thread	<b>Last Branch Record 13 To IP (R/W)</b> See description of MSR_LASTBRANCH_0_TO_IP.
6CEH	1742	MSR_LASTBRANCH_14_TO_IP	Thread	<b>Last Branch Record 14 To IP (R/W)</b> See description of MSR_LASTBRANCH_0_TO_IP.
6CFH	1743	MSR_LASTBRANCH_15_TO_IP	Thread	<b>Last Branch Record 15 To IP (R/W)</b> See description of MSR_LASTBRANCH_0_TO_IP.
802H	2050	IA32_X2APIC_APICID	Thread	x2APIC ID register (R/O) See x2APIC Specification.
803H	2051	IA32_X2APIC_VERSION	Thread	x2APIC Version register (R/O)
808H	2056	IA32_X2APIC_TPR	Thread	x2APIC Task Priority register (R/W)
80AH	2058	IA32_X2APIC_PPR	Thread	x2APIC Processor Priority register (R/O)
80BH	2059	IA32_X2APIC_EOI	Thread	x2APIC EOI register (W/O)
80DH	2061	IA32_X2APIC_LDR	Thread	x2APIC Logical Destination register (R/O)
80FH	2063	IA32_X2APIC_SIVR	Thread	x2APIC Spurious Interrupt Vector register (R/W)
810H	2064	IA32_X2APIC_ISR0	Thread	x2APIC In-Service register bits [31:0] (R/O)
811H	2065	IA32_X2APIC_ISR1	Thread	x2APIC In-Service register bits [63:32] (R/O)
812H	2066	IA32_X2APIC_ISR2	Thread	x2APIC In-Service register bits [95:64] (R/O)
813H	2067	IA32_X2APIC_ISR3	Thread	x2APIC In-Service register bits [127:96] (R/O)
814H	2068	IA32_X2APIC_ISR4	Thread	x2APIC In-Service register bits [159:128] (R/O)
815H	2069	IA32_X2APIC_ISR5	Thread	x2APIC In-Service register bits [191:160] (R/O)
816H	2070	IA32_X2APIC_ISR6	Thread	x2APIC In-Service register bits [223:192] (R/O)
817H	2071	IA32_X2APIC_ISR7	Thread	x2APIC In-Service register bits [255:224] (R/O)
818H	2072	IA32_X2APIC_TMRO	Thread	x2APIC Trigger Mode register bits [31:0] (R/O)
819H	2073	IA32_X2APIC_TMR1	Thread	x2APIC Trigger Mode register bits [63:32] (R/O)
81AH	2074	IA32_X2APIC_TMR2	Thread	x2APIC Trigger Mode register bits [95:64] (R/O)
81BH	2075	IA32_X2APIC_TMR3	Thread	x2APIC Trigger Mode register bits [127:96] (R/O)
81CH	2076	IA32_X2APIC_TMR4	Thread	x2APIC Trigger Mode register bits [159:128] (R/O)
81DH	2077	IA32_X2APIC_TMR5	Thread	x2APIC Trigger Mode register bits [191:160] (R/O)
81EH	2078	IA32_X2APIC_TMR6	Thread	x2APIC Trigger Mode register bits [223:192] (R/O)
81FH	2079	IA32_X2APIC_TMR7	Thread	x2APIC Trigger Mode register bits [255:224] (R/O)

Table 35-13. MSRs in Processors Based on Intel® Microarchitecture Code Name Nehalem (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
820H	2080	IA32_X2APIC_IRR0	Thread	x2APIC Interrupt Request register bits [31:0] (R/O)
821H	2081	IA32_X2APIC_IRR1	Thread	x2APIC Interrupt Request register bits [63:32] (R/O)
822H	2082	IA32_X2APIC_IRR2	Thread	x2APIC Interrupt Request register bits [95:64] (R/O)
823H	2083	IA32_X2APIC_IRR3	Thread	x2APIC Interrupt Request register bits [127:96] (R/O)
824H	2084	IA32_X2APIC_IRR4	Thread	x2APIC Interrupt Request register bits [159:128] (R/O)
825H	2085	IA32_X2APIC_IRR5	Thread	x2APIC Interrupt Request register bits [191:160] (R/O)
826H	2086	IA32_X2APIC_IRR6	Thread	x2APIC Interrupt Request register bits [223:192] (R/O)
827H	2087	IA32_X2APIC_IRR7	Thread	x2APIC Interrupt Request register bits [255:224] (R/O)
828H	2088	IA32_X2APIC_ESR	Thread	x2APIC Error Status register (R/W)
82FH	2095	IA32_X2APIC_LVT_CMCI	Thread	x2APIC LVT Corrected Machine Check Interrupt register (R/W)
830H	2096	IA32_X2APIC_ICR	Thread	x2APIC Interrupt Command register (R/W)
832H	2098	IA32_X2APIC_LVT_TIMER	Thread	x2APIC LVT Timer Interrupt register (R/W)
833H	2099	IA32_X2APIC_LVT_THERMAL	Thread	x2APIC LVT Thermal Sensor Interrupt register (R/W)
834H	2100	IA32_X2APIC_LVT_PMI	Thread	x2APIC LVT Performance Monitor register (R/W)
835H	2101	IA32_X2APIC_LVT_LINT0	Thread	x2APIC LVT LINT0 register (R/W)
836H	2102	IA32_X2APIC_LVT_LINT1	Thread	x2APIC LVT LINT1 register (R/W)
837H	2103	IA32_X2APIC_LVT_ERROR	Thread	x2APIC LVT Error register (R/W)
838H	2104	IA32_X2APIC_INIT_COUNT	Thread	x2APIC Initial Count register (R/W)
839H	2105	IA32_X2APIC_CUR_COUNT	Thread	x2APIC Current Count register (R/O)
83EH	2110	IA32_X2APIC_DIV_CONF	Thread	x2APIC Divide Configuration register (R/W)
83FH	2111	IA32_X2APIC_SELF_IPI	Thread	x2APIC Self IPI register (W/O)
C000_0080H		IA32_EFER	Thread	<b>Extended Feature Enables</b> See Table 35-2.
C000_0081H		IA32_STAR	Thread	<b>System Call Target Address (R/W)</b> See Table 35-2.
C000_0082H		IA32_LSTAR	Thread	<b>IA-32e Mode System Call Target Address (R/W)</b> See Table 35-2.
C000_0084H		IA32_FMASK	Thread	<b>System Call Flag Mask (R/W)</b> See Table 35-2.
C000_0100H		IA32_FS_BASE	Thread	<b>Map of BASE Address of FS (R/W)</b> See Table 35-2.
C000_0101H		IA32_GS_BASE	Thread	<b>Map of BASE Address of GS (R/W)</b> See Table 35-2.
C000_0102H		IA32_KERNEL_GSBASE	Thread	<b>Swap Target of BASE Address of GS (R/W)</b> See Table 35-2.

**Table 35-13. MSRs in Processors Based on Intel® Microarchitecture Code Name Nehalem (Contd.)**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
C000_0103H		IA32_TSC_AUX	Thread	<b>AUXILIARY TSC Signature. (R/W)</b> See Table 35-2 and Section 17.15.2, "IA32_TSC_AUX Register and RDTSCP Support."

### 35.6.1 Additional MSRs in the Intel® Xeon® Processor 5500 and 3400 Series

Intel Xeon Processor 5500 and 3400 series support additional model-specific registers listed in Table 35-14. These MSRs also apply to Intel Core i7 and i5 processor family CPUID signature with DisplayFamily\_DisplayModel of 06\_1AH, 06\_1EH and 06\_1FH, see Table 35-1.

**Table 35-14. Additional MSRs in Intel® Xeon® Processor 5500 and 3400 Series**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
1ADH	429	MSR_TURBO_RATIO_LIMIT	Package	Actual maximum turbo frequency is multiplied by 133.33MHz. (not available to model 06_2EH)
		7:0		<b>Maximum Turbo Ratio Limit 1C (R/O)</b> Maximum Turbo mode ratio limit with 1 core active.
		15:8		<b>Maximum Turbo Ratio Limit 2C (R/O)</b> Maximum Turbo mode ratio limit with 2cores active.
		23:16		<b>Maximum Turbo Ratio Limit 3C (R/O)</b> Maximum Turbo mode ratio limit with 3cores active.
		31:24		<b>Maximum Turbo Ratio Limit 4C (R/O)</b> Maximum Turbo mode ratio limit with 4 cores active.
		63:32		Reserved.
301H	769	MSR_GQ_SNOOP_MESF	Package	
		0		<b>From M to S (R/W)</b>
		1		<b>From E to S (R/W)</b>
		2		<b>From S to S (R/W)</b>
		3		<b>From F to S (R/W)</b>
		4		<b>From M to I (R/W)</b>
		5		<b>From E to I (R/W)</b>
		6		<b>From S to I (R/W)</b>
		7		<b>From F to I (R/W)</b>
63:8		Reserved.		
391H	913	MSR_UNCORE_PERF_GLOBAL_CTRL	Package	See Section 18.7.2.1, "Uncore Performance Monitoring Management Facility."
392H	914	MSR_UNCORE_PERF_GLOBAL_STATUS	Package	See Section 18.7.2.1, "Uncore Performance Monitoring Management Facility."
393H	915	MSR_UNCORE_PERF_GLOBAL_OVF_CTRL	Package	See Section 18.7.2.1, "Uncore Performance Monitoring Management Facility."



**Table 35-14. Additional MSRs in Intel® Xeon® Processor 5500 and 3400 Series (Contd.)**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
394H	916	MSR_UNCORE_FIXED_CTR0	Package	See Section 18.7.2.1, "Uncore Performance Monitoring Management Facility."
395H	917	MSR_UNCORE_FIXED_CTR_CTRL	Package	See Section 18.7.2.1, "Uncore Performance Monitoring Management Facility."
396H	918	MSR_UNCORE_ADDR_OPCODE_MATCH	Package	See Section 18.7.2.3, "Uncore Address/Opcode Match MSR."
3B0H	960	MSR_UNCORE_PMC0	Package	See Section 18.7.2.2, "Uncore Performance Event Configuration Facility."
3B1H	961	MSR_UNCORE_PMC1	Package	See Section 18.7.2.2, "Uncore Performance Event Configuration Facility."
3B2H	962	MSR_UNCORE_PMC2	Package	See Section 18.7.2.2, "Uncore Performance Event Configuration Facility."
3B3H	963	MSR_UNCORE_PMC3	Package	See Section 18.7.2.2, "Uncore Performance Event Configuration Facility."
3B4H	964	MSR_UNCORE_PMC4	Package	See Section 18.7.2.2, "Uncore Performance Event Configuration Facility."
3B5H	965	MSR_UNCORE_PMC5	Package	See Section 18.7.2.2, "Uncore Performance Event Configuration Facility."
3B6H	966	MSR_UNCORE_PMC6	Package	See Section 18.7.2.2, "Uncore Performance Event Configuration Facility."
3B7H	967	MSR_UNCORE_PMC7	Package	See Section 18.7.2.2, "Uncore Performance Event Configuration Facility."
3C0H	944	MSR_UNCORE_PERFEVTSEL0	Package	See Section 18.7.2.2, "Uncore Performance Event Configuration Facility."
3C1H	945	MSR_UNCORE_PERFEVTSEL1	Package	See Section 18.7.2.2, "Uncore Performance Event Configuration Facility."
3C2H	946	MSR_UNCORE_PERFEVTSEL2	Package	See Section 18.7.2.2, "Uncore Performance Event Configuration Facility."
3C3H	947	MSR_UNCORE_PERFEVTSEL3	Package	See Section 18.7.2.2, "Uncore Performance Event Configuration Facility."
3C4H	948	MSR_UNCORE_PERFEVTSEL4	Package	See Section 18.7.2.2, "Uncore Performance Event Configuration Facility."
3C5H	949	MSR_UNCORE_PERFEVTSEL5	Package	See Section 18.7.2.2, "Uncore Performance Event Configuration Facility."
3C6H	950	MSR_UNCORE_PERFEVTSEL6	Package	See Section 18.7.2.2, "Uncore Performance Event Configuration Facility."
3C7H	951	MSR_UNCORE_PERFEVTSEL7	Package	See Section 18.7.2.2, "Uncore Performance Event Configuration Facility."

### 35.6.2 Additional MSRs in the Intel® Xeon® Processor 7500 Series

Intel Xeon Processor 7500 series support MSRs listed in Table 35-13 (except MSR address 1ADH) and additional model-specific registers listed in Table 35-15. These processors have a CPUID signature with DisplayFamily\_DisplayModel of 06\_2EH.



Table 35-15. Additional MSRs in Intel® Xeon® Processor 7500 Series

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
1ADH	429	MSR_TURBO_RATIO_LIMIT	Package	<b>Reserved</b> Attempt to read/write will cause #UD.
289H	649	IA32_MC9_CTL2	Package	See Table 35-2.
28AH	650	IA32_MC10_CTL2	Package	See Table 35-2.
28BH	651	IA32_MC11_CTL2	Package	See Table 35-2.
28CH	652	IA32_MC12_CTL2	Package	See Table 35-2.
28DH	653	IA32_MC13_CTL2	Package	See Table 35-2.
28EH	654	IA32_MC14_CTL2	Package	See Table 35-2.
28FH	655	IA32_MC15_CTL2	Package	See Table 35-2.
290H	656	IA32_MC16_CTL2	Package	See Table 35-2.
291H	657	IA32_MC17_CTL2	Package	See Table 35-2.
292H	658	IA32_MC18_CTL2	Package	See Table 35-2.
293H	659	IA32_MC19_CTL2	Package	See Table 35-2.
294H	660	IA32_MC20_CTL2	Package	See Table 35-2.
295H	661	IA32_MC21_CTL2	Package	See Table 35-2.
394H	816	MSR_W_PMON_FIXED_CTR	Package	Uncore W-box perfmon fixed counter
395H	817	MSR_W_PMON_FIXED_CTR_CTL	Package	Uncore U-box perfmon fixed counter control MSR
424H	1060	IA32_MC9_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
425H	1061	IA32_MC9_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRs," and Chapter 16.
426H	1062	IA32_MC9_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
427H	1063	IA32_MC9_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
428H	1064	IA32_MC10_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
429H	1065	IA32_MC10_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRs," and Chapter 16.
42AH	1066	IA32_MC10_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
42BH	1067	IA32_MC10_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
42CH	1068	IA32_MC11_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
42DH	1069	IA32_MC11_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRs," and Chapter 16.
42EH	1070	IA32_MC11_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
42FH	1071	IA32_MC11_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
430H	1072	IA32_MC12_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
431H	1073	IA32_MC12_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRs," and Chapter 16.
432H	1074	IA32_MC12_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
433H	1075	IA32_MC12_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
434H	1076	IA32_MC13_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
435H	1077	IA32_MC13_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRs," and Chapter 16.

Table 35-15. Additional MSRs in Intel® Xeon® Processor 7500 Series (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
436H	1078	IA32_MC13_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
437H	1079	IA32_MC13_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
438H	1080	IA32_MC14_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
439H	1081	IA32_MC14_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 16.
43AH	1082	IA32_MC14_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
43BH	1083	IA32_MC14_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
43CH	1084	IA32_MC15_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
43DH	1085	IA32_MC15_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 16.
43EH	1086	IA32_MC15_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
43FH	1087	IA32_MC15_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
440H	1088	IA32_MC16_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
441H	1089	IA32_MC16_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 16.
442H	1090	IA32_MC16_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
443H	1091	IA32_MC16_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
444H	1092	IA32_MC17_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
445H	1093	IA32_MC17_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 16.
446H	1094	IA32_MC17_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
447H	1095	IA32_MC17_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
448H	1096	IA32_MC18_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
449H	1097	IA32_MC18_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 16.
44AH	1098	IA32_MC18_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
44BH	1099	IA32_MC18_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
44CH	1100	IA32_MC19_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
44DH	1101	IA32_MC19_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 16.
44EH	1102	IA32_MC19_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
44FH	1103	IA32_MC19_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
450H	1104	IA32_MC20_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
451H	1105	IA32_MC20_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 16.
452H	1106	IA32_MC20_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
453H	1107	IA32_MC20_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
454H	1108	IA32_MC21_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
455H	1109	IA32_MC21_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 16.
456H	1110	IA32_MC21_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
457H	1111	IA32_MC21_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
C00H	3072	MSR_U_PMON_GLOBAL_CTRL	Package	Uncore U-box perfmon global control MSR.

**Table 35-15. Additional MSRs in Intel® Xeon® Processor 7500 Series (Contd.)**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
C01H	3073	MSR_U_PMON_GLOBAL_STATUS	Package	Uncore U-box perfmon global status MSR.
C02H	3074	MSR_U_PMON_GLOBAL_OVF_CTRL	Package	Uncore U-box perfmon global overflow control MSR.
C10H	3088	MSR_U_PMON_EVNT_SEL	Package	Uncore U-box perfmon event select MSR.
C11H	3089	MSR_U_PMON_CTR	Package	Uncore U-box perfmon counter MSR.
C20H	3104	MSR_B0_PMON_BOX_CTRL	Package	Uncore B-box 0 perfmon local box control MSR.
C21H	3105	MSR_B0_PMON_BOX_STATUS	Package	Uncore B-box 0 perfmon local box status MSR.
C22H	3106	MSR_B0_PMON_BOX_OVF_CTRL	Package	Uncore B-box 0 perfmon local box overflow control MSR.
C30H	3120	MSR_B0_PMON_EVNT_SELO	Package	Uncore B-box 0 perfmon event select MSR.
C31H	3121	MSR_B0_PMON_CTR0	Package	Uncore B-box 0 perfmon counter MSR.
C32H	3122	MSR_B0_PMON_EVNT_SEL1	Package	Uncore B-box 0 perfmon event select MSR.
C33H	3123	MSR_B0_PMON_CTR1	Package	Uncore B-box 0 perfmon counter MSR.
C34H	3124	MSR_B0_PMON_EVNT_SEL2	Package	Uncore B-box 0 perfmon event select MSR.
C35H	3125	MSR_B0_PMON_CTR2	Package	Uncore B-box 0 perfmon counter MSR.
C36H	3126	MSR_B0_PMON_EVNT_SEL3	Package	Uncore B-box 0 perfmon event select MSR.
C37H	3127	MSR_B0_PMON_CTR3	Package	Uncore B-box 0 perfmon counter MSR.
C40H	3136	MSR_S0_PMON_BOX_CTRL	Package	Uncore S-box 0 perfmon local box control MSR.
C41H	3137	MSR_S0_PMON_BOX_STATUS	Package	Uncore S-box 0 perfmon local box status MSR.
C42H	3138	MSR_S0_PMON_BOX_OVF_CTRL	Package	Uncore S-box 0 perfmon local box overflow control MSR.
C50H	3152	MSR_S0_PMON_EVNT_SELO	Package	Uncore S-box 0 perfmon event select MSR.
C51H	3153	MSR_S0_PMON_CTR0	Package	Uncore S-box 0 perfmon counter MSR.
C52H	3154	MSR_S0_PMON_EVNT_SEL1	Package	Uncore S-box 0 perfmon event select MSR.
C53H	3155	MSR_S0_PMON_CTR1	Package	Uncore S-box 0 perfmon counter MSR.
C54H	3156	MSR_S0_PMON_EVNT_SEL2	Package	Uncore S-box 0 perfmon event select MSR.
C55H	3157	MSR_S0_PMON_CTR2	Package	Uncore S-box 0 perfmon counter MSR.

Table 35-15. Additional MSRs in Intel® Xeon® Processor 7500 Series (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
C56H	3158	MSR_S0_PMON_EVNT_SEL3	Package	Uncore S-box 0 perfmon event select MSR.
C57H	3159	MSR_S0_PMON_CTR3	Package	Uncore S-box 0 perfmon counter MSR.
C60H	3168	MSR_B1_PMON_BOX_CTRL	Package	Uncore B-box 1 perfmon local box control MSR.
C61H	3169	MSR_B1_PMON_BOX_STATUS	Package	Uncore B-box 1 perfmon local box status MSR.
C62H	3170	MSR_B1_PMON_BOX_OVF_CTRL	Package	Uncore B-box 1 perfmon local box overflow control MSR.
C70H	3184	MSR_B1_PMON_EVNT_SELO	Package	Uncore B-box 1 perfmon event select MSR.
C71H	3185	MSR_B1_PMON_CTR0	Package	Uncore B-box 1 perfmon counter MSR.
C72H	3186	MSR_B1_PMON_EVNT_SEL1	Package	Uncore B-box 1 perfmon event select MSR.
C73H	3187	MSR_B1_PMON_CTR1	Package	Uncore B-box 1 perfmon counter MSR.
C74H	3188	MSR_B1_PMON_EVNT_SEL2	Package	Uncore B-box 1 perfmon event select MSR.
C75H	3189	MSR_B1_PMON_CTR2	Package	Uncore B-box 1 perfmon counter MSR.
C76H	3190	MSR_B1_PMON_EVNT_SEL3	Package	Uncore B-box 1 vperfmon event select MSR.
C77H	3191	MSR_B1_PMON_CTR3	Package	Uncore B-box 1 perfmon counter MSR.
C80H	3120	MSR_W_PMON_BOX_CTRL	Package	Uncore W-box perfmon local box control MSR.
C81H	3121	MSR_W_PMON_BOX_STATUS	Package	Uncore W-box perfmon local box status MSR.
C82H	3122	MSR_W_PMON_BOX_OVF_CTRL	Package	Uncore W-box perfmon local box overflow control MSR.
C90H	3136	MSR_W_PMON_EVNT_SELO	Package	Uncore W-box perfmon event select MSR.
C91H	3137	MSR_W_PMON_CTR0	Package	Uncore W-box perfmon counter MSR.
C92H	3138	MSR_W_PMON_EVNT_SEL1	Package	Uncore W-box perfmon event select MSR.
C93H	3139	MSR_W_PMON_CTR1	Package	Uncore W-box perfmon counter MSR.
C94H	3140	MSR_W_PMON_EVNT_SEL2	Package	Uncore W-box perfmon event select MSR.
C95H	3141	MSR_W_PMON_CTR2	Package	Uncore W-box perfmon counter MSR.
C96H	3142	MSR_W_PMON_EVNT_SEL3	Package	Uncore W-box perfmon event select MSR.
C97H	3143	MSR_W_PMON_CTR3	Package	Uncore W-box perfmon counter MSR.

Table 35-15. Additional MSRs in Intel® Xeon® Processor 7500 Series (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
CA0H	3232	MSR_M0_PMON_BOX_CTRL	Package	Uncore M-box 0 perfmon local box control MSR.
CA1H	3233	MSR_M0_PMON_BOX_STATUS	Package	Uncore M-box 0 perfmon local box status MSR.
CA2H	3234	MSR_M0_PMON_BOX_OVF_CTRL	Package	Uncore M-box 0 perfmon local box overflow control MSR.
CA4H	3236	MSR_M0_PMON_TIMESTAMP	Package	Uncore M-box 0 perfmon time stamp unit select MSR.
CA5H	3237	MSR_M0_PMON_DSP	Package	Uncore M-box 0 perfmon DSP unit select MSR.
CA6H	3238	MSR_M0_PMON_ISS	Package	Uncore M-box 0 perfmon ISS unit select MSR.
CA7H	3239	MSR_M0_PMON_MAP	Package	Uncore M-box 0 perfmon MAP unit select MSR.
CA8H	3240	MSR_M0_PMON_MSC_THR	Package	Uncore M-box 0 perfmon MIC THR select MSR.
CA9H	3241	MSR_M0_PMON_PGT	Package	Uncore M-box 0 perfmon PGT unit select MSR.
CAAH	3242	MSR_M0_PMON_PLD	Package	Uncore M-box 0 perfmon PLD unit select MSR.
CABH	3243	MSR_M0_PMON_ZDP	Package	Uncore M-box 0 perfmon ZDP unit select MSR.
CBOH	3248	MSR_M0_PMON_EVNT_SELO	Package	Uncore M-box 0 perfmon event select MSR.
CB1H	3249	MSR_M0_PMON_CTR0	Package	Uncore M-box 0 perfmon counter MSR.
CB2H	3250	MSR_M0_PMON_EVNT_SEL1	Package	Uncore M-box 0 perfmon event select MSR.
CB3H	3251	MSR_M0_PMON_CTR1	Package	Uncore M-box 0 perfmon counter MSR.
CB4H	3252	MSR_M0_PMON_EVNT_SEL2	Package	Uncore M-box 0 perfmon event select MSR.
CB5H	3253	MSR_M0_PMON_CTR2	Package	Uncore M-box 0 perfmon counter MSR.
CB6H	3254	MSR_M0_PMON_EVNT_SEL3	Package	Uncore M-box 0 perfmon event select MSR.
CB7H	3255	MSR_M0_PMON_CTR3	Package	Uncore M-box 0 perfmon counter MSR.
CB8H	3256	MSR_M0_PMON_EVNT_SEL4	Package	Uncore M-box 0 perfmon event select MSR.
CB9H	3257	MSR_M0_PMON_CTR4	Package	Uncore M-box 0 perfmon counter MSR.
CBAH	3258	MSR_M0_PMON_EVNT_SEL5	Package	Uncore M-box 0 perfmon event select MSR.
CBBH	3259	MSR_M0_PMON_CTR5	Package	Uncore M-box 0 perfmon counter MSR.
CC0H	3264	MSR_S1_PMON_BOX_CTRL	Package	Uncore S-box 1 perfmon local box control MSR.
CC1H	3265	MSR_S1_PMON_BOX_STATUS	Package	Uncore S-box 1 perfmon local box status MSR.
CC2H	3266	MSR_S1_PMON_BOX_OVF_CTRL	Package	Uncore S-box 1 perfmon local box overflow control MSR.

Table 35-15. Additional MSRs in Intel® Xeon® Processor 7500 Series (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
CD0H	3280	MSR_S1_PMON_EVNT_SEL0	Package	Uncore S-box 1 perfmon event select MSR.
CD1H	3281	MSR_S1_PMON_CTRL0	Package	Uncore S-box 1 perfmon counter MSR.
CD2H	3282	MSR_S1_PMON_EVNT_SEL1	Package	Uncore S-box 1 perfmon event select MSR.
CD3H	3283	MSR_S1_PMON_CTRL1	Package	Uncore S-box 1 perfmon counter MSR.
CD4H	3284	MSR_S1_PMON_EVNT_SEL2	Package	Uncore S-box 1 perfmon event select MSR.
CD5H	3285	MSR_S1_PMON_CTRL2	Package	Uncore S-box 1 perfmon counter MSR.
CD6H	3286	MSR_S1_PMON_EVNT_SEL3	Package	Uncore S-box 1 perfmon event select MSR.
CD7H	3287	MSR_S1_PMON_CTRL3	Package	Uncore S-box 1 perfmon counter MSR.
CE0H	3296	MSR_M1_PMON_BOX_CTRL	Package	Uncore M-box 1 perfmon local box control MSR.
CE1H	3297	MSR_M1_PMON_BOX_STATUS	Package	Uncore M-box 1 perfmon local box status MSR.
CE2H	3298	MSR_M1_PMON_BOX_OVF_CTRL	Package	Uncore M-box 1 perfmon local box overflow control MSR.
CE4H	3300	MSR_M1_PMON_TIMESTAMP	Package	Uncore M-box 1 perfmon time stamp unit select MSR.
CE5H	3301	MSR_M1_PMON_DSP	Package	Uncore M-box 1 perfmon DSP unit select MSR.
CE6H	3302	MSR_M1_PMON_ISS	Package	Uncore M-box 1 perfmon ISS unit select MSR.
CE7H	3303	MSR_M1_PMON_MAP	Package	Uncore M-box 1 perfmon MAP unit select MSR.
CE8H	3304	MSR_M1_PMON_MSC_THR	Package	Uncore M-box 1 perfmon MIC THR select MSR.
CE9H	3305	MSR_M1_PMON_PGT	Package	Uncore M-box 1 perfmon PGT unit select MSR.
CEAH	3306	MSR_M1_PMON_PLD	Package	Uncore M-box 1 perfmon PLD unit select MSR.
CEBH	3307	MSR_M1_PMON_ZDP	Package	Uncore M-box 1 perfmon ZDP unit select MSR.
CFOH	3312	MSR_M1_PMON_EVNT_SEL0	Package	Uncore M-box 1 perfmon event select MSR.
CF1H	3313	MSR_M1_PMON_CTRL0	Package	Uncore M-box 1 perfmon counter MSR.
CF2H	3314	MSR_M1_PMON_EVNT_SEL1	Package	Uncore M-box 1 perfmon event select MSR.
CF3H	3315	MSR_M1_PMON_CTRL1	Package	Uncore M-box 1 perfmon counter MSR.
CF4H	3316	MSR_M1_PMON_EVNT_SEL2	Package	Uncore M-box 1 perfmon event select MSR.
CF5H	3317	MSR_M1_PMON_CTRL2	Package	Uncore M-box 1 perfmon counter MSR.
CF6H	3318	MSR_M1_PMON_EVNT_SEL3	Package	Uncore M-box 1 perfmon event select MSR.
CF7H	3319	MSR_M1_PMON_CTRL3	Package	Uncore M-box 1 perfmon counter MSR.

Table 35-15. Additional MSRs in Intel® Xeon® Processor 7500 Series (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
CF8H	3320	MSR_M1_PMON_EVNT_SEL4	Package	Uncore M-box 1 perfmon event select MSR.
CF9H	3321	MSR_M1_PMON_CTR4	Package	Uncore M-box 1 perfmon counter MSR.
CFAH	3322	MSR_M1_PMON_EVNT_SEL5	Package	Uncore M-box 1 perfmon event select MSR.
CFBH	3323	MSR_M1_PMON_CTR5	Package	Uncore M-box 1 perfmon counter MSR.
D00H	3328	MSR_C0_PMON_BOX_CTRL	Package	Uncore C-box 0 perfmon local box control MSR.
D01H	3329	MSR_C0_PMON_BOX_STATUS	Package	Uncore C-box 0 perfmon local box status MSR.
D02H	3330	MSR_C0_PMON_BOX_OVF_CTRL	Package	Uncore C-box 0 perfmon local box overflow control MSR.
D10H	3344	MSR_C0_PMON_EVNT_SELO	Package	Uncore C-box 0 perfmon event select MSR.
D11H	3345	MSR_C0_PMON_CTR0	Package	Uncore C-box 0 perfmon counter MSR.
D12H	3346	MSR_C0_PMON_EVNT_SEL1	Package	Uncore C-box 0 perfmon event select MSR.
D13H	3347	MSR_C0_PMON_CTR1	Package	Uncore C-box 0 perfmon counter MSR.
D14H	3348	MSR_C0_PMON_EVNT_SEL2	Package	Uncore C-box 0 perfmon event select MSR.
D15H	3349	MSR_C0_PMON_CTR2	Package	Uncore C-box 0 perfmon counter MSR.
D16H	3350	MSR_C0_PMON_EVNT_SEL3	Package	Uncore C-box 0 perfmon event select MSR.
D17H	3351	MSR_C0_PMON_CTR3	Package	Uncore C-box 0 perfmon counter MSR.
D18H	3352	MSR_C0_PMON_EVNT_SEL4	Package	Uncore C-box 0 perfmon event select MSR.
D19H	3353	MSR_C0_PMON_CTR4	Package	Uncore C-box 0 perfmon counter MSR.
D1AH	3354	MSR_C0_PMON_EVNT_SEL5	Package	Uncore C-box 0 perfmon event select MSR.
D1BH	3355	MSR_C0_PMON_CTR5	Package	Uncore C-box 0 perfmon counter MSR.
D20H	3360	MSR_C4_PMON_BOX_CTRL	Package	Uncore C-box 4 perfmon local box control MSR.
D21H	3361	MSR_C4_PMON_BOX_STATUS	Package	Uncore C-box 4 perfmon local box status MSR.
D22H	3362	MSR_C4_PMON_BOX_OVF_CTRL	Package	Uncore C-box 4 perfmon local box overflow control MSR.
D30H	3376	MSR_C4_PMON_EVNT_SELO	Package	Uncore C-box 4 perfmon event select MSR.
D31H	3377	MSR_C4_PMON_CTR0	Package	Uncore C-box 4 perfmon counter MSR.

Table 35-15. Additional MSRs in Intel® Xeon® Processor 7500 Series (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
D32H	3378	MSR_C4_PMON_EVNT_SEL1	Package	Uncore C-box 4 perfmon event select MSR.
D33H	3379	MSR_C4_PMON_CTR1	Package	Uncore C-box 4 perfmon counter MSR.
D34H	3380	MSR_C4_PMON_EVNT_SEL2	Package	Uncore C-box 4 perfmon event select MSR.
D35H	3381	MSR_C4_PMON_CTR2	Package	Uncore C-box 4 perfmon counter MSR.
D36H	3382	MSR_C4_PMON_EVNT_SEL3	Package	Uncore C-box 4 perfmon event select MSR.
D37H	3383	MSR_C4_PMON_CTR3	Package	Uncore C-box 4 perfmon counter MSR.
D38H	3384	MSR_C4_PMON_EVNT_SEL4	Package	Uncore C-box 4 perfmon event select MSR.
D39H	3385	MSR_C4_PMON_CTR4	Package	Uncore C-box 4 perfmon counter MSR.
D3AH	3386	MSR_C4_PMON_EVNT_SEL5	Package	Uncore C-box 4 perfmon event select MSR.
D3BH	3387	MSR_C4_PMON_CTR5	Package	Uncore C-box 4 perfmon counter MSR.
D40H	3392	MSR_C2_PMON_BOX_CTRL	Package	Uncore C-box 2 perfmon local box control MSR.
D41H	3393	MSR_C2_PMON_BOX_STATUS	Package	Uncore C-box 2 perfmon local box status MSR.
D42H	3394	MSR_C2_PMON_BOX_OVF_CTRL	Package	Uncore C-box 2 perfmon local box overflow control MSR.
D50H	3408	MSR_C2_PMON_EVNT_SELO	Package	Uncore C-box 2 perfmon event select MSR.
D51H	3409	MSR_C2_PMON_CTR0	Package	Uncore C-box 2 perfmon counter MSR.
D52H	3410	MSR_C2_PMON_EVNT_SEL1	Package	Uncore C-box 2 perfmon event select MSR.
D53H	3411	MSR_C2_PMON_CTR1	Package	Uncore C-box 2 perfmon counter MSR.
D54H	3412	MSR_C2_PMON_EVNT_SEL2	Package	Uncore C-box 2 perfmon event select MSR.
D55H	3413	MSR_C2_PMON_CTR2	Package	Uncore C-box 2 perfmon counter MSR.
D56H	3414	MSR_C2_PMON_EVNT_SEL3	Package	Uncore C-box 2 perfmon event select MSR.
D57H	3415	MSR_C2_PMON_CTR3	Package	Uncore C-box 2 perfmon counter MSR.
D58H	3416	MSR_C2_PMON_EVNT_SEL4	Package	Uncore C-box 2 perfmon event select MSR.
D59H	3417	MSR_C2_PMON_CTR4	Package	Uncore C-box 2 perfmon counter MSR.
D5AH	3418	MSR_C2_PMON_EVNT_SEL5	Package	Uncore C-box 2 perfmon event select MSR.
D5BH	3419	MSR_C2_PMON_CTR5	Package	Uncore C-box 2 perfmon counter MSR.



Table 35-15. Additional MSRs in Intel® Xeon® Processor 7500 Series (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
D60H	3424	MSR_C6_PMON_BOX_CTRL	Package	Uncore C-box 6 perfmon local box control MSR.
D61H	3425	MSR_C6_PMON_BOX_STATUS	Package	Uncore C-box 6 perfmon local box status MSR.
D62H	3426	MSR_C6_PMON_BOX_OVF_CTRL	Package	Uncore C-box 6 perfmon local box overflow control MSR.
D70H	3440	MSR_C6_PMON_EVNT_SELO	Package	Uncore C-box 6 perfmon event select MSR.
D71H	3441	MSR_C6_PMON_CTR0	Package	Uncore C-box 6 perfmon counter MSR.
D72H	3442	MSR_C6_PMON_EVNT_SEL1	Package	Uncore C-box 6 perfmon event select MSR.
D73H	3443	MSR_C6_PMON_CTR1	Package	Uncore C-box 6 perfmon counter MSR.
D74H	3444	MSR_C6_PMON_EVNT_SEL2	Package	Uncore C-box 6 perfmon event select MSR.
D75H	3445	MSR_C6_PMON_CTR2	Package	Uncore C-box 6 perfmon counter MSR.
D76H	3446	MSR_C6_PMON_EVNT_SEL3	Package	Uncore C-box 6 perfmon event select MSR.
D77H	3447	MSR_C6_PMON_CTR3	Package	Uncore C-box 6 perfmon counter MSR.
D78H	3448	MSR_C6_PMON_EVNT_SEL4	Package	Uncore C-box 6 perfmon event select MSR.
D79H	3449	MSR_C6_PMON_CTR4	Package	Uncore C-box 6 perfmon counter MSR.
D7AH	3450	MSR_C6_PMON_EVNT_SEL5	Package	Uncore C-box 6 perfmon event select MSR.
D7BH	3451	MSR_C6_PMON_CTR5	Package	Uncore C-box 6 perfmon counter MSR.
D80H	3456	MSR_C1_PMON_BOX_CTRL	Package	Uncore C-box 1 perfmon local box control MSR.
D81H	3457	MSR_C1_PMON_BOX_STATUS	Package	Uncore C-box 1 perfmon local box status MSR.
D82H	3458	MSR_C1_PMON_BOX_OVF_CTRL	Package	Uncore C-box 1 perfmon local box overflow control MSR.
D90H	3472	MSR_C1_PMON_EVNT_SELO	Package	Uncore C-box 1 perfmon event select MSR.
D91H	3473	MSR_C1_PMON_CTR0	Package	Uncore C-box 1 perfmon counter MSR.
D92H	3474	MSR_C1_PMON_EVNT_SEL1	Package	Uncore C-box 1 perfmon event select MSR.
D93H	3475	MSR_C1_PMON_CTR1	Package	Uncore C-box 1 perfmon counter MSR.
D94H	3476	MSR_C1_PMON_EVNT_SEL2	Package	Uncore C-box 1 perfmon event select MSR.
D95H	3477	MSR_C1_PMON_CTR2	Package	Uncore C-box 1 perfmon counter MSR.

Table 35-15. Additional MSRs in Intel® Xeon® Processor 7500 Series (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
D96H	3478	MSR_C1_PMON_EVNT_SEL3	Package	Uncore C-box 1 perfmon event select MSR.
D97H	3479	MSR_C1_PMON_CTR3	Package	Uncore C-box 1 perfmon counter MSR.
D98H	3480	MSR_C1_PMON_EVNT_SEL4	Package	Uncore C-box 1 perfmon event select MSR.
D99H	3481	MSR_C1_PMON_CTR4	Package	Uncore C-box 1 perfmon counter MSR.
D9AH	3482	MSR_C1_PMON_EVNT_SEL5	Package	Uncore C-box 1 perfmon event select MSR.
D9BH	3483	MSR_C1_PMON_CTR5	Package	Uncore C-box 1 perfmon counter MSR.
DA0H	3488	MSR_C5_PMON_BOX_CTRL	Package	Uncore C-box 5 perfmon local box control MSR.
DA1H	3489	MSR_C5_PMON_BOX_STATUS	Package	Uncore C-box 5 perfmon local box status MSR.
DA2H	3490	MSR_C5_PMON_BOX_OVF_CTRL	Package	Uncore C-box 5 perfmon local box overflow control MSR.
DB0H	3504	MSR_C5_PMON_EVNT_SELO	Package	Uncore C-box 5 perfmon event select MSR.
DB1H	3505	MSR_C5_PMON_CTR0	Package	Uncore C-box 5 perfmon counter MSR.
DB2H	3506	MSR_C5_PMON_EVNT_SEL1	Package	Uncore C-box 5 perfmon event select MSR.
DB3H	3507	MSR_C5_PMON_CTR1	Package	Uncore C-box 5 perfmon counter MSR.
DB4H	3508	MSR_C5_PMON_EVNT_SEL2	Package	Uncore C-box 5 perfmon event select MSR.
DB5H	3509	MSR_C5_PMON_CTR2	Package	Uncore C-box 5 perfmon counter MSR.
DB6H	3510	MSR_C5_PMON_EVNT_SEL3	Package	Uncore C-box 5 perfmon event select MSR.
DB7H	3511	MSR_C5_PMON_CTR3	Package	Uncore C-box 5 perfmon counter MSR.
DB8H	3512	MSR_C5_PMON_EVNT_SEL4	Package	Uncore C-box 5 perfmon event select MSR.
DB9H	3513	MSR_C5_PMON_CTR4	Package	Uncore C-box 5 perfmon counter MSR.
DBAH	3514	MSR_C5_PMON_EVNT_SEL5	Package	Uncore C-box 5 perfmon event select MSR.
DBBH	3515	MSR_C5_PMON_CTR5	Package	Uncore C-box 5 perfmon counter MSR.
DC0H	3520	MSR_C3_PMON_BOX_CTRL	Package	Uncore C-box 3 perfmon local box control MSR.
DC1H	3521	MSR_C3_PMON_BOX_STATUS	Package	Uncore C-box 3 perfmon local box status MSR.
DC2H	3522	MSR_C3_PMON_BOX_OVF_CTRL	Package	Uncore C-box 3 perfmon local box overflow control MSR.

Table 35-15. Additional MSRs in Intel® Xeon® Processor 7500 Series (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
DD0H	3536	MSR_C3_PMON_EVNT_SELO	Package	Uncore C-box 3 perfmon event select MSR.
DD1H	3537	MSR_C3_PMON_CTRL0	Package	Uncore C-box 3 perfmon counter MSR.
DD2H	3538	MSR_C3_PMON_EVNT_SEL1	Package	Uncore C-box 3 perfmon event select MSR.
DD3H	3539	MSR_C3_PMON_CTRL1	Package	Uncore C-box 3 perfmon counter MSR.
DD4H	3540	MSR_C3_PMON_EVNT_SEL2	Package	Uncore C-box 3 perfmon event select MSR.
DD5H	3541	MSR_C3_PMON_CTRL2	Package	Uncore C-box 3 perfmon counter MSR.
DD6H	3542	MSR_C3_PMON_EVNT_SEL3	Package	Uncore C-box 3 perfmon event select MSR.
DD7H	3543	MSR_C3_PMON_CTRL3	Package	Uncore C-box 3 perfmon counter MSR.
DD8H	3544	MSR_C3_PMON_EVNT_SEL4	Package	Uncore C-box 3 perfmon event select MSR.
DD9H	3545	MSR_C3_PMON_CTRL4	Package	Uncore C-box 3 perfmon counter MSR.
DDAH	3546	MSR_C3_PMON_EVNT_SEL5	Package	Uncore C-box 3 perfmon event select MSR.
DDBH	3547	MSR_C3_PMON_CTRL5	Package	Uncore C-box 3 perfmon counter MSR.
DE0H	3552	MSR_C7_PMON_BOX_CTRL	Package	Uncore C-box 7 perfmon local box control MSR.
DE1H	3553	MSR_C7_PMON_BOX_STATUS	Package	Uncore C-box 7 perfmon local box status MSR.
DE2H	3554	MSR_C7_PMON_BOX_OVF_CTRL	Package	Uncore C-box 7 perfmon local box overflow control MSR.
DF0H	3568	MSR_C7_PMON_EVNT_SELO	Package	Uncore C-box 7 perfmon event select MSR.
DF1H	3569	MSR_C7_PMON_CTRL0	Package	Uncore C-box 7 perfmon counter MSR.
DF2H	3570	MSR_C7_PMON_EVNT_SEL1	Package	Uncore C-box 7 perfmon event select MSR.
DF3H	3571	MSR_C7_PMON_CTRL1	Package	Uncore C-box 7 perfmon counter MSR.
DF4H	3572	MSR_C7_PMON_EVNT_SEL2	Package	Uncore C-box 7 perfmon event select MSR.
DF5H	3573	MSR_C7_PMON_CTRL2	Package	Uncore C-box 7 perfmon counter MSR.
DF6H	3574	MSR_C7_PMON_EVNT_SEL3	Package	Uncore C-box 7 perfmon event select MSR.
DF7H	3575	MSR_C7_PMON_CTRL3	Package	Uncore C-box 7 perfmon counter MSR.
DF8H	3576	MSR_C7_PMON_EVNT_SEL4	Package	Uncore C-box 7 perfmon event select MSR.
DF9H	3577	MSR_C7_PMON_CTRL4	Package	Uncore C-box 7 perfmon counter MSR.

Table 35-15. Additional MSRs in Intel® Xeon® Processor 7500 Series (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
DFAH	3578	MSR_C7_PMON_EVNT_SEL5	Package	Uncore C-box 7 perfmon event select MSR.
DFBH	3579	MSR_C7_PMON_CTR5	Package	Uncore C-box 7 perfmon counter MSR.
E00H	3584	MSR_R0_PMON_BOX_CTRL	Package	Uncore R-box 0 perfmon local box control MSR.
E01H	3585	MSR_R0_PMON_BOX_STATUS	Package	Uncore R-box 0 perfmon local box status MSR.
E02H	3586	MSR_R0_PMON_BOX_OVF_CTRL	Package	Uncore R-box 0 perfmon local box overflow control MSR.
E04H	3588	MSR_R0_PMON_IPERFO_P0	Package	Uncore R-box 0 perfmon IPERFO unit Port 0 select MSR.
E05H	3589	MSR_R0_PMON_IPERFO_P1	Package	Uncore R-box 0 perfmon IPERFO unit Port 1 select MSR.
E06H	3590	MSR_R0_PMON_IPERFO_P2	Package	Uncore R-box 0 perfmon IPERFO unit Port 2 select MSR.
E07H	3591	MSR_R0_PMON_IPERFO_P3	Package	Uncore R-box 0 perfmon IPERFO unit Port 3 select MSR.
E08H	3592	MSR_R0_PMON_IPERFO_P4	Package	Uncore R-box 0 perfmon IPERFO unit Port 4 select MSR.
E09H	3593	MSR_R0_PMON_IPERFO_P5	Package	Uncore R-box 0 perfmon IPERFO unit Port 5 select MSR.
E0AH	3594	MSR_R0_PMON_IPERFO_P6	Package	Uncore R-box 0 perfmon IPERFO unit Port 6 select MSR.
E0BH	3595	MSR_R0_PMON_IPERFO_P7	Package	Uncore R-box 0 perfmon IPERFO unit Port 7 select MSR.
E0CH	3596	MSR_R0_PMON_QLX_P0	Package	Uncore R-box 0 perfmon QLX unit Port 0 select MSR.
E0DH	3597	MSR_R0_PMON_QLX_P1	Package	Uncore R-box 0 perfmon QLX unit Port 1 select MSR.
E0EH	3598	MSR_R0_PMON_QLX_P2	Package	Uncore R-box 0 perfmon QLX unit Port 2 select MSR.
E0FH	3599	MSR_R0_PMON_QLX_P3	Package	Uncore R-box 0 perfmon QLX unit Port 3 select MSR.
E10H	3600	MSR_R0_PMON_EVNT_SEL0	Package	Uncore R-box 0 perfmon event select MSR.
E11H	3601	MSR_R0_PMON_CTR0	Package	Uncore R-box 0 perfmon counter MSR.
E12H	3602	MSR_R0_PMON_EVNT_SEL1	Package	Uncore R-box 0 perfmon event select MSR.
E13H	3603	MSR_R0_PMON_CTR1	Package	Uncore R-box 0 perfmon counter MSR.
E14H	3604	MSR_R0_PMON_EVNT_SEL2	Package	Uncore R-box 0 perfmon event select MSR.
E15H	3605	MSR_R0_PMON_CTR2	Package	Uncore R-box 0 perfmon counter MSR.
E16H	3606	MSR_R0_PMON_EVNT_SEL3	Package	Uncore R-box 0 perfmon event select MSR.
E17H	3607	MSR_R0_PMON_CTR3	Package	Uncore R-box 0 perfmon counter MSR.
E18H	3608	MSR_R0_PMON_EVNT_SEL4	Package	Uncore R-box 0 perfmon event select MSR.
E19H	3609	MSR_R0_PMON_CTR4	Package	Uncore R-box 0 perfmon counter MSR.
E1AH	3610	MSR_R0_PMON_EVNT_SEL5	Package	Uncore R-box 0 perfmon event select MSR.
E1BH	3611	MSR_R0_PMON_CTR5	Package	Uncore R-box 0 perfmon counter MSR.

Table 35-15. Additional MSRs in Intel® Xeon® Processor 7500 Series (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
E1CH	3612	MSR_R0_PMON_EVNT_SEL6	Package	Uncore R-box 0 perfmon event select MSR.
E1DH	3613	MSR_R0_PMON_CTR6	Package	Uncore R-box 0 perfmon counter MSR.
E1EH	3614	MSR_R0_PMON_EVNT_SEL7	Package	Uncore R-box 0 perfmon event select MSR.
E1FH	3615	MSR_R0_PMON_CTR7	Package	Uncore R-box 0 perfmon counter MSR.
E20H	3616	MSR_R1_PMON_BOX_CTRL	Package	Uncore R-box 1 perfmon local box control MSR.
E21H	3617	MSR_R1_PMON_BOX_STATUS	Package	Uncore R-box 1 perfmon local box status MSR.
E22H	3618	MSR_R1_PMON_BOX_OVF_CTRL	Package	Uncore R-box 1 perfmon local box overflow control MSR.
E24H	3620	MSR_R1_PMON_IPERF1_P8	Package	Uncore R-box 1 perfmon IPERF1 unit Port 8 select MSR.
E25H	3621	MSR_R1_PMON_IPERF1_P9	Package	Uncore R-box 1 perfmon IPERF1 unit Port 9 select MSR.
E26H	3622	MSR_R1_PMON_IPERF1_P10	Package	Uncore R-box 1 perfmon IPERF1 unit Port 10 select MSR.
E27H	3623	MSR_R1_PMON_IPERF1_P11	Package	Uncore R-box 1 perfmon IPERF1 unit Port 11 select MSR.
E28H	3624	MSR_R1_PMON_IPERF1_P12	Package	Uncore R-box 1 perfmon IPERF1 unit Port 12 select MSR.
E29H	3625	MSR_R1_PMON_IPERF1_P13	Package	Uncore R-box 1 perfmon IPERF1 unit Port 13 select MSR.
E2AH	3626	MSR_R1_PMON_IPERF1_P14	Package	Uncore R-box 1 perfmon IPERF1 unit Port 14 select MSR.
E2BH	3627	MSR_R1_PMON_IPERF1_P15	Package	Uncore R-box 1 perfmon IPERF1 unit Port 15 select MSR.
E2CH	3628	MSR_R1_PMON_QLX_P4	Package	Uncore R-box 1 perfmon QLX unit Port 4 select MSR.
E2DH	3629	MSR_R1_PMON_QLX_P5	Package	Uncore R-box 1 perfmon QLX unit Port 5 select MSR.
E2EH	3630	MSR_R1_PMON_QLX_P6	Package	Uncore R-box 1 perfmon QLX unit Port 6 select MSR.
E2FH	3631	MSR_R1_PMON_QLX_P7	Package	Uncore R-box 1 perfmon QLX unit Port 7 select MSR.
E30H	3632	MSR_R1_PMON_EVNT_SEL8	Package	Uncore R-box 1 perfmon event select MSR.
E31H	3633	MSR_R1_PMON_CTR8	Package	Uncore R-box 1 perfmon counter MSR.
E32H	3634	MSR_R1_PMON_EVNT_SEL9	Package	Uncore R-box 1 perfmon event select MSR.
E33H	3635	MSR_R1_PMON_CTR9	Package	Uncore R-box 1 perfmon counter MSR.
E34H	3636	MSR_R1_PMON_EVNT_SEL10	Package	Uncore R-box 1 perfmon event select MSR.
E35H	3637	MSR_R1_PMON_CTR10	Package	Uncore R-box 1 perfmon counter MSR.
E36H	3638	MSR_R1_PMON_EVNT_SEL11	Package	Uncore R-box 1 perfmon event select MSR.

Table 35-15. Additional MSRs in Intel® Xeon® Processor 7500 Series (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
E37H	3639	MSR_R1_PMON_CTR11	Package	Uncore R-box 1 perfmon counter MSR.
E38H	3640	MSR_R1_PMON_EVNT_SEL12	Package	Uncore R-box 1 perfmon event select MSR.
E39H	3641	MSR_R1_PMON_CTR12	Package	Uncore R-box 1 perfmon counter MSR.
E3AH	3642	MSR_R1_PMON_EVNT_SEL13	Package	Uncore R-box 1 perfmon event select MSR.
E3BH	3643	MSR_R1_PMON_CTR13	Package	Uncore R-box 1 perfmon counter MSR.
E3CH	3644	MSR_R1_PMON_EVNT_SEL14	Package	Uncore R-box 1 perfmon event select MSR.
E3DH	3645	MSR_R1_PMON_CTR14	Package	Uncore R-box 1 perfmon counter MSR.
E3EH	3646	MSR_R1_PMON_EVNT_SEL15	Package	Uncore R-box 1 perfmon event select MSR.
E3FH	3647	MSR_R1_PMON_CTR15	Package	Uncore R-box 1 perfmon counter MSR.
E45H	3653	MSR_B0_PMON_MATCH	Package	Uncore B-box 0 perfmon local box match MSR.
E46H	3654	MSR_B0_PMON_MASK	Package	Uncore B-box 0 perfmon local box mask MSR.
E49H	3657	MSR_S0_PMON_MATCH	Package	Uncore S-box 0 perfmon local box match MSR.
E4AH	3658	MSR_S0_PMON_MASK	Package	Uncore S-box 0 perfmon local box mask MSR.
E4DH	3661	MSR_B1_PMON_MATCH	Package	Uncore B-box 1 perfmon local box match MSR.
E4EH	3662	MSR_B1_PMON_MASK	Package	Uncore B-box 1 perfmon local box mask MSR.
E54H	3668	MSR_M0_PMON_MM_CONFIG	Package	Uncore M-box 0 perfmon local box address match/mask config MSR.
E55H	3669	MSR_M0_PMON_ADDR_MATCH	Package	Uncore M-box 0 perfmon local box address match MSR.
E56H	3670	MSR_M0_PMON_ADDR_MASK	Package	Uncore M-box 0 perfmon local box address mask MSR.
E59H	3673	MSR_S1_PMON_MATCH	Package	Uncore S-box 1 perfmon local box match MSR.
E5AH	3674	MSR_S1_PMON_MASK	Package	Uncore S-box 1 perfmon local box mask MSR.
E5CH	3676	MSR_M1_PMON_MM_CONFIG	Package	Uncore M-box 1 perfmon local box address match/mask config MSR.
E5DH	3677	MSR_M1_PMON_ADDR_MATCH	Package	Uncore M-box 1 perfmon local box address match MSR.
E5EH	3678	MSR_M1_PMON_ADDR_MASK	Package	Uncore M-box 1 perfmon local box address mask MSR.
3B5H	965	MSR_UNCORE_PMC5	Package	See Section 18.7.2.2, "Uncore Performance Event Configuration Facility."

### 35.7 MSRS IN THE INTEL® XEON® PROCESSOR 5600 SERIES (BASED ON INTEL® MICROARCHITECTURE CODE NAME WESTMERE)

Intel® Xeon® Processor 5600 Series (based on Intel® microarchitecture code name Westmere) supports the MSR interfaces listed in Table 35-13, Table 35-14, plus additional MSR listed in Table 35-16. These MSRs apply to Intel Core i7, i5 and i3 processor family with CPUID signature DisplayFamily\_DisplayModel of 06\_25H and 06\_2CH, see Table 35-1.

**Table 35-16. Additional MSRs Supported by Intel Processors (Based on Intel® Microarchitecture Code Name Westmere)**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
13CH	52	MSR_FEATURE_CONFIG	Core	<b>AES Configuration (RW-L)</b> Privileged post-BIOS agent must provide a #GP handler to handle unsuccessful read of this MSR.
		1:0		<b>AES Configuration (RW-L)</b> Upon a successful read of this MSR, the configuration of AES instruction set availability is as follows: 11b: AES instructions are not available until next RESET. otherwise, AES instructions are available. Note, AES instruction set is not available if read is unsuccessful. If the configuration is not 01b, AES instruction can be mis-configured if a privileged agent unintentionally writes 11b.
		63:2		Reserved.
1A7H	423	MSR_OFFCORE_RSP_1	Thread	<b>Offcore Response Event Select Register (R/W)</b>
1ADH	429	MSR_TURBO_RATIO_LIMIT	Package	<b>Maximum Ratio Limit of Turbo Mode</b> RO if MSR_PLATFORM_INFO.[28] = 0, RW if MSR_PLATFORM_INFO.[28] = 1
		7:0	Package	<b>Maximum Ratio Limit for 1C</b> Maximum turbo ratio limit of 1 core active.
		15:8	Package	<b>Maximum Ratio Limit for 2C</b> Maximum turbo ratio limit of 2 core active.
		23:16	Package	<b>Maximum Ratio Limit for 3C</b> Maximum turbo ratio limit of 3 core active.
		31:24	Package	<b>Maximum Ratio Limit for 4C</b> Maximum turbo ratio limit of 4 core active.
		39:32	Package	<b>Maximum Ratio Limit for 5C</b> Maximum turbo ratio limit of 5 core active.
		47:40	Package	<b>Maximum Ratio Limit for 6C</b> Maximum turbo ratio limit of 6 core active.
63:48		Reserved.		
1B0H	432	IA32_ENERGY_PERF_BIAS	Package	See Table 35-2.

## 35.8 MSRS IN THE INTEL® XEON® PROCESSOR E7 FAMILY (BASED ON INTEL® MICROARCHITECTURE CODE NAME WESTMERE)

Intel® Xeon® Processor E7 Family (based on Intel® microarchitecture code name Westmere) supports the MSR interfaces listed in Table 35-13 (except MSR address 1ADH), Table 35-14, plus additional MSR listed in Table 35-17. These processors have a CPUID signature with DisplayFamily\_DisplayModel of 06\_2FH.

**Table 35-17. Additional MSRs Supported by Intel® Xeon® Processor E7 Family**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
13CH	52	MSR_FEATURE_CONFIG	Core	<b>AES Configuration (RW-L)</b> Privileged post-BIOS agent must provide a #GP handler to handle unsuccessful read of this MSR.
		1:0		<b>AES Configuration (RW-L)</b> Upon a successful read of this MSR, the configuration of AES instruction set availability is as follows: 11b: AES instructions are not available until next RESET. otherwise, AES instructions are available. Note, AES instruction set is not available if read is unsuccessful. If the configuration is not 01b, AES instruction can be mis-configured if a privileged agent unintentionally writes 11b.
		63:2		Reserved.
1A7H	423	MSR_OFFCORE_RSP_1	Thread	<b>Offcore Response Event Select Register (R/W)</b>
1ADH	429	MSR_TURBO_RATIO_LIMIT	Package	<b>Reserved</b> Attempt to read/write will cause #UD.
1B0H	432	IA32_ENERGY_PERF_BIAS	Package	See Table 35-2.
F40H	3904	MSR_C8_PMON_BOX_CTRL	Package	Uncore C-box 8 perfmon local box control MSR.
F41H	3905	MSR_C8_PMON_BOX_STATUS	Package	Uncore C-box 8 perfmon local box status MSR.
F42H	3906	MSR_C8_PMON_BOX_OVF_CTRL	Package	Uncore C-box 8 perfmon local box overflow control MSR.
F50H	3920	MSR_C8_PMON_EVNT_SELO	Package	Uncore C-box 8 perfmon event select MSR.
F51H	3921	MSR_C8_PMON_CTR0	Package	Uncore C-box 8 perfmon counter MSR.
F52H	3922	MSR_C8_PMON_EVNT_SEL1	Package	Uncore C-box 8 perfmon event select MSR.
F53H	3923	MSR_C8_PMON_CTR1	Package	Uncore C-box 8 perfmon counter MSR.
F54H	3924	MSR_C8_PMON_EVNT_SEL2	Package	Uncore C-box 8 perfmon event select MSR.
F55H	3925	MSR_C8_PMON_CTR2	Package	Uncore C-box 8 perfmon counter MSR.
F56H	3926	MSR_C8_PMON_EVNT_SEL3	Package	Uncore C-box 8 perfmon event select MSR.
F57H	3927	MSR_C8_PMON_CTR3	Package	Uncore C-box 8 perfmon counter MSR.



**Table 35-17. Additional MSRs Supported by Intel® Xeon® Processor E7 Family (Contd.)**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
F58H	3928	MSR_C8_PMON_EVNT_SEL4	Package	Uncore C-box 8 perfmon event select MSR.
F59H	3929	MSR_C8_PMON_CTR4	Package	Uncore C-box 8 perfmon counter MSR.
F5AH	3930	MSR_C8_PMON_EVNT_SEL5	Package	Uncore C-box 8 perfmon event select MSR.
F5BH	3931	MSR_C8_PMON_CTR5	Package	Uncore C-box 8 perfmon counter MSR.
FC0H	4032	MSR_C9_PMON_BOX_CTRL	Package	Uncore C-box 9 perfmon local box control MSR.
FC1H	4033	MSR_C9_PMON_BOX_STATUS	Package	Uncore C-box 9 perfmon local box status MSR.
FC2H	4034	MSR_C9_PMON_BOX_OVF_CTRL	Package	Uncore C-box 9 perfmon local box overflow control MSR.
FD0H	4048	MSR_C9_PMON_EVNT_SELO	Package	Uncore C-box 9 perfmon event select MSR.
FD1H	4049	MSR_C9_PMON_CTR0	Package	Uncore C-box 9 perfmon counter MSR.
FD2H	4050	MSR_C9_PMON_EVNT_SEL1	Package	Uncore C-box 9 perfmon event select MSR.
FD3H	4051	MSR_C9_PMON_CTR1	Package	Uncore C-box 9 perfmon counter MSR.
FD4H	4052	MSR_C9_PMON_EVNT_SEL2	Package	Uncore C-box 9 perfmon event select MSR.
FD5H	4053	MSR_C9_PMON_CTR2	Package	Uncore C-box 9 perfmon counter MSR.
FD6H	4054	MSR_C9_PMON_EVNT_SEL3	Package	Uncore C-box 9 perfmon event select MSR.
FD7H	4055	MSR_C9_PMON_CTR3	Package	Uncore C-box 9 perfmon counter MSR.
FD8H	4056	MSR_C9_PMON_EVNT_SEL4	Package	Uncore C-box 9 perfmon event select MSR.
FD9H	4057	MSR_C9_PMON_CTR4	Package	Uncore C-box 9 perfmon counter MSR.
FDAH	4058	MSR_C9_PMON_EVNT_SEL5	Package	Uncore C-box 9 perfmon event select MSR.
FDBH	4059	MSR_C9_PMON_CTR5	Package	Uncore C-box 9 perfmon counter MSR.

## 35.9 MSRS IN INTEL® PROCESSOR FAMILY BASED ON INTEL® MICROARCHITECTURE CODE NAME SANDY BRIDGE

Table 35-18 lists model-specific registers (MSRs) that are common to Intel® processor family based on Intel micro-architecture code name Sandy Bridge. These processors have a CPUID signature with DisplayFamily\_DisplayModel of 06\_2AH, 06\_2DH, see Table 35-1. Additional MSRs specific to 06\_2AH are listed in Table 35-19.

**Table 35-18. MSRs Supported by Intel® Processors  
based on Intel® microarchitecture code name Sandy Bridge**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
0H	0	IA32_P5_MC_ADDR	Thread	See Section 35.22, "MSRs in Pentium Processors."
1H	1	IA32_P5_MC_TYPE	Thread	See Section 35.22, "MSRs in Pentium Processors."
6H	6	IA32_MONITOR_FILTER_SIZE	Thread	See Section 8.10.5, "Monitor/Mwait Address Range Determination," and Table 35-2.
10H	16	IA32_TIME_STAMP_COUNTER	Thread	See Section 17.15, "Time-Stamp Counter," and see Table 35-2.
17H	23	IA32_PLATFORM_ID	Package	<b>Platform ID (R)</b> See Table 35-2.
1BH	27	IA32_APIC_BASE	Thread	See Section 10.4.4, "Local APIC Status and Location," and Table 35-2.
34H	52	MSR_SMI_COUNT	Thread	<b>SMI Counter (R/O)</b>
		31:0		<b>SMI Count (R/O)</b> Count SMIs.
		63:32		Reserved.
3AH	58	IA32_FEATURE_CONTROL	Thread	<b>Control Features in Intel 64 Processor (R/W)</b> See Table 35-2.
		0		<b>Lock (R/WL)</b>
		1		<b>Enable VMX inside SMX operation (R/WL)</b>
		2		<b>Enable VMX outside SMX operation (R/WL)</b>
		14:8		<b>SENTER local functions enables (R/WL)</b>
15		<b>SENTER global functions enable (R/WL)</b>		
79H	121	IA32_BIOS_UPDT_TRIG	Core	<b>BIOS Update Trigger Register (W)</b> See Table 35-2.
8BH	139	IA32_BIOS_SIGN_ID	Thread	<b>BIOS Update Signature ID (RO)</b> See Table 35-2.
C1H	193	IA32_PMC0	Thread	<b>Performance Counter Register</b> See Table 35-2.
C2H	194	IA32_PMC1	Thread	<b>Performance Counter Register</b> See Table 35-2.
C3H	195	IA32_PMC2	Thread	<b>Performance Counter Register</b> See Table 35-2.
C4H	196	IA32_PMC3	Thread	<b>Performance Counter Register</b> See Table 35-2.
C5H	197	IA32_PMC4	Core	<b>Performance Counter Register (if core not shared by threads)</b>
C6H	198	IA32_PMC5	Core	<b>Performance Counter Register (if core not shared by threads)</b>
C7H	199	IA32_PMC6	Core	<b>Performance Counter Register (if core not shared by threads)</b>
C8H	200	IA32_PMC7	Core	<b>Performance Counter Register (if core not shared by threads)</b>

**Table 35-18. MSRs Supported by Intel® Processors  
based on Intel® microarchitecture code name Sandy Bridge (Contd.)**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
CEH	206	MSR_PLATFORM_INFO	Package	See <a href="http://biosbits.org">http://biosbits.org</a> .
		7:0		Reserved.
		15:8	Package	<b>Maximum Non-Turbo Ratio (R/O)</b> The is the ratio of the frequency that invariant TSC runs at. Frequency = ratio * 100 MHz.
		27:16		Reserved.
		28	Package	<b>Programmable Ratio Limit for Turbo Mode (R/O)</b> When set to 1, indicates that Programmable Ratio Limits for Turbo mode is enabled, and when set to 0, indicates Programmable Ratio Limits for Turbo mode is disabled.
		29	Package	<b>Programmable TDP Limit for Turbo Mode (R/O)</b> When set to 1, indicates that TDP Limits for Turbo mode are programmable, and when set to 0, indicates TDP Limit for Turbo mode is not programmable.
		39:30		Reserved.
		47:40	Package	<b>Maximum Efficiency Ratio (R/O)</b> The is the minimum ratio (maximum efficiency) that the processor can operate, in units of 100MHz.
		63:48		Reserved.
E2H	226	MSR_PKG_CST_CONFIG_CONTROL	Core	<b>C-State Configuration Control (R/W)</b> Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States. See <a href="http://biosbits.org">http://biosbits.org</a> .
		2:0		<b>Package C-State Limit (R/W)</b> Specifies the lowest processor-specific C-state code name (consuming the least power), for the package. The default is set as factory-configured package C-state limit. The following C-state code name encodings are supported: 000b: C0/C1 (no package C-sate support) 001b: C2 010b: C6 no retention 011b: C6 retention 100b: C7 101b: C7s 111: No package C-state limit. Note: This field cannot be used to limit package C-state to C3.
		9:3		Reserved.
		10		<b>I/O MWAIT Redirection Enable (R/W)</b> When set, will map IO_read instructions sent to IO register specified by MSR_PMG_IO_CAPTURE_BASE to MWAIT instructions
		14:11		Reserved.

**Table 35-18. MSRs Supported by Intel® Processors  
based on Intel® microarchitecture code name Sandy Bridge (Contd.)**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		15		<b>CFG Lock (R/WO)</b> When set, lock bits 15:0 of this register until next reset.
		24:16		Reserved.
		25		<b>C3 state auto demotion enable (R/W)</b> When set, the processor will conditionally demote C6/C7 requests to C3 based on uncore auto-demote information.
		26		<b>C1 state auto demotion enable (R/W)</b> When set, the processor will conditionally demote C3/C6/C7 requests to C1 based on uncore auto-demote information.
		27		<b>Enable C3 undemotion (R/W)</b> When set, enables undemotion from demoted C3.
		28		<b>Enable C1 undemotion (R/W)</b> When set, enables undemotion from demoted C1.
		63:29		Reserved.
E4H	228	MSR_PMG_IO_CAPTURE_BASE	Core	<b>Power Management IO Redirection in C-state (R/W)</b> See <a href="http://biosbits.org">http://biosbits.org</a> .
		15:0		<b>LVL_2 Base Address (R/W)</b> Specifies the base address visible to software for IO redirection. If IO MWAIT Redirection is enabled, reads to this address will be consumed by the power management logic and decoded to MWAIT instructions. When IO port address redirection is enabled, this is the IO port address reported to the OS/software.
		18:16		<b>C-state Range (R/W)</b> Specifies the encoding value of the maximum C-State code name to be included when IO read to MWAIT redirection is enabled by MSR_PKG_CST_CONFIG_CONTROL[bit10]: 000b - C3 is the max C-State to include 001b - C6 is the max C-State to include 010b - C7 is the max C-State to include
		63:19		Reserved.
E7H	231	IA32_MPERF	Thread	<b>Maximum Performance Frequency Clock Count (RW)</b> See Table 35-2.
E8H	232	IA32_APERF	Thread	<b>Actual Performance Frequency Clock Count (RW)</b> See Table 35-2.
FEH	254	IA32_MTRRCAP	Thread	See Table 35-2.
13CH	52	MSR_FEATURE_CONFIG	Core	<b>AES Configuration (RW-L)</b> Privileged post-BIOS agent must provide a #GP handler to handle unsuccessful read of this MSR.

**Table 35-18. MSRs Supported by Intel® Processors  
based on Intel® microarchitecture code name Sandy Bridge (Contd.)**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		1:0		<b>AES Configuration (RW-L)</b> Upon a successful read of this MSR, the configuration of AES instruction set availability is as follows: 11b: AES instructions are not available until next RESET. otherwise, AES instructions are available. Note, AES instruction set is not available if read is unsuccessful. If the configuration is not 01b, AES instruction can be mis-configured if a privileged agent unintentionally writes 11b.
		63:2		Reserved.
174H	372	IA32_SYSENTER_CS	Thread	See Table 35-2.
175H	373	IA32_SYSENTER_ESP	Thread	See Table 35-2.
176H	374	IA32_SYSENTER_EIP	Thread	See Table 35-2.
179H	377	IA32_MCG_CAP	Thread	See Table 35-2.
17AH	378	IA32_MCG_STATUS	Thread	
		0		<b>RIPV</b> When set, bit indicates that the instruction addressed by the instruction pointer pushed on the stack (when the machine check was generated) can be used to restart the program. If cleared, the program cannot be reliably restarted.
		1		<b>EIPV</b> When set, bit indicates that the instruction addressed by the instruction pointer pushed on the stack (when the machine check was generated) is directly associated with the error.
		2		<b>MCIP</b> When set, bit indicates that a machine check has been generated. If a second machine check is detected while this bit is still set, the processor enters a shutdown state. Software should write this bit to 0 after processing a machine check exception.
		63:3		Reserved.
186H	390	IA32_PERFVTSEL0	Thread	See Table 35-2.
187H	391	IA32_PERFVTSEL1	Thread	See Table 35-2.
188H	392	IA32_PERFVTSEL2	Thread	See Table 35-2.
189H	393	IA32_PERFVTSEL3	Thread	See Table 35-2.
18AH	394	IA32_PERFVTSEL4	Core	See Table 35-2; If CPUID.0AH:EAX[15:8] = 8
18BH	395	IA32_PERFVTSEL5	Core	See Table 35-2; If CPUID.0AH:EAX[15:8] = 8
18CH	396	IA32_PERFVTSEL6	Core	See Table 35-2; If CPUID.0AH:EAX[15:8] = 8

**Table 35-18. MSRs Supported by Intel® Processors  
based on Intel® microarchitecture code name Sandy Bridge (Contd.)**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
18DH	397	IA32_PERFEVTSEL7	Core	See Table 35-2; If CPUID.0AH:EAX[15:8] = 8
198H	408	IA32_PERF_STATUS	Package	See Table 35-2.
		15:0		Current Performance State Value.
		63:16		Reserved.
198H	408	MSR_PERF_STATUS	Package	
		47:32		Core Voltage (R/O) P-state core voltage can be computed by MSR_PERF_STATUS[37:32] * (float) 1/(2 <sup>13</sup> ).
199H	409	IA32_PERF_CTL	Thread	See Table 35-2.
19AH	410	IA32_CLOCK_MODULATION	Thread	<b>Clock Modulation (R/W)</b> See Table 35-2 IA32_CLOCK_MODULATION MSR was originally named IA32_THERM_CONTROL MSR.
		3:0		<b>On demand Clock Modulation Duty Cycle (R/W)</b> In 6.25% increment
		4		<b>On demand Clock Modulation Enable (R/W)</b>
		63:5		Reserved.
19BH	411	IA32_THERM_INTERRUPT	Core	<b>Thermal Interrupt Control (R/W)</b> See Table 35-2.
19CH	412	IA32_THERM_STATUS	Core	<b>Thermal Monitor Status (R/W)</b> See Table 35-2.
		0		<b>Thermal status (RO)</b> See Table 35-2.
		1		<b>Thermal status log (R/WCO)</b> See Table 35-2.
		2		<b>PROTCHOT # or FORCEPR# status (RO)</b> See Table 35-2.
		3		<b>PROTCHOT # or FORCEPR# log (R/WCO)</b> See Table 35-2.
		4		<b>Critical Temperature status (RO)</b> See Table 35-2.
		5		<b>Critical Temperature status log (R/WCO)</b> See Table 35-2.
		6		<b>Thermal threshold #1 status (RO)</b> See Table 35-2.
		7		<b>Thermal threshold #1 log (R/WCO)</b> See Table 35-2.

**Table 35-18. MSRs Supported by Intel® Processors  
based on Intel® microarchitecture code name Sandy Bridge (Contd.)**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		8		<b>Thermal threshold #2 status (RO)</b> See Table 35-2.
		9		<b>Thermal threshold #2 log (R/WCO)</b> See Table 35-2.
		10		<b>Power Limitation status (RO)</b> See Table 35-2.
		11		<b>Power Limitation log (R/WCO)</b> See Table 35-2.
		15:12		Reserved.
		22:16		<b>Digital Readout (RO)</b> See Table 35-2.
		26:23		Reserved.
		30:27		<b>Resolution in degrees Celsius (RO)</b> See Table 35-2.
		31		<b>Reading Valid (RO)</b> See Table 35-2.
		63:32		Reserved.
1A0H	416	IA32_MISC_ENABLE		<b>Enable Misc. Processor Features (R/W)</b> Allows a variety of processor functions to be enabled and disabled.
		0	Thread	<b>Fast-Strings Enable</b> See Table 35-2
		6:1		Reserved.
		7	Thread	<b>Performance Monitoring Available (R)</b> See Table 35-2.
		10:8		Reserved.
		11	Thread	<b>Branch Trace Storage Unavailable (RO)</b> See Table 35-2.
		12	Thread	<b>Processor Event Based Sampling Unavailable (RO)</b> See Table 35-2.
		15:13		Reserved.
		16	Package	<b>Enhanced Intel SpeedStep Technology Enable (R/W)</b> See Table 35-2.
		18	Thread	ENABLE MONITOR FSM. (R/W) See Table 35-2.
		21:19		Reserved.
		22	Thread	<b>Limit CPUID Maxval (R/W)</b> See Table 35-2.
		23	Thread	<b>xTPR Message Disable (R/W)</b> See Table 35-2.

**Table 35-18. MSRs Supported by Intel® Processors  
based on Intel® microarchitecture code name Sandy Bridge (Contd.)**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		33:24		Reserved.
		34	Thread	<b>XD Bit Disable (R/W)</b> See Table 35-2.
		37:35		Reserved.
		38	Package	<b>Turbo Mode Disable (R/W)</b> When set to 1 on processors that support Intel Turbo Boost Technology, the turbo mode feature is disabled and the IDA_Enable feature flag will be clear (CPUID.06H: EAX[1]=0). When set to a 0 on processors that support IDA, CPUID.06H: EAX[1] reports the processor's support of turbo mode is enabled. <b>Note:</b> the power-on default value is used by BIOS to detect hardware support of turbo mode. If power-on default value is 1, turbo mode is available in the processor. If power-on default value is 0, turbo mode is not available.
		63:39		Reserved.
1A2H	418	MSR_TEMPERATURE_TARGET	Unique	
		15:0		Reserved.
		23:16		<b>Temperature Target (R)</b> The minimum temperature at which PROCHOT# will be asserted. The value is degree C.
		63:24		Reserved.
1A4H	420	MSR_MISC_FEATURE_CONTROL		<b>Miscellaneous Feature Control (R/W)</b>
		0	Core	<b>L2 Hardware Prefetcher Disable (R/W)</b> If 1, disables the L2 hardware prefetcher, which fetches additional lines of code or data into the L2 cache.
		1	Core	<b>L2 Adjacent Cache Line Prefetcher Disable (R/W)</b> If 1, disables the adjacent cache line prefetcher, which fetches the cache line that comprises a cache line pair (128 bytes).
		2	Core	<b>DCU Hardware Prefetcher Disable (R/W)</b> If 1, disables the L1 data cache prefetcher, which fetches the next cache line into L1 data cache.
		3	Core	<b>DCU IP Prefetcher Disable (R/W)</b> If 1, disables the L1 data cache IP prefetcher, which uses sequential load history (based on instruction Pointer of previous loads) to determine whether to prefetch additional lines.
		63:4		Reserved.
1A6H	422	MSR_OFFCORE_RSP_0	Thread	<b>Offcore Response Event Select Register (R/W)</b>
1A7H	422	MSR_OFFCORE_RSP_1	Thread	<b>Offcore Response Event Select Register (R/W)</b>
1AAH	426	MSR_MISC_PWR_MGMT		See <a href="http://biosbits.org">http://biosbits.org</a> .
1B0H	432	IA32_ENERGY_PERF_BIAS	Package	See Table 35-2.



**Table 35-18. MSRs Supported by Intel® Processors  
based on Intel® microarchitecture code name Sandy Bridge (Contd.)**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
1B1H	433	IA32_PACKAGE_THERM_STATUS	Package	See Table 35-2.
1B2H	434	IA32_PACKAGE_THERM_INTERRUPT	Package	See Table 35-2.
1C8H	456	MSR_LBR_SELECT	Thread	<b>Last Branch Record Filtering Select Register (R/W)</b> See Section 17.7.2, "Filtering of Last Branch Records."
		0		CPL_EQ_0
		1		CPL_NEQ_0
		2		JCC
		3		NEAR_REL_CALL
		4		NEAR_IND_CALL
		5		NEAR_RET
		6		NEAR_IND_JMP
		7		NEAR_REL_JMP
		8		FAR_BRANCH
		63:9		Reserved.
1C9H	457	MSR_LASTBRANCH_TOS	Thread	<b>Last Branch Record Stack TOS (R/W)</b> Contains an index (bits 0-3) that points to the MSR containing the most recent branch record. See MSR_LASTBRANCH_0_FROM_IP (at 680H).
1D9H	473	IA32_DEBUGCTL	Thread	<b>Debug Control (R/W)</b> See Table 35-2.
		0		<b>LBR: Last Branch Record</b>
		1		<b>BTF</b>
		5:2		Reserved.
		6		<b>TR: Branch Trace</b>
		7		<b>BTS: Log Branch Trace Message to BTS buffer</b>
		8		<b>BTINT</b>
		9		<b>BTS_OFF_OS</b>
		10		<b>BTS_OFF_USER</b>
		11		<b>FREEZE_LBR_ON_PMI</b>
		12		<b>FREEZE_PERFMON_ON_PMI</b>
		13		<b>ENABLE_UNCORE_PMI</b>
		14		<b>FREEZE_WHILE_SMM</b>
63:15		Reserved.		

**Table 35-18. MSRs Supported by Intel® Processors  
based on Intel® microarchitecture code name Sandy Bridge (Contd.)**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
1DDH	477	MSR_LER_FROM_LIP	Thread	<b>Last Exception Record From Linear IP (R)</b> Contains a pointer to the last branch instruction that the processor executed prior to the last exception that was generated or the last interrupt that was handled.
1DEH	478	MSR_LER_TO_LIP	Thread	<b>Last Exception Record To Linear IP (R)</b> This area contains a pointer to the target of the last branch instruction that the processor executed prior to the last exception that was generated or the last interrupt that was handled.
1F2H	498	IA32_SMRR_PHYSBASE	Core	See Table 35-2.
1F3H	499	IA32_SMRR_PHYSMASK	Core	See Table 35-2.
1FCH	508	MSR_POWER_CTL	Core	See <a href="http://biosbits.org">http://biosbits.org</a> .
200H	512	IA32_MTRR_PHYSBASE0	Thread	See Table 35-2.
201H	513	IA32_MTRR_PHYSMASK0	Thread	See Table 35-2.
202H	514	IA32_MTRR_PHYSBASE1	Thread	See Table 35-2.
203H	515	IA32_MTRR_PHYSMASK1	Thread	See Table 35-2.
204H	516	IA32_MTRR_PHYSBASE2	Thread	See Table 35-2.
205H	517	IA32_MTRR_PHYSMASK2	Thread	See Table 35-2.
206H	518	IA32_MTRR_PHYSBASE3	Thread	See Table 35-2.
207H	519	IA32_MTRR_PHYSMASK3	Thread	See Table 35-2.
208H	520	IA32_MTRR_PHYSBASE4	Thread	See Table 35-2.
209H	521	IA32_MTRR_PHYSMASK4	Thread	See Table 35-2.
20AH	522	IA32_MTRR_PHYSBASE5	Thread	See Table 35-2.
20BH	523	IA32_MTRR_PHYSMASK5	Thread	See Table 35-2.
20CH	524	IA32_MTRR_PHYSBASE6	Thread	See Table 35-2.
20DH	525	IA32_MTRR_PHYSMASK6	Thread	See Table 35-2.
20EH	526	IA32_MTRR_PHYSBASE7	Thread	See Table 35-2.
20FH	527	IA32_MTRR_PHYSMASK7	Thread	See Table 35-2.
210H	528	IA32_MTRR_PHYSBASE8	Thread	See Table 35-2.
211H	529	IA32_MTRR_PHYSMASK8	Thread	See Table 35-2.
212H	530	IA32_MTRR_PHYSBASE9	Thread	See Table 35-2.
213H	531	IA32_MTRR_PHYSMASK9	Thread	See Table 35-2.
250H	592	IA32_MTRR_FIX64K_00000	Thread	See Table 35-2.
258H	600	IA32_MTRR_FIX16K_80000	Thread	See Table 35-2.
259H	601	IA32_MTRR_FIX16K_A0000	Thread	See Table 35-2.

**Table 35-18. MSRs Supported by Intel® Processors  
based on Intel® microarchitecture code name Sandy Bridge (Contd.)**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
268H	616	IA32_MTRR_FIX4K_C0000	Thread	See Table 35-2.
269H	617	IA32_MTRR_FIX4K_C8000	Thread	See Table 35-2.
26AH	618	IA32_MTRR_FIX4K_D0000	Thread	See Table 35-2.
26BH	619	IA32_MTRR_FIX4K_D8000	Thread	See Table 35-2.
26CH	620	IA32_MTRR_FIX4K_E0000	Thread	See Table 35-2.
26DH	621	IA32_MTRR_FIX4K_E8000	Thread	See Table 35-2.
26EH	622	IA32_MTRR_FIX4K_F0000	Thread	See Table 35-2.
26FH	623	IA32_MTRR_FIX4K_F8000	Thread	See Table 35-2.
277H	631	IA32_PAT	Thread	See Table 35-2.
280H	640	IA32_MCO_CTL2	Core	See Table 35-2.
281H	641	IA32_MC1_CTL2	Core	See Table 35-2.
282H	642	IA32_MC2_CTL2	Core	See Table 35-2.
283H	643	IA32_MC3_CTL2	Core	See Table 35-2.
284H	644	IA32_MC4_CTL2	Package	Always 0 (CMCI not supported).
2FFH	767	IA32_MTRR_DEF_TYPE	Thread	<b>Default Memory Types (R/W)</b> See Table 35-2.
309H	777	IA32_FIXED_CTR0	Thread	<b>Fixed-Function Performance Counter Register 0 (R/W)</b> See Table 35-2.
30AH	778	IA32_FIXED_CTR1	Thread	<b>Fixed-Function Performance Counter Register 1 (R/W)</b> See Table 35-2.
30BH	779	IA32_FIXED_CTR2	Thread	<b>Fixed-Function Performance Counter Register 2 (R/W)</b> See Table 35-2.
345H	837	IA32_PERF_CAPABILITIES	Thread	See Table 35-2. See Section 17.4.1, "IA32_DEBUGCTL MSR."
		5:0		LBR Format. See Table 35-2.
		6		PEBS Record Format.
		7		PEBSSaveArchRegs. See Table 35-2.
		11:8		PEBS_REC_FORMAT. See Table 35-2.
		12		SMM_FREEZE. See Table 35-2.
		63:13		Reserved.
38DH	909	IA32_FIXED_CTR_CTRL	Thread	<b>Fixed-Function-Counter Control Register (R/W)</b> See Table 35-2.
38EH	910	IA32_PERF_GLOBAL_STATUS		See Table 35-2. See Section 18.4.2, "Global Counter Control Facilities."
		0	Thread	<b>Ovf_PMC0</b>
		1	Thread	<b>Ovf_PMC1</b>
		2	Thread	<b>Ovf_PMC2</b>

**Table 35-18. MSRs Supported by Intel® Processors  
based on Intel® microarchitecture code name Sandy Bridge (Contd.)**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		3	Thread	<b>Ovf_PMC3</b>
		4	Core	<b>Ovf_PMC4</b> (if CPUID.0AH:EAX[15:8] > 4)
		5	Core	<b>Ovf_PMC5</b> (if CPUID.0AH:EAX[15:8] > 5)
		6	Core	<b>Ovf_PMC6</b> (if CPUID.0AH:EAX[15:8] > 6)
		7	Core	<b>Ovf_PMC7</b> (if CPUID.0AH:EAX[15:8] > 7)
		31:8		Reserved.
		32	Thread	<b>Ovf_FixedCtr0</b>
		33	Thread	<b>Ovf_FixedCtr1</b>
		34	Thread	<b>Ovf_FixedCtr2</b>
		60:35		Reserved.
		61	Thread	<b>Ovf_Uncore</b>
		62	Thread	<b>Ovf_BufDSSAVE</b>
		63	Thread	<b>CondChgd</b>
38FH	911	IA32_PERF_GLOBAL_CTRL	Thread	See Table 35-2. See Section 18.4.2, "Global Counter Control Facilities."
		0	Thread	<b>Set 1 to enable PMC0 to count</b>
		1	Thread	<b>Set 1 to enable PMC1 to count</b>
		2	Thread	<b>Set 1 to enable PMC2 to count</b>
		3	Thread	<b>Set 1 to enable PMC3 to count</b>
		4	Core	<b>Set 1 to enable PMC4 to count</b> (if CPUID.0AH:EAX[15:8] > 4)
		5	Core	<b>Set 1 to enable PMC5 to count</b> (if CPUID.0AH:EAX[15:8] > 5)
		6	Core	<b>Set 1 to enable PMC6 to count</b> (if CPUID.0AH:EAX[15:8] > 6)
		7	Core	<b>Set 1 to enable PMC7 to count</b> (if CPUID.0AH:EAX[15:8] > 7)
		31:8		Reserved.
		32	Thread	<b>Set 1 to enable FixedCtr0 to count</b>
		33	Thread	<b>Set 1 to enable FixedCtr1 to count</b>
		34	Thread	<b>Set 1 to enable FixedCtr2 to count</b>
63:35		Reserved.		
390H	912	IA32_PERF_GLOBAL_OVF_CTRL		See Table 35-2. See Section 18.4.2, "Global Counter Control Facilities."
		0	Thread	<b>Set 1 to clear Ovf_PMC0</b>
		1	Thread	<b>Set 1 to clear Ovf_PMC1</b>
		2	Thread	<b>Set 1 to clear Ovf_PMC2</b>
		3	Thread	<b>Set 1 to clear Ovf_PMC3</b>
		4	Core	<b>Set 1 to clear Ovf_PMC4</b> (if CPUID.0AH:EAX[15:8] > 4)
		5	Core	<b>Set 1 to clear Ovf_PMC5</b> (if CPUID.0AH:EAX[15:8] > 5)
6	Core	<b>Set 1 to clear Ovf_PMC6</b> (if CPUID.0AH:EAX[15:8] > 6)		

**Table 35-18. MSRs Supported by Intel® Processors based on Intel® microarchitecture code name Sandy Bridge (Contd.)**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		7	Core	Set 1 to clear Ovf_PMC7 (if CPUID.0AH:EAX[15:8] > 7)
		31:8		Reserved.
		32	Thread	Set 1 to clear Ovf_FixedCtr0
		33	Thread	Set 1 to clear Ovf_FixedCtr1
		34	Thread	Set 1 to clear Ovf_FixedCtr2
		60:35		Reserved.
		61	Thread	Set 1 to clear Ovf_Uncore
		62	Thread	Set 1 to clear Ovf_BufDSSAVE
		63	Thread	Set 1 to clear CondChgd
3F1H	1009	MSR_PEBS_ENABLE	Thread	See Section 18.7.1.1, "Precise Event Based Sampling (PEBS)."
		0		Enable PEBS on IA32_PMC0. (R/W)
		1		Enable PEBS on IA32_PMC1. (R/W)
		2		Enable PEBS on IA32_PMC2. (R/W)
		3		Enable PEBS on IA32_PMC3. (R/W)
		31:4		Reserved.
		32		Enable Load Latency on IA32_PMC0. (R/W)
		33		Enable Load Latency on IA32_PMC1. (R/W)
		34		Enable Load Latency on IA32_PMC2. (R/W)
		35		Enable Load Latency on IA32_PMC3. (R/W)
		62:36		Reserved.
		63		Enable Precise Store. (R/W)
3F6H	1014	MSR_PEBS_LD_LAT	Thread	see See Section 18.7.1.2, "Load Latency Performance Monitoring Facility."
		15:0		Minimum threshold latency value of tagged load operation that will be counted. (R/W)
		63:36		Reserved.
3F8H	1016	MSR_PKG_C3_RESIDENCY	Package	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		63:0		Package C3 Residency Counter. (R/O) Value since last reset that this package is in processor-specific C3 states. Count at the same frequency as the TSC.
3F9H	1017	MSR_PKG_C6_RESIDENCY	Package	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		63:0		Package C6 Residency Counter. (R/O) Value since last reset that this package is in processor-specific C6 states. Count at the same frequency as the TSC.

**Table 35-18. MSRs Supported by Intel® Processors  
based on Intel® microarchitecture code name Sandy Bridge (Contd.)**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
3FAH	1018	MSR_PKG_C7_RESIDENCY	Package	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		63:0		Package C7 Residency Counter. (R/O) Value since last reset that this package is in processor-specific C7 states. Count at the same frequency as the TSC.
3FCH	1020	MSR_CORE_C3_RESIDENCY	Core	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		63:0		CORE C3 Residency Counter. (R/O) Value since last reset that this core is in processor-specific C3 states. Count at the same frequency as the TSC.
3FDH	1021	MSR_CORE_C6_RESIDENCY	Core	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		63:0		CORE C6 Residency Counter. (R/O) Value since last reset that this core is in processor-specific C6 states. Count at the same frequency as the TSC.
3FEH	1022	MSR_CORE_C7_RESIDENCY	Core	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		63:0		CORE C7 Residency Counter. (R/O) Value since last reset that this core is in processor-specific C7 states. Count at the same frequency as the TSC.
400H	1024	IA32_MC0_CTL	Core	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
401H	1025	IA32_MC0_STATUS	Core	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 16.
402H	1026	IA32_MC0_ADDR	Core	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
403H	1027	IA32_MC0_MISC	Core	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
404H	1028	IA32_MC1_CTL	Core	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
405H	1029	IA32_MC1_STATUS	Core	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 16.
406H	1030	IA32_MC1_ADDR	Core	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
407H	1031	IA32_MC1_MISC	Core	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
408H	1032	IA32_MC2_CTL	Core	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
409H	1033	IA32_MC2_STATUS	Core	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 16.
40AH	1034	IA32_MC2_ADDR	Core	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
40BH	1035	IA32_MC2_MISC	Core	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
40CH	1036	IA32_MC3_CTL	Core	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
40DH	1037	IA32_MC3_STATUS	Core	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 16.
40EH	1038	IA32_MC3_ADDR	Core	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
40FH	1039	IA32_MC3_MISC	Core	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."

**Table 35-18. MSRs Supported by Intel® Processors  
based on Intel® microarchitecture code name Sandy Bridge (Contd.)**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
410H	1040	IA32_MC4_CTL	Core	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
		0		<b>PCU Hardware Error (R/W)</b> When set, enables signaling of PCU hardware detected errors.
		1		<b>PCU Controller Error (R/W)</b> When set, enables signaling of PCU controller detected errors
		2		<b>PCU Firmware Error (R/W)</b> When set, enables signaling of PCU firmware detected errors
		63:2		Reserved.
411H	1041	IA32_MC4_STATUS	Core	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 16.
480H	1152	IA32_VMX_BASIC	Thread	<b>Reporting Register of Basic VMX Capabilities (R/O)</b> See Table 35-2. See Appendix A.1, "Basic VMX Information."
481H	1153	IA32_VMX_PINBASED_ CTLS	Thread	<b>Capability Reporting Register of Pin-based VM-execution Controls (R/O)</b> See Table 35-2. See Appendix A.3, "VM-Execution Controls."
482H	1154	IA32_VMX_PROCBASED_ CTLS	Thread	<b>Capability Reporting Register of Primary Processor-based VM-execution Controls (R/O)</b> See Appendix A.3, "VM-Execution Controls."
483H	1155	IA32_VMX_EXIT_ CTLS	Thread	<b>Capability Reporting Register of VM-exit Controls (R/O)</b> See Table 35-2. See Appendix A.4, "VM-Exit Controls."
484H	1156	IA32_VMX_ENTRY_ CTLS	Thread	<b>Capability Reporting Register of VM-entry Controls (R/O)</b> See Table 35-2. See Appendix A.5, "VM-Entry Controls."
485H	1157	IA32_VMX_MISC	Thread	<b>Reporting Register of Miscellaneous VMX Capabilities (R/O)</b> See Table 35-2. See Appendix A.6, "Miscellaneous Data."
486H	1158	IA32_VMX_CR0_FIXED0	Thread	<b>Capability Reporting Register of CR0 Bits Fixed to 0 (R/O)</b> See Table 35-2. See Appendix A.7, "VMX-Fixed Bits in CR0."
487H	1159	IA32_VMX_CR0_FIXED1	Thread	<b>Capability Reporting Register of CR0 Bits Fixed to 1 (R/O)</b> See Table 35-2. See Appendix A.7, "VMX-Fixed Bits in CR0."
488H	1160	IA32_VMX_CR4_FIXED0	Thread	<b>Capability Reporting Register of CR4 Bits Fixed to 0 (R/O)</b> See Table 35-2. See Appendix A.8, "VMX-Fixed Bits in CR4."

**Table 35-18. MSRs Supported by Intel® Processors  
based on Intel® microarchitecture code name Sandy Bridge (Contd.)**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
489H	1161	IA32_VMX_CR4_FIXED1	Thread	<b>Capability Reporting Register of CR4 Bits Fixed to 1 (R/O)</b> See Table 35-2. See Appendix A.8, "VMX-Fixed Bits in CR4."
48AH	1162	IA32_VMX_VMCS_ENUM	Thread	<b>Capability Reporting Register of VMCS Field Enumeration (R/O)</b> See Table 35-2. See Appendix A.9, "VMCS Enumeration."
48BH	1163	IA32_VMX_PROCBASED_CTLDS2	Thread	<b>Capability Reporting Register of Secondary Processor-based VM-execution Controls (R/O)</b> See Appendix A.3, "VM-Execution Controls."
48CH	1164	IA32_VMX_EPT_VPID_ENUM	Thread	<b>Capability Reporting Register of EPT and VPID (R/O)</b> See Table 35-2
48DH	1165	IA32_VMX_TRUE_PINBASED_CTLDS	Thread	<b>Capability Reporting Register of Pin-based VM-execution Flex Controls (R/O)</b> See Table 35-2
48EH	1166	IA32_VMX_TRUE_PROCBASED_CTLDS	Thread	<b>Capability Reporting Register of Primary Processor-based VM-execution Flex Controls (R/O)</b> See Table 35-2
48FH	1167	IA32_VMX_TRUE_EXIT_CTLDS	Thread	<b>Capability Reporting Register of VM-exit Flex Controls (R/O)</b> See Table 35-2
490H	1168	IA32_VMX_TRUE_ENTRY_CTLDS	Thread	<b>Capability Reporting Register of VM-entry Flex Controls (R/O)</b> See Table 35-2
4C1H	1217	IA32_A_PMC0	Thread	See Table 35-2.
4C2H	1218	IA32_A_PMC1	Thread	See Table 35-2.
4C3H	1219	IA32_A_PMC2	Thread	See Table 35-2.
4C4H	1220	IA32_A_PMC3	Thread	See Table 35-2.
4C5H	1221	IA32_A_PMC4	Core	See Table 35-2.
4C6H	1222	IA32_A_PMC5	Core	See Table 35-2.
4C7H	1223	IA32_A_PMC6	Core	See Table 35-2.
4C8H	1224	IA32_A_PMC7	Core	See Table 35-2.
600H	1536	IA32_DS_AREA	Thread	<b>DS Save Area (R/W)</b> See Table 35-2. See Section 18.12.4, "Debug Store (DS) Mechanism."
606H	1542	MSR_RAPL_POWER_UNIT	Package	<b>Unit Multipliers used in RAPL Interfaces (R/O)</b> See Section 14.9.1, "RAPL Interfaces."
60AH	1546	MSR_PKG_C3_INTERRUPT_RESPONSE_LIMIT	Package	<b>Package C3 Interrupt Response Limit (R/W)</b> Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.



**Table 35-18. MSRs Supported by Intel® Processors  
based on Intel® microarchitecture code name Sandy Bridge (Contd.)**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		9:0		<b>Interrupt response time limit (R/W)</b> Specifies the limit that should be used to decide if the package should be put into a package C3 state.
		12:10		<b>Time Unit (R/W)</b> Specifies the encoding value of time unit of the interrupt response time limit. The following time unit encodings are supported: 000b: 1 ns 001b: 32 ns 010b: 1024 ns 011b: 32768 ns 100b: 1048576 ns 101b: 33554432 ns
		14:13		Reserved.
		15		<b>Valid (R/W)</b> Indicates whether the values in bits 12:0 are valid and can be used by the processor for package C-state management.
		63:16		Reserved.
60BH	1547	MSR_PKGC6_IRTL	Package	<b>Package C6 Interrupt Response Limit (R/W)</b> This MSR defines the budget allocated for the package to exit from C6 to a C0 state, where interrupt request can be delivered to the core and serviced. Additional core-exit latency may be applicable depending on the actual C-state the core is in. Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		9:0		<b>Interrupt response time limit (R/W)</b> Specifies the limit that should be used to decide if the package should be put into a package C6 state.
		12:10		<b>Time Unit (R/W)</b> Specifies the encoding value of time unit of the interrupt response time limit. The following time unit encodings are supported: 000b: 1 ns 001b: 32 ns 010b: 1024 ns 011b: 32768 ns 100b: 1048576 ns 101b: 33554432 ns
		14:13		Reserved.
		15		<b>Valid (R/W)</b> Indicates whether the values in bits 12:0 are valid and can be used by the processor for package C-state management.
		63:16		Reserved.

**Table 35-18. MSRs Supported by Intel® Processors  
based on Intel® microarchitecture code name Sandy Bridge (Contd.)**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
60DH	1549	MSR_PKG_C2_RESIDENCY	Package	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		63:0		<b>Package C2 Residency Counter. (R/O)</b> Value since last reset that this package is in processor-specific C2 states. Count at the same frequency as the TSC.
610H	1552	MSR_PKG_POWER_LIMIT	Package	<b>PKG RAPL Power Limit Control (R/W)</b> See Section 14.9.3, "Package RAPL Domain."
611H	1553	MSR_PKG_ENERGY_STATUS	Package	<b>PKG Energy Status (R/O)</b> See Section 14.9.3, "Package RAPL Domain."
614H	1556	MSR_PKG_POWER_INFO	Package	<b>PKG RAPL Parameters (R/W)</b> See Section 14.9.3, "Package RAPL Domain."
638H	1592	MSR_PPO_POWER_LIMIT	Package	<b>PPO RAPL Power Limit Control (R/W)</b> See Section 14.9.4, "PPO/PP1 RAPL Domains."
680H	1664	MSR_LASTBRANCH_0_FROM_IP	Thread	<b>Last Branch Record 0 From IP (R/W)</b> One of sixteen pairs of last branch record registers on the last branch record stack. This part of the stack contains pointers to the <b>source instruction</b> . See also: <ul style="list-style-type: none"> <li>▪ Last Branch Record Stack TOS at 1C9H</li> <li>▪ Section 17.7.1 and record format in Section 17.4.8.1</li> </ul>
681H	1665	MSR_LASTBRANCH_1_FROM_IP	Thread	<b>Last Branch Record 1 From IP (R/W)</b> See description of MSR_LASTBRANCH_0_FROM_IP.
682H	1666	MSR_LASTBRANCH_2_FROM_IP	Thread	<b>Last Branch Record 2 From IP (R/W)</b> See description of MSR_LASTBRANCH_0_FROM_IP.
683H	1667	MSR_LASTBRANCH_3_FROM_IP	Thread	<b>Last Branch Record 3 From IP (R/W)</b> See description of MSR_LASTBRANCH_0_FROM_IP.
684H	1668	MSR_LASTBRANCH_4_FROM_IP	Thread	<b>Last Branch Record 4 From IP (R/W)</b> See description of MSR_LASTBRANCH_0_FROM_IP.
685H	1669	MSR_LASTBRANCH_5_FROM_IP	Thread	<b>Last Branch Record 5 From IP (R/W)</b> See description of MSR_LASTBRANCH_0_FROM_IP.
686H	1670	MSR_LASTBRANCH_6_FROM_IP	Thread	<b>Last Branch Record 6 From IP (R/W)</b> See description of MSR_LASTBRANCH_0_FROM_IP.
687H	1671	MSR_LASTBRANCH_7_FROM_IP	Thread	<b>Last Branch Record 7 From IP (R/W)</b> See description of MSR_LASTBRANCH_0_FROM_IP.
688H	1672	MSR_LASTBRANCH_8_FROM_IP	Thread	<b>Last Branch Record 8 From IP (R/W)</b> See description of MSR_LASTBRANCH_0_FROM_IP.
689H	1673	MSR_LASTBRANCH_9_FROM_IP	Thread	<b>Last Branch Record 9 From IP (R/W)</b> See description of MSR_LASTBRANCH_0_FROM_IP.
68AH	1674	MSR_LASTBRANCH_10_FROM_IP	Thread	<b>Last Branch Record 10 From IP (R/W)</b> See description of MSR_LASTBRANCH_0_FROM_IP.

**Table 35-18. MSRs Supported by Intel® Processors  
based on Intel® microarchitecture code name Sandy Bridge (Contd.)**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
68BH	1675	MSR_LASTBRANCH_11_FROM_IP	Thread	<b>Last Branch Record 11 From IP (R/W)</b> See description of MSR_LASTBRANCH_0_FROM_IP.
68CH	1676	MSR_LASTBRANCH_12_FROM_IP	Thread	<b>Last Branch Record 12 From IP (R/W)</b> See description of MSR_LASTBRANCH_0_FROM_IP.
68DH	1677	MSR_LASTBRANCH_13_FROM_IP	Thread	<b>Last Branch Record 13 From IP (R/W)</b> See description of MSR_LASTBRANCH_0_FROM_IP.
68EH	1678	MSR_LASTBRANCH_14_FROM_IP	Thread	<b>Last Branch Record 14 From IP (R/W)</b> See description of MSR_LASTBRANCH_0_FROM_IP.
68FH	1679	MSR_LASTBRANCH_15_FROM_IP	Thread	<b>Last Branch Record 15 From IP (R/W)</b> See description of MSR_LASTBRANCH_0_FROM_IP.
6C0H	1728	MSR_LASTBRANCH_0_TO_IP	Thread	<b>Last Branch Record 0 To IP (R/W)</b> One of sixteen pairs of last branch record registers on the last branch record stack. This part of the stack contains pointers to the destination instruction.
6C1H	1729	MSR_LASTBRANCH_1_TO_IP	Thread	<b>Last Branch Record 1 To IP (R/W)</b> See description of MSR_LASTBRANCH_0_TO_IP.
6C2H	1730	MSR_LASTBRANCH_2_TO_IP	Thread	<b>Last Branch Record 2 To IP (R/W)</b> See description of MSR_LASTBRANCH_0_TO_IP.
6C3H	1731	MSR_LASTBRANCH_3_TO_IP	Thread	<b>Last Branch Record 3 To IP (R/W)</b> See description of MSR_LASTBRANCH_0_TO_IP.
6C4H	1732	MSR_LASTBRANCH_4_TO_IP	Thread	<b>Last Branch Record 4 To IP (R/W)</b> See description of MSR_LASTBRANCH_0_TO_IP.
6C5H	1733	MSR_LASTBRANCH_5_TO_IP	Thread	<b>Last Branch Record 5 To IP (R/W)</b> See description of MSR_LASTBRANCH_0_TO_IP.
6C6H	1734	MSR_LASTBRANCH_6_TO_IP	Thread	<b>Last Branch Record 6 To IP (R/W)</b> See description of MSR_LASTBRANCH_0_TO_IP.
6C7H	1735	MSR_LASTBRANCH_7_TO_IP	Thread	<b>Last Branch Record 7 To IP (R/W)</b> See description of MSR_LASTBRANCH_0_TO_IP.
6C8H	1736	MSR_LASTBRANCH_8_TO_IP	Thread	<b>Last Branch Record 8 To IP (R/W)</b> See description of MSR_LASTBRANCH_0_TO_IP.
6C9H	1737	MSR_LASTBRANCH_9_TO_IP	Thread	<b>Last Branch Record 9 To IP (R/W)</b> See description of MSR_LASTBRANCH_0_TO_IP.
6CAH	1738	MSR_LASTBRANCH_10_TO_IP	Thread	<b>Last Branch Record 10 To IP (R/W)</b> See description of MSR_LASTBRANCH_0_TO_IP.
6CBH	1739	MSR_LASTBRANCH_11_TO_IP	Thread	<b>Last Branch Record 11 To IP (R/W)</b> See description of MSR_LASTBRANCH_0_TO_IP.
6CCH	1740	MSR_LASTBRANCH_12_TO_IP	Thread	<b>Last Branch Record 12 To IP (R/W)</b> See description of MSR_LASTBRANCH_0_TO_IP.

**Table 35-18. MSRs Supported by Intel® Processors  
based on Intel® microarchitecture code name Sandy Bridge (Contd.)**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
6CDH	1741	MSR_LASTBRANCH_13_TO_IP	Thread	<b>Last Branch Record 13 To IP (R/W)</b> See description of MSR_LASTBRANCH_0_TO_IP.
6CEH	1742	MSR_LASTBRANCH_14_TO_IP	Thread	<b>Last Branch Record 14 To IP (R/W)</b> See description of MSR_LASTBRANCH_0_TO_IP.
6CFH	1743	MSR_LASTBRANCH_15_TO_IP	Thread	<b>Last Branch Record 15 To IP (R/W)</b> See description of MSR_LASTBRANCH_0_TO_IP.
6E0H	1760	IA32_TSC_DEADLINE	Thread	See Table 35-2.
802H-83FH		X2APIC MSRs	Thread	See Table 35-2.
C000_0080H		IA32_EFER	Thread	<b>Extended Feature Enables</b> See Table 35-2.
C000_0081H		IA32_STAR	Thread	<b>System Call Target Address (R/W)</b> See Table 35-2.
C000_0082H		IA32_LSTAR	Thread	<b>IA-32e Mode System Call Target Address (R/W)</b> See Table 35-2.
C000_0084H		IA32_FMASK	Thread	<b>System Call Flag Mask (R/W)</b> See Table 35-2.
C000_0100H		IA32_FS_BASE	Thread	<b>Map of BASE Address of FS (R/W)</b> See Table 35-2.
C000_0101H		IA32_GS_BASE	Thread	<b>Map of BASE Address of GS (R/W)</b> See Table 35-2.
C000_0102H		IA32_KERNEL_GSBASE	Thread	<b>Swap Target of BASE Address of GS (R/W)</b> See Table 35-2.
C000_0103H		IA32_TSC_AUX	Thread	<b>AUXILIARY TSC Signature (R/W)</b> See Table 35-2 and Section 17.15.2, "IA32_TSC_AUX Register and RDTSCP Support."

### 35.9.1 MSRs In 2nd Generation Intel® Core™ Processor Family (Based on Intel® Microarchitecture Code Name Sandy Bridge)

Table 35-19 and Table 35-20 list model-specific registers (MSRs) that are specific to the 2nd generation Intel® Core™ processor family (based on Intel microarchitecture code name Sandy Bridge). These processors have a CPUID signature with DisplayFamily\_DisplayModel of 06\_2AH; see Table 35-1.

**Table 35-19. MSRs Supported by 2nd Generation Intel® Core™ Processors (Intel® microarchitecture code name Sandy Bridge)**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
1ADH	429	MSR_TURBO_RATIO_LIMIT	Package	<b>Maximum Ratio Limit of Turbo Mode</b> RO if MSR_PLATFORM_INFO.[28] = 0, RW if MSR_PLATFORM_INFO.[28] = 1
		7:0	Package	<b>Maximum Ratio Limit for 1C</b> Maximum turbo ratio limit of 1 core active.
		15:8	Package	<b>Maximum Ratio Limit for 2C</b> Maximum turbo ratio limit of 2 core active.
		23:16	Package	<b>Maximum Ratio Limit for 3C</b> Maximum turbo ratio limit of 3 core active.
		31:24	Package	<b>Maximum Ratio Limit for 4C</b> Maximum turbo ratio limit of 4 core active.
		63:32		Reserved.
60CH	1548	MSR_PKGC7_IRTL	Package	<b>Package C7 Interrupt Response Limit (R/W)</b> This MSR defines the budget allocated for the package to exit from C7 to a C0 state, where interrupt request can be delivered to the core and serviced. Additional core-exit latency may be applicable depending on the actual C-state the core is in. Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		9:0		<b>Interrupt response time limit (R/W)</b> Specifies the limit that should be used to decide if the package should be put into a package C7 state.
		12:10		<b>Time Unit (R/W)</b> Specifies the encoding value of time unit of the interrupt response time limit. The following time unit encodings are supported: 000b: 1 ns 001b: 32 ns 010b: 1024 ns 011b: 32768 ns 100b: 1048576 ns 101b: 33554432 ns
		14:13		Reserved.
		15		<b>Valid (R/W)</b> Indicates whether the values in bits 12:0 are valid and can be used by the processor for package C-state management.
		63:16		Reserved.
639H	1593	MSR_PP0_ENERGY_STATUS	Package	<b>PP0 Energy Status (R/O)</b> See Section 14.9.4, "PP0/PP1 RAPL Domains."

**Table 35-19. MSRs Supported by 2nd Generation Intel® Core™ Processors (Intel® microarchitecture code name Sandy Bridge) (Contd.)**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
63AH	1594	MSR_PP0_POLICY	Package	<b>PP0 Balance Policy (R/W)</b> See Section 14.9.4, "PP0/PP1 RAPL Domains."
640H	1600	MSR_PP1_POWER_LIMIT	Package	<b>PP1 RAPL Power Limit Control (R/W)</b> See Section 14.9.4, "PP0/PP1 RAPL Domains."
641H	1601	MSR_PP1_ENERGY_STATUS	Package	<b>PP1 Energy Status (R/O)</b> See Section 14.9.4, "PP0/PP1 RAPL Domains."
642H	1602	MSR_PP1_POLICY	Package	<b>PP1 Balance Policy (R/W)</b> See Section 14.9.4, "PP0/PP1 RAPL Domains."

See Table 35-18, Table 35-19, and Table 35-20 for MSR definitions applicable to processors with CPUID signature 06\_2AH.

Table 35-20 lists the MSRs of uncore PMU for Intel processors with CPUID signature 06\_2AH.

**Table 35-20. Uncore PMU MSRs Supported by 2nd Generation Intel® Core™ Processors**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
391H	913	MSR_UNC_PERF_GLOBAL_CTRL	Package	Uncore PMU global control
		0		Slice 0 select
		1		Slice 1 select
		2		Slice 2 select
		3		Slice 3 select
		4		Slice 4 select
		18:5		Reserved.
		29		Enable all uncore counters
		30		Enable wake on PMI
		31		Enable Freezing counter when overflow
63:32	Reserved.			
392H	914	MSR_UNC_PERF_GLOBAL_STATUS	Package	Uncore PMU main status
		0		Fixed counter overflowed
		1		An ARB counter overflowed
		2		Reserved
		3		A CBox counter overflowed (on any slice)
		63:4		Reserved.
394H	916	MSR_UNC_PERF_FIXED_CTRL	Package	Uncore fixed counter control (R/W)
		19:0		Reserved
		20		Enable overflow propagation

**Table 35-20. Uncore PMU MSRs Supported by 2nd Generation Intel® Core™ Processors**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		21		Reserved
		22		Enable counting
		63:23		Reserved.
395H	917	MSR_UNC_PERF_FIXED_CTR	Package	Uncore fixed counter
		47:0		Current count
		63:48		Reserved.
396H	918	MSR_UNC_CBO_CONFIG	Package	Uncore C-Box configuration information (R/O)
		3:0		Report the number of C-Box units with performance counters, including processor cores and processor graphics"
		63:4		Reserved.
3B0H	946	MSR_UNC_ARB_PERFCTR0	Package	Uncore Arb unit, performance counter 0
3B1H	947	MSR_UNC_ARB_PERFCTR1	Package	Uncore Arb unit, performance counter 1
3B2H	944	MSR_UNC_ARB_PERFEVTSELO	Package	Uncore Arb unit, counter 0 event select MSR
3B3H	945	MSR_UNC_ARB_PERFEVTSEL1	Package	Uncore Arb unit, counter 1 event select MSR
700H	1792	MSR_UNC_CBO_0_PERFEVTSELO	Package	Uncore C-Box 0, counter 0 event select MSR
701H	1793	MSR_UNC_CBO_0_PERFEVTSEL1	Package	Uncore C-Box 0, counter 1 event select MSR
702H	1794	MSR_UNC_CBO_0_PERFEVTSEL2	Package	Uncore C-Box 0, counter 2 event select MSR.
703H	1795	MSR_UNC_CBO_0_PERFEVTSEL3	Package	Uncore C-Box 0, counter 3 event select MSR.
705H	1797	MSR_UNC_CBO_0_UNIT_STATUS	Package	Uncore C-Box 0, unit status for counter 0-3
706H	1798	MSR_UNC_CBO_0_PERFCTR0	Package	Uncore C-Box 0, performance counter 0
707H	1799	MSR_UNC_CBO_0_PERFCTR1	Package	Uncore C-Box 0, performance counter 1
708H	1800	MSR_UNC_CBO_0_PERFCTR2	Package	Uncore C-Box 0, performance counter 2.
709H	1801	MSR_UNC_CBO_0_PERFCTR3	Package	Uncore C-Box 0, performance counter 3.
710H	1808	MSR_UNC_CBO_1_PERFEVTSELO	Package	Uncore C-Box 1, counter 0 event select MSR
711H	1809	MSR_UNC_CBO_1_PERFEVTSEL1	Package	Uncore C-Box 1, counter 1 event select MSR
712H	1810	MSR_UNC_CBO_1_PERFEVTSEL2	Package	Uncore C-Box 1, counter 2 event select MSR.
713H	1811	MSR_UNC_CBO_1_PERFEVTSEL3	Package	Uncore C-Box 1, counter 3 event select MSR.

**Table 35-20. Uncore PMU MSRs Supported by 2nd Generation Intel® Core™ Processors**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
715H	1813	MSR_UNC_CBO_1_UNIT_STATUS	Package	Uncore C-Box 1, unit status for counter 0-3
716H	1814	MSR_UNC_CBO_1_PERFCTR0	Package	Uncore C-Box 1, performance counter 0
717H	1815	MSR_UNC_CBO_1_PERFCTR1	Package	Uncore C-Box 1, performance counter 1
718H	1816	MSR_UNC_CBO_1_PERFCTR2	Package	Uncore C-Box 1, performance counter 2.
719H	1817	MSR_UNC_CBO_1_PERFCTR3	Package	Uncore C-Box 1, performance counter 3.
720H	1824	MSR_UNC_CBO_2_PERFEVTSELO	Package	Uncore C-Box 2, counter 0 event select MSR
721H	1825	MSR_UNC_CBO_2_PERFEVTSEL1	Package	Uncore C-Box 2, counter 1 event select MSR
722H	1826	MSR_UNC_CBO_2_PERFEVTSEL2	Package	Uncore C-Box 2, counter 2 event select MSR.
723H	1827	MSR_UNC_CBO_2_PERFEVTSEL3	Package	Uncore C-Box 2, counter 3 event select MSR.
725H	1829	MSR_UNC_CBO_2_UNIT_STATUS	Package	Uncore C-Box 2, unit status for counter 0-3
726H	1830	MSR_UNC_CBO_2_PERFCTR0	Package	Uncore C-Box 2, performance counter 0
727H	1831	MSR_UNC_CBO_2_PERFCTR1	Package	Uncore C-Box 2, performance counter 1
728H	1832	MSR_UNC_CBO_3_PERFCTR2	Package	Uncore C-Box 3, performance counter 2.
729H	1833	MSR_UNC_CBO_3_PERFCTR3	Package	Uncore C-Box 3, performance counter 3.
730H	1840	MSR_UNC_CBO_3_PERFEVTSELO	Package	Uncore C-Box 3, counter 0 event select MSR
731H	1841	MSR_UNC_CBO_3_PERFEVTSEL1	Package	Uncore C-Box 3, counter 1 event select MSR.
732H	1842	MSR_UNC_CBO_3_PERFEVTSEL2	Package	Uncore C-Box 3, counter 2 event select MSR.
733H	1843	MSR_UNC_CBO_3_PERFEVTSEL3	Package	Uncore C-Box 3, counter 3 event select MSR.
735H	1845	MSR_UNC_CBO_3_UNIT_STATUS	Package	Uncore C-Box 3, unit status for counter 0-3
736H	1846	MSR_UNC_CBO_3_PERFCTR0	Package	Uncore C-Box 3, performance counter 0.
737H	1847	MSR_UNC_CBO_3_PERFCTR1	Package	Uncore C-Box 3, performance counter 1.
738H	1848	MSR_UNC_CBO_3_PERFCTR2	Package	Uncore C-Box 3, performance counter 2.
739H	1849	MSR_UNC_CBO_3_PERFCTR3	Package	Uncore C-Box 3, performance counter 3.
740H	1856	MSR_UNC_CBO_4_PERFEVTSELO	Package	Uncore C-Box 4, counter 0 event select MSR
741H	1857	MSR_UNC_CBO_4_PERFEVTSEL1	Package	Uncore C-Box 4, counter 1 event select MSR.
742H	1858	MSR_UNC_CBO_4_PERFEVTSEL2	Package	Uncore C-Box 4, counter 2 event select MSR.



**Table 35-20. Uncore PMU MSRs Supported by 2nd Generation Intel® Core™ Processors**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
743H	1859	MSR_UNC_CBO_4_PERFEVTSEL3	Package	Uncore C-Box 4, counter 3 event select MSR.
745H	1861	MSR_UNC_CBO_4_UNIT_STATUS	Package	Uncore C-Box 4, unit status for counter 0-3
746H	1862	MSR_UNC_CBO_4_PERFCTR0	Package	Uncore C-Box 4, performance counter 0.
747H	1863	MSR_UNC_CBO_4_PERFCTR1	Package	Uncore C-Box 4, performance counter 1.
748H	1864	MSR_UNC_CBO_4_PERFCTR2	Package	Uncore C-Box 4, performance counter 2.
749H	1865	MSR_UNC_CBO_4_PERFCTR3	Package	Uncore C-Box 4, performance counter 3.

### 35.9.2 MSRs In Intel® Xeon® Processor E5 Family (Based on Intel® Microarchitecture Code Name Sandy Bridge)

Table 35-21 lists additional model-specific registers (MSRs) that are specific to the Intel® Xeon® Processor E5 Family (based on Intel® microarchitecture code name Sandy Bridge). These processors have a CPUID signature with DisplayFamily\_DisplayModel of 06\_2DH, and also supports MSRs listed in Table 35-18 and Table 35-22.

**Table 35-21. Selected MSRs Supported by Intel® Xeon® Processors E5 Family (based on Sandy Bridge microarchitecture)**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
17FH	383	MSR_ERROR_CONTROL	Package	<b>MC Bank Error Configuration (R/W)</b>
		0		Reserved
		1		<b>MemError Log Enable (R/W)</b> When set, enables IMC status bank to log additional info in bits 36:32.
		63:2		Reserved.
1ADH	429	MSR_TURBO_RATIO_LIMIT	Package	<b>Maximum Ratio Limit of Turbo Mode</b> RO if MSR_PLATFORM_INFO.[28] = 0, RW if MSR_PLATFORM_INFO.[28] = 1
		7:0	Package	<b>Maximum Ratio Limit for 1C</b> Maximum turbo ratio limit of 1 core active.
		15:8	Package	<b>Maximum Ratio Limit for 2C</b> Maximum turbo ratio limit of 2 core active.
		23:16	Package	<b>Maximum Ratio Limit for 3C</b> Maximum turbo ratio limit of 3 core active.
		31:24	Package	<b>Maximum Ratio Limit for 4C</b> Maximum turbo ratio limit of 4 core active.
		39:32	Package	<b>Maximum Ratio Limit for 5C</b> Maximum turbo ratio limit of 5 core active.

**Table 35-21. Selected MSRs Supported by Intel® Xeon® Processors E5 Family (based on Sandy Bridge microarchitecture) (Contd.)**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		47:40	Package	<b>Maximum Ratio Limit for 6C</b> Maximum turbo ratio limit of 6 core active.
		55:48	Package	<b>Maximum Ratio Limit for 7C</b> Maximum turbo ratio limit of 7 core active.
		63:56	Package	<b>Maximum Ratio Limit for 8C</b> Maximum turbo ratio limit of 8 core active.
285H	645	IA32_MC5_CTL2	Package	See Table 35-2.
286H	646	IA32_MC6_CTL2	Package	See Table 35-2.
287H	647	IA32_MC7_CTL2	Package	See Table 35-2.
288H	648	IA32_MC8_CTL2	Package	See Table 35-2.
289H	649	IA32_MC9_CTL2	Package	See Table 35-2.
28AH	650	IA32_MC10_CTL2	Package	See Table 35-2.
28BH	651	IA32_MC11_CTL2	Package	See Table 35-2.
28CH	652	IA32_MC12_CTL2	Package	See Table 35-2.
28DH	653	IA32_MC13_CTL2	Package	See Table 35-2.
28EH	654	IA32_MC14_CTL2	Package	See Table 35-2.
28FH	655	IA32_MC15_CTL2	Package	See Table 35-2.
290H	656	IA32_MC16_CTL2	Package	See Table 35-2.
291H	657	IA32_MC17_CTL2	Package	See Table 35-2.
292H	658	IA32_MC18_CTL2	Package	See Table 35-2.
293H	659	IA32_MC19_CTL2	Package	See Table 35-2.
39CH	924	MSR_PEBS_NUM_ALT	Package	
		0		<b>ENABLE_PEBS_NUM_ALT (RW)</b> Write 1 to enable alternate PEBS counting logic for specific events requiring additional configuration, see Table 19-9
		63:1		Reserved (must be zero).
414H	1044	IA32_MC5_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
415H	1045	IA32_MC5_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 16.
416H	1046	IA32_MC5_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
417H	1047	IA32_MC5_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
418H	1048	IA32_MC6_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
419H	1049	IA32_MC6_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 16.
41AH	1050	IA32_MC6_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
41BH	1051	IA32_MC6_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
41CH	1052	IA32_MC7_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
41DH	1053	IA32_MC7_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 16.
41EH	1054	IA32_MC7_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."

**Table 35-21. Selected MSRs Supported by Intel® Xeon® Processors E5 Family (based on Sandy Bridge microarchitecture) (Contd.)**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
41FH	1055	IA32_MC7_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
420H	1056	IA32_MC8_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
421H	1057	IA32_MC8_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 16.
422H	1058	IA32_MC8_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
423H	1059	IA32_MC8_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
424H	1060	IA32_MC9_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
425H	1061	IA32_MC9_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 16.
426H	1062	IA32_MC9_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
427H	1063	IA32_MC9_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
428H	1064	IA32_MC10_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
429H	1065	IA32_MC10_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 16.
42AH	1066	IA32_MC10_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
42BH	1067	IA32_MC10_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
42CH	1068	IA32_MC11_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
42DH	1069	IA32_MC11_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 16.
42EH	1070	IA32_MC11_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
42FH	1071	IA32_MC11_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
430H	1072	IA32_MC12_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
431H	1073	IA32_MC12_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 16.
432H	1074	IA32_MC12_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
433H	1075	IA32_MC12_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
434H	1076	IA32_MC13_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
435H	1077	IA32_MC13_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 16.
436H	1078	IA32_MC13_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
437H	1079	IA32_MC13_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
438H	1080	IA32_MC14_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
439H	1081	IA32_MC14_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 16.
43AH	1082	IA32_MC14_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
43BH	1083	IA32_MC14_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
43CH	1084	IA32_MC15_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
43DH	1085	IA32_MC15_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 16.
43EH	1086	IA32_MC15_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
43FH	1087	IA32_MC15_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
440H	1088	IA32_MC16_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
441H	1089	IA32_MC16_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 16.
442H	1090	IA32_MC16_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."

**Table 35-21. Selected MSRs Supported by Intel® Xeon® Processors E5 Family (based on Sandy Bridge microarchitecture) (Contd.)**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
443H	1091	IA32_MC16_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
444H	1092	IA32_MC17_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
445H	1093	IA32_MC17_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 16.
446H	1094	IA32_MC17_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
447H	1095	IA32_MC17_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
448H	1096	IA32_MC18_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
449H	1097	IA32_MC18_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 16.
44AH	1098	IA32_MC18_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
44BH	1099	IA32_MC18_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
44CH	1100	IA32_MC19_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
44DH	1101	IA32_MC19_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 16.
44EH	1102	IA32_MC19_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
44FH	1103	IA32_MC19_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
613H	1555	MSR_PKG_PERF_STATUS	Package	<b>Package RAPL Perf Status (R/O)</b>
618H	1560	MSR_DRAM_POWER_LIMIT	Package	<b>DRAM RAPL Power Limit Control (R/W)</b> See Section 14.9.5, "DRAM RAPL Domain."
619H	1561	MSR_DRAM_ENERGY_STATUS	Package	<b>DRAM Energy Status (R/O)</b> See Section 14.9.5, "DRAM RAPL Domain."
61BH	1563	MSR_DRAM_PERF_STATUS	Package	<b>DRAM Performance Throttling Status (R/O)</b> See Section 14.9.5, "DRAM RAPL Domain."
61CH	1564	MSR_DRAM_POWER_INFO	Package	<b>DRAM RAPL Parameters (R/W)</b> See Section 14.9.5, "DRAM RAPL Domain."
639H	1593	MSR_PPO_ENERGY_STATUS	Package	<b>PPO Energy Status (R/O)</b> See Section 14.9.4, "PPO/PP1 RAPL Domains."

See Table 35-18, Table 35-21, and Table 35-22 for MSR definitions applicable to processors with CPUID signature 06\_2DH.

### 35.9.3 Additional Uncore PMU MSRs in the Intel® Xeon® Processor E5 Family

Intel Xeon Processor E5 family is based on the Sandy Bridge microarchitecture. The MSR-based uncore PMU interfaces are listed in Table 35-22. For complete detail of the uncore PMU, refer to Intel Xeon Processor E5 Product Family Uncore Performance Monitoring Guide. These processors have a CPUID signature with DisplayFamily\_DisplayModel of 06\_2DH

**Table 35-22. Uncore PMU MSRs in Intel® Xeon® Processor E5 Family**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
C08H		MSR_U_PMON_UCLK_FIXED_CTL	Package	Uncore U-box UCLK fixed counter control
C09H		MSR_U_PMON_UCLK_FIXED_CTR	Package	Uncore U-box UCLK fixed counter

**Table 35-22. Uncore PMU MSRs in Intel® Xeon® Processor E5 Family (Contd.)**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
C10H		MSR_U_PMON_EVNTSELO	Package	Uncore U-box perfmon event select for U-box counter 0.
C11H		MSR_U_PMON_EVNTSEL1	Package	Uncore U-box perfmon event select for U-box counter 1.
C16H		MSR_U_PMON_CTR0	Package	Uncore U-box perfmon counter 0
C17H		MSR_U_PMON_CTR1	Package	Uncore U-box perfmon counter 1
C24H		MSR_PCU_PMON_BOX_CTL	Package	Uncore PCU perfmon for PCU-box-wide control
C30H		MSR_PCU_PMON_EVNTSELO	Package	Uncore PCU perfmon event select for PCU counter 0.
C31H		MSR_PCU_PMON_EVNTSEL1	Package	Uncore PCU perfmon event select for PCU counter 1.
C32H		MSR_PCU_PMON_EVNTSEL2	Package	Uncore PCU perfmon event select for PCU counter 2.
C33H		MSR_PCU_PMON_EVNTSEL3	Package	Uncore PCU perfmon event select for PCU counter 3.
C34H		MSR_PCU_PMON_BOX_FILTER	Package	Uncore PCU perfmon box-wide filter.
C36H		MSR_PCU_PMON_CTR0	Package	Uncore PCU perfmon counter 0.
C37H		MSR_PCU_PMON_CTR1	Package	Uncore PCU perfmon counter 1.
C38H		MSR_PCU_PMON_CTR2	Package	Uncore PCU perfmon counter 2.
C39H		MSR_PCU_PMON_CTR3	Package	Uncore PCU perfmon counter 3.
D04H		MSR_C0_PMON_BOX_CTL	Package	Uncore C-box 0 perfmon local box wide control.
D10H		MSR_C0_PMON_EVNTSELO	Package	Uncore C-box 0 perfmon event select for C-box 0 counter 0.
D11H		MSR_C0_PMON_EVNTSEL1	Package	Uncore C-box 0 perfmon event select for C-box 0 counter 1.
D12H		MSR_C0_PMON_EVNTSEL2	Package	Uncore C-box 0 perfmon event select for C-box 0 counter 2.
D13H		MSR_C0_PMON_EVNTSEL3	Package	Uncore C-box 0 perfmon event select for C-box 0 counter 3.
D14H		MSR_C0_PMON_BOX_FILTER	Package	Uncore C-box 0 perfmon box wide filter.
D16H		MSR_C0_PMON_CTR0	Package	Uncore C-box 0 perfmon counter 0.
D17H		MSR_C0_PMON_CTR1	Package	Uncore C-box 0 perfmon counter 1.
D18H		MSR_C0_PMON_CTR2	Package	Uncore C-box 0 perfmon counter 2.
D19H		MSR_C0_PMON_CTR3	Package	Uncore C-box 0 perfmon counter 3.
D24H		MSR_C1_PMON_BOX_CTL	Package	Uncore C-box 1 perfmon local box wide control.
D30H		MSR_C1_PMON_EVNTSELO	Package	Uncore C-box 1 perfmon event select for C-box 1 counter 0.
D31H		MSR_C1_PMON_EVNTSEL1	Package	Uncore C-box 1 perfmon event select for C-box 1 counter 1.
D32H		MSR_C1_PMON_EVNTSEL2	Package	Uncore C-box 1 perfmon event select for C-box 1 counter 2.
D33H		MSR_C1_PMON_EVNTSEL3	Package	Uncore C-box 1 perfmon event select for C-box 1 counter 3.
D34H		MSR_C1_PMON_BOX_FILTER	Package	Uncore C-box 1 perfmon box wide filter.
D36H		MSR_C1_PMON_CTR0	Package	Uncore C-box 1 perfmon counter 0.
D37H		MSR_C1_PMON_CTR1	Package	Uncore C-box 1 perfmon counter 1.
D38H		MSR_C1_PMON_CTR2	Package	Uncore C-box 1 perfmon counter 2.
D39H		MSR_C1_PMON_CTR3	Package	Uncore C-box 1 perfmon counter 3.
D44H		MSR_C2_PMON_BOX_CTL	Package	Uncore C-box 2 perfmon local box wide control.
D50H		MSR_C2_PMON_EVNTSELO	Package	Uncore C-box 2 perfmon event select for C-box 2 counter 0.
D51H		MSR_C2_PMON_EVNTSEL1	Package	Uncore C-box 2 perfmon event select for C-box 2 counter 1.

Table 35-22. Uncore PMU MSRs in Intel® Xeon® Processor E5 Family (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
D52H		MSR_C2_PMON_EVTSEL2	Package	Uncore C-box 2 perfmon event select for C-box 2 counter 2.
D53H		MSR_C2_PMON_EVTSEL3	Package	Uncore C-box 2 perfmon event select for C-box 2 counter 3.
D54H		MSR_C2_PMON_BOX_FILTER	Package	Uncore C-box 2 perfmon box wide filter.
D56H		MSR_C2_PMON_CTR0	Package	Uncore C-box 2 perfmon counter 0.
D57H		MSR_C2_PMON_CTR1	Package	Uncore C-box 2 perfmon counter 1.
D58H		MSR_C2_PMON_CTR2	Package	Uncore C-box 2 perfmon counter 2.
D59H		MSR_C2_PMON_CTR3	Package	Uncore C-box 2 perfmon counter 3.
D64H		MSR_C3_PMON_BOX_CTL	Package	Uncore C-box 3 perfmon local box wide control.
D70H		MSR_C3_PMON_EVTSEL0	Package	Uncore C-box 3 perfmon event select for C-box 3 counter 0.
D71H		MSR_C3_PMON_EVTSEL1	Package	Uncore C-box 3 perfmon event select for C-box 3 counter 1.
D72H		MSR_C3_PMON_EVTSEL2	Package	Uncore C-box 3 perfmon event select for C-box 3 counter 2.
D73H		MSR_C3_PMON_EVTSEL3	Package	Uncore C-box 3 perfmon event select for C-box 3 counter 3.
D74H		MSR_C3_PMON_BOX_FILTER	Package	Uncore C-box 3 perfmon box wide filter.
D76H		MSR_C3_PMON_CTR0	Package	Uncore C-box 3 perfmon counter 0.
D77H		MSR_C3_PMON_CTR1	Package	Uncore C-box 3 perfmon counter 1.
D78H		MSR_C3_PMON_CTR2	Package	Uncore C-box 3 perfmon counter 2.
D79H		MSR_C3_PMON_CTR3	Package	Uncore C-box 3 perfmon counter 3.
D84H		MSR_C4_PMON_BOX_CTL	Package	Uncore C-box 4 perfmon local box wide control.
D90H		MSR_C4_PMON_EVTSEL0	Package	Uncore C-box 4 perfmon event select for C-box 4 counter 0.
D91H		MSR_C4_PMON_EVTSEL1	Package	Uncore C-box 4 perfmon event select for C-box 4 counter 1.
D92H		MSR_C4_PMON_EVTSEL2	Package	Uncore C-box 4 perfmon event select for C-box 4 counter 2.
D93H		MSR_C4_PMON_EVTSEL3	Package	Uncore C-box 4 perfmon event select for C-box 4 counter 3.
D94H		MSR_C4_PMON_BOX_FILTER	Package	Uncore C-box 4 perfmon box wide filter.
D96H		MSR_C4_PMON_CTR0	Package	Uncore C-box 4 perfmon counter 0.
D97H		MSR_C4_PMON_CTR1	Package	Uncore C-box 4 perfmon counter 1.
D98H		MSR_C4_PMON_CTR2	Package	Uncore C-box 4 perfmon counter 2.
D99H		MSR_C4_PMON_CTR3	Package	Uncore C-box 4 perfmon counter 3.
DA4H		MSR_C5_PMON_BOX_CTL	Package	Uncore C-box 5 perfmon local box wide control.
DB0H		MSR_C5_PMON_EVTSEL0	Package	Uncore C-box 5 perfmon event select for C-box 5 counter 0.
DB1H		MSR_C5_PMON_EVTSEL1	Package	Uncore C-box 5 perfmon event select for C-box 5 counter 1.
DB2H		MSR_C5_PMON_EVTSEL2	Package	Uncore C-box 5 perfmon event select for C-box 5 counter 2.
DB3H		MSR_C5_PMON_EVTSEL3	Package	Uncore C-box 5 perfmon event select for C-box 5 counter 3.
DB4H		MSR_C5_PMON_BOX_FILTER	Package	Uncore C-box 5 perfmon box wide filter.
DB6H		MSR_C5_PMON_CTR0	Package	Uncore C-box 5 perfmon counter 0.
DB7H		MSR_C5_PMON_CTR1	Package	Uncore C-box 5 perfmon counter 1.
DB8H		MSR_C5_PMON_CTR2	Package	Uncore C-box 5 perfmon counter 2.
DB9H		MSR_C5_PMON_CTR3	Package	Uncore C-box 5 perfmon counter 3.

**Table 35-22. Uncore PMU MSRs in Intel® Xeon® Processor E5 Family (Contd.)**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
DC4H		MSR_C6_PMON_BOX_CTL	Package	Uncore C-box 6 perfmon local box wide control.
DD0H		MSR_C6_PMON_EVTSEL0	Package	Uncore C-box 6 perfmon event select for C-box 6 counter 0.
DD1H		MSR_C6_PMON_EVTSEL1	Package	Uncore C-box 6 perfmon event select for C-box 6 counter 1.
DD2H		MSR_C6_PMON_EVTSEL2	Package	Uncore C-box 6 perfmon event select for C-box 6 counter 2.
DD3H		MSR_C6_PMON_EVTSEL3	Package	Uncore C-box 6 perfmon event select for C-box 6 counter 3.
DD4H		MSR_C6_PMON_BOX_FILTER	Package	Uncore C-box 6 perfmon box wide filter.
DD6H		MSR_C6_PMON_CTR0	Package	Uncore C-box 6 perfmon counter 0.
DD7H		MSR_C6_PMON_CTR1	Package	Uncore C-box 6 perfmon counter 1.
DD8H		MSR_C6_PMON_CTR2	Package	Uncore C-box 6 perfmon counter 2.
DD9H		MSR_C6_PMON_CTR3	Package	Uncore C-box 6 perfmon counter 3.
DE4H		MSR_C7_PMON_BOX_CTL	Package	Uncore C-box 7 perfmon local box wide control.
DF0H		MSR_C7_PMON_EVTSEL0	Package	Uncore C-box 7 perfmon event select for C-box 7 counter 0.
DF1H		MSR_C7_PMON_EVTSEL1	Package	Uncore C-box 7 perfmon event select for C-box 7 counter 1.
DF2H		MSR_C7_PMON_EVTSEL2	Package	Uncore C-box 7 perfmon event select for C-box 7 counter 2.
DF3H		MSR_C7_PMON_EVTSEL3	Package	Uncore C-box 7 perfmon event select for C-box 7 counter 3.
DF4H		MSR_C7_PMON_BOX_FILTER	Package	Uncore C-box 7 perfmon box wide filter.
DF6H		MSR_C7_PMON_CTR0	Package	Uncore C-box 7 perfmon counter 0.
DF7H		MSR_C7_PMON_CTR1	Package	Uncore C-box 7 perfmon counter 1.
DF8H		MSR_C7_PMON_CTR2	Package	Uncore C-box 7 perfmon counter 2.
DF9H		MSR_C7_PMON_CTR3	Package	Uncore C-box 7 perfmon counter 3.

### 35.10 MSRS IN THE 3RD GENERATION INTEL® CORE™ PROCESSOR FAMILY (BASED ON INTEL® MICROARCHITECTURE CODE NAME IVY BRIDGE)

The 3rd generation Intel® Core™ processor family and the Intel® Xeon® processor E3-1200v2 product family (based on Intel microarchitecture code name Ivy Bridge) support the MSR interfaces listed in Table 35-18, Table 35-19, Table 35-20, and Table 35-23. These processors have a CPUID signature with DisplayFamily\_DisplayModel of 06\_3AH.

**Table 35-23. Additional MSRs Supported by 3rd Generation Intel® Core™ Processors (based on Intel® microarchitecture code name Ivy Bridge)**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
CEH	206	MSR_PLATFORM_INFO	Package	See <a href="http://biosbits.org">http://biosbits.org</a> .
		7:0		Reserved.
		15:8	Package	<b>Maximum Non-Turbo Ratio (R/O)</b> The is the ratio of the frequency that invariant TSC runs at. Frequency = ratio * 100 MHz.



**Table 35-23. Additional MSRs Supported by 3rd Generation Intel® Core™ Processors (based on Intel® microarchitecture code name Ivy Bridge) (Contd.)**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		27:16		Reserved.
		28	Package	<b>Programmable Ratio Limit for Turbo Mode (R/O)</b> When set to 1, indicates that Programmable Ratio Limits for Turbo mode is enabled, and when set to 0, indicates Programmable Ratio Limits for Turbo mode is disabled.
		29	Package	<b>Programmable TDP Limit for Turbo Mode (R/O)</b> When set to 1, indicates that TDP Limits for Turbo mode are programmable, and when set to 0, indicates TDP Limit for Turbo mode is not programmable.
		31:30		Reserved.
		32	Package	<b>Low Power Mode Support (LPM) (R/O)</b> When set to 1, indicates that LPM is supported, and when set to 0, indicates LPM is not supported.
		34:33	Package	<b>Number of ConfigTDP Levels (R/O)</b> 00: Only Base TDP level available. 01: One additional TDP level available. 02: Two additional TDP level available. 11: Reserved
		39:35		Reserved.
		47:40	Package	<b>Maximum Efficiency Ratio (R/O)</b> The is the minimum ratio (maximum efficiency) that the processor can operates, in units of 100MHz.
		55:48	Package	<b>Minimum Operating Ratio (R/O)</b> Contains the minimum supported operating ratio in units of 100 MHz.
		63:56		Reserved.
E2H	226	MSR_PKG_CST_CONFIG_CONTROL	Core	<b>C-State Configuration Control (R/W)</b> Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States. See <a href="http://biosbits.org">http://biosbits.org</a> .



**Table 35-23. Additional MSRs Supported by 3rd Generation Intel® Core™ Processors (based on Intel® microarchitecture code name Ivy Bridge) (Contd.)**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		2:0		<b>Package C-State Limit (R/W)</b> Specifies the lowest processor-specific C-state code name (consuming the least power), for the package. The default is set as factory-configured package C-state limit. The following C-state code name encodings are supported: 000b: C0/C1 (no package C-state support) 001b: C2 010b: C6 no retention 011b: C6 retention 100b: C7 101b: C7s 111: No package C-state limit. Note: This field cannot be used to limit package C-state to C3.
		9:3		Reserved.
		10		<b>I/O MWAIT Redirection Enable (R/W)</b> When set, will map IO_read instructions sent to IO register specified by MSR_PMG_IO_CAPTURE_BASE to MWAIT instructions
		14:11		Reserved.
		15		<b>CFG Lock (R/WO)</b> When set, lock bits 15:0 of this register until next reset.
		24:16		Reserved.
		25		<b>C3 state auto demotion enable (R/W)</b> When set, the processor will conditionally demote C6/C7 requests to C3 based on uncore auto-demote information.
		26		<b>C1 state auto demotion enable (R/W)</b> When set, the processor will conditionally demote C3/C6/C7 requests to C1 based on uncore auto-demote information.
		27		<b>Enable C3 undemotion (R/W)</b> When set, enables undemotion from demoted C3.
		28		<b>Enable C1 undemotion (R/W)</b> When set, enables undemotion from demoted C1.
		63:29		Reserved.
639H	1593	MSR_PPO_ENERGY_STATUS	Package	<b>PPO Energy Status (R/O)</b> See Section 14.9.4, "PPO/PP1 RAPL Domains."
648H	1608	MSR_CONFIG_TDP_NOMINAL	Package	<b>Base TDP Ratio (R/O)</b>
		7:0		<b>Config_TDP_Base</b> Base TDP level ratio to be used for this specific processor (in units of 100 MHz).
		63:8		Reserved.

**Table 35-23. Additional MSRs Supported by 3rd Generation Intel® Core™ Processors (based on Intel® microarchitecture code name Ivy Bridge) (Contd.)**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
649H	1609	MSR_CONFIG_TDP_LEVEL1	Package	ConfigTDP Level 1 ratio and power level (R/O)
		14:0		PKG_TDP_LVL1. Power setting for ConfigTDP Level 1.
		15		Reserved
		23:16		Config_TDP_LVL1_Ratio. ConfigTDP level 1 ratio to be used for this specific processor.
		31:24		Reserved
		46:32		PKG_MAX_PWR_LVL1. Max Power setting allowed for ConfigTDP Level 1.
		47		Reserved
		62:48		PKG_MIN_PWR_LVL1. MIN Power setting allowed for ConfigTDP Level 1.
		63		Reserved.
64AH	1610	MSR_CONFIG_TDP_LEVEL2	Package	ConfigTDP Level 2 ratio and power level (R/O)
		14:0		PKG_TDP_LVL2. Power setting for ConfigTDP Level 2.
		15		Reserved
		23:16		Config_TDP_LVL2_Ratio. ConfigTDP level 2 ratio to be used for this specific processor.
		31:24		Reserved
		46:32		PKG_MAX_PWR_LVL2. Max Power setting allowed for ConfigTDP Level 2.
		47		Reserved
		62:48		PKG_MIN_PWR_LVL2. MIN Power setting allowed for ConfigTDP Level 2.
		63		Reserved.
64BH	1611	MSR_CONFIG_TDP_CONTROL	Package	<b>ConfigTDP Control (R/W)</b>
		1:0		<b>TDP_LEVEL (RW/L)</b> System BIOS can program this field.
		30:2		Reserved.
		31		<b>Config_TDP_Lock (RW/L)</b> When this bit is set, the content of this register is locked until a reset.
		63:32		Reserved.
64CH	1612	MSR_TURBO_ACTIVATION_RATIO	Package	<b>ConfigTDP Control (R/W)</b>
		7:0		<b>MAX_NON_TURBO_RATIO (RW/L)</b> System BIOS can program this field.
		30:8		Reserved.

**Table 35-23. Additional MSRs Supported by 3rd Generation Intel® Core™ Processors (based on Intel® microarchitecture code name Ivy Bridge) (Contd.)**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		31		<b>TURBO_ACTIVATION_RATIO_Lock (Rw/L)</b> When this bit is set, the content of this register is locked until a reset.
		63:32		Reserved.
See Table 35-18, Table 35-19 and Table 35-20 for other MSR definitions applicable to processors with CPUID signature 06_3AH				

### 35.10.1 MSRs In Intel® Xeon® Processor E5 v2 Product Family (Based on Ivy Bridge-E Microarchitecture)

Table 35-24 lists model-specific registers (MSRs) that are specific to the Intel® Xeon® Processor E5 v2 Product Family (based on Ivy Bridge-E microarchitecture). These processors have a CPUID signature with DisplayFamily\_DisplayModel of 06\_3EH, see Table 35-1. These processors supports the MSR interfaces listed in Table 35-18, and Table 35-24.

**Table 35-24. MSRs Supported by Intel® Xeon® Processors E5 v2 Product Family (based on Ivy Bridge-E microarchitecture)**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
4EH	78	MSR_PPIN_CTL	Package	Protected Processor Inventory Number Enable Control (R/W)
		0		LockOut (R/W0) Set 1 to prevent further writes to MSR_PPIN_CTL. Writing 1 to MSR_PPIN_CTL[bit 0] is permitted only if MSR_PPIN_CTL[bit 1] is clear, Default is 0. BIOS should provide an opt-in menu to enable the user to turn on MSR_PPIN_CTL[bit 1] for privileged inventory initialization agent to access MSR_PPIN. After reading MSR_PPIN, the privileged inventory initialization agent should write '01b' to MSR_PPIN_CTL to disable further access to MSR_PPIN and prevent unauthorized modification to MSR_PPIN_CTL.
		1		Enable_PPIN (R/W) If 1, enables MSR_PPIN to be accessible using RDMSR. Once set, attempt to write 1 to MSR_PPIN_CTL[bit 0] will cause #GP. If 0, an attempt to read MSR_PPIN will cause #GP. Default is 0.
		63:2		Reserved.
4FH	79	MSR_PPIN	Package	Protected Processor Inventory Number (R/O)

**Table 35-24. MSRs Supported by Intel® Xeon® Processors E5 v2 Product Family (based on Ivy Bridge-E microarchitecture) (Contd.)**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		63:0		<b>Protected Processor Inventory Number (R/O)</b> A unique value within a given CUID family/model/stepping signature that a privileged inventory initialization agent can access to identify each physical processor, when access to MSR_PPIN is enabled. Access to MSR_PPIN is permitted only if MSR_PPIN_CTL[bits 1:0] = '10b'
CEH	206	MSR_PLATFORM_INFO	Package	See <a href="http://biosbits.org">http://biosbits.org</a> .
		7:0		Reserved.
		15:8	Package	<b>Maximum Non-Turbo Ratio (R/O)</b> The is the ratio of the frequency that invariant TSC runs at. Frequency = ratio * 100 MHz.
		22:16		Reserved.
		23	Package	<b>PPIN_CAP (R/O)</b> When set to 1, indicates that Protected Processor Inventory Number (PPIN) capability can be enabled for privileged system inventory agent to read PPIN from MSR_PPIN. When set to 0, PPIN capability is not supported. An attempt to access MSR_PPIN_CTL or MSR_PPIN will cause #GP.
		27:24		Reserved.
		28	Package	<b>Programmable Ratio Limit for Turbo Mode (R/O)</b> When set to 1, indicates that Programmable Ratio Limits for Turbo mode is enabled, and when set to 0, indicates Programmable Ratio Limits for Turbo mode is disabled.
		29	Package	<b>Programmable TDP Limit for Turbo Mode (R/O)</b> When set to 1, indicates that TDP Limits for Turbo mode are programmable, and when set to 0, indicates TDP Limit for Turbo mode is not programmable.
		30	Package	<b>Programmable TJ OFFSET (R/O)</b> When set to 1, indicates that MSR_TEMPERATURE_TARGET.[27:24] is valid and writable to specify an temperature offset.
		39:31		Reserved.
		47:40	Package	<b>Maximum Efficiency Ratio (R/O)</b> The is the minimum ratio (maximum efficiency) that the processor can operates, in units of 100MHz.
		63:48		Reserved.
E2H	226	MSR_PKG_CST_CONFIG_CONTROL	Core	<b>C-State Configuration Control (R/W)</b> Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States. See <a href="http://biosbits.org">http://biosbits.org</a> .

**Table 35-24. MSRs Supported by Intel® Xeon® Processors E5 v2 Product Family (based on Ivy Bridge-E microarchitecture) (Contd.)**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		2:0		<b>Package C-State Limit (R/W)</b> Specifies the lowest processor-specific C-state code name (consuming the least power), for the package. The default is set as factory-configured package C-state limit. The following C-state code name encodings are supported: 000b: C0/C1 (no package C-state support) 001b: C2 010b: C6 no retention 011b: C6 retention 100b: C7 101b: C7s 111: No package C-state limit. Note: This field cannot be used to limit package C-state to C3.
		9:3		Reserved.
		10		<b>I/O MWAIT Redirection Enable (R/W)</b> When set, will map IO_read instructions sent to IO register specified by MSR_PMG_IO_CAPTURE_BASE to MWAIT instructions
		14:11		Reserved.
		15		<b>CFG Lock (R/W0)</b> When set, lock bits 15:0 of this register until next reset.
		63:16		Reserved.
179H	377	IA32_MCG_CAP	Thread	<b>Global Machine Check Capability (R/O)</b>
		7:0		<b>Count</b>
		8		<b>MCG_CTL_P</b>
		9		<b>MCG_EXT_P</b>
		10		<b>MCP_CMCI_P</b>
		11		<b>MCG_TES_P</b>
		15:12		Reserved.
		23:16		<b>MCG_EXT_CNT</b>
		24		<b>MCG_SER_P</b>
		25		Reserved.
		26		<b>MCG_ELOG_P</b>
63:27		Reserved.		
17FH	383	MSR_ERROR_CONTROL	Package	<b>MC Bank Error Configuration (R/W)</b>
		0		Reserved
		1		<b>MemError Log Enable (R/W)</b> When set, enables IMC status bank to log additional info in bits 36:32.
		63:2		Reserved.

**Table 35-24. MSRs Supported by Intel® Xeon® Processors E5 v2 Product Family (based on Ivy Bridge-E microarchitecture) (Contd.)**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
1A2H	418	MSR_TEMPERATURE_TARGET	Package	
		15:0		Reserved.
		23:16		<b>Temperature Target (RO)</b> The minimum temperature at which PROCHOT# will be asserted. The value is degree C.
		27:24		<b>TCC Activation Offset (R/W)</b> Specifies a temperature offset in degrees C from the temperature target (bits 23:16). PROCHOT# will assert at the offset target temperature. Write is permitted only MSR_PLATFORM_INFO.[30] is set.
		63:28		Reserved.
1AEH	430	MSR_TURBO_RATIO_LIMIT_1	Package	<b>Maximum Ratio Limit of Turbo Mode</b> RO if MSR_PLATFORM_INFO.[28] = 0, RW if MSR_PLATFORM_INFO.[28] = 1
		7:0	Package	<b>Maximum Ratio Limit for 9C</b> Maximum turbo ratio limit of 9 core active.
		15:8	Package	<b>Maximum Ratio Limit for 10C</b> Maximum turbo ratio limit of 10core active.
		23:16	Package	<b>Maximum Ratio Limit for 11C</b> Maximum turbo ratio limit of 11 core active.
		31:24	Package	<b>Maximum Ratio Limit for 12C</b> Maximum turbo ratio limit of 12 core active.
		63:32		Reserved
285H	645	IA32_MC5_CTL2	Package	See Table 35-2.
286H	646	IA32_MC6_CTL2	Package	See Table 35-2.
287H	647	IA32_MC7_CTL2	Package	See Table 35-2.
288H	648	IA32_MC8_CTL2	Package	See Table 35-2.
289H	649	IA32_MC9_CTL2	Package	See Table 35-2.
28AH	650	IA32_MC10_CTL2	Package	See Table 35-2.
28BH	651	IA32_MC11_CTL2	Package	See Table 35-2.
28CH	652	IA32_MC12_CTL2	Package	See Table 35-2.
28DH	653	IA32_MC13_CTL2	Package	See Table 35-2.
28EH	654	IA32_MC14_CTL2	Package	See Table 35-2.
28FH	655	IA32_MC15_CTL2	Package	See Table 35-2.
290H	656	IA32_MC16_CTL2	Package	See Table 35-2.
291H	657	IA32_MC17_CTL2	Package	See Table 35-2.
292H	658	IA32_MC18_CTL2	Package	See Table 35-2.
293H	659	IA32_MC19_CTL2	Package	See Table 35-2.

**Table 35-24. MSRs Supported by Intel® Xeon® Processors E5 v2 Product Family (based on Ivy Bridge-E microarchitecture) (Contd.)**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
294H	660	IA32_MC20_CTL2	Package	See Table 35-2.
295H	661	IA32_MC21_CTL2	Package	See Table 35-2.
296H	662	IA32_MC22_CTL2	Package	See Table 35-2.
297H	663	IA32_MC23_CTL2	Package	See Table 35-2.
298H	664	IA32_MC24_CTL2	Package	See Table 35-2.
299H	665	IA32_MC25_CTL2	Package	See Table 35-2.
29AH	666	IA32_MC26_CTL2	Package	See Table 35-2.
29BH	667	IA32_MC27_CTL2	Package	See Table 35-2.
29CH	668	IA32_MC28_CTL2	Package	See Table 35-2.
414H	1044	IA32_MC5_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Bank MC5 reports MC error from the Intel QPI module.
415H	1045	IA32_MC5_STATUS	Package	
416H	1046	IA32_MC5_ADDR	Package	
417H	1047	IA32_MC5_MISC	Package	
418H	1048	IA32_MC6_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Bank MC6 reports MC error from the integrated I/O module.
419H	1049	IA32_MC6_STATUS	Package	
41AH	1050	IA32_MC6_ADDR	Package	
41BH	1051	IA32_MC6_MISC	Package	
41CH	1052	IA32_MC7_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Banks MC7 and MC 8 report MC error from the two home agents.
41DH	1053	IA32_MC7_STATUS	Package	
41EH	1054	IA32_MC7_ADDR	Package	
41FH	1055	IA32_MC7_MISC	Package	
420H	1056	IA32_MC8_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Banks MC7 and MC 8 report MC error from the two home agents.
421H	1057	IA32_MC8_STATUS	Package	
422H	1058	IA32_MC8_ADDR	Package	
423H	1059	IA32_MC8_MISC	Package	
424H	1060	IA32_MC9_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Banks MC9 through MC 16 report MC error from each channel of the integrated memory controllers.
425H	1061	IA32_MC9_STATUS	Package	
426H	1062	IA32_MC9_ADDR	Package	
427H	1063	IA32_MC9_MISC	Package	
428H	1064	IA32_MC10_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Banks MC9 through MC 16 report MC error from each channel of the integrated memory controllers.
429H	1065	IA32_MC10_STATUS	Package	
42AH	1066	IA32_MC10_ADDR	Package	
42BH	1067	IA32_MC10_MISC	Package	
42CH	1068	IA32_MC11_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."

**Table 35-24. MSRs Supported by Intel® Xeon® Processors E5 v2 Product Family (based on Ivy Bridge-E microarchitecture) (Contd.)**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
42DH	1069	IA32_MC11_STATUS	Package	Bank MC11 reports MC error from a specific channel of the integrated memory controller.
42EH	1070	IA32_MC11_ADDR	Package	
42FH	1071	IA32_MC11_MISC	Package	
430H	1072	IA32_MC12_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Banks MC9 through MC 16 report MC error from each channel of the integrated memory controllers.
431H	1073	IA32_MC12_STATUS	Package	
432H	1074	IA32_MC12_ADDR	Package	
433H	1075	IA32_MC12_MISC	Package	
434H	1076	IA32_MC13_CTL	Package	
435H	1077	IA32_MC13_STATUS	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Banks MC9 through MC 16 report MC error from each channel of the integrated memory controllers.
436H	1078	IA32_MC13_ADDR	Package	
437H	1079	IA32_MC13_MISC	Package	
438H	1080	IA32_MC14_CTL	Package	
439H	1081	IA32_MC14_STATUS	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Banks MC9 through MC 16 report MC error from each channel of the integrated memory controllers.
43AH	1082	IA32_MC14_ADDR	Package	
43BH	1083	IA32_MC14_MISC	Package	
43CH	1084	IA32_MC15_CTL	Package	
43DH	1085	IA32_MC15_STATUS	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Banks MC9 through MC 16 report MC error from each channel of the integrated memory controllers.
43EH	1086	IA32_MC15_ADDR	Package	
43FH	1087	IA32_MC15_MISC	Package	
440H	1088	IA32_MC16_CTL	Package	
441H	1089	IA32_MC16_STATUS	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Banks MC9 through MC 16 report MC error from each channel of the integrated memory controllers.
442H	1090	IA32_MC16_ADDR	Package	
443H	1091	IA32_MC16_MISC	Package	
444H	1092	IA32_MC17_CTL	Package	
445H	1093	IA32_MC17_STATUS	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Bank MC17 reports MC error from a specific CBo (core broadcast) and its corresponding slice of L3.
446H	1094	IA32_MC17_ADDR	Package	
447H	1095	IA32_MC17_MISC	Package	
448H	1096	IA32_MC18_CTL	Package	
449H	1097	IA32_MC18_STATUS	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Bank MC18 reports MC error from a specific CBo (core broadcast) and its corresponding slice of L3.
44AH	1098	IA32_MC18_ADDR	Package	
44BH	1099	IA32_MC18_MISC	Package	
44CH	1100	IA32_MC19_CTL	Package	
44DH	1101	IA32_MC19_STATUS	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Bank MC19 reports MC error from a specific CBo (core broadcast) and its corresponding slice of L3.
44EH	1102	IA32_MC19_ADDR	Package	
44FH	1103	IA32_MC19_MISC	Package	



**Table 35-24. MSRs Supported by Intel® Xeon® Processors E5 v2 Product Family (based on Ivy Bridge-E microarchitecture) (Contd.)**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
450H	1104	IA32_MC20_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
451H	1105	IA32_MC20_STATUS	Package	Bank MC20 reports MC error from a specific CBo (core broadcast) and its corresponding slice of L3.
452H	1106	IA32_MC20_ADDR	Package	
453H	1107	IA32_MC20_MISC	Package	
454H	1108	IA32_MC21_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Bank MC21 reports MC error from a specific CBo (core broadcast) and its corresponding slice of L3.
455H	1109	IA32_MC21_STATUS	Package	
456H	1110	IA32_MC21_ADDR	Package	
457H	1111	IA32_MC21_MISC	Package	
458H	1112	IA32_MC22_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Bank MC22 reports MC error from a specific CBo (core broadcast) and its corresponding slice of L3.
459H	1113	IA32_MC22_STATUS	Package	
45AH	1114	IA32_MC22_ADDR	Package	
45BH	1115	IA32_MC22_MISC	Package	
45CH	1116	IA32_MC23_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Bank MC23 reports MC error from a specific CBo (core broadcast) and its corresponding slice of L3.
45DH	1117	IA32_MC23_STATUS	Package	
45EH	1118	IA32_MC23_ADDR	Package	
45FH	1119	IA32_MC23_MISC	Package	
460H	1120	IA32_MC24_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Bank MC24 reports MC error from a specific CBo (core broadcast) and its corresponding slice of L3.
461H	1121	IA32_MC24_STATUS	Package	
462H	1122	IA32_MC24_ADDR	Package	
463H	1123	IA32_MC24_MISC	Package	
464H	1124	IA32_MC25_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Bank MC25 reports MC error from a specific CBo (core broadcast) and its corresponding slice of L3.
465H	1125	IA32_MC25_STATUS	Package	
466H	1126	IA32_MC25_ADDR	Package	
467H	1127	IA32_MC2MISC	Package	
468H	1128	IA32_MC26_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Bank MC26 reports MC error from a specific CBo (core broadcast) and its corresponding slice of L3.
469H	1129	IA32_MC26_STATUS	Package	
46AH	1130	IA32_MC26_ADDR	Package	
46BH	1131	IA32_MC26_MISC	Package	
46CH	1132	IA32_MC27_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Bank MC27 reports MC error from a specific CBo (core broadcast) and its corresponding slice of L3.
46DH	1133	IA32_MC27_STATUS	Package	
46EH	1134	IA32_MC27_ADDR	Package	
46FH	1135	IA32_MC27_MISC	Package	

**Table 35-24. MSRs Supported by Intel® Xeon® Processors E5 v2 Product Family (based on Ivy Bridge-E microarchitecture) (Contd.)**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
470H	1136	IA32_MC28_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs:" through Section 15.3.2.4, "IA32_MCi_MISC MSRs:". Bank MC28 reports MC error from a specific CBo (core broadcast) and its corresponding slice of L3.
471H	1137	IA32_MC28_STATUS	Package	
472H	1138	IA32_MC28_ADDR	Package	
473H	1139	IA32_MC28_MISC	Package	
613H	1555	MSR_PKG_PERF_STATUS	Package	<b>Package RAPL Perf Status (R/O)</b>
618H	1560	MSR_DRAM_POWER_LIMIT	Package	<b>DRAM RAPL Power Limit Control (R/W)</b> See Section 14.9.5, "DRAM RAPL Domain."
619H	1561	MSR_DRAM_ENERGY_STATUS	Package	<b>DRAM Energy Status (R/O)</b> See Section 14.9.5, "DRAM RAPL Domain."
61BH	1563	MSR_DRAM_PERF_STATUS	Package	<b>DRAM Performance Throttling Status (R/O)</b> See Section 14.9.5, "DRAM RAPL Domain."
61CH	1564	MSR_DRAM_POWER_INFO	Package	<b>DRAM RAPL Parameters (R/W)</b> See Section 14.9.5, "DRAM RAPL Domain."
639H	1593	MSR_PP0_ENERGY_STATUS	Package	<b>PP0 Energy Status (R/O)</b> See Section 14.9.4, "PP0/PP1 RAPL Domains."

See Table 35-18, for other MSR definitions applicable to Intel Xeon processor E5 v2 with CPUID signature 06\_3EH

### 35.10.2 Additional MSRs Supported by Intel® Xeon® Processor E7 v2 Family

Intel® Xeon® processor E7 v2 family (based on Ivy Bridge-E microarchitecture) with CPUID DisplayFamily\_DisplayModel signature 06\_3EH supports the MSR interfaces listed in Table 35-18, Table 35-24, and Table 35-25.

**Table 35-25. Additional MSRs Supported by Intel® Xeon® Processor E7 v2 Family with DisplayFamily\_DisplayModel Signature 06\_3EH**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
3AH	58	IA32_FEATURE_CONTROL	Thread	<b>Control Features in Intel 64 Processor (R/W)</b> See Table 35-2.
		0		<b>Lock (R/WL)</b>
		1		<b>Enable VMX inside SMX operation (R/WL)</b>
		2		<b>Enable VMX outside SMX operation (R/WL)</b>
		14:8		<b>SENTER local functions enables (R/WL)</b>
		15		<b>SENTER global functions enable (R/WL)</b>
		63:16		Reserved.
179H	377	IA32_MCG_CAP	Thread	<b>Global Machine Check Capability (R/O)</b>
		7:0		<b>Count</b>
		8		<b>MCG_CTL_P</b>

**Table 35-25. Additional MSRs Supported by Intel® Xeon® Processor E7 v2 Family with DisplayFamily\_DisplayModel Signature 06\_3EH**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		9		MCG_EXT_P
		10		MCP_CMCL_P
		11		MCG_TES_P
		15:12		Reserved.
		23:16		MCG_EXT_CNT
		24		MCG_SER_P
		63:25		Reserved.
17AH	378	IA32_MCG_STATUS	Thread	(R/W0)
		0		RIPV
		1		EIPV
		2		MCIP
		3		LMCE signaled
		63:4		Reserved.
1AEH	430	MSR_TURBO_RATIO_LIMIT1	Package	<b>Maximum Ratio Limit of Turbo Mode</b> RO if MSR_PLATFORM_INFO.[28] = 0, RW if MSR_PLATFORM_INFO.[28] = 1
		7:0	Package	<b>Maximum Ratio Limit for 9C</b> Maximum turbo ratio limit of 9 core active.
		15:8	Package	<b>Maximum Ratio Limit for 10C</b> Maximum turbo ratio limit of 10core active.
		23:16	Package	<b>Maximum Ratio Limit for 11C</b> Maximum turbo ratio limit of 11 core active.
		31:24	Package	<b>Maximum Ratio Limit for 12C</b> Maximum turbo ratio limit of 12 core active.
		39:32	Package	<b>Maximum Ratio Limit for 13C</b> Maximum turbo ratio limit of 13 core active.
		47:40	Package	<b>Maximum Ratio Limit for 14C</b> Maximum turbo ratio limit of 14 core active.
		55:48	Package	<b>Maximum Ratio Limit for 15C</b> Maximum turbo ratio limit of 15 core active.
		62:56		Reserved
		63	Package	<b>Semaphore for Turbo Ratio Limit Configuration</b> If 1, the processor uses override configuration <sup>1</sup> specified in MSR_TURBO_RATIO_LIMIT and MSR_TURBO_RATIO_LIMIT1. If 0, the processor uses factory-set configuration (Default).
29DH	669	IA32_MC29_CTL2	Package	See Table 35-2.
29EH	670	IA32_MC30_CTL2	Package	See Table 35-2.
29FH	671	IA32_MC31_CTL2	Package	See Table 35-2.

**Table 35-25. Additional MSRs Supported by Intel® Xeon® Processor E7 v2 Family with DisplayFamily\_DisplayModel Signature 06\_3EH**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
3F1H	1009	MSR_PEBS_ENABLE	Thread	See Section 18.7.1.1, "Precise Event Based Sampling (PEBS)."
		0		Enable PEBS on IA32_PMC0. (R/W)
		1		Enable PEBS on IA32_PMC1. (R/W)
		2		Enable PEBS on IA32_PMC2. (R/W)
		3		Enable PEBS on IA32_PMC3. (R/W)
		31:4		Reserved.
		32		Enable Load Latency on IA32_PMC0. (R/W)
		33		Enable Load Latency on IA32_PMC1. (R/W)
		34		Enable Load Latency on IA32_PMC2. (R/W)
		35		Enable Load Latency on IA32_PMC3. (R/W)
		63:36		Reserved.
41BH	1051	IA32_MC6_MISC	Package	Misc MAC information of Integrated I/O. (R/O) see Section 15.3.2.4
		5:0		Recoverable Address LSB
		8:6		Address Mode
		15:9		Reserved
		31:16		PCI Express Requestor ID
		39:32		PCI Express Segment Number
		63:32		Reserved
474H	1140	IA32_MC29_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Bank MC29 reports MC error from a specific CBo (core broadcast) and its corresponding slice of L3.
475H	1141	IA32_MC29_STATUS	Package	
476H	1142	IA32_MC29_ADDR	Package	
477H	1143	IA32_MC29_MISC	Package	
478H	1144	IA32_MC30_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Bank MC30 reports MC error from a specific CBo (core broadcast) and its corresponding slice of L3.
479H	1145	IA32_MC30_STATUS	Package	
47AH	1146	IA32_MC30_ADDR	Package	
47BH	1147	IA32_MC30_MISC	Package	
47CH	1148	IA32_MC31_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Bank MC31 reports MC error from a specific CBo (core broadcast) and its corresponding slice of L3.
47DH	1149	IA32_MC31_STATUS	Package	
47EH	1150	IA32_MC31_ADDR	Package	
47FH	1147	IA32_MC31_MISC	Package	

See Table 35-18, Table 35-24 for other MSR definitions applicable to Intel Xeon processor E7 v2 with CPUID signature 06\_3AH.

**NOTES:**

1. An override configuration lower than the factory-set configuration is always supported. An override configuration higher than the factory-set configuration is dependent on features specific to the processor and the platform.

### 35.10.3 Additional Uncore PMU MSRs in the Intel® Xeon® Processor E5 v2 and E7 v2 Families

Intel Xeon Processor E5 v2 and E7 v2 families are based on the Ivy Bridge-E microarchitecture. The MSR-based uncore PMU interfaces are listed in Table 35-22 and Table 35-26. For complete detail of the uncore PMU, refer to Intel Xeon Processor E5 v2 Product Family Uncore Performance Monitoring Guide. These processors have a CPUID signature with DisplayFamily\_DisplayModel of 06\_3EH.

**Table 35-26. Uncore PMU MSRs in Intel® Xeon® Processor E5 v2 and E7 v2 Families**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
C00H		MSR_PMON_GLOBAL_CTL	Package	Uncore perfmon per-socket global control.
C01H		MSR_PMON_GLOBAL_STATUS	Package	Uncore perfmon per-socket global status.
C06H		MSR_PMON_GLOBAL_CONFIG	Package	Uncore perfmon per-socket global configuration.
C15H		MSR_U_PMON_BOX_STATUS	Package	Uncore U-box perfmon U-box wide status.
C35H		MSR_PCU_PMON_BOX_STATUS	Package	Uncore PCU perfmon box wide status.
D1AH		MSR_C0_PMON_BOX_FILTER1	Package	Uncore C-box 0 perfmon box wide filter1.
D3AH		MSR_C1_PMON_BOX_FILTER1	Package	Uncore C-box 1 perfmon box wide filter1.
D5AH		MSR_C2_PMON_BOX_FILTER1	Package	Uncore C-box 2 perfmon box wide filter1.
D7AH		MSR_C3_PMON_BOX_FILTER1	Package	Uncore C-box 3 perfmon box wide filter1.
D9AH		MSR_C4_PMON_BOX_FILTER1	Package	Uncore C-box 4 perfmon box wide filter1.
DBAH		MSR_C5_PMON_BOX_FILTER1	Package	Uncore C-box 5 perfmon box wide filter1.
DDAH		MSR_C6_PMON_BOX_FILTER1	Package	Uncore C-box 6 perfmon box wide filter1.
DFAH		MSR_C7_PMON_BOX_FILTER1	Package	Uncore C-box 7 perfmon box wide filter1.
E04H		MSR_C8_PMON_BOX_CTL	Package	Uncore C-box 8 perfmon local box wide control.
E10H		MSR_C8_PMON_EVNTSEL0	Package	Uncore C-box 8 perfmon event select for C-box 8 counter 0.
E11H		MSR_C8_PMON_EVNTSEL1	Package	Uncore C-box 8 perfmon event select for C-box 8 counter 1.
E12H		MSR_C8_PMON_EVNTSEL2	Package	Uncore C-box 8 perfmon event select for C-box 8 counter 2.
E13H		MSR_C8_PMON_EVNTSEL3	Package	Uncore C-box 8 perfmon event select for C-box 8 counter 3.
E14H		MSR_C8_PMON_BOX_FILTER	Package	Uncore C-box 8 perfmon box wide filter.
E16H		MSR_C8_PMON_CTR0	Package	Uncore C-box 8 perfmon counter 0.
E17H		MSR_C8_PMON_CTR1	Package	Uncore C-box 8 perfmon counter 1.
E18H		MSR_C8_PMON_CTR2	Package	Uncore C-box 8 perfmon counter 2.
E19H		MSR_C8_PMON_CTR3	Package	Uncore C-box 8 perfmon counter 3.
E1AH		MSR_C8_PMON_BOX_FILTER1	Package	Uncore C-box 8 perfmon box wide filter1.
E24H		MSR_C9_PMON_BOX_CTL	Package	Uncore C-box 9 perfmon local box wide control.
E30H		MSR_C9_PMON_EVNTSEL0	Package	Uncore C-box 9 perfmon event select for C-box 9 counter 0.
E31H		MSR_C9_PMON_EVNTSEL1	Package	Uncore C-box 9 perfmon event select for C-box 9 counter 1.
E32H		MSR_C9_PMON_EVNTSEL2	Package	Uncore C-box 9 perfmon event select for C-box 9 counter 2.
E33H		MSR_C9_PMON_EVNTSEL3	Package	Uncore C-box 9 perfmon event select for C-box 9 counter 3.
E34H		MSR_C9_PMON_BOX_FILTER	Package	Uncore C-box 9 perfmon box wide filter.
E36H		MSR_C9_PMON_CTR0	Package	Uncore C-box 9 perfmon counter 0.
E37H		MSR_C9_PMON_CTR1	Package	Uncore C-box 9 perfmon counter 1.

Table 35-26. Uncore PMU MSRs in Intel® Xeon® Processor E5 v2 and E7 v2 Families (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
E38H		MSR_C9_PMON_CTR2	Package	Uncore C-box 9 perfmon counter 2.
E39H		MSR_C9_PMON_CTR3	Package	Uncore C-box 9 perfmon counter 3.
E3AH		MSR_C9_PMON_BOX_FILTER1	Package	Uncore C-box 9 perfmon box wide filter1.
E44H		MSR_C10_PMON_BOX_CTL	Package	Uncore C-box 10 perfmon local box wide control.
E50H		MSR_C10_PMON_EVNTSEL0	Package	Uncore C-box 10 perfmon event select for C-box 10 counter 0.
E51H		MSR_C10_PMON_EVNTSEL1	Package	Uncore C-box 10 perfmon event select for C-box 10 counter 1.
E52H		MSR_C10_PMON_EVNTSEL2	Package	Uncore C-box 10 perfmon event select for C-box 10 counter 2.
E53H		MSR_C10_PMON_EVNTSEL3	Package	Uncore C-box 10 perfmon event select for C-box 10 counter 3.
E54H		MSR_C10_PMON_BOX_FILTER	Package	Uncore C-box 10 perfmon box wide filter.
E56H		MSR_C10_PMON_CTR0	Package	Uncore C-box 10 perfmon counter 0.
E57H		MSR_C10_PMON_CTR1	Package	Uncore C-box 10 perfmon counter 1.
E58H		MSR_C10_PMON_CTR2	Package	Uncore C-box 10 perfmon counter 2.
E59H		MSR_C10_PMON_CTR3	Package	Uncore C-box 10 perfmon counter 3.
E5AH		MSR_C10_PMON_BOX_FILTER1	Package	Uncore C-box 10 perfmon box wide filter1.
E64H		MSR_C11_PMON_BOX_CTL	Package	Uncore C-box 11 perfmon local box wide control.
E70H		MSR_C11_PMON_EVNTSEL0	Package	Uncore C-box 11 perfmon event select for C-box 11 counter 0.
E71H		MSR_C11_PMON_EVNTSEL1	Package	Uncore C-box 11 perfmon event select for C-box 11 counter 1.
E72H		MSR_C11_PMON_EVNTSEL2	Package	Uncore C-box 11 perfmon event select for C-box 11 counter 2.
E73H		MSR_C11_PMON_EVNTSEL3	Package	Uncore C-box 11 perfmon event select for C-box 11 counter 3.
E74H		MSR_C11_PMON_BOX_FILTER	Package	Uncore C-box 11 perfmon box wide filter.
E76H		MSR_C11_PMON_CTR0	Package	Uncore C-box 11 perfmon counter 0.
E77H		MSR_C11_PMON_CTR1	Package	Uncore C-box 11 perfmon counter 1.
E78H		MSR_C11_PMON_CTR2	Package	Uncore C-box 11 perfmon counter 2.
E79H		MSR_C11_PMON_CTR3	Package	Uncore C-box 11 perfmon counter 3.
E7AH		MSR_C11_PMON_BOX_FILTER1	Package	Uncore C-box 11 perfmon box wide filter1.
E84H		MSR_C12_PMON_BOX_CTL	Package	Uncore C-box 12 perfmon local box wide control.
E90H		MSR_C12_PMON_EVNTSEL0	Package	Uncore C-box 12 perfmon event select for C-box 12 counter 0.
E91H		MSR_C12_PMON_EVNTSEL1	Package	Uncore C-box 12 perfmon event select for C-box 12 counter 1.
E92H		MSR_C12_PMON_EVNTSEL2	Package	Uncore C-box 12 perfmon event select for C-box 12 counter 2.
E93H		MSR_C12_PMON_EVNTSEL3	Package	Uncore C-box 12 perfmon event select for C-box 12 counter 3.
E94H		MSR_C12_PMON_BOX_FILTER	Package	Uncore C-box 12 perfmon box wide filter.
E96H		MSR_C12_PMON_CTR0	Package	Uncore C-box 12 perfmon counter 0.
E97H		MSR_C12_PMON_CTR1	Package	Uncore C-box 12 perfmon counter 1.
E98H		MSR_C12_PMON_CTR2	Package	Uncore C-box 12 perfmon counter 2.
E99H		MSR_C12_PMON_CTR3	Package	Uncore C-box 12 perfmon counter 3.
E9AH		MSR_C12_PMON_BOX_FILTER1	Package	Uncore C-box 12 perfmon box wide filter1.
EA4H		MSR_C13_PMON_BOX_CTL	Package	Uncore C-box 13 perfmon local box wide control.

**Table 35-26. Uncore PMU MSRs in Intel® Xeon® Processor E5 v2 and E7 v2 Families (Contd.)**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
EB0H		MSR_C13_PMON_EVNTSELO	Package	Uncore C-box 13 perfmon event select for C-box 13 counter 0.
EB1H		MSR_C13_PMON_EVNTSEL1	Package	Uncore C-box 13 perfmon event select for C-box 13 counter 1.
EB2H		MSR_C13_PMON_EVNTSEL2	Package	Uncore C-box 13 perfmon event select for C-box 13 counter 2.
EB3H		MSR_C13_PMON_EVNTSEL3	Package	Uncore C-box 13 perfmon event select for C-box 13 counter 3.
EB4H		MSR_C13_PMON_BOX_FILTER	Package	Uncore C-box 13 perfmon box wide filter.
EB6H		MSR_C13_PMON_CTR0	Package	Uncore C-box 13 perfmon counter 0.
EB7H		MSR_C13_PMON_CTR1	Package	Uncore C-box 13 perfmon counter 1.
EB8H		MSR_C13_PMON_CTR2	Package	Uncore C-box 13 perfmon counter 2.
EB9H		MSR_C13_PMON_CTR3	Package	Uncore C-box 13 perfmon counter 3.
EBAH		MSR_C13_PMON_BOX_FILTER1	Package	Uncore C-box 13 perfmon box wide filter1.
EC4H		MSR_C14_PMON_BOX_CTL	Package	Uncore C-box 14 perfmon local box wide control.
ED0H		MSR_C14_PMON_EVNTSELO	Package	Uncore C-box 14 perfmon event select for C-box 14 counter 0.
ED1H		MSR_C14_PMON_EVNTSEL1	Package	Uncore C-box 14 perfmon event select for C-box 14 counter 1.
ED2H		MSR_C14_PMON_EVNTSEL2	Package	Uncore C-box 14 perfmon event select for C-box 14 counter 2.
ED3H		MSR_C14_PMON_EVNTSEL3	Package	Uncore C-box 14 perfmon event select for C-box 14 counter 3.
ED4H		MSR_C14_PMON_BOX_FILTER	Package	Uncore C-box 14 perfmon box wide filter.
ED6H		MSR_C14_PMON_CTR0	Package	Uncore C-box 14 perfmon counter 0.
ED7H		MSR_C14_PMON_CTR1	Package	Uncore C-box 14 perfmon counter 1.
ED8H		MSR_C14_PMON_CTR2	Package	Uncore C-box 14 perfmon counter 2.
ED9H		MSR_C14_PMON_CTR3	Package	Uncore C-box 14 perfmon counter 3.
EDAH		MSR_C14_PMON_BOX_FILTER1	Package	Uncore C-box 14 perfmon box wide filter1.

### 35.11 MSRS IN THE 4TH GENERATION INTEL® CORE™ PROCESSORS (BASED ON HASWELL MICROARCHITECTURE)

The 4th generation Intel® Core™ processor family and Intel® Xeon® processor E3-1200v3 product family (based on Haswell microarchitecture), with CPUID DisplayFamily\_DisplayModel signature 06\_3CH/06\_45H/06\_46H, support the MSR interfaces listed in Table 35-18, Table 35-19, Table 35-20, and Table 35-27. For an MSR listed in Table 35-18 that also appears in Table 35-27, Table 35-27 supercede Table 35-18.

The MSRs listed in Table 35-27 also apply to processors based on Haswell-E microarchitecture (see Section 35.12).

**Table 35-27. Additional MSRs Supported by Processors based on the Haswell or Haswell-E microarchitectures**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
3BH	59	IA32_TSC_ADJUST	THREAD	<b>Per-Logical-Processor TSC ADJUST (R/W)</b> See Table 35-2.
CEH	206	MSR_PLATFORM_INFO	Package	
		7:0		Reserved.

**Table 35-27. Additional MSRs Supported by Processors based on the Haswell or Haswell-E microarchitectures**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		15:8	Package	<b>Maximum Non-Turbo Ratio (R/O)</b> The is the ratio of the frequency that invariant TSC runs at. Frequency = ratio * 100 MHz.
		27:16		Reserved.
		28	Package	<b>Programmable Ratio Limit for Turbo Mode (R/O)</b> When set to 1, indicates that Programmable Ratio Limits for Turbo mode is enabled, and when set to 0, indicates Programmable Ratio Limits for Turbo mode is disabled.
		29	Package	<b>Programmable TDP Limit for Turbo Mode (R/O)</b> When set to 1, indicates that TDP Limits for Turbo mode are programmable, and when set to 0, indicates TDP Limit for Turbo mode is not programmable.
		31:30		Reserved.
		32	Package	<b>Low Power Mode Support (LPM) (R/O)</b> When set to 1, indicates that LPM is supported, and when set to 0, indicates LPM is not supported.
		34:33	Package	<b>Number of ConfigTDP Levels (R/O)</b> 00: Only Base TDP level available. 01: One additional TDP level available. 02: Two additional TDP level available. 11: Reserved
		39:35		Reserved.
		47:40	Package	<b>Maximum Efficiency Ratio (R/O)</b> The is the minimum ratio (maximum efficiency) that the processor can operate, in units of 100MHz.
		55:48	Package	<b>Minimum Operating Ratio (R/O)</b> Contains the minimum supported operating ratio in units of 100 MHz.
		63:56		Reserved.
186H	390	IA32_PERFEVTSELO	THREAD	<b>Performance Event Select for Counter 0 (R/W)</b> Supports all fields described inTable 35-2 and the fields below.
		32		IN_TX: see <b>Section 18.11.5.1</b> When IN_TX (bit 32) is set, AnyThread (bit 21) should be cleared to prevent incorrect results
187H	391	IA32_PERFEVTSEL1	THREAD	<b>Performance Event Select for Counter 1 (R/W)</b> Supports all fields described inTable 35-2 and the fields below.
		32		IN_TX: see <b>Section 18.11.5.1</b> When IN_TX (bit 32) is set, AnyThread (bit 21) should be cleared to prevent incorrect results
188H	392	IA32_PERFEVTSEL2	THREAD	<b>Performance Event Select for Counter 2 (R/W)</b> Supports all fields described inTable 35-2 and the fields below.



**Table 35-27. Additional MSRs Supported by Processors based on the Haswell or Haswell-E microarchitectures**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		32		IN_TX: see <b>Section 18.11.5.1</b> When IN_TX (bit 32) is set, AnyThread (bit 21) should be cleared to prevent incorrect results
		33		IN_TXCP: see <b>Section 18.11.5.1</b> When IN_TXCP=1 & IN_TX=1 and in sampling, spurious PMI may occur and transactions may continuously abort near overflow conditions. Software should favor using IN_TXCP for counting over sampling. If sampling, software should use large “sample-after” value after clearing the counter configured to use IN_TXCP and also always reset the counter even when no overflow condition was reported.
189H	393	IA32_PERFEVTSEL3	THREAD	<b>Performance Event Select for Counter 3 (R/W)</b> Supports all fields described in Table 35-2 and the fields below.
		32		IN_TX: see <b>Section 18.11.5.1</b> When IN_TX (bit 32) is set, AnyThread (bit 21) should be cleared to prevent incorrect results
1C8H	456	MSR_LBR_SELECT	Thread	<b>Last Branch Record Filtering Select Register (R/W)</b>
		0		CPL_EQ_0
		1		CPL_NEQ_0
		2		JCC
		3		NEAR_REL_CALL
		4		NEAR_IND_CALL
		5		NEAR_RET
		6		NEAR_IND_JMP
		7		NEAR_REL_JMP
		8		FAR_BRANCH
		9		EN_CALL_STACK
		63:9		Reserved.
1D9H	473	IA32_DEBUGCTL	Thread	<b>Debug Control (R/W)</b> See Table 35-2.
		0		LBR: Last Branch Record
		1		BTF
		5:2		Reserved.
		6		TR: Branch Trace
		7		BTS: Log Branch Trace Message to BTS buffer
		8		BTINT
		9		BTS_OFF_OS
		10		BTS_OFF_USER
		11		FREEZE_LBR_ON_PMI
12		FREEZE_PERFMON_ON_PMI		

**Table 35-27. Additional MSRs Supported by Processors based on the Haswell or Haswell-E microarchitectures**

Register Address		Register Name	Scope	Bit Description	
Hex	Dec				
		13		<b>ENABLE_UNCORE_PMI</b>	
		14		<b>FREEZE_WHILE_SMM</b>	
		15		<b>RTM_DEBUG</b>	
		63:15		Reserved.	
491H	1169	IA32_VMX_VMFUNC	THREAD	<b>Capability Reporting Register of VM-function Controls (R/O)</b> See Table 35-2	
60BH	1548	MSR_PKG_C_IRTL1	Package	<b>Package C6/C7 Interrupt Response Limit 1 (R/W)</b> This MSR defines the interrupt response time limit used by the processor to manage transition to package C6 or C7 state. The latency programmed in this register is for the shorter-latency sub C-states used by an MWAIT hint to C6 or C7 state.  Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.	
				9:0	<b>Interrupt response time limit (R/W)</b> Specifies the limit that should be used to decide if the package should be put into a package C6 or C7 state.
				12:10	<b>Time Unit (R/W)</b> Specifies the encoding value of time unit of the interrupt response time limit. See Table 35-18 for supported time unit encodings.
				14:13	Reserved.
				15	<b>Valid (R/W)</b> Indicates whether the values in bits 12:0 are valid and can be used by the processor for package C-state management.
				63:16	Reserved.
60CH	1548	MSR_PKG_C_IRTL2	Package	<b>Package C6/C7 Interrupt Response Limit 2 (R/W)</b> This MSR defines the interrupt response time limit used by the processor to manage transition to package C6 or C7 state. The latency programmed in this register is for the longer-latency sub C-states used by an MWAIT hint to C6 or C7 state.  Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.	
				9:0	<b>Interrupt response time limit (R/W)</b> Specifies the limit that should be used to decide if the package should be put into a package C6 or C7 state.
				12:10	<b>Time Unit (R/W)</b> Specifies the encoding value of time unit of the interrupt response time limit. See Table 35-18 for supported time unit encodings.
				14:13	Reserved.
				15	<b>Valid (R/W)</b> Indicates whether the values in bits 12:0 are valid and can be used by the processor for package C-state management.
				63:16	Reserved.

**Table 35-27. Additional MSRs Supported by Processors based on the Haswell or Haswell-E microarchitectures**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
613H	1555	MSR_PKG_PERF_STATUS	Package	<b>PKG Perf Status (R/O)</b> See Section 14.9.3, "Package RAPL Domain."
619H	1561	MSR_DRAM_ENERGY_STATUS	Package	<b>DRAM Energy Status (R/O)</b> See Section 14.9.5, "DRAM RAPL Domain."
61BH	1563	MSR_DRAM_PERF_STATUS	Package	<b>DRAM Performance Throttling Status (R/O)</b> See Section 14.9.5, "DRAM RAPL Domain."
648H	1608	MSR_CONFIG_TDP_NOMINAL	Package	<b>Base TDP Ratio (R/O)</b>
		7:0		<b>Config_TDP_Base</b> Base TDP level ratio to be used for this specific processor (in units of 100 MHz).
		63:8		Reserved.
649H	1609	MSR_CONFIG_TDP_LEVEL1	Package	ConfigTDP Level 1 ratio and power level (R/O)
		14:0		PKG_TDP_LVL1. Power setting for ConfigTDP Level 1.
		15		Reserved
		23:16		Config_TDP_LVL1_Ratio. ConfigTDP level 1 ratio to be used for this specific processor.
		31:24		Reserved
		46:32		PKG_MAX_PWR_LVL1. Max Power setting allowed for ConfigTDP Level 1.
		62:47		PKG_MIN_PWR_LVL1. MIN Power setting allowed for ConfigTDP Level 1.
		63		Reserved.
64AH	1610	MSR_CONFIG_TDP_LEVEL2	Package	ConfigTDP Level 2 ratio and power level (R/O)
		14:0		PKG_TDP_LVL2. Power setting for ConfigTDP Level 2.
		15		Reserved
		23:16		Config_TDP_LVL2_Ratio. ConfigTDP level 2 ratio to be used for this specific processor.
		31:24		Reserved
		46:32		PKG_MAX_PWR_LVL2. Max Power setting allowed for ConfigTDP Level 2.
		62:47		PKG_MIN_PWR_LVL2. MIN Power setting allowed for ConfigTDP Level 2.
		63		Reserved.
64BH	1611	MSR_CONFIG_TDP_CONTROL	Package	<b>ConfigTDP Control (R/W)</b>
		1:0		<b>TDP_LEVEL (RW/L)</b> System BIOS can program this field.
		30:2		Reserved.

**Table 35-27. Additional MSRs Supported by Processors based on the Haswell or Haswell-E microarchitectures**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		31		<b>Config_TDP_Lock (R/W/L)</b> When this bit is set, the content of this register is locked until a reset.
		63:32		Reserved.
64CH	1612	MSR_TURBO_ACTIVATION_RATIO	Package	<b>ConfigTDP Control (R/W)</b>
		7:0		<b>MAX_NON_TURBO_RATIO (R/W/L)</b> System BIOS can program this field.
		30:8		Reserved.
		31		<b>TURBO_ACTIVATION_RATIO_Lock (R/W/L)</b> When this bit is set, the content of this register is locked until a reset.
		63:32		Reserved.
C80H	3200	IA32_DEBUG_INTERFACE	Package	<b>Silicon Debug Feature Control (R/W)</b> See Table 35-2.

### 35.11.1 MSRs in 4th Generation Intel® Core™ Processor Family (based on Haswell Microarchitecture)

Table 35-28 lists model-specific registers (MSRs) that are specific to 4th generation Intel® Core™ processor family and Intel® Xeon® processor E3-1200 v3 product family (based on Haswell microarchitecture). These processors have a CPUID signature with DisplayFamily\_DisplayModel of 06\_3CH/06\_45H/06\_46H, see Table 35-1.

**Table 35-28. MSRs Supported by 4th Generation Intel® Core™ Processors (Haswell microarchitecture)**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
E2H	226	MSR_PKG_CST_CONFIG_CONTROL	Core	<b>C-State Configuration Control (R/W)</b> Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-states. See <a href="http://biosbits.org">http://biosbits.org</a> .

**Table 35-28. MSRs Supported by 4th Generation Intel® Core™ Processors (Haswell microarchitecture) (Contd.)**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		3:0		<b>Package C-State Limit (R/W)</b> Specifies the lowest processor-specific C-state code name (consuming the least power) for the package. The default is set as factory-configured package C-state limit. The following C-state code name encodings are supported: 0000b: C0/C1 (no package C-state support) 0001b: C2 0010b: C3 0011b: C6 0100b: C7 0101b: C7s Package C states C7 are not available to processor with signature 06_3CH
		9:4		Reserved
		10		<b>I/O MWAIT Redirection Enable (R/W)</b>
		14:11		Reserved
		15		<b>CFG Lock (R/WO)</b>
		24:16		Reserved
		25		<b>C3 State Auto Demotion Enable (R/W)</b>
		26		<b>C1 State Auto Demotion Enable (R/W)</b>
		27		<b>Enable C3 Undemotion (R/W)</b>
		28		<b>Enable C1 Undemotion (R/W)</b>
		63:29		Reserved
17DH	390	MSR_SMM_MCA_CAP	THREAD	<b>Enhanced SMM Capabilities (SMM-RO)</b> Reports SMM capability Enhancement. Accessible only while in SMM.
		57:0		<b>Reserved</b>
		58		<b>SMM_Code_Access_Chk (SMM-RO)</b> If set to 1 indicates that the SMM code access restriction is supported and the MSR_SMM_FEATURE_CONTROL is supported.
		59		<b>Long_Flow_Indication (SMM-RO)</b> If set to 1 indicates that the SMM long flow indicator is supported and the MSR_SMM_DELAYED is supported.
		63:60		Reserved
1ADH	429	MSR_TURBO_RATIO_LIMIT	Package	<b>Maximum Ratio Limit of Turbo Mode</b> RO if MSR_PLATFORM_INFO.[28] = 0, RW if MSR_PLATFORM_INFO.[28] = 1
		7:0	Package	<b>Maximum Ratio Limit for 1C</b> Maximum turbo ratio limit of 1 core active.
		15:8	Package	<b>Maximum Ratio Limit for 2C</b> Maximum turbo ratio limit of 2 core active.

**Table 35-28. MSRs Supported by 4th Generation Intel® Core™ Processors (Haswell microarchitecture) (Contd.)**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		23:16	Package	<b>Maximum Ratio Limit for 3C</b> Maximum turbo ratio limit of 3 core active.
		31:24	Package	<b>Maximum Ratio Limit for 4C</b> Maximum turbo ratio limit of 4 core active.
		63:32		Reserved.
391H	913	MSR_UNC_PERF_GLOBAL_CTRL	Package	Uncore PMU global control
		0		Core 0 select
		1		Core 1 select
		2		Core 2 select
		3		Core 3 select
		18:4		Reserved.
		29		Enable all uncore counters
		30		Enable wake on PMI
		31		Enable Freezing counter when overflow
		63:32		Reserved.
392H	914	MSR_UNC_PERF_GLOBAL_STATUS	Package	Uncore PMU main status
		0		Fixed counter overflowed
		1		An ARB counter overflowed
		2		Reserved
		3		A CBox counter overflowed (on any slice)
		63:4		Reserved.
394H	916	MSR_UNC_PERF_FIXED_CTRL	Package	Uncore fixed counter control (R/W)
		19:0		Reserved
		20		Enable overflow propagation
		21		Reserved
		22		Enable counting
		63:23		Reserved.
395H	917	MSR_UNC_PERF_FIXED_CTR	Package	Uncore fixed counter
		47:0		Current count
		63:48		Reserved.
396H	918	MSR_UNC_CBO_CONFIG	Package	Uncore C-Box configuration information (R/O)
		3:0		Encoded number of C-Box, derive value by "-1"
		63:4		Reserved.
3B0H	946	MSR_UNC_ARB_PERFCTRO	Package	Uncore Arb unit, performance counter 0

**Table 35-28. MSRs Supported by 4th Generation Intel® Core™ Processors (Haswell microarchitecture) (Contd.)**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
3B1H	947	MSR_UNC_ARB_PERFCTR1	Package	Uncore Arb unit, performance counter 1
3B2H	944	MSR_UNC_ARB_PERFEVTSELO	Package	Uncore Arb unit, counter 0 event select MSR
3B3H	945	MSR_UNC_ARB_PERFEVTSEL1	Package	Uncore Arb unit, counter 1 event select MSR
391H	913	MSR_UNC_PERF_GLOBAL_CTRL	Package	Uncore PMU global control
		0		Core 0 select
		1		Core 1 select
		2		Core 2 select
		3		Core 3 select
		18:4		Reserved.
		29		Enable all uncore counters
		30		Enable wake on PMI
		31		Enable Freezing counter when overflow
63:32		Reserved.		
395H	917	MSR_UNC_PERF_FIXED_CTR	Package	Uncore fixed counter
		47:0		Current count
		63:48		Reserved.
3B3H	945	MSR_UNC_ARB_PERFEVTSEL1	Package	Uncore Arb unit, counter 1 event select MSR
4E0H	1248	MSR_SMM_FEATURE_CONTROL	Package	<b>Enhanced SMM Feature Control (SMM-RW)</b> Reports SMM capability Enhancement. Accessible only while in SMM.
		0		<b>Lock (SMM-RWO)</b> When set to '1' locks this register from further changes
		1		Reserved
		2		<b>SMM_Code_Chk_En (SMM-RW)</b> This control bit is available only if MSR_SMM_MCA_CAP[58] == 1. When set to '0' (default) none of the logical processors are prevented from executing SMM code outside the ranges defined by the SMRR. When set to '1' any logical processor in the package that attempts to execute SMM code not within the ranges defined by the SMRR will assert an unrecoverable MCE.
		63:3		Reserved
4E2H	1250	MSR_SMM_DELAYED	Package	<b>SMM Delayed (SMM-RO)</b> Reports the interruptible state of all logical processors in the package. Available only while in SMM and MSR_SMM_MCA_CAP[LONG_FLOW_INDICATION] == 1.

Table 35-28. MSRs Supported by 4th Generation Intel® Core™ Processors (Haswell microarchitecture) (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		N-1:0		<p><b>LOG_PROC_STATE (SMM-RO)</b></p> <p>Each bit represents a logical processor of its state in a long flow of internal operation which delays servicing an interrupt. The corresponding bit will be set at the start of long events such as: Microcode Update Load, C6, WBINVD, Ratio Change, Throttle.</p> <p>The bit is automatically cleared at the end of each long event. The reset value of this field is 0.</p> <p>Only bit positions below N = CPUID.(EAX=0BH, ECX=PKG_LVL):EBX[15:0] can be updated.</p>
		63:N		Reserved
4E3H	1251	MSR_SMM_BLOCKED	Package	<p><b>SMM Blocked (SMM-RO)</b></p> <p>Reports the blocked state of all logical processors in the package. Available only while in SMM.</p>
		N-1:0		<p><b>LOG_PROC_STATE (SMM-RO)</b></p> <p>Each bit represents a logical processor of its blocked state to service an SMI. The corresponding bit will be set if the logical processor is in one of the following states: Wait For SIPI or SENTER Sleep.</p> <p>The reset value of this field is OFFFH.</p> <p>Only bit positions below N = CPUID.(EAX=0BH, ECX=PKG_LVL):EBX[15:0] can be updated.</p>
		63:N		Reserved
606H	1542	MSR_RAPL_POWER_UNIT	Package	<p><b>Unit Multipliers used in RAPL Interfaces (R/O)</b></p>
		3:0	Package	<p><b>Power Units</b></p> <p>See Section 14.9.1, "RAPL Interfaces."</p>
		7:4	Package	<b>Reserved</b>
		12:8	Package	<p><b>Energy Status Units</b></p> <p>Energy related information (in Joules) is based on the multiplier, 1/2*ESU; where ESU is an unsigned integer represented by bits 12:8. Default value is 0EH (or 61 micro-joules)</p>
		15:13	Package	<b>Reserved</b>
		19:16	Package	<p><b>Time Units</b></p> <p>See Section 14.9.1, "RAPL Interfaces."</p>
		63:20		Reserved
639H	1593	MSR_PP0_ENERGY_STATUS	Package	<p><b>PP0 Energy Status (R/O)</b></p> <p>See Section 14.9.4, "PP0/PP1 RAPL Domains."</p>
640H	1600	MSR_PP1_POWER_LIMIT	Package	<p><b>PP1 RAPL Power Limit Control (R/W)</b></p> <p>See Section 14.9.4, "PP0/PP1 RAPL Domains."</p>
641H	1601	MSR_PP1_ENERGY_STATUS	Package	<p><b>PP1 Energy Status (R/O)</b></p> <p>See Section 14.9.4, "PP0/PP1 RAPL Domains."</p>
642H	1602	MSR_PP1_POLICY	Package	<p><b>PP1 Balance Policy (R/W)</b></p> <p>See Section 14.9.4, "PP0/PP1 RAPL Domains."</p>



**Table 35-28. MSRs Supported by 4th Generation Intel® Core™ Processors (Haswell microarchitecture) (Contd.)**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
690H	1680	MSR_CORE_PERF_LIMIT_REASONS	Package	<b>Indicator of Frequency Clipping in Processor Cores (R/W) (frequency refers to processor core frequency)</b>
		0		<b>PROCHOT Status (R0)</b> When set, processor core frequency is reduced below the operating system request due to assertion of external PROCHOT.
		1		<b>Thermal Status (R0)</b> When set, frequency is reduced below the operating system request due to a thermal event.
		3:2		Reserved.
		4		<b>Graphics Driver Status (R0)</b> When set, frequency is reduced below the operating system request due to Processor Graphics driver override.
		5		<b>Autonomous Utilization-Based Frequency Control Status (R0)</b> When set, frequency is reduced below the operating system request because the processor has detected that utilization is low.
		6		<b>VR Therm Alert Status (R0)</b> When set, frequency is reduced below the operating system request due to a thermal alert from the Voltage Regulator.
		7		Reserved.
		8		<b>Electrical Design Point Status (R0)</b> When set, frequency is reduced below the operating system request due to electrical design point constraints (e.g. maximum electrical current consumption).
		9		<b>Core Power Limiting Status (R0)</b> When set, frequency is reduced below the operating system request due to domain-level power limiting.
		10		<b>Package-Level Power Limiting PL1 Status (R0)</b> When set, frequency is reduced below the operating system request due to package-level power limiting PL1.
		11		<b>Package-Level PL2 Power Limiting Status (R0)</b> When set, frequency is reduced below the operating system request due to package-level power limiting PL2.
		12		<b>Max Turbo Limit Status (R0)</b> When set, frequency is reduced below the operating system request due to multi-core turbo limits.
		13		<b>Turbo Transition Attenuation Status (R0)</b> When set, frequency is reduced below the operating system request due to Turbo transition attenuation. This prevents performance degradation due to frequent operating ratio changes.
15:14		<b>Reserved</b>		

Table 35-28. MSRs Supported by 4th Generation Intel® Core™ Processors (Haswell microarchitecture) (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		16		<b>PROCHOT Log</b> When set, indicates that the PROCHOT Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		17		<b>Thermal Log</b> When set, indicates that the Thermal Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		19:18		Reserved.
		20		<b>Graphics Driver Log</b> When set, indicates that the Graphics Driver Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		21		<b>Autonomous Utilization-Based Frequency Control Log</b> When set, indicates that the Autonomous Utilization-Based Frequency Control Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		22		<b>VR Therm Alert Log</b> When set, indicates that the VR Therm Alert Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		23		Reserved.
		24		<b>Electrical Design Point Log</b> When set, indicates that the EDP Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		25		<b>Core Power Limiting Log</b> When set, indicates that the Core Power Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		26		<b>Package-Level PL1 Power Limiting Log</b> When set, indicates that the Package Level PL1 Power Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		27		<b>Package-Level PL2 Power Limiting Log</b> When set, indicates that the Package Level PL2 Power Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		28		<b>Max Turbo Limit Log</b> When set, indicates that the Max Turbo Limit Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.

**Table 35-28. MSRs Supported by 4th Generation Intel® Core™ Processors (Haswell microarchitecture) (Contd.)**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		29		<b>Turbo Transition Attenuation Log</b> When set, indicates that the Turbo Transition Attenuation Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		63:30		Reserved.
6B0H	1712	MSR_GRAPHICS_PERF_LIMIT_REASONS	Package	<b>Indicator of Frequency Clipping in the Processor Graphics (R/W)</b> <b>(frequency refers to processor graphics frequency)</b>
		0		<b>PROCHOT Status (R0)</b> When set, frequency is reduced below the operating system request due to assertion of external PROCHOT.
		1		<b>Thermal Status (R0)</b> When set, frequency is reduced below the operating system request due to a thermal event.
		3:2		Reserved.
		4		<b>Graphics Driver Status (R0)</b> When set, frequency is reduced below the operating system request due to Processor Graphics driver override.
		5		<b>Autonomous Utilization-Based Frequency Control Status (R0)</b> When set, frequency is reduced below the operating system request because the processor has detected that utilization is low
		6		<b>VR Therm Alert Status (R0)</b> When set, frequency is reduced below the operating system request due to a thermal alert from the Voltage Regulator.
		7		Reserved.
		8		<b>Electrical Design Point Status (R0)</b> When set, frequency is reduced below the operating system request due to electrical design point constraints (e.g. maximum electrical current consumption).
		9		<b>Graphics Power Limiting Status (R0)</b> When set, frequency is reduced below the operating system request due to domain-level power limiting.
		10		<b>Package-Level Power Limiting PL1 Status (R0)</b> When set, frequency is reduced below the operating system request due to package-level power limiting PL1.
		11		<b>Package-Level PL2 Power Limiting Status (R0)</b> When set, frequency is reduced below the operating system request due to package-level power limiting PL2.
		15:12		Reserved
		16		<b>PROCHOT Log</b> When set, indicates that the PROCHOT Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.

Table 35-28. MSRs Supported by 4th Generation Intel® Core™ Processors (Haswell microarchitecture) (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		17		<b>Thermal Log</b> When set, indicates that the Thermal Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		19:18		Reserved.
		20		<b>Graphics Driver Log</b> When set, indicates that the Graphics Driver Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		21		<b>Autonomous Utilization-Based Frequency Control Log</b> When set, indicates that the Autonomous Utilization-Based Frequency Control Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		22		<b>VR Therm Alert Log</b> When set, indicates that the VR Therm Alert Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		23		Reserved.
		24		<b>Electrical Design Point Log</b> When set, indicates that the EDP Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		25		<b>Core Power Limiting Log</b> When set, indicates that the Core Power Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		26		<b>Package-Level PL1 Power Limiting Log</b> When set, indicates that the Package Level PL1 Power Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		27		<b>Package-Level PL2 Power Limiting Log</b> When set, indicates that the Package Level PL2 Power Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		28		<b>Max Turbo Limit Log</b> When set, indicates that the Max Turbo Limit Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		29		<b>Turbo Transition Attenuation Log</b> When set, indicates that the Turbo Transition Attenuation Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.

**Table 35-28. MSRs Supported by 4th Generation Intel® Core™ Processors (Haswell microarchitecture) (Contd.)**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		63:30		Reserved.
6B1H	1713	MSR_RING_PERF_LIMIT_REASONS	Package	<b>Indicator of Frequency Clipping in the Ring Interconnect (R/W)</b> (frequency refers to ring interconnect in the uncore)
		0		<b>PROCHOT Status (R0)</b> When set, frequency is reduced below the operating system request due to assertion of external PROCHOT.
		1		<b>Thermal Status (R0)</b> When set, frequency is reduced below the operating system request due to a thermal event.
		5:2		Reserved.
		6		<b>VR Therm Alert Status (R0)</b> When set, frequency is reduced below the operating system request due to a thermal alert from the Voltage Regulator.
		7		Reserved.
		8		<b>Electrical Design Point Status (R0)</b> When set, frequency is reduced below the operating system request due to electrical design point constraints (e.g. maximum electrical current consumption).
		9		Reserved.
		10		<b>Package-Level Power Limiting PL1 Status (R0)</b> When set, frequency is reduced below the operating system request due to package-level power limiting PL1.
		11		<b>Package-Level PL2 Power Limiting Status (R0)</b> When set, frequency is reduced below the operating system request due to package-level power limiting PL2.
		15:12		<b>Reserved</b>
		16		<b>PROCHOT Log</b> When set, indicates that the PROCHOT Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		17		<b>Thermal Log</b> When set, indicates that the Thermal Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		19:18		Reserved.
20		<b>Graphics Driver Log</b> When set, indicates that the Graphics Driver Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.		

Table 35-28. MSRs Supported by 4th Generation Intel® Core™ Processors (Haswell microarchitecture) (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		21		<b>Autonomous Utilization-Based Frequency Control Log</b> When set, indicates that the Autonomous Utilization-Based Frequency Control Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		22		<b>VR Therm Alert Log</b> When set, indicates that the VR Therm Alert Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		23		Reserved.
		24		<b>Electrical Design Point Log</b> When set, indicates that the EDP Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		25		<b>Core Power Limiting Log</b> When set, indicates that the Core Power Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		26		<b>Package-Level PL1 Power Limiting Log</b> When set, indicates that the Package Level PL1 Power Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		27		<b>Package-Level PL2 Power Limiting Log</b> When set, indicates that the Package Level PL2 Power Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		28		<b>Max Turbo Limit Log</b> When set, indicates that the Max Turbo Limit Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		29		<b>Turbo Transition Attenuation Log</b> When set, indicates that the Turbo Transition Attenuation Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		63:30		Reserved.
700H	1792	MSR_UNC_CBO_0_PERFEVTSEL0	Package	Uncore C-Box 0, counter 0 event select MSR
701H	1793	MSR_UNC_CBO_0_PERFEVTSEL1	Package	Uncore C-Box 0, counter 1 event select MSR
706H	1798	MSR_UNC_CBO_0_PERFCTRO	Package	Uncore C-Box 0, performance counter 0
707H	1799	MSR_UNC_CBO_0_PERFCTR1	Package	Uncore C-Box 0, performance counter 1

**Table 35-28. MSRs Supported by 4th Generation Intel® Core™ Processors (Haswell microarchitecture) (Contd.)**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
710H	1808	MSR_UNC_CBO_1_PERFEVTSELO	Package	Uncore C-Box 1, counter 0 event select MSR
711H	1809	MSR_UNC_CBO_1_PERFEVTSEL1	Package	Uncore C-Box 1, counter 1 event select MSR
716H	1814	MSR_UNC_CBO_1_PERFCTRO	Package	Uncore C-Box 1, performance counter 0
717H	1815	MSR_UNC_CBO_1_PERFCTR1	Package	Uncore C-Box 1, performance counter 1
720H	1824	MSR_UNC_CBO_2_PERFEVTSELO	Package	Uncore C-Box 2, counter 0 event select MSR
721H	1824	MSR_UNC_CBO_2_PERFEVTSEL1	Package	Uncore C-Box 2, counter 1 event select MSR
726H	1830	MSR_UNC_CBO_2_PERFCTRO	Package	Uncore C-Box 2, performance counter 0
727H	1831	MSR_UNC_CBO_2_PERFCTR1	Package	Uncore C-Box 2, performance counter 1
730H	1840	MSR_UNC_CBO_3_PERFEVTSELO	Package	Uncore C-Box 3, counter 0 event select MSR
731H	1841	MSR_UNC_CBO_3_PERFEVTSEL1	Package	Uncore C-Box 3, counter 1 event select MSR.
736H	1846	MSR_UNC_CBO_3_PERFCTRO	Package	Uncore C-Box 3, performance counter 0.
737H	1847	MSR_UNC_CBO_3_PERFCTR1	Package	Uncore C-Box 3, performance counter 1.

See Table 35-18, Table 35-19, Table 35-20, Table 35-23, Table 35-27 for other MSR definitions applicable to processors with CPUID signatures 063CH, 06\_46H.

### 35.11.2 Additional Residency MSRs Supported in 4th Generation Intel® Core™ Processors

The 4th generation Intel® Core™ processor family (based on Haswell microarchitecture) with CPUID DisplayFamily\_DisplayModel signature 06\_45H supports the MSR interfaces listed in Table 35-18, Table 35-19, Table 35-27, Table 35-28, and Table 35-29.

**Table 35-29. Additional Residency MSRs Supported by 4th Generation Intel® Core™ Processors with DisplayFamily\_DisplayModel Signature 06\_45H**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
E2H	226	MSR_PKG_CST_CONFIG_CONTROL	Core	<p><b>C-State Configuration Control (R/W)</b></p> <p>Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-states. See <a href="http://biosbits.org">http://biosbits.org</a>.</p>

**Table 35-29. Additional Residency MSRs Supported by 4th Generation Intel® Core™ Processors with DisplayFamily\_DisplayModel Signature 06\_45H**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		3:0		<b>Package C-State Limit (R/W)</b> Specifies the lowest processor-specific C-state code name (consuming the least power) for the package. The default is set as factory-configured package C-state limit. The following C-state code name encodings are supported: 0000b: C0/C1 (no package C-state support) 0001b: C2 0010b: C3 0011b: C6 0100b: C7 0101b: C7s 0110b: C8 0111b: C9 1000b: C10
		9:4		Reserved
		10		<b>I/O MWAIT Redirection Enable (R/W)</b>
		14:11		Reserved
		15		<b>CFG Lock (R/WO)</b>
		24:16		Reserved
		25		<b>C3 State Auto Demotion Enable (R/W)</b>
		26		<b>C1 State Auto Demotion Enable (R/W)</b>
		27		<b>Enable C3 Undemotion (R/W)</b>
		28		<b>Enable C1 Undemotion (R/W)</b>
		63:29		Reserved
630H	1584	MSR_PKG_C8_RESIDENCY	Package	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		59:0		Package C8 Residency Counter. (R/O) Value since last reset that this package is in processor-specific C8 states. Count at the same frequency as the TSC.
		63:60		Reserved
631H	1585	MSR_PKG_C9_RESIDENCY	Package	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		59:0		Package C9 Residency Counter. (R/O) Value since last reset that this package is in processor-specific C9 states. Count at the same frequency as the TSC.
		63:60		Reserved
632H	1586	MSR_PKG_C10_RESIDENCY	Package	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.



**Table 35-29. Additional Residency MSRs Supported by 4th Generation Intel® Core™ Processors with DisplayFamily\_DisplayModel Signature 06\_45H**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		59:0		Package C10 Residency Counter. (R/O) Value since last reset that this package is in processor-specific C10 states. Count at the same frequency as the TSC.
		63:60		Reserved

See Table 35-18, Table 35-19, Table 35-20, Table 35-27, Table 35-28 for other MSR definitions applicable to processors with CPUID signature 06\_45H.

### 35.12 MSRS IN INTEL® XEON® PROCESSOR E5 V3 AND E7 V3 PRODUCT FAMILY

Intel® Xeon® processor E5 v3 family and Intel® Xeon® processor E7 v3 family are based on Haswell-E microarchitecture (CPUID DisplayFamily\_DisplayModel = 06\_3F). These processors supports the MSR interfaces listed in Table 35-18, Table 35-27, and Table 35-30.

**Table 35-30. Additional MSRs Supported by Intel® Xeon® Processor E5 v3 Family**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
35H	53	MSR_CORE_THREAD_COUNT	Package	<b>Configured State of Enabled Processor Core Count and Logical Processor Count (R0)</b> <ul style="list-style-type: none"> <li>After a Power-On RESET, enumerates factory configuration of the number of processor cores and logical processors in the physical package.</li> <li>Following the sequence of (i) BIOS modified a Configuration Mask which selects a subset of processor cores to be active post RESET and (ii) a RESET event after the modification, enumerates the current configuration of enabled processor core count and logical processor count in the physical package.</li> </ul>
		15:0		<b>Core_COUNT (R0)</b> The number of processor cores that are currently enabled (by either factory configuration or BIOS configuration) in the physical package.
		31:16		<b>THREAD_COUNT (R0)</b> The number of logical processors that are currently enabled (by either factory configuration or BIOS configuration) in the physical package.
		63:32		Reserved
53H	83	MSR_THREAD_ID_INFO	Thread	<b>A Hardware Assigned ID for the Logical Processor (R0)</b>
		7:0		<b>Logical_Processor_ID (R0)</b> An implementation-specific numerical. value physically assigned to each logical processor. This ID is not related to Initial APIC ID or x2APIC ID, it is unique within a physical package.
		63:8		Reserved

Table 35-30. Additional MSRs Supported by Intel® Xeon® Processor E5 v3 Family

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
E2H	226	MSR_PKG_CST_CONFIG_CONTROL	Core	<b>C-State Configuration Control (R/W)</b> Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-states. See <a href="http://biosbits.org">http://biosbits.org</a> .
		2:0		<b>Package C-State Limit (R/W)</b> Specifies the lowest processor-specific C-state code name (consuming the least power) for the package. The default is set as factory-configured package C-state limit. The following C-state code name encodings are supported: 000b: C0/C1 (no package C-state support) 001b: C2 010b: C6 (non-retention) 011b: C6 (retention) 111b: No Package C state limits. All C states supported by the processor are available.
		9:3		Reserved
		10		<b>I/O MWAIT Redirection Enable (R/W)</b>
		14:11		Reserved
		15		<b>CFG Lock (R/WO)</b>
		24:16		Reserved
		25		<b>C3 State Auto Demotion Enable (R/W)</b>
		26		<b>C1 State Auto Demotion Enable (R/W)</b>
		27		<b>Enable C3 Undemotion (R/W)</b>
		28		<b>Enable C1 Undemotion (R/W)</b>
		29		<b>Package C State Demotion Enable (R/W)</b>
		30		<b>Package C State UnDemotion Enable (R/W)</b>
		63:31		Reserved
179H	377	IA32_MCG_CAP	Thread	<b>Global Machine Check Capability (R/O)</b>
		7:0		<b>Count</b>
		8		<b>MCG_CTL_P</b>
		9		<b>MCG_EXT_P</b>
		10		<b>MCP_CMCI_P</b>
		11		<b>MCG_TES_P</b>
		15:12		Reserved.
		23:16		<b>MCG_EXT_CNT</b>
		24		<b>MCG_SER_P</b>
		25		<b>MCG_EM_P</b>
		26		<b>MCG_ELOG_P</b>
		63:27		Reserved.

**Table 35-30. Additional MSRs Supported by Intel® Xeon® Processor E5 v3 Family**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
17DH	390	MSR_SMM_MCA_CAP	THREAD	<b>Enhanced SMM Capabilities (SMM-RO)</b> Reports SMM capability Enhancement. Accessible only while in SMM.
		57:0		<b>Reserved</b>
		58		<b>SMM_Code_Access_Chk (SMM-RO)</b> If set to 1 indicates that the SMM code access restriction is supported and a host-space interface available to SMM handler.
		59		<b>Long_Flow_Indication (SMM-RO)</b> If set to 1 indicates that the SMM long flow indicator is supported and a host-space interface available to SMM handler.
		63:60		Reserved
17FH	383	MSR_ERROR_CONTROL	Package	<b>MC Bank Error Configuration (R/W)</b>
		0		Reserved
		1		<b>MemError Log Enable (R/W)</b> When set, enables IMC status bank to log additional info in bits 36:32.
		63:2		Reserved.
1ADH	429	MSR_TURBO_RATIO_LIMIT	Package	<b>Maximum Ratio Limit of Turbo Mode</b> RO if MSR_PLATFORM_INFO.[28] = 0, RW if MSR_PLATFORM_INFO.[28] = 1
		7:0	Package	<b>Maximum Ratio Limit for 1C</b> Maximum turbo ratio limit of 1 core active.
		15:8	Package	<b>Maximum Ratio Limit for 2C</b> Maximum turbo ratio limit of 2 core active.
		23:16	Package	<b>Maximum Ratio Limit for 3C</b> Maximum turbo ratio limit of 3 core active.
		31:24	Package	<b>Maximum Ratio Limit for 4C</b> Maximum turbo ratio limit of 4 core active.
		39:32	Package	<b>Maximum Ratio Limit for 5C</b> Maximum turbo ratio limit of 5 core active.
		47:40	Package	<b>Maximum Ratio Limit for 6C</b> Maximum turbo ratio limit of 6 core active.
		55:48	Package	<b>Maximum Ratio Limit for 7C</b> Maximum turbo ratio limit of 7 core active.
		63:56	Package	<b>Maximum Ratio Limit for 8C</b> Maximum turbo ratio limit of 8 core active.
1AEH	430	MSR_TURBO_RATIO_LIMIT1	Package	<b>Maximum Ratio Limit of Turbo Mode</b> RO if MSR_PLATFORM_INFO.[28] = 0, RW if MSR_PLATFORM_INFO.[28] = 1

Table 35-30. Additional MSRs Supported by Intel® Xeon® Processor E5 v3 Family

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		7:0	Package	<b>Maximum Ratio Limit for 9C</b> Maximum turbo ratio limit of 9 core active.
		15:8	Package	<b>Maximum Ratio Limit for 10C</b> Maximum turbo ratio limit of 10 core active.
		23:16	Package	<b>Maximum Ratio Limit for 11C</b> Maximum turbo ratio limit of 11 core active.
		31:24	Package	<b>Maximum Ratio Limit for 12C</b> Maximum turbo ratio limit of 12 core active.
		39:32	Package	<b>Maximum Ratio Limit for 13C</b> Maximum turbo ratio limit of 13 core active.
		47:40	Package	<b>Maximum Ratio Limit for 14C</b> Maximum turbo ratio limit of 14 core active.
		55:48	Package	<b>Maximum Ratio Limit for 15C</b> Maximum turbo ratio limit of 15 core active.
		63:56	Package	<b>Maximum Ratio Limit for 16C</b> Maximum turbo ratio limit of 16 core active.
1AFH	431	MSR_TURBO_RATIO_LIMIT2	Package	<b>Maximum Ratio Limit of Turbo Mode</b> RO if MSR_PLATFORM_INFO.[28] = 0, RW if MSR_PLATFORM_INFO.[28] = 1
		7:0	Package	<b>Maximum Ratio Limit for 17C</b> Maximum turbo ratio limit of 17 core active.
		15:8	Package	<b>Maximum Ratio Limit for 18C</b> Maximum turbo ratio limit of 18 core active.
		62:16	Package	Reserved
		63	Package	<b>Semaphore for Turbo Ratio Limit Configuration</b> If 1, the processor uses override configuration <sup>1</sup> specified in MSR_TURBO_RATIO_LIMIT, MSR_TURBO_RATIO_LIMIT1 and MSR_TURBO_RATIO_LIMIT2. If 0, the processor uses factory-set configuration (Default).
414H	1044	IA32_MC5_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Bank MC5 reports MC error from the Intel QPI 0 module.
415H	1045	IA32_MC5_STATUS	Package	
416H	1046	IA32_MC5_ADDR	Package	
417H	1047	IA32_MC5_MISC	Package	
418H	1048	IA32_MC6_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Bank MC6 reports MC error from the integrated I/O module.
419H	1049	IA32_MC6_STATUS	Package	
41AH	1050	IA32_MC6_ADDR	Package	
41BH	1051	IA32_MC6_MISC	Package	

**Table 35-30. Additional MSRs Supported by Intel® Xeon® Processor E5 v3 Family**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
41CH	1052	IA32_MC7_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Bank MC7 reports MC error from the home agent HA 0.
41DH	1053	IA32_MC7_STATUS	Package	
41EH	1054	IA32_MC7_ADDR	Package	
41FH	1055	IA32_MC7_MISC	Package	
420H	1056	IA32_MC8_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Bank MC8 reports MC error from the home agent HA 1.
421H	1057	IA32_MC8_STATUS	Package	
422H	1058	IA32_MC8_ADDR	Package	
423H	1059	IA32_MC8_MISC	Package	
424H	1060	IA32_MC9_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Banks MC9 through MC 16 report MC error from each channel of the integrated memory controllers.
425H	1061	IA32_MC9_STATUS	Package	
426H	1062	IA32_MC9_ADDR	Package	
427H	1063	IA32_MC9_MISC	Package	
428H	1064	IA32_MC10_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Banks MC9 through MC 16 report MC error from each channel of the integrated memory controllers.
429H	1065	IA32_MC10_STATUS	Package	
42AH	1066	IA32_MC10_ADDR	Package	
42BH	1067	IA32_MC10_MISC	Package	
42CH	1068	IA32_MC11_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Banks MC9 through MC 16 report MC error from each channel of the integrated memory controllers.
42DH	1069	IA32_MC11_STATUS	Package	
42EH	1070	IA32_MC11_ADDR	Package	
42FH	1071	IA32_MC11_MISC	Package	
430H	1072	IA32_MC12_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Banks MC9 through MC 16 report MC error from each channel of the integrated memory controllers.
431H	1073	IA32_MC12_STATUS	Package	
432H	1074	IA32_MC12_ADDR	Package	
433H	1075	IA32_MC12_MISC	Package	
434H	1076	IA32_MC13_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Banks MC9 through MC 16 report MC error from each channel of the integrated memory controllers.
435H	1077	IA32_MC13_STATUS	Package	
436H	1078	IA32_MC13_ADDR	Package	
437H	1079	IA32_MC13_MISC	Package	
438H	1080	IA32_MC14_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Banks MC9 through MC 16 report MC error from each channel of the integrated memory controllers.
439H	1081	IA32_MC14_STATUS	Package	
43AH	1082	IA32_MC14_ADDR	Package	
43BH	1083	IA32_MC14_MISC	Package	
43CH	1084	IA32_MC15_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Banks MC9 through MC 16 report MC error from each channel of the integrated memory controllers.
43DH	1085	IA32_MC15_STATUS	Package	
43EH	1086	IA32_MC15_ADDR	Package	
43FH	1087	IA32_MC15_MISC	Package	

Table 35-30. Additional MSRs Supported by Intel® Xeon® Processor E5 v3 Family

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
440H	1088	IA32_MC16_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Banks MC9 through MC 16 report MC error from each channel of the integrated memory controllers.
441H	1089	IA32_MC16_STATUS	Package	
442H	1090	IA32_MC16_ADDR	Package	
443H	1091	IA32_MC16_MISC	Package	
444H	1092	IA32_MC17_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Bank MC17 reports MC error from the following pair of CBo/L3 Slices (if the pair is present): CBo0, CBo3, CBo6, CBo9, CBo12, CBo15.
445H	1093	IA32_MC17_STATUS	Package	
446H	1094	IA32_MC17_ADDR	Package	
447H	1095	IA32_MC17_MISC	Package	
448H	1096	IA32_MC18_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Bank MC18 reports MC error from the following pair of CBo/L3 Slices (if the pair is present): CBo1, CBo4, CBo7, CBo10, CBo13, CBo16.
449H	1097	IA32_MC18_STATUS	Package	
44AH	1098	IA32_MC18_ADDR	Package	
44BH	1099	IA32_MC18_MISC	Package	
44CH	1100	IA32_MC19_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Bank MC19 reports MC error from the following pair of CBo/L3 Slices (if the pair is present): CBo2, CBo5, CBo8, CBo11, CBo14, CBo17.
44DH	1101	IA32_MC19_STATUS	Package	
44EH	1102	IA32_MC19_ADDR	Package	
44FH	1103	IA32_MC19_MISC	Package	
450H	1104	IA32_MC20_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Bank MC20 reports MC error from the Intel QPI 1 module.
451H	1105	IA32_MC20_STATUS	Package	
452H	1106	IA32_MC20_ADDR	Package	
453H	1107	IA32_MC20_MISC	Package	
454H	1108	IA32_MC21_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Bank MC21 reports MC error from the Intel QPI 2 module.
455H	1109	IA32_MC21_STATUS	Package	
456H	1110	IA32_MC21_ADDR	Package	
457H	1111	IA32_MC21_MISC	Package	
606H	1542	MSR_RAPL_POWER_UNIT	Package	<b>Unit Multipliers used in RAPL Interfaces (R/O)</b>
		3:0	Package	<b>Power Units</b> See Section 14.9.1, "RAPL Interfaces."
		7:4	Package	<b>Reserved</b>
		12:8	Package	<b>Energy Status Units</b> Energy related information (in Joules) is based on the multiplier, $1/2^{\text{ESU}}$ ; where ESU is an unsigned integer represented by bits 12:8. Default value is 0EH (or 61 micro-joules)
		15:13	Package	<b>Reserved</b>
		19:16	Package	<b>Time Units</b> See Section 14.9.1, "RAPL Interfaces."
		63:20		Reserved
618H	1560	MSR_DRAM_POWER_LIMIT	Package	<b>DRAM RAPL Power Limit Control (R/W)</b> See Section 14.9.5, "DRAM RAPL Domain."

**Table 35-30. Additional MSRs Supported by Intel® Xeon® Processor E5 v3 Family**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
619H	1561	MSR_DRAM_ENERGY_STATUS	Package	<b>DRAM Energy Status (R/O)</b> Energy Consumed by DRAM devices.
		31:0		Energy in 15.3 micro-joules. Requires BIOS configuration to enable DRAM RAPL mode 0 (Direct VR).
		63:32		Reserved
61BH	1563	MSR_DRAM_PERF_STATUS	Package	<b>DRAM Performance Throttling Status (R/O)</b> See Section 14.9.5, "DRAM RAPL Domain."
61CH	1564	MSR_DRAM_POWER_INFO	Package	<b>DRAM RAPL Parameters (R/W)</b> See Section 14.9.5, "DRAM RAPL Domain."
61EH	1566	MSR_PCIE_PLL_RATIO	Package	<b>Configuration of PCIE PLL Relative to BCLK(R/W)</b>
		1:0	Package	<b>PCIE Ratio (R/W)</b> 00b: Use 5:5 mapping for 100MHz operation (default) 01b: Use 5:4 mapping for 125MHz operation 10b: Use 5:3 mapping for 166MHz operation 11b: Use 5:2 mapping for 250MHz operation
		2	Package	<b>LPLL Select (R/W)</b> if 1, use configured setting of PCIE Ratio
		3	Package	<b>LONG RESET (R/W)</b> if 1, wait additional time-out before re-locking Gen2/Gen3 PLLs.
		63:4		Reserved
639H	1593	MSR_PPO_ENERGY_STATUS	Package	<b>Reserved (R/O)</b> Reads return 0
690H	1680	MSR_CORE_PERF_LIMIT_REASONS	Package	<b>Indicator of Frequency Clipping in Processor Cores (R/W) (frequency refers to processor core frequency)</b>
		0		<b>PROCHOT Status (R0)</b> When set, processor core frequency is reduced below the operating system request due to assertion of external PROCHOT.
		1		<b>Thermal Status (R0)</b> When set, frequency is reduced below the operating system request due to a thermal event.
		2		<b>Power Budget Management Status (R0)</b> When set, frequency is reduced below the operating system request due to PBM limit
		3		<b>Platform Configuration Services Status (R0)</b> When set, frequency is reduced below the operating system request due to PCS limit
		4		Reserved.
		5		<b>Autonomous Utilization-Based Frequency Control Status (R0)</b> When set, frequency is reduced below the operating system request because the processor has detected that utilization is low

Table 35-30. Additional MSRs Supported by Intel® Xeon® Processor E5 v3 Family

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		6		<b>VR Therm Alert Status (R0)</b> When set, frequency is reduced below the operating system request due to a thermal alert from the Voltage Regulator.
		7		Reserved.
		8		<b>Electrical Design Point Status (R0)</b> When set, frequency is reduced below the operating system request due to electrical design point constraints (e.g. maximum electrical current consumption).
		9		Reserved.
		10		<b>Multi-Core Turbo Status (R0)</b> When set, frequency is reduced below the operating system request due to Multi-Core Turbo limits
		12:11		Reserved.
		13		<b>Core Frequency P1 Status (R0)</b> When set, frequency is reduced below max non-turbo P1
		14		<b>Core Max n-core Turbo Frequency Limiting Status (R0)</b> When set, frequency is reduced below max n-core turbo frequency
		15		<b>Core Frequency Limiting Status (R0)</b> When set, frequency is reduced below the operating system request.
		16		<b>PROCHOT Log</b> When set, indicates that the PROCHOT Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		17		<b>Thermal Log</b> When set, indicates that the Thermal Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		18		<b>Power Budget Management Log</b> When set, indicates that the PBM Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		19		<b>Platform Configuration Services Log</b> When set, indicates that the PCS Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		20		Reserved.
		21		<b>Autonomous Utilization-Based Frequency Control Log</b> When set, indicates that the AUBFC Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.



**Table 35-30. Additional MSRs Supported by Intel® Xeon® Processor E5 v3 Family**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		22		<b>VR Therm Alert Log</b> When set, indicates that the VR Therm Alert Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		23		Reserved.
		24		<b>Electrical Design Point Log</b> When set, indicates that the EDP Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		25		Reserved.
		26		<b>Multi-Core Turbo Log</b> When set, indicates that the Multi-Core Turbo Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		28:27		Reserved.
		29		<b>Core Frequency P1 Log</b> When set, indicates that the Core Frequency P1 Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		30		<b>Core Max n-core Turbo Frequency Limiting Log</b> When set, indicates that the Core Max n-core Turbo Frequency Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		31		<b>Core Frequency Limiting Log</b> When set, indicates that the Core Frequency Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		63:32		Reserved.
C8DH	3213	IA32_QM_EVTSEL	THREAD	<b>Monitoring Event Select Register (R/W).</b> if CPUID.(EAX=07H, ECX=0):EBX.RDT-M[bit 12] = 1
		7:0		<b>EventID (RW)</b> <b>Event encoding:</b> <b>0x0: no monitoring</b> <b>0x1: L3 occupancy monitoring</b> <b>all other encoding reserved.</b>
		31:8		Reserved.
		41:32		<b>RMID (RW)</b>
		63:42		Reserved.
C8EH	3214	IA32_QM_CTR	THREAD	<b>Monitoring Counter Register (R/O).</b> if CPUID.(EAX=07H, ECX=0):EBX.RDT-M[bit 12] = 1
		61:0		<b>Resource Monitored Data</b>

**Table 35-30. Additional MSRs Supported by Intel® Xeon® Processor E5 v3 Family**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		62		<b>Unavailable:</b> If 1, indicates data for this RMID is not available or not monitored for this resource or RMID.
		63		<b>Error:</b> If 1, indicates and unsupported RMID or event type was written to IA32_PQR_QM_EVTSEL.
C8FH	3215	IA32_PQR_ASSOC	THREAD	<b>Resource Association Register (R/W).</b>
		9:0		<b>RMID</b>
		63: 10		<b>Reserved</b>

See Table 35-18, Table 35-27 for other MSR definitions applicable to processors with CPUID signature 06\_3FH.

**NOTES:**

1. An override configuration lower than the factory-set configuration is always supported. An override configuration higher than the factory-set configuration is dependent on features specific to the processor and the platform.

**35.12.1 Additional Uncore PMU MSRs in the Intel® Xeon® Processor E5 v3 Family**

Intel Xeon Processor E5 v3 and E7 v3 family are based on the Haswell-E microarchitecture. The MSR-based uncore PMU interfaces are listed in Table 35-31. For complete detail of the uncore PMU, refer to Intel Xeon Processor E5 v3 Product Family Uncore Performance Monitoring Guide. These processors have a CPUID signature with DisplayFamily\_DisplayModel of 06\_3FH.

**Table 35-31. Uncore PMU MSRs in Intel® Xeon® Processor E5 v3 Family**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
700H		MSR_PMON_GLOBAL_CTL	Package	Uncore perfmon per-socket global control.
701H		MSR_PMON_GLOBAL_STATUS	Package	Uncore perfmon per-socket global status.
702H		MSR_PMON_GLOBAL_CONFIG	Package	Uncore perfmon per-socket global configuration.
703H		MSR_U_PMON_UCLK_FIXED_CTL	Package	Uncore U-box UCLK fixed counter control
704H		MSR_U_PMON_UCLK_FIXED_CTR	Package	Uncore U-box UCLK fixed counter
705H		MSR_U_PMON_EVTSELO	Package	Uncore U-box perfmon event select for U-box counter 0.
706H		MSR_U_PMON_EVTSEL1	Package	Uncore U-box perfmon event select for U-box counter 1.
708H		MSR_U_PMON_BOX_STATUS	Package	Uncore U-box perfmon U-box wide status.
709H		MSR_U_PMON_CTR0	Package	Uncore U-box perfmon counter 0
70AH		MSR_U_PMON_CTR1	Package	Uncore U-box perfmon counter 1
710H		MSR_PCU_PMON_BOX_CTL	Package	Uncore PCU perfmon for PCU-box-wide control
711H		MSR_PCU_PMON_EVTSELO	Package	Uncore PCU perfmon event select for PCU counter 0.
712H		MSR_PCU_PMON_EVTSEL1	Package	Uncore PCU perfmon event select for PCU counter 1.
713H		MSR_PCU_PMON_EVTSEL2	Package	Uncore PCU perfmon event select for PCU counter 2.
714H		MSR_PCU_PMON_EVTSEL3	Package	Uncore PCU perfmon event select for PCU counter 3.
715H		MSR_PCU_PMON_BOX_FILTER	Package	Uncore PCU perfmon box-wide filter.
716H		MSR_PCU_PMON_BOX_STATUS	Package	Uncore PCU perfmon box wide status.

**Table 35-31. Uncore PMU MSRs in Intel® Xeon® Processor E5 v3 Family (Contd.)**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
717H		MSR_PCU_PMON_CTRL0	Package	Uncore PCU perfmon counter 0.
718H		MSR_PCU_PMON_CTRL1	Package	Uncore PCU perfmon counter 1.
719H		MSR_PCU_PMON_CTRL2	Package	Uncore PCU perfmon counter 2.
71AH		MSR_PCU_PMON_CTRL3	Package	Uncore PCU perfmon counter 3.
720H		MSR_S0_PMON_BOX_CTL	Package	Uncore SBo 0 perfmon for SBo 0 box-wide control
721H		MSR_S0_PMON_EVTSEL0	Package	Uncore SBo 0 perfmon event select for SBo 0 counter 0.
722H		MSR_S0_PMON_EVTSEL1	Package	Uncore SBo 0 perfmon event select for SBo 0 counter 1.
723H		MSR_S0_PMON_EVTSEL2	Package	Uncore SBo 0 perfmon event select for SBo 0 counter 2.
724H		MSR_S0_PMON_EVTSEL3	Package	Uncore SBo 0 perfmon event select for SBo 0 counter 3.
725H		MSR_S0_PMON_BOX_FILTER	Package	Uncore SBo 0 perfmon box-wide filter.
726H		MSR_S0_PMON_CTRL0	Package	Uncore SBo 0 perfmon counter 0.
727H		MSR_S0_PMON_CTRL1	Package	Uncore SBo 0 perfmon counter 1.
728H		MSR_S0_PMON_CTRL2	Package	Uncore SBo 0 perfmon counter 2.
729H		MSR_S0_PMON_CTRL3	Package	Uncore SBo 0 perfmon counter 3.
72AH		MSR_S1_PMON_BOX_CTL	Package	Uncore SBo 1 perfmon for SBo 1 box-wide control
72BH		MSR_S1_PMON_EVTSEL0	Package	Uncore SBo 1 perfmon event select for SBo 1 counter 0.
72CH		MSR_S1_PMON_EVTSEL1	Package	Uncore SBo 1 perfmon event select for SBo 1 counter 1.
72DH		MSR_S1_PMON_EVTSEL2	Package	Uncore SBo 1 perfmon event select for SBo 1 counter 2.
72EH		MSR_S1_PMON_EVTSEL3	Package	Uncore SBo 1 perfmon event select for SBo 1 counter 3.
72FH		MSR_S1_PMON_BOX_FILTER	Package	Uncore SBo 1 perfmon box-wide filter.
730H		MSR_S1_PMON_CTRL0	Package	Uncore SBo 1 perfmon counter 0.
731H		MSR_S1_PMON_CTRL1	Package	Uncore SBo 1 perfmon counter 1.
732H		MSR_S1_PMON_CTRL2	Package	Uncore SBo 1 perfmon counter 2.
733H		MSR_S1_PMON_CTRL3	Package	Uncore SBo 1 perfmon counter 3.
734H		MSR_S2_PMON_BOX_CTL	Package	Uncore SBo 2 perfmon for SBo 2 box-wide control
735H		MSR_S2_PMON_EVTSEL0	Package	Uncore SBo 2 perfmon event select for SBo 2 counter 0.
736H		MSR_S2_PMON_EVTSEL1	Package	Uncore SBo 2 perfmon event select for SBo 2 counter 1.
737H		MSR_S2_PMON_EVTSEL2	Package	Uncore SBo 2 perfmon event select for SBo 2 counter 2.
738H		MSR_S2_PMON_EVTSEL3	Package	Uncore SBo 2 perfmon event select for SBo 2 counter 3.
739H		MSR_S2_PMON_BOX_FILTER	Package	Uncore SBo 2 perfmon box-wide filter.
73AH		MSR_S2_PMON_CTRL0	Package	Uncore SBo 2 perfmon counter 0.
73BH		MSR_S2_PMON_CTRL1	Package	Uncore SBo 2 perfmon counter 1.
73CH		MSR_S2_PMON_CTRL2	Package	Uncore SBo 2 perfmon counter 2.
73DH		MSR_S2_PMON_CTRL3	Package	Uncore SBo 2 perfmon counter 3.
73EH		MSR_S3_PMON_BOX_CTL	Package	Uncore SBo 3 perfmon for SBo 3 box-wide control
73FH		MSR_S3_PMON_EVTSEL0	Package	Uncore SBo 3 perfmon event select for SBo 3 counter 0.
740H		MSR_S3_PMON_EVTSEL1	Package	Uncore SBo 3 perfmon event select for SBo 3 counter 1.

Table 35-31. Uncore PMU MSRs in Intel® Xeon® Processor E5 v3 Family (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
741H		MSR_S3_PMON_EVNTSEL2	Package	Uncore SBo 3 perfmon event select for SBo 3 counter 2.
742H		MSR_S3_PMON_EVNTSEL3	Package	Uncore SBo 3 perfmon event select for SBo 3 counter 3.
743H		MSR_S3_PMON_BOX_FILTER	Package	Uncore SBo 3 perfmon box-wide filter.
744H		MSR_S3_PMON_CTR0	Package	Uncore SBo 3 perfmon counter 0.
745H		MSR_S3_PMON_CTR1	Package	Uncore SBo 3 perfmon counter 1.
746H		MSR_S3_PMON_CTR2	Package	Uncore SBo 3 perfmon counter 2.
747H		MSR_S3_PMON_CTR3	Package	Uncore SBo 3 perfmon counter 3.
E00H		MSR_C0_PMON_BOX_CTL	Package	Uncore C-box 0 perfmon for box-wide control
E01H		MSR_C0_PMON_EVNTSEL0	Package	Uncore C-box 0 perfmon event select for C-box 0 counter 0.
E02H		MSR_C0_PMON_EVNTSEL1	Package	Uncore C-box 0 perfmon event select for C-box 0 counter 1.
E03H		MSR_C0_PMON_EVNTSEL2	Package	Uncore C-box 0 perfmon event select for C-box 0 counter 2.
E04H		MSR_C0_PMON_EVNTSEL3	Package	Uncore C-box 0 perfmon event select for C-box 0 counter 3.
E05H		MSR_C0_PMON_BOX_FILTER0	Package	Uncore C-box 0 perfmon box wide filter 0.
E06H		MSR_C0_PMON_BOX_FILTER1	Package	Uncore C-box 0 perfmon box wide filter 1.
E07H		MSR_C0_PMON_BOX_STATUS	Package	Uncore C-box 0 perfmon box wide status.
E08H		MSR_C0_PMON_CTR0	Package	Uncore C-box 0 perfmon counter 0.
E09H		MSR_C0_PMON_CTR1	Package	Uncore C-box 0 perfmon counter 1.
E0AH		MSR_C0_PMON_CTR2	Package	Uncore C-box 0 perfmon counter 2.
E0BH		MSR_C0_PMON_CTR3	Package	Uncore C-box 0 perfmon counter 3.
E10H		MSR_C1_PMON_BOX_CTL	Package	Uncore C-box 1 perfmon for box-wide control
E11H		MSR_C1_PMON_EVNTSEL0	Package	Uncore C-box 1 perfmon event select for C-box 1 counter 0.
E12H		MSR_C1_PMON_EVNTSEL1	Package	Uncore C-box 1 perfmon event select for C-box 1 counter 1.
E13H		MSR_C1_PMON_EVNTSEL2	Package	Uncore C-box 1 perfmon event select for C-box 1 counter 2.
E14H		MSR_C1_PMON_EVNTSEL3	Package	Uncore C-box 1 perfmon event select for C-box 1 counter 3.
E15H		MSR_C1_PMON_BOX_FILTER0	Package	Uncore C-box 1 perfmon box wide filter 0.
E16H		MSR_C1_PMON_BOX_FILTER1	Package	Uncore C-box 1 perfmon box wide filter1.
E17H		MSR_C1_PMON_BOX_STATUS	Package	Uncore C-box 1 perfmon box wide status.
E18H		MSR_C1_PMON_CTR0	Package	Uncore C-box 1 perfmon counter 0.
E19H		MSR_C1_PMON_CTR1	Package	Uncore C-box 1 perfmon counter 1.
E1AH		MSR_C1_PMON_CTR2	Package	Uncore C-box 1 perfmon counter 2.
E1BH		MSR_C1_PMON_CTR3	Package	Uncore C-box 1 perfmon counter 3.
E20H		MSR_C2_PMON_BOX_CTL	Package	Uncore C-box 2 perfmon for box-wide control
E21H		MSR_C2_PMON_EVNTSEL0	Package	Uncore C-box 2 perfmon event select for C-box 2 counter 0.
E22H		MSR_C2_PMON_EVNTSEL1	Package	Uncore C-box 2 perfmon event select for C-box 2 counter 1.
E23H		MSR_C2_PMON_EVNTSEL2	Package	Uncore C-box 2 perfmon event select for C-box 2 counter 2.
E24H		MSR_C2_PMON_EVNTSEL3	Package	Uncore C-box 2 perfmon event select for C-box 2 counter 3.
E25H		MSR_C2_PMON_BOX_FILTER0	Package	Uncore C-box 2 perfmon box wide filter 0.

**Table 35-31. Uncore PMU MSRs in Intel® Xeon® Processor E5 v3 Family (Contd.)**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
E26H		MSR_C2_PMON_BOX_FILTER1	Package	Uncore C-box 2 perfmon box wide filter1.
E27H		MSR_C2_PMON_BOX_STATUS	Package	Uncore C-box 2 perfmon box wide status.
E28H		MSR_C2_PMON_CTR0	Package	Uncore C-box 2 perfmon counter 0.
E29H		MSR_C2_PMON_CTR1	Package	Uncore C-box 2 perfmon counter 1.
E2AH		MSR_C2_PMON_CTR2	Package	Uncore C-box 2 perfmon counter 2.
E2BH		MSR_C2_PMON_CTR3	Package	Uncore C-box 2 perfmon counter 3.
E30H		MSR_C3_PMON_BOX_CTL	Package	Uncore C-box 3 perfmon for box-wide control
E31H		MSR_C3_PMON_EVNTSELO	Package	Uncore C-box 3 perfmon event select for C-box 3 counter 0.
E32H		MSR_C3_PMON_EVNTSEL1	Package	Uncore C-box 3 perfmon event select for C-box 3 counter 1.
E33H		MSR_C3_PMON_EVNTSEL2	Package	Uncore C-box 3 perfmon event select for C-box 3 counter 2.
E34H		MSR_C3_PMON_EVNTSEL3	Package	Uncore C-box 3 perfmon event select for C-box 3 counter 3.
E35H		MSR_C3_PMON_BOX_FILTER0	Package	Uncore C-box 3 perfmon box wide filter 0.
E36H		MSR_C3_PMON_BOX_FILTER1	Package	Uncore C-box 3 perfmon box wide filter1.
E37H		MSR_C3_PMON_BOX_STATUS	Package	Uncore C-box 3 perfmon box wide status.
E38H		MSR_C3_PMON_CTR0	Package	Uncore C-box 3 perfmon counter 0.
E39H		MSR_C3_PMON_CTR1	Package	Uncore C-box 3 perfmon counter 1.
E3AH		MSR_C3_PMON_CTR2	Package	Uncore C-box 3 perfmon counter 2.
E3BH		MSR_C3_PMON_CTR3	Package	Uncore C-box 3 perfmon counter 3.
E40H		MSR_C4_PMON_BOX_CTL	Package	Uncore C-box 4 perfmon for box-wide control
E41H		MSR_C4_PMON_EVNTSELO	Package	Uncore C-box 4 perfmon event select for C-box 4 counter 0.
E42H		MSR_C4_PMON_EVNTSEL1	Package	Uncore C-box 4 perfmon event select for C-box 4 counter 1.
E43H		MSR_C4_PMON_EVNTSEL2	Package	Uncore C-box 4 perfmon event select for C-box 4 counter 2.
E44H		MSR_C4_PMON_EVNTSEL3	Package	Uncore C-box 4 perfmon event select for C-box 4 counter 3.
E45H		MSR_C4_PMON_BOX_FILTER0	Package	Uncore C-box 4 perfmon box wide filter 0.
E46H		MSR_C4_PMON_BOX_FILTER1	Package	Uncore C-box 4 perfmon box wide filter1.
E47H		MSR_C4_PMON_BOX_STATUS	Package	Uncore C-box 4 perfmon box wide status.
E48H		MSR_C4_PMON_CTR0	Package	Uncore C-box 4 perfmon counter 0.
E49H		MSR_C4_PMON_CTR1	Package	Uncore C-box 4 perfmon counter 1.
E4AH		MSR_C4_PMON_CTR2	Package	Uncore C-box 4 perfmon counter 2.
E4BH		MSR_C4_PMON_CTR3	Package	Uncore C-box 4 perfmon counter 3.
E50H		MSR_C5_PMON_BOX_CTL	Package	Uncore C-box 5 perfmon for box-wide control
E51H		MSR_C5_PMON_EVNTSELO	Package	Uncore C-box 5 perfmon event select for C-box 5 counter 0.
E52H		MSR_C5_PMON_EVNTSEL1	Package	Uncore C-box 5 perfmon event select for C-box 5 counter 1.
E53H		MSR_C5_PMON_EVNTSEL2	Package	Uncore C-box 5 perfmon event select for C-box 5 counter 2.
E54H		MSR_C5_PMON_EVNTSEL3	Package	Uncore C-box 5 perfmon event select for C-box 5 counter 3.
E55H		MSR_C5_PMON_BOX_FILTER0	Package	Uncore C-box 5 perfmon box wide filter 0.
E56H		MSR_C5_PMON_BOX_FILTER1	Package	Uncore C-box 5 perfmon box wide filter1.

Table 35-31. Uncore PMU MSRs in Intel® Xeon® Processor E5 v3 Family (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
E57H		MSR_C5_PMON_BOX_STATUS	Package	Uncore C-box 5 perfmon box wide status.
E58H		MSR_C5_PMON_CTRL0	Package	Uncore C-box 5 perfmon counter 0.
E59H		MSR_C5_PMON_CTRL1	Package	Uncore C-box 5 perfmon counter 1.
E5AH		MSR_C5_PMON_CTRL2	Package	Uncore C-box 5 perfmon counter 2.
E5BH		MSR_C5_PMON_CTRL3	Package	Uncore C-box 5 perfmon counter 3.
E60H		MSR_C6_PMON_BOX_CTL	Package	Uncore C-box 6 perfmon for box-wide control
E61H		MSR_C6_PMON_EVTSEL0	Package	Uncore C-box 6 perfmon event select for C-box 6 counter 0.
E62H		MSR_C6_PMON_EVTSEL1	Package	Uncore C-box 6 perfmon event select for C-box 6 counter 1.
E63H		MSR_C6_PMON_EVTSEL2	Package	Uncore C-box 6 perfmon event select for C-box 6 counter 2.
E64H		MSR_C6_PMON_EVTSEL3	Package	Uncore C-box 6 perfmon event select for C-box 6 counter 3.
E65H		MSR_C6_PMON_BOX_FILTER0	Package	Uncore C-box 6 perfmon box wide filter 0.
E66H		MSR_C6_PMON_BOX_FILTER1	Package	Uncore C-box 6 perfmon box wide filter 1.
E67H		MSR_C6_PMON_BOX_STATUS	Package	Uncore C-box 6 perfmon box wide status.
E68H		MSR_C6_PMON_CTRL0	Package	Uncore C-box 6 perfmon counter 0.
E69H		MSR_C6_PMON_CTRL1	Package	Uncore C-box 6 perfmon counter 1.
E6AH		MSR_C6_PMON_CTRL2	Package	Uncore C-box 6 perfmon counter 2.
E6BH		MSR_C6_PMON_CTRL3	Package	Uncore C-box 6 perfmon counter 3.
E70H		MSR_C7_PMON_BOX_CTL	Package	Uncore C-box 7 perfmon for box-wide control.
E71H		MSR_C7_PMON_EVTSEL0	Package	Uncore C-box 7 perfmon event select for C-box 7 counter 0.
E72H		MSR_C7_PMON_EVTSEL1	Package	Uncore C-box 7 perfmon event select for C-box 7 counter 1.
E73H		MSR_C7_PMON_EVTSEL2	Package	Uncore C-box 7 perfmon event select for C-box 7 counter 2.
E74H		MSR_C7_PMON_EVTSEL3	Package	Uncore C-box 7 perfmon event select for C-box 7 counter 3.
E75H		MSR_C7_PMON_BOX_FILTER0	Package	Uncore C-box 7 perfmon box wide filter 0.
E76H		MSR_C7_PMON_BOX_FILTER1	Package	Uncore C-box 7 perfmon box wide filter 1.
E77H		MSR_C7_PMON_BOX_STATUS	Package	Uncore C-box 7 perfmon box wide status.
E78H		MSR_C7_PMON_CTRL0	Package	Uncore C-box 7 perfmon counter 0.
E79H		MSR_C7_PMON_CTRL1	Package	Uncore C-box 7 perfmon counter 1.
E7AH		MSR_C7_PMON_CTRL2	Package	Uncore C-box 7 perfmon counter 2.
E7BH		MSR_C7_PMON_CTRL3	Package	Uncore C-box 7 perfmon counter 3.
E80H		MSR_C8_PMON_BOX_CTL	Package	Uncore C-box 8 perfmon local box wide control.
E81H		MSR_C8_PMON_EVTSEL0	Package	Uncore C-box 8 perfmon event select for C-box 8 counter 0.
E82H		MSR_C8_PMON_EVTSEL1	Package	Uncore C-box 8 perfmon event select for C-box 8 counter 1.
E83H		MSR_C8_PMON_EVTSEL2	Package	Uncore C-box 8 perfmon event select for C-box 8 counter 2.
E84H		MSR_C8_PMON_EVTSEL3	Package	Uncore C-box 8 perfmon event select for C-box 8 counter 3.
E85H		MSR_C8_PMON_BOX_FILTER0	Package	Uncore C-box 8 perfmon box wide filter 0.
E86H		MSR_C8_PMON_BOX_FILTER1	Package	Uncore C-box 8 perfmon box wide filter 1.
E87H		MSR_C8_PMON_BOX_STATUS	Package	Uncore C-box 8 perfmon box wide status.

**Table 35-31. Uncore PMU MSRs in Intel® Xeon® Processor E5 v3 Family (Contd.)**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
E88H		MSR_C8_PMON_CTRL0	Package	Uncore C-box 8 perfmon counter 0.
E89H		MSR_C8_PMON_CTRL1	Package	Uncore C-box 8 perfmon counter 1.
E8AH		MSR_C8_PMON_CTRL2	Package	Uncore C-box 8 perfmon counter 2.
E8BH		MSR_C8_PMON_CTRL3	Package	Uncore C-box 8 perfmon counter 3.
E90H		MSR_C9_PMON_BOX_CTL	Package	Uncore C-box 9 perfmon local box wide control.
E91H		MSR_C9_PMON_EVTSEL0	Package	Uncore C-box 9 perfmon event select for C-box 9 counter 0.
E92H		MSR_C9_PMON_EVTSEL1	Package	Uncore C-box 9 perfmon event select for C-box 9 counter 1.
E93H		MSR_C9_PMON_EVTSEL2	Package	Uncore C-box 9 perfmon event select for C-box 9 counter 2.
E94H		MSR_C9_PMON_EVTSEL3	Package	Uncore C-box 9 perfmon event select for C-box 9 counter 3.
E95H		MSR_C9_PMON_BOX_FILTER0	Package	Uncore C-box 9 perfmon box wide filter0.
E96H		MSR_C9_PMON_BOX_FILTER1	Package	Uncore C-box 9 perfmon box wide filter1.
E97H		MSR_C9_PMON_BOX_STATUS	Package	Uncore C-box 9 perfmon box wide status.
E98H		MSR_C9_PMON_CTRL0	Package	Uncore C-box 9 perfmon counter 0.
E99H		MSR_C9_PMON_CTRL1	Package	Uncore C-box 9 perfmon counter 1.
E9AH		MSR_C9_PMON_CTRL2	Package	Uncore C-box 9 perfmon counter 2.
E9BH		MSR_C9_PMON_CTRL3	Package	Uncore C-box 9 perfmon counter 3.
EA0H		MSR_C10_PMON_BOX_CTL	Package	Uncore C-box 10 perfmon local box wide control.
EA1H		MSR_C10_PMON_EVTSEL0	Package	Uncore C-box 10 perfmon event select for C-box 10 counter 0.
EA2H		MSR_C10_PMON_EVTSEL1	Package	Uncore C-box 10 perfmon event select for C-box 10 counter 1.
EA3H		MSR_C10_PMON_EVTSEL2	Package	Uncore C-box 10 perfmon event select for C-box 10 counter 2.
EA4H		MSR_C10_PMON_EVTSEL3	Package	Uncore C-box 10 perfmon event select for C-box 10 counter 3.
EA5H		MSR_C10_PMON_BOX_FILTER0	Package	Uncore C-box 10 perfmon box wide filter0.
EA6H		MSR_C10_PMON_BOX_FILTER1	Package	Uncore C-box 10 perfmon box wide filter1.
EA7H		MSR_C10_PMON_BOX_STATUS	Package	Uncore C-box 10 perfmon box wide status.
EA8H		MSR_C10_PMON_CTRL0	Package	Uncore C-box 10 perfmon counter 0.
EA9H		MSR_C10_PMON_CTRL1	Package	Uncore C-box 10 perfmon counter 1.
EAAH		MSR_C10_PMON_CTRL2	Package	Uncore C-box 10 perfmon counter 2.
EABH		MSR_C10_PMON_CTRL3	Package	Uncore C-box 10 perfmon counter 3.
EBOH		MSR_C11_PMON_BOX_CTL	Package	Uncore C-box 11 perfmon local box wide control.
EB1H		MSR_C11_PMON_EVTSEL0	Package	Uncore C-box 11 perfmon event select for C-box 11 counter 0.
EB2H		MSR_C11_PMON_EVTSEL1	Package	Uncore C-box 11 perfmon event select for C-box 11 counter 1.
EB3H		MSR_C11_PMON_EVTSEL2	Package	Uncore C-box 11 perfmon event select for C-box 11 counter 2.
EB4H		MSR_C11_PMON_EVTSEL3	Package	Uncore C-box 11 perfmon event select for C-box 11 counter 3.
EB5H		MSR_C11_PMON_BOX_FILTER0	Package	Uncore C-box 11 perfmon box wide filter0.
EB6H		MSR_C11_PMON_BOX_FILTER1	Package	Uncore C-box 11 perfmon box wide filter1.
EB7H		MSR_C11_PMON_BOX_STATUS	Package	Uncore C-box 11 perfmon box wide status.
EB8H		MSR_C11_PMON_CTRL0	Package	Uncore C-box 11 perfmon counter 0.



Table 35-31. Uncore PMU MSRs in Intel® Xeon® Processor E5 v3 Family (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
EB9H		MSR_C11_PMON_CTR1	Package	Uncore C-box 11 perfmon counter 1.
EBAH		MSR_C11_PMON_CTR2	Package	Uncore C-box 11 perfmon counter 2.
EBBH		MSR_C11_PMON_CTR3	Package	Uncore C-box 11 perfmon counter 3.
EC0H		MSR_C12_PMON_BOX_CTL	Package	Uncore C-box 12 perfmon local box wide control.
EC1H		MSR_C12_PMON_EVNTSELO	Package	Uncore C-box 12 perfmon event select for C-box 12 counter 0.
EC2H		MSR_C12_PMON_EVNTSEL1	Package	Uncore C-box 12 perfmon event select for C-box 12 counter 1.
EC3H		MSR_C12_PMON_EVNTSEL2	Package	Uncore C-box 12 perfmon event select for C-box 12 counter 2.
EC4H		MSR_C12_PMON_EVNTSEL3	Package	Uncore C-box 12 perfmon event select for C-box 12 counter 3.
EC5H		MSR_C12_PMON_BOX_FILTER0	Package	Uncore C-box 12 perfmon box wide filter0.
EC6H		MSR_C12_PMON_BOX_FILTER1	Package	Uncore C-box 12 perfmon box wide filter1.
EC7H		MSR_C12_PMON_BOX_STATUS	Package	Uncore C-box 12 perfmon box wide status.
EC8H		MSR_C12_PMON_CTR0	Package	Uncore C-box 12 perfmon counter 0.
EC9H		MSR_C12_PMON_CTR1	Package	Uncore C-box 12 perfmon counter 1.
ECAH		MSR_C12_PMON_CTR2	Package	Uncore C-box 12 perfmon counter 2.
ECBH		MSR_C12_PMON_CTR3	Package	Uncore C-box 12 perfmon counter 3.
ED0H		MSR_C13_PMON_BOX_CTL	Package	Uncore C-box 13 perfmon local box wide control.
ED1H		MSR_C13_PMON_EVNTSELO	Package	Uncore C-box 13 perfmon event select for C-box 13 counter 0.
ED2H		MSR_C13_PMON_EVNTSEL1	Package	Uncore C-box 13 perfmon event select for C-box 13 counter 1.
ED3H		MSR_C13_PMON_EVNTSEL2	Package	Uncore C-box 13 perfmon event select for C-box 13 counter 2.
ED4H		MSR_C13_PMON_EVNTSEL3	Package	Uncore C-box 13 perfmon event select for C-box 13 counter 3.
ED5H		MSR_C13_PMON_BOX_FILTER0	Package	Uncore C-box 13 perfmon box wide filter0.
ED6H		MSR_C13_PMON_BOX_FILTER1	Package	Uncore C-box 13 perfmon box wide filter1.
ED7H		MSR_C13_PMON_BOX_STATUS	Package	Uncore C-box 13 perfmon box wide status.
ED8H		MSR_C13_PMON_CTR0	Package	Uncore C-box 13 perfmon counter 0.
ED9H		MSR_C13_PMON_CTR1	Package	Uncore C-box 13 perfmon counter 1.
EDAH		MSR_C13_PMON_CTR2	Package	Uncore C-box 13 perfmon counter 2.
EDBH		MSR_C13_PMON_CTR3	Package	Uncore C-box 13 perfmon counter 3.
EE0H		MSR_C14_PMON_BOX_CTL	Package	Uncore C-box 14 perfmon local box wide control.
EE1H		MSR_C14_PMON_EVNTSELO	Package	Uncore C-box 14 perfmon event select for C-box 14 counter 0.
EE2H		MSR_C14_PMON_EVNTSEL1	Package	Uncore C-box 14 perfmon event select for C-box 14 counter 1.
EE3H		MSR_C14_PMON_EVNTSEL2	Package	Uncore C-box 14 perfmon event select for C-box 14 counter 2.
EE4H		MSR_C14_PMON_EVNTSEL3	Package	Uncore C-box 14 perfmon event select for C-box 14 counter 3.
EE5H		MSR_C14_PMON_BOX_FILTER	Package	Uncore C-box 14 perfmon box wide filter0.
EE6H		MSR_C14_PMON_BOX_FILTER1	Package	Uncore C-box 14 perfmon box wide filter1.
EE7H		MSR_C14_PMON_BOX_STATUS	Package	Uncore C-box 14 perfmon box wide status.
EE8H		MSR_C14_PMON_CTR0	Package	Uncore C-box 14 perfmon counter 0.
EE9H		MSR_C14_PMON_CTR1	Package	Uncore C-box 14 perfmon counter 1.



Table 35-31. Uncore PMU MSRs in Intel® Xeon® Processor E5 v3 Family (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
E6AH		MSR_C14_PMON_CTR2	Package	Uncore C-box 14 perfmon counter 2.
E6BH		MSR_C14_PMON_CTR3	Package	Uncore C-box 14 perfmon counter 3.
E6FH		MSR_C15_PMON_BOX_CTL	Package	Uncore C-box 15 perfmon local box wide control.
E6FH		MSR_C15_PMON_EVNTSELO	Package	Uncore C-box 15 perfmon event select for C-box 15 counter 0.
E6FH		MSR_C15_PMON_EVNTSEL1	Package	Uncore C-box 15 perfmon event select for C-box 15 counter 1.
E6FH		MSR_C15_PMON_EVNTSEL2	Package	Uncore C-box 15 perfmon event select for C-box 15 counter 2.
E6FH		MSR_C15_PMON_EVNTSEL3	Package	Uncore C-box 15 perfmon event select for C-box 15 counter 3.
E6FH		MSR_C15_PMON_BOX_FILTER0	Package	Uncore C-box 15 perfmon box wide filter0.
E6FH		MSR_C15_PMON_BOX_FILTER1	Package	Uncore C-box 15 perfmon box wide filter1.
E6FH		MSR_C15_PMON_BOX_STATUS	Package	Uncore C-box 15 perfmon box wide status.
E6FH		MSR_C15_PMON_CTR0	Package	Uncore C-box 15 perfmon counter 0.
E6FH		MSR_C15_PMON_CTR1	Package	Uncore C-box 15 perfmon counter 1.
E6FH		MSR_C15_PMON_CTR2	Package	Uncore C-box 15 perfmon counter 2.
E6FH		MSR_C15_PMON_CTR3	Package	Uncore C-box 15 perfmon counter 3.
F00H		MSR_C16_PMON_BOX_CTL	Package	Uncore C-box 16 perfmon for box-wide control
F01H		MSR_C16_PMON_EVNTSELO	Package	Uncore C-box 16 perfmon event select for C-box 16 counter 0.
F02H		MSR_C16_PMON_EVNTSEL1	Package	Uncore C-box 16 perfmon event select for C-box 16 counter 1.
F03H		MSR_C16_PMON_EVNTSEL2	Package	Uncore C-box 16 perfmon event select for C-box 16 counter 2.
F04H		MSR_C16_PMON_EVNTSEL3	Package	Uncore C-box 16 perfmon event select for C-box 16 counter 3.
F05H		MSR_C16_PMON_BOX_FILTER0	Package	Uncore C-box 16 perfmon box wide filter 0.
F06H		MSR_C16_PMON_BOX_FILTER1	Package	Uncore C-box 16 perfmon box wide filter 1.
F07H		MSR_C16_PMON_BOX_STATUS	Package	Uncore C-box 16 perfmon box wide status.
F08H		MSR_C16_PMON_CTR0	Package	Uncore C-box 16 perfmon counter 0.
F09H		MSR_C16_PMON_CTR1	Package	Uncore C-box 16 perfmon counter 1.
F0AH		MSR_C16_PMON_CTR2	Package	Uncore C-box 16 perfmon counter 2.
F0BH		MSR_C16_PMON_CTR3	Package	Uncore C-box 16 perfmon counter 3.
F10H		MSR_C17_PMON_BOX_CTL	Package	Uncore C-box 17 perfmon for box-wide control
F11H		MSR_C17_PMON_EVNTSELO	Package	Uncore C-box 17 perfmon event select for C-box 17 counter 0.
F12H		MSR_C17_PMON_EVNTSEL1	Package	Uncore C-box 17 perfmon event select for C-box 17 counter 1.
F13H		MSR_C17_PMON_EVNTSEL2	Package	Uncore C-box 17 perfmon event select for C-box 17 counter 2.
F14H		MSR_C17_PMON_EVNTSEL3	Package	Uncore C-box 17 perfmon event select for C-box 17 counter 3.
F15H		MSR_C17_PMON_BOX_FILTER0	Package	Uncore C-box 17 perfmon box wide filter 0.
F16H		MSR_C17_PMON_BOX_FILTER1	Package	Uncore C-box 17 perfmon box wide filter1.
F17H		MSR_C17_PMON_BOX_STATUS	Package	Uncore C-box 17 perfmon box wide status.
F18H		MSR_C17_PMON_CTR0	Package	Uncore C-box 17 perfmon counter 0.
F19H		MSR_C17_PMON_CTR1	Package	Uncore C-box 17 perfmon counter 1.
F1AH		MSR_C17_PMON_CTR2	Package	Uncore C-box 17 perfmon counter 2.

**Table 35-31. Uncore PMU MSRs in Intel® Xeon® Processor E5 v3 Family (Contd.)**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
F1BH		MSR_C17_PMON_CTR3	Package	Uncore C-box 17 perfmon counter 3.

### 35.13 MSRS IN INTEL® CORE™ M PROCESSORS AND 5TH GENERATION INTEL CORE PROCESSORS

The Intel® Core™ M-5xxx processors and 5th generation Intel® Core™ Processors, and Intel® Xeon® Processor E3-1200 v4 family are based on the Broadwell microarchitecture. The Intel® Core™ M-5xxx processors and 5th generation Intel® Core™ Processors have CPUID DisplayFamily\_DisplayModel signature 06\_3DH. Intel® Xeon® Processor E3-1200 v4 family and the 5th generation Intel® Core™ Processors have CPUID DisplayFamily\_DisplayModel signature 06\_47H. Processors with signatures 06\_3DH and 06\_47H support the MSR interfaces listed in Table 35-18, Table 35-19, Table 35-20, Table 35-23, Table 35-27, Table 35-28, Table 35-32, and Table 35-33. For an MSR listed in Table 35-33 that also appears in the model-specific tables of prior generations, Table 35-33 supercede prior generation tables.

Table 35-32 lists MSRs that are common to processors based on the Broadwell microarchitectures (including CPUID signatures 06\_3DH, 06\_47H, 06\_4FH, and 06\_56H).

**Table 35-32. Additional MSRs Common to Processors Based the Broadwell Microarchitectures**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
38EH	910	IA32_PERF_GLOBAL_STATUS	Thread	See Table 35-2. See Section 18.4.2, "Global Counter Control Facilities."
		0		<b>Ovf_PMC0</b>
		1		<b>Ovf_PMC1</b>
		2		<b>Ovf_PMC2</b>
		3		<b>Ovf_PMC3</b>
		31:4		Reserved.
		32		<b>Ovf_FixedCtr0</b>
		33		<b>Ovf_FixedCtr1</b>
		34		<b>Ovf_FixedCtr2</b>
		54:35		Reserved.
		55		<b>Trace_ToPA_PMI</b> . See Section 36.2.6.2, "Table of Physical Addresses (ToPA)."
		60:56		Reserved.
		61		<b>Ovf_Uncore</b>
		62		<b>Ovf_BufDSSAVE</b>
63		<b>CondChgd</b>		
390H	912	IA32_PERF_GLOBAL_OVF_CTRL	Thread	See Table 35-2. See Section 18.4.2, "Global Counter Control Facilities."
		0		<b>Set 1 to clear Ovf_PMC0</b>
		1		<b>Set 1 to clear Ovf_PMC1</b>

**Table 35-32. Additional MSRs Common to Processors Based the Broadwell Microarchitectures**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		2		Set 1 to clear Ovf_PMC2
		3		Set 1 to clear Ovf_PMC3
		31:4		Reserved.
		32		Set 1 to clear Ovf_FixedCtr0
		33		Set 1 to clear Ovf_FixedCtr1
		34		Set 1 to clear Ovf_FixedCtr2
		54:35		Reserved.
		55		Set 1 to clear Trace_ToPA_PMI. See Section 36.2.6.2, "Table of Physical Addresses (ToPA)."
		60:56		Reserved.
		61		Set 1 to clear Ovf_Uncore
		62		Set 1 to clear Ovf_BufDSSAVE
		63		Set 1 to clear CondChgd
560H	1376	IA32_RTIT_OUTPUT_BASE	THREAD	Trace Output Base Register (R/W)
		6:0		Reserved.
		MAXPHYADDR <sup>1</sup> -1:7		Base physical address.
		63:MAXPHYADDR		Reserved.
561H	1377	IA32_RTIT_OUTPUT_MASK_PTRS	THREAD	Trace Output Mask Pointers Register (R/W)
		6:0		Reserved.
		31:7		MaskOrTableOffset
		63:32		Output Offset.
570H	1392	IA32_RTIT_CTL	Thread	Trace Control Register (R/W)
		0		TraceEn
		1		Reserved, MBZ.
		2		OS
		3		User
		6:4		Reserved, MBZ
		7		CR3 filter
		8		ToPA; writing 0 will #GP if also setting TraceEn
		9		Reserved, MBZ
		10		TSCEn
		11		DisRETC
		12		Reserved, MBZ
		13		Reserved; writing 0 will #GP if also setting TraceEn
		63:14		Reserved, MBZ.
571H	1393	IA32_RTIT_STATUS	Thread	Tracing Status Register (R/W)

Table 35-32. Additional MSRs Common to Processors Based the Broadwell Microarchitectures

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		0		Reserved, writes ignored.
		1		<b>ContexEn</b> , writes ignored.
		2		<b>TriggerEn</b> , writes ignored.
		3		Reserved
		4		<b>Error (R/W)</b>
		5		<b>Stopped</b>
		63:6		Reserved, MBZ.
572H	1394	IA32_RTIT_CR3_MATCH	THREAD	<b>Trace Filter CR3 Match Register (R/W)</b>
		4:0		Reserved
		63:5		CR3[63:5] value to match

**NOTES:**

1. MAXPHYADDR is reported by CPUID.80000008H:EAX[7:0].

Table 35-33 lists MSRs that are specific to Intel Core M processors and 5th Generation Intel Core Processors.

Table 35-33. Additional MSRs Supported by Intel® Core™ M Processors and 5th Generation Intel® Core™ Processors

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
E2H	226	MSR_PKG_CST_CONFIG_CONTROL	Core	<b>C-State Configuration Control (R/W)</b> Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-states. See <a href="http://biosbits.org">http://biosbits.org</a> .
		3:0		<b>Package C-State Limit (R/W)</b> Specifies the lowest processor-specific C-state code name (consuming the least power) for the package. The default is set as factory-configured package C-state limit. The following C-state code name encodings are supported: 0000b: C0/C1 (no package C-state support) 0001b: C2 0010b: C3 0011b: C6 0100b: C7 0101b: C7s 0110b: C8 0111b: C9 1000b: C10
		9:4		Reserved
		10		<b>I/O MWAIT Redirection Enable (R/W)</b>

**Table 35-33. Additional MSRs Supported by Intel® Core™ M Processors and 5th Generation Intel® Core™ Processors**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		14:11		Reserved
		15		<b>CFG Lock (R/WO)</b>
		24:16		Reserved
		25		<b>C3 State Auto Demotion Enable (R/W)</b>
		26		<b>C1 State Auto Demotion Enable (R/W)</b>
		27		<b>Enable C3 Undemotion (R/W)</b>
		28		<b>Enable C1 Undemotion (R/W)</b>
		29		<b>Enable Package C-State Auto-demotion (R/W)</b>
		30		<b>Enable Package C-State Undemotion (R/W)</b>
		63:31		Reserved
1ADH	429	MSR_TURBO_RATIO_LIMIT	Package	<b>Maximum Ratio Limit of Turbo Mode</b> RO if MSR_PLATFORM_INFO.[28] = 0, RW if MSR_PLATFORM_INFO.[28] = 1
		7:0	Package	<b>Maximum Ratio Limit for 1C</b> Maximum turbo ratio limit of 1 core active.
		15:8	Package	<b>Maximum Ratio Limit for 2C</b> Maximum turbo ratio limit of 2 core active.
		23:16	Package	<b>Maximum Ratio Limit for 3C</b> Maximum turbo ratio limit of 3 core active.
		31:24	Package	<b>Maximum Ratio Limit for 4C</b> Maximum turbo ratio limit of 4 core active.
		39:32	Package	<b>Maximum Ratio Limit for 5C</b> Maximum turbo ratio limit of 5core active.
		47:40	Package	<b>Maximum Ratio Limit for 6C</b> Maximum turbo ratio limit of 6core active.
		63:48		Reserved.
639H	1593	MSR_PPO_ENERGY_STATUS	Package	<b>PPO Energy Status (R/O)</b> See Section 14.9.4, "PPO/PP1 RAPL Domains."

See Table 35-18, Table 35-19, Table 35-20, Table 35-23, Table 35-27, Table 35-28, Table 35-32 for other MSR definitions applicable to processors with CPUID signature 06\_3DH.

### 35.14 MSRS IN INTEL® XEON® PROCESSORS E5 V4 FAMILY

The MSRs listed in Table 35-34 are available and common to Intel® Xeon® Processor D product Family (CPUID DisplayFamily\_DisplayModel = 06\_56H) and to Intel Xeon processors E5 v4, E7 v4 families (CPUID DisplayFamily\_DisplayModel = 06\_4FH). They are based on the Broadwell microarchitecture.

See Section 35.14.1 for lists of tables of MSRs that are supported by Intel® Xeon® Processor D Family.

**Table 35-34. Additional MSRs Common to Intel® Xeon® Processor D and Intel Xeon Processors E5 v4 Family Based on the Broadwell Microarchitecture**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
4EH	78	MSR_PPIN_CTL	Package	Protected Processor Inventory Number Enable Control (R/W)
		0		LockOut (R/W/O) See Table 35-24.
		1		Enable_PPIN (R/W) See Table 35-24.
		63:2		Reserved.
4FH	79	MSR_PPIN	Package	Protected Processor Inventory Number (R/O)
		63:0		<b>Protected Processor Inventory Number (R/O)</b> See Table 35-24.
CEH	206	MSR_PLATFORM_INFO	Package	See <a href="http://biosbits.org">http://biosbits.org</a> .
		7:0		Reserved.
		15:8	Package	<b>Maximum Non-Turbo Ratio (R/O)</b> See Table 35-24.
		22:16		Reserved.
		23	Package	<b>PPIN_CAP (R/O)</b> See Table 35-24.
		27:24		Reserved.
		28	Package	<b>Programmable Ratio Limit for Turbo Mode (R/O)</b> See Table 35-24.
		29	Package	<b>Programmable TDP Limit for Turbo Mode (R/O)</b> See Table 35-24.
		30	Package	<b>Programmable TJ OFFSET (R/O)</b> See Table 35-24.
		39:31		Reserved.
		47:40	Package	<b>Maximum Efficiency Ratio (R/O)</b> See Table 35-24.
		63:48		Reserved.
E2H	226	MSR_PKG_CST_CONFIG_CONTROL	Core	<b>C-State Configuration Control (R/W)</b> Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-states. See <a href="http://biosbits.org">http://biosbits.org</a> .

**Table 35-34. Additional MSRs Common to Intel® Xeon® Processor D and Intel Xeon Processors E5 v4 Family Based on the Broadwell Microarchitecture**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		2:0		<b>Package C-State Limit (R/W)</b> Specifies the lowest processor-specific C-state code name (consuming the least power) for the package. The default is set as factory-configured package C-state limit. The following C-state code name encodings are supported: 000b: C0/C1 (no package C-state support) 001b: C2 010b: C6 (non-retention) 011b: C6 (retention) 111b: No Package C state limits. All C states supported by the processor are available.
		9:3		Reserved
		10		<b>I/O MWAIT Redirection Enable (R/W)</b>
		14:11		Reserved
		15		<b>CFG Lock (R/WO)</b>
		16		<b>Automatic C-State Conversion Enable (R/W)</b> If 1, the processor will convert HALT or MWAIT(C1) to MWAIT(C6)
		24:17		Reserved
		25		<b>C3 State Auto Demotion Enable (R/W)</b>
		26		<b>C1 State Auto Demotion Enable (R/W)</b>
		27		<b>Enable C3 Undemotion (R/W)</b>
		28		<b>Enable C1 Undemotion (R/W)</b>
		29		<b>Package C State Demotion Enable (R/W)</b>
		30		<b>Package C State UnDemotion Enable (R/W)</b>
		63:31		Reserved
179H	377	IA32_MCG_CAP	Thread	<b>Global Machine Check Capability (R/O)</b>
		7:0		<b>Count</b>
		8		<b>MCG_CTL_P</b>
		9		<b>MCG_EXT_P</b>
		10		<b>MCP_CMCI_P</b>
		11		<b>MCG_TES_P</b>
		15:12		Reserved.
		23:16		<b>MCG_EXT_CNT</b>
		24		<b>MCG_SER_P</b>
		25		<b>MCG_EM_P</b>
		26		<b>MCG_ELOG_P</b>
63:27		Reserved.		

**Table 35-34. Additional MSRs Common to Intel® Xeon® Processor D and Intel Xeon Processors E5 v4 Family Based on the Broadwell Microarchitecture**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
17DH	390	MSR_SMM_MCA_CAP	THREAD	<b>Enhanced SMM Capabilities (SMM-RO)</b> Reports SMM capability Enhancement. Accessible only while in SMM.
		57:0		<b>Reserved</b>
		58		<b>SMM_Code_Access_Chk (SMM-RO)</b> If set to 1 indicates that the SMM code access restriction is supported and a host-space interface available to SMM handler.
		59		<b>Long_Flow_Indication (SMM-RO)</b> If set to 1 indicates that the SMM long flow indicator is supported and a host-space interface available to SMM handler.
		63:60		Reserved
19CH	412	IA32_THERM_STATUS	Core	<b>Thermal Monitor Status (R/W)</b> See Table 35-2.
		0		<b>Thermal status (RO)</b> See Table 35-2.
		1		<b>Thermal status log (R/WCO)</b> See Table 35-2.
		2		<b>PROTCHOT # or FORCEPR# status (RO)</b> See Table 35-2.
		3		<b>PROTCHOT # or FORCEPR# log (R/WCO)</b> See Table 35-2.
		4		<b>Critical Temperature status (RO)</b> See Table 35-2.
		5		<b>Critical Temperature status log (R/WCO)</b> See Table 35-2.
		6		<b>Thermal threshold #1 status (RO)</b> See Table 35-2.
		7		<b>Thermal threshold #1 log (R/WCO)</b> See Table 35-2.
		8		<b>Thermal threshold #2 status (RO)</b> See Table 35-2.
		9		<b>Thermal threshold #2 log (R/WCO)</b> See Table 35-2.
		10		<b>Power Limitation status (RO)</b> See Table 35-2.
		11		<b>Power Limitation log (R/WCO)</b> See Table 35-2.
12		<b>Current Limit status (RO)</b> See Table 35-2.		



**Table 35-34. Additional MSRs Common to Intel® Xeon® Processor D and Intel Xeon Processors E5 v4 Family Based on the Broadwell Microarchitecture**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		13		<b>Current Limit log (R/WCO)</b> See Table 35-2.
		14		<b>Cross Domain Limit status (RO)</b> See Table 35-2.
		15		<b>Cross Domain Limit log (R/WCO)</b> See Table 35-2.
		22:16		<b>Digital Readout (RO)</b> See Table 35-2.
		26:23		Reserved.
		30:27		<b>Resolution in degrees Celsius (RO)</b> See Table 35-2.
		31		<b>Reading Valid (RO)</b> See Table 35-2.
		63:32		Reserved.
1A2H	418	MSR_TEMPERATURE_TARGET	Package	
		15:0		Reserved.
		23:16		<b>Temperature Target (RO)</b> See Table 35-24.
		27:24		<b>TCC Activation Offset (R/W)</b> See Table 35-24.
		63:28		Reserved.
1ADH	429	MSR_TURBO_RATIO_LIMIT	Package	<b>Maximum Ratio Limit of Turbo Mode</b> RO if MSR_PLATFORM_INFO.[28] = 0, RW if MSR_PLATFORM_INFO.[28] = 1
		7:0	Package	<b>Maximum Ratio Limit for 1C</b>
		15:8	Package	<b>Maximum Ratio Limit for 2C</b>
		23:16	Package	<b>Maximum Ratio Limit for 3C</b>
		31:24	Package	<b>Maximum Ratio Limit for 4C</b>
		39:32	Package	<b>Maximum Ratio Limit for 5C</b>
		47:40	Package	<b>Maximum Ratio Limit for 6C</b>
		55:48	Package	<b>Maximum Ratio Limit for 7C</b>
		63:56	Package	<b>Maximum Ratio Limit for 8C</b>
1AEH	430	MSR_TURBO_RATIO_LIMIT1	Package	<b>Maximum Ratio Limit of Turbo Mode</b> RO if MSR_PLATFORM_INFO.[28] = 0, RW if MSR_PLATFORM_INFO.[28] = 1
		7:0	Package	<b>Maximum Ratio Limit for 9C</b>
		15:8	Package	<b>Maximum Ratio Limit for 10C</b>

**Table 35-34. Additional MSRs Common to Intel® Xeon® Processor D and Intel Xeon Processors E5 v4 Family Based on the Broadwell Microarchitecture**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		23:16	Package	<b>Maximum Ratio Limit for 11C</b>
		31:24	Package	<b>Maximum Ratio Limit for 12C</b>
		39:32	Package	<b>Maximum Ratio Limit for 13C</b>
		47:40	Package	<b>Maximum Ratio Limit for 14C</b>
		55:48	Package	<b>Maximum Ratio Limit for 15C</b>
		63:56	Package	<b>Maximum Ratio Limit for 16C</b>
606H	1542	MSR_RAPL_POWER_UNIT	Package	<b>Unit Multipliers used in RAPL Interfaces (R/O)</b>
		3:0	Package	<b>Power Units</b> See Section 14.9.1, "RAPL Interfaces."
		7:4	Package	<b>Reserved</b>
		12:8	Package	<b>Energy Status Units</b> Energy related information (in Joules) is based on the multiplier, $1/2^{\text{ESU}}$ ; where ESU is an unsigned integer represented by bits 12:8. Default value is 0EH (or 61 micro-joules)
		15:13	Package	<b>Reserved</b>
		19:16	Package	<b>Time Units</b> See Section 14.9.1, "RAPL Interfaces."
		63:20		Reserved
618H	1560	MSR_DRAM_POWER_LIMIT	Package	<b>DRAM RAPL Power Limit Control (R/W)</b> See Section 14.9.5, "DRAM RAPL Domain."
619H	1561	MSR_DRAM_ENERGY_STATUS	Package	<b>DRAM Energy Status (R/O)</b> Energy consumed by DRAM devices
		31:0		Energy in 15.3 micro-joules. Requires BIOS configuration to enable DRAM RAPL mode 0 (Direct VR).
		63:32		Reserved
61BH	1563	MSR_DRAM_PERF_STATUS	Package	<b>DRAM Performance Throttling Status (R/O)</b> See Section 14.9.5, "DRAM RAPL Domain."
61CH	1564	MSR_DRAM_POWER_INFO	Package	<b>DRAM RAPL Parameters (R/W)</b> See Section 14.9.5, "DRAM RAPL Domain."
639H	1593	MSR_PPO_ENERGY_STATUS	Package	<b>Reserved (R/O)</b> Reads return 0
690H	1680	MSR_CORE_PERF_LIMIT_REASONS	Package	<b>Indicator of Frequency Clipping in Processor Cores (R/W) (frequency refers to processor core frequency)</b>
		0		<b>PROCHOT Status (R0)</b> When set, processor core frequency is reduced below the operating system request due to assertion of external PROCHOT.
		1		<b>Thermal Status (R0)</b> When set, frequency is reduced below the operating system request due to a thermal event.

**Table 35-34. Additional MSRs Common to Intel® Xeon® Processor D and Intel Xeon Processors E5 v4 Family Based on the Broadwell Microarchitecture**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		2		<b>Power Budget Management Status (R0)</b> When set, frequency is reduced below the operating system request due to PBM limit
		3		<b>Platform Configuration Services Status (R0)</b> When set, frequency is reduced below the operating system request due to PCS limit
		4		Reserved.
		5		<b>Autonomous Utilization-Based Frequency Control Status (R0)</b> When set, frequency is reduced below the operating system request because the processor has detected that utilization is low
		6		<b>VR Therm Alert Status (R0)</b> When set, frequency is reduced below the operating system request due to a thermal alert from the Voltage Regulator.
		7		Reserved.
		8		<b>Electrical Design Point Status (R0)</b> When set, frequency is reduced below the operating system request due to electrical design point constraints (e.g. maximum electrical current consumption).
		9		Reserved.
		10		<b>Multi-Core Turbo Status (R0)</b> When set, frequency is reduced below the operating system request due to Multi-Core Turbo limits
		12:11		Reserved.
		13		<b>Core Frequency P1 Status (R0)</b> When set, frequency is reduced below max non-turbo P1
		14		<b>Core Max n-core Turbo Frequency Limiting Status (R0)</b> When set, frequency is reduced below max n-core turbo frequency
		15		<b>Core Frequency Limiting Status (R0)</b> When set, frequency is reduced below the operating system request.
		16		<b>PROCHOT Log</b> When set, indicates that the PROCHOT Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		17		<b>Thermal Log</b> When set, indicates that the Thermal Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.

**Table 35-34. Additional MSRs Common to Intel® Xeon® Processor D and Intel Xeon Processors E5 v4 Family Based on the Broadwell Microarchitecture**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		18		<b>Power Budget Management Log</b> When set, indicates that the PBM Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		19		<b>Platform Configuration Services Log</b> When set, indicates that the PCS Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		20		Reserved.
		21		<b>Autonomous Utilization-Based Frequency Control Log</b> When set, indicates that the AUBFC Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		22		<b>VR Therm Alert Log</b> When set, indicates that the VR Therm Alert Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		23		Reserved.
		24		<b>Electrical Design Point Log</b> When set, indicates that the EDP Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		25		Reserved.
		26		<b>Multi-Core Turbo Log</b> When set, indicates that the Multi-Core Turbo Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		28:27		Reserved.
		29		<b>Core Frequency P1 Log</b> When set, indicates that the Core Frequency P1 Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		30		<b>Core Max n-core Turbo Frequency Limiting Log</b> When set, indicates that the Core Max n-core Turbo Frequency Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		31		<b>Core Frequency Limiting Log</b> When set, indicates that the Core Frequency Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		63:32		Reserved.

**Table 35-34. Additional MSRs Common to Intel® Xeon® Processor D and Intel Xeon Processors E5 v4 Family Based on the Broadwell Microarchitecture**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
770H	1904	IA32_PM_ENABLE	Package	See Section 14.4.2, “Enabling HWP”
771H	1905	IA32_HWP_CAPABILITIES	Thread	See Section 14.4.3, “HWP Performance Range and Dynamic Capabilities”
774H	1908	IA32_HWP_REQUEST	Thread	See Section 14.4.4, “Managing HWP”
		7:0		<b>Minimum Performance (R/W)</b>
		15:8		<b>Maximum Performance (R/W)</b>
		23:16		<b>Desired Performance (R/W)</b>
		63:24		Reserved.
777H	1911	IA32_HWP_STATUS	Thread	See Section 14.4.5, “HWP Feedback”
		1:0		Reserved.
		2		<b>Excursion to Minimum (RO)</b>
		63:3		Reserved.
C8DH	3213	IA32_QM_EVTSEL	THREAD	<b>Monitoring Event Select Register (R/W)</b> if CPUID.(EAX=07H, ECX=0):EBX.RDT-M[bit 12] = 1
		7:0		<b>EventID (RW)</b> <b>Event encoding:</b> 0x00: no monitoring 0x01: L3 occupancy monitoring 0x02: Total memory bandwidth monitoring 0x03: Local memory bandwidth monitoring All other encoding reserved
		31:8		Reserved.
		41:32		<b>RMID (RW)</b>
		63:42		Reserved.
C8FH	3215	IA32_PQR_ASSOC	THREAD	<b>Resource Association Register (R/W)</b>
		9:0		<b>RMID</b>
		31:10		<b>Reserved</b>
		51:32		<b>COS (R/W).</b>
		63: 52		<b>Reserved</b>
C90H	3216	IA32_L3_QOS_MASK_0	Package	<b>L3 Class Of Service Mask - COS 0 (R/W)</b> if CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=0
		0:19		<b>CBM: Bit vector of available L3 ways for COS 0 enforcement</b>
		63:20		<b>Reserved</b>
C91H	3217	IA32_L3_QOS_MASK_1	Package	<b>L3 Class Of Service Mask - COS 1 (R/W)</b> if CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=1
		0:19		<b>CBM: Bit vector of available L3 ways for COS 1 enforcement</b>
		63:20		<b>Reserved</b>

**Table 35-34. Additional MSRs Common to Intel® Xeon® Processor D and Intel Xeon Processors E5 v4 Family Based on the Broadwell Microarchitecture**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
C92H	3218	IA32_L3_QOS_MASK_2	Package	<b>L3 Class Of Service Mask - COS 2 (R/W).</b> if CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=2
		0:19		<b>CBM: Bit vector of available L3 ways for COS 2 enforcement</b>
		63:20		<b>Reserved</b>
C93H	3219	IA32_L3_QOS_MASK_3	Package	<b>L3 Class Of Service Mask - COS 3 (R/W).</b> if CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=3
		0:19		<b>CBM: Bit vector of available L3 ways for COS 3 enforcement</b>
		63:20		<b>Reserved</b>
C94H	3220	IA32_L3_QOS_MASK_4	Package	<b>L3 Class Of Service Mask - COS 4 (R/W).</b> if CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=4
		0:19		<b>CBM: Bit vector of available L3 ways for COS 4 enforcement</b>
		63:20		<b>Reserved</b>
C95H	3221	IA32_L3_QOS_MASK_5	Package	<b>L3 Class Of Service Mask - COS 5 (R/W).</b> if CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=5
		0:19		<b>CBM: Bit vector of available L3 ways for COS 5 enforcement</b>
		63:20		<b>Reserved</b>
C96H	3222	IA32_L3_QOS_MASK_6	Package	<b>L3 Class Of Service Mask - COS 6 (R/W).</b> if CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=6
		0:19		<b>CBM: Bit vector of available L3 ways for COS 6 enforcement</b>
		63:20		<b>Reserved</b>
C97H	3223	IA32_L3_QOS_MASK_7	Package	<b>L3 Class Of Service Mask - COS 7 (R/W).</b> if CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=7
		0:19		<b>CBM: Bit vector of available L3 ways for COS 7 enforcement</b>
		63:20		<b>Reserved</b>
C98H	3224	IA32_L3_QOS_MASK_8	Package	<b>L3 Class Of Service Mask - COS 8 (R/W).</b> if CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=8
		0:19		<b>CBM: Bit vector of available L3 ways for COS 8 enforcement</b>
		63:20		<b>Reserved</b>
C99H	3225	IA32_L3_QOS_MASK_9	Package	<b>L3 Class Of Service Mask - COS 9 (R/W).</b> if CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=9
		0:19		<b>CBM: Bit vector of available L3 ways for COS 9 enforcement</b>
		63:20		<b>Reserved</b>
C9AH	3226	IA32_L3_QOS_MASK_10	Package	<b>L3 Class Of Service Mask - COS 10 (R/W).</b> if CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=10
		0:19		<b>CBM: Bit vector of available L3 ways for COS 10 enforcement</b>
		63:20		<b>Reserved</b>

**Table 35-34. Additional MSRs Common to Intel® Xeon® Processor D and Intel Xeon Processors E5 v4 Family Based on the Broadwell Microarchitecture**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
C9BH	3227	IA32_L3_QOS_MASK_11	Package	<b>L3 Class Of Service Mask - COS 11 (R/W).</b> if CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=11
		0:19		<b>CBM: Bit vector of available L3 ways for COS 11 enforcement</b>
		63:20		<b>Reserved</b>
C9CH	3228	IA32_L3_QOS_MASK_12	Package	<b>L3 Class Of Service Mask - COS 12 (R/W).</b> if CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=12
		0:19		<b>CBM: Bit vector of available L3 ways for COS 12 enforcement</b>
		63:20		<b>Reserved</b>
C9DH	3229	IA32_L3_QOS_MASK_13	Package	<b>L3 Class Of Service Mask - COS 13 (R/W).</b> if CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=13
		0:19		<b>CBM: Bit vector of available L3 ways for COS 13 enforcement</b>
		63:20		<b>Reserved</b>
C9EH	3230	IA32_L3_QOS_MASK_14	Package	<b>L3 Class Of Service Mask - COS 14 (R/W).</b> if CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=14
		0:19		<b>CBM: Bit vector of available L3 ways for COS 14 enforcement</b>
		63:20		<b>Reserved</b>
C9FH	3231	IA32_L3_QOS_MASK_15	Package	<b>L3 Class Of Service Mask - COS 15 (R/W).</b> if CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=15
		0:19		<b>CBM: Bit vector of available L3 ways for COS 15 enforcement</b>
		63:20		<b>Reserved</b>

### 35.14.1 Additional MSRs Supported in the Intel® Xeon® Processor D Product Family

The MSRs listed in Table 35-35 are available to Intel® Xeon® Processor D Product Family (CPUID DisplayFamily\_DisplayModel = 06\_56H). The Intel® Xeon® processor D product family is based on the Broadwell microarchitecture and supports the MSR interfaces listed in Table 35-18, Table 35-27, Table 35-32, Table 35-34, and Table 35-35.

**Table 35-35. Additional MSRs Supported by Intel® Xeon® Processor D with DisplayFamily\_DisplayModel 06\_56H**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
1ACH	428	MSR_TURBO_RATIO_LIMIT3	Package	<b>Config Ratio Limit of Turbo Mode</b> RO if MSR_PLATFORM_INFO.[28] = 0, RW if MSR_PLATFORM_INFO.[28] = 1
		62:0	Package	<b>Reserved</b>

**Table 35-35. Additional MSRs Supported by Intel® Xeon® Processor D with DisplayFamily\_DisplayModel 06\_56H**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		63	Package	<b>Semaphore for Turbo Ratio Limit Configuration</b> If 1, the processor uses override configuration <sup>1</sup> specified in MSR_TURBO_RATIO_LIMIT, MSR_TURBO_RATIO_LIMIT1. If 0, the processor uses factory-set configuration (Default).
286H	646	IA32_MC6_CTL2	Package	See Table 35-2.
287H	647	IA32_MC7_CTL2	Package	See Table 35-2.
289H	649	IA32_MC9_CTL2	Package	See Table 35-2.
28AH	650	IA32_MC10_CTL2	Package	See Table 35-2.
291H	657	IA32_MC17_CTL2	Package	See Table 35-2.
292H	658	IA32_MC18_CTL2	Package	See Table 35-2.
293H	659	IA32_MC19_CTL2	Package	See Table 35-2.
418H	1048	IA32_MC6_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Bank MC6 reports MC error from the integrated I/O module.
419H	1049	IA32_MC6_STATUS	Package	
41AH	1050	IA32_MC6_ADDR	Package	
41BH	1051	IA32_MC6_MISC	Package	
41CH	1052	IA32_MC7_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Bank MC7 reports MC error from the home agent HA 0.
41DH	1053	IA32_MC7_STATUS	Package	
41EH	1054	IA32_MC7_ADDR	Package	
41FH	1055	IA32_MC7_MISC	Package	
424H	1060	IA32_MC9_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Banks MC9 through MC 10 report MC error from each channel of the integrated memory controllers.
425H	1061	IA32_MC9_STATUS	Package	
426H	1062	IA32_MC9_ADDR	Package	
427H	1063	IA32_MC9_MISC	Package	
428H	1064	IA32_MC10_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Banks MC9 through MC 10 report MC error from each channel of the integrated memory controllers.
429H	1065	IA32_MC10_STATUS	Package	
42AH	1066	IA32_MC10_ADDR	Package	
42BH	1067	IA32_MC10_MISC	Package	
444H	1092	IA32_MC17_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Bank MC17 reports MC error from the following pair of CBo/L3 Slices (if the pair is present): CBo0, CBo3, CBo6, CBo9, CBo12, CBo15.
445H	1093	IA32_MC17_STATUS	Package	
446H	1094	IA32_MC17_ADDR	Package	
447H	1095	IA32_MC17_MISC	Package	
448H	1096	IA32_MC18_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Bank MC18 reports MC error from the following pair of CBo/L3 Slices (if the pair is present): CBo1, CBo4, CBo7, CBo10, CBo13, CBo16.
449H	1097	IA32_MC18_STATUS	Package	
44AH	1098	IA32_MC18_ADDR	Package	
44BH	1099	IA32_MC18_MISC	Package	



**Table 35-35. Additional MSRs Supported by Intel® Xeon® Processor D with DisplayFamily\_DisplayModel 06\_56H**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
44CH	1100	IA32_MC19_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs:" through Section 15.3.2.4, "IA32_MCi_MISC MSRs."  Bank MC19 reports MC error from the following pair of CBo/L3 Slices (if the pair is present): CBo2, CBo5, CBo8, CBo11, CBo14, CBo17.
44DH	1101	IA32_MC19_STATUS	Package	
44EH	1102	IA32_MC19_ADDR	Package	
44FH	1103	IA32_MC19_MISC	Package	
See Table 35-18, Table 35-27, Table 35-32, and Table 35-34 for other MSR definitions applicable to processors with CPUID signature 06_56H.				

**NOTES:**

1. An override configuration lower than the factory-set configuration is always supported. An override configuration higher than the factory-set configuration is dependent on features specific to the processor and the platform.

**35.14.2 Additional MSRs Supported in Intel® Xeon® Processors E5 v4 and E7 v4 Families**

The MSRs listed in Table 35-35 are available to Intel® Xeon® Processor E5 v4 and E7 v4 Families (CPUID DisplayFamily\_DisplayModel = 06\_4FH). The Intel® Xeon® processor E5 v4 family is based on the Broadwell micro-architecture and supports the MSR interfaces listed in Table 35-18, Table 35-19, Table 35-27, Table 35-32, Table 35-34, and Table 35-36.

**Table 35-36. Additional MSRs Supported by Intel® Xeon® Processors with DisplayFamily\_DisplayModel 06\_4FH**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
1ACH	428	MSR_TURBO_RATIO_LIMIT3	Package	<b>Config Ratio Limit of Turbo Mode</b> RO if MSR_PLATFORM_INFO.[28] = 0, RW if MSR_PLATFORM_INFO.[28] = 1
		62:0	Package	Reserved
		63	Package	<b>Semaphore for Turbo Ratio Limit Configuration</b> If 1, the processor uses override configuration <sup>1</sup> specified in MSR_TURBO_RATIO_LIMIT, MSR_TURBO_RATIO_LIMIT1 and MSR_TURBO_RATIO_LIMIT2. If 0, the processor uses factory-set configuration (Default).
285H	645	IA32_MC5_CTL2	Package	See Table 35-2.
286H	646	IA32_MC6_CTL2	Package	See Table 35-2.
287H	647	IA32_MC7_CTL2	Package	See Table 35-2.
288H	648	IA32_MC8_CTL2	Package	See Table 35-2.
289H	649	IA32_MC9_CTL2	Package	See Table 35-2.
28AH	650	IA32_MC10_CTL2	Package	See Table 35-2.
28BH	651	IA32_MC11_CTL2	Package	See Table 35-2.
28CH	652	IA32_MC12_CTL2	Package	See Table 35-2.
28DH	653	IA32_MC13_CTL2	Package	See Table 35-2.
28EH	654	IA32_MC14_CTL2	Package	See Table 35-2.
28FH	655	IA32_MC15_CTL2	Package	See Table 35-2.

**Table 35-36. Additional MSRs Supported by Intel® Xeon® Processors with DisplayFamily\_DisplayModel 06\_4FH**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
290H	656	IA32_MC16_CTL2	Package	See Table 35-2.
291H	657	IA32_MC17_CTL2	Package	See Table 35-2.
292H	658	IA32_MC18_CTL2	Package	See Table 35-2.
293H	659	IA32_MC19_CTL2	Package	See Table 35-2.
294H	660	IA32_MC20_CTL2	Package	See Table 35-2.
295H	661	IA32_MC21_CTL2	Package	See Table 35-2.
414H	1044	IA32_MC5_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Bank MC5 reports MC error from the Intel QPI 0 module.
415H	1045	IA32_MC5_STATUS	Package	
416H	1046	IA32_MC5_ADDR	Package	
417H	1047	IA32_MC5_MISC	Package	
418H	1048	IA32_MC6_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Bank MC6 reports MC error from the integrated I/O module.
419H	1049	IA32_MC6_STATUS	Package	
41AH	1050	IA32_MC6_ADDR	Package	
41BH	1051	IA32_MC6_MISC	Package	
41CH	1052	IA32_MC7_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Bank MC7 reports MC error from the home agent HA 0.
41DH	1053	IA32_MC7_STATUS	Package	
41EH	1054	IA32_MC7_ADDR	Package	
41FH	1055	IA32_MC7_MISC	Package	
420H	1056	IA32_MC8_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Bank MC8 reports MC error from the home agent HA 1.
421H	1057	IA32_MC8_STATUS	Package	
422H	1058	IA32_MC8_ADDR	Package	
423H	1059	IA32_MC8_MISC	Package	
424H	1060	IA32_MC9_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Banks MC9 through MC 16 report MC error from each channel of the integrated memory controllers.
425H	1061	IA32_MC9_STATUS	Package	
426H	1062	IA32_MC9_ADDR	Package	
427H	1063	IA32_MC9_MISC	Package	
428H	1064	IA32_MC10_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Banks MC9 through MC 16 report MC error from each channel of the integrated memory controllers.
429H	1065	IA32_MC10_STATUS	Package	
42AH	1066	IA32_MC10_ADDR	Package	
42BH	1067	IA32_MC10_MISC	Package	
42CH	1068	IA32_MC11_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Banks MC9 through MC 16 report MC error from each channel of the integrated memory controllers.
42DH	1069	IA32_MC11_STATUS	Package	
42EH	1070	IA32_MC11_ADDR	Package	
42FH	1071	IA32_MC11_MISC	Package	

**Table 35-36. Additional MSRs Supported by Intel® Xeon® Processors with DisplayFamily\_DisplayModel 06\_4FH**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
430H	1072	IA32_MC12_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Banks MC9 through MC 16 report MC error from each channel of the integrated memory controllers.
431H	1073	IA32_MC12_STATUS	Package	
432H	1074	IA32_MC12_ADDR	Package	
433H	1075	IA32_MC12_MISC	Package	
434H	1076	IA32_MC13_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Banks MC9 through MC 16 report MC error from each channel of the integrated memory controllers.
435H	1077	IA32_MC13_STATUS	Package	
436H	1078	IA32_MC13_ADDR	Package	
437H	1079	IA32_MC13_MISC	Package	
438H	1080	IA32_MC14_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Banks MC9 through MC 16 report MC error from each channel of the integrated memory controllers.
439H	1081	IA32_MC14_STATUS	Package	
43AH	1082	IA32_MC14_ADDR	Package	
43BH	1083	IA32_MC14_MISC	Package	
43CH	1084	IA32_MC15_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Banks MC9 through MC 16 report MC error from each channel of the integrated memory controllers.
43DH	1085	IA32_MC15_STATUS	Package	
43EH	1086	IA32_MC15_ADDR	Package	
43FH	1087	IA32_MC15_MISC	Package	
440H	1088	IA32_MC16_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Banks MC9 through MC 16 report MC error from each channel of the integrated memory controllers.
441H	1089	IA32_MC16_STATUS	Package	
442H	1090	IA32_MC16_ADDR	Package	
443H	1091	IA32_MC16_MISC	Package	
444H	1092	IA32_MC17_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Bank MC17 reports MC error from the following pair of CBo/L3 Slices (if the pair is present): CBo0, CBo3, CBo6, CBo9, CBo12, CBo15.
445H	1093	IA32_MC17_STATUS	Package	
446H	1094	IA32_MC17_ADDR	Package	
447H	1095	IA32_MC17_MISC	Package	
448H	1096	IA32_MC18_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Bank MC18 reports MC error from the following pair of CBo/L3 Slices (if the pair is present): CBo1, CBo4, CBo7, CBo10, CBo13, CBo16.
449H	1097	IA32_MC18_STATUS	Package	
44AH	1098	IA32_MC18_ADDR	Package	
44BH	1099	IA32_MC18_MISC	Package	
44CH	1100	IA32_MC19_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Bank MC19 reports MC error from the following pair of CBo/L3 Slices (if the pair is present): CBo2, CBo5, CBo8, CBo11, CBo14, CBo17.
44DH	1101	IA32_MC19_STATUS	Package	
44EH	1102	IA32_MC19_ADDR	Package	
44FH	1103	IA32_MC19_MISC	Package	
450H	1104	IA32_MC20_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Bank MC20 reports MC error from the Intel QPI 1 module.
451H	1105	IA32_MC20_STATUS	Package	
452H	1106	IA32_MC20_ADDR	Package	
453H	1107	IA32_MC20_MISC	Package	

**Table 35-36. Additional MSRs Supported by Intel® Xeon® Processors with DisplayFamily\_DisplayModel 06\_4FH**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
454H	1108	IA32_MC21_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Bank MC21 reports MC error from the Intel QPI 2 module.
455H	1109	IA32_MC21_STATUS	Package	
456H	1110	IA32_MC21_ADDR	Package	
457H	1111	IA32_MC21_MISC	Package	
630H	1584	MSR_PKG_C8_RESIDENCY	Package	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		59:0		Package C8 Residency Counter. (R/O) Value since last reset that this package is in processor-specific C8 states. Count at the same frequency as the TSC.
		63:60		Reserved
631H	1585	MSR_PKG_C9_RESIDENCY	Package	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		59:0		Package C9 Residency Counter. (R/O) Value since last reset that this package is in processor-specific C9 states. Count at the same frequency as the TSC.
		63:60		Reserved
632H	1586	MSR_PKG_C10_RESIDENCY	Package	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		59:0		Package C10 Residency Counter. (R/O) Value since last reset that this package is in processor-specific C10 states. Count at the same frequency as the TSC.
		63:60		Reserved
C81H	3201	IA32_L3_QOS_CFG	Package	Cache Allocation Technology Configuration (R/W)
		0		CAT Enable. Set 1 to enable Cache Allocation Technology
		63:1		Reserved.
See Table 35-18, Table 35-19, Table 35-27, and Table 35-28 for other MSR definitions applicable to processors with CPUID signature 06_45H.				

**NOTES:**

1. An override configuration lower than the factory-set configuration is always supported. An override configuration higher than the factory-set configuration is dependent on features specific to the processor and the platform.

## 35.15 MSRS IN THE 6TH GENERATION INTEL® CORE™ PROCESSORS

The 6th generation Intel® Core™ processor family is based on the Skylake microarchitecture. They have CPUID DisplayFamily\_DisplayModel signatures of 06\_4EH and 06\_5EH, supports the MSR interfaces listed in Table 35-18, Table 35-19, Table 35-23, Table 35-27, Table 35-33, Table 35-37, and Table 35-38. For an MSR listed in Table 35-37 that also appears in the model-specific tables of prior generations, Table 35-37 supercede prior generation tables.

The notation of "Platform" in the Scope column (with respect to MSR\_PLATFORM\_ENERGY\_COUNTER and MSR\_PLATFORM\_POWER\_LIMIT) is limited to the power-delivery domain and the specifics of the power delivery integration may vary by platform vendor's implementation.

**Table 35-37. Additional MSRs Supported by 6th Generation Intel® Core™ Processors Based on Skylake Microarchitecture**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
3AH	58	IA32_FEATURE_CONTROL	Thread	<b>Control Features in Intel 64 Processor (R/W)</b> See Table 35-2.
		0		<b>Lock (R/WL)</b>
		1		<b>Enable VMX inside SMX operation (R/WL)</b>
		2		<b>Enable VMX outside SMX operation (R/WL)</b>
		14:8		<b>SENTER local functions enables (R/WL)</b>
		15		<b>SENTER global functions enable (R/WL)</b>
		18		<b>SGX global functions enable (R/WL)</b>
		20		<b>LMCE_ON (R/WL)</b>
		63:21		Reserved.
FEH	254	IA32_MTRRCAP	Thread	<b>MTRR Capality (RO, Architectural).</b> See Table 35-2
19CH	412	IA32_THERM_STATUS	Core	<b>Thermal Monitor Status (R/W)</b> See Table 35-2.
		0		<b>Thermal status (RO)</b> See Table 35-2.
		1		<b>Thermal status log (R/WCO)</b> See Table 35-2.
		2		<b>PROTCHOT # or FORCEPR# status (RO)</b> See Table 35-2.
		3		<b>PROTCHOT # or FORCEPR# log (R/WCO)</b> See Table 35-2.
		4		<b>Critical Temperature status (RO)</b> See Table 35-2.
		5		<b>Critical Temperature status log (R/WCO)</b> See Table 35-2.
		6		<b>Thermal threshold #1 status (RO)</b> See Table 35-2.
		7		<b>Thermal threshold #1 log (R/WCO)</b> See Table 35-2.
		8		<b>Thermal threshold #2 status (RO)</b> See Table 35-2.
		9		<b>Thermal threshold #2 log (R/WCO)</b> See Table 35-2.
		10		<b>Power Limitation status (RO)</b> See Table 35-2.
11		<b>Power Limitation log (R/WCO)</b> See Table 35-2.		

**Table 35-37. Additional MSRs Supported by 6th Generation Intel® Core™ Processors Based on Skylake Microarchitecture**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		12		<b>Current Limit status (R0)</b> See Table 35-2.
		13		<b>Current Limit log (R/WC0)</b> See Table 35-2.
		14		<b>Cross Domain Limit status (R0)</b> See Table 35-2.
		15		<b>Cross Domain Limit log (R/WC0)</b> See Table 35-2.
		22:16		<b>Digital Readout (R0)</b> See Table 35-2.
		26:23		Reserved.
		30:27		<b>Resolution in degrees Celsius (R0)</b> See Table 35-2.
		31		<b>Reading Valid (R0)</b> See Table 35-2.
		63:32		Reserved.
1ADH	429	MSR_TURBO_RATIO_LIMIT	Package	<b>Maximum Ratio Limit of Turbo Mode</b> RO if MSR_PLATFORM_INFO.[28] = 0, RW if MSR_PLATFORM_INFO.[28] = 1
		7:0	Package	<b>Maximum Ratio Limit for 1C</b> Maximum turbo ratio limit of 1 core active.
		15:8	Package	<b>Maximum Ratio Limit for 2C</b> Maximum turbo ratio limit of 2 core active.
		23:16	Package	<b>Maximum Ratio Limit for 3C</b> Maximum turbo ratio limit of 3 core active.
		31:24	Package	<b>Maximum Ratio Limit for 4C</b> Maximum turbo ratio limit of 4 core active.
		63:32		Reserved.
1C9H	457	MSR_LASTBRANCH_TOS	Thread	<b>Last Branch Record Stack TOS (R/W)</b> Contains an index (bits 0-4) that points to the MSR containing the most recent branch record.
300H	768	MSR_SGXOWNER0	Package	<b>Lower 64 Bit OwnerEpoch Component of SGX Key (R0).</b>
		63:0		<b>Low 64 bits of an 128-bit external entropy value for key derivation of an enclave.</b>
301H	768	MSR_SGXOWNER1	Package	<b>Upper 64 Bit OwnerEpoch Component of SGX Key (R0).</b>
		63:0		<b>Upper 64 bits of an 128-bit external entropy value for key derivation of an enclave.</b>
38EH	910	IA32_PERF_GLOBAL_STATUS		See Table 35-2. See Section 18.2.2.3, "Full-Width Writes to Performance Counter Registers."

**Table 35-37. Additional MSRs Supported by 6th Generation Intel® Core™ Processors Based on Skylake Microarchitecture**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		0	Thread	Ovf_PMC0
		1	Thread	Ovf_PMC1
		2	Thread	Ovf_PMC2
		3	Thread	Ovf_PMC3
		4	Thread	Ovf_PMC4 (if CPUID.0AH:EAX[15:8] > 4)
		5	Thread	Ovf_PMC5 (if CPUID.0AH:EAX[15:8] > 5)
		6	Thread	Ovf_PMC6 (if CPUID.0AH:EAX[15:8] > 6)
		7	Thread	Ovf_PMC7 (if CPUID.0AH:EAX[15:8] > 7)
		31:8		Reserved.
		32	Thread	Ovf_FixedCtr0
		33	Thread	Ovf_FixedCtr1
		34	Thread	Ovf_FixedCtr2
		54:35		Reserved.
		55	Thread	Trace_ToPA_PMI.
		57:56		Reserved.
		58	Thread	LBR_Frz.
		59	Thread	CTR_Frz.
		60	Thread	ASCI.
		61	Thread	Ovf_Uncore
		62	Thread	Ovf_BufDSSAVE
63	Thread	CondChgd		
390H	912	IA32_PERF_GLOBAL_STAT_US_RESET		See Table 35-2. See Section 18.2.2.3, "Full-Width Writes to Performance Counter Registers."
		0	Thread	Set 1 to clear Ovf_PMC0
		1	Thread	Set 1 to clear Ovf_PMC1
		2	Thread	Set 1 to clear Ovf_PMC2
		3	Thread	Set 1 to clear Ovf_PMC3
		4	Thread	Set 1 to clear Ovf_PMC4 (if CPUID.0AH:EAX[15:8] > 4)
		5	Thread	Set 1 to clear Ovf_PMC5 (if CPUID.0AH:EAX[15:8] > 5)
		6	Thread	Set 1 to clear Ovf_PMC6 (if CPUID.0AH:EAX[15:8] > 6)
		7	Thread	Set 1 to clear Ovf_PMC7 (if CPUID.0AH:EAX[15:8] > 7)
		31:8		Reserved.
		32	Thread	Set 1 to clear Ovf_FixedCtr0
		33	Thread	Set 1 to clear Ovf_FixedCtr1
		34	Thread	Set 1 to clear Ovf_FixedCtr2
		54:35		Reserved.

**Table 35-37. Additional MSRs Supported by 6th Generation Intel® Core™ Processors Based on Skylake Microarchitecture**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		55	Thread	Set 1 to clear Trace_ToPA_PMI.
		57:56		Reserved.
		58	Thread	Set 1 to clear LBR_Frz.
		59	Thread	Set 1 to clear CTR_Frz.
		60	Thread	Set 1 to clear ASCI.
		61	Thread	Set 1 to clear Ovf_Uncore
		62	Thread	Set 1 to clear Ovf_BufDSSAVE
		63	Thread	Set 1 to clear CondChgd
391H	913	IA32_PERF_GLOBAL_STAT_US_SET		See Table 35-2. See Section 18.2.2.3, "Full-Width Writes to Performance Counter Registers."
		0	Thread	Set 1 to cause Ovf_PMC0 = 1
		1	Thread	Set 1 to cause Ovf_PMC1 = 1
		2	Thread	Set 1 to cause Ovf_PMC2 = 1
		3	Thread	Set 1 to cause Ovf_PMC3 = 1
		4	Thread	Set 1 to cause Ovf_PMC4=1 (if CPUID.0AH:EAX[15:8] > 4)
		5	Thread	Set 1 to cause Ovf_PMC5=1 (if CPUID.0AH:EAX[15:8] > 5)
		6	Thread	Set 1 to cause Ovf_PMC6=1 (if CPUID.0AH:EAX[15:8] > 6)
		7	Thread	Set 1 to cause Ovf_PMC7=1 (if CPUID.0AH:EAX[15:8] > 7)
		31:8		Reserved.
		32	Thread	Set 1 to cause Ovf_FixedCtr0 = 1
		33	Thread	Set 1 to cause Ovf_FixedCtr1 = 1
		34	Thread	Set 1 to cause Ovf_FixedCtr2 = 1
		54:35		Reserved.
		55	Thread	Set 1 to cause Trace_ToPA_PMI = 1
		57:56		Reserved.
		58	Thread	Set 1 to cause LBR_Frz = 1
		59	Thread	Set 1 to cause CTR_Frz = 1
		60	Thread	Set 1 to cause ASCI = 1
		61	Thread	Set 1 to cause Ovf_Uncore
62	Thread	Set 1 to cause Ovf_BufDSSAVE		
63		Reserved.		
392H	913	IA32_PERF_GLOBAL_INUSE		See Table 35-2.
3F7H	1015	MSR_PEBBS_FRONTEND	Thread	FrontEnd Precise Event Condition Select (R/W)
		2:0		Event Code Select
		3		Reserved.
		4		Event Code Select High



**Table 35-37. Additional MSRs Supported by 6th Generation Intel® Core™ Processors Based on Skylake Microarchitecture**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		7:5		Reserved.
		19:8		<b>IDQ_Bubble_Length Specifier</b>
		22:20		<b>IDQ_Bubble_Width Specifier</b>
		63:23		<b>Reserved</b>
500H	1280	IA32_SGX_SVN_STATUS	Thread	<b>Status and SVN Threshold of SGX Support for ACM (RO).</b>
		0		<b>Lock.</b> See Section 42.11.3, "Interactions with Authenticated Code Modules (ACMs)"
		15:1		Reserved.
		23:16		<b>SGX_SVN_SINIT.</b> See Section 42.11.3, "Interactions with Authenticated Code Modules (ACMs)"
		63:24		Reserved.
560H	1376	IA32_RTIT_OUTPUT_BASE	Thread	<b>Trace Output Base Register (R/W).</b> See Table 35-2.
561H	1377	IA32_RTIT_OUTPUT_MASK_PTRS	Thread	<b>Trace Output Mask Pointers Register (R/W).</b> See Table 35-2.
570H	1392	IA32_RTIT_CTL	Thread	<b>Trace Control Register (R/W)</b>
		0		<b>TraceEn</b>
		1		<b>CYCEn</b>
		2		<b>OS</b>
		3		<b>User</b>
		6:4		Reserved, MBZ
		7		<b>CR3 filter</b>
		8		<b>ToPA; writing 0 will #GP if also setting TraceEn</b>
		9		<b>MTCEn</b>
		10		<b>TSCEn</b>
		11		<b>DisRETC</b>
		12		Reserved, MBZ
		13		<b>BranchEn</b>
		17:14		<b>MTCFreq</b>
		18		Reserved, MBZ
		22:19		<b>CYCThresh</b>
		23		Reserved, MBZ
		27:24		<b>PSBFreq</b>
		31:28		Reserved, MBZ
		35:32		<b>ADDR0_CFG</b>
39:36		<b>ADDR1_CFG</b>		
63:40		Reserved, MBZ.		
571H	1393	IA32_RTIT_STATUS	Thread	<b>Tracing Status Register (R/W)</b>

**Table 35-37. Additional MSRs Supported by 6th Generation Intel® Core™ Processors Based on Skylake Microarchitecture**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		0		<b>FilterEn</b> , writes ignored.
		1		<b>ContexEn</b> , writes ignored.
		2		<b>TriggerEn</b> , writes ignored.
		3		Reserved
		4		<b>Error (R/W)</b>
		5		<b>Stopped</b>
		31:6		Reserved. MBZ
		48:32		<b>PacketByteCnt</b>
		63:49		Reserved, MBZ.
572H	1394	IA32_RTIT_CR3_MATCH	Thread	<b>Trace Filter CR3 Match Register (R/W)</b>
		4:0		Reserved
		63:5		CR3[63:5] value to match
580H	1408	IA32_RTIT_ADDR0_A	Thread	<b>Region 0 Start Address (R/W)</b>
		63:0		See Table 35-2.
581H	1409	IA32_RTIT_ADDR0_B	Thread	<b>Region 0 End Address (R/W)</b>
		63:0		See Table 35-2.
582H	1410	IA32_RTIT_ADDR1_A	Thread	<b>Region 1 Start Address (R/W)</b>
		63:0		See Table 35-2.
583H	1411	IA32_RTIT_ADDR1_B	Thread	<b>Region 1 End Address (R/W)</b>
		63:0		See Table 35-2.
639H	1593	MSR_PPO_ENERGY_STATUS	Package	<b>PPO Energy Status (R/O)</b> See Section 14.9.4, "PPO/PP1 RAPL Domains."
64DH	1613	MSR_PLATFORM_ENERGY_COUNTER	Platform*	Platform Energy Counter. (R/O). This MSR is valid only if both platform vendor hardware implementation and BIOS enablement support it. This MSR will read 0 if not valid.
		31:0		Total energy consumed by all devices in the platform that receive power from integrated power delivery mechanism, Included platform devices are processor cores, SOC, memory, add-on or peripheral devices that get powered directly from the platform power delivery means. The energy units are specified in the MSR_RAPL_POWER_UNIT.Energy_Status_Unit.
		63:32		Reserved.
64EH	1614	MSR_PPERF	Thread	Productive Performance Count. (R/O).
		63:0		Hardware's view of workload scalability. See Section 14.4.5.1
64FH	1615	MSR_CORE_PERF_LIMIT_REASONS	Package	<b>Indicator of Frequency Clipping in Processor Cores (R/W)</b> <b>(frequency refers to processor core frequency)</b>

**Table 35-37. Additional MSRs Supported by 6th Generation Intel® Core™ Processors Based on Skylake Microarchitecture**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		0		<b>PROCHOT Status (R0)</b> When set, frequency is reduced below the operating system request due to assertion of external PROCHOT.
		1		<b>Thermal Status (R0)</b> When set, frequency is reduced below the operating system request due to a thermal event.
		3:2		Reserved.
		4		<b>Residency State Regulation Status (R0)</b> When set, frequency is reduced below the operating system request due to residency state regulation limit.
		5		<b>Running Average Thermal Limit Status (R0)</b> When set, frequency is reduced below the operating system request due to Running Average Thermal Limit (RATL).
		6		<b>VR Therm Alert Status (R0)</b> When set, frequency is reduced below the operating system request due to a thermal alert from a processor Voltage Regulator (VR).
		7		<b>VR Therm Design Current Status (R0)</b> When set, frequency is reduced below the operating system request due to VR thermal design current limit.
		8		<b>Other Status (R0)</b> When set, frequency is reduced below the operating system request due to electrical or other constraints.
		9		<b>Reserved</b>
		10		<b>Package/Platform-Level Power Limiting PL1 Status (R0)</b> When set, frequency is reduced below the operating system request due to package/platform-level power limiting PL1.
		11		<b>Package/Platform-Level PL2 Power Limiting Status (R0)</b> When set, frequency is reduced below the operating system request due to package/platform-level power limiting PL2/PL3.
		12		<b>Max Turbo Limit Status (R0)</b> When set, frequency is reduced below the operating system request due to multi-core turbo limits.
		13		<b>Turbo Transition Attenuation Status (R0)</b> When set, frequency is reduced below the operating system request due to Turbo transition attenuation. This prevents performance degradation due to frequent operating ratio changes.
		15:14		<b>Reserved</b>
		16		<b>PROCHOT Log</b> When set, indicates that the PROCHOT Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.

**Table 35-37. Additional MSRs Supported by 6th Generation Intel® Core™ Processors Based on Skylake Microarchitecture**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		17		<b>Thermal Log</b> When set, indicates that the Thermal Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		19:18		Reserved.
		20		<b>Residency State Regulation Log</b> When set, indicates that the Residency State Regulation Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		21		<b>Running Average Thermal Limit Log</b> When set, indicates that the RATL Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		22		<b>VR Therm Alert Log</b> When set, indicates that the VR Therm Alert Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		23		<b>VR Thermal Design Current Log</b> When set, indicates that the VR TDC Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		24		<b>Other Log</b> When set, indicates that the Other Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		25		<b>Reserved</b>
		26		<b>Package/Platform-Level PL1 Power Limiting Log</b> When set, indicates that the Package or Platform Level PL1 Power Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		27		<b>Package/Platform-Level PL2 Power Limiting Log</b> When set, indicates that the Package or Platform Level PL2/PL3 Power Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		28		<b>Max Turbo Limit Log</b> When set, indicates that the Max Turbo Limit Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.

**Table 35-37. Additional MSRs Supported by 6th Generation Intel® Core™ Processors Based on Skylake Microarchitecture**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		29		<b>Turbo Transition Attenuation Log</b> When set, indicates that the Turbo Transition Attenuation Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		63:30		Reserved.
652H	1618	MSR_PKG_HDC_CONFIG	Package	<b>HDC Configuration (R/W).</b>
		2:0		<b>PKG_Cx_Monitor.</b> Configures Package Cx state threshold for MSR_PKG_HDC_DEEP_RESIDENCY
		63:3		<b>Reserved</b>
653H	1619	MSR_CORE_HDC_RESIDENCY	Core	Core HDC Idle Residency. (R/O).
		63:0		Core_Cx_Duty_Cycle_Cnt.
655H	1621	MSR_PKG_HDC_SHALLOW_RESIDENCY	Package	Accumulate the cycles the package was in C2 state and at least one logical processor was in forced idle. (R/O).
		63:0		Pkg_C2_Duty_Cycle_Cnt.
656H	1622	MSR_PKG_HDC_DEEP_RESIDENCY	Package	Package Cx HDC Idle Residency. (R/O).
		63:0		Pkg_Cx_Duty_Cycle_Cnt.
658H	1624	MSR_WEIGHTED_CORE_CO	Package	Core-count Weighted C0 Residency. (R/O).
		63:0		Increment at the same rate as the TSC. The increment each cycle is weighted by the number of processor cores in the package that reside in C0. If N cores are simultaneously in C0, then each cycle the counter increments by N.
659H	1625	MSR_ANY_CORE_CO	Package	Any Core C0 Residency. (R/O)
		63:0		Increment at the same rate as the TSC. The increment each cycle is one if any processor core in the package is in C0.
65AH	1626	MSR_ANY_GFXE_CO	Package	Any Graphics Engine C0 Residency. (R/O)
		63:0		Increment at the same rate as the TSC. The increment each cycle is one if any processor graphic device's compute engines are in C0.
65BH	1627	MSR_CORE_GFXE_OVERLAP_CO	Package	Core and Graphics Engine Overlapped C0 Residency. (R/O)
		63:0		Increment at the same rate as the TSC. The increment each cycle is one if at least one compute engine of the processor graphics is in C0 and at least one processor core in the package is also in C0.

**Table 35-37. Additional MSRs Supported by 6th Generation Intel® Core™ Processors Based on Skylake Microarchitecture**

Register Address		Register Name	Scope	Bit Description	
Hex	Dec				
65CH	1628	MSR_PLATFORM_POWER_LIMIT	Platform*	<b>Platform Power Limit Control (R/W-L)</b> Allows platform BIOS to limit power consumption of the platform devices to the specified values. The Long Duration power consumption is specified via Platform_Power_Limit_1 and Platform_Power_Limit_1_Time. The Short Duration power consumption limit is specified via the Platform_Power_Limit_2 with duration chosen by the processor.  The processor implements an exponential-weighted algorithm in the placement of the time windows.	
				14:0	Platform Power Limit #1.  Average Power limit value which the platform must not exceed over a time window as specified by Power_Limit_1_TIME field.  The default value is the Thermal Design Power (TDP) and varies with product skus. The unit is specified in MSR_RAPLPOWER_UNIT.
				15	Enable Platform Power Limit #1.  When set, enables the processor to apply control policy such that the platform power does not exceed Platform Power limit #1 over the time window specified by Power Limit #1 Time Window.
				16	Platform Clamping Limitation #1.  When set, allows the processor to go below the OS requested P states in order to maintain the power below specified Platform Power Limit #1 value.  This bit is writeable only when CPUID (EAX=6):EAX[4] is set.
				23:17	Time Window for Platform Power Limit #1.  Specifies the duration of the time window over which Platform Power Limit 1 value should be maintained for sustained long duration. This field is made up of two numbers from the following equation:  $\text{Time Window} = (\text{float}) ((1+(X/4)) * (2^Y))$ , where: X = POWER_LIMIT_1_TIME[23:22] Y = POWER_LIMIT_1_TIME[21:17].  The maximum allowed value in this field is defined in MSR_PKG_POWER_INFO[PKG_MAX_WIN].  The default value is 0DH, The unit is specified in MSR_RAPLPOWER_UNIT[Time Unit].
				31:24	Reserved
				46:32	Platform Power Limit #2.  Average Power limit value which the platform must not exceed over the Short Duration time window chosen by the processor.  The recommended default value is 1.25 times the Long Duration Power Limit (i.e. Platform Power Limit # 1)
				47	Enable Platform Power Limit #2.  When set, enables the processor to apply control policy such that the platform power does not exceed Platform Power limit #2 over the Short Duration time window.

**Table 35-37. Additional MSRs Supported by 6th Generation Intel® Core™ Processors Based on Skylake Microarchitecture**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		48		Platform Clamping Limitation #2. When set, allows the processor to go below the OS requested P states in order to maintain the power below specified Platform Power Limit #2 value.
		62:49		Reserved
		63		Lock. Setting this bit will lock all other bits of this MSR until system RESET.
690H	1680	MSR_LASTBRANCH_16_FROM_IP	Thread	<b>Last Branch Record 16 From IP (R/W)</b> One of 32 triplets of last branch record registers on the last branch record stack. This part of the stack contains pointers to the source instruction. See also: <ul style="list-style-type: none"> <li>▪ Last Branch Record Stack TOS at 1C9H</li> <li>▪ Section 17.10</li> </ul>
691H	1681	MSR_LASTBRANCH_17_FROM_IP	Thread	<b>Last Branch Record 17 From IP (R/W)</b> See description of MSR_LASTBRANCH_0_FROM_IP.
692H	1682	MSR_LASTBRANCH_18_FROM_IP	Thread	<b>Last Branch Record 18 From IP (R/W)</b> See description of MSR_LASTBRANCH_0_FROM_IP.
693H	1683	MSR_LASTBRANCH_19_FROM_IP	Thread	<b>Last Branch Record 19 From IP (R/W)</b> See description of MSR_LASTBRANCH_0_FROM_IP.
694H	1684	MSR_LASTBRANCH_20_FROM_IP	Thread	<b>Last Branch Record 20 From IP (R/W)</b> See description of MSR_LASTBRANCH_0_FROM_IP.
695H	1685	MSR_LASTBRANCH_21_FROM_IP	Thread	<b>Last Branch Record 21 From IP (R/W)</b> See description of MSR_LASTBRANCH_0_FROM_IP.
696H	1686	MSR_LASTBRANCH_22_FROM_IP	Thread	<b>Last Branch Record 22 From IP (R/W)</b> See description of MSR_LASTBRANCH_0_FROM_IP.
697H	1687	MSR_LASTBRANCH_23_FROM_IP	Thread	<b>Last Branch Record 23 From IP (R/W)</b> See description of MSR_LASTBRANCH_0_FROM_IP.
698H	1688	MSR_LASTBRANCH_24_FROM_IP	Thread	<b>Last Branch Record 24 From IP (R/W)</b> See description of MSR_LASTBRANCH_0_FROM_IP.
699H	1689	MSR_LASTBRANCH_25_FROM_IP	Thread	<b>Last Branch Record 25 From IP (R/W)</b> See description of MSR_LASTBRANCH_0_FROM_IP.
69AH	1690	MSR_LASTBRANCH_26_FROM_IP	Thread	<b>Last Branch Record 26 From IP (R/W)</b> See description of MSR_LASTBRANCH_0_FROM_IP.
69BH	1691	MSR_LASTBRANCH_27_FROM_IP	Thread	<b>Last Branch Record 27 From IP (R/W)</b> See description of MSR_LASTBRANCH_0_FROM_IP.
69CH	1692	MSR_LASTBRANCH_28_FROM_IP	Thread	<b>Last Branch Record 28 From IP (R/W)</b> See description of MSR_LASTBRANCH_0_FROM_IP.
69DH	1693	MSR_LASTBRANCH_29_FROM_IP	Thread	<b>Last Branch Record 29 From IP (R/W)</b> See description of MSR_LASTBRANCH_0_FROM_IP.

**Table 35-37. Additional MSRs Supported by 6th Generation Intel® Core™ Processors Based on Skylake Microarchitecture**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
69EH	1694	MSR_LASTBRANCH_30_FROM_IP	Thread	<b>Last Branch Record 30 From IP (R/W)</b> See description of MSR_LASTBRANCH_0_FROM_IP.
69FH	1695	MSR_LASTBRANCH_31_FROM_IP	Thread	<b>Last Branch Record 31 From IP (R/W)</b> See description of MSR_LASTBRANCH_0_FROM_IP.
6BOH	1712	MSR_GRAPHICS_PERF_LIMIT_REASONS	Package	<b>Indicator of Frequency Clipping in the Processor Graphics (R/W) (frequency refers to processor graphics frequency)</b>
		0		<b>PROCHOT Status (R0)</b> When set, frequency is reduced due to assertion of external PROCHOT.
		1		<b>Thermal Status (R0)</b> When set, frequency is reduced due to a thermal event.
		4:2		Reserved.
		5		<b>Running Average Thermal Limit Status (R0)</b> When set, frequency is reduced due to running average thermal limit.
		6		<b>VR Therm Alert Status (R0)</b> When set, frequency is reduced due to a thermal alert from a processor Voltage Regulator.
		7		<b>VR Thermal Design Current Status (R0)</b> When set, frequency is reduced due to VR TDC limit.
		8		<b>Other Status (R0)</b> When set, frequency is reduced due to electrical or other constraints.
		9		<b>Reserved</b>
		10		<b>Package/Platform-Level Power Limiting PL1 Status (R0)</b> When set, frequency is reduced due to package/platform-level power limiting PL1.
		11		<b>Package/Platform-Level PL2 Power Limiting Status (R0)</b> When set, frequency is reduced due to package/platform-level power limiting PL2/PL3.
		12		<b>Inefficient Operation Status (R0)</b> When set, processor graphics frequency is operating below target frequency.
		15:13		<b>Reserved</b>
		16		<b>PROCHOT Log</b> When set, indicates that the PROCHOT Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.



**Table 35-37. Additional MSRs Supported by 6th Generation Intel® Core™ Processors Based on Skylake Microarchitecture**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		17		<b>Thermal Log</b> When set, indicates that the Thermal Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		20:18		Reserved.
		21		<b>Running Average Thermal Limit Log</b> When set, indicates that the RATL Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		22		<b>VR Therm Alert Log</b> When set, indicates that the VR Therm Alert Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		23		<b>VR Thermal Design Current Log</b> When set, indicates that the VR Therm Alert Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		24		<b>Other Log</b> When set, indicates that the OTHER Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		25		<b>Reserved</b>
		26		<b>Package/Platform-Level PL1 Power Limiting Log</b> When set, indicates that the Package/Platform Level PL1 Power Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		27		<b>Package/Platform-Level PL2 Power Limiting Log</b> When set, indicates that the Package/Platform Level PL2 Power Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		28		<b>Inefficient Operation Log</b> When set, indicates that the Inefficient Operation Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		63:29		Reserved.
6B1H	1713	MSR_RING_PERF_LIMIT_REASONS	Package	<b>Indicator of Frequency Clipping in the Ring Interconnect (R/W) (frequency refers to ring interconnect in the uncore)</b>
		0		<b>PROCHOT Status (R0)</b> When set, frequency is reduced due to assertion of external PROCHOT.

**Table 35-37. Additional MSRs Supported by 6th Generation Intel® Core™ Processors Based on Skylake Microarchitecture**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		1		<b>Thermal Status (R0)</b> When set, frequency is reduced due to a thermal event.
		4:2		Reserved.
		5		<b>Running Average Thermal Limit Status (R0)</b> When set, frequency is reduced due to running average thermal limit.
		6		<b>VR Therm Alert Status (R0)</b> When set, frequency is reduced due to a thermal alert from a processor Voltage Regulator.
		7		<b>VR Thermal Design Current Status (R0)</b> When set, frequency is reduced due to VR TDC limit.
		8		<b>Other Status (R0)</b> When set, frequency is reduced due to electrical or other constraints.
		9		Reserved.
		10		<b>Package/Platform-Level Power Limiting PL1 Status (R0)</b> When set, frequency is reduced due to package/Platform-level power limiting PL1.
		11		<b>Package/Platform-Level PL2 Power Limiting Status (R0)</b> When set, frequency is reduced due to package/Platform-level power limiting PL2/PL3.
		15:12		<b>Reserved</b>
		16		<b>PROCHOT Log</b> When set, indicates that the PROCHOT Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		17		<b>Thermal Log</b> When set, indicates that the Thermal Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		20:18		Reserved.
		21		<b>Running Average Thermal Limit Log</b> When set, indicates that the RATL Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		22		<b>VR Therm Alert Log</b> When set, indicates that the VR Therm Alert Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.

**Table 35-37. Additional MSRs Supported by 6th Generation Intel® Core™ Processors Based on Skylake Microarchitecture**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		23		<b>VR Thermal Design Current Log</b> When set, indicates that the VR Therm Alert Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		24		<b>Other Log</b> When set, indicates that the OTHER Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		25		<b>Reserved</b>
		26		<b>Package/Platform-Level PL1 Power Limiting Log</b> When set, indicates that the Package/Platform Level PL1 Power Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		27		<b>Package/Platform-Level PL2 Power Limiting Log</b> When set, indicates that the Package/Platform Level PL2 Power Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		63:38		Reserved.
6D0H	1744	MSR_LASTBRANCH_16_TO_IP	Thread	<b>Last Branch Record 16 To IP (R/W)</b> One of 32 triplets of last branch record registers on the last branch record stack. This part of the stack contains pointers to the <b>destination instruction</b> . See also: <ul style="list-style-type: none"> <li>▪ Last Branch Record Stack TOS at 1C9H</li> <li>▪ Section 17.10</li> </ul>
6D1H	1745	MSR_LASTBRANCH_17_TO_IP	Thread	<b>Last Branch Record 17 To IP (R/W)</b> See description of MSR_LASTBRANCH_0_TO_IP.
6D2H	1746	MSR_LASTBRANCH_18_TO_IP	Thread	<b>Last Branch Record 18 To IP (R/W)</b> See description of MSR_LASTBRANCH_0_TO_IP.
6D3H	1747	MSR_LASTBRANCH_19_TO_IP	Thread	<b>Last Branch Record 19 To IP (R/W)</b> See description of MSR_LASTBRANCH_0_TO_IP.
6D4H	1748	MSR_LASTBRANCH_20_TO_IP	Thread	<b>Last Branch Record 20 To IP (R/W)</b> See description of MSR_LASTBRANCH_0_TO_IP.
6D5H	1749	MSR_LASTBRANCH_21_TO_IP	Thread	<b>Last Branch Record 21 To IP (R/W)</b> See description of MSR_LASTBRANCH_0_TO_IP.
6D6H	1750	MSR_LASTBRANCH_22_TO_IP	Thread	<b>Last Branch Record 22 To IP (R/W)</b> See description of MSR_LASTBRANCH_0_TO_IP.
6D7H	1751	MSR_LASTBRANCH_23_TO_IP	Thread	<b>Last Branch Record 23 To IP (R/W)</b> See description of MSR_LASTBRANCH_0_TO_IP.
6D8H	1752	MSR_LASTBRANCH_24_TO_IP	Thread	<b>Last Branch Record 24 To IP (R/W)</b> See description of MSR_LASTBRANCH_0_TO_IP.

**Table 35-37. Additional MSRs Supported by 6th Generation Intel® Core™ Processors Based on Skylake Microarchitecture**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
6D9H	1753	MSR_LASTBRANCH_25_TO_IP	Thread	<b>Last Branch Record 25 To IP (R/W)</b> See description of MSR_LASTBRANCH_0_TO_IP.
6DAH	1754	MSR_LASTBRANCH_26_TO_IP	Thread	<b>Last Branch Record 26 To IP (R/W)</b> See description of MSR_LASTBRANCH_0_TO_IP.
6DBH	1755	MSR_LASTBRANCH_27_TO_IP	Thread	<b>Last Branch Record 27 To IP (R/W)</b> See description of MSR_LASTBRANCH_0_TO_IP.
6DCH	1756	MSR_LASTBRANCH_28_TO_IP	Thread	<b>Last Branch Record 28 To IP (R/W)</b> See description of MSR_LASTBRANCH_0_TO_IP.
6DDH	1757	MSR_LASTBRANCH_29_TO_IP	Thread	<b>Last Branch Record 29 To IP (R/W)</b> See description of MSR_LASTBRANCH_0_TO_IP.
6DEH	1758	MSR_LASTBRANCH_30_TO_IP	Thread	<b>Last Branch Record 30 To IP (R/W)</b> See description of MSR_LASTBRANCH_0_TO_IP.
6DFH	1759	MSR_LASTBRANCH_31_TO_IP	Thread	<b>Last Branch Record 31 To IP (R/W)</b> See description of MSR_LASTBRANCH_0_TO_IP.
770H	1904	IA32_PM_ENABLE	Package	See Section 14.4.2, “Enabling HWP”
771H	1905	IA32_HWP_CAPABILITIES	Thread	See Section 14.4.3, “HWP Performance Range and Dynamic Capabilities”
772H	1906	IA32_HWP_REQUEST_PKG	Package	See Section 14.4.4, “Managing HWP”
773H	1907	IA32_HWP_INTERRUPT	Thread	See Section 14.4.6, “HWP Notifications”
774H	1908	IA32_HWP_REQUEST	Thread	See Section 14.4.4, “Managing HWP”
		7:0		<b>Minimum Performance (R/W).</b>
		15:8		<b>Maximum Performance (R/W).</b>
		23:16		<b>Desired Performance (R/W).</b>
		31:24		<b>Energy/Performance Preference (R/W).</b>
		41:32		<b>Activity Window (R/W).</b>
		42		<b>Package Control (R/W).</b>
		63:43		Reserved.
777H	1911	IA32_HWP_STATUS	Thread	See Section 14.4.5, “HWP Feedback”
D90H	3472	IA32_BNDCFGS	Thread	See Table 35-2.
DA0H	3488	IA32_XSS	Thread	See Table 35-2.
DB0H	3504	IA32_PKG_HDC_CTL	Package	See Section 14.5.2, “Package level Enabling HDC”
DB1H	3505	IA32_PM_CTL1	Thread	See Section 14.5.3, “Logical-Processor Level HDC Control”
DB2H	3506	IA32_THREAD_STALL	Thread	See Section 14.5.4.1, “IA32_THREAD_STALL”

**Table 35-37. Additional MSRs Supported by 6th Generation Intel® Core™ Processors Based on Skylake Microarchitecture**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
DC0H	3520	MSR_LBR_INFO_0	Thread	<b>Last Branch Record 0 Additional Information (R/W)</b> One of 32 triplet of last branch record registers on the last branch record stack. This part of the stack contains flag, TSX-related and elapsed cycle information. See also: <ul style="list-style-type: none"> <li>▪ Last Branch Record Stack TOS at 1C9H</li> <li>▪ Section 17.7.1, "LBR Stack."</li> </ul>
DC1H	3521	MSR_LBR_INFO_1	Thread	<b>Last Branch Record 1 Additional Information (R/W)</b> See description of MSR_LBR_INFO_0.
DC2H	3522	MSR_LBR_INFO_2	Thread	<b>Last Branch Record 2 Additional Information (R/W)</b> See description of MSR_LBR_INFO_0.
DC3H	3523	MSR_LBR_INFO_3	Thread	<b>Last Branch Record 3 Additional Information (R/W)</b> See description of MSR_LBR_INFO_0.
DC4H	3524	MSR_LBR_INFO_4	Thread	<b>Last Branch Record 4 Additional Information (R/W)</b> See description of MSR_LBR_INFO_0.
DC5H	3525	MSR_LBR_INFO_5	Thread	<b>Last Branch Record 5 Additional Information (R/W)</b> See description of MSR_LBR_INFO_0.
DC6H	3526	MSR_LBR_INFO_6	Thread	<b>Last Branch Record 6 Additional Information (R/W)</b> See description of MSR_LBR_INFO_0.
DC7H	3527	MSR_LBR_INFO_7	Thread	<b>Last Branch Record 7 Additional Information (R/W)</b> See description of MSR_LBR_INFO_0.
DC8H	3528	MSR_LBR_INFO_8	Thread	<b>Last Branch Record 8 Additional Information (R/W)</b> See description of MSR_LBR_INFO_0.
DC9H	3529	MSR_LBR_INFO_9	Thread	<b>Last Branch Record 9 Additional Information (R/W)</b> See description of MSR_LBR_INFO_0.
DCAH	3530	MSR_LBR_INFO_10	Thread	<b>Last Branch Record 10 Additional Information (R/W)</b> See description of MSR_LBR_INFO_0.
DCBH	3531	MSR_LBR_INFO_11	Thread	<b>Last Branch Record 11 Additional Information (R/W)</b> See description of MSR_LBR_INFO_0.
DCCH	3532	MSR_LBR_INFO_12	Thread	<b>Last Branch Record 12 Additional Information (R/W)</b> See description of MSR_LBR_INFO_0.
DCDH	3533	MSR_LBR_INFO_13	Thread	<b>Last Branch Record 13 Additional Information (R/W)</b> See description of MSR_LBR_INFO_0.
DCEH	3534	MSR_LBR_INFO_14	Thread	<b>Last Branch Record 14 Additional Information (R/W)</b> See description of MSR_LBR_INFO_0.
DCFH	3535	MSR_LBR_INFO_15	Thread	<b>Last Branch Record 15 Additional Information (R/W)</b> See description of MSR_LBR_INFO_0.
DDOH	3536	MSR_LBR_INFO_16	Thread	<b>Last Branch Record 16 Additional Information (R/W)</b> See description of MSR_LBR_INFO_0.

**Table 35-37. Additional MSRs Supported by 6th Generation Intel® Core™ Processors Based on Skylake Microarchitecture**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
DD1H	3537	MSR_LBR_INFO_17	Thread	<b>Last Branch Record 17 Additional Information (R/W)</b> See description of MSR_LBR_INFO_0.
DD2H	3538	MSR_LBR_INFO_18	Thread	<b>Last Branch Record 18 Additional Information (R/W)</b> See description of MSR_LBR_INFO_0.
DD3H	3539	MSR_LBR_INFO_19	Thread	<b>Last Branch Record 19 Additional Information (R/W)</b> See description of MSR_LBR_INFO_0.
DD4H	3520	MSR_LBR_INFO_20	Thread	<b>Last Branch Record 20 Additional Information (R/W)</b> See description of MSR_LBR_INFO_0.
DD5H	3521	MSR_LBR_INFO_21	Thread	<b>Last Branch Record 21 Additional Information (R/W)</b> See description of MSR_LBR_INFO_0.
DD6H	3522	MSR_LBR_INFO_22	Thread	<b>Last Branch Record 22 Additional Information (R/W)</b> See description of MSR_LBR_INFO_0.
DD7H	3523	MSR_LBR_INFO_23	Thread	<b>Last Branch Record 23 Additional Information (R/W)</b> See description of MSR_LBR_INFO_0.
DD8H	3524	MSR_LBR_INFO_24	Thread	<b>Last Branch Record 24 Additional Information (R/W)</b> See description of MSR_LBR_INFO_0.
DD9H	3525	MSR_LBR_INFO_25	Thread	<b>Last Branch Record 25 Additional Information (R/W)</b> See description of MSR_LBR_INFO_0.
DDAH	3526	MSR_LBR_INFO_26	Thread	<b>Last Branch Record 26 Additional Information (R/W)</b> See description of MSR_LBR_INFO_0.
DDBH	3527	MSR_LBR_INFO_27	Thread	<b>Last Branch Record 27 Additional Information (R/W)</b> See description of MSR_LBR_INFO_0.
DDCH	3528	MSR_LBR_INFO_28	Thread	<b>Last Branch Record 28 Additional Information (R/W)</b> See description of MSR_LBR_INFO_0.
DDDH	3529	MSR_LBR_INFO_29	Thread	<b>Last Branch Record 29 Additional Information (R/W)</b> See description of MSR_LBR_INFO_0.
DDEH	3530	MSR_LBR_INFO_30	Thread	<b>Last Branch Record 30 Additional Information (R/W)</b> See description of MSR_LBR_INFO_0.
DDFH	3531	MSR_LBR_INFO_31	Thread	<b>Last Branch Record 31 Additional Information (R/W)</b> See description of MSR_LBR_INFO_0.

Table 35-38 lists the MSRs of uncore PMU for Intel processors with CPUID DisplayFamily\_DisplayModel signatures of 06\_4EH and 06\_5EH.

**Table 35-38. Uncore PMU MSRs Supported by 6th Generation Intel® Core™ Processors**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
394H	916	MSR_UNC_PERF_FIXED_CTRL	Package	Uncore fixed counter control (R/W)

**Table 35-38. Uncore PMU MSRs Supported by 6th Generation Intel® Core™ Processors**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		19:0		Reserved
		20		Enable overflow propagation
		21		Reserved
		22		Enable counting
		63:23		Reserved.
395H	917	MSR_UNC_PERF_FIXED_CTR	Package	Uncore fixed counter
		43:0		Current count
		63:44		Reserved.
396H	918	MSR_UNC_CBO_CONFIG	Package	Uncore C-Box configuration information (R/O)
		3:0		Specifies the number of C-Box units with programmable counters (including processor cores and processor graphics),
		63:4		Reserved.
3B0H	946	MSR_UNC_ARB_PERFCTR0	Package	Uncore Arb unit, performance counter 0
3B1H	947	MSR_UNC_ARB_PERFCTR1	Package	Uncore Arb unit, performance counter 1
3B2H	944	MSR_UNC_ARB_PERFEVTSELO	Package	Uncore Arb unit, counter 0 event select MSR
3B3H	945	MSR_UNC_ARB_PERFEVTSEL1	Package	Uncore Arb unit, counter 1 event select MSR
700H	1792	MSR_UNC_CBO_0_PERFEVTSELO	Package	Uncore C-Box 0, counter 0 event select MSR
701H	1793	MSR_UNC_CBO_0_PERFEVTSEL1	Package	Uncore C-Box 0, counter 1 event select MSR
706H	1798	MSR_UNC_CBO_0_PERFCTR0	Package	Uncore C-Box 0, performance counter 0
707H	1799	MSR_UNC_CBO_0_PERFCTR1	Package	Uncore C-Box 0, performance counter 1
710H	1808	MSR_UNC_CBO_1_PERFEVTSELO	Package	Uncore C-Box 1, counter 0 event select MSR
711H	1809	MSR_UNC_CBO_1_PERFEVTSEL1	Package	Uncore C-Box 1, counter 1 event select MSR
716H	1814	MSR_UNC_CBO_1_PERFCTR0	Package	Uncore C-Box 1, performance counter 0
717H	1815	MSR_UNC_CBO_1_PERFCTR1	Package	Uncore C-Box 1, performance counter 1
720H	1824	MSR_UNC_CBO_2_PERFEVTSELO	Package	Uncore C-Box 2, counter 0 event select MSR
721H	1825	MSR_UNC_CBO_2_PERFEVTSEL1	Package	Uncore C-Box 2, counter 1 event select MSR
726H	1830	MSR_UNC_CBO_2_PERFCTR0	Package	Uncore C-Box 2, performance counter 0
727H	1831	MSR_UNC_CBO_2_PERFCTR1	Package	Uncore C-Box 2, performance counter 1

**Table 35-38. Uncore PMU MSRs Supported by 6th Generation Intel® Core™ Processors**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
730H	1840	MSR_UNC_CBO_3_PERFEVTSELO	Package	Uncore C-Box 3, counter 0 event select MSR
731H	1841	MSR_UNC_CBO_3_PERFEVTSEL1	Package	Uncore C-Box 3, counter 1 event select MSR.
736H	1846	MSR_UNC_CBO_3_PERFCTRO	Package	Uncore C-Box 3, performance counter 0.
737H	1847	MSR_UNC_CBO_3_PERFCTR1	Package	Uncore C-Box 3, performance counter 1.
E01H	3585	MSR_UNC_PERF_GLOBAL_CTRL	Package	Uncore PMU global control
		0		Slice 0 select
		1		Slice 1 select
		2		Slice 2 select
		3		Slice 3 select
		4		Slice 4select
		18:5		Reserved.
		29		Enable all uncore counters
		30		Enable wake on PMI
		31		Enable Freezing counter when overflow
63:32	Reserved.			
E02H	3586	MSR_UNC_PERF_GLOBAL_STATUS	Package	Uncore PMU main status
		0		Fixed counter overflowed
		1		An ARB counter overflowed
		2		Reserved
		3		A CBox counter overflowed (on any slice)
		63:4		Reserved.

## 35.16 MSRS IN FUTURE INTEL® XEON® PROCESSORS

Future Intel® Xeon® Processors (CPUID DisplayFamily\_DisplayModel = 06\_55H) support the machine check bank registers listed in Table 35-40.

**Table 35-39. Machine Check MSRs Supported by Future Intel® Xeon® Processors with DisplayFamily\_DisplayModel 06\_55H**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
280H	640	IA32_MC0_CTL2	Package	See Table 35-2.
281H	641	IA32_MC1_CTL2	Package	See Table 35-2.
282H	642	IA32_MC2_CTL2	Package	See Table 35-2.



**Table 35-39. Machine Check MSRs Supported by Future Intel® Xeon® Processors with DisplayFamily\_DisplayModel 06\_55H**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
283H	643	IA32_MC3_CTL2	Package	See Table 35-2.
284H	644	IA32_MC4_CTL2	Package	See Table 35-2.
285H	645	IA32_MC5_CTL2	Package	See Table 35-2.
286H	646	IA32_MC6_CTL2	Package	See Table 35-2.
287H	647	IA32_MC7_CTL2	Package	See Table 35-2.
288H	648	IA32_MC8_CTL2	Package	See Table 35-2.
289H	649	IA32_MC9_CTL2	Package	See Table 35-2.
28AH	650	IA32_MC10_CTL2	Package	See Table 35-2.
28BH	651	IA32_MC11_CTL2	Package	See Table 35-2.
28CH	652	IA32_MC12_CTL2	Package	See Table 35-2.
28DH	653	IA32_MC13_CTL2	Package	See Table 35-2.
28EH	654	IA32_MC14_CTL2	Package	See Table 35-2.
28FH	655	IA32_MC15_CTL2	Package	See Table 35-2.
290H	656	IA32_MC16_CTL2	Package	See Table 35-2.
291H	657	IA32_MC17_CTL2	Package	See Table 35-2.
292H	658	IA32_MC18_CTL2	Package	See Table 35-2.
293H	659	IA32_MC19_CTL2	Package	See Table 35-2.
400H	1024	IA32_MCO_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Bank MCO reports MC error from the IFU module.
401H	1025	IA32_MCO_STATUS	Package	
402H	1026	IA32_MCO_ADDR	Package	
403H	1027	IA32_MCO_MISC	Package	
404H	1028	IA32_MC1_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Bank MC1 reports MC error from the DCU module.
405H	1029	IA32_MC1_STATUS	Package	
406H	1030	IA32_MC1_ADDR	Package	
407H	1031	IA32_MC1_MISC	Package	
408H	1032	IA32_MC2_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Bank MC2 reports MC error from the DTLB module.
409H	1033	IA32_MC2_STATUS	Package	
40AH	1034	IA32_MC2_ADDR	Package	
40BH	1035	IA32_MC2_MISC	Package	
40CH	1036	IA32_MC3_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Bank MC3 reports MC error from the MLC module.
40DH	1037	IA32_MC3_STATUS	Package	
40EH	1038	IA32_MC3_ADDR	Package	
40FH	1039	IA32_MC3_MISC	Package	

**Table 35-39. Machine Check MSRs Supported by Future Intel® Xeon® Processors with DisplayFamily\_DisplayModel 06\_55H**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
410H	1040	IA32_MC4_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Bank MC4 reports MC error from the PCU module.
411H	1041	IA32_MC4_STATUS	Package	
412H	1042	IA32_MC4_ADDR	Package	
413H	1043	IA32_MC4_MISC	Package	
414H	1044	IA32_MC5_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Bank MC5 reports MC error from a link interconnect module.
415H	1045	IA32_MC5_STATUS	Package	
416H	1046	IA32_MC5_ADDR	Package	
417H	1047	IA32_MC5_MISC	Package	
418H	1048	IA32_MC6_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Bank MC6 reports MC error from the integrated I/O module.
419H	1049	IA32_MC6_STATUS	Package	
41AH	1050	IA32_MC6_ADDR	Package	
41BH	1051	IA32_MC6_MISC	Package	
41CH	1052	IA32_MC7_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Bank MC7 reports MC error from the M2M 0.
41DH	1053	IA32_MC7_STATUS	Package	
41EH	1054	IA32_MC7_ADDR	Package	
41FH	1055	IA32_MC7_MISC	Package	
420H	1056	IA32_MC8_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Bank MC8 reports MC error from the M2M 1.
421H	1057	IA32_MC8_STATUS	Package	
422H	1058	IA32_MC8_ADDR	Package	
423H	1059	IA32_MC8_MISC	Package	
424H	1060	IA32_MC9_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Banks MC9 - MC11 report MC error from the CHA
425H	1061	IA32_MC9_STATUS	Package	
426H	1062	IA32_MC9_ADDR	Package	
427H	1063	IA32_MC9_MISC	Package	
428H	1064	IA32_MC10_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Banks MC9 - MC11 report MC error from the CHA.
429H	1065	IA32_MC10_STATUS	Package	
42AH	1066	IA32_MC10_ADDR	Package	
42BH	1067	IA32_MC10_MISC	Package	
42CH	1068	IA32_MC11_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Banks MC9 - MC11 report MC error from the CHA.
42DH	1069	IA32_MC11_STATUS	Package	
42EH	1070	IA32_MC11_ADDR	Package	
42FH	1071	IA32_MC11_MISC	Package	
430H	1072	IA32_MC12_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Banks MC12 report MC error from each channel of a link interconnect module.
431H	1073	IA32_MC12_STATUS	Package	
432H	1074	IA32_MC12_ADDR	Package	
433H	1075	IA32_MC12_MISC	Package	

**Table 35-39. Machine Check MSRs Supported by Future Intel® Xeon® Processors with DisplayFamily\_DisplayModel 06\_55H**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
434H	1076	IA32_MC13_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Banks MC13 through MC 18 report MC error from the integrated memory controllers.
435H	1077	IA32_MC13_STATUS	Package	
436H	1078	IA32_MC13_ADDR	Package	
437H	1079	IA32_MC13_MISC	Package	
438H	1080	IA32_MC14_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Banks MC13 through MC 18 report MC error from the integrated memory controllers.
439H	1081	IA32_MC14_STATUS	Package	
43AH	1082	IA32_MC14_ADDR	Package	
43BH	1083	IA32_MC14_MISC	Package	
43CH	1084	IA32_MC15_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Banks MC13 through MC 18 report MC error from the integrated memory controllers.
43DH	1085	IA32_MC15_STATUS	Package	
43EH	1086	IA32_MC15_ADDR	Package	
43FH	1087	IA32_MC15_MISC	Package	
440H	1088	IA32_MC16_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Banks MC13 through MC 18 report MC error from the integrated memory controllers
441H	1089	IA32_MC16_STATUS	Package	
442H	1090	IA32_MC16_ADDR	Package	
443H	1091	IA32_MC16_MISC	Package	
444H	1092	IA32_MC17_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Banks MC13 through MC 18 report MC error from the integrated memory controllers.
445H	1093	IA32_MC17_STATUS	Package	
446H	1094	IA32_MC17_ADDR	Package	
447H	1095	IA32_MC17_MISC	Package	
448H	1096	IA32_MC18_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Banks MC13 through MC 18 report MC error from the integrated memory controllers.
449H	1097	IA32_MC18_STATUS	Package	
44AH	1098	IA32_MC18_ADDR	Package	
44BH	1099	IA32_MC18_MISC	Package	
44CH	1100	IA32_MC19_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs." Bank MC19 reports MC error from a link interconnect module.
44DH	1101	IA32_MC19_STATUS	Package	
44EH	1102	IA32_MC19_ADDR	Package	
44FH	1103	IA32_MC19_MISC	Package	

### 35.17 MSRS IN INTEL® XEON PHI™ PROCESSOR 3200/5200/7200 SERIES

Intel® Xeon Phi™ processor 3200, 5200, 7200 series, with CPUID DisplayFamily\_DisplayModel signature 06\_57H, supports the MSR interfaces listed in Table 35-40. These processors are based on the Knights Landing microarchitecture. Some MSRs are shared between a pair of processor cores, the scope is marked as module.

**Table 35-40. Selected MSRs Supported by Intel® Xeon Phi™ Processors with DisplayFamily\_DisplayModel Signature 06\_57H**

Address		Register Name	Scope	Bit Description
Hex	Dec			
0H	0	IA32_P5_MC_ADDR	Module	See Section 35.22, "MSRs in Pentium Processors."
1H	1	IA32_P5_MC_TYPE	Module	See Section 35.22, "MSRs in Pentium Processors."
6H	6	IA32_MONITOR_FILTER_SIZE	Thread	See Section 8.10.5, "Monitor/Mwait Address Range Determination," and Table 35-2
10H	16	IA32_TIME_STAMP_COUNTER	Thread	See Section 17.15, "Time-Stamp Counter," and see Table 35-2.
17H	23	IA32_PLATFORM_ID	Package	<b>Platform ID (R)</b> See Table 35-2.
1BH	27	IA32_APIC_BASE	Thread	See Section 10.4.4, "Local APIC Status and Location," and Table 35-2.
34H	52	MSR_SMI_COUNT	Thread	<b>SMI Counter (R/O)</b>
		31:0		<b>SMI Count (R/O)</b>
		63:32		Reserved.
3AH	58	IA32_FEATURE_CONTROL	Thread	<b>Control Features in Intel 64 Processor (R/W)</b> See Table 35-2.
		0		<b>Lock (R/WL)</b>
		1		<b>Reserved</b>
		2		<b>Enable VMX outside SMX operation (R/WL)</b>
3BH	59	IA32_TSC_ADJUST	THREAD	<b>Per-Logical-Processor TSC ADJUST (R/W)</b> See Table 35-2.
79H	121	IA32_BIOS_UPDT_TRIG	Core	<b>BIOS Update Trigger Register (W)</b> See Table 35-2.
8BH	139	IA32_BIOS_SIGN_ID	THREAD	<b>BIOS Update Signature ID (RO)</b> See Table 35-2.
C1H	193	IA32_PMC0	THREAD	<b>Performance counter register</b> See Table 35-2.
C2H	194	IA32_PMC1	THREAD	<b>Performance Counter Register</b> See Table 35-2.
CEH	206	MSR_PLATFORM_INFO	Package	See <a href="http://biosbits.org">http://biosbits.org</a> .
		7:0		Reserved.
		15:8	Package	<b>Maximum Non-Turbo Ratio (R/O)</b> This is the ratio of the frequency that invariant TSC runs at. Frequency = ratio * 100 MHz.
		27:16		Reserved.
		28	Package	<b>Programmable Ratio Limit for Turbo Mode (R/O)</b> When set to 1, indicates that Programmable Ratio Limits for Turbo mode is enabled, and when set to 0, indicates Programmable Ratio Limits for Turbo mode is disabled.

**Table 35-40. Selected MSRs Supported by Intel® Xeon Phi™ Processors with DisplayFamily\_DisplayModel Signature 06\_57H**

Address		Register Name	Scope	Bit Description
Hex	Dec			
		29	Package	<b>Programmable TDP Limit for Turbo Mode (R/O)</b> When set to 1, indicates that TDP Limits for Turbo mode are programmable, and when set to 0, indicates TDP Limit for Turbo mode is not programmable.
		39:30		Reserved.
		47:40	Package	<b>Maximum Efficiency Ratio (R/O)</b> The is the minimum ratio (maximum efficiency) that the processor can operates, in units of 100MHz.
		63:48		Reserved.
E2H	226	MSR_PKG_CST_CONFIG_CONTROL	Module	<b>C-State Configuration Control (R/W)</b>
		2:0		<b>Package C-State Limit (R/W)</b> The following C-state code name encodings are supported: 000b: C0/C1 001b: C2 010b: C6 No Retention 011b: C6 Retention 111b: No limit
		9:3		Reserved.
		10		<b>I/O MWAIT Redirection Enable (R/W)</b>
		14:11		Reserved.
		15		<b>CFG Lock (R/WO)</b>
		63:16		Reserved.
		E4H	228	MSR_PMG_IO_CAPTURE_BASE
15:0				<b>LVL_2 Base Address (R/W)</b>
18:16				<b>C-state Range (R/W)</b> Specifies the encoding value of the maximum C-State code name to be included when IO read to MWAIT redirection is enabled by MSR_PKG_CST_CONFIG_CONTROL[bit10]: 100b - C4 is the max C-State to include 110b - C6 is the max C-State to include
63:19				Reserved.
E7H	231	IA32_MPERF	Thread	<b>Maximum Performance Frequency Clock Count (RW)</b> See Table 35-2.
E8H	232	IA32_APERF	Thread	<b>Actual Performance Frequency Clock Count (RW)</b> See Table 35-2.
FEH	254	IA32_MTRRCAP	Core	<b>Memory Type Range Register (R)</b> See Table 35-2.

**Table 35-40. Selected MSRs Supported by Intel® Xeon Phi™ Processors with DisplayFamily\_DisplayModel Signature 06\_57H**

Address		Register Name	Scope	Bit Description
Hex	Dec			
13CH	52	MSR_FEATURE_CONFIG	Core	<b>AES Configuration (RW-L)</b> Privileged post-BIOS agent must provide a #GP handler to handle unsuccessful read of this MSR.
		1:0		<b>AES Configuration (RW-L)</b> Upon a successful read of this MSR, the configuration of AES instruction set availability is as follows: 11b: AES instructions are not available until next RESET. otherwise, AES instructions are available. Note, AES instruction set is not available if read is unsuccessful. If the configuration is not 01b, AES instruction can be mis-configured if a privileged agent unintentionally writes 11b.
		63:2		Reserved.
174H	372	IA32_SYSENTER_CS	Thread	See Table 35-2.
175H	373	IA32_SYSENTER_ESP	Thread	See Table 35-2.
176H	374	IA32_SYSENTER_EIP	Thread	See Table 35-2.
179H	377	IA32_MCG_CAP	Thread	See Table 35-2.
17AH	378	IA32_MCG_STATUS	Thread	See Table 35-2.
17DH	390	MSR_SMM_MCA_CAP	THREAD	<b>Enhanced SMM Capabilities (SMM-RO)</b> Reports SMM capability Enhancement. Accessible only while in SMM.
		57:0		<b>Reserved</b>
		58		<b>SMM_Code_Access_Chk (SMM-RO)</b> If set to 1 indicates that the SMM code access restriction is supported and a host-space interface available to SMM handler.
		59		<b>Long_Flow_Indication (SMM-RO)</b> If set to 1 indicates that the SMM long flow indicator is supported and a host-space interface available to SMM handler.
		63:60		Reserved
186H	390	IA32_PERFEVTSELO	Thread	Performance Monitoring Event Select Register (R/W) See Table 35-2.
		7:0		<b>Event Select</b>
		15:8		<b>UMask</b>
		16		<b>USR</b>
		17		<b>OS</b>
		18		<b>Edge</b>
		19		<b>PC</b>
		20		<b>INT</b>
		21		<b>AnyThread</b>
		22		<b>EN</b>
		23		<b>INV</b>

**Table 35-40. Selected MSRs Supported by Intel® Xeon Phi™ Processors with DisplayFamily\_DisplayModel Signature 06\_57H**

Address		Register Name	Scope	Bit Description
Hex	Dec			
		31:24		<b>CMASK</b>
		63:32		Reserved.
187H	391	IA32_PERFEVTSEL1	Thread	See Table 35-2.
198H	408	IA32_PERF_STATUS	Package	See Table 35-2.
199H	409	IA32_PERF_CTL	Thread	See Table 35-2.
19AH	410	IA32_CLOCK_MODULATION	Thread	<b>Clock Modulation (R/W)</b> See Table 35-2.
19BH	411	IA32_THERM_INTERRUPT	Module	<b>Thermal Interrupt Control (R/W)</b> See Table 35-2.
19CH	412	IA32_THERM_STATUS	Module	<b>Thermal Monitor Status (R/W)</b> See Table 35-2.
		0		<b>Thermal status (RO)</b>
		1		<b>Thermal status log (R/WCO)</b>
		2		<b>PROTCHOT # or FORCEPR# status (RO)</b>
		3		<b>PROTCHOT # or FORCEPR# log (R/WCO)</b>
		4		<b>Critical Temperature status (RO)</b>
		5		<b>Critical Temperature status log (R/WCO)</b>
		6		<b>Thermal threshold #1 status (RO)</b>
		7		<b>Thermal threshold #1 log (R/WCO)</b>
		8		<b>Thermal threshold #2 status (RO)</b>
		9		<b>Thermal threshold #2 log (R/WCO)</b>
		10		<b>Power Limitation status (RO)</b>
		11		<b>Power Limitation log (R/WCO)</b>
		15:12		Reserved.
		22:16		<b>Digital Readout (RO)</b>
		26:23		Reserved.
		30:27		<b>Resolution in degrees Celsius (RO)</b>
		31		<b>Reading Valid (RO)</b>
63:32		Reserved.		
1A0H	416	IA32_MISC_ENABLE	Thread	<b>Enable Misc. Processor Features (R/W)</b> Allows a variety of processor functions to be enabled and disabled.
		0		<b>Fast-Strings Enable</b>
		2:1		Reserved.
		3		<b>Automatic Thermal Control Circuit Enable (R/W)</b> Default value is 1.
		6:4		Reserved.
		7		<b>Performance Monitoring Available (R)</b>
		10:8		Reserved.

**Table 35-40. Selected MSRs Supported by Intel® Xeon Phi™ Processors with DisplayFamily\_DisplayModel Signature 06\_57H**

Address		Register Name	Scope	Bit Description
Hex	Dec			
		11		<b>Branch Trace Storage Unavailable (RO)</b>
		12		<b>Processor Event Based Sampling Unavailable (RO)</b>
		15:13		Reserved.
		16		<b>Enhanced Intel SpeedStep Technology Enable (R/W)</b>
		18		<b>ENABLE MONITOR FSM (R/W)</b>
		21:19		Reserved.
		22		<b>Limit CPUID Maxval (R/W)</b>
		23		<b>xTPR Message Disable (R/W)</b>
		33:24		Reserved.
		34		<b>XD Bit Disable (R/W)</b>
		37:35		Reserved.
		38		<b>Turbo Mode Disable (R/W)</b>
		63:39		Reserved.
1A2H	418	MSR_TEMPERATURE_TARGET	Package	
		15:0		Reserved.
		23:16		<b>Temperature Target (R)</b>
		29:24		<b>Target Offset (R/W)</b>
		63:30		Reserved.
1A4H	420	MSR_MISC_FEATURE_CONTROL		<b>Miscellaneous Feature Control (R/W)</b>
		0	Core	<b>DCU Hardware Prefetcher Disable (R/W)</b> If 1, disables the L1 data cache prefetcher.
		1	Core	<b>L2 Hardware Prefetcher Disable (R/W)</b> If 1, disables the L2 hardware prefetcher.
		63:2		Reserved.
1A6H	422	MSR_OFFCORE_RSP_0	Shared	<b>Offcore Response Event Select Register (R/W)</b>
1A7H	423	MSR_OFFCORE_RSP_1	Shared	<b>Offcore Response Event Select Register (R/W)</b>
1ADH	429	MSR_TURBO_RATIO_LIMIT	Package	<b>Maximum Ratio Limit of Turbo Mode for Groups of Cores (RW)</b>
		0		<b>Reserved</b>
		7:1	Package	<b>Maximum Number of Cores in Group 0</b> Number active processor cores which operates under the maximum ratio limit for group 0.
		15:8	Package	<b>Maximum Ratio Limit for Group 0</b> Maximum turbo ratio limit when the number of active cores are not more than the group 0 maximum core count.



**Table 35-40. Selected MSRs Supported by Intel® Xeon Phi™ Processors with DisplayFamily\_DisplayModel Signature 06\_57H**

Address		Register Name	Scope	Bit Description
Hex	Dec			
		20:16	Package	<b>Number of Incremental Cores Added to Group 1</b> Group 1, which includes the specified number of additional cores plus the cores in group 0, operates under the group 1 turbo max ratio limit = "group 0 Max ratio limit" - "group ratio delta for group 1".
		23:21	Package	<b>Group Ratio Delta for Group 1</b> An unsigned integer specifying the ratio decrement relative to the Max ratio limit to Group 0.
		28:24	Package	<b>Number of Incremental Cores Added to Group 2</b> Group 2, which includes the specified number of additional cores plus all the cores in group 1, operates under the group 2 turbo max ratio limit = "group 1 Max ratio limit" - "group ratio delta for group 2".
		31:29	Package	<b>Group Ratio Delta for Group 2</b> An unsigned integer specifying the ratio decrement relative to the Max ratio limit for Group 1.
		36:32	Package	<b>Number of Incremental Cores Added to Group 3</b> Group 3, which includes the specified number of additional cores plus all the cores in group 2, operates under the group 3 turbo max ratio limit = "group 2 Max ratio limit" - "group ratio delta for group 3".
		39:37	Package	<b>Group Ratio Delta for Group 3</b> An unsigned integer specifying the ratio decrement relative to the Max ratio limit for Group 2.
		44:40	Package	<b>Number of Incremental Cores Added to Group 4</b> Group 4, which includes the specified number of additional cores plus all the cores in group 3, operates under the group 4 turbo max ratio limit = "group 3 Max ratio limit" - "group ratio delta for group 4".
		47:45	Package	<b>Group Ratio Delta for Group 4</b> An unsigned integer specifying the ratio decrement relative to the Max ratio limit for Group 3.
		52:48	Package	<b>Number of Incremental Cores Added to Group 5</b> Group 5, which includes the specified number of additional cores plus all the cores in group 4, operates under the group 5 turbo max ratio limit = "group 4 Max ratio limit" - "group ratio delta for group 5".
		55:53	Package	<b>Group Ratio Delta for Group 5</b> An unsigned integer specifying the ratio decrement relative to the Max ratio limit for Group 4.
		60:56	Package	<b>Number of Incremental Cores Added to Group 6</b> Group 6, which includes the specified number of additional cores plus all the cores in group 5, operates under the group 6 turbo max ratio limit = "group 5 Max ratio limit" - "group ratio delta for group 6".

**Table 35-40. Selected MSRs Supported by Intel® Xeon Phi™ Processors with DisplayFamily\_DisplayModel Signature 06\_57H**

Address		Register Name	Scope	Bit Description
Hex	Dec			
		63:61	Package	<b>Group Ratio Delta for Group 6</b> An unsigned integer specifying the ratio decrement relative to the Max ratio limit for Group 5.
1B0H	432	IA32_ENERGY_PERF_BIAS	Thread	See Table 35-2.
1B1H	433	IA32_PACKAGE_THERM_STATUS	Package	See Table 35-2.
1B2H	434	IA32_PACKAGE_THERM_INTERRUPT	Package	See Table 35-2.
1C8H	456	MSR_LBR_SELECT	Thread	<b>Last Branch Record Filtering Select Register (R/W)</b>
1C9H	457	MSR_LASTBRANCH_TOS	Thread	<b>Last Branch Record Stack TOS (R/W)</b>
1D9H	473	IA32_DEBUGCTL	Thread	<b>Debug Control (R/W)</b> See Table 35-2.
1DDH	477	MSR_LER_FROM_LIP	Thread	<b>Last Exception Record From Linear IP (R)</b>
1DEH	478	MSR_LER_TO_LIP	Thread	<b>Last Exception Record To Linear IP (R)</b>
1F2H	498	IA32_SMRR_PHYSBASE	Core	See Table 35-2.
1F3H	499	IA32_SMRR_PHYSMASK	Core	See Table 35-2.
200H	512	IA32_MTRR_PHYSBASE0	Core	See Table 35-2.
201H	513	IA32_MTRR_PHYSMASK0	Core	See Table 35-2.
202H	514	IA32_MTRR_PHYSBASE1	Core	See Table 35-2.
203H	515	IA32_MTRR_PHYSMASK1	Core	See Table 35-2.
204H	516	IA32_MTRR_PHYSBASE2	Core	See Table 35-2.
205H	517	IA32_MTRR_PHYSMASK2	Core	See Table 35-2.
206H	518	IA32_MTRR_PHYSBASE3	Core	See Table 35-2.
207H	519	IA32_MTRR_PHYSMASK3	Core	See Table 35-2.
208H	520	IA32_MTRR_PHYSBASE4	Core	See Table 35-2.
209H	521	IA32_MTRR_PHYSMASK4	Core	See Table 35-2.
20AH	522	IA32_MTRR_PHYSBASE5	Core	See Table 35-2.
20BH	523	IA32_MTRR_PHYSMASK5	Core	See Table 35-2.
20CH	524	IA32_MTRR_PHYSBASE6	Core	See Table 35-2.
20DH	525	IA32_MTRR_PHYSMASK6	Core	See Table 35-2.
20EH	526	IA32_MTRR_PHYSBASE7	Core	See Table 35-2.
20FH	527	IA32_MTRR_PHYSMASK7	Core	See Table 35-2.
250H	592	IA32_MTRR_FIX64K_00000	Core	See Table 35-2.
258H	600	IA32_MTRR_FIX16K_80000	Core	See Table 35-2.
259H	601	IA32_MTRR_FIX16K_A0000	Core	See Table 35-2.
268H	616	IA32_MTRR_FIX4K_C0000	Core	See Table 35-2.
269H	617	IA32_MTRR_FIX4K_C8000	Core	See Table 35-2.

**Table 35-40. Selected MSRs Supported by Intel® Xeon Phi™ Processors with DisplayFamily\_DisplayModel Signature 06\_57H**

Address		Register Name	Scope	Bit Description
Hex	Dec			
26AH	618	IA32_MTRR_FIX4K_D0000	Core	See Table 35-2.
26BH	619	IA32_MTRR_FIX4K_D8000	Core	See Table 35-2.
26CH	620	IA32_MTRR_FIX4K_E0000	Core	See Table 35-2.
26DH	621	IA32_MTRR_FIX4K_E8000	Core	See Table 35-2.
26EH	622	IA32_MTRR_FIX4K_F0000	Core	See Table 35-2.
26FH	623	IA32_MTRR_FIX4K_F8000	Core	See Table 35-2.
277H	631	IA32_PAT	Core	See Table 35-2.
2FFH	767	IA32_MTRR_DEF_TYPE	Core	<b>Default Memory Types (R/W)</b> See Table 35-2.
309H	777	IA32_FIXED_CTR0	Thread	<b>Fixed-Function Performance Counter Register 0 (R/W)</b> See Table 35-2.
30AH	778	IA32_FIXED_CTR1	Thread	<b>Fixed-Function Performance Counter Register 1 (R/W)</b> See Table 35-2.
30BH	779	IA32_FIXED_CTR2	Thread	<b>Fixed-Function Performance Counter Register 2 (R/W)</b> See Table 35-2.
345H	837	IA32_PERF_CAPABILITIES	Package	See Table 35-2. See Section 17.4.1, "IA32_DEBUGCTL MSR."
38DH	909	IA32_FIXED_CTR_CTRL	Thread	<b>Fixed-Function-Counter Control Register (R/W)</b> See Table 35-2.
38EH	910	IA32_PERF_GLOBAL_STATU S	Thread	See Table 35-2.
38FH	911	IA32_PERF_GLOBAL_CTRL	Thread	See Table 35-2.
390H	912	IA32_PERF_GLOBAL_OVF_ CTRL	Thread	See Table 35-2.
3F1H	1009	MSR_PEBBS_ENABLE	Thread	See Table 35-2.
3F8H	1016	MSR_PKG_C3_RESIDENCY	Package	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		63:0		Package C3 Residency Counter. (R/O)
3F9H	1017	MSR_PKG_C6_RESIDENCY	Package	
		63:0		Package C6 Residency Counter. (R/O)
3FAH	1018	MSR_PKG_C7_RESIDENCY	Package	
		63:0		Package C7 Residency Counter. (R/O)
3FCH	1020	MSR_MCO_RESIDENCY	Module	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		63:0		Module C0 Residency Counter. (R/O)
3FDH	1021	MSR_MC6_RESIDENCY	Module	
		63:0		Module C6 Residency Counter. (R/O)
3FFH	1023	MSR_CORE_C6_RESIDENCY	Core	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		63:0		CORE C6 Residency Counter. (R/O)

**Table 35-40. Selected MSRs Supported by Intel® Xeon Phi™ Processors with DisplayFamily\_DisplayModel Signature 06\_57H**

Address		Register Name	Scope	Bit Description
Hex	Dec			
400H	1024	IA32_MC0_CTL	Core	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
401H	1025	IA32_MC0_STATUS	Core	See Section 15.3.2.2, "IA32_MCi_STATUS MSRs."
402H	1026	IA32_MC0_ADDR	Core	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
404H	1028	IA32_MC1_CTL	Core	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
405H	1029	IA32_MC1_STATUS	Core	See Section 15.3.2.2, "IA32_MCi_STATUS MSRs."
408H	1032	IA32_MC2_CTL	Core	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
409H	1033	IA32_MC2_STATUS	Core	See Section 15.3.2.2, "IA32_MCi_STATUS MSRs."
40AH	1034	IA32_MC2_ADDR	Core	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
40CH	1036	IA32_MC3_CTL	Core	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
40DH	1037	IA32_MC3_STATUS	Core	See Section 15.3.2.2, "IA32_MCi_STATUS MSRs."
40EH	1038	IA32_MC3_ADDR	Core	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
410H	1040	IA32_MC4_CTL	Core	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
411H	1041	IA32_MC4_STATUS	Core	See Section 15.3.2.2, "IA32_MCi_STATUS MSRs."
412H	1042	IA32_MC4_ADDR	Core	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The MSR_MC4_ADDR register is either not implemented or contains no address if the ADDR_V flag in the MSR_MC4_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
414H	1044	IA32_MC5_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
415H	1045	IA32_MC5_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRs."
416H	1046	IA32_MC5_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
480H	1152	IA32_VMX_BASIC	Core	<b>Reporting Register of Basic VMX Capabilities (R/O)</b> See Table 35-2.
481H	1153	IA32_VMX_PINBASED_CTL	Core	<b>Capability Reporting Register of Pin-based VM-execution Controls (R/O)</b> See Table 35-2.
482H	1154	IA32_VMX_PROCBASED_CTL	Core	<b>Capability Reporting Register of Primary Processor-based VM-execution Controls (R/O)</b>
483H	1155	IA32_VMX_EXIT_CTL	Core	<b>Capability Reporting Register of VM-exit Controls (R/O)</b> See Table 35-2.
484H	1156	IA32_VMX_ENTRY_CTL	Core	<b>Capability Reporting Register of VM-entry Controls (R/O)</b> See Table 35-2.
485H	1157	IA32_VMX_MISC	Core	<b>Reporting Register of Miscellaneous VMX Capabilities (R/O)</b> See Table 35-2.
486H	1158	IA32_VMX_CR0_FIXED0	Core	<b>Capability Reporting Register of CR0 Bits Fixed to 0 (R/O)</b> See Table 35-2.
487H	1159	IA32_VMX_CR0_FIXED1	Core	<b>Capability Reporting Register of CR0 Bits Fixed to 1 (R/O)</b> See Table 35-2.

**Table 35-40. Selected MSRs Supported by Intel® Xeon Phi™ Processors with DisplayFamily\_DisplayModel Signature 06\_57H**

Address		Register Name	Scope	Bit Description
Hex	Dec			
488H	1160	IA32_VMX_CR4_FIXED0	Core	<b>Capability Reporting Register of CR4 Bits Fixed to 0 (R/O)</b> See Table 35-2.
489H	1161	IA32_VMX_CR4_FIXED1	Core	<b>Capability Reporting Register of CR4 Bits Fixed to 1 (R/O)</b> See Table 35-2.
48AH	1162	IA32_VMX_VMCS_ENUM	Core	<b>Capability Reporting Register of VMCS Field Enumeration (R/O)</b> See Table 35-2.
48BH	1163	IA32_VMX_PROCBASED_CTLSS2	Core	<b>Capability Reporting Register of Secondary Processor-based VM-execution Controls (R/O)</b> See Table 35-2
48CH	1164	IA32_VMX_EPT_VPID_ENUM	Core	<b>Capability Reporting Register of EPT and VPID (R/O)</b> See Table 35-2
48DH	1165	IA32_VMX_TRUE_PINBASED_CTLSS	Core	<b>Capability Reporting Register of Pin-based VM-execution Flex Controls (R/O)</b> See Table 35-2
48EH	1166	IA32_VMX_TRUE_PROCBASED_CTLSS	Core	<b>Capability Reporting Register of Primary Processor-based VM-execution Flex Controls (R/O)</b> See Table 35-2
48FH	1167	IA32_VMX_TRUE_EXIT_CTLS	Core	<b>Capability Reporting Register of VM-exit Flex Controls (R/O)</b> See Table 35-2
490H	1168	IA32_VMX_TRUE_ENTRY_CTLS	Core	<b>Capability Reporting Register of VM-entry Flex Controls (R/O)</b> See Table 35-2
491H	1169	IA32_VMX_FMFUNC	Core	<b>Capability Reporting Register of VM-function Controls (R/O)</b> See Table 35-2
4C1H	1217	IA32_A_PMC0	Thread	See Table 35-2.
4C2H	1218	IA32_A_PMC1	Thread	See Table 35-2.
600H	1536	IA32_DS_AREA	Thread	<b>DS Save Area (R/W)</b> See Table 35-2.
606H	1542	MSR_RAPL_POWER_UNIT	Package	<b>Unit Multipliers used in RAPL Interfaces (R/O)</b>
		3:0	Package	<b>Power Units</b> See Section 14.9.1, "RAPL Interfaces."
		7:4	Package	<b>Reserved</b>
		12:8	Package	<b>Energy Status Units</b> Energy related information (in Joules) is based on the multiplier, 1/2 <sup>n</sup> ESU; where ESU is an unsigned integer represented by bits 12:8. Default value is 0EH (or 61 micro-joules)
		15:13	Package	<b>Reserved</b>
		19:16	Package	<b>Time Units</b> See Section 14.9.1, "RAPL Interfaces."
		63:20		Reserved

**Table 35-40. Selected MSRs Supported by Intel® Xeon Phi™ Processors with DisplayFamily\_DisplayModel Signature 06\_57H**

Address		Register Name	Scope	Bit Description
Hex	Dec			
60DH	1549	MSR_PKG_C2_RESIDENCY	Package	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		63:0		Package C2 Residency Counter. (R/O)
610H	1552	MSR_PKG_POWER_LIMIT	Package	<b>PKG RAPL Power Limit Control (R/W)</b> See Section 14.9.3, "Package RAPL Domain."
611H	1553	MSR_PKG_ENERGY_STATUS	Package	<b>PKG Energy Status (R/O)</b> See Section 14.9.3, "Package RAPL Domain."
613H	1555	MSR_PKG_PERF_STATUS	Package	<b>PKG Perf Status (R/O)</b> See Section 14.9.3, "Package RAPL Domain."
614H	1556	MSR_PKG_POWER_INFO	Package	<b>PKG RAPL Parameters (R/W)</b> See Section 14.9.3, "Package RAPL Domain."
618H	1560	MSR_DRAM_POWER_LIMIT	Package	<b>DRAM RAPL Power Limit Control (R/W)</b> See Section 14.9.5, "DRAM RAPL Domain."
619H	1561	MSR_DRAM_ENERGY_STATUS	Package	<b>DRAM Energy Status (R/O)</b> See Section 14.9.5, "DRAM RAPL Domain."
61BH	1563	MSR_DRAM_PERF_STATUS	Package	<b>DRAM Performance Throttling Status (R/O)</b> See Section 14.9.5, "DRAM RAPL Domain."
61CH	1564	MSR_DRAM_POWER_INFO	Package	<b>DRAM RAPL Parameters (R/W)</b> See Section 14.9.5, "DRAM RAPL Domain."
638H	1592	MSR_PPO_POWER_LIMIT	Package	<b>PPO RAPL Power Limit Control (R/W)</b> See Section 14.9.4, "PPO/PP1 RAPL Domains."
639H	1593	MSR_PPO_ENERGY_STATUS	Package	<b>PPO Energy Status (R/O)</b> See Section 14.9.4, "PPO/PP1 RAPL Domains."
648H	1608	MSR_CONFIG_TDP_NOMINAL	Package	<b>Base TDP Ratio (R/O)</b> See Table 35-23
649H	1609	MSR_CONFIG_TDP_LEVEL1	Package	ConfigTDP Level 1 ratio and power level (R/O). See Table 35-23
64AH	1610	MSR_CONFIG_TDP_LEVEL2	Package	ConfigTDP Level 2 ratio and power level (R/O). See Table 35-23
64BH	1611	MSR_CONFIG_TDP_CONTROL	Package	<b>ConfigTDP Control (R/W)</b> See Table 35-23
64CH	1612	MSR_TURBO_ACTIVATION_RATIO	Package	<b>ConfigTDP Control (R/W)</b> See Table 35-23
690H	1680	MSR_CORE_PERF_LIMIT_REASONS	Package	<b>Indicator of Frequency Clipping in Processor Cores (R/W)</b> <b>(frequency refers to processor core frequency)</b>
		0		<b>PROCHOT Status (R0)</b>
		1		<b>Thermal Status (R0)</b>
		5:2		Reserved.
		6		<b>VR Therm Alert Status (R0)</b>
		7		Reserved.
		8		<b>Electrical Design Point Status (R0)</b>

**Table 35-40. Selected MSRs Supported by Intel® Xeon Phi™ Processors with DisplayFamily\_DisplayModel Signature 06\_57H**

Address		Register Name	Scope	Bit Description
Hex	Dec			
		63:9		Reserved.
6E0H	1760	IA32_TSC_DEADLINE	Core	<b>TSC Target of Local APIC's TSC Deadline Mode (R/W)</b> See Table 35-2
802H	2050	IA32_X2APIC_APICID	Thread	x2APIC ID register (R/O) See x2APIC Specification.
803H	2051	IA32_X2APIC_VERSION	Thread	x2APIC Version register (R/O)
808H	2056	IA32_X2APIC_TPR	Thread	x2APIC Task Priority register (R/W)
80AH	2058	IA32_X2APIC_PPR	Thread	x2APIC Processor Priority register (R/O)
80BH	2059	IA32_X2APIC_EOI	Thread	x2APIC EOI register (W/O)
80DH	2061	IA32_X2APIC_LDR	Thread	x2APIC Logical Destination register (R/O)
80FH	2063	IA32_X2APIC_SIVR	Thread	x2APIC Spurious Interrupt Vector register (R/W)
810H	2064	IA32_X2APIC_ISR0	Thread	x2APIC In-Service register bits [31:0] (R/O)
811H	2065	IA32_X2APIC_ISR1	Thread	x2APIC In-Service register bits [63:32] (R/O)
812H	2066	IA32_X2APIC_ISR2	Thread	x2APIC In-Service register bits [95:64] (R/O)
813H	2067	IA32_X2APIC_ISR3	Thread	x2APIC In-Service register bits [127:96] (R/O)
814H	2068	IA32_X2APIC_ISR4	Thread	x2APIC In-Service register bits [159:128] (R/O)
815H	2069	IA32_X2APIC_ISR5	Thread	x2APIC In-Service register bits [191:160] (R/O)
816H	2070	IA32_X2APIC_ISR6	Thread	x2APIC In-Service register bits [223:192] (R/O)
817H	2071	IA32_X2APIC_ISR7	Thread	x2APIC In-Service register bits [255:224] (R/O)
818H	2072	IA32_X2APIC_TMR0	Thread	x2APIC Trigger Mode register bits [31:0] (R/O)
819H	2073	IA32_X2APIC_TMR1	Thread	x2APIC Trigger Mode register bits [63:32] (R/O)
81AH	2074	IA32_X2APIC_TMR2	Thread	x2APIC Trigger Mode register bits [95:64] (R/O)
81BH	2075	IA32_X2APIC_TMR3	Thread	x2APIC Trigger Mode register bits [127:96] (R/O)
81CH	2076	IA32_X2APIC_TMR4	Thread	x2APIC Trigger Mode register bits [159:128] (R/O)
81DH	2077	IA32_X2APIC_TMR5	Thread	x2APIC Trigger Mode register bits [191:160] (R/O)
81EH	2078	IA32_X2APIC_TMR6	Thread	x2APIC Trigger Mode register bits [223:192] (R/O)
81FH	2079	IA32_X2APIC_TMR7	Thread	x2APIC Trigger Mode register bits [255:224] (R/O)
820H	2080	IA32_X2APIC_IRR0	Thread	x2APIC Interrupt Request register bits [31:0] (R/O)
821H	2081	IA32_X2APIC_IRR1	Thread	x2APIC Interrupt Request register bits [63:32] (R/O)
822H	2082	IA32_X2APIC_IRR2	Thread	x2APIC Interrupt Request register bits [95:64] (R/O)
823H	2083	IA32_X2APIC_IRR3	Thread	x2APIC Interrupt Request register bits [127:96] (R/O)
824H	2084	IA32_X2APIC_IRR4	Thread	x2APIC Interrupt Request register bits [159:128] (R/O)
825H	2085	IA32_X2APIC_IRR5	Thread	x2APIC Interrupt Request register bits [191:160] (R/O)
826H	2086	IA32_X2APIC_IRR6	Thread	x2APIC Interrupt Request register bits [223:192] (R/O)
827H	2087	IA32_X2APIC_IRR7	Thread	x2APIC Interrupt Request register bits [255:224] (R/O)
828H	2088	IA32_X2APIC_ESR	Thread	x2APIC Error Status register (R/W)
82FH	2095	IA32_X2APIC_LVT_CMCI	Thread	x2APIC LVT Corrected Machine Check Interrupt register (R/W)
830H	2096	IA32_X2APIC_ICR	Thread	x2APIC Interrupt Command register (R/W)

**Table 35-40. Selected MSRs Supported by Intel® Xeon Phi™ Processors with DisplayFamily\_DisplayModel Signature 06\_57H**

Address		Register Name	Scope	Bit Description
Hex	Dec			
832H	2098	IA32_X2APIC_LVT_TIMER	Thread	x2APIC LVT Timer Interrupt register (R/W)
833H	2099	IA32_X2APIC_LVT_THERMAL	Thread	x2APIC LVT Thermal Sensor Interrupt register (R/W)
834H	2100	IA32_X2APIC_LVT_PMI	Thread	x2APIC LVT Performance Monitor register (R/W)
835H	2101	IA32_X2APIC_LVT_LINT0	Thread	x2APIC LVT LINT0 register (R/W)
836H	2102	IA32_X2APIC_LVT_LINT1	Thread	x2APIC LVT LINT1 register (R/W)
837H	2103	IA32_X2APIC_LVT_ERROR	Thread	x2APIC LVT Error register (R/W)
838H	2104	IA32_X2APIC_INIT_COUNT	Thread	x2APIC Initial Count register (R/W)
839H	2105	IA32_X2APIC_CUR_COUNT	Thread	x2APIC Current Count register (R/O)
83EH	2110	IA32_X2APIC_DIV_CONF	Thread	x2APIC Divide Configuration register (R/W)
83FH	2111	IA32_X2APIC_SELF_IPI	Thread	x2APIC Self IPI register (W/O)
C000_0080H		IA32_EFER	Thread	<b>Extended Feature Enables</b> See Table 35-2.
C000_0081H		IA32_STAR	Thread	<b>System Call Target Address (R/W)</b> See Table 35-2.
C000_0082H		IA32_LSTAR	Thread	<b>IA-32e Mode System Call Target Address (R/W)</b> See Table 35-2.
C000_0084H		IA32_FMASK	Thread	<b>System Call Flag Mask (R/W)</b> See Table 35-2.
C000_0100H		IA32_FS_BASE	Thread	<b>Map of BASE Address of FS (R/W)</b> See Table 35-2.
C000_0101H		IA32_GS_BASE	Thread	<b>Map of BASE Address of GS (R/W)</b> See Table 35-2.
C000_0102H		IA32_KERNEL_GSBASE	Thread	<b>Swap Target of BASE Address of GS (R/W)</b> See Table 35-2.
C000_0103H		IA32_TSC_AUX	Thread	<b>AUXILIARY TSC Signature. (R/W)</b> See Table 35-2

## 35.18 MSRS IN THE PENTIUM® 4 AND INTEL® XEON® PROCESSORS

Table 35-41 lists MSRs (architectural and model-specific) that are defined across processor generations based on Intel NetBurst microarchitecture. The processor can be identified by its CPUID signatures of DisplayFamily encoding of 0FH, see Table 35-1.

- MSRs with an “IA32\_” prefix are designated as “architectural.” This means that the functions of these MSRs and their addresses remain the same for succeeding families of IA-32 processors.
- MSRs with an “MSR\_” prefix are model specific with respect to address functionalities. The column “Model Availability” lists the model encoding value(s) within the Pentium 4 and Intel Xeon processor family at the specified register address. The model encoding value of a processor can be queried using CPUID. See “CPUID—CPU Identification” in Chapter 3 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A*.



**Table 35-41. MSRs in the Pentium® 4 and Intel® Xeon® Processors**

Register Address		Register Name Fields and Flags	Model Avail- ability	Shared/ Unique <sup>7</sup>	Bit Description
Hex	Dec				
0H	0	IA32_P5_MC_ADDR	0, 1, 2, 3, 4, 6	Shared	See Section 35.22, "MSRs in Pentium Processors."
1H	1	IA32_P5_MC_TYPE	0, 1, 2, 3, 4, 6	Shared	See Section 35.22, "MSRs in Pentium Processors."
6H	6	IA32_MONITOR_FILTER_LINE_SIZE	3, 4, 6	Shared	See Section 8.10.5, "Monitor/Mwait Address Range Determination."
10H	16	IA32_TIME_STAMP_COUNTER	0, 1, 2, 3, 4, 6	Unique	<b>Time Stamp Counter</b> See Table 35-2.
					On earlier processors, only the lower 32 bits are writable. On any write to the lower 32 bits, the upper 32 bits are cleared. For processor family 0FH, models 3 and 4: all 64 bits are writable.
17H	23	IA32_PLATFORM_ID	0, 1, 2, 3, 4, 6	Shared	<b>Platform ID (R)</b> See Table 35-2. The operating system can use this MSR to determine "slot" information for the processor and the proper microcode update to load.
1BH	27	IA32_APIC_BASE	0, 1, 2, 3, 4, 6	Unique	<b>APIC Location and Status (R/W)</b> See Table 35-2. See Section 10.4.4, "Local APIC Status and Location."
2AH	42	MSR_EBC_HARD_POWERON	0, 1, 2, 3, 4, 6	Shared	<b>Processor Hard Power-On Configuration (R/W)</b> Enables and disables processor features; <b>(R)</b> indicates current processor configuration.
		0			<b>Output Tri-state Enabled (R)</b> Indicates whether tri-state output is enabled (1) or disabled (0) as set by the strapping of SMI#. The value in this bit is written on the deassertion of RESET#; the bit is set to 1 when the address bus signal is asserted.
		1			<b>Execute BIST (R)</b> Indicates whether the execution of the BIST is enabled (1) or disabled (0) as set by the strapping of INIT#. The value in this bit is written on the deassertion of RESET#; the bit is set to 1 when the address bus signal is asserted.
		2			<b>In Order Queue Depth (R)</b> Indicates whether the in order queue depth for the system bus is 1 (1) or up to 12 (0) as set by the strapping of A7#. The value in this bit is written on the deassertion of RESET#; the bit is set to 1 when the address bus signal is asserted.

Table 35-41. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

Register Address		Register Name Fields and Flags	Model Avail- ability	Shared/ Unique <sup>1</sup>	Bit Description
Hex	Dec				
		3			<b>MCERR# Observation Disabled (R)</b> Indicates whether MCERR# observation is enabled (0) or disabled (1) as determined by the strapping of A9#. The value in this bit is written on the deassertion of RESET#; the bit is set to 1 when the address bus signal is asserted.
		4			<b>BINIT# Observation Enabled (R)</b> Indicates whether BINIT# observation is enabled (0) or disabled (1) as determined by the strapping of A10#. The value in this bit is written on the deassertion of RESET#; the bit is set to 1 when the address bus signal is asserted.
		6:5			<b>APIC Cluster ID (R)</b> Contains the logical APIC cluster ID value as set by the strapping of A12# and A11#. The logical cluster ID value is written into the field on the deassertion of RESET#; the field is set to 1 when the address bus signal is asserted.
		7			<b>Bus Park Disable (R)</b> Indicates whether bus park is enabled (0) or disabled (1) as set by the strapping of A15#. The value in this bit is written on the deassertion of RESET#; the bit is set to 1 when the address bus signal is asserted.
		11:8			Reserved.
		13:12			<b>Agent ID (R)</b> Contains the logical agent ID value as set by the strapping of BR[3:0]. The logical ID value is written into the field on the deassertion of RESET#; the field is set to 1 when the address bus signal is asserted.
		63:14			Reserved.
2BH	43	MSR_EBC_SOFT_POWERON	0, 1, 2, 3, 4, 6	Shared	<b>Processor Soft Power-On Configuration (R/W)</b> Enables and disables processor features.
		0			<b>RCNT/SCNT On Request Encoding Enable (R/W)</b> Controls the driving of RCNT/SCNT on the request encoding. Set to enable (1); clear to disabled (0, default).
		1			<b>Data Error Checking Disable (R/W)</b> Set to disable system data bus parity checking; clear to enable parity checking.
		2			<b>Response Error Checking Disable (R/W)</b> Set to disable (default); clear to enable.
		3			<b>Address/Request Error Checking Disable (R/W)</b> Set to disable (default); clear to enable.

**Table 35-41. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)**

Register Address		Register Name Fields and Flags	Model Avail- ability	Shared/ Unique <sup>7</sup>	Bit Description
Hex	Dec				
		4			<b>Initiator MCERR# Disable (R/W)</b> Set to disable MCERR# driving for initiator bus requests (default); clear to enable.
		5			<b>Internal MCERR# Disable (R/W)</b> Set to disable MCERR# driving for initiator internal errors (default); clear to enable.
		6			<b>BINIT# Driver Disable (R/W)</b> Set to disable BINIT# driver (default); clear to enable driver.
		63:7			Reserved.
2CH	44	MSR_EBC_FREQUENCY_ID	2,3, 4, 6	Shared	<b>Processor Frequency Configuration</b> The bit field layout of this MSR varies according to the MODEL value in the CPUID version information. The following bit field layout applies to Pentium 4 and Xeon Processors with MODEL encoding equal or greater than 2. (R) The field Indicates the current processor frequency configuration.
		15:0			Reserved.
		18:16			<b>Scalable Bus Speed (R/W)</b> Indicates the intended scalable bus speed: <u>Encoding Scalable Bus Speed</u> 000B 100 MHz (Model 2) 000B 266 MHz (Model 3 or 4) 001B 133 MHz 010B 200 MHz 011B 166 MHz 100B 333 MHz (Model 6)
					133.33 MHz should be utilized if performing calculation with System Bus Speed when encoding is 001B. 166.67 MHz should be utilized if performing calculation with System Bus Speed when encoding is 011B.
					266.67 MHz should be utilized if performing calculation with System Bus Speed when encoding is 000B and model encoding = 3 or 4. 333.33 MHz should be utilized if performing calculation with System Bus Speed when encoding is 100B and model encoding = 6. All other values are reserved.
		23:19			Reserved.

Table 35-41. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

Register Address		Register Name Fields and Flags	Model Avail- ability	Shared/ Unique <sup>1</sup>	Bit Description
Hex	Dec				
		31:24			<b>Core Clock Frequency to System Bus Frequency Ratio (R)</b> The processor core clock frequency to system bus frequency ratio observed at the de-assertion of the reset pin.
		63:25			Reserved.
2CH	44	MSR_EBC_FREQUENCY_ID	0, 1	Shared	<b>Processor Frequency Configuration (R)</b> The bit field layout of this MSR varies according to the MODEL value of the CPUID version information. This bit field layout applies to Pentium 4 and Xeon Processors with MODEL encoding less than 2. Indicates current processor frequency configuration.
		20:0			Reserved.
		23:21			<b>Scalable Bus Speed (R/W)</b> Indicates the intended scalable bus speed: <u>Encoding Scalable Bus Speed</u> 000B 100 MHz  All others values reserved.
		63:24			Reserved.
3AH	58	IA32_FEATURE_CONTROL	3, 4, 6	Unique	<b>Control Features in IA-32 Processor (R/W)</b> See Table 35-2 (If CPUID.01H:ECX.[bit 5])
79H	121	IA32_BIOS_UPDT_TRIG	0, 1, 2, 3, 4, 6	Shared	<b>BIOS Update Trigger Register (W)</b> See Table 35-2.
8BH	139	IA32_BIOS_SIGN_ID	0, 1, 2, 3, 4, 6	Unique	<b>BIOS Update Signature ID (R/W)</b> See Table 35-2.
9BH	155	IA32_SMM_MONITOR_CTL	3, 4, 6	Unique	<b>SMM Monitor Configuration (R/W)</b> See Table 35-2.
FEH	254	IA32_MTRRCAP	0, 1, 2, 3, 4, 6	Unique	<b>MTRR Information</b> See Section 11.11.1, "MTRR Feature Identification."
174H	372	IA32_SYSENTER_CS	0, 1, 2, 3, 4, 6	Unique	<b>CS register target for CPL 0 code (R/W)</b> See Table 35-2.  See Section 5.8.7, "Performing Fast Calls to System Procedures with the SYSENTER and SYSEXIT Instructions."
175H	373	IA32_SYSENTER_ESP	0, 1, 2, 3, 4, 6	Unique	<b>Stack pointer for CPL 0 stack (R/W)</b> See Table 35-2.  See Section 5.8.7, "Performing Fast Calls to System Procedures with the SYSENTER and SYSEXIT Instructions."

**Table 35-41. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)**

Register Address		Register Name Fields and Flags	Model Avail- ability	Shared/ Unique <sup>7</sup>	Bit Description
Hex	Dec				
176H	374	IA32_SYSENTER_EIP	0, 1, 2, 3, 4, 6	Unique	<b>CPL 0 code entry point (R/W)</b> See Table 35-2. See Section 5.8.7, "Performing Fast Calls to System Procedures with the SYSENTER and SYSEXIT Instructions."
179H	377	IA32_MCG_CAP	0, 1, 2, 3, 4, 6	Unique	<b>Machine Check Capabilities (R)</b> See Table 35-2. See Section 15.3.1.1, "IA32_MCG_CAP MSR."
17AH	378	IA32_MCG_STATUS	0, 1, 2, 3, 4, 6	Unique	<b>Machine Check Status. (R)</b> See Table 35-2. See Section 15.3.1.2, "IA32_MCG_STATUS MSR."
17BH	379	IA32_MCG_CTL			<b>Machine Check Feature Enable (R/W)</b> See Table 35-2. See Section 15.3.1.3, "IA32_MCG_CTL MSR."
180H	384	MSR_MCG_RAX	0, 1, 2, 3, 4, 6	Unique	<b>Machine Check EAX/RAX Save State</b> See Section 15.3.2.6, "IA32_MCG Extended Machine Check State MSRs."
		63:0			Contains register state at time of machine check error. When in non-64-bit modes at the time of the error, bits 63-32 do not contain valid data.
181H	385	MSR_MCG_RBX	0, 1, 2, 3, 4, 6	Unique	<b>Machine Check EBX/RBX Save State</b> See Section 15.3.2.6, "IA32_MCG Extended Machine Check State MSRs."
		63:0			Contains register state at time of machine check error. When in non-64-bit modes at the time of the error, bits 63-32 do not contain valid data.
182H	386	MSR_MCG_RCX	0, 1, 2, 3, 4, 6	Unique	<b>Machine Check ECX/RCX Save State</b> See Section 15.3.2.6, "IA32_MCG Extended Machine Check State MSRs."
		63:0			Contains register state at time of machine check error. When in non-64-bit modes at the time of the error, bits 63-32 do not contain valid data.
183H	387	MSR_MCG_RDX	0, 1, 2, 3, 4, 6	Unique	<b>Machine Check EDX/RDX Save State</b> See Section 15.3.2.6, "IA32_MCG Extended Machine Check State MSRs."
		63:0			Contains register state at time of machine check error. When in non-64-bit modes at the time of the error, bits 63-32 do not contain valid data.
184H	388	MSR_MCG_RSI	0, 1, 2, 3, 4, 6	Unique	<b>Machine Check ESI/RSI Save State</b> See Section 15.3.2.6, "IA32_MCG Extended Machine Check State MSRs."
		63:0			Contains register state at time of machine check error. When in non-64-bit modes at the time of the error, bits 63-32 do not contain valid data.

Table 35-41. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

Register Address		Register Name Fields and Flags	Model Avail- ability	Shared/ Unique <sup>1</sup>	Bit Description
Hex	Dec				
185H	389	MSR_MCG_RDI	0, 1, 2, 3, 4, 6	Unique	<b>Machine Check EDI/RDI Save State</b> See Section 15.3.2.6, "IA32_MCG Extended Machine Check State MSRs."
		63:0			Contains register state at time of machine check error. When in non-64-bit modes at the time of the error, bits 63-32 do not contain valid data.
186H	390	MSR_MCG_RBP	0, 1, 2, 3, 4, 6	Unique	<b>Machine Check EBP/RBP Save State</b> See Section 15.3.2.6, "IA32_MCG Extended Machine Check State MSRs."
		63:0			Contains register state at time of machine check error. When in non-64-bit modes at the time of the error, bits 63-32 do not contain valid data.
187H	391	MSR_MCG_RSP	0, 1, 2, 3, 4, 6	Unique	<b>Machine Check ESP/RSP Save State</b> See Section 15.3.2.6, "IA32_MCG Extended Machine Check State MSRs."
		63:0			Contains register state at time of machine check error. When in non-64-bit modes at the time of the error, bits 63-32 do not contain valid data.
188H	392	MSR_MCG_RFLAGS	0, 1, 2, 3, 4, 6	Unique	<b>Machine Check EFLAGS/RFLAG Save State</b> See Section 15.3.2.6, "IA32_MCG Extended Machine Check State MSRs."
		63:0			Contains register state at time of machine check error. When in non-64-bit modes at the time of the error, bits 63-32 do not contain valid data.
189H	393	MSR_MCG_RIP	0, 1, 2, 3, 4, 6	Unique	<b>Machine Check EIP/RIP Save State</b> See Section 15.3.2.6, "IA32_MCG Extended Machine Check State MSRs."
		63:0			Contains register state at time of machine check error. When in non-64-bit modes at the time of the error, bits 63-32 do not contain valid data.
18AH	394	MSR_MCG_MISC	0, 1, 2, 3, 4, 6	Unique	<b>Machine Check Miscellaneous</b> See Section 15.3.2.6, "IA32_MCG Extended Machine Check State MSRs."
		0			<b>DS</b> When set, the bit indicates that a page assist or page fault occurred during DS normal operation. The processors response is to shut down. The bit is used as an aid for debugging DS handling code. It is the responsibility of the user (BIOS or operating system) to clear this bit for normal operation.
		63:1			Reserved.
18BH- 18FH	395	MSR_MCG_RESERVED1 - MSR_MCG_RESERVED5			Reserved.

**Table 35-41. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)**

Register Address		Register Name Fields and Flags	Model Avail- ability	Shared/ Unique <sup>7</sup>	Bit Description
Hex	Dec				
190H	400	MSR_MCG_R8	0, 1, 2, 3, 4, 6	Unique	<b>Machine Check R8</b> See Section 15.3.2.6, "IA32_MCG Extended Machine Check State MSRs."
		63:0			Registers R8-15 (and the associated state-save MSRs) exist only in Intel 64 processors. These registers contain valid information only when the processor is operating in 64-bit mode at the time of the error.
191H	401	MSR_MCG_R9	0, 1, 2, 3, 4, 6	Unique	<b>Machine Check R9D/R9</b> See Section 15.3.2.6, "IA32_MCG Extended Machine Check State MSRs."
		63:0			Registers R8-15 (and the associated state-save MSRs) exist only in Intel 64 processors. These registers contain valid information only when the processor is operating in 64-bit mode at the time of the error.
192H	402	MSR_MCG_R10	0, 1, 2, 3, 4, 6	Unique	<b>Machine Check R10</b> See Section 15.3.2.6, "IA32_MCG Extended Machine Check State MSRs."
		63:0			Registers R8-15 (and the associated state-save MSRs) exist only in Intel 64 processors. These registers contain valid information only when the processor is operating in 64-bit mode at the time of the error.
193H	403	MSR_MCG_R11	0, 1, 2, 3, 4, 6	Unique	<b>Machine Check R11</b> See Section 15.3.2.6, "IA32_MCG Extended Machine Check State MSRs."
		63:0			Registers R8-15 (and the associated state-save MSRs) exist only in Intel 64 processors. These registers contain valid information only when the processor is operating in 64-bit mode at the time of the error.
194H	404	MSR_MCG_R12	0, 1, 2, 3, 4, 6	Unique	<b>Machine Check R12</b> See Section 15.3.2.6, "IA32_MCG Extended Machine Check State MSRs."
		63:0			Registers R8-15 (and the associated state-save MSRs) exist only in Intel 64 processors. These registers contain valid information only when the processor is operating in 64-bit mode at the time of the error.
195H	405	MSR_MCG_R13	0, 1, 2, 3, 4, 6	Unique	<b>Machine Check R13</b> See Section 15.3.2.6, "IA32_MCG Extended Machine Check State MSRs."

Table 35-41. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

Register Address		Register Name Fields and Flags	Model Avail- ability	Shared/ Unique <sup>1</sup>	Bit Description
Hex	Dec				
		63:0			Registers R8-15 (and the associated state-save MSRs) exist only in Intel 64 processors. These registers contain valid information only when the processor is operating in 64-bit mode at the time of the error.
196H	406	MSR_MCG_R14	0, 1, 2, 3, 4, 6	Unique	<b>Machine Check R14</b> See Section 15.3.2.6, "IA32_MCG Extended Machine Check State MSRs."
		63:0			Registers R8-15 (and the associated state-save MSRs) exist only in Intel 64 processors. These registers contain valid information only when the processor is operating in 64-bit mode at the time of the error.
197H	407	MSR_MCG_R15	0, 1, 2, 3, 4, 6	Unique	<b>Machine Check R15</b> See Section 15.3.2.6, "IA32_MCG Extended Machine Check State MSRs."
		63:0			Registers R8-15 (and the associated state-save MSRs) exist only in Intel 64 processors. These registers contain valid information only when the processor is operating in 64-bit mode at the time of the error.
198H	408	IA32_PERF_STATUS	3, 4, 6	Unique	See Table 35-2. See Section 14.1, "Enhanced Intel Speedstep® Technology."
199H	409	IA32_PERF_CTL	3, 4, 6	Unique	See Table 35-2. See Section 14.1, "Enhanced Intel Speedstep® Technology."
19AH	410	IA32_CLOCK_MODULATION	0, 1, 2, 3, 4, 6	Unique	<b>Thermal Monitor Control (R/W)</b> See Table 35-2. See Section 14.7.3, "Software Controlled Clock Modulation."
19BH	411	IA32_THERM_INTERRUPT	0, 1, 2, 3, 4, 6	Unique	<b>Thermal Interrupt Control (R/W)</b> See Section 14.7.2, "Thermal Monitor," and see Table 35-2.
19CH	412	IA32_THERM_STATUS	0, 1, 2, 3, 4, 6	Shared	<b>Thermal Monitor Status (R/W)</b> See Section 14.7.2, "Thermal Monitor," and see Table 35-2.
19DH	413	MSR_THERM2_CTL			Thermal Monitor 2 Control.
			3,	Shared	For Family F, Model 3 processors: When read, specifies the value of the target TM2 transition last written. When set, it sets the next target value for TM2 transition.
			4, 6	Shared	For Family F, Model 4 and Model 6 processors: When read, specifies the value of the target TM2 transition last written. Writes may cause #GP exceptions.
1A0H	416	IA32_MISC_ENABLE	0, 1, 2, 3, 4, 6	Shared	<b>Enable Miscellaneous Processor Features (R/W)</b>



**Table 35-41. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)**

Register Address		Register Name Fields and Flags	Model Availability	Shared/Unique <sup>7</sup>	Bit Description
Hex	Dec				
		0			Fast-Strings Enable. See Table 35-2.
		1			Reserved.
		2			<b>x87 FPU Fopcode Compatibility Mode Enable</b>
		3			<b>Thermal Monitor 1 Enable</b> See Section 14.7.2, "Thermal Monitor," and see Table 35-2.
		4			<b>Split-Lock Disable</b> When set, the bit causes an #AC exception to be issued instead of a split-lock cycle. Operating systems that set this bit must align system structures to avoid split-lock scenarios. When the bit is clear (default), normal split-locks are issued to the bus.
					This debug feature is specific to the Pentium 4 processor.
		5			Reserved.
		6			<b>Third-Level Cache Disable (R/W)</b> When set, the third-level cache is disabled; when clear (default) the third-level cache is enabled. This flag is reserved for processors that do not have a third-level cache.  Note that the bit controls only the third-level cache; and only if overall caching is enabled through the CD flag of control register CRO, the page-level cache controls, and/or the MTRRs. See Section 11.5.4, "Disabling and Enabling the L3 Cache."
		7			<b>Performance Monitoring Available (R)</b> See Table 35-2.
		8			<b>Suppress Lock Enable</b> When set, assertion of LOCK on the bus is suppressed during a Split Lock access. When clear (default), LOCK is not suppressed.
		9			<b>Prefetch Queue Disable</b> When set, disables the prefetch queue. When clear (default), enables the prefetch queue.
		10			<b>FERR# Interrupt Reporting Enable (R/W)</b> When set, interrupt reporting through the FERR# pin is enabled; when clear, this interrupt reporting function is disabled.

Table 35-41. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

Register Address		Register Name Fields and Flags	Model Availability	Shared/Unique <sup>7</sup>	Bit Description
Hex	Dec				
					<p>When this flag is set and the processor is in the stop-clock state (STPCLK# is asserted), asserting the FERR# pin signals to the processor that an interrupt (such as, INIT#, BINIT#, INTR, NMI, SMI#, or RESET#) is pending and that the processor should return to normal operation to handle the interrupt.</p> <p>This flag does not affect the normal operation of the FERR# pin (to indicate an unmasked floating-point error) when the STPCLK# pin is not asserted.</p>
		11			<p><b>Branch Trace Storage Unavailable (BTS_UNAVAILABLE) (R)</b> See Table 35-2. When set, the processor does not support branch trace storage (BTS); when clear, BTS is supported.</p>
		12			<p><b>PEBS_UNAVAILABLE: Processor Event Based Sampling Unavailable (R)</b> See Table 35-2. When set, the processor does not support processor event-based sampling (PEBS); when clear, PEBS is supported.</p>
		13	3		<p><b>TM2 Enable (R/W)</b> When this bit is set (1) and the thermal sensor indicates that the die temperature is at the pre-determined threshold, the Thermal Monitor 2 mechanism is engaged. TM2 will reduce the bus to core ratio and voltage according to the value last written to MSR_THERM2_CTL bits 15:0. When this bit is clear (0, default), the processor does not change the VID signals or the bus to core ratio when the processor enters a thermal managed state. If the TM2 feature flag (ECX[8]) is not set to 1 after executing CPUID with EAX = 1, then this feature is not supported and BIOS must not alter the contents of this bit location. The processor is operating out of spec if both this bit and the TM1 bit are set to disabled states.</p>
		17:14			Reserved.
		18	3, 4, 6		<p><b>ENABLE MONITOR FSM (R/W)</b> See Table 35-2.</p>
		19			<p><b>Adjacent Cache Line Prefetch Disable (R/W)</b> When set to 1, the processor fetches the cache line of the 128-byte sector containing currently required data. When set to 0, the processor fetches both cache lines in the sector.</p>

**Table 35-41. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)**

Register Address		Register Name Fields and Flags	Model Avail- ability	Shared/ Unique <sup>7</sup>	Bit Description
Hex	Dec				
					Single processor platforms should not set this bit. Server platforms should set or clear this bit based on platform performance observed in validation and testing. BIOS may contain a setup option that controls the setting of this bit.
		21:20			Reserved.
		22	3, 4, 6		<b>Limit CPUID MAXVAL (R/W)</b> See Table 35-2. Setting this can cause unexpected behavior to software that depends on the availability of CPUID leaves greater than 3.
		23		Shared	<b>xTPR Message Disable (R/W)</b> See Table 35-2.
		24			<b>L1 Data Cache Context Mode (R/W)</b> When set, the L1 data cache is placed in shared mode; when clear (default), the cache is placed in adaptive mode. This bit is only enabled for IA-32 processors that support Intel Hyper-Threading Technology. See Section 11.5.6, "L1 Data Cache Context Mode."  When L1 is running in adaptive mode and CR3s are identical, data in L1 is shared across logical processors. Otherwise, L1 is not shared and cache use is competitive.  If the Context ID feature flag (ECX[10]) is set to 0 after executing CPUID with EAX = 1, the ability to switch modes is not supported. BIOS must not alter the contents of IA32_MISC_ENABLE[24].
		33:25			Reserved.
		34		Unique	<b>XD Bit Disable (R/W)</b> See Table 35-2.
		63:35			Reserved.
		1A1H	417	MSR_PLATFORM_BRV	3, 4, 6
17:0					Reserved.
18					<b>PLATFORM Requirements</b> When set to 1, indicates the processor has specific platform requirements. The details of the platform requirements are listed in the respective data sheets of the processor.
63:19					Reserved.

Table 35-41. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

Register Address		Register Name Fields and Flags	Model Avail- ability	Shared/ Unique <sup>1</sup>	Bit Description
Hex	Dec				
1D7H	471	MSR_LER_FROM_LIP	0, 1, 2, 3, 4, 6	Unique	<b>Last Exception Record From Linear IP (R)</b> Contains a pointer to the last branch instruction that the processor executed prior to the last exception that was generated or the last interrupt that was handled. See Section 17.11.3, "Last Exception Records."
		31:0			<b>From Linear IP</b> Linear address of the last branch instruction.
		63:32			Reserved.
1D7H	471	63:0		Unique	<b>From Linear IP</b> Linear address of the last branch instruction (If IA-32e mode is active).
1D8H	472	MSR_LER_TO_LIP	0, 1, 2, 3, 4, 6	Unique	<b>Last Exception Record To Linear IP (R)</b> This area contains a pointer to the target of the last branch instruction that the processor executed prior to the last exception that was generated or the last interrupt that was handled. See Section 17.11.3, "Last Exception Records."
		31:0			<b>From Linear IP</b> Linear address of the target of the last branch instruction.
		63:32			Reserved.
1D8H	472	63:0		Unique	<b>From Linear IP</b> Linear address of the target of the last branch instruction (If IA-32e mode is active).
1D9H	473	MSR_DEBUGCTLA	0, 1, 2, 3, 4, 6	Unique	<b>Debug Control (R/W)</b> Controls how several debug features are used. Bit definitions are discussed in the referenced section. See Section 17.11.1, "MSR_DEBUGCTLA MSR."
1DAH	474	MSR_LASTBRANCH_TOS	0, 1, 2, 3, 4, 6	Unique	<b>Last Branch Record Stack TOS (R/W)</b> Contains an index (0-3 or 0-15) that points to the top of the last branch record stack (that is, that points the index of the MSR containing the most recent branch record). See Section 17.11.2, "LBR Stack for Processors Based on Intel NetBurst® Microarchitecture"; and addresses 1DBH-1DEH and 680H-68FH.

**Table 35-41. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)**

Register Address		Register Name Fields and Flags	Model Avail- ability	Shared/ Unique <sup>7</sup>	Bit Description
Hex	Dec				
1DBH	475	MSR_LASTBRANCH_0	0, 1, 2	Unique	<p><b>Last Branch Record 0 (R/W)</b> One of four last branch record registers on the last branch record stack. It contains pointers to the source and destination instruction for one of the last four branches, exceptions, or interrupts that the processor took.</p> <p>MSR_LASTBRANCH_0 through MSR_LASTBRANCH_3 at 1DBH-1DEH are available only on family 0FH, models 0H-02H. They have been replaced by the MSRs at 680H-68FH and 6C0H-6CFH.</p>
					See Section 17.10, "Last Branch, Call Stack, Interrupt, and Exception Recording for Processors based on Skylake Microarchitecture."
1DDH	477	MSR_LASTBRANCH_2	0, 1, 2	Unique	<p><b>Last Branch Record 2</b> See description of the MSR_LASTBRANCH_0 MSR at 1DBH.</p>
1DEH	478	MSR_LASTBRANCH_3	0, 1, 2	Unique	<p><b>Last Branch Record 3</b> See description of the MSR_LASTBRANCH_0 MSR at 1DBH.</p>
200H	512	IA32_MTRR_PHYSBASE0	0, 1, 2, 3, 4, 6	Shared	<p><b>Variable Range Base MTRR</b> See Section 11.11.2.3, "Variable Range MTRRs."</p>
201H	513	IA32_MTRR_PHYSMASK0	0, 1, 2, 3, 4, 6	Shared	<p><b>Variable Range Mask MTRR</b> See Section 11.11.2.3, "Variable Range MTRRs."</p>
202H	514	IA32_MTRR_PHYSBASE1	0, 1, 2, 3, 4, 6	Shared	<p><b>Variable Range Mask MTRR</b> See Section 11.11.2.3, "Variable Range MTRRs."</p>
203H	515	IA32_MTRR_PHYSMASK1	0, 1, 2, 3, 4, 6	Shared	<p><b>Variable Range Mask MTRR</b> See Section 11.11.2.3, "Variable Range MTRRs."</p>
204H	516	IA32_MTRR_PHYSBASE2	0, 1, 2, 3, 4, 6	Shared	<p><b>Variable Range Mask MTRR</b> See Section 11.11.2.3, "Variable Range MTRRs."</p>
205H	517	IA32_MTRR_PHYSMASK2	0, 1, 2, 3, 4, 6	Shared	<p><b>Variable Range Mask MTRR</b> See Section 11.11.2.3, "Variable Range MTRRs."</p>
206H	518	IA32_MTRR_PHYSBASE3	0, 1, 2, 3, 4, 6	Shared	<p><b>Variable Range Mask MTRR</b> See Section 11.11.2.3, "Variable Range MTRRs."</p>
207H	519	IA32_MTRR_PHYSMASK3	0, 1, 2, 3, 4, 6	Shared	<p><b>Variable Range Mask MTRR</b> See Section 11.11.2.3, "Variable Range MTRRs."</p>
208H	520	IA32_MTRR_PHYSBASE4	0, 1, 2, 3, 4, 6	Shared	<p><b>Variable Range Mask MTRR</b> See Section 11.11.2.3, "Variable Range MTRRs."</p>
209H	521	IA32_MTRR_PHYSMASK4	0, 1, 2, 3, 4, 6	Shared	<p><b>Variable Range Mask MTRR</b> See Section 11.11.2.3, "Variable Range MTRRs."</p>
20AH	522	IA32_MTRR_PHYSBASE5	0, 1, 2, 3, 4, 6	Shared	<p><b>Variable Range Mask MTRR</b> See Section 11.11.2.3, "Variable Range MTRRs."</p>

Table 35-41. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

Register Address		Register Name Fields and Flags	Model Avail- ability	Shared/ Unique <sup>1</sup>	Bit Description
Hex	Dec				
20BH	523	IA32_MTRR_PHYSMASK5	0, 1, 2, 3, 4, 6	Shared	<b>Variable Range Mask MTRR</b> See Section 11.11.2.3, "Variable Range MTRRs."
20CH	524	IA32_MTRR_PHYSBASE6	0, 1, 2, 3, 4, 6	Shared	<b>Variable Range Mask MTRR</b> See Section 11.11.2.3, "Variable Range MTRRs."
20DH	525	IA32_MTRR_PHYSMASK6	0, 1, 2, 3, 4, 6	Shared	<b>Variable Range Mask MTRR</b> See Section 11.11.2.3, "Variable Range MTRRs."
20EH	526	IA32_MTRR_PHYSBASE7	0, 1, 2, 3, 4, 6	Shared	<b>Variable Range Mask MTRR</b> See Section 11.11.2.3, "Variable Range MTRRs."
20FH	527	IA32_MTRR_PHYSMASK7	0, 1, 2, 3, 4, 6	Shared	<b>Variable Range Mask MTRR</b> See Section 11.11.2.3, "Variable Range MTRRs."
250H	592	IA32_MTRR_FIX64K_00000	0, 1, 2, 3, 4, 6	Shared	<b>Fixed Range MTRR</b> See Section 11.11.2.2, "Fixed Range MTRRs."
258H	600	IA32_MTRR_FIX16K_80000	0, 1, 2, 3, 4, 6	Shared	<b>Fixed Range MTRR</b> See Section 11.11.2.2, "Fixed Range MTRRs."
259H	601	IA32_MTRR_FIX16K_A0000	0, 1, 2, 3, 4, 6	Shared	<b>Fixed Range MTRR</b> See Section 11.11.2.2, "Fixed Range MTRRs."
268H	616	IA32_MTRR_FIX4K_C0000	0, 1, 2, 3, 4, 6	Shared	<b>Fixed Range MTRR</b> See Section 11.11.2.2, "Fixed Range MTRRs."
269H	617	IA32_MTRR_FIX4K_C8000	0, 1, 2, 3, 4, 6	Shared	<b>Fixed Range MTRR</b> See Section 11.11.2.2, "Fixed Range MTRRs."
26AH	618	IA32_MTRR_FIX4K_D0000	0, 1, 2, 3, 4, 6	Shared	<b>Fixed Range MTRR</b> See Section 11.11.2.2, "Fixed Range MTRRs."
26BH	619	IA32_MTRR_FIX4K_D8000	0, 1, 2, 3, 4, 6	Shared	<b>Fixed Range MTRR</b> See Section 11.11.2.2, "Fixed Range MTRRs."
26CH	620	IA32_MTRR_FIX4K_E0000	0, 1, 2, 3, 4, 6	Shared	<b>Fixed Range MTRR</b> See Section 11.11.2.2, "Fixed Range MTRRs."
26DH	621	IA32_MTRR_FIX4K_E8000	0, 1, 2, 3, 4, 6	Shared	<b>Fixed Range MTRR</b> See Section 11.11.2.2, "Fixed Range MTRRs."
26EH	622	IA32_MTRR_FIX4K_F0000	0, 1, 2, 3, 4, 6	Shared	<b>Fixed Range MTRR</b> See Section 11.11.2.2, "Fixed Range MTRRs."
26FH	623	IA32_MTRR_FIX4K_F8000	0, 1, 2, 3, 4, 6	Shared	<b>Fixed Range MTRR</b> See Section 11.11.2.2, "Fixed Range MTRRs."
277H	631	IA32_PAT	0, 1, 2, 3, 4, 6	Unique	<b>Page Attribute Table</b> See Section 11.11.2.2, "Fixed Range MTRRs."
2FFH	767	IA32_MTRR_DEF_TYPE	0, 1, 2, 3, 4, 6	Shared	<b>Default Memory Types (R/W)</b> See Table 35-2. See Section 11.11.2.1, "IA32_MTRR_DEF_TYPE MSR."
300H	768	MSR_BPU_COUNTER0	0, 1, 2, 3, 4, 6	Shared	See Section 18.12.2, "Performance Counters."

Table 35-41. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

Register Address		Register Name Fields and Flags	Model Avail- ability	Shared/ Unique <sup>7</sup>	Bit Description
Hex	Dec				
301H	769	MSR_BPU_COUNTER1	0, 1, 2, 3, 4, 6	Shared	See Section 18.12.2, "Performance Counters."
302H	770	MSR_BPU_COUNTER2	0, 1, 2, 3, 4, 6	Shared	See Section 18.12.2, "Performance Counters."
303H	771	MSR_BPU_COUNTER3	0, 1, 2, 3, 4, 6	Shared	See Section 18.12.2, "Performance Counters."
304H	772	MSR_MS_COUNTER0	0, 1, 2, 3, 4, 6	Shared	See Section 18.12.2, "Performance Counters."
305H	773	MSR_MS_COUNTER1	0, 1, 2, 3, 4, 6	Shared	See Section 18.12.2, "Performance Counters."
306H	774	MSR_MS_COUNTER2	0, 1, 2, 3, 4, 6	Shared	See Section 18.12.2, "Performance Counters."
307H	775	MSR_MS_COUNTER3	0, 1, 2, 3, 4, 6	Shared	See Section 18.12.2, "Performance Counters."
308H	776	MSR_FLAME_COUNTER0	0, 1, 2, 3, 4, 6	Shared	See Section 18.12.2, "Performance Counters."
309H	777	MSR_FLAME_COUNTER1	0, 1, 2, 3, 4, 6	Shared	See Section 18.12.2, "Performance Counters."
30AH	778	MSR_FLAME_COUNTER2	0, 1, 2, 3, 4, 6	Shared	See Section 18.12.2, "Performance Counters."
30BH	779	MSR_FLAME_COUNTER3	0, 1, 2, 3, 4, 6	Shared	See Section 18.12.2, "Performance Counters."
30CH	780	MSR_IQ_COUNTER0	0, 1, 2, 3, 4, 6	Shared	See Section 18.12.2, "Performance Counters."
30DH	781	MSR_IQ_COUNTER1	0, 1, 2, 3, 4, 6	Shared	See Section 18.12.2, "Performance Counters."
30EH	782	MSR_IQ_COUNTER2	0, 1, 2, 3, 4, 6	Shared	See Section 18.12.2, "Performance Counters."
30FH	783	MSR_IQ_COUNTER3	0, 1, 2, 3, 4, 6	Shared	See Section 18.12.2, "Performance Counters."
310H	784	MSR_IQ_COUNTER4	0, 1, 2, 3, 4, 6	Shared	See Section 18.12.2, "Performance Counters."
311H	785	MSR_IQ_COUNTER5	0, 1, 2, 3, 4, 6	Shared	See Section 18.12.2, "Performance Counters."
360H	864	MSR_BPU_CCCR0	0, 1, 2, 3, 4, 6	Shared	See Section 18.12.3, "CCCR MSRs."
361H	865	MSR_BPU_CCCR1	0, 1, 2, 3, 4, 6	Shared	See Section 18.12.3, "CCCR MSRs."
362H	866	MSR_BPU_CCCR2	0, 1, 2, 3, 4, 6	Shared	See Section 18.12.3, "CCCR MSRs."
363H	867	MSR_BPU_CCCR3	0, 1, 2, 3, 4, 6	Shared	See Section 18.12.3, "CCCR MSRs."
364H	868	MSR_MS_CCCR0	0, 1, 2, 3, 4, 6	Shared	See Section 18.12.3, "CCCR MSRs."

Table 35-41. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

Register Address		Register Name Fields and Flags	Model Avail- ability	Shared/ Unique <sup>1</sup>	Bit Description
Hex	Dec				
365H	869	MSR_MS_CCCR1	0, 1, 2, 3, 4, 6	Shared	See Section 18.12.3, "CCCR MSRs."
366H	870	MSR_MS_CCCR2	0, 1, 2, 3, 4, 6	Shared	See Section 18.12.3, "CCCR MSRs."
367H	871	MSR_MS_CCCR3	0, 1, 2, 3, 4, 6	Shared	See Section 18.12.3, "CCCR MSRs."
368H	872	MSR_FLAME_CCCRO	0, 1, 2, 3, 4, 6	Shared	See Section 18.12.3, "CCCR MSRs."
369H	873	MSR_FLAME_CCCR1	0, 1, 2, 3, 4, 6	Shared	See Section 18.12.3, "CCCR MSRs."
36AH	874	MSR_FLAME_CCCR2	0, 1, 2, 3, 4, 6	Shared	See Section 18.12.3, "CCCR MSRs."
36BH	875	MSR_FLAME_CCCR3	0, 1, 2, 3, 4, 6	Shared	See Section 18.12.3, "CCCR MSRs."
36CH	876	MSR_IQ_CCCRO	0, 1, 2, 3, 4, 6	Shared	See Section 18.12.3, "CCCR MSRs."
36DH	877	MSR_IQ_CCCR1	0, 1, 2, 3, 4, 6	Shared	See Section 18.12.3, "CCCR MSRs."
36EH	878	MSR_IQ_CCCR2	0, 1, 2, 3, 4, 6	Shared	See Section 18.12.3, "CCCR MSRs."
36FH	879	MSR_IQ_CCCR3	0, 1, 2, 3, 4, 6	Shared	See Section 18.12.3, "CCCR MSRs."
370H	880	MSR_IQ_CCCR4	0, 1, 2, 3, 4, 6	Shared	See Section 18.12.3, "CCCR MSRs."
371H	881	MSR_IQ_CCCR5	0, 1, 2, 3, 4, 6	Shared	See Section 18.12.3, "CCCR MSRs."
3A0H	928	MSR_BSU_ESCRO	0, 1, 2, 3, 4, 6	Shared	See Section 18.12.1, "ESCR MSRs."
3A1H	929	MSR_BSU_ESCR1	0, 1, 2, 3, 4, 6	Shared	See Section 18.12.1, "ESCR MSRs."
3A2H	930	MSR_FSB_ESCRO	0, 1, 2, 3, 4, 6	Shared	See Section 18.12.1, "ESCR MSRs."
3A3H	931	MSR_FSB_ESCR1	0, 1, 2, 3, 4, 6	Shared	See Section 18.12.1, "ESCR MSRs."
3A4H	932	MSR_FIRM_ESCRO	0, 1, 2, 3, 4, 6	Shared	See Section 18.12.1, "ESCR MSRs."
3A5H	933	MSR_FIRM_ESCR1	0, 1, 2, 3, 4, 6	Shared	See Section 18.12.1, "ESCR MSRs."
3A6H	934	MSR_FLAME_ESCRO	0, 1, 2, 3, 4, 6	Shared	See Section 18.12.1, "ESCR MSRs."
3A7H	935	MSR_FLAME_ESCR1	0, 1, 2, 3, 4, 6	Shared	See Section 18.12.1, "ESCR MSRs."
3A8H	936	MSR_DAC_ESCRO	0, 1, 2, 3, 4, 6	Shared	See Section 18.12.1, "ESCR MSRs."



Table 35-41. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

Register Address		Register Name Fields and Flags	Model Avail- ability	Shared/ Unique <sup>7</sup>	Bit Description
Hex	Dec				
3A9H	937	MSR_DAC_ESCR1	0, 1, 2, 3, 4, 6	Shared	See Section 18.12.1, "ESCR MSRs."
3AAH	938	MSR_MOB_ESCR0	0, 1, 2, 3, 4, 6	Shared	See Section 18.12.1, "ESCR MSRs."
3ABH	939	MSR_MOB_ESCR1	0, 1, 2, 3, 4, 6	Shared	See Section 18.12.1, "ESCR MSRs."
3ACH	940	MSR_PMH_ESCR0	0, 1, 2, 3, 4, 6	Shared	See Section 18.12.1, "ESCR MSRs."
3ADH	941	MSR_PMH_ESCR1	0, 1, 2, 3, 4, 6	Shared	See Section 18.12.1, "ESCR MSRs."
3AEH	942	MSR_SAA_T_ESCR0	0, 1, 2, 3, 4, 6	Shared	See Section 18.12.1, "ESCR MSRs."
3AFH	943	MSR_SAA_T_ESCR1	0, 1, 2, 3, 4, 6	Shared	See Section 18.12.1, "ESCR MSRs."
3B0H	944	MSR_U2L_ESCR0	0, 1, 2, 3, 4, 6	Shared	See Section 18.12.1, "ESCR MSRs."
3B1H	945	MSR_U2L_ESCR1	0, 1, 2, 3, 4, 6	Shared	See Section 18.12.1, "ESCR MSRs."
3B2H	946	MSR_BPU_ESCR0	0, 1, 2, 3, 4, 6	Shared	See Section 18.12.1, "ESCR MSRs."
3B3H	947	MSR_BPU_ESCR1	0, 1, 2, 3, 4, 6	Shared	See Section 18.12.1, "ESCR MSRs."
3B4H	948	MSR_IS_ESCR0	0, 1, 2, 3, 4, 6	Shared	See Section 18.12.1, "ESCR MSRs."
3B5H	949	MSR_IS_ESCR1	0, 1, 2, 3, 4, 6	Shared	See Section 18.12.1, "ESCR MSRs."
3B6H	950	MSR_ITLB_ESCR0	0, 1, 2, 3, 4, 6	Shared	See Section 18.12.1, "ESCR MSRs."
3B7H	951	MSR_ITLB_ESCR1	0, 1, 2, 3, 4, 6	Shared	See Section 18.12.1, "ESCR MSRs."
3B8H	952	MSR_CRU_ESCR0	0, 1, 2, 3, 4, 6	Shared	See Section 18.12.1, "ESCR MSRs."
3B9H	953	MSR_CRU_ESCR1	0, 1, 2, 3, 4, 6	Shared	See Section 18.12.1, "ESCR MSRs."
3BAH	954	MSR_IQ_ESCR0	0, 1, 2	Shared	See Section 18.12.1, "ESCR MSRs." This MSR is not available on later processors. It is only available on processor family OFH, models 01H-02H.
3BBH	955	MSR_IQ_ESCR1	0, 1, 2	Shared	See Section 18.12.1, "ESCR MSRs." This MSR is not available on later processors. It is only available on processor family OFH, models 01H-02H.
3BCH	956	MSR_RAT_ESCR0	0, 1, 2, 3, 4, 6	Shared	See Section 18.12.1, "ESCR MSRs."

Table 35-41. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

Register Address		Register Name Fields and Flags	Model Avail- ability	Shared/ Unique <sup>1</sup>	Bit Description
Hex	Dec				
3BDH	957	MSR_RAT_ESCR1	0, 1, 2, 3, 4, 6	Shared	See Section 18.12.1, "ESCR MSRs."
3BEH	958	MSR_SSU_ESCR0	0, 1, 2, 3, 4, 6	Shared	See Section 18.12.1, "ESCR MSRs."
3C0H	960	MSR_MS_ESCR0	0, 1, 2, 3, 4, 6	Shared	See Section 18.12.1, "ESCR MSRs."
3C1H	961	MSR_MS_ESCR1	0, 1, 2, 3, 4, 6	Shared	See Section 18.12.1, "ESCR MSRs."
3C2H	962	MSR_TBPU_ESCR0	0, 1, 2, 3, 4, 6	Shared	See Section 18.12.1, "ESCR MSRs."
3C3H	963	MSR_TBPU_ESCR1	0, 1, 2, 3, 4, 6	Shared	See Section 18.12.1, "ESCR MSRs."
3C4H	964	MSR_TC_ESCR0	0, 1, 2, 3, 4, 6	Shared	See Section 18.12.1, "ESCR MSRs."
3C5H	965	MSR_TC_ESCR1	0, 1, 2, 3, 4, 6	Shared	See Section 18.12.1, "ESCR MSRs."
3C8H	968	MSR_IX_ESCR0	0, 1, 2, 3, 4, 6	Shared	See Section 18.12.1, "ESCR MSRs."
3C9H	969	MSR_IX_ESCR1	0, 1, 2, 3, 4, 6	Shared	See Section 18.12.1, "ESCR MSRs."
3CAH	970	MSR_ALF_ESCR0	0, 1, 2, 3, 4, 6	Shared	See Section 18.12.1, "ESCR MSRs."
3CBH	971	MSR_ALF_ESCR1	0, 1, 2, 3, 4, 6	Shared	See Section 18.12.1, "ESCR MSRs."
3CCH	972	MSR_CRU_ESCR2	0, 1, 2, 3, 4, 6	Shared	See Section 18.12.1, "ESCR MSRs."
3CDH	973	MSR_CRU_ESCR3	0, 1, 2, 3, 4, 6	Shared	See Section 18.12.1, "ESCR MSRs."
3E0H	992	MSR_CRU_ESCR4	0, 1, 2, 3, 4, 6	Shared	See Section 18.12.1, "ESCR MSRs."
3E1H	993	MSR_CRU_ESCR5	0, 1, 2, 3, 4, 6	Shared	See Section 18.12.1, "ESCR MSRs."
3F0H	1008	MSR_TC_PRECISE_EVENT	0, 1, 2, 3, 4, 6	Shared	See Section 18.12.1, "ESCR MSRs."
3F1H	1009	MSR_PEBS_ENABLE	0, 1, 2, 3, 4, 6	Shared	<b>Processor Event Based Sampling (PEBS) (R/W)</b> Controls the enabling of processor event sampling and replay tagging.
		12:0			See Table 19-26.
		23:13			Reserved.
		24			<b>UOP Tag</b> Enables replay tagging when set.

**Table 35-41. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)**

Register Address		Register Name Fields and Flags	Model Avail- ability	Shared/ Unique <sup>7</sup>	Bit Description
Hex	Dec				
		25			<b>ENABLE_PEBS_MY_THR (R/W)</b> Enables PEBS for the target logical processor when set; disables PEBS when clear (default). See Section 18.13.3, "IA32_PEBS_ENABLE MSR," for an explanation of the target logical processor. This bit is called ENABLE_PEBS in IA-32 processors that do not support Intel Hyper-Threading Technology.
		26			<b>ENABLE_PEBS_OTH_THR (R/W)</b> Enables PEBS for the target logical processor when set; disables PEBS when clear (default). See Section 18.13.3, "IA32_PEBS_ENABLE MSR," for an explanation of the target logical processor. This bit is reserved for IA-32 processors that do not support Intel Hyper-Threading Technology.
		63:27			Reserved.
3F2H	1010	MSR_PEBS_MATRIX_VERT	0, 1, 2, 3, 4, 6	Shared	See Table 19-26.
400H	1024	IA32_MCO_CTL	0, 1, 2, 3, 4, 6	Shared	See Section 15.3.2.1, "IA32_MCI_CTL MSRs."
401H	1025	IA32_MCO_STATUS	0, 1, 2, 3, 4, 6	Shared	See Section 15.3.2.2, "IA32_MCI_STATUS MSRs."
402H	1026	IA32_MCO_ADDR	0, 1, 2, 3, 4, 6	Shared	See Section 15.3.2.3, "IA32_MCI_ADDR MSRs." The IA32_MCO_ADDR register is either not implemented or contains no address if the ADDR_V flag in the IA32_MCO_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
403H	1027	IA32_MCO_MISC	0, 1, 2, 3, 4, 6	Shared	See Section 15.3.2.4, "IA32_MCI_MISC MSRs." The IA32_MCO_MISC MSR is either not implemented or does not contain additional information if the MISCV flag in the IA32_MCO_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
404H	1028	IA32_MC1_CTL	0, 1, 2, 3, 4, 6	Shared	See Section 15.3.2.1, "IA32_MCI_CTL MSRs."
405H	1029	IA32_MC1_STATUS	0, 1, 2, 3, 4, 6	Shared	See Section 15.3.2.2, "IA32_MCI_STATUS MSRs."

Table 35-41. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

Register Address		Register Name Fields and Flags	Model Avail- ability	Shared/ Unique <sup>1</sup>	Bit Description
Hex	Dec				
406H	1030	IA32_MC1_ADDR	0, 1, 2, 3, 4, 6	Shared	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The IA32_MC1_ADDR register is either not implemented or contains no address if the ADDR_V flag in the IA32_MC1_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
407H	1031	IA32_MC1_MISC		Shared	See Section 15.3.2.4, "IA32_MCi_MISC MSRs." The IA32_MC1_MISC MSR is either not implemented or does not contain additional information if the MISC_V flag in the IA32_MC1_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
408H	1032	IA32_MC2_CTL	0, 1, 2, 3, 4, 6	Shared	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
409H	1033	IA32_MC2_STATUS	0, 1, 2, 3, 4, 6	Shared	See Section 15.3.2.2, "IA32_MCi_STATUS MSRs."
40AH	1034	IA32_MC2_ADDR			See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The IA32_MC2_ADDR register is either not implemented or contains no address if the ADDR_V flag in the IA32_MC2_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
40BH	1035	IA32_MC2_MISC			See Section 15.3.2.4, "IA32_MCi_MISC MSRs." The IA32_MC2_MISC MSR is either not implemented or does not contain additional information if the MISC_V flag in the IA32_MC2_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
40CH	1036	IA32_MC3_CTL	0, 1, 2, 3, 4, 6	Shared	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
40DH	1037	IA32_MC3_STATUS	0, 1, 2, 3, 4, 6	Shared	See Section 15.3.2.2, "IA32_MCi_STATUS MSRs."
40EH	1038	IA32_MC3_ADDR	0, 1, 2, 3, 4, 6	Shared	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The IA32_MC3_ADDR register is either not implemented or contains no address if the ADDR_V flag in the IA32_MC3_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.

**Table 35-41. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)**

Register Address		Register Name Fields and Flags	Model Avail- ability	Shared/ Unique <sup>7</sup>	Bit Description
Hex	Dec				
40FH	1039	IA32_MC3_MISC	0, 1, 2, 3, 4, 6	Shared	See Section 15.3.2.4, "IA32_MCi_MISC MSRs." The IA32_MC3_MISC MSR is either not implemented or does not contain additional information if the MISCV flag in the IA32_MC3_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
410H	1040	IA32_MC4_CTL	0, 1, 2, 3, 4, 6	Shared	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
411H	1041	IA32_MC4_STATUS	0, 1, 2, 3, 4, 6	Shared	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
412H	1042	IA32_MC4_ADDR			See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The IA32_MC2_ADDR register is either not implemented or contains no address if the ADDRIV flag in the IA32_MC4_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
413H	1043	IA32_MC4_MISC			See Section 15.3.2.4, "IA32_MCi_MISC MSRs." The IA32_MC2_MISC MSR is either not implemented or does not contain additional information if the MISCV flag in the IA32_MC4_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
480H	1152	IA32_VMX_BASIC	3, 4, 6	Unique	<b>Reporting Register of Basic VMX Capabilities (R/O)</b> See Table 35-2. See Appendix A.1, "Basic VMX Information."
481H	1153	IA32_VMX_PINBASED_CTL	3, 4, 6	Unique	<b>Capability Reporting Register of Pin-based VM-execution Controls (R/O)</b> See Table 35-2. See Appendix A.3, "VM-Execution Controls."
482H	1154	IA32_VMX_PROCBASED_CTL	3, 4, 6	Unique	<b>Capability Reporting Register of Primary Processor-based VM-execution Controls (R/O)</b> See Appendix A.3, "VM-Execution Controls," and see Table 35-2.
483H	1155	IA32_VMX_EXIT_CTL	3, 4, 6	Unique	<b>Capability Reporting Register of VM-exit Controls (R/O)</b> See Appendix A.4, "VM-Exit Controls," and see Table 35-2.

Table 35-41. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

Register Address		Register Name Fields and Flags	Model Avail- ability	Shared/ Unique <sup>1</sup>	Bit Description
Hex	Dec				
484H	1156	IA32_VMX_ENTRY_CTL5	3, 4, 6	Unique	<b>Capability Reporting Register of VM-entry Controls (R/O)</b> See Appendix A.5, "VM-Entry Controls," and see Table 35-2.
485H	1157	IA32_VMX_MISC	3, 4, 6	Unique	<b>Reporting Register of Miscellaneous VMX Capabilities (R/O)</b> See Appendix A.6, "Miscellaneous Data," and see Table 35-2.
486H	1158	IA32_VMX_CR0_FIXED0	3, 4, 6	Unique	<b>Capability Reporting Register of CR0 Bits Fixed to 0 (R/O)</b> See Appendix A.7, "VMX-Fixed Bits in CR0," and see Table 35-2.
487H	1159	IA32_VMX_CR0_FIXED1	3, 4, 6	Unique	<b>Capability Reporting Register of CR0 Bits Fixed to 1 (R/O)</b> See Appendix A.7, "VMX-Fixed Bits in CR0," and see Table 35-2.
488H	1160	IA32_VMX_CR4_FIXED0	3, 4, 6	Unique	<b>Capability Reporting Register of CR4 Bits Fixed to 0 (R/O)</b> See Appendix A.8, "VMX-Fixed Bits in CR4," and see Table 35-2.
489H	1161	IA32_VMX_CR4_FIXED1	3, 4, 6	Unique	<b>Capability Reporting Register of CR4 Bits Fixed to 1 (R/O)</b> See Appendix A.8, "VMX-Fixed Bits in CR4," and see Table 35-2.
48AH	1162	IA32_VMX_VMCS_ENUM	3, 4, 6	Unique	<b>Capability Reporting Register of VMCS Field Enumeration (R/O)</b> See Appendix A.9, "VMCS Enumeration," and see Table 35-2.
48BH	1163	IA32_VMX_PROCBASED_CTL52	3, 4, 6	Unique	<b>Capability Reporting Register of Secondary Processor-based VM-execution Controls (R/O)</b> See Appendix A.3, "VM-Execution Controls," and see Table 35-2.
600H	1536	IA32_DS_AREA	0, 1, 2, 3, 4, 6	Unique	<b>DS Save Area (R/W)</b> See Table 35-2. See Section 18.1.2.4, "Debug Store (DS) Mechanism."
680H	1664	MSR_LASTBRANCH_0_FROM_IP	3, 4, 6	Unique	<b>Last Branch Record 0 (R/W)</b> One of 16 pairs of last branch record registers on the last branch record stack (680H-68FH). This part of the stack contains pointers to the <b>source instruction</b> for one of the last 16 branches, exceptions, or interrupts taken by the processor.

**Table 35-41. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)**

Register Address		Register Name Fields and Flags	Model Avail- ability	Shared/ Unique <sup>7</sup>	Bit Description
Hex	Dec				
					The MSRs at 680H-68FH, 6C0H-6CfH are not available in processor releases before family 0FH, model 03H. These MSRs replace MSRs previously located at 1DBH-1DEH, which performed the same function for early releases.  See Section 17.10, "Last Branch, Call Stack, Interrupt, and Exception Recording for Processors based on Skylake Microarchitecture."
681H	1665	MSR_LASTBRANCH_1_FROM_IP	3, 4, 6	Unique	<b>Last Branch Record 1</b> See description of MSR_LASTBRANCH_0 at 680H.
682H	1666	MSR_LASTBRANCH_2_FROM_IP	3, 4, 6	Unique	<b>Last Branch Record 2</b> See description of MSR_LASTBRANCH_0 at 680H.
683H	1667	MSR_LASTBRANCH_3_FROM_IP	3, 4, 6	Unique	<b>Last Branch Record 3</b> See description of MSR_LASTBRANCH_0 at 680H.
684H	1668	MSR_LASTBRANCH_4_FROM_IP	3, 4, 6	Unique	<b>Last Branch Record 4</b> See description of MSR_LASTBRANCH_0 at 680H.
685H	1669	MSR_LASTBRANCH_5_FROM_IP	3, 4, 6	Unique	<b>Last Branch Record 5</b> See description of MSR_LASTBRANCH_0 at 680H.
686H	1670	MSR_LASTBRANCH_6_FROM_IP	3, 4, 6	Unique	<b>Last Branch Record 6</b> See description of MSR_LASTBRANCH_0 at 680H.
687H	1671	MSR_LASTBRANCH_7_FROM_IP	3, 4, 6	Unique	<b>Last Branch Record 7</b> See description of MSR_LASTBRANCH_0 at 680H.
688H	1672	MSR_LASTBRANCH_8_FROM_IP	3, 4, 6	Unique	<b>Last Branch Record 8</b> See description of MSR_LASTBRANCH_0 at 680H.
689H	1673	MSR_LASTBRANCH_9_FROM_IP	3, 4, 6	Unique	<b>Last Branch Record 9</b> See description of MSR_LASTBRANCH_0 at 680H.
68AH	1674	MSR_LASTBRANCH_10_FROM_IP	3, 4, 6	Unique	<b>Last Branch Record 10</b> See description of MSR_LASTBRANCH_0 at 680H.
68BH	1675	MSR_LASTBRANCH_11_FROM_IP	3, 4, 6	Unique	<b>Last Branch Record 11</b> See description of MSR_LASTBRANCH_0 at 680H.
68CH	1676	MSR_LASTBRANCH_12_FROM_IP	3, 4, 6	Unique	<b>Last Branch Record 12</b> See description of MSR_LASTBRANCH_0 at 680H.
68DH	1677	MSR_LASTBRANCH_13_FROM_IP	3, 4, 6	Unique	<b>Last Branch Record 13</b> See description of MSR_LASTBRANCH_0 at 680H.
68EH	1678	MSR_LASTBRANCH_14_FROM_IP	3, 4, 6	Unique	<b>Last Branch Record 14</b> See description of MSR_LASTBRANCH_0 at 680H.
68FH	1679	MSR_LASTBRANCH_15_FROM_IP	3, 4, 6	Unique	<b>Last Branch Record 15</b> See description of MSR_LASTBRANCH_0 at 680H.

Table 35-41. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

Register Address		Register Name Fields and Flags	Model Avail- ability	Shared/ Unique <sup>1</sup>	Bit Description
Hex	Dec				
6C0H	1728	MSR_LASTBRANCH_0_TO_IP	3, 4, 6	Unique	<b>Last Branch Record 0 (R/W)</b> One of 16 pairs of last branch record registers on the last branch record stack (6C0H-6CFH). This part of the stack contains pointers to the destination instruction for one of the last 16 branches, exceptions, or interrupts that the processor took. See Section 17.10, "Last Branch, Call Stack, Interrupt, and Exception Recording for Processors based on Skylake Microarchitecture."
6C1H	1729	MSR_LASTBRANCH_1_TO_IP	3, 4, 6	Unique	<b>Last Branch Record 1</b> See description of MSR_LASTBRANCH_0 at 6C0H.
6C2H	1730	MSR_LASTBRANCH_2_TO_IP	3, 4, 6	Unique	<b>Last Branch Record 2</b> See description of MSR_LASTBRANCH_0 at 6C0H.
6C3H	1731	MSR_LASTBRANCH_3_TO_IP	3, 4, 6	Unique	<b>Last Branch Record 3</b> See description of MSR_LASTBRANCH_0 at 6C0H.
6C4H	1732	MSR_LASTBRANCH_4_TO_IP	3, 4, 6	Unique	<b>Last Branch Record 4</b> See description of MSR_LASTBRANCH_0 at 6C0H.
6C5H	1733	MSR_LASTBRANCH_5_TO_IP	3, 4, 6	Unique	<b>Last Branch Record 5</b> See description of MSR_LASTBRANCH_0 at 6C0H.
6C6H	1734	MSR_LASTBRANCH_6_TO_IP	3, 4, 6	Unique	<b>Last Branch Record 6</b> See description of MSR_LASTBRANCH_0 at 6C0H.
6C7H	1735	MSR_LASTBRANCH_7_TO_IP	3, 4, 6	Unique	<b>Last Branch Record 7</b> See description of MSR_LASTBRANCH_0 at 6C0H.
6C8H	1736	MSR_LASTBRANCH_8_TO_IP	3, 4, 6	Unique	<b>Last Branch Record 8</b> See description of MSR_LASTBRANCH_0 at 6C0H.
6C9H	1737	MSR_LASTBRANCH_9_TO_IP	3, 4, 6	Unique	<b>Last Branch Record 9</b> See description of MSR_LASTBRANCH_0 at 6C0H.
6CAH	1738	MSR_LASTBRANCH_10_TO_IP	3, 4, 6	Unique	<b>Last Branch Record 10</b> See description of MSR_LASTBRANCH_0 at 6C0H.
6CBH	1739	MSR_LASTBRANCH_11_TO_IP	3, 4, 6	Unique	<b>Last Branch Record 11</b> See description of MSR_LASTBRANCH_0 at 6C0H.
6CCH	1740	MSR_LASTBRANCH_12_TO_IP	3, 4, 6	Unique	<b>Last Branch Record 12</b> See description of MSR_LASTBRANCH_0 at 6C0H.
6CDH	1741	MSR_LASTBRANCH_13_TO_IP	3, 4, 6	Unique	<b>Last Branch Record 13</b> See description of MSR_LASTBRANCH_0 at 6C0H.
6CEH	1742	MSR_LASTBRANCH_14_TO_IP	3, 4, 6	Unique	<b>Last Branch Record 14</b> See description of MSR_LASTBRANCH_0 at 6C0H.
6CFH	1743	MSR_LASTBRANCH_15_TO_IP	3, 4, 6	Unique	<b>Last Branch Record 15</b> See description of MSR_LASTBRANCH_0 at 6C0H.



**Table 35-41. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)**

Register Address		Register Name Fields and Flags	Model Availability	Shared/Unique <sup>1</sup>	Bit Description
Hex	Dec				
C000_0080H		IA32_EFER	3, 4, 6	Unique	<b>Extended Feature Enables</b> See Table 35-2.
C000_0081H		IA32_STAR	3, 4, 6	Unique	<b>System Call Target Address (R/W)</b> See Table 35-2.
C000_0082H		IA32_LSTAR	3, 4, 6	Unique	<b>IA-32e Mode System Call Target Address (R/W)</b> See Table 35-2.
C000_0084H		IA32_FMASK	3, 4, 6	Unique	<b>System Call Flag Mask (R/W)</b> See Table 35-2.
C000_0100H		IA32_FS_BASE	3, 4, 6	Unique	<b>Map of BASE Address of FS (R/W)</b> See Table 35-2.
C000_0101H		IA32_GS_BASE	3, 4, 6	Unique	<b>Map of BASE Address of GS (R/W)</b> See Table 35-2.
C000_0102H		IA32_KERNEL_GSBASE	3, 4, 6	Unique	<b>Swap Target of BASE Address of GS (R/W)</b> See Table 35-2.

**NOTES**

1. For HT-enabled processors, there may be more than one logical processors per physical unit. If an MSR is Shared, this means that one MSR is shared between logical processors. If an MSR is unique, this means that each logical processor has its own MSR.

**35.18.1 MSRs Unique to Intel® Xeon® Processor MP with L3 Cache**

The MSRs listed in Table 35-42 apply to Intel® Xeon® Processor MP with up to 8MB level three cache. These processors can be detected by enumerating the deterministic cache parameter leaf of CPUID instruction (with EAX = 4 as input) to detect the presence of the third level cache, and with CPUID reporting family encoding 0FH, model encoding 3 or 4 (see CPUID instruction for more details).

**Table 35-42. MSRs Unique to 64-bit Intel® Xeon® Processor MP with Up to an 8 MB L3 Cache**

Register Address		Register Name Fields and Flags	Model Availability	Shared/Unique	Bit Description
107CCH					
107CDH		MSR_IFSB_BUSQ1	3, 4	Shared	<b>IFSB BUSQ Event Control and Counter Register (R/W)</b>
107CEH		MSR_IFSB_SNPQ0	3, 4	Shared	<b>IFSB SNPQ Event Control and Counter Register (R/W)</b> See Section 18.17, "Performance Monitoring on 64-bit Intel Xeon Processor MP with Up to 8-MByte L3 Cache."
107CFH		MSR_IFSB_SNPQ1	3, 4	Shared	<b>IFSB SNPQ Event Control and Counter Register (R/W)</b>

**Table 35-42. MSRs Unique to 64-bit Intel® Xeon® Processor MP with Up to an 8 MB L3 Cache (Contd.)**

Register Address		Register Name Fields and Flags	Model Avail- ability	Shared/ Unique	Bit Description
107D0H		MSR_EFSB_DRDY0	3, 4	Shared	<b>EFSB DRDY Event Control and Counter Register (R/W)</b> See Section 18.17, "Performance Monitoring on 64-bit Intel Xeon Processor MP with Up to 8-MByte L3 Cache" for details.
107D1H		MSR_EFSB_DRDY1	3, 4	Shared	<b>EFSB DRDY Event Control and Counter Register (R/W)</b>
107D2H		MSR_IFSB_CTL6	3, 4	Shared	<b>IFSB Latency Event Control Register (R/W)</b> See Section 18.17, "Performance Monitoring on 64-bit Intel Xeon Processor MP with Up to 8-MByte L3 Cache" for details.
107D3H		MSR_IFSB_CNTR7	3, 4	Shared	<b>IFSB Latency Event Counter Register (R/W)</b> See Section 18.17, "Performance Monitoring on 64-bit Intel Xeon Processor MP with Up to 8-MByte L3 Cache."

The MSRs listed in Table 35-43 apply to Intel® Xeon® Processor 7100 series. These processors can be detected by enumerating the deterministic cache parameter leaf of CPUID instruction (with EAX = 4 as input) to detect the presence of the third level cache, and with CPUID reporting family encoding 0FH, model encoding 6 (See CPUID instruction for more details.). The performance monitoring MSRs listed in Table 35-43 are shared between logical processors in the same core, but are replicated for each core.

**Table 35-43. MSRs Unique to Intel® Xeon® Processor 7100 Series**

Register Address		Register Name Fields and Flags	Model Avail- ability	Shared/ Unique	Bit Description
107CCH		MSR_EMON_L3_CTR_CTL0	6	Shared	<b>GBUSQ Event Control and Counter Register (R/W)</b> See Section 18.17, "Performance Monitoring on 64-bit Intel Xeon Processor MP with Up to 8-MByte L3 Cache."
107CDH		MSR_EMON_L3_CTR_CTL1	6	Shared	<b>GBUSQ Event Control and Counter Register (R/W)</b>
107CEH		MSR_EMON_L3_CTR_CTL2	6	Shared	<b>GSNPQ Event Control and Counter Register (R/W)</b> See Section 18.17, "Performance Monitoring on 64-bit Intel Xeon Processor MP with Up to 8-MByte L3 Cache."
107CFH		MSR_EMON_L3_CTR_CTL3	6	Shared	<b>GSNPQ Event Control and Counter Register (R/W)</b>

**Table 35-43. MSRs Unique to Intel® Xeon® Processor 7100 Series (Contd.)**

Register Address		Register Name Fields and Flags	Model Avail- ability	Shared/ Unique	Bit Description
107D0H		MSR_EMON_L3_CTR_CTL4	6	Shared	<b>FSB Event Control and Counter Register (R/W)</b> See Section 18.17, "Performance Monitoring on 64-bit Intel Xeon Processor MP with Up to 8-MByte L3 Cache" for details.
107D1H		MSR_EMON_L3_CTR_CTL5	6	Shared	<b>FSB Event Control and Counter Register (R/W)</b>
107D2H		MSR_EMON_L3_CTR_CTL6	6	Shared	<b>FSB Event Control and Counter Register (R/W)</b>
107D3H		MSR_EMON_L3_CTR_CTL7	6	Shared	<b>FSB Event Control and Counter Register (R/W)</b>

### 35.19 MSRS IN INTEL® CORE™ SOLO AND INTEL® CORE™ DUO PROCESSORS

Model-specific registers (MSRs) for Intel Core Solo, Intel Core Duo processors, and Dual-core Intel Xeon processor LV are listed in Table 35-44. The column "Shared/Unique" applies to Intel Core Duo processor. "Unique" means each processor core has a separate MSR, or a bit field in an MSR governs only a core independently. "Shared" means the MSR or the bit field in an MSR address governs the operation of both processor cores.

**Table 35-44. MSRs in Intel® Core™ Solo, Intel® Core™ Duo Processors, and Dual-Core Intel® Xeon® Processor LV**

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
0H	0	P5_MC_ADDR	Unique	See Section 35.22, "MSRs in Pentium Processors," and see Table 35-2.
1H	1	P5_MC_TYPE	Unique	See Section 35.22, "MSRs in Pentium Processors," and see Table 35-2.
6H	6	IA32_MONITOR_FILTER_SIZE	Unique	See Section 8.10.5, "Monitor/Mwait Address Range Determination," and see Table 35-2.
10H	16	IA32_TIME_STAMP_COUNTER	Unique	See Section 17.15, "Time-Stamp Counter," and see Table 35-2.
17H	23	IA32_PLATFORM_ID	Shared	<b>Platform ID (R)</b> See Table 35-2. The operating system can use this MSR to determine "slot" information for the processor and the proper microcode update to load.
1BH	27	IA32_APIC_BASE	Unique	See Section 10.4.4, "Local APIC Status and Location," and see Table 35-2.
2AH	42	MSR_EBL_CR_POWERON	Shared	<b>Processor Hard Power-On Configuration (R/W)</b> Enables and disables processor features; (R) indicates current processor configuration.
	0			Reserved.

Table 35-44. MSRs in Intel® Core™ Solo, Intel® Core™ Duo Processors, and Dual-Core Intel® Xeon® Processor LV

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
		1		<b>Data Error Checking Enable (R/W)</b> 1 = Enabled; 0 = Disabled Note: Not all processor implements R/W.
		2		<b>Response Error Checking Enable (R/W)</b> 1 = Enabled; 0 = Disabled Note: Not all processor implements R/W.
		3		<b>MCERR# Drive Enable (R/W)</b> 1 = Enabled; 0 = Disabled Note: Not all processor implements R/W.
		4		<b>Address Parity Enable (R/W)</b> 1 = Enabled; 0 = Disabled Note: Not all processor implements R/W.
		6: 5		Reserved
		7		<b>BINIT# Driver Enable (R/W)</b> 1 = Enabled; 0 = Disabled Note: Not all processor implements R/W.
		8		<b>Output Tri-state Enabled (R/O)</b> 1 = Enabled; 0 = Disabled
		9		<b>Execute BIST (R/O)</b> 1 = Enabled; 0 = Disabled
		10		<b>MCERR# Observation Enabled (R/O)</b> 1 = Enabled; 0 = Disabled
		11		Reserved
		12		<b>BINIT# Observation Enabled (R/O)</b> 1 = Enabled; 0 = Disabled
		13		Reserved
		14		<b>1 MByte Power on Reset Vector (R/O)</b> 1 = 1 MByte; 0 = 4 GBytes
		15		Reserved
		17:16		<b>APIC Cluster ID (R/O)</b>
		18		<b>System Bus Frequency (R/O)</b> 0 = 100 MHz 1 = Reserved
		19		Reserved.
		21: 20		<b>Symmetric Arbitration ID (R/O)</b>
		26:22		<b>Clock Frequency Ratio (R/O)</b>
3AH	58	IA32_FEATURE_CONTROL	Unique	<b>Control Features in IA-32 Processor (R/W)</b> See Table 35-2.

**Table 35-44. MSRs in Intel® Core™ Solo, Intel® Core™ Duo Processors, and Dual-Core Intel® Xeon® Processor LV**

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
40H	64	MSR_LASTBRANCH_0	Unique	<b>Last Branch Record 0 (R/W)</b> One of 8 last branch record registers on the last branch record stack: bits 31-0 hold the ‘from’ address and bits 63-32 hold the ‘to’ address. See also: <ul style="list-style-type: none"> <li>▪ Last Branch Record Stack TOS at 1C9H</li> <li>▪ Section 17.13, “Last Branch, Interrupt, and Exception Recording (Pentium M Processors).”</li> </ul>
41H	65	MSR_LASTBRANCH_1	Unique	<b>Last Branch Record 1 (R/W)</b> See description of MSR_LASTBRANCH_0.
42H	66	MSR_LASTBRANCH_2	Unique	<b>Last Branch Record 2 (R/W)</b> See description of MSR_LASTBRANCH_0.
43H	67	MSR_LASTBRANCH_3	Unique	<b>Last Branch Record 3 (R/W)</b> See description of MSR_LASTBRANCH_0.
44H	68	MSR_LASTBRANCH_4	Unique	<b>Last Branch Record 4 (R/W)</b> See description of MSR_LASTBRANCH_0.
45H	69	MSR_LASTBRANCH_5	Unique	<b>Last Branch Record 5 (R/W)</b> See description of MSR_LASTBRANCH_0.
46H	70	MSR_LASTBRANCH_6	Unique	<b>Last Branch Record 6 (R/W)</b> See description of MSR_LASTBRANCH_0.
47H	71	MSR_LASTBRANCH_7	Unique	<b>Last Branch Record 7 (R/W)</b> See description of MSR_LASTBRANCH_0.
79H	121	IA32_BIOS_UPDT_TRIG	Unique	<b>BIOS Update Trigger Register (W)</b> See Table 35-2.
8BH	139	IA32_BIOS_SIGN_ID	Unique	<b>BIOS Update Signature ID (RO)</b> See Table 35-2.
C1H	193	IA32_PMC0	Unique	<b>Performance counter register</b> See Table 35-2.
C2H	194	IA32_PMC1	Unique	<b>Performance counter register</b> See Table 35-2.
CDH	205	MSR_FSB_FREQ	Shared	<b>Scaleable Bus Speed (RO)</b> This field indicates the scaleable bus clock speed:
		2:0		<ul style="list-style-type: none"> <li>▪ 101B: 100 MHz (FSB 400)</li> <li>▪ 001B: 133 MHz (FSB 533)</li> <li>▪ 011B: 167 MHz (FSB 667)</li> </ul> 133.33 MHz should be utilized if performing calculation with System Bus Speed when encoding is 101B. 166.67 MHz should be utilized if performing calculation with System Bus Speed when encoding is 001B.
		63:3		Reserved.
E7H	231	IA32_MPERF	Unique	<b>Maximum Performance Frequency Clock Count. (RW)</b> See Table 35-2.

**Table 35-44. MSRs in Intel® Core™ Solo, Intel® Core™ Duo Processors, and Dual-Core Intel® Xeon® Processor LV**

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
E8H	232	IA32_APERF	Unique	<b>Actual Performance Frequency Clock Count. (RW)</b> See Table 35-2.
FEH	254	IA32_MTRRCAP	Unique	See Table 35-2.
11EH	281	MSR_BBL_CR_CTL3	Shared	
		0		<b>L2 Hardware Enabled (RO)</b> 1 = If the L2 is hardware-enabled 0 = Indicates if the L2 is hardware-disabled
		7:1		Reserved.
		8		<b>L2 Enabled (R/W)</b> 1 = L2 cache has been initialized 0 = Disabled (default) Until this bit is set the processor will not respond to the WBINVD instruction or the assertion of the FLUSH# input.
		22:9		Reserved.
		23		<b>L2 Not Present (RO)</b> 0 = L2 Present 1 = L2 Not Present
		63:24		Reserved.
174H	372	IA32_SYSENTER_CS	Unique	See Table 35-2.
175H	373	IA32_SYSENTER_ESP	Unique	See Table 35-2.
176H	374	IA32_SYSENTER_EIP	Unique	See Table 35-2.
179H	377	IA32_MCG_CAP	Unique	See Table 35-2.
17AH	378	IA32_MCG_STATUS	Unique	
		0		<b>RIPV</b> When set, this bit indicates that the instruction addressed by the instruction pointer pushed on the stack (when the machine check was generated) can be used to restart the program. If this bit is cleared, the program cannot be reliably restarted.
		1		<b>EIPV</b> When set, this bit indicates that the instruction addressed by the instruction pointer pushed on the stack (when the machine check was generated) is directly associated with the error.
		2		<b>MCIP</b> When set, this bit indicates that a machine check has been generated. If a second machine check is detected while this bit is still set, the processor enters a shutdown state. Software should write this bit to 0 after processing a machine check exception.
		63:3		Reserved.
186H	390	IA32_PERFEVTSELO	Unique	See Table 35-2.

**Table 35-44. MSRs in Intel® Core™ Solo, Intel® Core™ Duo Processors, and Dual-Core Intel® Xeon® Processor LV**

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
187H	391	IA32_PERFEVTSEL1	Unique	See Table 35-2.
198H	408	IA32_PERF_STATUS	Shared	See Table 35-2.
199H	409	IA32_PERF_CTL	Unique	See Table 35-2.
19AH	410	IA32_CLOCK_MODULATION	Unique	<b>Clock Modulation (R/W)</b> See Table 35-2.
19BH	411	IA32_THERM_INTERRUPT	Unique	<b>Thermal Interrupt Control (R/W)</b> See Table 35-2. See Section 14.7.2, "Thermal Monitor."
19CH	412	IA32_THERM_STATUS	Unique	<b>Thermal Monitor Status (R/W)</b> See Table 35-2. See Section 14.7.2, "Thermal Monitor".
19DH	413	MSR_THERM2_CTL	Unique	
		15:0		Reserved.
		16		<b>TM_SELECT (R/W)</b> Mode of automatic thermal monitor: 0 = Thermal Monitor 1 (thermally-initiated on-die modulation of the stop-clock duty cycle) 1 = Thermal Monitor 2 (thermally-initiated frequency transitions) If bit 3 of the IA32_MISC_ENABLE register is cleared, TM_SELECT has no effect. Neither TM1 nor TM2 will be enabled.
		63:16		Reserved.
1A0H	416	IA32_MISC_ENABLE		<b>Enable Miscellaneous Processor Features (R/W)</b> Allows a variety of processor functions to be enabled and disabled.
		2:0		Reserved.
		3	Unique	<b>Automatic Thermal Control Circuit Enable (R/W)</b> See Table 35-2.
		6:4		Reserved.
		7	Shared	<b>Performance Monitoring Available (R)</b> See Table 35-2.
		9:8		Reserved.
		10	Shared	<b>FERR# Multiplexing Enable (R/W)</b> 1 = FERR# asserted by the processor to indicate a pending break event within the processor 0 = Indicates compatible FERR# signaling behavior This bit must be set to 1 to support XAPIC interrupt model usage.
		11	Shared	<b>Branch Trace Storage Unavailable (RO)</b> See Table 35-2.
		12		Reserved.

**Table 35-44. MSRs in Intel® Core™ Solo, Intel® Core™ Duo Processors, and Dual-Core Intel® Xeon® Processor LV**

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
		13	Shared	<p><b>TM2 Enable (R/W)</b></p> <p>When this bit is set (1) and the thermal sensor indicates that the die temperature is at the pre-determined threshold, the Thermal Monitor 2 mechanism is engaged. TM2 will reduce the bus to core ratio and voltage according to the value last written to MSR_THERM2_CTL bits 15:0.</p> <p>When this bit is clear (0, default), the processor does not change the VID signals or the bus to core ratio when the processor enters a thermal managed state.</p> <p>If the TM2 feature flag (ECX[8]) is not set to 1 after executing CPUID with EAX = 1, then this feature is not supported and BIOS must not alter the contents of this bit location. The processor is operating out of spec if both this bit and the TM1 bit are set to disabled states.</p>
		15:14		Reserved.
		16	Shared	<p><b>Enhanced Intel SpeedStep Technology Enable (R/W)</b></p> <p>1 = Enhanced Intel SpeedStep Technology enabled</p>
		18	Shared	<p><b>ENABLE MONITOR FSM (R/W)</b></p> <p>See Table 35-2.</p>
		19		<b>Reserved.</b>
		22	Shared	<p><b>Limit CPUID Maxval (R/W)</b></p> <p>See Table 35-2.</p> <p>Setting this bit may cause behavior in software that depends on the availability of CPUID leaves greater than 2.</p>
		33:23		Reserved.
		34	Shared	<p><b>XD Bit Disable (R/W)</b></p> <p>See Table 35-2.</p>
		63:35		Reserved.
1C9H	457	MSR_LASTBRANCH_TOS	Unique	<p><b>Last Branch Record Stack TOS (R/W)</b></p> <p>Contains an index (bits 0-3) that points to the MSR containing the most recent branch record.</p> <p>See MSR_LASTBRANCH_0_FROM_IP (at 40H).</p>
1D9H	473	IA32_DEBUGCTL	Unique	<p><b>Debug Control (R/W)</b></p> <p>Controls how several debug features are used. Bit definitions are discussed in the referenced section.</p>
1DDH	477	MSR_LER_FROM_LIP	Unique	<p><b>Last Exception Record From Linear IP (R)</b></p> <p>Contains a pointer to the last branch instruction that the processor executed prior to the last exception that was generated or the last interrupt that was handled.</p>
1DEH	478	MSR_LER_TO_LIP	Unique	<p><b>Last Exception Record To Linear IP (R)</b></p> <p>This area contains a pointer to the target of the last branch instruction that the processor executed prior to the last exception that was generated or the last interrupt that was handled.</p>



**Table 35-44. MSRs in Intel® Core™ Solo, Intel® Core™ Duo Processors, and Dual-Core Intel® Xeon® Processor LV**

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
1E0H	480	ROB_CR_BKUPTMPDR6	Unique	
		1:0		Reserved.
		2		Fast String Enable bit. (Default, enabled)
200H	512	MTRRphysBase0	Unique	
201H	513	MTRRphysMask0	Unique	
202H	514	MTRRphysBase1	Unique	
203H	515	MTRRphysMask1	Unique	
204H	516	MTRRphysBase2	Unique	
205H	517	MTRRphysMask2	Unique	
206H	518	MTRRphysBase3	Unique	
207H	519	MTRRphysMask3	Unique	
208H	520	MTRRphysBase4	Unique	
209H	521	MTRRphysMask4	Unique	
20AH	522	MTRRphysBase5	Unique	
20BH	523	MTRRphysMask5	Unique	
20CH	524	MTRRphysBase6	Unique	
20DH	525	MTRRphysMask6	Unique	
20EH	526	MTRRphysBase7	Unique	
20FH	527	MTRRphysMask7	Unique	
250H	592	MTRRfix64K_00000	Unique	
258H	600	MTRRfix16K_80000	Unique	
259H	601	MTRRfix16K_A0000	Unique	
268H	616	MTRRfix4K_C0000	Unique	
269H	617	MTRRfix4K_C8000	Unique	
26AH	618	MTRRfix4K_D0000	Unique	
26BH	619	MTRRfix4K_D8000	Unique	
26CH	620	MTRRfix4K_E0000	Unique	
26DH	621	MTRRfix4K_E8000	Unique	
26EH	622	MTRRfix4K_F0000	Unique	
26FH	623	MTRRfix4K_F8000	Unique	
2FFH	767	IA32_MTRR_DEF_TYPE	Unique	<b>Default Memory Types (R/W)</b> See Table 35-2. See Section 11.11.2.1, "IA32_MTRR_DEF_TYPE MSR."
400H	1024	IA32_MCO_CTL	Unique	See Section 15.3.2.1, "IA32_MCI_CTL MSRs."
401H	1025	IA32_MCO_STATUS	Unique	See Section 15.3.2.2, "IA32_MCI_STATUS MSRS."

**Table 35-44. MSRs in Intel® Core™ Solo, Intel® Core™ Duo Processors, and Dual-Core Intel® Xeon® Processor LV**

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
402H	1026	IA32_MC0_ADDR	Unique	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The IA32_MC0_ADDR register is either not implemented or contains no address if the ADDR_V flag in the IA32_MC0_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
404H	1028	IA32_MC1_CTL	Unique	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
405H	1029	IA32_MC1_STATUS	Unique	See Section 15.3.2.2, "IA32_MCi_STATUS MSRs."
406H	1030	IA32_MC1_ADDR	Unique	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The IA32_MC1_ADDR register is either not implemented or contains no address if the ADDR_V flag in the IA32_MC1_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
408H	1032	IA32_MC2_CTL	Unique	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
409H	1033	IA32_MC2_STATUS	Unique	See Section 15.3.2.2, "IA32_MCi_STATUS MSRs."
40AH	1034	IA32_MC2_ADDR	Unique	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The IA32_MC2_ADDR register is either not implemented or contains no address if the ADDR_V flag in the IA32_MC2_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
40CH	1036	MSR_MC4_CTL	Unique	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
40DH	1037	MSR_MC4_STATUS	Unique	See Section 15.3.2.2, "IA32_MCi_STATUS MSRs."
40EH	1038	MSR_MC4_ADDR	Unique	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The MSR_MC4_ADDR register is either not implemented or contains no address if the ADDR_V flag in the MSR_MC4_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
410H	1040	IA32_MC3_CTL		See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
411H	1041	IA32_MC3_STATUS		See Section 15.3.2.2, "IA32_MCi_STATUS MSRs."
412H	1042	MSR_MC3_ADDR	Unique	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The MSR_MC3_ADDR register is either not implemented or contains no address if the ADDR_V flag in the MSR_MC3_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
413H	1043	MSR_MC3_MISC	Unique	
414H	1044	MSR_MC5_CTL	Unique	
415H	1045	MSR_MC5_STATUS	Unique	
416H	1046	MSR_MC5_ADDR	Unique	
417H	1047	MSR_MC5_MISC	Unique	
480H	1152	IA32_VMX_BASIC	Unique	<b>Reporting Register of Basic VMX Capabilities (R/O)</b> See Table 35-2. See Appendix A.1, "Basic VMX Information" (If CPUID.01H:ECX.[bit 9])

**Table 35-44. MSRs in Intel® Core™ Solo, Intel® Core™ Duo Processors, and Dual-Core Intel® Xeon® Processor LV**

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
481H	1153	IA32_VMX_PINBASED_CTLS	Unique	<b>Capability Reporting Register of Pin-based VM-execution Controls (R/O)</b> See Appendix A.3, “VM-Execution Controls” (If CPUID.01H:ECX.[bit 9])
482H	1154	IA32_VMX_PROCBASED_CTLS	Unique	<b>Capability Reporting Register of Primary Processor-based VM-execution Controls (R/O)</b> See Appendix A.3, “VM-Execution Controls” (If CPUID.01H:ECX.[bit 9])
483H	1155	IA32_VMX_EXIT_CTLS	Unique	<b>Capability Reporting Register of VM-exit Controls (R/O)</b> See Appendix A.4, “VM-Exit Controls” (If CPUID.01H:ECX.[bit 9])
484H	1156	IA32_VMX_ENTRY_CTLS	Unique	<b>Capability Reporting Register of VM-entry Controls (R/O)</b> See Appendix A.5, “VM-Entry Controls” (If CPUID.01H:ECX.[bit 9])
485H	1157	IA32_VMX_MISC	Unique	<b>Reporting Register of Miscellaneous VMX Capabilities (R/O)</b> See Appendix A.6, “Miscellaneous Data” (If CPUID.01H:ECX.[bit 9])
486H	1158	IA32_VMX_CRO_FIXED0	Unique	<b>Capability Reporting Register of CRO Bits Fixed to 0 (R/O)</b> See Appendix A.7, “VMX-Fixed Bits in CRO” (If CPUID.01H:ECX.[bit 9])
487H	1159	IA32_VMX_CRO_FIXED1	Unique	<b>Capability Reporting Register of CRO Bits Fixed to 1 (R/O)</b> See Appendix A.7, “VMX-Fixed Bits in CRO” (If CPUID.01H:ECX.[bit 9])
488H	1160	IA32_VMX_CR4_FIXED0	Unique	<b>Capability Reporting Register of CR4 Bits Fixed to 0 (R/O)</b> See Appendix A.8, “VMX-Fixed Bits in CR4” (If CPUID.01H:ECX.[bit 9])
489H	1161	IA32_VMX_CR4_FIXED1	Unique	<b>Capability Reporting Register of CR4 Bits Fixed to 1 (R/O)</b> See Appendix A.8, “VMX-Fixed Bits in CR4” (If CPUID.01H:ECX.[bit 9])
48AH	1162	IA32_VMX_VMCS_ENUM	Unique	<b>Capability Reporting Register of VMCS Field Enumeration (R/O)</b> See Appendix A.9, “VMCS Enumeration” (If CPUID.01H:ECX.[bit 9])
48BH	1163	IA32_VMX_PROCBASED_CTLS2	Unique	<b>Capability Reporting Register of Secondary Processor-based VM-execution Controls (R/O)</b> See Appendix A.3, “VM-Execution Controls” (If CPUID.01H:ECX.[bit 9] and IA32_VMX_PROCBASED_CTLS[bit 63])
600H	1536	IA32_DS_AREA	Unique	<b>DS Save Area (R/W)</b> See Table 35-2. See Section 18.12.4, “Debug Store (DS) Mechanism.”

**Table 35-44. MSRs in Intel® Core™ Solo, Intel® Core™ Duo Processors, and Dual-Core Intel® Xeon® Processor LV**

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
		31:0		<b>DS Buffer Management Area</b> Linear address of the first byte of the DS buffer management area.
		63:32		Reserved.
C000_0080H		IA32_EFER	Unique	See Table 35-2.
		10:0		Reserved.
		11		<b>Execute Disable Bit Enable</b>
		63:12		Reserved.

## 35.20 MSRS IN THE PENTIUM M PROCESSOR

Model-specific registers (MSRs) for the Pentium M processor are similar to those described in Section 35.21 for P6 family processors. The following table describes new MSRs and MSRs whose behavior has changed on the Pentium M processor.

**Table 35-45. MSRs in Pentium M Processors**

Register Address		Register Name	Bit Description
Hex	Dec		
0H	0	P5_MC_ADDR	See Section 35.22, "MSRs in Pentium Processors."
1H	1	P5_MC_TYPE	See Section 35.22, "MSRs in Pentium Processors."
10H	16	IA32_TIME_STAMP_COUNTER	See Section 17.15, "Time-Stamp Counter," and see Table 35-2.
17H	23	IA32_PLATFORM_ID	<b>Platform ID (R)</b> See Table 35-2. The operating system can use this MSR to determine "slot" information for the processor and the proper microcode update to load.
2AH	42	MSR_EBL_CR_POWERON	<b>Processor Hard Power-On Configuration (R/W)</b> Enables and disables processor features. <b>(R)</b> Indicates current processor configuration.
		0	Reserved.
		1	<b>Data Error Checking Enable (R)</b> 0 = Disabled Always 0 on the Pentium M processor.
		2	<b>Response Error Checking Enable (R)</b> 0 = Disabled Always 0 on the Pentium M processor.
		3	<b>MCERR# Drive Enable (R)</b> 0 = Disabled Always 0 on the Pentium M processor.

**Table 35-45. MSRs in Pentium M Processors (Contd.)**

Register Address		Register Name	Bit Description
Hex	Dec		
		4	<b>Address Parity Enable (R)</b> 0 = Disabled Always 0 on the Pentium M processor.
		6:5	Reserved.
		7	<b>BINIT# Driver Enable (R)</b> 1 = Enabled; 0 = Disabled Always 0 on the Pentium M processor.
		8	<b>Output Tri-state Enabled (R/O)</b> 1 = Enabled; 0 = Disabled
		9	<b>Execute BIST (R/O)</b> 1 = Enabled; 0 = Disabled
		10	<b>MCERR# Observation Enabled (R/O)</b> 1 = Enabled; 0 = Disabled Always 0 on the Pentium M processor.
		11	Reserved.
		12	<b>BINIT# Observation Enabled (R/O)</b> 1 = Enabled; 0 = Disabled Always 0 on the Pentium M processor.
		13	Reserved.
		14	<b>1 MByte Power on Reset Vector (R/O)</b> 1 = 1 MByte; 0 = 4 GBytes Always 0 on the Pentium M processor.
		15	Reserved.
		17:16	<b>APIC Cluster ID (R/O)</b> Always 00B on the Pentium M processor.
		18	<b>System Bus Frequency (R/O)</b> 0 = 100 MHz 1 = Reserved Always 0 on the Pentium M processor.
		19	Reserved.
		21:20	<b>Symmetric Arbitration ID (R/O)</b> Always 00B on the Pentium M processor.
		26:22	Clock Frequency Ratio (R/O)
40H	64	MSR_LASTBRANCH_0	<b>Last Branch Record 0 (R/W)</b> One of 8 last branch record registers on the last branch record stack: bits 31-0 hold the 'from' address and bits 63-32 hold the to address. See also: <ul style="list-style-type: none"> <li>▪ Last Branch Record Stack TOS at 1C9H</li> <li>▪ Section 17.13, "Last Branch, Interrupt, and Exception Recording (Pentium M Processors)"</li> </ul>

Table 35-45. MSRs in Pentium M Processors (Contd.)

Register Address		Register Name	Bit Description
Hex	Dec		
41H	65	MSR_LASTBRANCH_1	<b>Last Branch Record 1 (R/W)</b> See description of MSR_LASTBRANCH_0.
42H	66	MSR_LASTBRANCH_2	<b>Last Branch Record 2 (R/W)</b> See description of MSR_LASTBRANCH_0.
43H	67	MSR_LASTBRANCH_3	<b>Last Branch Record 3 (R/W)</b> See description of MSR_LASTBRANCH_0.
44H	68	MSR_LASTBRANCH_4	<b>Last Branch Record 4 (R/W)</b> See description of MSR_LASTBRANCH_0.
45H	69	MSR_LASTBRANCH_5	<b>Last Branch Record 5 (R/W)</b> See description of MSR_LASTBRANCH_0.
46H	70	MSR_LASTBRANCH_6	<b>Last Branch Record 6 (R/W)</b> See description of MSR_LASTBRANCH_0.
47H	71	MSR_LASTBRANCH_7	<b>Last Branch Record 7 (R/W)</b> See description of MSR_LASTBRANCH_0.
119H	281	MSR_BBL_CR_CTL	
		63:0	Reserved.
11EH	281	MSR_BBL_CR_CTL3	
		0	<b>L2 Hardware Enabled (RO)</b> 1 = If the L2 is hardware-enabled 0 = Indicates if the L2 is hardware-disabled
		4:1	Reserved.
		5	<b>ECC Check Enable (RO)</b> This bit enables ECC checking on the cache data bus. ECC is always generated on write cycles. 0 = Disabled (default) 1 = Enabled For the Pentium M processor, ECC checking on the cache data bus is always enabled.
		7:6	Reserved.
		8	<b>L2 Enabled (R/W)</b> 1 = L2 cache has been initialized 0 = Disabled (default) Until this bit is set the processor will not respond to the WBINVD instruction or the assertion of the FLUSH# input.
		22:9	Reserved.
		23	<b>L2 Not Present (RO)</b> 0 = L2 Present 1 = L2 Not Present
	63:24	Reserved.	
179H	377	IA32_MCG_CAP	

**Table 35-45. MSRs in Pentium M Processors (Contd.)**

Register Address		Register Name	Bit Description
Hex	Dec		
		7:0	<b>Count (RO)</b> Indicates the number of hardware unit error reporting banks available in the processor.
		8	<b>IA32_MCG_CTL Present (RO)</b> 1 = Indicates that the processor implements the MSR_MCG_CTL register found at MSR 17BH. 0 = Not supported.
		63:9	Reserved.
17AH	378	IA32_MCG_STATUS	
		0	<b>RIPV</b> When set, this bit indicates that the instruction addressed by the instruction pointer pushed on the stack (when the machine check was generated) can be used to restart the program. If this bit is cleared, the program cannot be reliably restarted.
		1	<b>EIPV</b> When set, this bit indicates that the instruction addressed by the instruction pointer pushed on the stack (when the machine check was generated) is directly associated with the error.
		2	<b>MCIP</b> When set, this bit indicates that a machine check has been generated. If a second machine check is detected while this bit is still set, the processor enters a shutdown state. Software should write this bit to 0 after processing a machine check exception.
		63:3	Reserved.
198H	408	IA32_PERF_STATUS	See Table 35-2.
199H	409	IA32_PERF_CTL	See Table 35-2.
19AH	410	IA32_CLOCK_MODULATION	<b>Clock Modulation (R/W).</b> See Table 35-2. See Section 14.7.3, "Software Controlled Clock Modulation."
19BH	411	IA32_THERM_INTERRUPT	<b>Thermal Interrupt Control (R/W)</b> See Table 35-2. See Section 14.7.2, "Thermal Monitor."
19CH	412	IA32_THERM_STATUS	<b>Thermal Monitor Status (R/W)</b> See Table 35-2. See Section 14.7.2, "Thermal Monitor."
19DH	413	MSR_THERM2_CTL	
		15:0	Reserved.

Table 35-45. MSRs in Pentium M Processors (Contd.)

Register Address		Register Name	Bit Description
Hex	Dec		
		16	<b>TM_SELECT (R/W)</b> Mode of automatic thermal monitor: 0 = Thermal Monitor 1 (thermally-initiated on-die modulation of the stop-clock duty cycle) 1 = Thermal Monitor 2 (thermally-initiated frequency transitions) If bit 3 of the IA32_MISC_ENABLE register is cleared, TM_SELECT has no effect. Neither TM1 nor TM2 will be enabled.
		63:16	Reserved.
1A0H	416	IA32_MISC_ENABLE	<b>Enable Miscellaneous Processor Features (R/W)</b> Allows a variety of processor functions to be enabled and disabled.
		2:0	Reserved.
		3	<b>Automatic Thermal Control Circuit Enable (R/W)</b> 1 = Setting this bit enables the thermal control circuit (TCC) portion of the Intel Thermal Monitor feature. This allows processor clocks to be automatically modulated based on the processor's thermal sensor operation. 0 = Disabled (default). The automatic thermal control circuit enable bit determines if the thermal control circuit (TCC) will be activated when the processor's internal thermal sensor determines the processor is about to exceed its maximum operating temperature. When the TCC is activated and TM1 is enabled, the processors clocks will be forced to a 50% duty cycle. BIOS must enable this feature. The bit should not be confused with the on-demand thermal control circuit enable bit.
		6:4	Reserved.
		7	<b>Performance Monitoring Available (R)</b> 1 = Performance monitoring enabled 0 = Performance monitoring disabled
		9:8	Reserved.
		10	<b>FERR# Multiplexing Enable (R/W)</b> 1 = FERR# asserted by the processor to indicate a pending break event within the processor 0 = Indicates compatible FERR# signaling behavior This bit must be set to 1 to support XAPIC interrupt model usage.
			<b>Branch Trace Storage Unavailable (RO)</b> 1 = Processor doesn't support branch trace storage (BTS) 0 = BTS is supported
		12	<b>Processor Event Based Sampling Unavailable (RO)</b> 1 = Processor does not support processor event based sampling (PEBS); 0 = PEBS is supported. The Pentium M processor does not support PEBS.
		15:13	Reserved.



**Table 35-45. MSRs in Pentium M Processors (Contd.)**

Register Address		Register Name	Bit Description
Hex	Dec		
		16	<b>Enhanced Intel SpeedStep Technology Enable (R/W)</b> 1 = Enhanced Intel SpeedStep Technology enabled. On the Pentium M processor, this bit may be configured to be read-only.
		22:17	Reserved.
		23	<b>xTPR Message Disable (R/W)</b> When set to 1, xTPR messages are disabled. xTPR messages are optional messages that allow the processor to inform the chipset of its priority. The default is processor specific.
		63:24	Reserved.
1C9H	457	MSR_LASTBRANCH_TOS	<b>Last Branch Record Stack TOS (R/W)</b> Contains an index (bits 0-3) that points to the MSR containing the most recent branch record. See also: <ul style="list-style-type: none"> <li>▪ MSR_LASTBRANCH_0_FROM_IP (at 40H)</li> <li>▪ Section 17.13, "Last Branch, Interrupt, and Exception Recording (Pentium M Processors)"</li> </ul>
1D9H	473	MSR_DEBUGCTLB	<b>Debug Control (R/W)</b> Controls how several debug features are used. Bit definitions are discussed in the referenced section. See Section 17.13, "Last Branch, Interrupt, and Exception Recording (Pentium M Processors)."
1DDH	477	MSR_LER_TO_LIP	<b>Last Exception Record To Linear IP (R)</b> This area contains a pointer to the target of the last branch instruction that the processor executed prior to the last exception that was generated or the last interrupt that was handled. See Section 17.13, "Last Branch, Interrupt, and Exception Recording (Pentium M Processors)" and Section 17.14.2, "Last Branch and Last Exception MSRs."
1DEH	478	MSR_LER_FROM_LIP	<b>Last Exception Record From Linear IP (R)</b> Contains a pointer to the last branch instruction that the processor executed prior to the last exception that was generated or the last interrupt that was handled. See Section 17.13, "Last Branch, Interrupt, and Exception Recording (Pentium M Processors)" and Section 17.14.2, "Last Branch and Last Exception MSRs."
2FFH	767	IA32_MTRR_DEF_TYPE	<b>Default Memory Types (R/W)</b> Sets the memory type for the regions of physical memory that are not mapped by the MTRRs. See Section 11.11.2.1, "IA32_MTRR_DEF_TYPE MSR."
400H	1024	IA32_MCO_CTL	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
401H	1025	IA32_MCO_STATUS	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
402H	1026	IA32_MCO_ADDR	See Section 14.3.2.3., "IA32_MCi_ADDR MSRs". The IA32_MCO_ADDR register is either not implemented or contains no address if the ADDR_V flag in the IA32_MCO_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.

Table 35-45. MSRs in Pentium M Processors (Contd.)

Register Address		Register Name	Bit Description
Hex	Dec		
404H	1028	IA32_MC1_CTL	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
405H	1029	IA32_MC1_STATUS	See Section 15.3.2.2, "IA32_MCi_STATUS MSRs."
406H	1030	IA32_MC1_ADDR	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The IA32_MC1_ADDR register is either not implemented or contains no address if the ADDR_V flag in the IA32_MC1_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
408H	1032	IA32_MC2_CTL	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
409H	1033	IA32_MC2_STATUS	See Chapter 15.3.2.2, "IA32_MCi_STATUS MSRs."
40AH	1034	IA32_MC2_ADDR	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The IA32_MC2_ADDR register is either not implemented or contains no address if the ADDR_V flag in the IA32_MC2_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
40CH	1036	MSR_MC4_CTL	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
40DH	1037	MSR_MC4_STATUS	See Section 15.3.2.2, "IA32_MCi_STATUS MSRs."
40EH	1038	MSR_MC4_ADDR	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The MSR_MC4_ADDR register is either not implemented or contains no address if the ADDR_V flag in the MSR_MC4_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
410H	1040	MSR_MC3_CTL	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
411H	1041	MSR_MC3_STATUS	See Section 15.3.2.2, "IA32_MCi_STATUS MSRs."
412H	1042	MSR_MC3_ADDR	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The MSR_MC3_ADDR register is either not implemented or contains no address if the ADDR_V flag in the MSR_MC3_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
600H	1536	IA32_DS_AREA	<b>DS Save Area (R/W)</b> See Table 35-2. Points to the DS buffer management area, which is used to manage the BTS and PEBS buffers. See Section 18.12.4, "Debug Store (DS) Mechanism."
		31:0	<b>DS Buffer Management Area</b> Linear address of the first byte of the DS buffer management area.
		63:32	Reserved.

## 35.21 MSRS IN THE P6 FAMILY PROCESSORS

The following MSRs are defined for the P6 family processors. The MSRs in this table that are shaded are available only in the Pentium II and Pentium III processors. Beginning with the Pentium 4 processor, some of the MSRs in this list have been designated as "architectural" and have had their names changed. See Table 35-2 for a list of the architectural MSRs.

**Table 35-46. MSRs in the P6 Family Processors**

Register Address		Register Name	Bit Description
Hex	Dec		
0H	0	P5_MC_ADDR	See Section 35.22, "MSRs in Pentium Processors."
1H	1	P5_MC_TYPE	See Section 35.22, "MSRs in Pentium Processors."
10H	16	TSC	See Section 17.15, "Time-Stamp Counter."
17H	23	IA32_PLATFORM_ID	<b>Platform ID (R)</b> The operating system can use this MSR to determine "slot" information for the processor and the proper microcode update to load.
		49:0	Reserved.
		52:50	<b>Platform Id (R)</b> Contains information concerning the intended platform for the processor. 52 51 50 0 0 0 Processor Flag 0 0 0 1 Processor Flag 1 0 1 0 Processor Flag 2 0 1 1 Processor Flag 3 1 0 0 Processor Flag 4 1 0 1 Processor Flag 5 1 1 0 Processor Flag 6 1 1 1 Processor Flag 7
		56:53	L2 Cache Latency Read.
		59:57	Reserved.
		60	Clock Frequency Ratio Read.
		63:61	Reserved.
		1BH	27
7:0	Reserved.		
8	<b>Boot Strap Processor indicator Bit</b> 1 = BSP		
10:9	Reserved.		
11	<b>APIC Global Enable Bit - Permanent till reset</b> 1 = Enabled 0 = Disabled		
31:12	APIC Base Address.		
63:32	Reserved.		
2AH	42	EBL_CR_POWERON	<b>Processor Hard Power-On Configuration (R/W)</b> Enables and disables processor features; <b>(R)</b> indicates current processor configuration.
		0	Reserved. <sup>1</sup>
		1	<b>Data Error Checking Enable (R/W)</b> 1 = Enabled 0 = Disabled
		2	<b>Response Error Checking Enable FRCERR Observation Enable (R/W)</b> 1 = Enabled 0 = Disabled

Table 35-46. MSRs in the P6 Family Processors (Contd.)

Register Address		Register Name	Bit Description
Hex	Dec		
		3	<b>AERR# Drive Enable (R/W)</b> 1 = Enabled 0 = Disabled
		4	<b>BERR# Enable for Initiator Bus Requests (R/W)</b> 1 = Enabled 0 = Disabled
		5	Reserved.
		6	<b>BERR# Driver Enable for Initiator Internal Errors (R/W)</b> 1 = Enabled 0 = Disabled
		7	<b>BINIT# Driver Enable (R/W)</b> 1 = Enabled 0 = Disabled
		8	<b>Output Tri-state Enabled (R)</b> 1 = Enabled 0 = Disabled
		9	<b>Execute BIST (R)</b> 1 = Enabled 0 = Disabled
		10	<b>AERR# Observation Enabled (R)</b> 1 = Enabled 0 = Disabled
		11	Reserved.
		12	<b>BINIT# Observation Enabled (R)</b> 1 = Enabled 0 = Disabled
		13	<b>In Order Queue Depth (R)</b> 1 = 1 0 = 8
		14	<b>1-MByte Power on Reset Vector (R)</b> 1 = 1MByte 0 = 4GBytes
		15	<b>FRC Mode Enable (R)</b> 1 = Enabled 0 = Disabled
		17:16	<b>APIC Cluster ID (R)</b>
		19:18	<b>System Bus Frequency (R)</b> 00 = 66MHz 10 = 100MHz 01 = 133MHz 11 = Reserved

**Table 35-46. MSRs in the P6 Family Processors (Contd.)**

Register Address		Register Name	Bit Description
Hex	Dec		
		21:20	<b>Symmetric Arbitration ID (R)</b>
		25:22	<b>Clock Frequency Ratio (R)</b>
		26	<b>Low Power Mode Enable (R/W)</b>
		27	<b>Clock Frequency Ratio</b>
		63:28	Reserved. <sup>1</sup>
33H	51	TEST_CTL	<b>Test Control Register</b>
		29:0	Reserved.
		30	<b>Streaming Buffer Disable</b>
		31	<b>Disable LOCK#</b> Assertion for split locked access.
79H	121	BIOS_UPDT_TRIG	BIOS Update Trigger Register.
88H	136	BBL_CR_D0[63:0]	Chunk 0 data register D[63:0]: used to write to and read from the L2
89H	137	BBL_CR_D1[63:0]	Chunk 1 data register D[63:0]: used to write to and read from the L2
8AH	138	BBL_CR_D2[63:0]	Chunk 2 data register D[63:0]: used to write to and read from the L2
8BH	139	BIOS_SIGN/BBL_CR_D3[63:0]	<b>BIOS Update Signature Register or Chunk 3 data register D[63:0]</b> Used to write to and read from the L2 depending on the usage model.
C1H	193	PerfCtr0 (PERFCTR0)	
C2H	194	PerfCtr1 (PERFCTR1)	
FEH	254	MTRRcap	
116H	278	BBL_CR_ADDR [63:0]	Address register: used to send specified address (A31-A3) to L2 during cache initialization accesses.
		BBL_CR_ADDR [63:32]	Reserved,
		BBL_CR_ADDR [31:3]	Address bits [35:3]
		BBL_CR_ADDR [2:0]	Reserved Set to 0.
118H	280	BBL_CR_DECC[63:0]	Data ECC register D[7:0]: used to write ECC and read ECC to/from L2
119H	281	BBL_CR_CTL	Control register: used to program L2 commands to be issued via cache configuration accesses mechanism. Also receives L2 lookup response
		BL_CR_CTL[63:22]	Reserved
		BBL_CR_CTL[21]	Processor number <sup>2</sup> Disable = 1 Enable = 0 Reserved

Table 35-46. MSRs in the P6 Family Processors (Contd.)

Register Address		Register Name	Bit Description
Hex	Dec		
		BBL_CR_CTL[20:19] BBL_CR_CTL[18] BBL_CR_CTL[17] BBL_CR_CTL[16] BBL_CR_CTL[15:14] BBL_CR_CTL[13:12]  BBL_CR_CTL[11:10]  BBL_CR_CTL[9:8] BBL_CR_CTL[7] BBL_CR_CTL[6:5]	User supplied ECC Reserved L2 Hit Reserved State from L2 Modified - 11, Exclusive - 10, Shared - 01, Invalid - 00 Way from L2 Way 0 - 00, Way 1 - 01, Way 2 - 10, Way 3 - 11 Way to L2 Reserved State to L2
		BBL_CR_CTL[4:0] 01100 01110 01111 00010 00011 010 + MESI encode 111 + MESI encode 100 + MESI encode	L2 Command Data Read w/ LRU update (RLU) Tag Read w/ Data Read (TRR) Tag Inquire (TI) L2 Control Register Read (CR) L2 Control Register Write (CW) Tag Write w/ Data Read (TWR) Tag Write w/ Data Write (TWW) Tag Write (TW)
11AH	282	BBL_CR_TRIG	Trigger register: used to initiate a cache configuration accesses access, Write only with Data = 0.
11BH	283	BBL_CR_BUSY	Busy register: indicates when a cache configuration accesses L2 command is in progress. D[0] = 1 = BUSY
11EH	286	BBL_CR_CTL3  BBL_CR_CTL3[63:26] BBL_CR_CTL3[25] BBL_CR_CTL3[24] BBL_CR_CTL3[23]  BBL_CR_CTL3[22:20] 111 110 101 100 011 010 001 000  BBL_CR_CTL3[19] BBL_CR_CTL3[18]	Control register 3: used to configure the L2 Cache  Reserved Cache bus fraction (read only) Reserved L2 Hardware Disable (read only)  L2 Physical Address Range support 64GBytes 32GBytes 16GBytes 8GBytes 4GBytes 2GBytes 1GBytes 512MBytes  Reserved Cache State error checking enable (read/write)

**Table 35-46. MSRs in the P6 Family Processors (Contd.)**

Register Address		Register Name	Bit Description
Hex	Dec		
		BBL_CR_CTL3[17:13] 00001 00010 00100 01000 10000  BBL_CR_CTL3[12:11] BBL_CR_CTL3[10:9] 00 01 10 11  BBL_CR_CTL3[8] BBL_CR_CTL3[7] BBL_CR_CTL3[6] BBL_CR_CTL3[5] BBL_CR_CTL3[4:1] BBL_CR_CTL3[0]	Cache size per bank (read/write) 256KBytes 512KBytes 1MByte 2MByte 4MBytes  Number of L2 banks (read only) L2 Associativity (read only) Direct Mapped 2 Way 4 Way Reserved  L2 Enabled (read/write) CRTN Parity Check Enable (read/write) Address Parity Check Enable (read/write) ECC Check Enable (read/write) L2 Cache Latency (read/write) L2 Configured (read/write )
174H	372	SYSENTER_CS_MSR	CS register target for CPL 0 code
175H	373	SYSENTER_ESP_MSR	Stack pointer for CPL 0 stack
176H	374	SYSENTER_EIP_MSR	CPL 0 code entry point
179H	377	MCG_CAP	
17AH	378	MCG_STATUS	
17BH	379	MCG_CTL	
186H	390	PerfEvtSel0 (EVNTSELO)	
		7:0	<b>Event Select</b> Refer to Performance Counter section for a list of event encodings.
		15:8	<b>UMASK (Unit Mask)</b> Unit mask register set to 0 to enable all count options.
		16	<b>USER</b> Controls the counting of events at Privilege levels of 1, 2, and 3.
		17	<b>OS</b> Controls the counting of events at Privilege level of 0.
		18	<b>E</b> Occurrence/Duration Mode Select 1 = Occurrence 0 = Duration
		19	<b>PC</b> Enabled the signaling of performance counter overflow via BPO pin

Table 35-46. MSRs in the P6 Family Processors (Contd.)

Register Address		Register Name	Bit Description
Hex	Dec		
		20	<b>INT</b> Enables the signaling of counter overflow via input to APIC 1 = Enable 0 = Disable
		22	<b>ENABLE</b> Enables the counting of performance events in both counters 1 = Enable 0 = Disable
		23	<b>INV</b> Inverts the result of the CMASK condition 1 = Inverted 0 = Non-Inverted
		31:24	CMASK (Counter Mask).
187H	391	PerfEvtSel1 (EVNTSEL1)	
		7:0	<b>Event Select</b> Refer to Performance Counter section for a list of event encodings.
		15:8	<b>UMASK (Unit Mask)</b> Unit mask register set to 0 to enable all count options.
		16	<b>USER</b> Controls the counting of events at Privilege levels of 1, 2, and 3.
		17	<b>OS</b> Controls the counting of events at Privilege level of 0
		18	<b>E</b> Occurrence/Duration Mode Select 1 = Occurrence 0 = Duration
		19	<b>PC</b> Enabled the signaling of performance counter overflow via BPO pin.
		20	<b>INT</b> Enables the signaling of counter overflow via input to APIC 1 = Enable 0 = Disable
		23	<b>INV</b> Inverts the result of the CMASK condition 1 = Inverted 0 = Non-Inverted



**Table 35-46. MSRs in the P6 Family Processors (Contd.)**

Register Address		Register Name	Bit Description
Hex	Dec		
		31:24	<b>CMASK (Counter Mask)</b>
1D9H	473	DEBUGCTLSMR	
		0	Enable/Disable Last Branch Records
		1	Branch Trap Flag
		2	Performance Monitoring/Break Point Pins
		3	Performance Monitoring/Break Point Pins
		4	Performance Monitoring/Break Point Pins
		5	Performance Monitoring/Break Point Pins
		6	Enable/Disable Execution Trace Messages
		31:7	Reserved
1DBH	475	LASTBRANCHFROMIP	
1DCH	476	LASTBRANCHTOIP	
1DDH	477	LASTINTFROMIP	
1DEH	478	LASTINTTOIP	
1E0H	480	ROB_CR_BKUPTMPDR6	
		1:0	Reserved
		2	Fast String Enable bit. Default is enabled
200H	512	MTRRphysBase0	
201H	513	MTRRphysMask0	
202H	514	MTRRphysBase1	
203H	515	MTRRphysMask1	
204H	516	MTRRphysBase2	
205H	517	MTRRphysMask2	
206H	518	MTRRphysBase3	
207H	519	MTRRphysMask3	
208H	520	MTRRphysBase4	
209H	521	MTRRphysMask4	
20AH	522	MTRRphysBase5	
20BH	523	MTRRphysMask5	
20CH	524	MTRRphysBase6	
20DH	525	MTRRphysMask6	
20EH	526	MTRRphysBase7	
20FH	527	MTRRphysMask7	
250H	592	MTRRfix64K_00000	
258H	600	MTRRfix16K_80000	
259H	601	MTRRfix16K_A0000	
268H	616	MTRRfix4K_C0000	

Table 35-46. MSRs in the P6 Family Processors (Contd.)

Register Address		Register Name	Bit Description
Hex	Dec		
269H	617	MTRRfix4K_C8000	
26AH	618	MTRRfix4K_D0000	
26BH	619	MTRRfix4K_D8000	
26CH	620	MTRRfix4K_E0000	
26DH	621	MTRRfix4K_E8000	
26EH	622	MTRRfix4K_F0000	
26FH	623	MTRRfix4K_F8000	
2FFH	767	MTRRdefType	
		2:0	Default memory type
		10	Fixed MTRR enable
		11	MTRR Enable
400H	1024	MCO_CTL	
401H	1025	MCO_STATUS	
		15:0	MC_STATUS_MCACOD
		31:16	MC_STATUS_MSCOD
		57	MC_STATUS_DAM
		58	MC_STATUS_ADDRV
		59	MC_STATUS_MISCV
		60	MC_STATUS_EN. (Note: For MCO_STATUS only, this bit is hardcoded to 1.)
		61	MC_STATUS_UC
		62	MC_STATUS_O
		63	MC_STATUS_V
402H	1026	MCO_ADDR	
403H	1027	MCO_MISC	Defined in MCA architecture but not implemented in the P6 family processors.
404H	1028	MC1_CTL	
405H	1029	MC1_STATUS	Bit definitions same as MCO_STATUS.
406H	1030	MC1_ADDR	
407H	1031	MC1_MISC	Defined in MCA architecture but not implemented in the P6 family processors.
408H	1032	MC2_CTL	
409H	1033	MC2_STATUS	Bit definitions same as MCO_STATUS.
40AH	1034	MC2_ADDR	
40BH	1035	MC2_MISC	Defined in MCA architecture but not implemented in the P6 family processors.
40CH	1036	MC4_CTL	
40DH	1037	MC4_STATUS	Bit definitions same as MCO_STATUS, except bits 0, 4, 57, and 61 are hardcoded to 1.

**Table 35-46. MSRs in the P6 Family Processors (Contd.)**

Register Address		Register Name	Bit Description
Hex	Dec		
40EH	1038	MC4_ADDR	Defined in MCA architecture but not implemented in P6 Family processors.
40FH	1039	MC4_MISC	Defined in MCA architecture but not implemented in the P6 family processors.
410H	1040	MC3_CTL	
411H	1041	MC3_STATUS	Bit definitions same as MCO_STATUS.
412H	1042	MC3_ADDR	
413H	1043	MC3_MISC	Defined in MCA architecture but not implemented in the P6 family processors.

**NOTES**

1. Bit 0 of this register has been redefined several times, and is no longer used in P6 family processors.
2. The processor number feature may be disabled by setting bit 21 of the BBL\_CR\_CTL MSR (model-specific register address 119h) to "1". Once set, bit 21 of the BBL\_CR\_CTL may not be cleared. This bit is write-once. The processor number feature will be disabled until the processor is reset.
3. The Pentium III processor will prevent FSB frequency overclocking with a new shutdown mechanism. If the FSB frequency selected is greater than the internal FSB frequency the processor will shutdown. If the FSB selected is less than the internal FSB frequency the BIOS may choose to use bit 11 to implement its own shutdown policy.

## 35.22 MSRS IN PENTIUM PROCESSORS

The following MSRs are defined for the Pentium processors. The P5\_MC\_ADDR, P5\_MC\_TYPE, and TSC MSRs (named IA32\_P5\_MC\_ADDR, IA32\_P5\_MC\_TYPE, and IA32\_TIME\_STAMP\_COUNTER in the Pentium 4 processor) are architectural; that is, code that accesses these registers will run on Pentium 4 and P6 family processors without generating exceptions (see Section 35.1, "Architectural MSRs"). The CESR, CTR0, and CTR1 MSRs are unique to Pentium processors; code that accesses these registers will generate exceptions on Pentium 4 and P6 family processors.

**Table 35-47. MSRs in the Pentium Processor**

Register Address		Register Name	Bit Description
Hex	Dec		
0H	0	P5_MC_ADDR	See Section 15.10.2, "Pentium Processor Machine-Check Exception Handling."
1H	1	P5_MC_TYPE	See Section 15.10.2, "Pentium Processor Machine-Check Exception Handling."
10H	16	TSC	See Section 17.15, "Time-Stamp Counter."
11H	17	CESR	See Section 18.20.1, "Control and Event Select Register (CESR)."
12H	18	CTR0	Section 18.20.3, "Events Counted."
13H	19	CTR1	Section 18.20.3, "Events Counted."

## 35.23 MSR INDEX

MSRs of recent processors are indexed here for convenience. IA32 MSRs are excluded from this index.

<b>MSR Name and CPUID DisplayFamily_DisplayModel</b>	<b>Location</b>
MSR_ALF_ESCR0 0FH .....	See Table 35-41
MSR_ALF_ESCR1 0FH .....	See Table 35-41
MSR_ANY_CORE_C0 06_4EH, 06_5EH .....	See Table 35-37
MSR_ANY_GFXE_C0 06_4EH, 06_5EH .....	See Table 35-37
MSR_BO_PMON_BOX_CTRL 06_2EH .....	See Table 35-15
MSR_BO_PMON_BOX_OVF_CTRL 06_2EH .....	See Table 35-15
MSR_BO_PMON_BOX_STATUS 06_2EH .....	See Table 35-15
MSR_BO_PMON_CTRL0 06_2EH .....	See Table 35-15
MSR_BO_PMON_CTRL1 06_2EH .....	See Table 35-15
MSR_BO_PMON_CTRL2 06_2EH .....	See Table 35-15
MSR_BO_PMON_CTRL3 06_2EH .....	See Table 35-15
MSR_BO_PMON_EVNT_SELO 06_2EH .....	See Table 35-15
MSR_BO_PMON_EVNT_SEL1 06_2EH .....	See Table 35-15
MSR_BO_PMON_EVNT_SEL2 06_2EH .....	See Table 35-15
MSR_BO_PMON_EVNT_SEL3 06_2EH .....	See Table 35-15
MSR_BO_PMON_MASK 06_2EH .....	See Table 35-15
MSR_BO_PMON_MATCH 06_2EH .....	See Table 35-15
MSR_B1_PMON_BOX_CTRL 06_2EH .....	See Table 35-15
MSR_B1_PMON_BOX_OVF_CTRL 06_2EH .....	See Table 35-15
MSR_B1_PMON_BOX_STATUS 06_2EH .....	See Table 35-15
MSR_B1_PMON_CTRL0	

## MODEL-SPECIFIC REGISTERS (MSRS)

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_2EH .....	See Table 35-15
MSR_B1_PMON_CTR1	
06_2EH .....	See Table 35-15
MSR_B1_PMON_CTR2	
06_2EH .....	See Table 35-15
MSR_B1_PMON_CTR3	
06_2EH .....	See Table 35-15
MSR_B1_PMON_EVNT_SELO	
06_2EH .....	See Table 35-15
MSR_B1_PMON_EVNT_SEL1	
06_2EH .....	See Table 35-15
MSR_B1_PMON_EVNT_SEL2	
06_2EH .....	See Table 35-15
MSR_B1_PMON_EVNT_SEL3	
06_2EH .....	See Table 35-15
MSR_B1_PMON_MASK	
06_2EH .....	See Table 35-15
MSR_B1_PMON_MATCH	
06_2EH .....	See Table 35-15
MSR_BBL_CR_CTL	
06_09H .....	See Table 35-45
MSR_BBL_CR_CTL3	
06_0FH, 06_17H .....	See Table 35-3
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H .....	See Table 35-4
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH .....	See Table 35-7
06_0EH .....	See Table 35-44
06_09H .....	See Table 35-45
MSR_BPU_CCCR0	
0FH .....	See Table 35-41
MSR_BPU_CCCR1	
0FH .....	See Table 35-41
MSR_BPU_CCCR2	
0FH .....	See Table 35-41
MSR_BPU_CCCR3	
0FH .....	See Table 35-41
MSR_BPU_COUNTER0	
0FH .....	See Table 35-41
MSR_BPU_COUNTER1	
0FH .....	See Table 35-41
MSR_BPU_COUNTER2	
0FH .....	See Table 35-41
MSR_BPU_COUNTER3	

<b>MSR Name and CPUID DisplayFamily_DisplayModel</b>	<b>Location</b>
0FH .....	See Table 35-41
MSR_BPU_ESCR0	
0FH .....	See Table 35-41
MSR_BPU_ESCR1	
0FH .....	See Table 35-41
MSR_BSU_ESCR0	
0FH .....	See Table 35-41
MSR_BSU_ESCR1	
0FH .....	See Table 35-41
MSR_CO_PMON_BOX_CTRL	
06_2EH .....	See Table 35-15
06_2DH .....	See Table 35-22
06_3FH .....	See Table 35-31
MSR_CO_PMON_BOX_FILTER	
06_2DH .....	See Table 35-22
MSR_CO_PMON_BOX_FILTER0	
06_3FH .....	See Table 35-31
MSR_CO_PMON_BOX_FILTER1	
06_3EH .....	See Table 35-26
06_3FH .....	See Table 35-31
MSR_CO_PMON_BOX_OVF_CTRL	
06_2EH .....	See Table 35-15
MSR_CO_PMON_BOX_STATUS	
06_2EH .....	See Table 35-15
06_3FH .....	See Table 35-31
MSR_CO_PMON_CTRL0	
06_2EH .....	See Table 35-15
06_2DH .....	See Table 35-22
06_3FH .....	See Table 35-31
MSR_CO_PMON_CTRL1	
06_2EH .....	See Table 35-15
06_2DH .....	See Table 35-22
06_3FH .....	See Table 35-31
MSR_CO_PMON_CTRL2	
06_2EH .....	See Table 35-15
06_2DH .....	See Table 35-22
06_3FH .....	See Table 35-31
MSR_CO_PMON_CTRL3	
06_2EH .....	See Table 35-15
06_2DH .....	See Table 35-22
06_3FH .....	See Table 35-31
MSR_CO_PMON_CTRL4	

## MODEL-SPECIFIC REGISTERS (MSRS)

<b>MSR Name and CPUID DisplayFamily_DisplayModel</b>	<b>Location</b>
06_2EH .....	See Table 35-15
MSR_CO_PMON_CTR5	
06_2EH .....	See Table 35-15
MSR_CO_PMON_EVNT_SELO	
06_2EH .....	See Table 35-15
06_2DH .....	See Table 35-22
06_3FH .....	See Table 35-31
MSR_CO_PMON_EVNT_SEL1	
06_2EH .....	See Table 35-15
06_2DH .....	See Table 35-22
06_3FH .....	See Table 35-31
MSR_CO_PMON_CTR1	
06_2EH .....	See Table 35-15
06_2DH .....	See Table 35-22
06_3FH .....	See Table 35-31
MSR_CO_PMON_EVNT_SEL2	
06_2EH .....	See Table 35-15
06_2DH .....	See Table 35-22
06_3FH .....	See Table 35-31
MSR_CO_PMON_CTR2	
06_2EH .....	See Table 35-15
06_2DH .....	See Table 35-22
06_3FH .....	See Table 35-31
MSR_CO_PMON_EVNT_SEL3	
06_2EH .....	See Table 35-15
06_2DH .....	See Table 35-22
06_3FH .....	See Table 35-31
MSR_CO_PMON_EVNT_SEL4	
06_2EH .....	See Table 35-15
MSR_CO_PMON_EVNT_SEL5	
06_2EH .....	See Table 35-15
MSR_C1_PMON_BOX_CTRL	
06_2EH .....	See Table 35-15
06_2DH .....	See Table 35-22
06_3FH .....	See Table 35-31
MSR_C1_PMON_BOX_FILTER	
06_2DH .....	See Table 35-22
MSR_C1_PMON_BOX_FILTER0	
06_3FH .....	See Table 35-31
MSR_C1_PMON_BOX_FILTER1	
06_3EH .....	See Table 35-26
06_3FH .....	See Table 35-31

<b>MSR Name and CPUID DisplayFamily_DisplayModel</b>	<b>Location</b>
MSR_C1_PMON_BOX_OVF_CTRL	
06_2EH .....	See Table 35-15
MSR_C1_PMON_BOX_STATUS	
06_2EH .....	See Table 35-15
06_3FH .....	See Table 35-31
MSR_C1_PMON_CTRL0	
06_2EH .....	See Table 35-15
06_2DH .....	See Table 35-22
06_3FH .....	See Table 35-31
MSR_C1_PMON_CTRL1	
06_2EH .....	See Table 35-15
06_2DH .....	See Table 35-22
06_3FH .....	See Table 35-31
MSR_C1_PMON_CTRL2	
06_2EH .....	See Table 35-15
06_2DH .....	See Table 35-22
06_3FH .....	See Table 35-31
MSR_C1_PMON_CTRL3	
06_2EH .....	See Table 35-15
06_2DH .....	See Table 35-22
06_3FH .....	See Table 35-31
MSR_C1_PMON_CTRL4	
06_2EH .....	See Table 35-15
MSR_C1_PMON_CTRL5	
06_2EH .....	See Table 35-15
MSR_C1_PMON_EVNT_SELO	
06_2EH .....	See Table 35-15
06_2DH .....	See Table 35-22
06_3FH .....	See Table 35-31
MSR_C1_PMON_EVNT_SEL1	
06_2EH .....	See Table 35-15
06_2DH .....	See Table 35-22
06_3FH .....	See Table 35-31
MSR_C1_PMON_EVNT_SEL2	
06_2EH .....	See Table 35-15
06_2DH .....	See Table 35-22
06_3FH .....	See Table 35-31
MSR_C1_PMON_EVNT_SEL3	
06_2EH .....	See Table 35-15
06_2DH .....	See Table 35-22
06_3FH .....	See Table 35-31
MSR_C1_PMON_EVNT_SEL4	



## MODEL-SPECIFIC REGISTERS (MSRS)

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_2EH .....	See Table 35-15
MSR_C1_PMON_EVT_SEL5	
06_2EH .....	See Table 35-15
MSR_C10_PMON_BOX_FILTER	
06_3EH .....	See Table 35-26
MSR_C10_PMON_BOX_FILTER0	
06_3FH .....	See Table 35-31
MSR_C10_PMON_BOX_FILTER1	
06_3EH .....	See Table 35-26
06_3FH .....	See Table 35-31
MSR_C11_PMON_BOX_FILTER	
06_3EH .....	See Table 35-26
MSR_C11_PMON_BOX_FILTER0	
06_3FH .....	See Table 35-31
MSR_C11_PMON_BOX_FILTER1	
06_3EH .....	See Table 35-26
06_3FH .....	See Table 35-31
MSR_C12_PMON_BOX_FILTER	
06_3EH .....	See Table 35-26
MSR_C12_PMON_BOX_FILTER0	
06_3FH .....	See Table 35-31
MSR_C12_PMON_BOX_FILTER1	
06_3EH .....	See Table 35-26
06_3FH .....	See Table 35-31
MSR_C13_PMON_BOX_FILTER	
06_3EH .....	See Table 35-26
MSR_C13_PMON_BOX_FILTER0	
06_3FH .....	See Table 35-31
MSR_C13_PMON_BOX_FILTER1	
06_3EH .....	See Table 35-26
06_3FH .....	See Table 35-31
MSR_C14_PMON_BOX_FILTER	
06_3EH .....	See Table 35-26
MSR_C14_PMON_BOX_FILTER0	
06_3FH .....	See Table 35-31
MSR_C14_PMON_BOX_FILTER1	
06_3EH .....	See Table 35-26
06_3FH .....	See Table 35-31
MSR_C15_PMON_BOX_CTL	
06_3FH .....	See Table 35-31
MSR_C15_PMON_BOX_FILTER0	
06_3FH .....	See Table 35-31

<b>MSR Name and CPUID DisplayFamily_DisplayModel</b>	<b>Location</b>
MSR_C15_PMON_BOX_FILTER1 06_3FH .....	See Table 35-31
MSR_C15_PMON_BOX_STATUS 06_3FH .....	See Table 35-31
MSR_C15_PMON_CTRL0 06_3FH .....	See Table 35-31
MSR_C15_PMON_CTRL1 06_3FH .....	See Table 35-31
MSR_C15_PMON_CTRL2 06_3FH .....	See Table 35-31
MSR_C15_PMON_CTRL3 06_3FH .....	See Table 35-31
MSR_C15_PMON_EVNTSELO 06_3FH .....	See Table 35-31
MSR_C15_PMON_EVNTSEL1 06_3FH .....	See Table 35-31
MSR_C15_PMON_EVNTSEL2 06_3FH .....	See Table 35-31
MSR_C15_PMON_EVNTSEL3 06_3FH .....	See Table 35-31
MSR_C16_PMON_BOX_CTL 06_3FH .....	See Table 35-31
MSR_C16_PMON_BOX_FILTER0 06_3FH .....	See Table 35-31
MSR_C16_PMON_BOX_FILTER1 06_3FH .....	See Table 35-31
MSR_C16_PMON_BOX_STATUS 06_3FH .....	See Table 35-31
MSR_C16_PMON_CTRL0 06_3FH .....	See Table 35-31
MSR_C16_PMON_CTRL3 06_3FH .....	See Table 35-31
MSR_C16_PMON_CTRL2 06_3FH .....	See Table 35-31
MSR_C16_PMON_CTRL3 06_3FH .....	See Table 35-31
MSR_C16_PMON_EVNTSELO 06_3FH .....	See Table 35-31
MSR_C16_PMON_EVNTSEL1 06_3FH .....	See Table 35-31
MSR_C16_PMON_EVNTSEL2 06_3FH .....	See Table 35-31

## MODEL-SPECIFIC REGISTERS (MSRS)

<b>MSR Name and CPUID DisplayFamily_DisplayModel</b>	<b>Location</b>
MSR_C16_PMON_EVNTSEL3 06_3FH .....	See Table 35-31
MSR_C17_PMON_BOX_CTL 06_3FH .....	See Table 35-31
MSR_C17_PMON_BOX_FILTER0 06_3FH .....	See Table 35-31
MSR_C17_PMON_BOX_FILTER1 06_3FH .....	See Table 35-31
MSR_C17_PMON_BOX_STATUS 06_3FH .....	See Table 35-31
MSR_C17_PMON_CTRL0 06_3FH .....	See Table 35-31
MSR_C17_PMON_CTRL1 06_3FH .....	See Table 35-31
MSR_C17_PMON_CTRL2 06_3FH .....	See Table 35-31
MSR_C17_PMON_CTRL3 06_3FH .....	See Table 35-31
MSR_C17_PMON_EVNTSELO 06_3FH .....	See Table 35-31
MSR_C17_PMON_EVNTSEL1 06_3FH .....	See Table 35-31
MSR_C17_PMON_EVNTSEL2 06_3FH .....	See Table 35-31
MSR_C17_PMON_EVNTSEL3 06_3FH .....	See Table 35-31
MSR_C2_PMON_BOX_CTRL 06_2EH .....	See Table 35-15
06_2DH .....	See Table 35-22
06_3FH .....	See Table 35-31
MSR_C2_PMON_BOX_FILTER 06_2DH .....	See Table 35-22
MSR_C2_PMON_BOX_FILTER0 06_3FH .....	See Table 35-31
MSR_C2_PMON_BOX_FILTER1 06_3EH .....	See Table 35-26
06_3FH .....	See Table 35-31
MSR_C2_PMON_BOX_OVF_CTRL 06_2EH .....	See Table 35-15
MSR_C2_PMON_BOX_STATUS 06_2EH .....	See Table 35-15
06_3FH .....	See Table 35-31

<b>MSR Name and CPUID DisplayFamily_DisplayModel</b>	<b>Location</b>
<b>MSR_C2_PMON_CTRL0</b>	
06_2EH .....	See Table 35-15
06_2DH .....	See Table 35-22
06_3FH .....	See Table 35-31
<b>MSR_C2_PMON_CTRL1</b>	
06_2EH .....	See Table 35-15
06_2DH .....	See Table 35-22
06_3FH .....	See Table 35-31
<b>MSR_C2_PMON_CTRL2</b>	
06_2EH .....	See Table 35-15
06_2DH .....	See Table 35-22
06_3FH .....	See Table 35-31
<b>MSR_C2_PMON_CTRL3</b>	
06_2EH .....	See Table 35-15
06_2DH .....	See Table 35-22
06_3FH .....	See Table 35-31
<b>MSR_C2_PMON_CTRL4</b>	
06_2EH .....	See Table 35-15
<b>MSR_C2_PMON_CTRL5</b>	
06_2EH .....	See Table 35-15
<b>MSR_C2_PMON_EVNT_SELO</b>	
06_2EH .....	See Table 35-15
06_2DH .....	See Table 35-22
06_3FH .....	See Table 35-31
<b>MSR_C2_PMON_EVNT_SEL1</b>	
06_2EH .....	See Table 35-15
06_2DH .....	See Table 35-22
06_3FH .....	See Table 35-31
<b>MSR_C2_PMON_EVNT_SEL2</b>	
06_2EH .....	See Table 35-15
06_2DH .....	See Table 35-22
06_3FH .....	See Table 35-31
<b>MSR_C2_PMON_EVNT_SEL3</b>	
06_2EH .....	See Table 35-15
06_2DH .....	See Table 35-22
06_3FH .....	See Table 35-31
<b>MSR_C2_PMON_EVNT_SEL4</b>	
06_2EH .....	See Table 35-15
<b>MSR_C2_PMON_EVNT_SEL5</b>	
06_2EH .....	See Table 35-15
<b>MSR_C3_PMON_BOX_CTRL</b>	
06_2EH .....	See Table 35-15

## MODEL-SPECIFIC REGISTERS (MSRS)

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_2DH .....	See Table 35-22
06_3FH .....	See Table 35-31
MSR_C3_PMON_BOX_FILTER	
06_2DH .....	See Table 35-22
MSR_C3_PMON_BOX_FILTER0	
06_3FH .....	See Table 35-31
MSR_C3_PMON_BOX_FILTER1	
06_3EH .....	See Table 35-26
06_3FH .....	See Table 35-31
MSR_C3_PMON_BOX_OVF_CTRL	
06_2EH .....	See Table 35-15
MSR_C3_PMON_BOX_STATUS	
06_2EH .....	See Table 35-15
06_3FH .....	See Table 35-31
MSR_C3_PMON_CTRL0	
06_2EH .....	See Table 35-15
06_2DH .....	See Table 35-22
06_3FH .....	See Table 35-31
MSR_C3_PMON_CTRL1	
06_2EH .....	See Table 35-15
06_2DH .....	See Table 35-22
06_3FH .....	See Table 35-31
MSR_C3_PMON_CTRL2	
06_2EH .....	See Table 35-15
06_2DH .....	See Table 35-22
06_3FH .....	See Table 35-31
MSR_C3_PMON_CTRL3	
06_2EH .....	See Table 35-15
06_2DH .....	See Table 35-22
06_3FH .....	See Table 35-31
MSR_C3_PMON_CTRL4	
06_2EH .....	See Table 35-15
MSR_C3_PMON_CTRL5	
06_2EH .....	See Table 35-15
MSR_C3_PMON_EVNT_SEL0	
06_2EH .....	See Table 35-15
06_2DH .....	See Table 35-22
06_3FH .....	See Table 35-31
MSR_C3_PMON_EVNT_SEL1	
06_2EH .....	See Table 35-15
06_2DH .....	See Table 35-22
06_3FH .....	See Table 35-31

<b>MSR Name and CPUID DisplayFamily_DisplayModel</b>	<b>Location</b>
MSR_C3_PMON_EVNT_SEL2	
06_2EH .....	See Table 35-15
06_2DH .....	See Table 35-22
06_3FH .....	See Table 35-31
MSR_C3_PMON_EVNT_SEL3	
06_2EH .....	See Table 35-15
06_2DH .....	See Table 35-22
06_3FH .....	See Table 35-31
MSR_C3_PMON_EVNT_SEL4	
06_2EH .....	See Table 35-15
MSR_C3_PMON_EVNT_SEL5	
06_2EH .....	See Table 35-15
MSR_C4_PMON_BOX_CTRL	
06_2EH .....	See Table 35-15
06_2DH .....	See Table 35-22
06_3FH .....	See Table 35-31
MSR_C4_PMON_BOX_FILTER	
06_2DH .....	See Table 35-22
MSR_C4_PMON_BOX_FILTER0	
06_3FH .....	See Table 35-31
MSR_C4_PMON_BOX_FILTER1	
06_3EH .....	See Table 35-26
06_3FH .....	See Table 35-31
MSR_C4_PMON_BOX_OVF_CTRL	
06_2EH .....	See Table 35-15
MSR_C4_PMON_BOX_STATUS	
06_2EH .....	See Table 35-15
06_3FH .....	See Table 35-31
MSR_C4_PMON_CTRL0	
06_2EH .....	See Table 35-15
06_2DH .....	See Table 35-22
06_3FH .....	See Table 35-31
MSR_C4_PMON_CTRL1	
06_2EH .....	See Table 35-15
06_2DH .....	See Table 35-22
06_3FH .....	See Table 35-31
MSR_C4_PMON_CTRL2	
06_2EH .....	See Table 35-15
06_2DH .....	See Table 35-22
06_3FH .....	See Table 35-31
MSR_C4_PMON_CTRL3	
06_2EH .....	See Table 35-15

## MODEL-SPECIFIC REGISTERS (MSRS)

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_2DH .....	See Table 35-22
06_3FH .....	See Table 35-31
MSR_C4_PMON_CTR4	
06_2EH .....	See Table 35-15
MSR_C4_PMON_CTR5	
06_2EH .....	See Table 35-15
MSR_C4_PMON_EVNT_SEL0	
06_2EH .....	See Table 35-15
06_2DH .....	See Table 35-22
06_3FH .....	See Table 35-31
MSR_C4_PMON_EVNT_SEL1	
06_2EH .....	See Table 35-15
06_2DH .....	See Table 35-22
06_3FH .....	See Table 35-31
MSR_C4_PMON_EVNT_SEL2	
06_2EH .....	See Table 35-15
06_2DH .....	See Table 35-22
06_3FH .....	See Table 35-31
MSR_C4_PMON_EVNT_SEL3	
06_2EH .....	See Table 35-15
06_2DH .....	See Table 35-22
06_3FH .....	See Table 35-31
MSR_C4_PMON_EVNT_SEL4	
06_2EH .....	See Table 35-15
MSR_C4_PMON_EVNT_SEL5	
06_2EH .....	See Table 35-15
MSR_C5_PMON_BOX_CTRL	
06_2EH .....	See Table 35-15
06_2DH .....	See Table 35-22
06_3FH .....	See Table 35-31
MSR_C5_PMON_BOX_FILTER	
06_2DH .....	See Table 35-22
MSR_C5_PMON_BOX_FILTER0	
06_3FH .....	See Table 35-31
MSR_C5_PMON_BOX_FILTER1	
06_3EH .....	See Table 35-26
06_3FH .....	See Table 35-31
MSR_C5_PMON_BOX_OVF_CTRL	
06_2EH .....	See Table 35-15
MSR_C5_PMON_BOX_STATUS	
06_2EH .....	See Table 35-15
06_3FH .....	See Table 35-31

<b>MSR Name and CPUID DisplayFamily_DisplayModel</b>	<b>Location</b>
<b>MSR_C5_PMON_CTRL0</b>	
06_2EH .....	See Table 35-15
06_2DH .....	See Table 35-22
06_3FH .....	See Table 35-31
<b>MSR_C5_PMON_CTRL1</b>	
06_2EH .....	See Table 35-15
06_2DH .....	See Table 35-22
06_3FH .....	See Table 35-31
<b>MSR_C5_PMON_CTRL2</b>	
06_2EH .....	See Table 35-15
06_2DH .....	See Table 35-22
06_3FH .....	See Table 35-31
<b>MSR_C5_PMON_CTRL3</b>	
06_2EH .....	See Table 35-15
06_2DH .....	See Table 35-22
06_3FH .....	See Table 35-31
<b>MSR_C5_PMON_CTRL4</b>	
06_2EH .....	See Table 35-15
<b>MSR_C5_PMON_CTRL5</b>	
06_2EH .....	See Table 35-15
<b>MSR_C5_PMON_EVNT_SELO</b>	
06_2EH .....	See Table 35-15
06_2DH .....	See Table 35-22
06_3FH .....	See Table 35-31
<b>MSR_C5_PMON_EVNT_SEL1</b>	
06_2EH .....	See Table 35-15
06_2DH .....	See Table 35-22
06_3FH .....	See Table 35-31
<b>MSR_C5_PMON_EVNT_SEL2</b>	
06_2EH .....	See Table 35-15
06_2DH .....	See Table 35-22
06_3FH .....	See Table 35-31
<b>MSR_C5_PMON_EVNT_SEL3</b>	
06_2EH .....	See Table 35-15
06_2DH .....	See Table 35-22
06_3FH .....	See Table 35-31
<b>MSR_C5_PMON_EVNT_SEL4</b>	
06_2EH .....	See Table 35-15
<b>MSR_C5_PMON_EVNT_SEL5</b>	
06_2EH .....	See Table 35-15
<b>MSR_C6_PMON_BOX_CTRL</b>	
06_2EH .....	See Table 35-15



## MODEL-SPECIFIC REGISTERS (MSRS)

<b>MSR Name and CPUID DisplayFamily_DisplayModel</b>	<b>Location</b>
06_2DH .....	See Table 35-22
06_3FH .....	See Table 35-31
<b>MSR_C6_PMON_BOX_FILTER</b>	
06_2DH .....	See Table 35-22
<b>MSR_C6_PMON_BOX_FILTER0</b>	
06_3FH .....	See Table 35-31
<b>MSR_C6_PMON_BOX_FILTER1</b>	
06_3EH .....	See Table 35-26
06_3FH .....	See Table 35-31
<b>MSR_C6_PMON_BOX_OVF_CTRL</b>	
06_2EH .....	See Table 35-15
<b>MSR_C6_PMON_BOX_STATUS</b>	
06_2EH .....	See Table 35-15
06_3FH .....	See Table 35-31
<b>MSR_C6_PMON_CTRL0</b>	
06_2EH .....	See Table 35-15
06_2DH .....	See Table 35-22
06_3FH .....	See Table 35-31
<b>MSR_C6_PMON_CTRL1</b>	
06_2EH .....	See Table 35-15
06_2DH .....	See Table 35-22
06_3FH .....	See Table 35-31
<b>MSR_C6_PMON_CTRL2</b>	
06_2EH .....	See Table 35-15
06_2DH .....	See Table 35-22
06_3FH .....	See Table 35-31
<b>MSR_C6_PMON_CTRL3</b>	
06_2EH .....	See Table 35-15
06_2DH .....	See Table 35-22
06_3FH .....	See Table 35-31
<b>MSR_C6_PMON_CTRL4</b>	
06_2EH .....	See Table 35-15
<b>MSR_C6_PMON_CTRL5</b>	
06_2EH .....	See Table 35-15
<b>MSR_C6_PMON_EVNT_SEL0</b>	
06_2EH .....	See Table 35-15
06_2DH .....	See Table 35-22
06_3FH .....	See Table 35-31
<b>MSR_C6_PMON_EVNT_SEL1</b>	
06_2EH .....	See Table 35-15
06_2DH .....	See Table 35-22
06_3FH .....	See Table 35-31

<b>MSR Name and CPUID DisplayFamily_DisplayModel</b>	<b>Location</b>
MSR_C6_PMON_EVNT_SEL2	
06_2EH .....	See Table 35-15
06_2DH .....	See Table 35-22
06_3FH .....	See Table 35-31
MSR_C6_PMON_EVNT_SEL3	
06_2EH .....	See Table 35-15
06_2DH .....	See Table 35-22
06_3FH .....	See Table 35-31
MSR_C6_PMON_EVNT_SEL4	
06_2EH .....	See Table 35-15
MSR_C6_PMON_EVNT_SEL5	
06_2EH .....	See Table 35-15
MSR_C7_PMON_BOX_CTRL	
06_2EH .....	See Table 35-15
06_2DH .....	See Table 35-22
06_3FH .....	See Table 35-31
MSR_C7_PMON_BOX_FILTER	
06_2DH .....	See Table 35-22
MSR_C7_PMON_BOX_FILTER0	
06_3FH .....	See Table 35-31
MSR_C7_PMON_BOX_FILTER1	
06_3EH .....	See Table 35-26
06_3FH .....	See Table 35-31
MSR_C7_PMON_BOX_OVF_CTRL	
06_2EH .....	See Table 35-15
MSR_C7_PMON_BOX_STATUS	
06_2EH .....	See Table 35-15
06_3FH .....	See Table 35-31
MSR_C7_PMON_CTRL0	
06_2EH .....	See Table 35-15
06_2DH .....	See Table 35-22
06_3FH .....	See Table 35-31
MSR_C7_PMON_CTRL1	
06_2EH .....	See Table 35-15
06_2DH .....	See Table 35-22
06_3FH .....	See Table 35-31
MSR_C7_PMON_CTRL2	
06_2EH .....	See Table 35-15
06_2DH .....	See Table 35-22
06_3FH .....	See Table 35-31
MSR_C7_PMON_CTRL3	
06_2EH .....	See Table 35-15

## MODEL-SPECIFIC REGISTERS (MSRS)

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_2DH .....	See Table 35-22
06_3FH .....	See Table 35-31
MSR_C7_PMON_CTRL4	
06_2EH .....	See Table 35-15
MSR_C7_PMON_CTRL5	
06_2EH .....	See Table 35-15
MSR_C7_PMON_EVTN_SEL0	
06_2EH .....	See Table 35-15
06_2DH .....	See Table 35-22
06_3FH .....	See Table 35-31
MSR_C7_PMON_EVTN_SEL1	
06_2EH .....	See Table 35-15
06_2DH .....	See Table 35-22
06_3FH .....	See Table 35-31
MSR_C7_PMON_EVTN_SEL2	
06_2EH .....	See Table 35-15
06_2DH .....	See Table 35-22
06_3FH .....	See Table 35-31
MSR_C7_PMON_EVTN_SEL3	
06_2EH .....	See Table 35-15
06_2DH .....	See Table 35-22
06_3FH .....	See Table 35-31
MSR_C7_PMON_EVTN_SEL4	
06_2EH .....	See Table 35-15
MSR_C7_PMON_EVTN_SEL5	
06_2EH .....	See Table 35-15
MSR_C8_PMON_BOX_CTRL	
06_2FH .....	See Table 35-17
06_3EH .....	See Table 35-26
06_3FH .....	See Table 35-31
MSR_C8_PMON_BOX_FILTER	
06_3EH .....	See Table 35-26
MSR_C8_PMON_BOX_FILTER0	
06_3FH .....	See Table 35-31
MSR_C8_PMON_BOX_FILTER1	
06_3EH .....	See Table 35-26
06_3FH .....	See Table 35-31
MSR_C8_PMON_BOX_OVF_CTRL	
06_2FH .....	See Table 35-17
MSR_C8_PMON_BOX_STATUS	
06_2FH .....	See Table 35-17
06_3FH .....	See Table 35-31

<b>MSR Name and CPUID DisplayFamily_DisplayModel</b>	<b>Location</b>
<b>MSR_C8_PMON_CTRL0</b>	
06_2FH .....	See Table 35-17
06_3EH .....	See Table 35-26
06_3FH .....	See Table 35-31
<b>MSR_C8_PMON_CTRL1</b>	
06_2FH .....	See Table 35-17
06_3EH .....	See Table 35-26
06_3FH .....	See Table 35-31
<b>MSR_C8_PMON_CTRL2</b>	
06_2FH .....	See Table 35-17
06_3EH .....	See Table 35-26
06_3FH .....	See Table 35-31
<b>MSR_C8_PMON_CTRL3</b>	
06_2FH .....	See Table 35-17
06_3EH .....	See Table 35-26
06_3FH .....	See Table 35-31
<b>MSR_C8_PMON_CTRL4</b>	
06_2FH .....	See Table 35-17
<b>MSR_C8_PMON_CTRL5</b>	
06_2FH .....	See Table 35-17
<b>MSR_C8_PMON_EVNT_SELO</b>	
06_2FH .....	See Table 35-17
06_3EH .....	See Table 35-26
06_3FH .....	See Table 35-31
<b>MSR_C8_PMON_EVNT_SEL1</b>	
06_2FH .....	See Table 35-17
06_3EH .....	See Table 35-26
06_3FH .....	See Table 35-31
<b>MSR_C8_PMON_EVNT_SEL2</b>	
06_2FH .....	See Table 35-17
06_3EH .....	See Table 35-26
06_3FH .....	See Table 35-31
<b>MSR_C8_PMON_EVNT_SEL3</b>	
06_2FH .....	See Table 35-17
06_3EH .....	See Table 35-26
06_3FH .....	See Table 35-31
<b>MSR_C8_PMON_EVNT_SEL4</b>	
06_2FH .....	See Table 35-17
<b>MSR_C8_PMON_EVNT_SEL5</b>	
06_2FH .....	See Table 35-17
<b>MSR_C9_PMON_BOX_CTRL</b>	
06_2FH .....	See Table 35-17

## MODEL-SPECIFIC REGISTERS (MSRS)

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_3EH .....	See Table 35-26
06_3FH .....	See Table 35-31
MSR_C9_PMON_BOX_FILTER	
06_3EH .....	See Table 35-26
MSR_C9_PMON_BOX_FILTER0	
06_3FH .....	See Table 35-31
MSR_C9_PMON_BOX_FILTER1	
06_3EH .....	See Table 35-26
06_3FH .....	See Table 35-31
MSR_C9_PMON_BOX_OVF_CTRL	
06_2FH .....	See Table 35-17
MSR_C9_PMON_BOX_STATUS	
06_2FH .....	See Table 35-17
06_3FH .....	See Table 35-31
MSR_C9_PMON_CTRL0	
06_2FH .....	See Table 35-17
06_3EH .....	See Table 35-26
06_3FH .....	See Table 35-31
MSR_C9_PMON_CTRL1	
06_2FH .....	See Table 35-17
06_3EH .....	See Table 35-26
06_3FH .....	See Table 35-31
MSR_C9_PMON_CTRL2	
06_2FH .....	See Table 35-17
06_3EH .....	See Table 35-26
06_3FH .....	See Table 35-31
MSR_C9_PMON_CTRL3	
06_2FH .....	See Table 35-17
06_3EH .....	See Table 35-26
06_3FH .....	See Table 35-31
MSR_C9_PMON_CTRL4	
06_2FH .....	See Table 35-17
MSR_C9_PMON_CTRL5	
06_2FH .....	See Table 35-17
MSR_C9_PMON_EVNT_SEL0	
06_2FH .....	See Table 35-17
06_3EH .....	See Table 35-26
06_3FH .....	See Table 35-31
MSR_C9_PMON_EVNT_SEL1	
06_2FH .....	See Table 35-17
06_3EH .....	See Table 35-26
06_3FH .....	See Table 35-31

<b>MSR Name and CPUID DisplayFamily_DisplayModel</b>	<b>Location</b>
MSR_C9_PMON_EVNT_SEL2	
06_2FH .....	See Table 35-17
06_3EH .....	See Table 35-26
06_3FH .....	See Table 35-31
MSR_C9_PMON_EVNT_SEL3	
06_2FH .....	See Table 35-17
06_3EH .....	See Table 35-26
06_3FH .....	See Table 35-31
MSR_C9_PMON_EVNT_SEL4	
06_2FH .....	See Table 35-17
MSR_C9_PMON_EVNT_SEL5	
06_2FH .....	See Table 35-17
MSR_CC6_DEMOTION_POLICY_CONFIG	
06_37H .....	See Table 35-9
MSR_CONFIG_TDP_CONTROL	
06_3AH .....	See Table 35-23
06_3CH, 06_45H, 06_46H .....	See Table 35-27
06_57H .....	See Table 35-40
MSR_CONFIG_TDP_LEVEL1	
06_3AH .....	See Table 35-23
06_3CH, 06_45H, 06_46H .....	See Table 35-27
06_57H .....	See Table 35-40
MSR_CONFIG_TDP_LEVEL2	
06_3AH .....	See Table 35-23
06_3CH, 06_45H, 06_46H .....	See Table 35-27
06_57H .....	See Table 35-40
MSR_CONFIG_TDP_NOMINAL	
06_3AH .....	See Table 35-23
06_3CH, 06_45H, 06_46H .....	See Table 35-27
06_57H .....	See Table 35-40
MSR_CORE_C1_RESIDENCY	
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH .....	See Table 35-6
MSR_CORE_C3_RESIDENCY	
06_5CH .....	See Table 35-12
06_1AH, 06_1EH, 06_1FH, 06_2EH, 06_25H, 06_2CH, 06_2FH .....	See Table 35-13
06_2AH, 06_2DH, 06_3AH, 06_3CH, 06_3EH, 06_3FH, 06_45H, 06_46H .....	See Table 35-18
MSR_CORE_C6_RESIDENCY	
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH .....	See Table 35-6
06_1AH, 06_1EH, 06_1FH, 06_2EH, 06_25H, 06_2CH, 06_2FH .....	See Table 35-13
06_2AH, 06_2DH, 06_3AH, 06_3CH, 06_3EH, 06_3FH, 06_45H, 06_46H .....	See Table 35-18
06_57H .....	See Table 35-40
MSR_CORE_C7_RESIDENCY	

## MODEL-SPECIFIC REGISTERS (MSRS)

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_2AH, 06_2DH, 06_3AH, 06_3CH, 06_3EH, 06_3FH, 06_45H, 06_46H	See Table 35-18
MSR_CORE_GFXE_OVERLAP_CO	
06_4EH, 06_5EH	See Table 35-37
MSR_CORE_HDC_RESIDENCY	
06_4EH, 06_5EH	See Table 35-37
MSR_CORE_PERF_LIMIT_REASONS	
06_5CH	See Table 35-12
06_3CH, 06_45H, 06_46H	See Table 35-28
06_3F	See Table 35-30
06_56H, 06_4FH	See Table 35-34
06_57H	See Table 35-40
MSR_CORE_THREAD_COUNT	
06_3FH	See Table 35-30
MSR_CRU_ESCR0	
0FH	See Table 35-41
MSR_CRU_ESCR1	
0FH	See Table 35-41
MSR_CRU_ESCR2	
0FH	See Table 35-41
MSR_CRU_ESCR3	
0FH	See Table 35-41
MSR_CRU_ESCR4	
0FH	See Table 35-41
MSR_CRU_ESCR5	
0FH	See Table 35-41
MSR_DAC_ESCR0	
0FH	See Table 35-41
MSR_DAC_ESCR1	
0FH	See Table 35-41
MSR_DRAM_ENERGY_STATUS	
06_5CH	See Table 35-12
06_2DH	See Table 35-21
06_3EH, 06_3FH	See Table 35-24
06_3CH, 06_45H, 06_46H	See Table 35-27
06_3F	See Table 35-30
06_56H, 06_4FH	See Table 35-34
06_57H	See Table 35-40
MSR_DRAM_PERF_STATUS	
06_5CH	See Table 35-12
06_2DH	See Table 35-21
06_3EH, 06_3FH	See Table 35-24
06_3CH, 06_45H, 06_46H	See Table 35-27

<b>MSR Name and CPUID DisplayFamily_DisplayModel</b>	<b>Location</b>
06_3F .....	See Table 35-30
06_56H, 06_4FH .....	See Table 35-34
06_57H .....	See Table 35-40
<b>MSR_DRAM_POWER_INFO</b>	
06_5CH .....	See Table 35-12
06_2DH .....	See Table 35-21
06_3EH, 06_3FH .....	See Table 35-24
06_3F .....	See Table 35-30
06_56H, 06_4FH .....	See Table 35-34
06_57H .....	See Table 35-40
<b>MSR_DRAM_POWER_LIMIT</b>	
06_5CH .....	See Table 35-12
06_2DH .....	See Table 35-21
06_3EH, 06_3FH .....	See Table 35-24
06_3F .....	See Table 35-30
06_56H, 06_4FH .....	See Table 35-34
06_57H .....	See Table 35-40
<b>MSR_EBC_FREQUENCY_ID</b>	
0FH .....	See Table 35-41
<b>MSR_EBC_HARD_POWERON</b>	
0FH .....	See Table 35-41
<b>MSR_EBC_SOFT_POWERON</b>	
0FH .....	See Table 35-41
<b>MSR_EBL_CR_POWERON</b>	
06_0FH, 06_17H .....	See Table 35-3
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H .....	See Table 35-4
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH .....	See Table 35-6
06_0EH .....	See Table 35-44
06_09H .....	See Table 35-45
<b>MSR_EFSB_DRDY0</b>	
0F_03H, 0F_04H .....	See Table 35-42
<b>MSR_EFSB_DRDY1</b>	
0F_03H, 0F_04H .....	See Table 35-42
<b>MSR_EMON_L3_CTR_CTL0</b>	
06_0FH, 06_17H .....	See Table 35-3
0F_06H .....	See Table 35-43
<b>MSR_EMON_L3_CTR_CTL1</b>	
06_0FH, 06_17H .....	See Table 35-3
0F_06H .....	See Table 35-43
<b>MSR_EMON_L3_CTR_CTL2</b>	
06_0FH, 06_17H .....	See Table 35-3
0F_06H .....	See Table 35-43



<b>MSR Name and CPUID DisplayFamily_DisplayModel</b>	<b>Location</b>
<b>MSR_EMON_L3_CTR_CTL3</b>	
06_0FH, 06_17H .....	See Table 35-3
0F_06H .....	See Table 35-43
<b>MSR_EMON_L3_CTR_CTL4</b>	
06_0FH, 06_17H .....	See Table 35-3
0F_06H .....	See Table 35-43
<b>MSR_EMON_L3_CTR_CTL5</b>	
06_0FH, 06_17H .....	See Table 35-3
0F_06H .....	See Table 35-43
<b>MSR_EMON_L3_CTR_CTL6</b>	
06_0FH, 06_17H .....	See Table 35-3
0F_06H .....	See Table 35-43
<b>MSR_EMON_L3_CTR_CTL7</b>	
06_0FH, 06_17H .....	See Table 35-3
0F_06H .....	See Table 35-43
<b>MSR_EMON_L3_GL_CTL</b>	
06_0FH, 06_17H .....	See Table 35-3
<b>MSR_ERROR_CONTROL</b>	
06_2DH .....	See Table 35-21
06_3EH .....	See Table 35-24
06_3F .....	See Table 35-30
<b>MSR_FEATURE_CONFIG</b>	
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH.....	See Table 35-6
06_25H, 06_2CH.....	See Table 35-16
06_2FH .....	See Table 35-17
06_2AH, 06_2DH.....	See Table 35-18
06_57H .....	See Table 35-40
<b>MSR_FIRM_ESCRO</b>	
0FH .....	See Table 35-41
<b>MSR_FIRM_ESCR1</b>	
0FH .....	See Table 35-41
<b>MSR_FLAME_CCCR0</b>	
0FH .....	See Table 35-41
<b>MSR_FLAME_CCCR1</b>	
0FH .....	See Table 35-41
<b>MSR_FLAME_CCCR2</b>	
0FH .....	See Table 35-41
<b>MSR_FLAME_CCCR3</b>	
0FH .....	See Table 35-41
<b>MSR_FLAME_COUNTER0</b>	
0FH .....	See Table 35-41
<b>MSR_FLAME_COUNTER1</b>	

<b>MSR Name and CPUID DisplayFamily_DisplayModel</b>	<b>Location</b>
0FH .....	See Table 35-41
MSR_FLAME_COUNTER2	
0FH .....	See Table 35-41
MSR_FLAME_COUNTER3	
0FH .....	See Table 35-41
MSR_FLAME_ESCRO	
0FH .....	See Table 35-41
MSR_FLAME_ESCR1	
0FH .....	See Table 35-41
MSR_FSB_ESCRO	
0FH .....	See Table 35-41
MSR_FSB_ESCR1	
0FH .....	See Table 35-41
MSR_FSB_FREQ	
06_0FH, 06_17H .....	See Table 35-3
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H .....	See Table 35-4
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH .....	See Table 35-7
06_4CH .....	See Table 35-11
06_0EH .....	See Table 35-44
MSR_GQ_SNOOP_MESF	
06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH .....	See Table 35-14
MSR_GRAPHICS_PERF_LIMIT_REASONS	
06_3CH, 06_45H, 06_46H .....	See Table 35-28
MSR_IFSB_BUSQ0	
0F_03H, 0F_04H .....	See Table 35-42
MSR_IFSB_BUSQ1	
0F_03H, 0F_04H .....	See Table 35-42
MSR_IFSB_CNTR7	
0F_03H, 0F_04H .....	See Table 35-42
MSR_IFSB_CTL6	
0F_03H, 0F_04H .....	See Table 35-42
MSR_IFSB_SNPQ0	
0F_03H, 0F_04H .....	See Table 35-42
MSR_IFSB_SNPQ1	
0F_03H, 0F_04H .....	See Table 35-42
MSR_IQ_CCCR0	
0FH .....	See Table 35-41
MSR_IQ_CCCR1	
0FH .....	See Table 35-41
MSR_IQ_CCCR2	
0FH .....	See Table 35-41
MSR_IQ_CCCR3	

## MODEL-SPECIFIC REGISTERS (MSRS)

MSR Name and CPUID DisplayFamily_DisplayModel	Location
0FH .....	See Table 35-41
MSR_IQ_CCCR4	
0FH .....	See Table 35-41
MSR_IQ_CCCR5	
0FH .....	See Table 35-41
MSR_IQ_COUNTER0	
0FH .....	See Table 35-41
MSR_IQ_COUNTER1	
0FH .....	See Table 35-41
MSR_IQ_COUNTER2	
0FH .....	See Table 35-41
MSR_IQ_COUNTER3	
0FH .....	See Table 35-41
MSR_IQ_COUNTER4	
0FH .....	See Table 35-41
MSR_IQ_COUNTER5	
0FH .....	See Table 35-41
MSR_IQ_ESCR0	
0FH .....	See Table 35-41
MSR_IQ_ESCR1	
0FH .....	See Table 35-41
MSR_IS_ESCR0	
0FH .....	See Table 35-41
MSR_IS_ESCR1	
0FH .....	See Table 35-41
MSR_ITLB_ESCR0	
0FH .....	See Table 35-41
MSR_ITLB_ESCR1	
0FH .....	See Table 35-41
MSR_IX_ESCR0	
0FH .....	See Table 35-41
MSR_IX_ESCR1	
0FH .....	See Table 35-41
MSR_LASTBRANCH_0	
0FH .....	See Table 35-41
06_0EH .....	See Table 35-44
06_09H .....	See Table 35-45
MSR_LASTBRANCH_0_FROM_IP	
06_0FH, 06_17H .....	See Table 35-3
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H .....	See Table 35-4
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH .....	See Table 35-7
06_5CH .....	See Table 35-12

<b>MSR Name and CPUID DisplayFamily_DisplayModel</b>	<b>Location</b>
06_1AH, 06_1EH, 06_1FH, 06_2EH .....	See Table 35-13
06_2AH, 06_2DH .....	See Table 35-18
0FH .....	See Table 35-41
<b>MSR_LASTBRANCH_0_TO_IP</b>	
06_0FH, 06_17H .....	See Table 35-3
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H .....	See Table 35-4
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH .....	See Table 35-7
06_5CH .....	See Table 35-12
06_1AH, 06_1EH, 06_1FH, 06_2EH .....	See Table 35-13
06_2AH, 06_2DH .....	See Table 35-18
0FH .....	See Table 35-41
<b>MSR_LASTBRANCH_1_FROM_IP</b>	
06_0FH, 06_17H .....	See Table 35-3
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H .....	See Table 35-4
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH .....	See Table 35-7
06_5CH .....	See Table 35-12
06_1AH, 06_1EH, 06_1FH, 06_2EH .....	See Table 35-13
06_2AH, 06_2DH .....	See Table 35-18
0FH .....	See Table 35-41
<b>MSR_LASTBRANCH_1_TO_IP</b>	
06_0FH, 06_17H .....	See Table 35-3
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H .....	See Table 35-4
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH .....	See Table 35-7
06_5CH .....	See Table 35-12
06_1AH, 06_1EH, 06_1FH, 06_2EH .....	See Table 35-13
06_2AH, 06_2DH .....	See Table 35-18
0FH .....	See Table 35-41
<b>MSR_LASTBRANCH_10_FROM_IP</b>	
06_5CH .....	See Table 35-12
06_1AH, 06_1EH, 06_1FH, 06_2EH .....	See Table 35-13
06_2AH, 06_2DH .....	See Table 35-18
0FH .....	See Table 35-41
<b>MSR_LASTBRANCH_10_TO_IP</b>	
06_5CH .....	See Table 35-12
06_1AH, 06_1EH, 06_1FH, 06_2EH .....	See Table 35-13
06_2AH, 06_2DH .....	See Table 35-18
0FH .....	See Table 35-41
<b>MSR_LASTBRANCH_11_FROM_IP</b>	
06_5CH .....	See Table 35-12
06_1AH, 06_1EH, 06_1FH, 06_2EH .....	See Table 35-13
06_2AH, 06_2DH .....	See Table 35-18
0FH .....	See Table 35-41

<b>MSR Name and CPUID DisplayFamily_DisplayModel</b>	<b>Location</b>
<b>MSR_LASTBRANCH_11_TO_IP</b>	
06_5CH.....	See Table 35-12
06_1AH, 06_1EH, 06_1FH, 06_2EH.....	See Table 35-13
06_2AH, 06_2DH.....	See Table 35-18
0FH.....	See Table 35-41
<b>MSR_LASTBRANCH_12_FROM_IP</b>	
06_5CH.....	See Table 35-12
06_1AH, 06_1EH, 06_1FH, 06_2EH.....	See Table 35-13
06_2AH, 06_2DH.....	See Table 35-18
0FH.....	See Table 35-41
<b>MSR_LASTBRANCH_12_TO_IP</b>	
06_5CH.....	See Table 35-12
06_1AH, 06_1EH, 06_1FH, 06_2EH.....	See Table 35-13
06_2AH, 06_2DH.....	See Table 35-18
0FH.....	See Table 35-41
<b>MSR_LASTBRANCH_13_FROM_IP</b>	
06_5CH.....	See Table 35-12
06_1AH, 06_1EH, 06_1FH, 06_2EH.....	See Table 35-13
06_2AH, 06_2DH.....	See Table 35-18
0FH.....	See Table 35-41
<b>MSR_LASTBRANCH_13_TO_IP</b>	
06_5CH.....	See Table 35-12
06_1AH, 06_1EH, 06_1FH, 06_2EH.....	See Table 35-13
06_2AH, 06_2DH.....	See Table 35-18
0FH.....	See Table 35-41
<b>MSR_LASTBRANCH_14_FROM_IP</b>	
06_5CH.....	See Table 35-12
06_1AH, 06_1EH, 06_1FH, 06_2EH.....	See Table 35-13
06_2AH, 06_2DH.....	See Table 35-18
0FH.....	See Table 35-41
<b>MSR_LASTBRANCH_14_TO_IP</b>	
06_5CH.....	See Table 35-12
06_1AH, 06_1EH, 06_1FH, 06_2EH.....	See Table 35-13
06_2AH, 06_2DH.....	See Table 35-18
0FH.....	See Table 35-41
<b>MSR_LASTBRANCH_15_FROM_IP</b>	
06_5CH.....	See Table 35-12
06_1AH, 06_1EH, 06_1FH, 06_2EH.....	See Table 35-13
06_2AH, 06_2DH.....	See Table 35-18
0FH.....	See Table 35-41
<b>MSR_LASTBRANCH_15_TO_IP</b>	
06_5CH.....	See Table 35-12

<b>MSR Name and CPUID DisplayFamily_DisplayModel</b>	<b>Location</b>
06_1AH, 06_1EH, 06_1FH, 06_2EH .....	See Table 35-13
06_2AH, 06_2DH .....	See Table 35-18
0FH .....	See Table 35-41
<b>MSR_LASTBRANCH_16_FROM_IP</b>	
06_5CH .....	See Table 35-12
06_4EH, 06_5EH .....	See Table 35-37
<b>MSR_LASTBRANCH_16_TO_IP</b>	
06_5CH .....	See Table 35-12
06_4EH, 06_5EH .....	See Table 35-37
<b>MSR_LASTBRANCH_17_FROM_IP</b>	
06_5CH .....	See Table 35-12
06_4EH, 06_5EH .....	See Table 35-37
<b>MSR_LASTBRANCH_17_TO_IP</b>	
06_5CH .....	See Table 35-12
06_4EH, 06_5EH .....	See Table 35-37
<b>MSR_LASTBRANCH_18_FROM_IP</b>	
06_5CH .....	See Table 35-12
06_4EH, 06_5EH .....	See Table 35-37
<b>MSR_LASTBRANCH_18_TO_IP</b>	
06_5CH .....	See Table 35-12
06_4EH, 06_5EH .....	See Table 35-37
<b>MSR_LASTBRANCH_19_FROM_IP</b>	
06_5CH .....	See Table 35-12
06_4EH, 06_5EH .....	See Table 35-37
<b>MSR_LASTBRANCH_19_TO_IP</b>	
06_5CH .....	See Table 35-12
06_4EH, 06_5EH .....	See Table 35-37
<b>MSR_LASTBRANCH_2</b>	
0FH .....	See Table 35-41
06_0EH .....	See Table 35-44
06_09H .....	See Table 35-45
<b>MSR_LASTBRANCH_2_FROM_IP</b>	
06_0FH, 06_17H .....	See Table 35-3
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H .....	See Table 35-4
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH .....	See Table 35-7
06_5CH .....	See Table 35-12
06_1AH, 06_1EH, 06_1FH, 06_2EH .....	See Table 35-13
06_2AH, 06_2DH .....	See Table 35-18
0FH .....	See Table 35-41
<b>MSR_LASTBRANCH_2_TO_IP</b>	
06_0FH, 06_17H .....	See Table 35-3
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H .....	See Table 35-4

<b>MSR Name and CPUID DisplayFamily_DisplayModel</b>	<b>Location</b>
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH.....	See Table 35-7
06_5CH.....	See Table 35-12
06_1AH, 06_1EH, 06_1FH, 06_2EH.....	See Table 35-13
06_2AH, 06_2DH.....	See Table 35-18
0FH.....	See Table 35-41
<b>MSR_LASTBRANCH_20_FROM_IP</b>	
06_5CH.....	See Table 35-12
06_4EH, 06_5EH.....	See Table 35-37
<b>MSR_LASTBRANCH_20_TO_IP</b>	
06_5CH.....	See Table 35-12
06_4EH, 06_5EH.....	See Table 35-37
<b>MSR_LASTBRANCH_21_FROM_IP</b>	
06_5CH.....	See Table 35-12
06_4EH, 06_5EH.....	See Table 35-37
<b>MSR_LASTBRANCH_21_TO_IP</b>	
06_5CH.....	See Table 35-12
06_4EH, 06_5EH.....	See Table 35-37
<b>MSR_LASTBRANCH_22_FROM_IP</b>	
06_5CH.....	See Table 35-12
06_4EH, 06_5EH.....	See Table 35-37
<b>MSR_LASTBRANCH_22_TO_IP</b>	
06_5CH.....	See Table 35-12
06_4EH, 06_5EH.....	See Table 35-37
<b>MSR_LASTBRANCH_23_FROM_IP</b>	
06_5CH.....	See Table 35-12
06_4EH, 06_5EH.....	See Table 35-37
<b>MSR_LASTBRANCH_23_TO_IP</b>	
06_5CH.....	See Table 35-12
06_4EH, 06_5EH.....	See Table 35-37
<b>MSR_LASTBRANCH_24_FROM_IP</b>	
06_5CH.....	See Table 35-12
06_4EH, 06_5EH.....	See Table 35-37
<b>MSR_LASTBRANCH_24_TO_IP</b>	
06_5CH.....	See Table 35-12
06_4EH, 06_5EH.....	See Table 35-37
<b>MSR_LASTBRANCH_25_FROM_IP</b>	
06_5CH.....	See Table 35-12
06_4EH, 06_5EH.....	See Table 35-37
<b>MSR_LASTBRANCH_25_TO_IP</b>	
06_5CH.....	See Table 35-12
06_4EH, 06_5EH.....	See Table 35-37
<b>MSR_LASTBRANCH_26_FROM_IP</b>	

<b>MSR Name and CPUID DisplayFamily_DisplayModel</b>	<b>Location</b>
06_5CH.....	See Table 35-12
06_4EH, 06_5EH .....	See Table 35-37
<b>MSR_LASTBRANCH_26_TO_IP</b>	
06_5CH.....	See Table 35-12
06_4EH, 06_5EH .....	See Table 35-37
<b>MSR_LASTBRANCH_27_FROM_IP</b>	
06_5CH.....	See Table 35-12
06_4EH, 06_5EH .....	See Table 35-37
<b>MSR_LASTBRANCH_27_TO_IP</b>	
06_5CH.....	See Table 35-12
06_4EH, 06_5EH .....	See Table 35-37
<b>MSR_LASTBRANCH_28_FROM_IP</b>	
06_5CH.....	See Table 35-12
06_4EH, 06_5EH .....	See Table 35-37
<b>MSR_LASTBRANCH_28_TO_IP</b>	
06_5CH.....	See Table 35-12
06_4EH, 06_5EH .....	See Table 35-37
<b>MSR_LASTBRANCH_29_FROM_IP</b>	
06_5CH.....	See Table 35-12
06_4EH, 06_5EH .....	See Table 35-37
<b>MSR_LASTBRANCH_29_TO_IP</b>	
06_5CH.....	See Table 35-12
06_4EH, 06_5EH .....	See Table 35-37
<b>MSR_LASTBRANCH_3</b>	
0FH.....	See Table 35-41
06_0EH.....	See Table 35-44
06_09H.....	See Table 35-45
<b>MSR_LASTBRANCH_3_FROM_IP</b>	
06_0FH, 06_17H .....	See Table 35-3
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H .....	See Table 35-4
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH .....	See Table 35-7
06_5CH.....	See Table 35-12
06_1AH, 06_1EH, 06_1FH, 06_2EH .....	See Table 35-13
06_2AH, 06_2DH .....	See Table 35-18
0FH.....	See Table 35-41
<b>MSR_LASTBRANCH_3_TO_IP</b>	
06_0FH, 06_17H .....	See Table 35-3
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H .....	See Table 35-4
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH .....	See Table 35-7
06_5CH.....	See Table 35-12
06_1AH, 06_1EH, 06_1FH, 06_2EH .....	See Table 35-13
06_2AH, 06_2DH .....	See Table 35-18



## MODEL-SPECIFIC REGISTERS (MSRS)

MSR Name and CPUID DisplayFamily_DisplayModel	Location
0FH.....	See Table 35-41
MSR_LASTBRANCH_30_FROM_IP	
06_5CH.....	See Table 35-12
06_4EH, 06_5EH.....	See Table 35-37
MSR_LASTBRANCH_30_TO_IP	
06_5CH.....	See Table 35-12
06_4EH, 06_5EH.....	See Table 35-37
MSR_LASTBRANCH_31_FROM_IP	
06_5CH.....	See Table 35-12
06_4EH, 06_5EH.....	See Table 35-37
MSR_LASTBRANCH_31_TO_IP	
06_5CH.....	See Table 35-12
06_4EH, 06_5EH.....	See Table 35-37
MSR_LASTBRANCH_4	
06_0EH.....	See Table 35-44
06_09H.....	See Table 35-45
MSR_LASTBRANCH_4_FROM_IP	
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H.....	See Table 35-4
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH.....	See Table 35-7
06_5CH.....	See Table 35-12
06_1AH, 06_1EH, 06_1FH, 06_2EH.....	See Table 35-13
06_2AH, 06_2DH.....	See Table 35-18
0FH.....	See Table 35-41
MSR_LASTBRANCH_4_TO_IP	
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H.....	See Table 35-4
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH.....	See Table 35-7
06_5CH.....	See Table 35-12
06_1AH, 06_1EH, 06_1FH, 06_2EH.....	See Table 35-13
06_2AH, 06_2DH.....	See Table 35-18
0FH.....	See Table 35-41
MSR_LASTBRANCH_5	
06_0EH.....	See Table 35-44
06_09H.....	See Table 35-45
MSR_LASTBRANCH_5_FROM_IP	
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H.....	See Table 35-4
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH.....	See Table 35-7
06_5CH.....	See Table 35-12
06_1AH, 06_1EH, 06_1FH, 06_2EH.....	See Table 35-13
06_2AH, 06_2DH.....	See Table 35-18
0FH.....	See Table 35-41
MSR_LASTBRANCH_5_TO_IP	
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H.....	See Table 35-4

<b>MSR Name and CPUID DisplayFamily_DisplayModel</b>	<b>Location</b>
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH .....	See Table 35-7
06_5CH .....	See Table 35-12
06_1AH, 06_1EH, 06_1FH, 06_2EH .....	See Table 35-13
06_2AH, 06_2DH .....	See Table 35-18
0FH .....	See Table 35-41
<b>MSR_LASTBRANCH_6</b>	
06_0EH .....	See Table 35-44
06_09H .....	See Table 35-45
<b>MSR_LASTBRANCH_6_FROM_IP</b>	
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H .....	See Table 35-4
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH .....	See Table 35-7
06_5CH .....	See Table 35-12
06_1AH, 06_1EH, 06_1FH, 06_2EH .....	See Table 35-13
06_2AH, 06_2DH .....	See Table 35-18
0FH .....	See Table 35-41
<b>MSR_LASTBRANCH_6_TO_IP</b>	
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H .....	See Table 35-4
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH .....	See Table 35-7
06_5CH .....	See Table 35-12
06_1AH, 06_1EH, 06_1FH, 06_2EH .....	See Table 35-13
06_2AH, 06_2DH .....	See Table 35-18
0FH .....	See Table 35-41
<b>MSR_LASTBRANCH_7</b>	
06_0EH .....	See Table 35-44
06_09H .....	See Table 35-45
<b>MSR_LASTBRANCH_7_FROM_IP</b>	
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H .....	See Table 35-4
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH .....	See Table 35-7
06_5CH .....	See Table 35-12
06_1AH, 06_1EH, 06_1FH, 06_2EH .....	See Table 35-13
06_2AH, 06_2DH .....	See Table 35-18
0FH .....	See Table 35-41
<b>MSR_LASTBRANCH_7_TO_IP</b>	
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H .....	See Table 35-4
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH .....	See Table 35-7
06_5CH .....	See Table 35-12
06_1AH, 06_1EH, 06_1FH, 06_2EH .....	See Table 35-13
06_2AH, 06_2DH .....	See Table 35-18
0FH .....	See Table 35-41
<b>MSR_LASTBRANCH_8_FROM_IP</b>	
06_5CH .....	See Table 35-12
06_1AH, 06_1EH, 06_1FH, 06_2EH .....	See Table 35-13

## MODEL-SPECIFIC REGISTERS (MSRS)

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_2AH, 06_2DH .....	See Table 35-18
0FH .....	See Table 35-41
<b>MSR_LASTBRANCH_8_TO_IP</b>	
06_5CH .....	See Table 35-12
06_1AH, 06_1EH, 06_1FH, 06_2EH .....	See Table 35-13
06_2AH, 06_2DH .....	See Table 35-18
0FH .....	See Table 35-41
<b>MSR_LASTBRANCH_9_FROM_IP</b>	
06_5CH .....	See Table 35-12
06_1AH, 06_1EH, 06_1FH, 06_2EH .....	See Table 35-13
06_2AH, 06_2DH .....	See Table 35-18
0FH .....	See Table 35-41
<b>MSR_LASTBRANCH_9_TO_IP</b>	
06_5CH .....	See Table 35-12
06_1AH, 06_1EH, 06_1FH, 06_2EH .....	See Table 35-13
06_2AH, 06_2DH .....	See Table 35-18
0FH .....	See Table 35-41
<b>MSR_LASTBRANCH_TOS</b>	
06_0FH, 06_17H .....	See Table 35-3
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H .....	See Table 35-4
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH .....	See Table 35-7
06_5CH .....	See Table 35-12
06_1AH, 06_1EH, 06_1FH, 06_2EH .....	See Table 35-13
06_2AH, 06_2DH .....	See Table 35-18
06_4EH, 06_5EH .....	See Table 35-37
06_57H .....	See Table 35-40
06_0EH .....	See Table 35-44
06_09H .....	See Table 35-45
<b>MSR_LBR_INFO_1</b>	
06_4EH, 06_5EH .....	See Table 35-37
<b>MSR_LBR_INFO_10</b>	
06_4EH, 06_5EH .....	See Table 35-37
<b>MSR_LBR_INFO_11</b>	
06_4EH, 06_5EH .....	See Table 35-37
<b>MSR_LBR_INFO_12</b>	
06_4EH, 06_5EH .....	See Table 35-37
<b>MSR_LBR_INFO_13</b>	
06_4EH, 06_5EH .....	See Table 35-37
<b>MSR_LBR_INFO_14</b>	
06_4EH, 06_5EH .....	See Table 35-37
<b>MSR_LBR_INFO_15</b>	
06_4EH, 06_5EH .....	See Table 35-37

<b>MSR Name and CPUID DisplayFamily_DisplayModel</b>	<b>Location</b>
MSR_LBR_INFO_16 06_4EH, 06_5EH .....	See Table 35-37
MSR_LBR_INFO_17 06_4EH, 06_5EH .....	See Table 35-37
MSR_LBR_INFO_18 06_4EH, 06_5EH .....	See Table 35-37
MSR_LBR_INFO_19 06_4EH, 06_5EH .....	See Table 35-37
MSR_LBR_INFO_2 06_4EH, 06_5EH .....	See Table 35-37
MSR_LBR_INFO_20 06_4EH, 06_5EH .....	See Table 35-37
MSR_LBR_INFO_21 06_4EH, 06_5EH .....	See Table 35-37
MSR_LBR_INFO_22 06_4EH, 06_5EH .....	See Table 35-37
MSR_LBR_INFO_23 06_4EH, 06_5EH .....	See Table 35-37
MSR_LBR_INFO_24 06_4EH, 06_5EH .....	See Table 35-37
MSR_LBR_INFO_25 06_4EH, 06_5EH .....	See Table 35-37
MSR_LBR_INFO_26 06_4EH, 06_5EH .....	See Table 35-37
MSR_LBR_INFO_27 06_4EH, 06_5EH .....	See Table 35-37
MSR_LBR_INFO_28 06_4EH, 06_5EH .....	See Table 35-37
MSR_LBR_INFO_29 06_4EH, 06_5EH .....	See Table 35-37
MSR_LBR_INFO_3 06_4EH, 06_5EH .....	See Table 35-37
MSR_LBR_INFO_30 06_4EH, 06_5EH .....	See Table 35-37
MSR_LBR_INFO_31 06_4EH, 06_5EH .....	See Table 35-37
MSR_LBR_INFO_4 06_4EH, 06_5EH .....	See Table 35-37
MSR_LBR_INFO_5 06_4EH, 06_5EH .....	See Table 35-37
MSR_LBR_INFO_6 06_4EH, 06_5EH .....	See Table 35-37

## MODEL-SPECIFIC REGISTERS (MSRS)

MSR Name and CPUID DisplayFamily_DisplayModel	Location
MSR_LBR_INFO_7 06_4EH, 06_5EH .....	See Table 35-37
MSR_LBR_INFO_8 06_4EH, 06_5EH .....	See Table 35-37
MSR_LBR_INFO_9 06_4EH, 06_5EH .....	See Table 35-37
MSR_LBR_SELECT 06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH .....	See Table 35-7
06_5CH .....	See Table 35-12
06_1AH, 06_1EH, 06_1FH, 06_2EH .....	See Table 35-13
06_2AH, 06_2DH .....	See Table 35-18
06_3CH, 06_45H, 06_46H .....	See Table 35-27
06_57H .....	See Table 35-40
MSR_LER_FROM_LIP 06_0FH, 06_17H .....	See Table 35-3
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H .....	See Table 35-4
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH .....	See Table 35-6
06_1AH, 06_1EH, 06_1FH, 06_2EH .....	See Table 35-13
06_2AH, 06_2DH .....	See Table 35-18
06_57H .....	See Table 35-40
0FH .....	See Table 35-41
06_0EH .....	See Table 35-44
06_09H .....	See Table 35-45
MSR_LER_TO_LIP 06_0FH, 06_17H .....	See Table 35-3
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H .....	See Table 35-4
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH .....	See Table 35-6
06_1AH, 06_1EH, 06_1FH, 06_2EH .....	See Table 35-13
06_2AH, 06_2DH .....	See Table 35-18
06_57H .....	See Table 35-40
0FH .....	See Table 35-41
06_0EH .....	See Table 35-44
06_09H .....	See Table 35-45
MSR_MO_PMON_ADDR_MASK 06_2EH .....	See Table 35-15
MSR_MO_PMON_ADDR_MATCH 06_2EH .....	See Table 35-15
MSR_MO_PMON_BOX_CTRL 06_2EH .....	See Table 35-15
MSR_MO_PMON_BOX_OVF_CTRL 06_2EH .....	See Table 35-15
MSR_MO_PMON_BOX_STATUS	

<b>MSR Name and CPUID DisplayFamily_DisplayModel</b>	<b>Location</b>
MSR_MO_PMON_CTRL0 06_2EH.....	See Table 35-15
MSR_MO_PMON_CTRL1 06_2EH.....	See Table 35-15
MSR_MO_PMON_CTRL2 06_2EH.....	See Table 35-15
MSR_MO_PMON_CTRL3 06_2EH.....	See Table 35-15
MSR_MO_PMON_CTRL4 06_2EH.....	See Table 35-15
MSR_MO_PMON_CTRL5 06_2EH.....	See Table 35-15
MSR_MO_PMON_DSP 06_2EH.....	See Table 35-15
MSR_MO_PMON_EVNT_SEL0 06_2EH.....	See Table 35-15
MSR_MO_PMON_EVNT_SEL1 06_2EH.....	See Table 35-15
MSR_MO_PMON_EVNT_SEL2 06_2EH.....	See Table 35-15
MSR_MO_PMON_EVNT_SEL3 06_2EH.....	See Table 35-15
MSR_MO_PMON_EVNT_SEL4 06_2EH.....	See Table 35-15
MSR_MO_PMON_EVNT_SEL5 06_2EH.....	See Table 35-15
MSR_MO_PMON_ISS 06_2EH.....	See Table 35-15
MSR_MO_PMON_MAP 06_2EH.....	See Table 35-15
MSR_MO_PMON_MM_CONFIG 06_2EH.....	See Table 35-15
MSR_MO_PMON_MSC_THR 06_2EH.....	See Table 35-15
MSR_MO_PMON_PGT 06_2EH.....	See Table 35-15
MSR_MO_PMON_PLD 06_2EH.....	See Table 35-15
MSR_MO_PMON_TIMESTAMP 06_2EH.....	See Table 35-15
MSR_MO_PMON_ZDP	

## MODEL-SPECIFIC REGISTERS (MSRS)

<b>MSR Name and CPUID DisplayFamily_DisplayModel</b>	<b>Location</b>
06_2EH.....	See Table 35-15
MSR_M1_PMON_ADDR_MASK	
06_2EH.....	See Table 35-15
MSR_M1_PMON_ADDR_MATCH	
06_2EH.....	See Table 35-15
MSR_M1_PMON_BOX_CTRL	
06_2EH.....	See Table 35-15
MSR_M1_PMON_BOX_OVF_CTRL	
06_2EH.....	See Table 35-15
MSR_M1_PMON_BOX_STATUS	
06_2EH.....	See Table 35-15
MSR_M1_PMON_CTRL0	
06_2EH.....	See Table 35-15
MSR_M1_PMON_CTRL1	
06_2EH.....	See Table 35-15
MSR_M1_PMON_CTRL2	
06_2EH.....	See Table 35-15
MSR_M1_PMON_CTRL3	
06_2EH.....	See Table 35-15
MSR_M1_PMON_CTRL4	
06_2EH.....	See Table 35-15
MSR_M1_PMON_CTRL5	
06_2EH.....	See Table 35-15
MSR_M1_PMON_DSP	
06_2EH.....	See Table 35-15
MSR_M1_PMON_EVNT_SELO	
06_2EH.....	See Table 35-15
MSR_M1_PMON_EVNT_SEL1	
06_2EH.....	See Table 35-15
MSR_M1_PMON_EVNT_SEL2	
06_2EH.....	See Table 35-15
MSR_M1_PMON_EVNT_SEL3	
06_2EH.....	See Table 35-15
MSR_M1_PMON_EVNT_SEL4	
06_2EH.....	See Table 35-15
MSR_M1_PMON_EVNT_SEL5	
06_2EH.....	See Table 35-15
MSR_M1_PMON_ISS	
06_2EH.....	See Table 35-15
MSR_M1_PMON_MAP	
06_2EH.....	See Table 35-15
MSR_M1_PMON_MM_CONFIG	

<b>MSR Name and CPUID DisplayFamily_DisplayModel</b>	<b>Location</b>
06_2EH.....	See Table 35-15
MSR_M1_PMON_MSC_THR	
06_2EH.....	See Table 35-15
MSR_M1_PMON_PGT	
06_2EH.....	See Table 35-15
MSR_M1_PMON_PLD	
06_2EH.....	See Table 35-15
MSR_M1_PMON_TIMESTAMP	
06_2EH.....	See Table 35-15
MSR_M1_PMON_ZDP	
06_2EH.....	See Table 35-15
IA32_MCO_MISC / MSR_MCO_MISC	
06_1AH, 06_1EH, 06_1FH, 06_2EH.....	See Table 35-13
MSR_MCO_RESIDENCY	
06_57H.....	See Table 35-40
IA32_MC1_MISC / MSR_MC1_MISC	
06_1AH, 06_1EH, 06_1FH, 06_2EH.....	See Table 35-13
IA32_MC10_ADDR / MSR_MC10_ADDR	
06_2EH.....	See Table 35-15
06_2DH.....	See Table 35-21
06_3EH.....	See Table 35-24
06_3F.....	See Table 35-30
06_56H, 06_4FH.....	See Table 35-35
06_4FH.....	See Table 35-36
IA32_MC10_CTL / MSR_MC10_CTL	
06_2EH.....	See Table 35-15
06_2DH.....	See Table 35-21
06_3EH.....	See Table 35-24
06_3F.....	See Table 35-30
06_56H, 06_4FH.....	See Table 35-35
06_4FH.....	See Table 35-36
IA32_MC10_MISC / MSR_MC10_MISC	
06_2EH.....	See Table 35-15
06_2DH.....	See Table 35-21
06_3EH.....	See Table 35-24
06_3F.....	See Table 35-30
06_56H, 06_4FH.....	See Table 35-35
06_4FH.....	See Table 35-36
IA32_MC10_STATUS / MSR_MC10_STATUS	
06_2EH.....	See Table 35-15
06_2DH.....	See Table 35-21
06_3EH.....	See Table 35-24



<b>MSR Name and CPUID DisplayFamily_DisplayModel</b>	<b>Location</b>
06_3F.....	See Table 35-30
06_56H, 06_4FH.....	See Table 35-35
06_4FH.....	See Table 35-36
<b>IA32_MC11_ADDR / MSR_MC11_ADDR</b>	
06_2EH.....	See Table 35-15
06_2DH.....	See Table 35-21
06_3EH.....	See Table 35-24
06_3F.....	See Table 35-30
06_4FH.....	See Table 35-36
<b>IA32_MC11_CTL / MSR_MC11_CTL</b>	
06_2EH.....	See Table 35-15
06_2DH.....	See Table 35-21
06_3EH.....	See Table 35-24
06_3F.....	See Table 35-30
06_4FH.....	See Table 35-36
<b>IA32_MC11_MISC / MSR_MC11_MISC</b>	
06_2EH.....	See Table 35-15
06_2DH.....	See Table 35-21
06_3EH.....	See Table 35-24
06_3F.....	See Table 35-30
06_4FH.....	See Table 35-36
<b>IA32_MC11_STATUS / MSR_MC11_STATUS</b>	
06_2EH.....	See Table 35-15
06_2DH.....	See Table 35-21
06_3EH.....	See Table 35-24
06_3F.....	See Table 35-30
06_4FH.....	See Table 35-36
<b>IA32_MC12_ADDR / MSR_MC12_ADDR</b>	
06_2EH.....	See Table 35-15
06_2DH.....	See Table 35-21
06_3EH.....	See Table 35-24
06_3F.....	See Table 35-30
06_4FH.....	See Table 35-36
<b>IA32_MC12_CTL / MSR_MC12_CTL</b>	
06_2EH.....	See Table 35-15
06_2DH.....	See Table 35-21
06_3EH.....	See Table 35-24
06_3F.....	See Table 35-30
06_4FH.....	See Table 35-36
<b>IA32_MC12_MISC / MSR_MC12_MISC</b>	
06_2EH.....	See Table 35-15
06_2DH.....	See Table 35-21

<b>MSR Name and CPUID DisplayFamily_DisplayModel</b>	<b>Location</b>
06_3EH.....	See Table 35-24
06_3F.....	See Table 35-30
06_4FH.....	See Table 35-36
<b>IA32_MC12_STATUS / MSR_MC12_STATUS</b>	
06_2EH.....	See Table 35-15
06_2DH.....	See Table 35-21
06_3EH.....	See Table 35-24
06_3F.....	See Table 35-30
06_4FH.....	See Table 35-36
<b>IA32_MC13_ADDR / MSR_MC13_ADDR</b>	
06_2EH.....	See Table 35-15
06_2DH.....	See Table 35-21
06_3EH.....	See Table 35-24
06_3F.....	See Table 35-30
06_4FH.....	See Table 35-36
<b>IA32_MC13_CTL / MSR_MC13_CTL</b>	
06_2EH.....	See Table 35-15
06_2DH.....	See Table 35-21
06_3EH.....	See Table 35-24
06_3F.....	See Table 35-30
06_4FH.....	See Table 35-36
<b>IA32_MC13_MISC / MSR_MC13_MISC</b>	
06_2EH.....	See Table 35-15
06_2DH.....	See Table 35-21
06_3EH.....	See Table 35-24
06_3F.....	See Table 35-30
06_4FH.....	See Table 35-36
<b>IA32_MC13_STATUS / MSR_MC13_STATUS</b>	
06_2EH.....	See Table 35-15
06_2DH.....	See Table 35-21
06_3EH.....	See Table 35-24
06_3F.....	See Table 35-30
06_4FH.....	See Table 35-36
<b>IA32_MC14_ADDR / MSR_MC14_ADDR</b>	
06_2EH.....	See Table 35-15
06_2DH.....	See Table 35-21
06_3EH.....	See Table 35-24
06_3F.....	See Table 35-30
06_4FH.....	See Table 35-36
<b>IA32_MC14_CTL / MSR_MC14_CTL</b>	
06_2EH.....	See Table 35-15
06_2DH.....	See Table 35-21

<b>MSR Name and CPUID DisplayFamily_DisplayModel</b>	<b>Location</b>
06_3EH.....	See Table 35-24
06_3F.....	See Table 35-30
06_4FH.....	See Table 35-36
<b>IA32_MC14_MISC / MSR_MC14_MISC</b>	
06_2EH.....	See Table 35-15
06_2DH.....	See Table 35-21
06_3EH.....	See Table 35-24
06_3F.....	See Table 35-30
06_4FH.....	See Table 35-36
<b>IA32_MC14_STATUS / MSR_MC14_STATUS</b>	
06_2EH.....	See Table 35-15
06_2DH.....	See Table 35-21
06_3EH.....	See Table 35-24
06_3F.....	See Table 35-30
06_4FH.....	See Table 35-36
<b>IA32_MC15_ADDR / MSR_MC15_ADDR</b>	
06_2EH.....	See Table 35-15
06_2DH.....	See Table 35-21
06_3EH.....	See Table 35-24
06_3F.....	See Table 35-30
06_4FH.....	See Table 35-36
<b>IA32_MC15_CTL / MSR_MC15_CTL</b>	
06_2EH.....	See Table 35-15
06_2DH.....	See Table 35-21
06_3EH.....	See Table 35-24
06_3F.....	See Table 35-30
06_4FH.....	See Table 35-36
<b>IA32_MC15_MISC / MSR_MC15_MISC</b>	
06_2EH.....	See Table 35-15
06_2DH.....	See Table 35-21
06_3EH.....	See Table 35-24
06_3F.....	See Table 35-30
06_4FH.....	See Table 35-36
<b>IA32_MC15_STATUS / MSR_MC15_STATUS</b>	
06_2EH.....	See Table 35-15
06_2DH.....	See Table 35-21
06_3EH.....	See Table 35-24
06_3F.....	See Table 35-30
06_4FH.....	See Table 35-36
<b>IA32_MC16_ADDR / MSR_MC16_ADDR</b>	
06_2EH.....	See Table 35-15
06_2DH.....	See Table 35-21

<b>MSR Name and CPUID DisplayFamily_DisplayModel</b>	<b>Location</b>
06_3EH.....	See Table 35-24
06_3F.....	See Table 35-30
06_4FH.....	See Table 35-36
<b>IA32_MC16_CTL / MSR_MC16_CTL</b>	
06_2EH.....	See Table 35-15
06_2DH.....	See Table 35-21
06_3EH.....	See Table 35-24
06_3F.....	See Table 35-30
06_4FH.....	See Table 35-36
<b>IA32_MC16_MISC / MSR_MC16_MISC</b>	
06_2EH.....	See Table 35-15
06_2DH.....	See Table 35-21
06_3EH.....	See Table 35-24
06_3F.....	See Table 35-30
06_4FH.....	See Table 35-36
<b>IA32_MC16_STATUS / MSR_MC16_STATUS</b>	
06_2EH.....	See Table 35-15
06_2DH.....	See Table 35-21
06_3EH.....	See Table 35-24
06_3F.....	See Table 35-30
06_4FH.....	See Table 35-36
<b>IA32_MC17_ADDR / MSR_MC17_ADDR</b>	
06_2EH.....	See Table 35-15
06_2DH.....	See Table 35-21
06_3EH.....	See Table 35-24
06_3F.....	See Table 35-30
06_56H, 06_4FH.....	See Table 35-35
06_4FH.....	See Table 35-36
<b>IA32_MC17_CTL / MSR_MC17_CTL</b>	
06_2EH.....	See Table 35-15
06_2DH.....	See Table 35-21
06_3EH.....	See Table 35-24
06_3F.....	See Table 35-30
06_56H, 06_4FH.....	See Table 35-35
06_4FH.....	See Table 35-36
<b>IA32_MC17_MISC / MSR_MC17_MISC</b>	
06_2EH.....	See Table 35-15
06_2DH.....	See Table 35-21
06_3EH.....	See Table 35-24
06_3F.....	See Table 35-30
06_56H, 06_4FH.....	See Table 35-35
06_4FH.....	See Table 35-36

MSR Name and CPUID DisplayFamily_DisplayModel	Location
<b>IA32_MC17_STATUS / MSR_MC17_STATUS</b>	
06_2EH.....	See Table 35-15
06_2DH.....	See Table 35-21
06_3EH.....	See Table 35-24
06_3F.....	See Table 35-30
06_56H, 06_4FH.....	See Table 35-35
06_4FH.....	See Table 35-36
<b>IA32_MC18_ADDR / MSR_MC18_ADDR</b>	
06_2EH.....	See Table 35-15
06_2DH.....	See Table 35-21
06_3EH.....	See Table 35-24
06_3F.....	See Table 35-30
06_56H, 06_4FH.....	See Table 35-35
06_4FH.....	See Table 35-36
<b>IA32_MC18_CTL / MSR_MC18_CTL</b>	
06_2EH.....	See Table 35-15
06_2DH.....	See Table 35-21
06_3EH.....	See Table 35-24
06_3F.....	See Table 35-30
06_56H, 06_4FH.....	See Table 35-35
06_4FH.....	See Table 35-36
<b>IA32_MC18_MISC / MSR_MC18_MISC</b>	
06_2EH.....	See Table 35-15
06_2DH.....	See Table 35-21
06_3EH.....	See Table 35-24
06_3F.....	See Table 35-30
06_56H, 06_4FH.....	See Table 35-35
06_4FH.....	See Table 35-36
<b>IA32_MC18_STATUS / MSR_MC18_STATUS</b>	
06_2EH.....	See Table 35-15
06_2DH.....	See Table 35-21
06_3EH.....	See Table 35-24
06_3F.....	See Table 35-30
06_56H, 06_4FH.....	See Table 35-35
06_4FH.....	See Table 35-36
<b>IA32_MC19_ADDR / MSR_MC19_ADDR</b>	
06_2EH.....	See Table 35-15
06_2DH.....	See Table 35-21
06_3EH.....	See Table 35-24
06_3F.....	See Table 35-30
06_56H, 06_4FH.....	See Table 35-35
06_4FH.....	See Table 35-36

MSR Name and CPUID DisplayFamily_DisplayModel	Location
IA32_MC19_CTL / MSR_MC19_CTL	
06_2EH.....	See Table 35-15
06_2DH.....	See Table 35-21
06_3EH.....	See Table 35-24
06_3F.....	See Table 35-30
06_56H, 06_4FH.....	See Table 35-35
06_4FH.....	See Table 35-36
IA32_MC19_MISC / MSR_MC19_MISC	
06_2EH.....	See Table 35-15
06_2DH.....	See Table 35-21
06_3EH.....	See Table 35-24
06_3F.....	See Table 35-30
06_56H, 06_4FH.....	See Table 35-35
06_4FH.....	See Table 35-36
IA32_MC19_STATUS / MSR_MC19_STATUS	
06_2EH.....	See Table 35-15
06_2DH.....	See Table 35-21
06_3EH.....	See Table 35-24
06_3F.....	See Table 35-30
06_56H, 06_4FH.....	See Table 35-35
06_4FH.....	See Table 35-36
IA32_MC2_MISC / MSR_MC2_MISC	
06_1AH, 06_1EH, 06_1FH, 06_2EH.....	See Table 35-13
IA32_MC20_ADDR / MSR_MC20_ADDR	
06_2EH.....	See Table 35-15
06_3EH.....	See Table 35-24
06_3F.....	See Table 35-30
06_4FH.....	See Table 35-36
IA32_MC20_CTL / MSR_MC20_CTL	
06_2EH.....	See Table 35-15
06_3EH.....	See Table 35-24
06_3F.....	See Table 35-30
06_4FH.....	See Table 35-36
IA32_MC20_MISC / MSR_MC20_MISC	
06_2EH.....	See Table 35-15
06_3EH.....	See Table 35-24
06_3F.....	See Table 35-30
06_4FH.....	See Table 35-36
IA32_MC20_STATUS / MSR_MC20_STATUS	
06_2EH.....	See Table 35-15
06_3EH.....	See Table 35-24
06_3F.....	See Table 35-30

<b>MSR Name and CPUID DisplayFamily_DisplayModel</b>	<b>Location</b>
06_4FH.....	See Table 35-36
IA32_MC21_ADDR / MSR_MC21_ADDR	
06_2EH.....	See Table 35-15
06_3EH.....	See Table 35-24
06_3F.....	See Table 35-30
06_4F.....	See Table 35-36
IA32_MC21_CTL / MSR_MC21_CTL	
06_2EH.....	See Table 35-15
06_3EH.....	See Table 35-24
06_3F.....	See Table 35-30
06_4F.....	See Table 35-36
IA32_MC21_MISC / MSR_MC21_MISC	
06_2EH.....	See Table 35-15
06_3EH.....	See Table 35-24
06_3F.....	See Table 35-30
06_4F.....	See Table 35-36
IA32_MC21_STATUS / MSR_MC21_STATUS	
06_2EH.....	See Table 35-15
06_3EH.....	See Table 35-24
06_3F.....	See Table 35-30
06_4F.....	See Table 35-36
IA32_MC22_ADDR / MSR_MC22_ADDR	
06_3EH.....	See Table 35-24
IA32_MC22_CTL / MSR_MC22_CTL	
06_3EH.....	See Table 35-24
IA32_MC22_MISC / MSR_MC22_MISC	
06_3EH.....	See Table 35-24
IA32_MC22_STATUS / MSR_MC22_STATUS	
06_3EH.....	See Table 35-24
IA32_MC23_ADDR / MSR_MC23_ADDR	
06_3EH.....	See Table 35-24
IA32_MC23_CTL / MSR_MC23_CTL	
06_3EH.....	See Table 35-24
IA32_MC23_MISC / MSR_MC23_MISC	
06_3EH.....	See Table 35-24
IA32_MC23_STATUS / MSR_MC23_STATUS	
06_3EH.....	See Table 35-24
IA32_MC24_ADDR / MSR_MC24_ADDR	
06_3EH.....	See Table 35-24
IA32_MC24_CTL / MSR_MC24_CTL	
06_3EH.....	See Table 35-24
IA32_MC24_MISC / MSR_MC24_MISC	

<b>MSR Name and CPUID DisplayFamily_DisplayModel</b>	<b>Location</b>
06_3EH.....	See Table 35-24
IA32_MC24_STATUS / MSR_MC24_STATUS	
06_3EH.....	See Table 35-24
IA32_MC25_ADDR / MSR_MC25_ADDR	
06_3EH.....	See Table 35-24
IA32_MC25_CTL / MSR_MC25_CTL	
06_3EH.....	See Table 35-24
IA32_MC25_MISC / MSR_MC25_MISC	
06_3EH.....	See Table 35-24
IA32_MC25_STATUS / MSR_MC25_STATUS	
06_3EH.....	See Table 35-24
IA32_MC26_ADDR / MSR_MC26_ADDR	
06_3EH.....	See Table 35-24
IA32_MC26_CTL / MSR_MC26_CTL	
06_3EH.....	See Table 35-24
IA32_MC26_MISC / MSR_MC26_MISC	
06_3EH.....	See Table 35-24
IA32_MC26_STATUS / MSR_MC26_STATUS	
06_3EH.....	See Table 35-24
IA32_MC27_ADDR / MSR_MC27_ADDR	
06_3EH.....	See Table 35-24
IA32_MC27_CTL / MSR_MC27_CTL	
06_3EH.....	See Table 35-24
IA32_MC27_MISC / MSR_MC27_MISC	
06_3EH.....	See Table 35-24
IA32_MC27_STATUS / MSR_MC27_STATUS	
06_3EH.....	See Table 35-24
IA32_MC28_ADDR / MSR_MC28_ADDR	
06_3EH.....	See Table 35-24
IA32_MC28_CTL / MSR_MC28_CTL	
06_3EH.....	See Table 35-24
IA32_MC28_MISC / MSR_MC28_MISC	
06_3EH.....	See Table 35-24
IA32_MC28_STATUS / MSR_MC28_STATUS	
06_3EH.....	See Table 35-24
IA32_MC29_ADDR / MSR_MC29_ADDR	
06_3EH.....	See Table 35-25
IA32_MC29_CTL / MSR_MC29_CTL	
06_3EH.....	See Table 35-25
IA32_MC29_MISC / MSR_MC29_MISC	
06_3EH.....	See Table 35-25
IA32_MC29_STATUS / MSR_MC29_STATUS	



<b>MSR Name and CPUID DisplayFamily_DisplayModel</b>	<b>Location</b>
06_3EH.....	See Table 35-25
<b>IA32_MC3_ADDR / MSR_MC3_ADDR</b>	
06_0FH, 06_17H .....	See Table 35-3
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H .....	See Table 35-4
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH .....	See Table 35-6
06_1AH, 06_1EH, 06_1FH, 06_2EH .....	See Table 35-13
06_57H.....	See Table 35-40
06_0EH.....	See Table 35-44
06_09H.....	See Table 35-45
<b>IA32_MC3_CTL / MSR_MC3_CTL</b>	
06_0FH, 06_17H .....	See Table 35-3
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H .....	See Table 35-4
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH .....	See Table 35-6
06_1AH, 06_1EH, 06_1FH, 06_2EH .....	See Table 35-13
06_57H.....	See Table 35-40
06_0EH.....	See Table 35-44
06_09H.....	See Table 35-45
<b>IA32_MC3_MISC / MSR_MC3_MISC</b>	
06_0FH, 06_17H .....	See Table 35-3
06_1AH, 06_1EH, 06_1FH, 06_2EH .....	See Table 35-13
06_0EH.....	See Table 35-44
<b>IA32_MC3_STATUS / MSR_MC3_STATUS</b>	
06_0FH, 06_17H .....	See Table 35-3
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H .....	See Table 35-4
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH .....	See Table 35-6
06_1AH, 06_1EH, 06_1FH, 06_2EH .....	See Table 35-13
06_57H.....	See Table 35-40
06_0EH.....	See Table 35-44
06_09H.....	See Table 35-45
<b>IA32_MC30_ADDR / MSR_MC30_ADDR</b>	
06_3EH.....	See Table 35-25
<b>IA32_MC30_CTL / MSR_MC30_CTL</b>	
06_3EH.....	See Table 35-25
<b>IA32_MC30_MISC / MSR_MC30_MISC</b>	
06_3EH.....	See Table 35-25
<b>IA32_MC30_STATUS / MSR_MC30_STATUS</b>	
06_3EH.....	See Table 35-25
<b>IA32_MC31_ADDR / MSR_MC31_ADDR</b>	
06_3EH.....	See Table 35-25
<b>IA32_MC31_CTL / MSR_MC31_CTL</b>	
06_3EH.....	See Table 35-25
<b>IA32_MC31_MISC / MSR_MC31_MISC</b>	

<b>MSR Name and CPUID DisplayFamily_DisplayModel</b>	<b>Location</b>
06_3EH.....	See Table 35-25
<b>IA32_MC31_STATUS / MSR_MC31_STATUS</b>	
06_3EH.....	See Table 35-25
<b>IA32_MC4_ADDR / MSR_MC4_ADDR</b>	
06_0FH, 06_17H.....	See Table 35-3
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H.....	See Table 35-4
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH.....	See Table 35-6
06_1AH, 06_1EH, 06_1FH, 06_2EH.....	See Table 35-13
06_57H.....	See Table 35-40
06_0EH.....	See Table 35-44
06_09H.....	See Table 35-45
<b>IA32_MC4_CTL / MSR_MC4_CTL</b>	
06_0FH, 06_17H.....	See Table 35-3
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H.....	See Table 35-4
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH.....	See Table 35-6
06_1AH, 06_1EH, 06_1FH, 06_2EH.....	See Table 35-13
06_57H.....	See Table 35-40
06_0EH.....	See Table 35-44
06_09H.....	See Table 35-45
<b>IA32_MC4_CTL2 / MSR_MC4_CTL2</b>	
06_2AH, 06_2DH.....	See Table 35-18
<b>IA32_MC4_STATUS / MSR_MC4_STATUS</b>	
06_0FH, 06_17H.....	See Table 35-3
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H.....	See Table 35-4
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH.....	See Table 35-6
06_1AH, 06_1EH, 06_1FH, 06_2EH.....	See Table 35-13
06_57H.....	See Table 35-40
06_0EH.....	See Table 35-44
06_09H.....	See Table 35-45
<b>MSR_MC5_ADDR / MSR_MC5_ADDR</b>	
06_0FH, 06_17H.....	See Table 35-3
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH.....	See Table 35-6
06_1AH, 06_1EH, 06_1FH, 06_2EH.....	See Table 35-13
06_2DH.....	See Table 35-21
06_3EH.....	See Table 35-24
06_3FH.....	See Table 35-30
06_4FH.....	See Table 35-36
06_57H.....	See Table 35-40
06_0EH.....	See Table 35-44
<b>IA32_MC5_CTL / MSR_MC5_CTL</b>	
06_0FH, 06_17H.....	See Table 35-3
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH.....	See Table 35-6

<b>MSR Name and CPUID DisplayFamily_DisplayModel</b>	<b>Location</b>
06_1AH, 06_1EH, 06_1FH, 06_2EH .....	See Table 35-13
06_2DH.....	See Table 35-21
06_3EH.....	See Table 35-24
06_3FH.....	See Table 35-30
06_4FH.....	See Table 35-36
06_57H.....	See Table 35-40
06_0EH.....	See Table 35-44
<b>IA32_MC5_MISC / MSR_MC5_MISC</b>	
06_0FH, 06_17H .....	See Table 35-3
06_1AH, 06_1EH, 06_1FH, 06_2EH .....	See Table 35-13
06_2DH.....	See Table 35-21
06_3EH.....	See Table 35-24
06_3FH.....	See Table 35-30
06_4FH.....	See Table 35-36
06_0EH.....	See Table 35-44
<b>IA32_MC5_STATUS / MSR_MC5_STATUS</b>	
06_0FH, 06_17H .....	See Table 35-3
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH .....	See Table 35-6
06_1AH, 06_1EH, 06_1FH, 06_2EH .....	See Table 35-13
06_2DH.....	See Table 35-21
06_3EH.....	See Table 35-24
06_3FH.....	See Table 35-30
06_4FH.....	See Table 35-36
06_57H.....	See Table 35-40
06_0EH.....	See Table 35-44
<b>IA32_MC6_ADDR / MSR_MC6_ADDR</b>	
06_1AH, 06_1EH, 06_1FH, 06_2EH .....	See Table 35-13
06_2DH.....	See Table 35-21
06_3EH.....	See Table 35-24
06_3F.....	See Table 35-30
06_56H, 06_4FH .....	See Table 35-35
06_4FH.....	See Table 35-36
<b>IA32_MC6_CTL / MSR_MC6_CTL</b>	
06_1AH, 06_1EH, 06_1FH, 06_2EH .....	See Table 35-13
06_2DH.....	See Table 35-21
06_3EH.....	See Table 35-24
06_3F.....	See Table 35-30
06_56H, 06_4FH .....	See Table 35-35
06_4FH.....	See Table 35-36
<b>MSR_MC6_DEMOTION_POLICY_CONFIG</b>	
06_37H.....	See Table 35-9
<b>IA32_MC6_MISC / MSR_MC6_MISC</b>	

<b>MSR Name and CPUID DisplayFamily_DisplayModel</b>	<b>Location</b>
06_1AH, 06_1EH, 06_1FH, 06_2EH .....	See Table 35-13
06_2DH.....	See Table 35-21
06_3EH.....	See Table 35-24
06_3F.....	See Table 35-30
06_56H, 06_4FH .....	See Table 35-35
06_4FH.....	See Table 35-36
<b>MSR_MC6_RESIDENCY_COUNTER</b>	
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH .....	See Table 35-7
06_37H.....	See Table 35-9
06_57H.....	See Table 35-40
<b>IA32_MC6_STATUS / MSR_MC6_STATUS</b>	
06_0FH, 06_17H .....	See Table 35-3
06_1AH, 06_1EH, 06_1FH, 06_2EH .....	See Table 35-13
06_2DH.....	See Table 35-21
06_3EH.....	See Table 35-24
06_3FH.....	See Table 35-30
06_56H, 06_4FH .....	See Table 35-35
06_4FH.....	See Table 35-36
<b>IA32_MC7_ADDR / MSR_MC7_ADDR</b>	
06_1AH, 06_1EH, 06_1FH, 06_2EH .....	See Table 35-13
06_2DH.....	See Table 35-21
06_3EH.....	See Table 35-24
06_3F.....	See Table 35-30
06_56H, 06_4FH .....	See Table 35-35
06_4FH.....	See Table 35-36
<b>IA32_MC7_CTL / MSR_MC7_CTL</b>	
06_1AH, 06_1EH, 06_1FH, 06_2EH .....	See Table 35-13
06_2DH.....	See Table 35-21
06_3EH.....	See Table 35-24
06_3F.....	See Table 35-30
06_56H, 06_4FH .....	See Table 35-35
06_4FH.....	See Table 35-36
<b>IA32_MC7_MISC / MSR_MC7_MISC</b>	
06_1AH, 06_1EH, 06_1FH, 06_2EH .....	See Table 35-13
06_2DH.....	See Table 35-21
06_3EH.....	See Table 35-24
06_3F.....	See Table 35-30
06_56H, 06_4FH .....	See Table 35-35
06_4FH.....	See Table 35-36
<b>IA32_MC7_STATUS / MSR_MC7_STATUS</b>	
06_1AH, 06_1EH, 06_1FH, 06_2EH .....	See Table 35-13
06_2DH.....	See Table 35-21

<b>MSR Name and CPUID DisplayFamily_DisplayModel</b>	<b>Location</b>
06_3EH.....	See Table 35-24
06_3F.....	See Table 35-30
06_56H, 06_4FH.....	See Table 35-35
06_4FH.....	See Table 35-36
<b>IA32_MC8_ADDR / MSR_MC8_ADDR</b>	
06_1AH, 06_1EH, 06_1FH, 06_2EH.....	See Table 35-13
06_2DH.....	See Table 35-21
06_3EH.....	See Table 35-24
06_3F.....	See Table 35-30
06_4FH.....	See Table 35-36
<b>IA32_MC8_CTL / MSR_MC8_CTL</b>	
06_1AH, 06_1EH, 06_1FH, 06_2EH.....	See Table 35-13
06_2DH.....	See Table 35-21
06_3EH.....	See Table 35-24
06_3F.....	See Table 35-30
06_4FH.....	See Table 35-36
<b>IA32_MC8_MISC / MSR_MC8_MISC</b>	
06_1AH, 06_1EH, 06_1FH, 06_2EH.....	See Table 35-13
06_2DH.....	See Table 35-21
06_3EH.....	See Table 35-24
06_3F.....	See Table 35-30
06_4FH.....	See Table 35-36
<b>IA32_MC8_STATUS / MSR_MC8_STATUS</b>	
06_1AH, 06_1EH, 06_1FH, 06_2EH.....	See Table 35-13
06_2DH.....	See Table 35-21
06_3EH.....	See Table 35-24
06_3F.....	See Table 35-30
06_4FH.....	See Table 35-36
<b>IA32_MC9_ADDR / MSR_MC9_ADDR</b>	
06_2EH.....	See Table 35-15
06_2DH.....	See Table 35-21
06_3EH.....	See Table 35-24
06_3F.....	See Table 35-30
06_56H, 06_4FH.....	See Table 35-35
06_4FH.....	See Table 35-36
<b>IA32_MC9_CTL / MSR_MC9_CTL</b>	
06_2EH.....	See Table 35-15
06_2DH.....	See Table 35-21
06_3EH.....	See Table 35-24
06_3F.....	See Table 35-30
06_56H, 06_4FH.....	See Table 35-35
06_4FH.....	See Table 35-36

<b>MSR Name and CPUID DisplayFamily_DisplayModel</b>	<b>Location</b>
IA32_MC9_MISC / MSR_MC9_MISC	
06_2EH.....	See Table 35-15
06_2DH.....	See Table 35-21
06_3EH.....	See Table 35-24
06_3F.....	See Table 35-30
06_56H, 06_4FH.....	See Table 35-35
06_4FH.....	See Table 35-36
IA32_MC9_STATUS / MSR_MC9_STATUS	
06_2EH.....	See Table 35-15
06_2DH.....	See Table 35-21
06_3EH.....	See Table 35-24
06_3F.....	See Table 35-30
06_56H, 06_4FH.....	See Table 35-35
06_4FH.....	See Table 35-36
MSR_MCG_MISC	
0FH.....	See Table 35-41
MSR_MCG_R10	
0FH.....	See Table 35-41
MSR_MCG_R11	
0FH.....	See Table 35-41
MSR_MCG_R12	
0FH.....	See Table 35-41
MSR_MCG_R13	
0FH.....	See Table 35-41
MSR_MCG_R14	
0FH.....	See Table 35-41
MSR_MCG_R15	
0FH.....	See Table 35-41
MSR_MCG_R8	
0FH.....	See Table 35-41
MSR_MCG_R9	
0FH.....	See Table 35-41
MSR_MCG_RAX	
0FH.....	See Table 35-41
MSR_MCG_RBP	
0FH.....	See Table 35-41
MSR_MCG_RBX	
0FH.....	See Table 35-41
MSR_MCG_RCX	
0FH.....	See Table 35-41
MSR_MCG_RDI	
0FH.....	See Table 35-41

## MODEL-SPECIFIC REGISTERS (MSRS)

MSR Name and CPUID DisplayFamily_DisplayModel	Location
MSR_MCG_RDX	
OFH.....	See Table 35-41
MSR_MCG_RESERVED1 - MSR_MCG_RESERVED5	
OFH.....	See Table 35-41
MSR_MCG_RFLAGS	
OFH.....	See Table 35-41
MSR_MCG_RIP	
OFH.....	See Table 35-41
MSR_MCG_RSI	
OFH.....	See Table 35-41
MSR_MCG_RSP	
OFH.....	See Table 35-41
MSR_MISC_FEATURE_CONTROL	
06_5CH.....	See Table 35-12
06_1AH, 06_1EH, 06_1FH, 06_2EH.....	See Table 35-13
06_2AH, 06_2DH.....	See Table 35-18
MSR_MISC_PWR_MGMT	
06_5CH.....	See Table 35-12
06_1AH, 06_1EH, 06_1FH, 06_2EH.....	See Table 35-13
06_2AH, 06_2DH.....	See Table 35-18
MSR_MOB_ESCRO	
OFH.....	See Table 35-41
MSR_MOB_ESCR1	
OFH.....	See Table 35-41
MSR_MS_CCCRO	
OFH.....	See Table 35-41
MSR_MS_CCCR1	
OFH.....	See Table 35-41
MSR_MS_CCCR2	
OFH.....	See Table 35-41
MSR_MS_CCCR3	
OFH.....	See Table 35-41
MSR_MS_COUNTER0	
OFH.....	See Table 35-41
MSR_MS_COUNTER1	
OFH.....	See Table 35-41
MSR_MS_COUNTER2	
OFH.....	See Table 35-41
MSR_MS_COUNTER3	
OFH.....	See Table 35-41
MSR_MS_ESCRO	
OFH.....	See Table 35-41

<b>MSR Name and CPUID DisplayFamily_DisplayModel</b>	<b>Location</b>
MSR_MS_ESCR1	
0FH.....	See Table 35-41
MSR_MTRRCAP	
06_4EH, 06_5EH.....	See Table 35-37
MSR_OFFCORE_RSP_0	
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH.....	See Table 35-6
06_1AH, 06_1EH, 06_1FH, 06_2EH.....	See Table 35-13
06_2AH, 06_2DH.....	See Table 35-18
06_57H.....	See Table 35-40
MSR_OFFCORE_RSP_1	
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH.....	See Table 35-6
06_25H, 06_2CH.....	See Table 35-16
06_2FH.....	See Table 35-17
06_2AH, 06_2DH.....	See Table 35-18
06_57H.....	See Table 35-40
MSR_PCIE_PLL_RATIO	
06_3FH.....	See Table 35-30
MSR_PCU_PMON_BOX_CTL	
06_2DH.....	See Table 35-22
06_3FH.....	See Table 35-31
MSR_PCU_PMON_BOX_FILTER	
06_2DH.....	See Table 35-22
06_3FH.....	See Table 35-31
MSR_PCU_PMON_BOX_STATUS	
06_3EH.....	See Table 35-26
06_3FH.....	See Table 35-31
MSR_PCU_PMON_CTR0	
06_2DH.....	See Table 35-22
06_3FH.....	See Table 35-31
MSR_PCU_PMON_CTR1	
06_2DH.....	See Table 35-22
06_3FH.....	See Table 35-31
MSR_PCU_PMON_CTR2	
06_2DH.....	See Table 35-22
06_3FH.....	See Table 35-31
MSR_PCU_PMON_CTR3	
06_2DH.....	See Table 35-22
06_3FH.....	See Table 35-31
MSR_PCU_PMON_EVNTSELO	
06_2DH.....	See Table 35-22
06_3FH.....	See Table 35-31
MSR_PCU_PMON_EVNTSEL1	



## MODEL-SPECIFIC REGISTERS (MSRS)

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_2DH.....	See Table 35-22
06_3FH.....	See Table 35-31
MSR_PCU_PMON_EVNTSEL2	
06_2DH.....	See Table 35-22
06_3FH.....	See Table 35-31
MSR_PCU_PMON_EVNTSEL3	
06_2DH.....	See Table 35-22
06_3FH.....	See Table 35-31
MSR_PEBS_ENABLE	
06_0FH, 06_17H .....	See Table 35-3
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H .....	See Table 35-4
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH .....	See Table 35-7
06_5CH.....	See Table 35-12
06_1AH, 06_1EH, 06_1FH, 06_2EH .....	See Table 35-13
06_2AH, 06_2DH .....	See Table 35-18
06_3EH.....	See Table 35-25
06_57H.....	See Table 35-40
0FH.....	See Table 35-41
MSR_PEBS_FRONTEND	
06_4EH, 06_5EH.....	See Table 35-37
MSR_PEBS_LD_LAT	
06_1AH, 06_1EH, 06_1FH, 06_2EH .....	See Table 35-13
06_2AH, 06_2DH .....	See Table 35-18
MSR_PEBS_MATRIX_VERT	
0FH.....	See Table 35-41
MSR_PEBS_NUM_ALT	
06_2DH.....	See Table 35-21
MSR_PERF_CAPABILITIES	
06_0FH, 06_17H .....	See Table 35-3
MSR_PERF_FIXED_CTR_CTRL	
06_0FH, 06_17H .....	See Table 35-3
MSR_PERF_FIXED_CTR0	
06_0FH, 06_17H .....	See Table 35-3
MSR_PERF_FIXED_CTR1	
06_0FH, 06_17H .....	See Table 35-3
MSR_PERF_FIXED_CTR2	
06_0FH, 06_17H .....	See Table 35-3
MSR_PERF_GLOBAL_CTRL	
06_0FH, 06_17H .....	See Table 35-3
MSR_PERF_GLOBAL_OVF_CTRL	
06_0FH, 06_17H .....	See Table 35-3
06_1AH, 06_1EH, 06_1FH, 06_2EH .....	See Table 35-13

<b>MSR Name and CPUID DisplayFamily_DisplayModel</b>	<b>Location</b>
<b>MSR_PERF_GLOBAL_STATUS</b>	
06_0FH, 06_17H .....	See Table 35-3
06_1AH, 06_1EH, 06_1FH, 06_2EH .....	See Table 35-13
<b>MSR_PERF_STATUS</b>	
06_0FH, 06_17H .....	See Table 35-3
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H .....	See Table 35-4
06_2AH, 06_2DH .....	See Table 35-18
<b>MSR_PKG_C10_RESIDENCY</b>	
06_5CH .....	See Table 35-12
06_45H .....	See Table 35-28 and Table 35-29
06_4FH .....	See Table 35-36
<b>MSR_PKG_C2_RESIDENCY</b>	
06_27H .....	See Table 35-5
06_5CH .....	See Table 35-12
06_2AH, 06_2DH, 06_3AH, 06_3CH, 06_3EH, 06_3FH, 06_45H, 06_46H .....	See Table 35-18
06_57H .....	See Table 35-40
<b>MSR_PKG_C3_RESIDENCY</b>	
06_5CH .....	See Table 35-12
06_1AH, 06_1EH, 06_1FH, 06_2EH, 06_25H, 06_2CH, 06_2FH .....	See Table 35-13
06_2AH, 06_2DH, 06_3AH, 06_3CH, 06_3EH, 06_3FH, 06_45H, 06_46H .....	See Table 35-18
06_57H .....	See Table 35-40
<b>MSR_PKG_C4_RESIDENCY</b>	
06_27H .....	See Table 35-5
<b>MSR_PKG_C6_RESIDENCY</b>	
06_27H .....	See Table 35-5
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH .....	See Table 35-7
06_5CH .....	See Table 35-12
06_1AH, 06_1EH, 06_1FH, 06_2EH, 06_25H, 06_2CH, 06_2FH .....	See Table 35-13
06_2AH, 06_2DH, 06_3AH, 06_3CH, 06_3EH, 06_3FH, 06_45H, 06_46H .....	See Table 35-18
06_57H .....	See Table 35-40
<b>MSR_PKG_C7_RESIDENCY</b>	
06_1AH, 06_1EH, 06_1FH, 06_2EH, 06_25H, 06_2CH, 06_2FH .....	See Table 35-13
06_2AH, 06_2DH, 06_3AH, 06_3CH, 06_3EH, 06_3FH, 06_45H, 06_46H .....	See Table 35-18
06_57H .....	See Table 35-40
<b>MSR_PKG_C8_RESIDENCY</b>	
06_45H .....	See Table 35-29
06_4FH .....	See Table 35-36
<b>MSR_PKG_C9_RESIDENCY</b>	
06_45H .....	See Table 35-29
06_4FH .....	See Table 35-36
<b>MSR_PKG_CST_CONFIG_CONTROL</b>	

<b>MSR Name and CPUID DisplayFamily_DisplayModel</b>	<b>Location</b>
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH .....	See Table 35-7
06_4CH.....	See Table 35-11
06_5CH.....	See Table 35-12
06_1AH, 06_1EH, 06_1FH, 06_2EH .....	See Table 35-13
06_2AH, 06_2DH .....	See Table 35-18
06_3AH.....	See Table 35-23
06_3EH.....	See Table 35-24
06_3CH, 06_45H, 06_46H .....	See Table 35-28
06_45H.....	See Table 35-29
06_3F.....	See Table 35-30
06_3DH.....	See Table 35-33
06_56H, 06_4FH .....	See Table 35-34
06_57H.....	See Table 35-40
<b>MSR_PKG_ENERGY_STATUS</b>	
06_37H, 06_4AH, 06_5AH, 06_5DH .....	See Table 35-8
06_5CH.....	See Table 35-12
06_2AH, 06_2DH, 06_3AH, 06_3CH, 06_3EH, 06_3FH, 06_45H, 06_46H .....	See Table 35-18
<b>MSR_PKG_HDC_CONFIG</b>	
06_4EH, 06_5EH.....	See Table 35-37
<b>MSR_PKG_HDC_DEEP_RESIDENCY</b>	
06_4EH, 06_5EH .....	See Table 35-37
<b>MSR_PKG_HDC_SHALLOW_RESIDENCY</b>	
06_4EH, 06_5EH .....	See Table 35-37
<b>MSR_PKG_PERF_STATUS</b>	
06_5CH.....	See Table 35-12
06_2DH.....	See Table 35-21
06_3EH, 06_3FH .....	See Table 35-24
06_3CH, 06_45H, 06_46H .....	See Table 35-28
06_57H.....	See Table 35-40
<b>MSR_PKG_POWER_INFO</b>	
06_4DH.....	See Table 35-10
06_5CH.....	See Table 35-12
06_2AH, 06_2DH, 06_3AH, 06_3CH, 06_3EH, 06_3FH, 06_45H, 06_46H .....	See Table 35-18
06_57H.....	See Table 35-40
<b>MSR_PKG_POWER_LIMIT</b>	
06_37H, 06_4AH, 06_5AH, 06_5DH .....	See Table 35-8
06_4DH.....	See Table 35-10
06_5CH.....	See Table 35-12
06_2AH, 06_2DH, 06_3AH, 06_3CH, 06_3EH, 06_3FH, 06_45H, 06_46H .....	See Table 35-18
06_57H.....	See Table 35-40
<b>MSR_PKGCC_IRTL1</b>	
06_5CH.....	See Table 35-12

<b>MSR Name and CPUID DisplayFamily_DisplayModel</b>	<b>Location</b>
06_3CH, 06_45H, 06_46H. ....	See Table 35-27
<b>MSR_PKG_C2_IRTL2</b>	
06_5CH. ....	See Table 35-12
06_3CH, 06_45H, 06_46H. ....	See Table 35-27
<b>MSR_PKG_C3_IRTL</b>	
06_5CH. ....	See Table 35-12
06_2AH, 06_2DH. ....	See Table 35-18
<b>MSR_PKG_C6_IRTL</b>	
06_2AH, 06_2DH. ....	See Table 35-18
<b>MSR_PKG_C7_IRTL</b>	
06_2AH. ....	See Table 35-19
<b>MSR_PLATFORM_BRV</b>	
0FH. ....	See Table 35-41
<b>MSR_PLATFORM_ENERGY_COUNTER</b>	
06_4EH, 06_5EH. ....	See Table 35-37
<b>MSR_PLATFORM_ID</b>	
06_0FH, 06_17H. ....	See Table 35-3
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H. ....	See Table 35-4
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH. ....	See Table 35-7
06_5CH. ....	See Table 35-12
06_1AH, 06_1EH, 06_1FH, 06_2EH. ....	See Table 35-13
<b>MSR_PLATFORM_INFO</b>	
06_5CH. ....	See Table 35-12
06_1AH, 06_1EH, 06_1FH, 06_2EH. ....	See Table 35-13
06_2AH, 06_2DH. ....	See Table 35-18
06_3AH. ....	See Table 35-23
06_3EH. ....	See Table 35-24
06_3CH, 06_45H, 06_46H. ....	See Table 35-27 and Table 35-28
06_56H, 06_4FH. ....	See Table 35-34
06_57H. ....	See Table 35-40
<b>MSR_PLATFORM_POWER_LIMIT</b>	
06_4EH, 06_5EH. ....	See Table 35-37
<b>MSR_PMG_IO_CAPTURE_BASE</b>	
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH. ....	See Table 35-6
06_4CH. ....	See Table 35-11
06_1AH, 06_1EH, 06_1FH, 06_2EH. ....	See Table 35-13
06_2AH, 06_2DH. ....	See Table 35-18
06_3AH. ....	See Table 35-23
06_3EH. ....	See Table 35-24
06_57H. ....	See Table 35-40
<b>MSR_PMH_ESCRO</b>	

## MODEL-SPECIFIC REGISTERS (MSRS)

MSR Name and CPUID DisplayFamily_DisplayModel	Location
0FH.....	See Table 35-41
MSR_PMH_ESCR1	
0FH.....	See Table 35-41
MSR_PMON_GLOBAL_CONFIG	
06_3EH.....	See Table 35-26
06_3FH.....	See Table 35-31
MSR_PMON_GLOBAL_CTL	
06_3EH.....	See Table 35-26
06_3FH.....	See Table 35-31
MSR_PMON_GLOBAL_STATUS	
06_3EH.....	See Table 35-26
06_3FH.....	See Table 35-31
MSR_POWER_CTL	
06_5CH.....	See Table 35-12
06_1AH, 06_1EH, 06_1FH, 06_2EH.....	See Table 35-13
06_2AH, 06_2DH.....	See Table 35-18
MSR_PPO_ENERGY_STATUS	
06_37H, 06_4AH, 06_5AH, 06_5DH.....	See Table 35-8
06_5CH.....	See Table 35-12
06_2AH, 06_2DH, 06_3AH, 06_3CH, 06_3EH, 06_3FH, 06_45H, 06_46H.....	See Table 35-18
06_57H.....	See Table 35-40
MSR_PPO_POLICY	
06_2AH, 06_45H.....	See Table 35-19
MSR_PPO_POWER_LIMIT	
06_4CH.....	See Table 35-11
06_2AH, 06_2DH, 06_3AH, 06_3CH, 06_3EH, 06_3FH, 06_45H, 06_46H.....	See Table 35-18
06_57H.....	See Table 35-40
MSR_PP1_ENERGY_STATUS	
06_5CH.....	See Table 35-12
06_2AH, 06_45H.....	See Table 35-19
06_3CH, 06_45H, 06_46H.....	See Table 35-28
MSR_PP1_POLICY	
06_2AH, 06_45H.....	See Table 35-19
06_3CH, 06_45H, 06_46H.....	See Table 35-28
MSR_PP1_POWER_LIMIT	
06_2AH, 06_45H.....	See Table 35-19
06_3CH, 06_45H, 06_46H.....	See Table 35-28
MSR_PPERF	
06_4EH, 06_5EH.....	See Table 35-37
MSR_PPIN	
06_3EH.....	See Table 35-24
06_56H, 06_4FH.....	See Table 35-34

<b>MSR Name and CPUID DisplayFamily_DisplayModel</b>	<b>Location</b>
MSR_PPIN_CTL	
06_3EH.....	See Table 35-24
06_56H, 06_4FH.....	See Table 35-34
MSR_RO_PMON_BOX_CTRL	
06_2EH.....	See Table 35-15
MSR_RO_PMON_BOX_OVF_CTRL	
06_2EH.....	See Table 35-15
MSR_RO_PMON_BOX_STATUS	
06_2EH.....	See Table 35-15
MSR_RO_PMON_CTRL0	
06_2EH.....	See Table 35-15
MSR_RO_PMON_CTRL1	
06_2EH.....	See Table 35-15
MSR_RO_PMON_CTRL2	
06_2EH.....	See Table 35-15
MSR_RO_PMON_CTRL3	
06_2EH.....	See Table 35-15
MSR_RO_PMON_CTRL4	
06_2EH.....	See Table 35-15
MSR_RO_PMON_CTRL5	
06_2EH.....	See Table 35-15
MSR_RO_PMON_CTRL6	
06_2EH.....	See Table 35-15
MSR_RO_PMON_CTRL7	
06_2EH.....	See Table 35-15
MSR_RO_PMON_EVNT_SEL0	
06_2EH.....	See Table 35-15
MSR_RO_PMON_EVNT_SEL1	
06_2EH.....	See Table 35-15
MSR_RO_PMON_EVNT_SEL2	
06_2EH.....	See Table 35-15
MSR_RO_PMON_EVNT_SEL3	
06_2EH.....	See Table 35-15
MSR_RO_PMON_EVNT_SEL4	
06_2EH.....	See Table 35-15
MSR_RO_PMON_EVNT_SEL5	
06_2EH.....	See Table 35-15
MSR_RO_PMON_EVNT_SEL6	
06_2EH.....	See Table 35-15
MSR_RO_PMON_EVNT_SEL7	
06_2EH.....	See Table 35-15
MSR_RO_PMON_IPERFO_PO	

## MODEL-SPECIFIC REGISTERS (MSRS)

<b>MSR Name and CPUID DisplayFamily_DisplayModel</b>	<b>Location</b>
06_2EH.....	See Table 35-15
MSR_RO_PMON_IPERFO_P1	
06_2EH.....	See Table 35-15
MSR_RO_PMON_IPERFO_P2	
06_2EH.....	See Table 35-15
MSR_RO_PMON_IPERFO_P3	
06_2EH.....	See Table 35-15
MSR_RO_PMON_IPERFO_P4	
06_2EH.....	See Table 35-15
MSR_RO_PMON_IPERFO_P5	
06_2EH.....	See Table 35-15
MSR_RO_PMON_IPERFO_P6	
06_2EH.....	See Table 35-15
MSR_RO_PMON_IPERFO_P7	
06_2EH.....	See Table 35-15
MSR_RO_PMON_QLX_P0	
06_2EH.....	See Table 35-15
MSR_RO_PMON_QLX_P1	
06_2EH.....	See Table 35-15
MSR_RO_PMON_QLX_P2	
06_2EH.....	See Table 35-15
MSR_RO_PMON_QLX_P3	
06_2EH.....	See Table 35-15
MSR_R1_PMON_BOX_CTRL	
06_2EH.....	See Table 35-15
MSR_R1_PMON_BOX_OVF_CTRL	
06_2EH.....	See Table 35-15
MSR_R1_PMON_BOX_STATUS	
06_2EH.....	See Table 35-15
MSR_R1_PMON_CTR10	
06_2EH.....	See Table 35-15
MSR_R1_PMON_CTR11	
06_2EH.....	See Table 35-15
MSR_R1_PMON_CTR12	
06_2EH.....	See Table 35-15
MSR_R1_PMON_CTR13	
06_2EH.....	See Table 35-15
MSR_R1_PMON_CTR14	
06_2EH.....	See Table 35-15
MSR_R1_PMON_CTR15	
06_2EH.....	See Table 35-15
MSR_R1_PMON_CTR8	

<b>MSR Name and CPUID DisplayFamily_DisplayModel</b>	<b>Location</b>
06_2EH.....	See Table 35-15
MSR_R1_PMON_CTR9	
06_2EH.....	See Table 35-15
MSR_R1_PMON_EVTN_SEL10	
06_2EH.....	See Table 35-15
MSR_R1_PMON_EVTN_SEL11	
06_2EH.....	See Table 35-15
MSR_R1_PMON_EVTN_SEL12	
06_2EH.....	See Table 35-15
MSR_R1_PMON_EVTN_SEL13	
06_2EH.....	See Table 35-15
MSR_R1_PMON_EVTN_SEL14	
06_2EH.....	See Table 35-15
MSR_R1_PMON_EVTN_SEL15	
06_2EH.....	See Table 35-15
MSR_R1_PMON_EVTN_SEL8	
06_2EH.....	See Table 35-15
MSR_R1_PMON_EVTN_SEL9	
06_2EH.....	See Table 35-15
MSR_R1_PMON_IPERF1_P10	
06_2EH.....	See Table 35-15
MSR_R1_PMON_IPERF1_P11	
06_2EH.....	See Table 35-15
MSR_R1_PMON_IPERF1_P12	
06_2EH.....	See Table 35-15
MSR_R1_PMON_IPERF1_P13	
06_2EH.....	See Table 35-15
MSR_R1_PMON_IPERF1_P14	
06_2EH.....	See Table 35-15
MSR_R1_PMON_IPERF1_P15	
06_2EH.....	See Table 35-15
MSR_R1_PMON_IPERF1_P8	
06_2EH.....	See Table 35-15
MSR_R1_PMON_IPERF1_P9	
06_2EH.....	See Table 35-15
MSR_R1_PMON_QLX_P4	
06_2EH.....	See Table 35-15
MSR_R1_PMON_QLX_P5	
06_2EH.....	See Table 35-15
MSR_R1_PMON_QLX_P6	
06_2EH.....	See Table 35-15
MSR_R1_PMON_QLX_P7	



## MODEL-SPECIFIC REGISTERS (MSRS)

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_2EH.....	See Table 35-15
MSR_RAPL_POWER_UNIT	
06_37H, 06_4AH, 06_5AH, 06_5DH.....	See Table 35-8
06_4DH.....	See Table 35-10
06_5CH.....	See Table 35-12
06_2AH, 06_2DH, 06_3AH, 06_3CH, 06_3EH, 06_3FH, 06_45H, 06_46H.....	See Table 35-18
06_3FH.....	See Table 35-30
06_56H, 06_4FH.....	See Table 35-34
06_57H.....	See Table 35-40
MSR_RAT_ESCR0	
0FH.....	See Table 35-41
MSR_RAT_ESCR1	
0FH.....	See Table 35-41
MSR_RING_PERF_LIMIT_REASONS	
06_3CH, 06_45H, 06_46H.....	See Table 35-28
MSR_SO_PMON_BOX_CTRL	
06_2EH.....	See Table 35-15
06_3FH.....	See Table 35-31
MSR_SO_PMON_BOX_FILTER	
06_3FH.....	See Table 35-31
MSR_SO_PMON_BOX_OVF_CTRL	
06_2EH.....	See Table 35-15
MSR_SO_PMON_BOX_STATUS	
06_2EH.....	See Table 35-15
MSR_SO_PMON_CTR0	
06_2EH.....	See Table 35-15
06_3FH.....	See Table 35-31
MSR_SO_PMON_CTR1	
06_2EH.....	See Table 35-15
06_3FH.....	See Table 35-31
MSR_SO_PMON_CTR2	
06_2EH.....	See Table 35-15
06_3FH.....	See Table 35-31
MSR_SO_PMON_CTR3	
06_2EH.....	See Table 35-15
06_3FH.....	See Table 35-31
MSR_SO_PMON_EVNT_SELO	
06_2EH.....	See Table 35-15
06_3FH.....	See Table 35-31
MSR_SO_PMON_EVNT_SEL1	
06_2EH.....	See Table 35-15
06_3FH.....	See Table 35-31

<b>MSR Name and CPUID DisplayFamily_DisplayModel</b>	<b>Location</b>
MSR_S0_PMON_EVNT_SEL2	
06_2EH.....	See Table 35-15
06_3FH.....	See Table 35-31
MSR_S0_PMON_EVNT_SEL3	
06_2EH.....	See Table 35-15
06_3FH.....	See Table 35-31
MSR_S0_PMON_MASK	
06_2EH.....	See Table 35-15
MSR_S0_PMON_MATCH	
06_2EH.....	See Table 35-15
MSR_S1_PMON_BOX_CTRL	
06_2EH.....	See Table 35-15
06_3FH.....	See Table 35-31
MSR_S1_PMON_BOX_FILTER	
06_3FH.....	See Table 35-31
MSR_S1_PMON_BOX_OVF_CTRL	
06_2EH.....	See Table 35-15
MSR_S1_PMON_BOX_STATUS	
06_2EH.....	See Table 35-15
MSR_S1_PMON_CTRL0	
06_2EH.....	See Table 35-15
06_3FH.....	See Table 35-31
MSR_S1_PMON_CTRL1	
06_2EH.....	See Table 35-15
06_3FH.....	See Table 35-31
MSR_S1_PMON_CTRL2	
06_2EH.....	See Table 35-15
06_3FH.....	See Table 35-31
MSR_S1_PMON_CTRL3	
06_2EH.....	See Table 35-15
06_3FH.....	See Table 35-31
MSR_S1_PMON_EVNT_SELO	
06_2EH.....	See Table 35-15
06_3FH.....	See Table 35-31
MSR_S1_PMON_EVNT_SEL1	
06_2EH.....	See Table 35-15
06_3FH.....	See Table 35-31
MSR_S1_PMON_EVNT_SEL2	
06_2EH.....	See Table 35-15
06_3FH.....	See Table 35-31
MSR_S1_PMON_EVNT_SEL3	
06_2EH.....	See Table 35-15

## MODEL-SPECIFIC REGISTERS (MSRS)

<b>MSR Name and CPUID DisplayFamily_DisplayModel</b>	<b>Location</b>
06_3FH.....	See Table 35-31
MSR_S1_PMON_MASK	
06_2EH.....	See Table 35-15
MSR_S1_PMON_MATCH	
06_2EH.....	See Table 35-15
MSR_S2_PMON_BOX_CTL	
06_3FH.....	See Table 35-31
MSR_S2_PMON_BOX_FILTER	
06_3FH.....	See Table 35-31
MSR_S2_PMON_CTR0	
06_3FH.....	See Table 35-31
MSR_S2_PMON_CTR1	
06_3FH.....	See Table 35-31
MSR_S2_PMON_CTR2	
06_3FH.....	See Table 35-31
MSR_S2_PMON_CTR3	
06_3FH.....	See Table 35-31
MSR_S2_PMON_EVTSELO	
06_3FH.....	See Table 35-31
MSR_S2_PMON_EVTSEL1	
06_3FH.....	See Table 35-31
MSR_S2_PMON_EVTSEL2	
06_3FH.....	See Table 35-31
MSR_S2_PMON_EVTSEL3	
06_3FH.....	See Table 35-31
MSR_S3_PMON_BOX_CTL	
06_3FH.....	See Table 35-31
MSR_S3_PMON_BOX_FILTER	
06_3FH.....	See Table 35-31
MSR_S3_PMON_CTR0	
06_3FH.....	See Table 35-31
MSR_S3_PMON_CTR1	
06_3FH.....	See Table 35-31
MSR_S3_PMON_CTR2	
06_3FH.....	See Table 35-31
MSR_S3_PMON_CTR3	
06_3FH.....	See Table 35-31
MSR_S3_PMON_EVTSELO	
06_3FH.....	See Table 35-31
MSR_S3_PMON_EVTSEL1	
06_3FH.....	See Table 35-31
MSR_S3_PMON_EVTSEL2	

<b>MSR Name and CPUID DisplayFamily_DisplayModel</b>	<b>Location</b>
06_3FH.....	See Table 35-31
MSR_S3_PMON_EVTSEL3	
06_3FH.....	See Table 35-31
MSR_SAA_T_ESCR0	
0FH.....	See Table 35-41
MSR_SAA_T_ESCR1	
0FH.....	See Table 35-41
MSR_SGXOWNER0	
06_5CH.....	See Table 35-12
06_4EH, 06_5EH.....	See Table 35-37
MSR_SGXOWNER1	
06_5CH.....	See Table 35-12
06_4EH, 06_5EH.....	See Table 35-37
MSR_SMI_COUNT	
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH.....	See Table 35-6
06_1AH, 06_1EH, 06_1FH, 06_2EH.....	See Table 35-13
06_2AH, 06_2DH.....	See Table 35-18
06_57H.....	See Table 35-40
MSR_SMM_BLOCKED	
06_5CH.....	See Table 35-12
06_3CH, 06_45H, 06_46H.....	See Table 35-28
MSR_SMM_DELAYED	
06_5CH.....	See Table 35-12
06_3CH, 06_45H, 06_46H.....	See Table 35-28
MSR_SMM_FEATURE_CONTROL	
06_5CH.....	See Table 35-12
06_3CH, 06_45H, 06_46H.....	See Table 35-28
MSR_SMM_MCA_CAP	
06_5CH.....	See Table 35-12
06_3CH, 06_45H, 06_46H.....	See Table 35-28
06_3FH.....	See Table 35-30
06_56H, 06_4FH.....	See Table 35-34
06_57H.....	See Table 35-40
MSR_SMRR_PHYSBASE	
06_0FH, 06_17H.....	See Table 35-3
MSR_SMRR_PHYSMASK	
06_0FH, 06_17H.....	See Table 35-3
MSR_SSU_ESCR0	
0FH.....	See Table 35-41
MSR_TBPU_ESCR0	
0FH.....	See Table 35-41
MSR_TBPU_ESCR1	

## MODEL-SPECIFIC REGISTERS (MSRS)

MSR Name and CPUID DisplayFamily_DisplayModel	Location
0FH.....	See Table 35-41
MSR_TC_ESCR0	
0FH.....	See Table 35-41
MSR_TC_ESCR1	
0FH.....	See Table 35-41
MSR_TC_PRECISE_EVENT	
0FH.....	See Table 35-41
MSR_TEMPERATURE_TARGET	
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH.....	See Table 35-6
06_1AH, 06_1EH, 06_1FH, 06_2EH.....	See Table 35-13
06_2AH, 06_2DH.....	See Table 35-18
06_3EH.....	See Table 35-24
06_56H, 06_4FH.....	See Table 35-34
06_57H.....	See Table 35-40
MSR_THERM2_CTL	
06_0FH, 06_17H.....	See Table 35-3
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H.....	See Table 35-4
0FH.....	See Table 35-41
06_0EH.....	See Table 35-44
06_09H.....	See Table 35-45
MSR_THREAD_ID_INFO	
06_3FH.....	See Table 35-30
MSR_TURBO_ACTIVATION_RATIO	
06_5CH.....	See Table 35-12
06_3AH.....	See Table 35-23
06_3CH, 06_45H, 06_46H.....	See Table 35-27
06_57H.....	See Table 35-40
MSR_TURBO_GROUP_CORECNT	
06_5CH.....	See Table 35-12
MSR_TURBO_POWER_CURRENT_LIMIT	
06_1AH, 06_1EH, 06_1FH, 06_2EH.....	See Table 35-13
MSR_TURBO_RATIO_LIMIT	
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH.....	See Table 35-6
06_4DH.....	See Table 35-10
06_5CH.....	See Table 35-12
06_1AH, 06_1EH, 06_1FH, 06_2EH, 06_25H, 06_2CH.....	See Table 35-13
06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH.....	See Table 35-14
06_2EH.....	See Table 35-15
06_25H, 06_2CH.....	See Table 35-16
06_2FH.....	See Table 35-17
06_2AH, 06_45H.....	See Table 35-19
06_2DH.....	See Table 35-21

<b>MSR Name and CPUID DisplayFamily_DisplayModel</b>	<b>Location</b>
06_3EH.....	See Table 35-24 and Table 35-25
06_3CH, 06_45H, 06_46H .....	See Table 35-28
06_3FH.....	See Table 35-30
06_3DH.....	See Table 35-33
06_56H, 06_4FH .....	See Table 35-34
06_57H.....	See Table 35-40
<b>MSR_TURBO_RATIO_LIMIT1</b>	
06_3EH.....	See Table 35-24 and Table 35-25
06_3FH.....	See Table 35-30
06_56H, 06_4FH .....	See Table 35-34
<b>MSR_TURBO_RATIO_LIMIT2</b>	
06_3FH.....	See Table 35-30
<b>MSR_TURBO_RATIO_LIMIT3</b>	
06_56H.....	See Table 35-35
06_4FH.....	See Table 35-36
<b>MSR_U_PMON_BOX_STATUS</b>	
06_3EH.....	See Table 35-26
06_3FH.....	See Table 35-31
<b>MSR_U_PMON_CTR</b>	
06_2EH.....	See Table 35-15
<b>MSR_U_PMON_CTR0</b>	
06_2DH.....	See Table 35-22
06_3FH.....	See Table 35-31
<b>MSR_U_PMON_CTR1</b>	
06_2DH.....	See Table 35-22
06_3FH.....	See Table 35-31
<b>MSR_U_PMON_EVNT_SEL</b>	
06_2EH.....	See Table 35-15
<b>MSR_U_PMON_EVNTSELO</b>	
06_2DH.....	See Table 35-22
06_3FH.....	See Table 35-31
<b>MSR_U_PMON_EVNTSEL1</b>	
06_2DH.....	See Table 35-22
06_3FH.....	See Table 35-31
<b>MSR_U_PMON_GLOBAL_CTRL</b>	
06_2EH.....	See Table 35-15
<b>MSR_U_PMON_GLOBAL_OVF_CTRL</b>	
06_2EH.....	See Table 35-15
<b>MSR_U_PMON_GLOBAL_STATUS</b>	
06_2EH.....	See Table 35-15

## MODEL-SPECIFIC REGISTERS (MSRS)

MSR Name and CPUID DisplayFamily_DisplayModel	Location
MSR_U_PMON_UCLK_FIXED_CTL	
06_2DH .....	See Table 35-22
06_3FH .....	See Table 35-31
MSR_U_PMON_UCLK_FIXED_CTR	
06_2DH .....	See Table 35-22
06_3FH .....	See Table 35-31
MSR_U2L_ESCR0	
0FH .....	See Table 35-41
MSR_U2L_ESCR1	
0FH .....	See Table 35-41
MSR_UNC_ARB_PERFCTR0	
06_2AH .....	See Table 35-20
06_3CH, 06_45H, 06_46H .....	See Table 35-28
06_4EH, 06_5EH .....	See Table 35-38
MSR_UNC_ARB_PERFCTR1	
06_2AH .....	See Table 35-20
06_3CH, 06_45H, 06_46H .....	See Table 35-28
06_4EH, 06_5EH .....	See Table 35-38
MSR_UNC_ARB_PERFEVTSELO	
06_2AH .....	See Table 35-20
06_3CH, 06_45H, 06_46H .....	See Table 35-28
06_4EH, 06_5EH .....	See Table 35-38
MSR_UNC_ARB_PERFEVTSEL1	
06_2AH .....	See Table 35-20
06_3CH, 06_45H, 06_46H .....	See Table 35-28
06_4EH, 06_5EH .....	See Table 35-38
MSR_UNC_CBO_0_PERFCTR0	
06_2AH .....	See Table 35-20
06_3CH, 06_45H, 06_46H .....	See Table 35-28
06_4EH, 06_5EH .....	See Table 35-38
MSR_UNC_CBO_0_PERFCTR1	
06_2AH .....	See Table 35-20
06_3CH, 06_45H, 06_46H .....	See Table 35-28
06_4EH, 06_5EH .....	See Table 35-38
MSR_UNC_CBO_0_PERFCTR2	
06_2AH .....	See Table 35-20
MSR_UNC_CBO_0_PERFCTR3	
06_2AH .....	See Table 35-20
MSR_UNC_CBO_0_PERFEVTSELO	
06_2AH .....	See Table 35-20
06_3CH, 06_45H, 06_46H .....	See Table 35-28
06_4EH, 06_5EH .....	See Table 35-38

<b>MSR Name and CPUID DisplayFamily_DisplayModel</b>	<b>Location</b>
MSR_UNC_CBO_0_PERFEVTSEL1	
06_2AH .....	See Table 35-20
06_3CH, 06_45H, 06_46H .....	See Table 35-28
06_4EH, 06_5EH .....	See Table 35-38
MSR_UNC_CBO_0_PERFEVTSEL2	
06_2AH .....	See Table 35-20
MSR_UNC_CBO_0_PERFEVTSEL3	
06_2AH .....	See Table 35-20
MSR_UNC_CBO_0_UNIT_STATUS	
06_2AH .....	See Table 35-20
MSR_UNC_CBO_1_PERFCTR0	
06_2AH .....	See Table 35-20
06_3CH, 06_45H, 06_46H .....	See Table 35-28
06_4EH, 06_5EH .....	See Table 35-38
MSR_UNC_CBO_1_PERFCTR1	
06_2AH .....	See Table 35-20
06_3CH, 06_45H, 06_46H .....	See Table 35-28
06_4EH, 06_5EH .....	See Table 35-38
MSR_UNC_CBO_1_PERFCTR2	
06_2AH .....	See Table 35-20
MSR_UNC_CBO_1_PERFCTR3	
06_2AH .....	See Table 35-20
MSR_UNC_CBO_1_PERFEVTSELO	
06_2AH .....	See Table 35-20
06_3CH, 06_45H, 06_46H .....	See Table 35-28
06_4EH, 06_5EH .....	See Table 35-38
MSR_UNC_CBO_1_PERFEVTSEL1	
06_2AH .....	See Table 35-20
06_3CH, 06_45H, 06_46H .....	See Table 35-28
06_4EH, 06_5EH .....	See Table 35-38
MSR_UNC_CBO_1_PERFEVTSEL2	
06_2AH .....	See Table 35-20
MSR_UNC_CBO_1_PERFEVTSEL3	
06_2AH .....	See Table 35-20
MSR_UNC_CBO_1_UNIT_STATUS	
06_2AH .....	See Table 35-20
MSR_UNC_CBO_2_PERFCTR0	
06_2AH .....	See Table 35-20
06_3CH, 06_45H, 06_46H .....	See Table 35-28
06_4EH, 06_5EH .....	See Table 35-38
MSR_UNC_CBO_2_PERFCTR1	
06_2AH .....	See Table 35-20



## MODEL-SPECIFIC REGISTERS (MSRS)

<b>MSR Name and CPUID DisplayFamily_DisplayModel</b>	<b>Location</b>
06_3CH, 06_45H, 06_46H .....	See Table 35-28
06_4EH, 06_5EH .....	See Table 35-38
<b>MSR_UNC_CBO_2_PERFCTR2</b>	
06_2AH .....	See Table 35-20
<b>MSR_UNC_CBO_2_PERFCTR3</b>	
06_2AH .....	See Table 35-20
<b>MSR_UNC_CBO_2_PERFEVTSELO</b>	
06_2AH .....	See Table 35-20
06_3CH, 06_45H, 06_46H .....	See Table 35-28
06_4EH, 06_5EH .....	See Table 35-38
<b>MSR_UNC_CBO_2_PERFEVTSEL1</b>	
06_2AH .....	See Table 35-20
06_3CH, 06_45H, 06_46H .....	See Table 35-28
06_4EH, 06_5EH .....	See Table 35-38
<b>MSR_UNC_CBO_2_PERFEVTSEL2</b>	
06_2AH .....	See Table 35-20
<b>MSR_UNC_CBO_2_PERFEVTSEL3</b>	
06_2AH .....	See Table 35-20
<b>MSR_UNC_CBO_2_UNIT_STATUS</b>	
06_2AH .....	See Table 35-20
<b>MSR_UNC_CBO_3_PERFCTR0</b>	
06_2AH .....	See Table 35-20
06_3CH, 06_45H, 06_46H .....	See Table 35-28
06_4EH, 06_5EH .....	See Table 35-38
<b>MSR_UNC_CBO_3_PERFCTR1</b>	
06_2AH .....	See Table 35-20
06_3CH, 06_45H, 06_46H .....	See Table 35-28
06_4EH, 06_5EH .....	See Table 35-38
<b>MSR_UNC_CBO_3_PERFCTR2</b>	
06_2AH .....	See Table 35-20
<b>MSR_UNC_CBO_3_PERFCTR3</b>	
06_2AH .....	See Table 35-20
<b>MSR_UNC_CBO_3_PERFEVTSELO</b>	
06_2AH .....	See Table 35-20
06_3CH, 06_45H, 06_46H .....	See Table 35-28
06_4EH, 06_5EH .....	See Table 35-38
<b>MSR_UNC_CBO_3_PERFEVTSEL1</b>	
06_2AH .....	See Table 35-20
06_3CH, 06_45H, 06_46H .....	See Table 35-28
06_4EH, 06_5EH .....	See Table 35-38
<b>MSR_UNC_CBO_3_PERFEVTSEL2</b>	
06_2AH .....	See Table 35-20

<b>MSR Name and CPUID DisplayFamily_DisplayModel</b>	<b>Location</b>
MSR_UNC_CBO_3_PERFEVTSEL3 06_2AH .....	See Table 35-20
MSR_UNC_CBO_3_UNIT_STATUS 06_2AH .....	See Table 35-20
MSR_UNC_CBO_4_PERFCTR0 06_2AH .....	See Table 35-20
MSR_UNC_CBO_4_PERFCTR1 06_2AH .....	See Table 35-20
MSR_UNC_CBO_4_PERFCTR2 06_2AH .....	See Table 35-20
MSR_UNC_CBO_4_PERFCTR3 06_2AH .....	See Table 35-20
MSR_UNC_CBO_4_PERFEVTSELO 06_2AH .....	See Table 35-20
MSR_UNC_CBO_4_PERFEVTSEL1 06_2AH .....	See Table 35-20
MSR_UNC_CBO_4_PERFEVTSEL2 06_2AH .....	See Table 35-20
MSR_UNC_CBO_4_PERFEVTSEL3 06_2AH .....	See Table 35-20
MSR_UNC_CBO_4_UNIT_STATUS 06_2AH .....	See Table 35-20
MSR_UNC_CBO_CONFIG 06_2AH .....	See Table 35-20
06_3CH, 06_45H, 06_46H .....	See Table 35-28
06_4EH, 06_5EH .....	See Table 35-38
MSR_UNC_PERF_FIXED_CTR 06_2AH .....	See Table 35-20
06_3CH, 06_45H, 06_46H .....	See Table 35-28
06_4EH, 06_5EH .....	See Table 35-38
MSR_UNC_PERF_FIXED_CTRL 06_2AH .....	See Table 35-20
06_3CH, 06_45H, 06_46H .....	See Table 35-28
06_4EH, 06_5EH .....	See Table 35-38
MSR_UNC_PERF_GLOBAL_CTRL 06_2AH .....	See Table 35-20
06_3CH, 06_45H, 06_46H .....	See Table 35-28
06_4EH, 06_5EH .....	See Table 35-38
MSR_UNC_PERF_GLOBAL_STATUS 06_2AH .....	See Table 35-20
06_3CH, 06_45H, 06_46H .....	See Table 35-28
06_4EH, 06_5EH .....	See Table 35-38

## MODEL-SPECIFIC REGISTERS (MSRS)

<b>MSR Name and CPUID DisplayFamily_DisplayModel</b>	<b>Location</b>
MSR_UNCORE_ADDR_OPCODE_MATCH 06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH .....	See Table 35-14
MSR_UNCORE_FIXED_CTR_CTRL 06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH .....	See Table 35-14
MSR_UNCORE_FIXED_CTR0 06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH .....	See Table 35-14
MSR_UNCORE_PERF_GLOBAL_CTRL 06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH .....	See Table 35-14
MSR_UNCORE_PERF_GLOBAL_OVF_CTRL 06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH .....	See Table 35-14
MSR_UNCORE_PERF_GLOBAL_STATUS 06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH .....	See Table 35-14
MSR_UNCORE_PERFEVTSELO 06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH .....	See Table 35-14
MSR_UNCORE_PERFEVTSEL1 06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH .....	See Table 35-14
MSR_UNCORE_PERFEVTSEL2 06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH .....	See Table 35-14
MSR_UNCORE_PERFEVTSEL3 06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH .....	See Table 35-14
MSR_UNCORE_PERFEVTSEL4 06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH .....	See Table 35-14
MSR_UNCORE_PERFEVTSEL5 06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH .....	See Table 35-14
MSR_UNCORE_PERFEVTSEL6 06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH .....	See Table 35-14
MSR_UNCORE_PERFEVTSEL7 06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH .....	See Table 35-14
MSR_UNCORE_PMC0 06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH .....	See Table 35-14
MSR_UNCORE_PMC1 06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH .....	See Table 35-14
MSR_UNCORE_PMC2 06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH .....	See Table 35-14
MSR_UNCORE_PMC3 06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH .....	See Table 35-14
MSR_UNCORE_PMC4 06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH .....	See Table 35-14
MSR_UNCORE_PMC5 06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH .....	See Table 35-14
06_2EH .....	See Table 35-15
MSR_UNCORE_PMC6	

<b>MSR Name and CPUID DisplayFamily_DisplayModel</b>	<b>Location</b>
06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH .....	See Table 35-14
MSR_UNCORE_PMC7	
06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH .....	See Table 35-14
MSR_UNCORE_PRMRR_BASE	
06_4EH, 06_5EH .....	See Table 35-37
MSR_UNCORE_PRMRR_MASK	
06_4EH, 06_5EH .....	See Table 35-37
MSR_W_PMON_BOX_CTRL	
06_2EH .....	See Table 35-15
MSR_W_PMON_BOX_OVF_CTRL	
06_2EH .....	See Table 35-15
MSR_W_PMON_BOX_STATUS	
06_2EH .....	See Table 35-15
MSR_W_PMON_CTRL0	
06_2EH .....	See Table 35-15
MSR_W_PMON_CTRL1	
06_2EH .....	See Table 35-15
MSR_W_PMON_CTRL2	
06_2EH .....	See Table 35-15
MSR_W_PMON_CTRL3	
06_2EH .....	See Table 35-15
MSR_W_PMON_EVNT_SELO	
06_2EH .....	See Table 35-15
MSR_W_PMON_EVNT_SEL1	
06_2EH .....	See Table 35-15
MSR_W_PMON_EVNT_SEL2	
06_2EH .....	See Table 35-15
MSR_W_PMON_EVNT_SEL3	
06_2EH .....	See Table 35-15
MSR_W_PMON_FIXED_CTR	
06_2EH .....	See Table 35-15
MSR_W_PMON_FIXED_CTR_CTL	
06_2EH .....	See Table 35-15
MSR_WEIGHTED_CORE_CO	
06_4EH, 06_5EH .....	See Table 35-37
MTRRfix16K_80000	
06_0EH .....	See Table 35-44
P6 Family .....	See Table 35-46
MTRRfix16K_A0000	
06_0EH .....	See Table 35-44
P6 Family .....	See Table 35-46
MTRRfix4K_C0000	

MODEL-SPECIFIC REGISTERS (MSRS)

<b>MSR Name and CPUID DisplayFamily_DisplayModel</b>	<b>Location</b>
06_0EH .....	See Table 35-44
P6 Family .....	See Table 35-46
<b>MTRRfix4K_C8000</b>	
06_0EH .....	See Table 35-44
P6 Family .....	See Table 35-46
<b>MTRRfix4K_D0000</b>	
06_0EH .....	See Table 35-44
P6 Family .....	See Table 35-46
<b>MTRRfix4K_D8000</b>	
06_0EH .....	See Table 35-44
P6 Family .....	See Table 35-46
<b>MTRRfix4K_E0000</b>	
06_0EH .....	See Table 35-44
P6 Family .....	See Table 35-46
<b>MTRRfix4K_E8000</b>	
06_0EH .....	See Table 35-44
P6 Family .....	See Table 35-46
<b>MTRRfix4K_F0000</b>	
06_0EH .....	See Table 35-44
P6 Family .....	See Table 35-46
<b>MTRRfix4K_F8000</b>	
06_0EH .....	See Table 35-44
P6 Family .....	See Table 35-46
<b>MTRRfix64K_00000</b>	
06_0EH .....	See Table 35-44
P6 Family .....	See Table 35-46
<b>MTRRphysBase0</b>	
06_0EH .....	See Table 35-44
P6 Family .....	See Table 35-46
<b>MTRRphysBase1</b>	
06_0EH .....	See Table 35-44
P6 Family .....	See Table 35-46
<b>MTRRphysBase2</b>	
06_0EH .....	See Table 35-44
P6 Family .....	See Table 35-46
<b>MTRRphysBase3</b>	
06_0EH .....	See Table 35-44
P6 Family .....	See Table 35-46
<b>MTRRphysBase4</b>	
06_0EH .....	See Table 35-44
P6 Family .....	See Table 35-46
<b>MTRRphysBase5</b>	

<b>MSR Name and CPUID DisplayFamily_DisplayModel</b>	<b>Location</b>
06_0EH .....	See Table 35-44
P6 Family .....	See Table 35-46
<b>MTRRphysBase6</b>	
06_0EH .....	See Table 35-44
P6 Family .....	See Table 35-46
<b>MTRRphysBase7</b>	
06_0EH .....	See Table 35-44
P6 Family .....	See Table 35-46
<b>MTRRphysMask0</b>	
06_0EH .....	See Table 35-44
P6 Family .....	See Table 35-46
<b>MTRRphysMask1</b>	
06_0EH .....	See Table 35-44
P6 Family .....	See Table 35-46
<b>MTRRphysMask2</b>	
06_0EH .....	See Table 35-44
P6 Family .....	See Table 35-46
<b>MTRRphysMask3</b>	
06_0EH .....	See Table 35-44
P6 Family .....	See Table 35-46
<b>MTRRphysMask4</b>	
06_0EH .....	See Table 35-44
P6 Family .....	See Table 35-46
<b>MTRRphysMask5</b>	
06_0EH .....	See Table 35-44
P6 Family .....	See Table 35-46
<b>MTRRphysMask6</b>	
06_0EH .....	See Table 35-44
P6 Family .....	See Table 35-46
<b>MTRRphysMask7</b>	
06_0EH .....	See Table 35-44
P6 Family .....	See Table 35-46
<b>ROB_CR_BKUPTMPDR6</b>	
06_0EH .....	See Table 35-44
P6 Family .....	See Table 35-46



## 20. Updates to Chapter 36, Volume 3C

Change bars show changes to Chapter 36 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C: System Programming Guide, Part 3*.

-----

Changes include Intel PTWRITE addition and updates.



# CHAPTER 36

## INTEL® PROCESSOR TRACE

---

### 36.1 OVERVIEW

Intel® Processor Trace (**Intel PT**) is an extension of Intel® Architecture that captures information about software execution using dedicated hardware facilities that cause only minimal performance perturbation to the software being traced. This information is collected in **data packets**. The initial implementations of Intel PT offer **control flow tracing**, which generates a variety of packets to be processed by a software decoder. The packets include timing, program flow information (e.g. branch targets, branch taken/not taken indications) and program-induced mode related information (e.g. Intel TSX state transitions, CR3 changes). These packets may be buffered internally before being sent to the memory subsystem or other output mechanism available in the platform. Debug software can process the trace data and reconstruct the program flow.

Later generations include additional trace sources, including software trace instrumentation using PTWRITE, and Power Event tracing.

#### 36.1.1 Features and Capabilities

Intel PT's control flow trace generates a variety of packets that, when combined with the binaries of a program by a post-processing tool, can be used to produce an exact execution trace. The packets record flow information such as instruction pointers (IP), indirect branch targets, and directions of conditional branches within contiguous code regions (basic blocks).

Intel PT can also be configured to log software-generated packets using PTWRITE, and packets describing processor power management events.

In addition, the packets record other contextual, timing, and bookkeeping information that enables both functional and performance debugging of applications. Intel PT has several control and filtering capabilities available to customize the tracing information collected and to append other processor state and timing information to enable debugging. For example, there are modes that allow packets to be filtered based on the current privilege level (CPL) or the value of CR3.

Configuration of the packet generation and filtering capabilities are programmed via a set of MSRs. The MSRs generally follow the naming convention of IA32\_RTIT\_\*. The capability provided by these configuration MSRs are enumerated by CPUID, see Section 36.3. Details of the MSRs for configuring Intel PT are described in Section 36.2.7.

##### 36.1.1.1 Packet Summary

After a tracing tool has enabled and configured the appropriate MSRs, the processor will collect and generate trace information in the following categories of packets (for more details on the packets, see Section 36.4):

- Packets about basic information on program execution: These include:
  - Packet Stream Boundary (**PSB**) packets: PSB packets act as 'heartbeats' that are generated at regular intervals (e.g., every 4K trace packet bytes). These packets allow the packet decoder to find the packet boundaries within the output data stream; a PSB packet should be the first packet that a decoder looks for when beginning to decode a trace.
  - Paging Information Packet (**PIP**): PIPs record modifications made to the CR3 register. This information, along with information from the operating system on the CR3 value of each process, allows the debugger to attribute linear addresses to their correct application source.
  - Time-Stamp Counter (**TSC**) packets: TSC packets aid in tracking wall-clock time, and contain some portion of the software-visible time-stamp counter.
  - Core Bus Ratio (**CBR**) packets: CBR packets contain the core:bus clock ratio.

- Overflow (**OVF**) packets: OVF packets are sent when the processor experiences an internal buffer overflow, resulting in packets being dropped. This packet notifies the decoder of the loss and can help the decoder to respond to this situation.
- Packets about control flow information:
  - Taken Not-Taken (**TNT**) packets: TNT packets track the “direction” of direct conditional branches (taken or not taken).
  - Target IP (**TIP**) packets: TIP packets record the target IP of indirect branches, exceptions, interrupts, and other branches or events. These packets can contain the IP, although that IP value may be compressed by eliminating upper bytes that match the last IP. There are various types of TIP packets; they are covered in more detail in Section 36.4.2.2.
  - Flow Update Packets (**FUP**): FUPs provide the source IP addresses for asynchronous events (interrupt and exceptions), as well as other cases where the source address cannot be determined from the binary.
  - **MODE** packets: These packets provide the decoder with important processor execution information so that it can properly interpret the dis-assembled binary and trace log. MODE packets have a variety of formats that indicate details such as the execution mode (16-bit, 32-bit, or 64-bit).
- Packets inserted by software:
  - PTWRITE (PTW) packets: includes the value of the operand passed to the PTWRITE instruction (see “PTWRITE - Write Data to a Processor Trace Packet” in *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2B*).
- Packets about processor power management events:
  - MWAIT packets: Indicate successful completion of an MWAIT operation to a C-state deeper than C0.0.
  - Power State Entry (PWRE) packets: Indicate entry to a C-state deeper than C0.0.
  - Power State Exit (PWRX) packets: Indicate exit from a C-state deeper than C0.0, returning to C0.
  - Execution Stopped (EXSTOP) packets: Indicate that software execution has stopped, due to events such as P-state change, C-state change, or thermal throttling.

## 36.2 INTEL® PROCESSOR TRACE OPERATIONAL MODEL

This section describes the overall Intel Processor Trace mechanism and the essential concepts relevant to how it operates.

### 36.2.1 Change of Flow Instruction (COFI) Tracing

A basic program block is a section of code where no jumps or branches occur. The instruction pointers (IPs) in this block of code need not be traced, as the processor will execute them from start to end without redirecting code flow. Instructions such as branches, and events such as exceptions or interrupts, can change the program flow. These instructions and events that change program flow are called Change of Flow Instructions (COFI). There are three categories of COFI:

- Direct transfer COFI.
- Indirect transfer COFI.
- Far transfer COFI.

The following subsections describe the COFI events that result in trace packet generation. Table 36-1 lists branch instruction by COFI types. For detailed description of specific instructions, see *Intel® 64 and IA-32 Architectures Software Developer’s Manual*.

**Table 36-1. COFI Type for Branch Instructions**

COFI Type	Instructions
Conditional Branch	JA, JAE, JB, JBE, JC, JCXZ< JECXZ, JRCXZ, JE, JG, JGE, JL, JLE, JNA, JNAE, JNB, JNBE, JNC, JNE, JNG, JNGE, JNL, JNLE, JNO, JNP, JNS, JNZ, JO, JP, JPE, JPO, JS, JZ, LOOP, LOOPE, LOOPNE, LOOPNZ, LOOPZ

**Table 36-1. COFI Type for Branch Instructions**

COFI Type	Instructions
Unconditional Direct Branch	JMP (E9 xx, EB xx), CALL (E8 xx)
Indirect Branch	JMP (FF /4), CALL (FF /2)
Near Ret	RET (C3, C2 xx)
Far Transfers	INT3, INTn, INTO, IRET, IRETD, IRETQ, JMP (EA xx, FF /5), CALL (9A xx, FF /3), RET (CB, CA xx), SYS-CALL, SYSRET, SYSENTER, SYSEXIT, VMLAUNCH, VMRESUME

### 36.2.1.1 Direct Transfer COFI

Direct Transfer COFI are relative branches. This means that their target is an IP whose offset from the current IP is embedded in the instruction bytes. It is not necessary to indicate target of these instructions in the trace output since it can be obtained through the source disassembly. Conditional branches need to indicate only whether the branch is taken or not. Unconditional branches do not need any recording in the trace output. There are two sub-categories:

- **Conditional Branch (Jcc, J\*CXZ) and LOOP**

To track this type of instruction, the processor encodes a single bit (taken or not taken — TNT) to indicate the program flow after the instruction.

Jcc, J\*CXZ, and LOOP can be traced with TNT bits. To improve the trace packet output efficiency, the processor will compact several TNT bits into a single packet.

- **Unconditional Direct Jumps**

There is no trace output required for direct unconditional jumps (like JMP near relative or CALL near relative) since they can be directly inferred from the application assembly. Direct unconditional jumps do not generate a TNT bit or a Target IP packet, though TIP.PGD and TIP.PGE packets can be generated by unconditional direct jumps that toggle Intel PT enables (see Section 36.2.5).

### 36.2.1.2 Indirect Transfer COFI

Indirect transfer instructions involve updating the IP from a register or memory location. Since the register or memory contents can vary at any time during execution, there is no way to know the target of the indirect transfer until the register or memory contents are read. As a result, the disassembled code is not sufficient to determine the target of this type of COFI. Therefore, tracing hardware must send out the destination IP in the trace packet for debug software to determine the target address of the COFI. Note that this IP may be a linear or effective address (see Section 36.3.1.1).

An indirect transfer instruction generates a Target IP Packet (TIP) that contains the target address of the branch. There are two sub-categories:

- **Near JMP Indirect and Near Call Indirect**

As previously mentioned, the target of an indirect COFI resides in the contents of either a register or memory location. Therefore, the processor must generate a packet that includes this target address to allow the decoder to determine the program flow.

- **Near RET**

When a CALL instruction executes, it pushes onto the stack the address of the next instruction following the CALL. Upon completion of the call procedure, the RET instruction is often used to pop the return address off of the call stack and redirect code flow back to the instruction following the CALL.

A RET instruction simply transfers program flow to the address it popped off the stack. Because a called procedure may change the return address on the stack before executing the RET instruction, debug software can be misled if it assumes that code flow will return to the instruction following the last CALL. Therefore, even for near RET, a Target IP Packet may be sent.

- **RET Compression**

A special case is applied if the target of the RET is consistent with what would be expected from tracking the CALL stack. If it is assured that the decoder has seen the corresponding CALL (with “corresponding” defined

as the CALL with matching stack depth), and the RET target is the instruction after that CALL, the RET target may be “compressed”. In this case, only a single TNT bit of “taken” is generated instead of a Target IP Packet. To ensure that the decoder will not be confused in cases of RET compression, only RETs that correspond to CALLs which have been seen since the last PSB packet may be compressed in a given logical processor. For details, see “Indirect Transfer Compression for Returns (RET)” in Section 36.4.2.2.

### 36.2.1.3 Far Transfer COFI

All operations that change the instruction pointer and are not near jumps are “far transfers”. This includes exceptions, interrupts, traps, TSX aborts, and instructions that do far transfers.

All far transfers will produce a Target IP (TIP) packet, which provides the destination IP address. For those far transfers that cannot be inferred from the binary source (e.g., asynchronous events such as exceptions and interrupts), the TIP will be preceded by a Flow Update packet (FUP), which provides the source IP address at which the event was taken. Table 36-23 indicates exactly which IP will be included in the FUP generated by a far transfer.

## 36.2.2 Software Trace Instrumentation with PTWRITE

PTWRITE provides a mechanism by which software can instrument the Intel PT trace. PTWRITE is a ring3-accessible instruction that can be passed a register or memory variable, see “PTWRITE - Write Data to a Processor Trace Packet” in *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2B* for details. The contents of that variable will be used as the payload for the PTW packet (see Table 36-40 “PTW Packet Definition”), inserted at the time of PTWRITE retirement, assuming PTWRITE is enabled and all other filtering conditions are met. Decode and analysis software will then be able to determine the meaning of the PTWRITE packet based on the IP of the associated PTWRITE instruction.

PTWRITE is enabled via IA32\_RTIT\_CTL.PTWEn[12] (see Table 36-6). Optionally, the user can use IA32\_RTIT\_CTL.FUPonPTW[5] to enable PTW packets to be followed by FUP packets containing the IP of the associated PTWRITE instruction.

## 36.2.3 Power Event Tracing

Power Event Trace is a capability that exposes core- and thread-level sleep state and power down transition information. When this capability is enabled, the trace will expose information about:

- Scenarios where software execution stops.
  - Due to sleep state entry, frequency change, or other powerdown.
  - Includes the IP, when in the tracing context.
- The requested and resolved hardware thread C-state.
  - Including indication of hardware autonomous C-state entry.
- The last and deepest core C-state achieved during a sleep session.
- The reason for C-state wake.

This information is in addition to the bus ratio (CBR) information provided by default after any powerdown, and the timing information (TSC, TMA, MTC, CYC) provided during or after a powerdown state.

Power Event Trace is enabled via IA32\_RTIT\_CTL.PwrEvtEn[4].

## 36.2.4 Trace Filtering

Intel Processor Trace provides filtering capabilities, by which the debug/profile tool can control what code is traced.

### 36.2.4.1 Filtering by Current Privilege Level (CPL)

Intel PT provides the ability to configure a logical processor to generate trace packets only when CPL = 0, when CPL > 0, or regardless of CPL.

CPL filtering ensures that no IPs or other architectural state information associated with the filtered CPL can be seen in the log. For example, if the processor is configured to trace only when  $CPL > 0$ , and software executes SYSCALL (changing the CPL to 0), the destination IP of the SYSCALL will be suppressed from the generated packet (see the discussion of TIP.PGD in Section 36.4.2.5).

It should be noted that CPL is always 0 in real-address mode and that CPL is always 3 in virtual-8086 mode. To trace code in these modes, filtering should be configured accordingly.

When software is executing in a non-enabled CPL, ContextEn is cleared. See Section 36.2.5.1 for details.

### 36.2.4.2 Filtering by CR3

Intel PT supports a CR3-filtering mechanism by which the generation of packets containing architectural states can be enabled or disabled based on the value of CR3. A debugger can use CR3 filtering to trace only a single application without context switching the state of the RTIT MSR. For the reconstruction of traces from software with multiple threads, debug software may wish to context-switch for the state of the RTIT MSRs (if the operating system does not provide context-switch support) to separate the output for the different threads (see Section 36.3.5, "Context Switch Consideration").

To trace for only a single CR3 value, software can write that value to the IA32\_RTIT\_CR3\_MATCH MSR, and set IA32\_RTIT\_CTL.CR3Filter. When CR3 value does not match IA32\_RTIT\_CR3\_MATCH and IA32\_RTIT\_CTL.CR3Filter is 1, ContextEn is forced to 0, and packets containing architectural states will not be generated. Some other packets can be generated when ContextEn is 0; see Section 36.2.5.3 for details. When CR3 does match IA32\_RTIT\_CR3\_MATCH (or when IA32\_RTIT\_CTL.CR3Filter is 0), CR3 filtering does not force ContextEn to 0 (although it could be 0 due to other filters or modes).

CR3 matches IA32\_RTIT\_CR3\_MATCH if the two registers are identical for bits 63:12, or 63:5 when in PAE paging mode; the lower 5 bits of CR3 and IA32\_RTIT\_CR3\_MATCH are ignored. CR3 filtering is independent of the value of CR0.PG.

When CR3 filtering is in use, PIP packets may still be seen in the log if the processor is configured to trace when  $CPL = 0$  (IA32\_RTIT\_CTL.OS = 1). If not, no PIP packets will be seen.

### 36.2.4.3 Filtering by IP

Trace packet generation with configurable filtering by IP is supported if  $CPUID.(EAX=14H, ECX=0):EBX[\text{bit } 2] = 1$ . Intel PT can be configured to enable the generation of packets containing architectural states only when the processor is executing code within certain IP ranges. If the IP is outside of these ranges, generation of some packets is blocked.

IP filtering is enabled using the ADDRn\_CFG fields in the IA32\_RTIT\_CTL MSR (Section 36.2.7.2), where the digit 'n' is a zero-based number that selects which address range is being configured. Each ADDRn\_CFG field configures the use of the register pair IA32\_RTIT\_ADDRn\_A and IA32\_RTIT\_ADDRn\_B (Section 36.2.7.5).

IA32\_RTIT\_ADDRn\_A defines the base and IA32\_RTIT\_ADDRn\_B specifies the limit of the range in which tracing is enabled. Thus each range, referred to as the ADDRn range, is defined by [IA32\_RTIT\_ADDRn\_A.

IA32\_RTIT\_ADDRn\_B]. There can be multiple such ranges, software can query CPUID (Section 36.3.1) for the number of ranges supported on a processor.

Default behavior (ADDRn\_CFG=0) defines no IP filter range, meaning FilterEn is always set. In this case code at any IP can be traced, though other filters, such as CR3 or CPL, could limit tracing. When ADDRn\_CFG is set to enable IP filtering (see Section 36.3.1), tracing will commence when a taken branch or event is seen whose target address is in the ADDRn range.

While inside a tracing region and with FilterEn is set, leaving the tracing region may only be detected once a taken branch or event with a target outside the range is retired. If an ADDRn range is entered or exited by executing the next sequential instruction, rather than by a control flow transfer, FilterEn may not toggle immediately. See Section 36.2.5.5 for more details on FilterEn.

Note that these address range base and limit values are inclusive, such that the range includes the first and last instruction whose first instruction byte is in the ADDRn range.

Depending upon processor implementation, IP filtering may be based on linear or effective address. This can cause different behavior between implementations if CSbase is not equal to zero or in real mode. See Section 36.3.1.1 for details. Software can query CPUID to determine filters are based on linear or effective address (Section 36.3.1).

Note that some packets, such as MTC (Section 36.3.7) and other timing packets, do not depend on FilterEn. For details on which packets depend on FilterEn, and hence are impacted by IP filtering, see Section 36.4.1.

### TraceStop

The ADDRn ranges can also be configured to cause tracing to be disabled upon entry to the specified region. This is intended for cases where unexpected code is executed, and the user wishes to immediately stop generating packets in order to avoid overwriting previously written packets.

The TraceStop mechanism works much the same way that IP filtering does, and uses the same address comparison logic. The TraceStop region base and limit values are programmed into one or more ADDRn ranges, but IA32\_RTIT\_CTL.ADDRn\_CFG is configured with the TraceStop encoding. Like FilterEn, TraceStop is detected when a taken branch or event lands in a TraceStop region.

Further, TraceStop requires that TriggerEn=1 at the beginning of the branch/event, and ContextEn=1 upon completion of the branch/event. When this happens, the CPU will set IA32\_RTIT\_STATUS.Stopped, thereby clearing TriggerEn and hence disabling packet generation. This may generate a TIP.PGD packet with the target IP of the branch or event that entered the TraceStop region. Finally, a TraceStop packet will be inserted, to indicate that the condition was hit.

If a TraceStop condition is encountered during buffer overflow (Section 36.3.8), it will not be dropped, but will instead be signaled once the overflow has resolved.

Note that a TraceStop event does not guarantee that all internally buffered packets are flushed out of internal buffers. To ensure that this has occurred, the user should clear TraceEn.

To resume tracing after a TraceStop event, the user must first disable Intel PT by clearing IA32\_RTIT\_CTL.TraceEn before the IA32\_RTIT\_STATUS.Stopped bit can be cleared. At this point Intel PT can be reconfigured, and tracing resumed.

Note that the IA32\_RTIT\_STATUS.Stopped bit can also be set using the ToPA STOP bit. See Section 36.2.6.2.

### IP Filtering Example

The following table gives an example of IP filtering behavior. Assume that IA32\_RTIT\_ADDRn\_A = the IP of RangeBase, and that IA32\_RTIT\_ADDRn\_B = the IP of RangeLimit, while IA32\_RTIT\_CTL.ADDRn\_CFG = 0x1 (enable ADDRn range as a FilterEn range).

**Table 36-2. IP Filtering Packet Example**

Code Flow	Packets
<pre> Bar:     jmp RangeBase // jump into filter range RangeBase:     jcc Foo // not taken     add eax, 1 Foo:     jmp RangeLimit+1 // jump out of filter range RangeLimit:     nop     jcc Bar                     </pre>	<pre> TIP.PGE(RangeBase) TNT(0) TIP.PGD(RangeLimit+1)                     </pre>

### IP Filtering and TraceStop

It is possible for the user to configure IP filter range(s) and TraceStop range(s) that overlap. In this case, code executing in the non-overlapping portion of either range will behave as would be expected from that range. Code executing in the overlapping range will get TraceStop behavior.

## 36.2.5 Packet Generation Enable Controls

Intel Processor Trace includes a variety of controls that determine whether a packet is generated. In general, most packets are sent only if Packet Enable (**PacketEn**) is set. PacketEn is an internal state maintained in hardware in



response to software configurable enable controls, PacketEn is not visible to software directly. The relationship of PacketEn to the software-visible controls in the configuration MSRs is described in this section.

### 36.2.5.1 Packet Enable (PacketEn)

When PacketEn is set, the processor is in the mode that Intel PT is monitoring and all packets can be generated to log what is being executed. PacketEn is composed of other states according to this relationship:

$$\text{PacketEn} \leftarrow \text{TriggerEn} \text{ AND } \text{ContextEn} \text{ AND } \text{FilterEn} \text{ AND } \text{BranchEn}$$

These constituent controls are detailed in the following subsections.

PacketEn ultimately determines when the processor is tracing. When PacketEn is set, all control flow packets are enabled. When PacketEn is clear, no control flow packets are generated, though other packets (timing and book-keeping packets) may still be sent. See Section 36.2.6 for details of PacketEn and packet generation.

Note that, on processors that do not support IP filtering (i.e., CPUID.(EAX=14H, ECX=0):EBX.IPFILT\_WRSTPRSV[bit 2] = 0), FilterEn is treated as always set.

### 36.2.5.2 Trigger Enable (TriggerEn)

Trigger Enable (**TriggerEn**) is the primary indicator that trace packet generation is active. TriggerEn is set when IA32\_RTIT\_CTL.TraceEn is set, and cleared by any of the following conditions:

- TraceEn is cleared by software.
- A TraceStop condition is encountered and IA32\_RTIT\_STATUS.Stopped is set.
- IA32\_RTIT\_STATUS.Error is set due to an operational error (see Section 36.3.9).

Software can discover the current TriggerEn value by reading the IA32\_RTIT\_STATUS.TriggerEn bit. When TriggerEn is clear, tracing is inactive and no packets are generated.

### 36.2.5.3 Context Enable (ContextEn)

Context Enable (**ContextEn**) indicates whether the processor is in the state or mode that software configured hardware to trace. For example, if execution with CPL = 0 code is not being traced (IA32\_RTIT\_CTL.OS = 0), then ContextEn will be 0 when the processor is in CPL0.

Software can discover the current ContextEn value by reading the IA32\_RTIT\_STATUS.ContextEn bit. ContextEn is defined as follows:

$$\begin{aligned} \text{ContextEn} = & !((\text{IA32\_RTIT\_CTL.OS} = 0 \text{ AND } \text{CPL} = 0) \text{ OR} \\ & (\text{IA32\_RTIT\_CTL.USER} = 0 \text{ AND } \text{CPL} > 0) \text{ OR } (\text{IS\_IN\_A\_PRODUCTION\_ENCLAVE}^1) \text{ OR} \\ & (\text{IA32\_RTIT\_CTL.CR3Filter} = 1 \text{ AND } \text{IA32\_RTIT\_CR3\_MATCH} \text{ does not match CR3})) \end{aligned}$$

If the clearing of ContextEn causes PacketEn to be cleared, a Packet Generation Disable (TIP.PGD) packet is generated, but its IP payload is suppressed. If the setting of ContextEn causes PacketEn to be set, a Packet Generation Enable (TIP.PGE) packet is generated.

When ContextEn is 0, control flow packets (TNT, FUP, TIP.\*, MODE.\*) are not generated, and no Linear Instruction Pointers (LIPs) are exposed. However, some packets, such as MTC and PSB (see Section 36.4.2.16 and Section 36.4.2.17), may still be generated while ContextEn is 0. For details of which packets are generated only when ContextEn is set, see Section 36.4.1.

The processor does not update ContextEn when TriggerEn = 0.

The value of ContextEn will toggle only when TriggerEn = 1.

### 36.2.5.4 Branch Enable (BranchEn)

This value is based purely on the IA32\_RTIT\_CTL.BranchEn value. If **BranchEn** is not set, then relevant COFI packets (TNT, TIP\*, FUP, MODE.\*) are suppressed. Other packets related to timing (TSC, TMA, MTC, CYC), as well

1. Trace packets generation is disabled in a production enclave, see Section 36.2.8.3. See *Intel® Software Guard Extensions Programming Reference* about differences between a production enclave and a debug enclave.

as PSB, will be generated normally regardless. Further, PIP and VMCS continue to be generated, as indicators of what software is running.

### 36.2.5.5 Filter Enable (FilterEn)

Filter Enable indicates that the Instruction Pointer (IP) is within the range of IPs that Intel PT is configured to watch. Software can get the state of Filter Enable by a RDMSR of IA32\_RTIT\_STATUS.FilterEn. For details on configuration and use of IP filtering, see Section 36.2.4.3.

On clearing of FilterEn that also clears PacketEn, a Packet Generation Disable (TIP.PGD) will be generated, but unlike the ContextEn case, the IP payload may not be suppressed. For direct, unconditional branches, as well as for indirect branches (including RETs), the PGD generated by leaving the tracing region and clearing FilterEn will contain the target IP. This means that IPs from outside the configured range can be exposed in the trace, as long as they are within context.

When FilterEn is 0, control flow packets are not generated (e.g., TNT, TIP). However, some packets, such as PIP, MTC, and PSB, may still be generated while FilterEn is clear. For details on packet enable dependencies, see Section 36.4.1.

After TraceEn is set, FilterEn is set to 1 at all times if there is no IP filter range configured by software (IA32\_RTIT\_CTL.ADDRn\_CFG != 1, for all n), or if the processor does not support IP filtering (i.e., CPUID.(EAX=14H, ECX=0):EBX.IPFILT\_WRSTPRSV[bit 2] = 0). FilterEn will toggle only when TraceEn=1 and ContextEn=1, and when at least one range is configured for IP filtering.

## 36.2.6 Trace Output

Intel PT output should be viewed independently from trace content and filtering mechanisms. The options available for trace output can vary across processor generations and platforms.

Trace output is written out using one of the following output schemes, as configured by the ToPA and FabricEn bit fields of IA32\_RTIT\_CTL (see Section 36.2.7.2):

- A single, contiguous region of physical address space.
- A collection of variable-sized regions of physical memory. These regions are linked together by tables of pointers to those regions, referred to as Table of Physical Addresses (**ToPA**). The trace output stores bypass the caches and the TLBs, but are not serializing. This is intended to minimize the performance impact of the output.
- A platform-specific trace transport subsystem.

Regardless of the output scheme chosen, Intel PT stores bypass the processor caches by default. This ensures that they don't consume precious cache space, but they do not have the serializing aspects associated with un-cacheable (UC) stores. Software should avoid using MTRRs to mark any portion of the Intel PT output region as UC, as this may override the behavior described above and force Intel PT stores to UC, thereby incurring severe performance impact.

There is no guarantee that a packet will be written to memory or other trace endpoint after some fixed number of cycles after a packet-producing instruction executes. The only way to assure that all packets generated have reached their endpoint is to clear TraceEn and follow that with a store, fence, or serializing instruction; doing so ensures that all buffered packets are flushed out of the processor.

### 36.2.6.1 Single Range Output

When IA32\_RTIT\_CTL.ToPA and IA32\_RTIT\_CTL.FabricEn bits are clear, trace packet output is sent to a single, contiguous memory (or MMIO if DRAM is not available) range defined by a base address in IA32\_RTIT\_OUTPUT\_BASE (Section 36.2.7.7) and mask value in IA32\_RTIT\_OUTPUT\_MASK\_PTRS (Section 36.2.7.8). The current write pointer in this range is also stored in IA32\_RTIT\_OUTPUT\_MASK\_PTRS. This output range is circular, meaning that when the writes wrap around the end of the buffer they begin again at the base address.

This output method is best suited for cases where Intel PT output is either:

- Configured to be directed to a sufficiently large contiguous region of DRAM.



- Configured to go to an MMIO debug port, in order to route Intel PT output to a platform-specific trace endpoint (e.g., JTAG). In this scenario, a specific range of addresses is written in a circular manner, and SoC will intercept these writes and direct them to the proper device. Repeated writes to the same address do not overwrite each other, but are accumulated by the debugger, and hence no data is lost by the circular nature of the buffer.

The processor will determine the address to which to write the next trace packet output byte as follows:

```
OutputBase[63:0] ← IA32_RTIT_OUTPUT_BASE[63:0]
OutputMask[63:0] ← ZeroExtend64(IA32_RTIT_OUTPUT_MASK_PTRS[31:0])
OutputOffset[63:0] ← ZeroExtend64(IA32_RTIT_OUTPUT_MASK_PTRS[63:32])
trace_store_phys_addr ← (OutputBase & ~OutputMask) + (OutputOffset & OutputMask)
```

### Single-Range Output Errors

If the output base and mask are not properly configured by software, an operational error (see Section 36.3.9) will be signaled, and tracing disabled. Error scenarios with single-range output are:

- Mask value is non-contiguous.  
IA32\_RTIT\_OUTPUT\_MASK\_PTRS.MaskOrTablePointer value has a 0 in a less significant bit position than the most significant bit containing a 1.
- Base address and Mask are mis-aligned, and have overlapping bits set.  
IA32\_RTIT\_OUTPUT\_BASE && IA32\_RTIT\_OUTPUT\_MASK\_PTRS.MaskOrTableOffset > 0.
- Illegal Output Offset  
IA32\_RTIT\_OUTPUT\_MASK\_PTRS.OutputOffset is greater than the mask value (IA32\_RTIT\_OUTPUT\_MASK\_PTRS.MaskOrTableOffset).

Also note that errors can be signaled due to trace packet output overlapping with restricted memory, see Section 36.2.6.4.

#### 36.2.6.2 Table of Physical Addresses (ToPA)

When IA32\_RTIT\_CTL.ToPA is set and IA32\_RTIT\_CTL.FabricEn is clear, the ToPA output mechanism is utilized. The ToPA mechanism uses a linked list of tables; see Figure 36-1 for an illustrative example. Each entry in the table contains some attribute bits, a pointer to an output region, and the size of the region. The last entry in the table may hold a pointer to the next table. This pointer can either point to the top of the current table (for circular array) or to the base of another table. The table size is not fixed, since the link to the next table can exist at any entry.

The processor treats the various output regions referenced by the ToPA table(s) as a unified buffer. This means that a single packet may span the boundary between one output region and the next.

The ToPA mechanism is controlled by three values maintained by the processor:

- proc\_trace\_table\_base.**  
This is the physical address of the base of the current ToPA table. When tracing is enabled, the processor loads this value from the IA32\_RTIT\_OUTPUT\_BASE MSR. While tracing is enabled, the processor updates the IA32\_RTIT\_OUTPUT\_BASE MSR with changes to proc\_trace\_table\_base, but these updates may not be synchronous to software execution. When tracing is disabled, the processor ensures that the MSR contains the latest value of proc\_trace\_table\_base.
- proc\_trace\_table\_offset.**  
This indicates the entry of the current table that is currently in use. (This entry contains the address of the current output region.) When tracing is enabled, the processor loads this value from bits 31:7 (MaskOrTableOffset) of the IA32\_RTIT\_OUTPUT\_MASK\_PTRS. While tracing is enabled, the processor updates IA32\_RTIT\_OUTPUT\_MASK\_PTRS.MaskOrTableOffset with changes to proc\_trace\_table\_offset, but these updates may not be synchronous to software execution. When tracing is disabled, the processor ensures that the MSR contains the latest value of proc\_trace\_table\_offset.
- proc\_trace\_output\_offset.**  
This is a pointer into the current output region and indicates the location of the next write. When tracing is enabled, the processor loads this value from bits 63:32 (OutputOffset) of the

IA32\_RTIT\_OUTPUT\_MASK\_PTRS. While tracing is enabled, the processor updates IA32\_RTIT\_OUTPUT\_MASK\_PTRS.OutputOffset with changes to proc\_trace\_output\_offset, but these updates may not be synchronous to software execution. When tracing is disabled, the processor ensures that the MSR contains the latest value of proc\_trace\_output\_offset.

Figure 36-1 provides an illustration (not to scale) of the table and associated pointers.

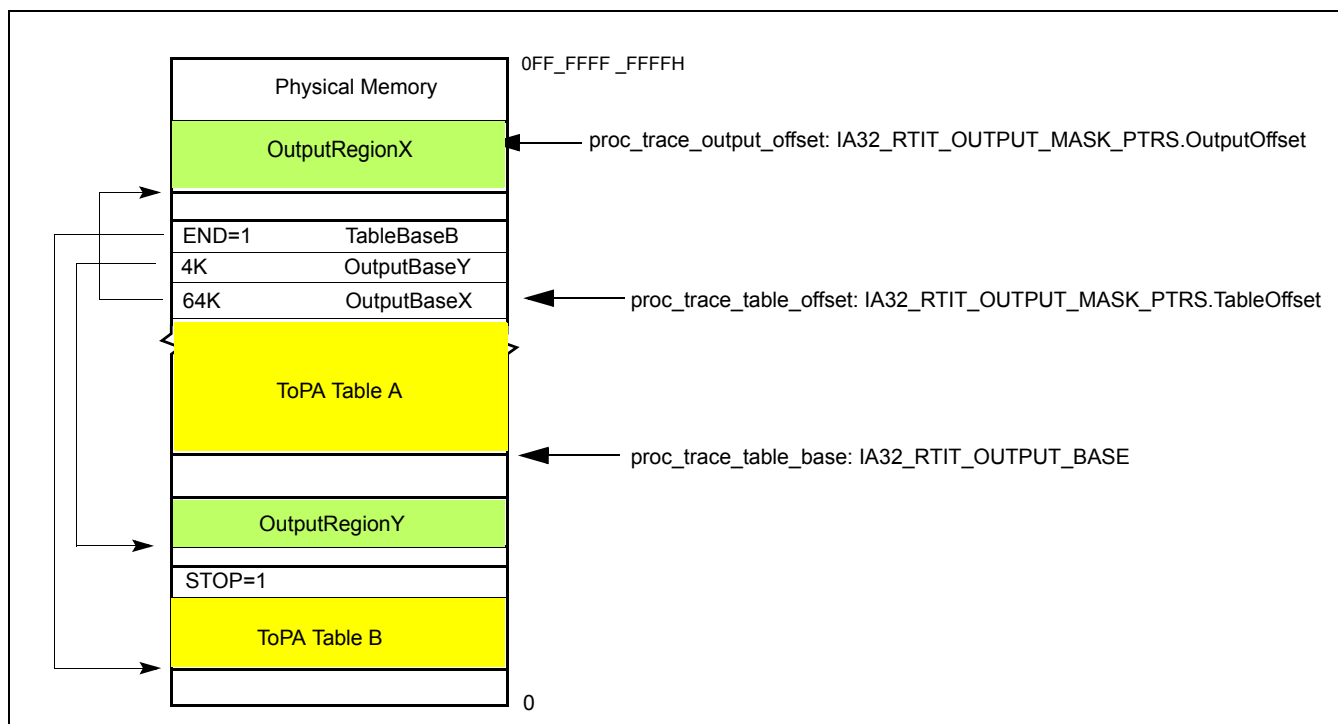


Figure 36-1. ToPA Memory Illustration

With the ToPA mechanism, the processor writes packets to the current output region (identified by `proc_trace_table_base` and the `proc_trace_table_offset`). The offset within that region to which the next byte will be written is identified by `proc_trace_output_offset`. When that region is filled with packet output (thus `proc_trace_output_offset = RegionSize-1`), `proc_trace_table_offset` is moved to the next ToPA entry, `proc_trace_output_offset` is set to 0, and packet writes begin filling the new output region specified by `proc_trace_table_offset`.

As packets are written out, each store derives its physical address as follows:

$$\text{trace\_store\_phys\_addr} \leftarrow \text{Base address from current ToPA table entry} + \text{proc\_trace\_output\_offset}$$

Eventually, the regions represented by all entries in the table may become full, and the final entry of the table is reached. An entry can be identified as the final entry because it has either the END or STOP attribute. The END attribute indicates that the address in the entry does not point to another output region, but rather to another ToPA table. The STOP attribute indicates that tracing will be disabled once the corresponding region is filled. See Section 36.2.6.2 for details on STOP.

When an END entry is reached, the processor loads `proc_trace_table_base` with the base address held in this END entry, thereby moving the current table pointer to this new table. The `proc_trace_table_offset` is reset to 0, as is the `proc_trace_output_offset`, and packet writes will resume at the base address indicated in the first entry.

If the table has no STOP or END entry, and trace-packet generation remains enabled, eventually the maximum table size will be reached (`proc_trace_table_offset = FFFFFFFFH`). In this case, the `proc_trace_table_offset` and `proc_trace_output_offset` are reset to 0 (wrapping back to the beginning of the current table) once the last output region is filled.

It is important to note that processor updates to the IA32\_RTIT\_OUTPUT\_BASE and IA32\_RTIT\_OUTPUT\_MASK\_PTRS MSRs are asynchronous to instruction execution. Thus, reads of these MSRs while Intel PT is enabled may return stale values. Like all IA32\_RTIT\_\* MSRs, the values of these MSRs should not be trusted or saved unless trace packet generation is first disabled by clearing IA32\_RTIT\_CTL.TraceEn. This ensures that the output MSR values account for all packets generated to that point, after which the output MSR values will be frozen until tracing resumes. <sup>1</sup>

The processor may cache internally any number of entries from the current table or from tables that it references (directly or indirectly). If tracing is enabled, the processor may ignore or delay detection of modifications to these tables. To ensure that table changes are detected by the processor in a predictable manner, software should clear TraceEn before modifying the current table (or tables that it references) and only then re-enable packet generation.

### Single Output Region ToPA Implementation

The first processor generation to implement Intel PT supports only ToPA configurations with a single ToPA entry followed by an END entry that points back to the first entry (creating one circular output buffer). Such processors enumerate CPUID.(EAX=14H,ECX=0):ECX.MENTRY[bit 1] = 0 and CPUID.(EAX=14H,ECX=0):ECX.TOPAOUT[bit 0] = 1.

If CPUID.(EAX=14H,ECX=0):ECX.MENTRY[bit 1] = 0, ToPA tables can hold only one output entry, which must be followed by an END=1 entry which points back to the base of the table. Hence only one contiguous block can be used as output.

The lone output entry can have INT or STOP set, but nonetheless must be followed by an END entry as described above. Note that, if INT=1, the PMI will actually be delivered before the region is filled.

### ToPA Table Entry Format

The format of ToPA table entries is shown in Figure 36-2. The size of the address field is determined by the processor’s physical-address width (MAXPHYADDR) in bits, as reported in CPUID.80000008H:EAX[7:0].

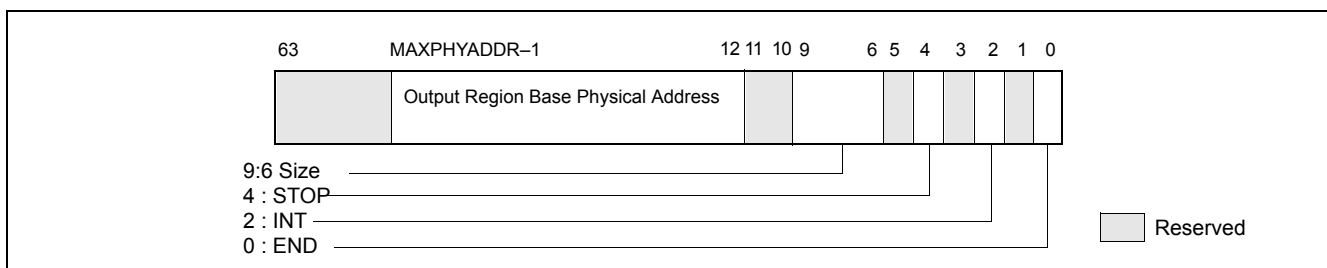


Figure 36-2. Layout of ToPA Table Entry

Table 36-3 describes the details of the ToPA table entry fields. If reserved bits are set to 1, an error is signaled.

Table 36-3. ToPA Table Entry Fields

ToPA Entry Field	Description
Output Region Base Physical Address	If END=0, this is the base physical address of the output region specified by this entry. Note that all regions must be aligned based on their size. Thus a 2M region must have bits 20:12 clear. If the region is not properly aligned, an operational error will be signaled when the entry is reached. If END=1, this is the 4K-aligned base physical address of the next ToPA table (which may be the base of the current table, or the first table in the linked list if a circular buffer is desired). If the processor supports only a single ToPA output region (see above), this address must be the value currently in the IA32_RTIT_OUTPUT_BASE MSR.

1. Although WRMSR is a serializing instruction, the execution of WRMSR that forces packet writes by clearing TraceEn does not itself cause these writes to be globally observed.

**Table 36-3. ToPA Table Entry Fields (Contd.)**

ToPA Entry Field	Description
Size	Indicates the size of the associated output region. Encodings are: 0: 4K, 1: 8K, 2: 16K, 3: 32K, 4: 64K, 5: 128K, 6: 256K, 7: 512K, 8: 1M, 9: 2M, 10: 4M, 11: 8M, 12: 16M, 13: 32M, 14: 64M, 15: 128M This field is ignored if END=1.
STOP	When the output region indicated by this entry is filled, software should disable packet generation. This will be accomplished by setting IA32_RTIT_STATUS.Stopped, which clears TriggerEn. This bit must be 0 if END=1; otherwise it is treated as reserved bit violation (see ToPA Errors).
INT	When the output region indicated by this entry is filled, signal Perfmon LVT interrupt. Note that if both INT and STOP are set in the same entry, the STOP will happen before the INT. Thus the interrupt handler should expect that the IA32_RTIT_STATUS.Stopped bit will be set, and will need to be reset before tracing can be resumed. This bit must be 0 if END=1; otherwise it is treated as reserved bit violation (see ToPA Errors).
END	If set, indicates that this is an END entry, and thus the address field points to a table base rather than an output region base. If END=1, INT and STOP must be set to 0; otherwise it is treated as reserved bit violation (see ToPA Errors). The Size field is ignored in this case. If the processor supports only a single ToPA output region (see above), END must be set in the second table entry.

### ToPA STOP

Each ToPA entry has a STOP bit. If this bit is set, the processor will set the IA32\_RTIT\_STATUS.Stopped bit when the corresponding trace output region is filled. This will clear TriggerEn and thereby cease packet generation. See Section 36.2.7.4 for details on IA32\_RTIT\_STATUS.Stopped. This sequence is known as “ToPA Stop”.

No TIP.PGD packet will be seen in the output when the ToPA stop occurs, since the disable happens only when the region is already full. When this occurs, output ceases after the last byte of the region is filled, which may mean that a packet is cut off in the middle. Any packets remaining in internal buffers are lost and cannot be recovered.

When ToPA stop occurs, the IA32\_RTIT\_OUTPUT\_BASE MSR will hold the base address of the table whose entry had STOP=1. IA32\_RTIT\_OUTPUT\_MASK\_PTRS.MaskOffsetTableOffset will hold the index value for that entry, and the IA32\_RTIT\_OUTPUT\_MASK\_PTRS.OutputOffset should be set to the size of the region.

Note that this means the offset pointer is pointing to the next byte after the end of the region, a configuration that would produce an operational error if the configuration remained when tracing is re-enabled with IA32\_RTIT\_STATUS.Stopped cleared.

### ToPA PMI

Each ToPA entry has an INT bit. If this bit is set, the processor will signal a performance-monitoring interrupt (PMI) when the corresponding trace output region is filled. This interrupt is not precise, and it is thus likely that writes to the next region will occur by the time the interrupt is taken.

The following steps should be taken to configure this interrupt:

1. Enable PMI via the LVT Performance Monitor register (at MMIO offset 340H in xAPIC mode; via MSR 834H in x2APIC mode). See *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B* for more details on this register. For ToPA PMI, set all fields to 0, save for the interrupt vector, which can be selected by software.
2. Set up an interrupt handler to service the interrupt vector that a ToPA PMI can raise.
3. Set the interrupt flag by executing STI.
4. Set the INT bit in the ToPA entry of interest and enable packet generation, using the ToPA output option. Thus, TraceEn=ToPA=1 in the IA32\_RTIT\_CTL MSR.

Once the INT region has been filled with packet output data, the interrupt will be signaled. This PMI can be distinguished from others by checking bit 55 (Trace\_ToPA\_PMI) of the IA32\_PERF\_GLOBAL\_STATUS MSR (MSR 38EH). Once the ToPA PMI handler has serviced the relevant buffer, writing 1 to bit 55 of the MSR at 390H (IA32\_GLOBAL\_STATUS\_RESET) clears IA32\_PERF\_GLOBAL\_STATUS.Trace\_ToPA\_PMI.

Intel PT is not frozen on PMI, and thus the interrupt handler will be traced (though filtering can prevent this). The Freeze\_Perfmon\_on\_PMI and Freeze\_LBRs\_on\_PMI settings in IA32\_DEBUGCTL will be applied on ToPA PMI just as on other PMIs, and hence Perfmon counters are frozen.

Assuming the PMI handler wishes to read any buffered packets for persistent output, or wishes to modify any Intel PT MSRs, software should first disable packet generation by clearing TraceEn. This ensures that all buffered packets are written to memory and avoids tracing of the PMI handler. The configuration MSRs can then be used to determine where tracing has stopped. If packet generation is disabled by the handler, it should then be manually re-enabled before the IRET if continued tracing is desired.

In rare cases, it may be possible to trigger a second ToPA PMI before the first is handled. This can happen if another ToPA region with INT=1 is filled before, or shortly after, the first PMI is taken, perhaps due to EFLAGS.IF being cleared for an extended period of time. This can manifest in two ways: either the second PMI is triggered before the first is taken, and hence only one PMI is taken, or the second is triggered after the first is taken, and thus will be taken when the handler for the first completes. Software can minimize the likelihood of the second case by clearing TraceEn at the beginning of the PMI handler. Further, it can detect such cases by then checking the Interrupt Request Register (IRR) for PMI pending, and checking the ToPA table base and off-set pointers (in IA32\_RTIT\_OUTPUT\_BASE and IA32\_RTIT\_OUTPUT\_MASK\_PTRS) to see if multiple entries with INT=1 have been filled.

### ToPA PMI and Single Output Region ToPA Implementation

A processor that supports only a single ToPA output region implementation (such that only one output region is supported; see above) will attempt to signal a ToPA PMI interrupt before the output wraps and overwrites the top of the buffer. To support this functionality, the PMI handler should disable packet generation as soon as possible.

Due to PMI skid, it is possible that, in rare cases, the wrap will have occurred before the PMI is delivered. Software can avoid this by setting the STOP bit in the ToPA entry (see Table 36-3); this will disable tracing once the region is filled, and no wrap will occur. This approach has the downside of disabling packet generation so that some of the instructions that led up to the PMI will not be traced. If the PMI skid is significant enough to cause the region to fill and tracing to be disabled, the PMI handler will need to clear the IA32\_RTIT\_STATUS.Stopped indication before tracing can resume.

### ToPA PMI and XSAVES/XRSTORS State Handling

In some cases the ToPA PMI may be taken after completion of an XSAVES instruction that switches Intel PT state, and in such cases any modification of Intel PT MSRs within the PMI handler will not persist when the saved Intel PT context is later restored with XRSTORS. To account for such a scenario, it is recommended that the Intel PT output configuration be modified by altering the ToPA tables themselves, rather than the Intel PT output MSRs.

Table 36-4 depicts a recommended PMI handler algorithm for managing multi-region ToPA output and handling ToPA PMIs that may arrive between XSAVES and XRSTORS. This algorithm is flexible to allow software to choose between adding entries to the current ToPA table, adding a new ToPA table, or using the current ToPA table as a circular buffer. It assumes that the ToPA entry that triggers the PMI is not the last entry in the table, which is the recommended treatment.

**Table 36-4. Algorithm to Manage Intel PT ToPA PMI and XSAVES/XRSTORS**

Pseudo Code Flow
<pre> IF (IA32_PERF_GLOBAL_STATUS.ToPA)   Save IA32_RTIT_CTL value;   IF ( IA32_RTIT_CTL.TraceEN )     Disable Intel PT by clearing TraceEn;   FI;   IF ( there is space available to grow the current ToPA table )     Add one or more ToPA entries after the last entry in the ToPA table;     Point new ToPA entry address field(s) to new output region base(s);   ELSE     Modify an upcoming ToPA entry in the current table to have END=1;     IF (output should transition to a new ToPA table )       Point the address of the "END=1" entry of the current table to the new table base;     ELSE       /* Continue to use the current ToPA table, make a circular. */       Point the address of the "END=1" entry to the base of the current table;       Modify the ToPA entry address fields for filled output regions to point to new, unused output regions;       /* Filled regions are those with index in the range of 0 to (IA32_RTIT_MASK_PTRS.MaskOrTableOffset -1). */     FI;   FI;   Restore saved IA32_RTIT_CTL.value; FI; </pre>

### ToPA Errors

When a malformed ToPA entry is found, an **operation error** results (see Section 36.3.9). A malformed entry can be any of the following:

1. **ToPA entry reserved bit violation.**  
This describes cases where a bit marked as reserved in Section 36.2.6.2 above is set to 1.
2. **ToPA alignment violation.**  
This includes cases where illegal ToPA entry base address bits are set to 1:
  - a. ToPA table base address is not 4KB-aligned. The table base can be from a WRMSR to IA32\_RTIT\_OUTPUT\_BASE, or from a ToPA entry with END=1.
  - b. ToPA entry base address is not aligned to the ToPA entry size (e.g., a 2MB region with base address[20:12] not equal to 0).
  - c. ToPA entry base address sets upper physical address bits not supported by the processor.
3. **Illegal ToPA Output Offset** (if IA32\_RTIT\_STATUS.Stopped=0).  
IA32\_RTIT\_OUTPUT\_MASK\_PTRS.OutputOffset is greater than or equal to the size of the current ToPA output region size.
4. **ToPA rules violations.**  
These are similar to ToPA entry reserved bit violations; they are cases when a ToPA entry is encountered with illegal field combinations. They include the following:
  - a. Setting the STOP or INT bit on an entry with END=1.
  - b. Setting the END bit in entry 0 of a ToPA table.
  - c. On processors that support only a single ToPA entry (see above), two additional illegal settings apply:
    - i) ToPA table entry 1 with END=0.
    - ii) ToPA table entry 1 with base address not matching the table base.

In all cases, the error will be logged by setting `IA32_RTIT_STATUS.Error`, thereby disabling tracing when the problematic ToPA entry is reached (when `proc_trace_table_offset` points to the entry containing the error). Any packet bytes that are internally buffered when the error is detected may be lost.

Note that operational errors may also be signaled due to attempts to access restricted memory. See Section 36.2.6.4 for details.

A tracing software have a range of flexibility using ToPA to manage the interaction of Intel PT with application buffers, see Section 36.5.

### 36.2.6.3 Trace Transport Subsystem

When `IA32_RTIT_CTL.FabricEn` is set, the `IA32_RTIT_CTL.ToPA` bit is ignored, and trace output is written to the trace transport subsystem. The endpoints of this transport are platform-specific, and details of configuration options should refer to the specific platform documentation. The `FabricEn` bit is available to be set if `CPUID(EAX=14H,ECX=0):EBX[bit 3] = 1`.

### 36.2.6.4 Restricted Memory Access

Packet output cannot be directed to any regions of memory that are restricted by the platform. In particular, all memory accesses on behalf of packet output are checked against the SMRR regions. If there is any overlap with these regions, trace data collection will not function properly. Exact processor behavior is implementation-dependent; Table 36-5 summarizes several scenarios.

**Table 36-5. Behavior on Restricted Memory Access**

Scenario	Description
ToPA output region overlaps with SMRR	Stores to the restricted memory region will be dropped, and that packet data will be lost. Any attempt to read from that restricted region will return all 1s. The processor also may signal an error (Section 36.3.9) and disable tracing when the output pointer reaches the restricted region. If packet generation remains enabled, then packet output may continue once stores are no longer directed to restricted memory (on wrap, or if the output region is larger than the restricted memory region).
ToPA table overlaps with SMRR	The processor will signal an error (Section 36.3.9) and disable tracing when the ToPA read pointer ( <code>IA32_RTIT_OUTPUT_BASE + (proc_trace_table_offset &lt;&lt; 3)</code> ) enters the restricted region.

It should also be noted that packet output should not be routed to the 4KB APIC MMIO region, as defined by the `IA32_APIC_BASE` MSR. For details about the APIC, refer to *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*. No error is signaled for this case.

### Modifications to Restricted Memory Regions

It is recommended that software disable packet generation before modifying the SMRRs to change the scope of the SMRR regions. This is because the processor reserves the right to cache any number of ToPA table entries internally, after checking them against restricted memory ranges. Once cached, the entries will not be checked again, meaning one could potentially route packet output to a newly restricted region. Software can ensure that any cached entries are written to memory by clearing `IA32_RTIT_CTL.TraceEn`.

## 36.2.7 Enabling and Configuration MSRs

### 36.2.7.1 General Considerations

Trace packet generation is enabled and configured by a collection of model-specific registers (MSRs), which are detailed below. Some notes on the configuration MSR behavior:

- If Intel Processor Trace is not supported by the processor (see Section 36.3.1), RDMSR or WRMSR of the `IA32_RTIT_*` MSRs will cause `#GP`.
- A WRMSR to any of these configuration MSRs that begins and ends with `IA32_RTIT_CTL.TraceEn` set will `#GP` fault. Packet generation must be disabled before the configuration MSRs can be changed.



Note: Software may write the same value back to IA32\_RTIT\_CTL without #GP, even if TraceEn=1.

- All configuration MSRs for Intel PT are duplicated per logical processor
- For each configuration MSR, any MSR write that attempts to change bits marked reserved, or utilize encodings marked reserved, will cause a #GP fault.
- All configuration MSRs for Intel PT are cleared on a cold RESET.
  - If CPUID.(EAX=14H, ECX=0):EBX.IPFILT\_WRSTPRSV[bit 2] = 1, only the TraceEn bit is cleared on warm RESET; though this may have the impact of clearing other bits in IA32\_RTIT\_STATUS. Other MSR values of the trace configuration MSRs are preserved on warm RESET.
- The semantics of MSR writes to trace configuration MSRs in this chapter generally apply to explicit WRMSR to these registers, using VM-exit or VM-entry MSR load list to these MSRs, XRSTORS with requested feature bit map including XSAVE map component of state\_8 (corresponding to IA32\_XSS[bit 8]), and the write to IA32\_RTIT\_CTL.TraceEn by XSAVES (Section 36.3.5.2).

### 36.2.7.2 IA32\_RTIT\_CTL MSR

IA32\_RTIT\_CTL, at address 570H, is the primary enable and control MSR for trace packet generation. Bit positions are listed in Table 36-6.

Table 36-6. IA32\_RTIT\_CTL MSR

Position	Bit Name	At Reset	Bit Description
0	TraceEn	0	If 1, enables tracing; else tracing is disabled if 0. When this bit transitions from 1 to 0, all buffered packets are flushed out of internal buffers. A further store, fence, or architecturally serializing instruction may be required to ensure that packet data can be observed at the trace endpoint. See Section 36.2.7.3 for details of enabling and disabling packet generation. Note that the processor will clear this bit on #SMI (Section ) and warm reset. Other MSR bits of IA32_RTIT_CTL (and other trace configuration MSRs) are not impacted by these events.
1	CYCEn	0	0: Disables CYC Packet (see Section 36.4.2.14). 1: Enables CYC Packet. This bit is reserved if CPUID.(EAX=14H, ECX=0):EBX.CPSB_CAM[bit 1] = 0.
2	OS	0	0: Packet generation is disabled when CPL = 0. 1: Packet generation may be enabled when CPL = 0.
3	User	0	0: Packet generation is disabled when CPL > 0. 1: Packet generation may be enabled when CPL > 0.
4	PwrEvtEn	0	0: Power Event Trace packets are disabled. 1: Power Event Trace packets are enabled (see Section 36.2.3, “Power Event Tracing”).
5	FUPonPTW	0	0: PTW packets are not followed by FUPs. 1: PTW packets are followed by FUPs.
6	FabricEn	0	0: Trace output is directed to the memory subsystem, mechanism depends on IA32_RTIT_CTL.ToPA. 1: Trace output is directed to the trace transport subsystem, IA32_RTIT_CTL.ToPA is ignored. This bit is reserved if CPUID.(EAX=14H, ECX=0):ECX[bit 3] = 0.
7	CR3Filter	0	0: Disables CR3 filtering. 1: Enables CR3 filtering.



Table 36-6. IA32\_RTIT\_CTL MSR (Contd.)

Position	Bit Name	At Reset	Bit Description
8	ToPA	0	0: Single-range output scheme enabled if CPUID.(EAX=14H, ECX=0):ECX.SNGLRGNOUT[bit 2] = 1 and IA32_RTIT_CTL.FabricEn=0. 1: ToPA output scheme enabled (see Section 36.2.6.2) if CPUID.(EAX=14H, ECX=0):ECX.TOPA[bit 0] = 1, and IA32_RTIT_CTL.FabricEn=0. Note: WRMSR to IA32_RTIT_CTL that sets TraceEn but clears this bit and FabricEn would cause #GP, if CPUID.(EAX=14H, ECX=0):ECX.SNGLRGNOUT[bit 2] = 0. WRMSR to IA32_RTIT_CTL that sets this bit causes #GP, if CPUID.(EAX=14H, ECX=0):ECX.TOPA[bit 0] = 0.
9	MTCEn	0	0: Disables MTC Packet (see Section 36.4.2.16). 1: Enables MTC Packet. This bit is reserved if CPUID.(EAX=14H, ECX=0):EBX.MTC[bit 3] = 0.
10	TSCEn	0	0: Disable TSC packets. 1: Enable TSC packets (see Section 36.4.2.11).
11	DisRETC	0	0: Enable RET compression. 1: Disable RET compression (see Section 36.2.1.2).
12	PTWEn	0	0: PTWRITE packet generation disabled. 1: PTWRITE packet generation enabled (see Table 36-40 "PTW Packet Definition").
13	BranchEn	0	0: Disable COFI-based packets. 1: Enable COFI-based packets: FUP, TIP, TIP.PGE, TIP.PGD, TNT, MODE.Exec, MODE.TSX. see Section 36.2.6 for details on BranchEn.
17:14	MTCFreq	0	Defines MTC packet Frequency, which is based on the core crystal clock, or Always Running Timer (ART). MTC will be sent each time the selected ART bit toggles. The following Encodings are defined: 0: ART(0), 1: ART(1), 2: ART(2), 3: ART(3), 4: ART(4), 5: ART(5), 6: ART(6), 7: ART(7), 8: ART(8), 9: ART(9), 10: ART(10), 11: ART(11), 12: ART(12), 13: ART(13), 14: ART(14), 15: ART(15) Software must use CPUID to query the supported encodings in the processor, see Section 36.3.1. Use of unsupported encodings will result in a #GP fault. This field is reserved if CPUID.(EAX=14H, ECX=0):EBX.MTC[bit 3] = 0.
18	Reserved	0	Must be 0.
22:19	CycThresh	0	CYC packet threshold, see Section 36.3.6 for details. CYC packets will be sent with the first eligible packet after N cycles have passed since the last CYC packet. If CycThresh is 0 then N=0, otherwise N is defined as $2^{(\text{CycThresh}-1)}$ . The following Encodings are defined: 0: 0, 1: 1, 2: 2, 3: 4, 4: 8, 5: 16, 6: 32, 7: 64, 8: 128, 9: 256, 10: 512, 11: 1024, 12: 2048, 13: 4096, 14: 8192, 15: 16384 Software must use CPUID to query the supported encodings in the processor, see Section 36.3.1. Use of unsupported encodings will result in a #GP fault. This field is reserved if CPUID.(EAX=14H, ECX=0):EBX.CPSB_CAM[bit 1] = 0.
23	Reserved	0	Must be 0.

Table 36-6. IA32\_RTIT\_CTL MSR (Contd.)

Position	Bit Name	At Reset	Bit Description
27:24	PSBFreq	0	Indicates the frequency of PSB packets. PSB packet frequency is based on the number of Intel PT packet bytes output, so this field allows the user to determine the increment of IA32_RTIT_STATUS.PacketByteCnt that should cause a PSB to be generated. Note that PSB insertion is not precise, but the average output bytes per PSB should approximate the SW selected period. The following Encodings are defined: 0: 2K, 1: 4K, 2: 8K, 3: 16K, 4: 32K, 5: 64K, 6: 128K, 7: 256K, 8: 512K, 9: 1M, 10: 2M, 11: 4M, 12: 8M, 13: 16M, 14: 32M, 15: 64M Software must use CPUID to query the supported encodings in the processor, see Section 36.3.1. Use of unsupported encodings will result in a #GP fault. This field is reserved if CPUID.(EAX=14H, ECX=0):EBX.CPSB_CAM[bit 1] = 0.
31:28	Reserved	0	Must be 0.
35:32	ADDR0_CFG	0	Configures the base/limit register pair IA32_RTIT_ADDR0_A/B based on the following encodings: 0: ADDR0 range unused. 1: The [IA32_RTIT_ADDR0_A..IA32_RTIT_ADDR0_B] range defines a FilterEn range. FilterEn will only be set when the IP is within this range, though other FilterEn ranges can additionally be used. See Section 36.2.4.3 for details on IP filtering. 2: The [IA32_RTIT_ADDR0_A..IA32_RTIT_ADDR0_B] range defines a TraceStop range. TraceStop will be asserted if code branches into this range. See 4.2.8 for details on TraceStop. 3..15: Reserved (#GP). This field is reserved if CPUID.(EAX=14H, ECX=1):EBX.RANGECONT[2:0] >= 0.
39:36	ADDR1_CFG	0	Configures the base/limit register pair IA32_RTIT_ADDR1_A/B based on the following encodings: 0: ADDR1 range unused. 1: The [IA32_RTIT_ADDR1_A..IA32_RTIT_ADDR1_B] range defines a FilterEn range. FilterEn will only be set when the IP is within this range, though other FilterEn ranges can additionally be used. See Section 36.2.4.3 for details on IP filtering. 2: The [IA32_RTIT_ADDR1_A..IA32_RTIT_ADDR1_B] range defines a TraceStop range. TraceStop will be asserted if code branches into this range. See Section 36.4.2.10 for details on TraceStop. 3..15: Reserved (#GP). This field is reserved if CPUID.(EAX=14H, ECX=1):EBX.RANGECONT[2:0] < 2.
43:40	ADDR2_CFG	0	Configures the base/limit register pair IA32_RTIT_ADDR2_A/B based on the following encodings: 0: ADDR2 range unused. 1: The [IA32_RTIT_ADDR2_A..IA32_RTIT_ADDR2_B] range defines a FilterEn range. FilterEn will only be set when the IP is within this range, though other FilterEn ranges can additionally be used. See Section 36.2.4.3 for details on IP filtering. 2: The [IA32_RTIT_ADDR2_A..IA32_RTIT_ADDR2_B] range defines a TraceStop range. TraceStop will be asserted if code branches into this range. See Section 36.4.2.10 for details on TraceStop. 3..15: Reserved (#GP). This field is reserved if CPUID.(EAX=14H, ECX=1):EBX.RANGECONT[2:0] < 3.

Table 36-6. IA32\_RTIT\_CTL MSR (Contd.)

Position	Bit Name	At Reset	Bit Description
47:44	ADDR3_CFG	0	Configures the base/limit register pair IA32_RTIT_ADDR3_A/B based on the following encodings: 0: ADDR3 range unused. 1: The [IA32_RTIT_ADDR3_A..IA32_RTIT_ADDR3_B] range defines a FilterEn range. FilterEn will only be set when the IP is within this range, though other FilterEn ranges can additionally be used. See Section 36.2.4.3 for details on IP filtering. 2: The [IA32_RTIT_ADDR3_A..IA32_RTIT_ADDR3_B] range defines a TraceStop range. TraceStop will be asserted if code branches into this range. See Section 36.4.2.10 for details on TraceStop. 3..15: Reserved (#GP). This field is reserved if CPUID.(EAX=14H, ECX=1):EBX.RANGECNT[2:0] < 4.
59:48	Reserved	0	Reserved only for future trace content enables, or address filtering configuration enables. Must be 0.
63:60	Reserved	0	Must be 0.

### 36.2.7.3 Enabling and Disabling Packet Generation with TraceEn

When TraceEn transitions from 0 to 1, Intel Processor Trace is enabled, and a series of packets may be generated. These packets help ensure that the decoder is aware of the state of the processor when the trace begins, and that it can keep track of any timing or state changes that may have occurred while packet generation was disabled. A full PSB+ (see Section 36.4.2.17) will be generated if IA32\_RTIT\_STATUS.PacketByteCnt=0, and may be generated in other cases as well. Otherwise, timing packets will be generated, including TSC, TMA, and CBR (see Section 36.4.2).

In addition to the packets discussed above, if and when PacketEn (Section 36.2.5.1) transitions from 0 to 1 (which may happen immediately, depending on filtering settings), a TIP.PGE packet (Section 36.4.2.3) will be generated.

When TraceEn is set, the processor may read ToPA entries from memory and cache them internally. For this reason, software should disable packet generation before making modifications to the ToPA tables (or changing the configuration of restricted memory regions). See Section 36.7 for more details of packets that may be generated with modifications to TraceEn.

#### Disabling Packet Generation

Clearing TraceEn causes any packet data buffered within the logical processor to be flushed out, after which the output MSRs (IA32\_RTIT\_OUTPUT\_BASE and IA32\_RTIT\_OUTPUT\_MASK\_PTRS) will have stable values. When output is directed to memory, a store, fence, or architecturally serializing instruction may be required to ensure that the packet data is globally observed. No special packets are generated by disabling packet generation, though a TIP.PGD may result if PacketEn=1 at the time of disable.

#### Other Writes to IA32\_RTIT\_CTL

Any attempt to modify IA32\_RTIT\_CTL while TraceEn is set will result in a general-protection fault (#GP) unless the same write also clears TraceEn. However, writes to IA32\_RTIT\_CTL that do not modify any bits will not cause a #GP, even if TraceEn remains set.

### 36.2.7.4 IA32\_RTIT\_STATUS MSR

The IA32\_RTIT\_STATUS MSR is readable and writable by software, but some bits (ContextEn, TriggerEn) are read-only and cannot be directly modified. The WRMSR instruction ignores these bits in the source operand (attempts to modify these bits are ignored and do not cause WRMSR to fault).

This MSR can only be written when IA32\_RTIT\_CTL.TraceEn is 0; otherwise WRMSR causes a general-protection fault (#GP). The processor does not modify the value of this MSR while TraceEn is 0 (software can modify it with WRMSR).

Table 36-7. IA32\_RTIT\_STATUS MSR

Position	Bit Name	At Reset	Bit Description
0	FilterEn	0	This bit is written by the processor, and indicates that tracing is allowed for the current IP, see Section 36.2.5.5. Writes are ignored.
1	ContextEn	0	The processor sets this bit to indicate that tracing is allowed for the current context. See Section 36.2.5.3. Writes are ignored.
2	TriggerEn	0	The processor sets this bit to indicate that tracing is enabled. See Section 36.2.5.2. Writes are ignored.
3	Reserved	0	Must be 0.
4	Error	0	The processor sets this bit to indicate that an operational error has been encountered. When this bit is set, TriggerEn is cleared to 0 and packet generation is disabled. For details, see “ToPA Errors” in Section 36.2.6.2.  When TraceEn is cleared, software can write this bit. Once it is set, only software can clear it. It is not recommended that software ever set this bit, except in cases where it is restoring a prior saved state.
5	Stopped	0	The processor sets this bit to indicate that a ToPA Stop condition has been encountered. When this bit is set, TriggerEn is cleared to 0 and packet generation is disabled. For details, see “ToPA STOP” in Section 36.2.6.2.  When TraceEn is cleared, software can write this bit. Once it is set, only software can clear it. It is not recommended that software ever set this bit, except in cases where it is restoring a prior saved state.
31:6	Reserved	0	Must be 0.
48:32	PacketByteCnt	0	This field is written by the processor, and holds a count of packet bytes that have been sent out. The processor also uses this field to determine when the next PSB packet should be inserted. Note that the processor may clear or modify this field at any time while IA32_RTIT_CTL.TraceEn=1. It will have a stable value when IA32_RTIT_CTL.TraceEn=0. See Section 36.4.2.17 for details.
63:49	Reserved	0	Must be 0.

### 36.2.7.5 IA32\_RTIT\_ADDRn\_A and IA32\_RTIT\_ADDRn\_B MSRs

The role of the IA32\_RTIT\_ADDRn\_A/B register pairs, for each n, is determined by the corresponding ADDRn\_CFG fields in IA32\_RTIT\_CTL (see Section 36.2.7.2). The number of these register pairs is enumerated by CPUID.(EAX=14H, ECX=1):EAX.RANGE CNT[2:0].

- Processors that enumerate support for 1 range support:

IA32\_RTIT\_ADDR0\_A, IA32\_RTIT\_ADDR0\_B

- Processors that enumerate support for 2 ranges support:  
IA32\_RTIT\_ADDR0\_A, IA32\_RTIT\_ADDR0\_B  
IA32\_RTIT\_ADDR1\_A, IA32\_RTIT\_ADDR1\_B
- Processors that enumerate support for 3 ranges support:  
IA32\_RTIT\_ADDR0\_A, IA32\_RTIT\_ADDR0\_B  
IA32\_RTIT\_ADDR1\_A, IA32\_RTIT\_ADDR1\_B  
IA32\_RTIT\_ADDR2\_A, IA32\_RTIT\_ADDR2\_B
- Processors that enumerate support for 4 ranges support:  
IA32\_RTIT\_ADDR0\_A, IA32\_RTIT\_ADDR0\_B  
IA32\_RTIT\_ADDR1\_A, IA32\_RTIT\_ADDR1\_B  
IA32\_RTIT\_ADDR2\_A, IA32\_RTIT\_ADDR2\_B  
IA32\_RTIT\_ADDR3\_A, IA32\_RTIT\_ADDR3\_B

Each register has a single 64-bit field that holds a linear address value. Writes must ensure that the address is properly sign-extended, otherwise a #GP fault will result.

### 36.2.7.6 IA32\_RTIT\_CR3\_MATCH MSR

The IA32\_RTIT\_CR3\_MATCH register is compared against CR3 when IA32\_RTIT\_CTL.CR3Filter is 1. Bits 63:5 hold the CR3 address value to match, bits 4:0 are reserved to 0. For more details on CR3 filtering and the treatment of this register, see Section 36.2.4.2.

This MSR can be written only when IA32\_RTIT\_CTL.TraceEn is 0; otherwise WRMSR causes a general-protection fault (#GP). IA32\_RTIT\_CR3\_MATCH[4:0] are reserved and must be 0; an attempt to set those bits using WRMSR causes a #GP.

### 36.2.7.7 IA32\_RTIT\_OUTPUT\_BASE MSR

This MSR is used to configure the trace output destination, when output is directed to memory (IA32\_RTIT\_CTL.FabricEn = 0). The size of the address field is determined by the maximum physical address width (MAXPHYADDR), as reported by CPUID.80000008H:EAX[7:0].

When the ToPA output scheme is used, the processor may update this MSR when packet generation is enabled, and those updates are asynchronous to instruction execution. Therefore, the values in this MSR should be considered unreliable unless packet generation is disabled (IA32\_RTIT\_CTL.TraceEn = 0).

Accesses to this MSR are supported only if Intel PT output to memory is supported, hence when either CPUID.(EAX=14H, ECX=0):ECX[bit 0] or CPUID.(EAX=14H, ECX=0):ECX[bit 2] are set. Otherwise WRMSR or RDMSR cause a general-protection fault (#GP). If supported, this MSR can be written only when IA32\_RTIT\_CTL.TraceEn is 0; otherwise WRMSR causes a general-protection fault (#GP).

**Table 36-8. IA32\_RTIT\_OUTPUT\_BASE MSR**

Position	Bit Name	At Reset	Bit Description
6:0	Reserved	0	Must be 0.
MAXPHYADDR-1:7	BasePhysAddr	0	<p>The base physical address. How this address is used depends on the value of IA32_RTIT_CTL.ToPA:</p> <p>0: This is the base physical address of a single, contiguous physical output region. This could be mapped to DRAM or to MMIO, depending on the value.</p> <p>The base address should be aligned with the size of the region, such that none of the 1s in the mask value(Section 36.2.7.8) overlap with 1s in the base address. If the base is not aligned, an operational error will result (see Section 36.3.9).</p> <p>1: The base physical address of the current ToPA table. The address must be 4K aligned. Writing an address in which bits 11:7 are non-zero will not cause a #GP, but an operational error will be signaled once TraceEn is set. See “ToPA Errors” in Section 36.2.6.2 as well as Section 36.3.9.</p>
63:MAXPHYADDR	Reserved	0	Must be 0.

**36.2.7.8 IA32\_RTIT\_OUTPUT\_MASK\_PTRS MSR**

This MSR holds any mask or pointer values needed to indicate where the next byte of trace output should be written. The meaning of the values held in this MSR depend on whether the ToPA output mechanism is in use. See Section 36.2.6.2 for details.

The processor updates this MSR while when packet generation is enabled, and those updates are asynchronous to instruction execution. Therefore, the values in this MSR should be considered unreliable unless packet generation is disabled (IA32\_RTIT\_CTL.TraceEn = 0).

Accesses to this MSR are supported only if Intel PT output to memory is supported, hence when either CPUID.(EAX=14H, ECX=0):ECX[bit 0] or CPUID.(EAX=14H, ECX=0):ECX[bit 2] are set. Otherwise WRMSR or RDMSR cause a general-protection fault (#GP). If supported, this MSR can be written only when IA32\_RTIT\_CTL.TraceEn is 0; otherwise WRMSR causes a general-protection fault (#GP).

**Table 36-9. IA32\_RTIT\_OUTPUT\_MASK\_PTRS MSR**

Position	Bit Name	At Reset	Bit Description
6:0	LowerMask	7FH	Forced to 1, writes are ignored.
31:7	MaskOrTableOffset	0	<p>The use of this field depends on the value of IA32_RTIT_CTL.ToPA:</p> <p>0: This field holds bits 31:7 of the mask value for the single, contiguous physical output region. The size of this field indicates that regions can be of size 128B up to 4GB. This value (combined with the lower 7 bits, which are reserved to 1) will be ANDed with the OutputOffset field to determine the next write address. All 1s in this field should be consecutive and starting at bit 7, otherwise the region will not be contiguous, and an operational error (Section 36.3.9) will be signaled when TraceEn is set.</p> <p>1: This field holds bits 27:3 of the offset pointer into the current ToPA table. This value can be added to the IA32_RTIT_OUTPUT_BASE value to produce a pointer to the current ToPA table entry, which itself is a pointer to the current output region. In this scenario, the lower 7 reserved bits are ignored. This field supports tables up to 256 MBytes in size.</p>

Table 36-9. IA32\_RTIT\_OUTPUT\_MASK\_PTRS MSR (Contd.)

Position	Bit Name	At Reset	Bit Description
63:32	OutputOffset	0	<p>The use of this field depends on the value of IA32_RTIT_CTL.ToPA:</p> <p>0: This is bits 31:0 of the offset pointer into the single, contiguous physical output region. This value will be added to the IA32_RTIT_OUTPUT_BASE value to form the physical address at which the next byte of packet output data will be written. This value must be less than or equal to the MaskOffsetTableOffset field, otherwise an operational error (Section 36.3.9) will be signaled when TraceEn is set.</p> <p>1: This field holds bits 31:0 of the offset pointer into the current ToPA output region. This value will be added to the output region base field, found in the current ToPA table entry, to form the physical address at which the next byte of trace output data will be written. This value must be less than the ToPA entry size, otherwise an operational error (Section 36.3.9) will be signaled when TraceEn is set.</p>

## 36.2.8 Interaction of Intel® Processor Trace and Other Processor Features

### 36.2.8.1 Intel® Transactional Synchronization Extensions (Intel® TSX)

The operation of Intel TSX is described in Chapter 14 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*. For tracing purpose, packet generation does not distinguish between hardware lock elision (HLE) and restricted transactional memory (RTM), but speculative execution does have impacts on the trace output. Specifically, packets are generated as instructions complete, even for instructions in a transactional region that is later aborted. For this reason, debugging software will need indication of the beginning and end of a transactional region; this will allow software to understand when instructions are part of a transactional region and whether that region has been committed.

To enable this, TSX information is included in a MODE packet leaf. The mode bits in the leaf are:

- **InTX**: Set to 1 on an TSX transaction begin, and cleared on transaction commit or abort.
- **TXAbort**: Set to 1 only when InTX transitions from 1 to 0 on an abort. Cleared otherwise.

If BranchEn=1, this MODE packet will be sent each time the transaction status changes. See Table 36-10 for details.

Table 36-10. TSX Packet Scenarios

TSX Event	Instruction	Packets
Transaction Begin	Either XBEGIN or XACQUIRE lock (the latter if executed transactionally)	MODE(TXAbort=0, InTX=1), FUP(CurrentIP)
Transaction Commit	Either XEND or XRELEASE lock, if transactional execution ends. This happens only on the outermost commit	MODE(TXAbort=0, InTX=0), FUP(CurrentIP)
Transaction Abort	XABORT or other transactional abort	MODE(TXAbort=1, InTX=0), FUP(CurrentIP), TIP(TargetIP)
Other	One of the following: <ul style="list-style-type: none"> <li>▪ Nested XBEGIN or XACQUIRE lock</li> <li>▪ An outer XACQUIRE lock that doesn't begin a transaction (InTX not set)</li> <li>▪ Non-outermost XEND or XRELEASE lock</li> </ul>	None. No change to TSX mode bits for these cases.

The CurrentIP listed above is the IP of the associated instruction. The TargetIP is the IP of the next instruction to be executed; for HLE, this is the XACQUIRE lock; for RTM, this is the fallback handler.

Intel PT stores are non-transactional, and thus packet writes are not rolled back on TSX abort.

## TSX and IP Filtering

A complication with tracking transactions is handling transactions that start or end outside of the tracing region. Transactions can't span across a change in ContextEn, because CPL changes and CR3 changes each cause aborts. But a transaction can start within the IP filter region and end outside it.

To assist the decoder handling this situation, MODE.TSX packets can be sent even if FilterEn=0, though there will be no FUP attached. Instead, they will merely serve to indicate to the decoder when transactions are active and when they are not. When tracing resumes (due to PacketEn=1), the last MODE.TSX preceding the TIP.PGE will indicate the current transaction status.

## System Management Mode (SMM)

SMM code has special privileges that non-SMM code does not have. Intel Processor Trace can be used to trace SMM code, but special care is taken to ensure that SMM handler context is not exposed in any non-SMM trace collection. Additionally, packet output from tracing non-SMM code cannot be written into memory space that is either protected by SMRR or used by the SMM handler.

SMM is entered via a system management interrupt (SMI). SMI delivery saves the value of IA32\_RTIT\_CTL.TraceEn into SMRAM and then clears it, thereby disabling packet generation.

The saving and clearing of IA32\_RTIT\_CTL.TraceEn ensures two things:

1. All internally buffered packet data is flushed before entering SMM (see Section 36.2.7.2).
2. Packet generation ceases before entering SMM, so any tracing that was configured outside SMM does not continue into SMM. No SMM instruction pointers or other state will be exposed in the non-SMM trace.

When the RSM instruction is executed to return from SMM, the TraceEn value that was saved by SMI delivery is restored, allowing tracing to be resumed. As is done any time packet generation is enabled, ContextEn is re-evaluated, based on the values of CPL, CR3, etc., established by RSM.

Like other interrupts, delivery of an SMI produces a FUP containing the IP of the next instruction to execute. By toggling TraceEn, SMI and RSM can produce TIP.PGD and TIP.PGE packets, respectively, indicating that tracing was disabled or re-enabled. See Table 36.7 for more information about packets entering and leaving SMM.

Although #SMI and RSM change CR3, PIP packets are not generated in these cases. With #SMI tracing is disabled before the CR3 change; with RSM TraceEn is restored after CR3 is written.

TraceEn must be cleared before executing RSM, otherwise it will cause a shutdown. Further, on processors that restrict use of Intel PT with LBRs (see Section 36.3.1.2), any RSM that results in enabling of both will cause a shutdown.

Intel PT can support tracing of System Transfer Monitor operating in SMM, see Section 36.6.

### 36.2.8.2 Virtual-Machine Extensions (VMX)

Initial implementations of Intel Processor Trace do not support tracing in VMX operation. Such processors indicate this by returning 0 for IA32\_VMX\_MISC[bit 14]. On these processors, execution of the VMXON instruction clears IA32\_RTIT\_CTL.TraceEn and any attempt to set that bit in VMX operation using WRMSR causes a general-protection exception (#GP).

Processors that support Intel Processor Trace in VMX operation return 1 for IA32\_VMX\_MISC[bit 14]. Details of tracing in VMX operation are described in Section 36.5.

### 36.2.8.3 Intel Software Guard Extensions (SGX)

SGX provides an application with ability to instantiate a protective container (an enclave) with confidentiality and integrity (see *Intel® Software Guard Extensions Programming Reference*). On a processor with both Intel PT and SGX enabled, when executing code within a production enclave, no control flow packets are produced by Intel PT. Enclave entry will clear ContextEn, thereby blocking control flow packet generation. A TIP.PGD packet will be generated if PacketEn=1 at the time of the entry.

Upon enclave exit, ContextEn will no longer be forced to 0. If other enables are set at the time, a TIP.PGE may be generated to indicate that tracing is resumed.



During the enclave execution, Intel PT remains enabled, and periodic or timing packets such as PSB, TSC, MTC, or CBR can still be generated. No IPs or other architectural state will be exposed.

For packet generation examples on enclave entry or exit, see Section 36.7.

### Debug Enclaves

SGX allows an enclave to be configured with relaxed protection of confidentiality for debug purposes, see *Intel® Software Guard Extensions Programming Reference*. In a debug enclave, Intel PT continues to function normally. Specifically, ContextEn is not impacted by enclave entry or exit. Hence the generation of ContextEn-dependent packets within a debug enclave is allowed.

#### 36.2.8.4 SENTER/ENTERACCS and ACM

GETSEC[SENDER] and GETSEC[ENTERACCS] instructions clear TraceEn, and it is not restored when those instruction complete. SENTER also causes TraceEn to be cleared on other logical processors when they rendezvous and enter the SENTER sleep state. In these two cases, the disabling of packet generation is not guaranteed to flush internally buffered packets. Some packets may be dropped.

When executing an authenticated code module (ACM), packet generation is silently disabled during ACRAM setup. TraceEn will be cleared, but no TIP.PGD packet is generated. After completion of the module, the TraceEn value will be restored. There will be no TIP.PGE packet, but timing packets, like TSC and CBR, may be produced.

#### 36.2.8.5 Intel® Memory Protection Extensions (Intel® MPX)

Bounds exceptions (#BR) caused by Intel MPX are treated like other exceptions, producing FUP and TIP packets that indicate the source and destination IPs.

## 36.3 CONFIGURATION AND PROGRAMMING GUIDELINE

### 36.3.1 Detection of Intel Processor Trace and Capability Enumeration

Processor support for Intel Processor Trace is indicated by CPUID.(EAX=07H,ECX=0H):EBX[bit 25] = 1. CPUID function 14H is dedicated to enumerate the resource and capability of processors that report CPUID.(EAX=07H,ECX=0H):EBX[bit 25] = 1. Different processor generations may have architecturally-defined variation in capabilities. Table 36-11 describes details of the enumerable capabilities that software must use across generations of processors that support Intel Processor Trace.

**Table 36-11. CPUID Leaf 14H Enumeration of Intel Processor Trace Capabilities**

CPUID.(EAX=14H,ECX=0)		Name	Description Behavior
Register	Bits		
EAX	31:0	Maximum valid sub-leaf Index	Specifies the index of the maximum valid sub-leaf for this CPUID leaf
EBX	0	CR3 Filtering Support	1: Indicates that IA32_RTIT_CTL.CR3Filter can be set to 1, and that IA32_RTIT_CR3_MATCH MSR can be accessed. See Section 36.2.7. 0: Indicates that writes that set IA32_RTIT_CTL.CR3Filter to 1, or any access to IA32_RTIT_CR3_MATCH, will #GP fault.
	1	Configurable PSB and Cycle-Accurate Mode Supported	1: (a) IA32_RTIT_CTL.PSBFreq can be set to a non-zero value, in order to select the preferred PSB frequency (see below for allowed values). (b) IA32_RTIT_STATUS.PacketByteCnt can be set to a non-zero value, and will be incremented by the processor when tracing to indicate progress towards the next PSB. If trace packet generation is enabled by setting TraceEn, a PSB will only be generated if PacketByteCnt=0. (c) IA32_RTIT_CTL.CYCEn can be set to 1 to enable Cycle-Accurate Mode. See Section 36.2.7. 0: (a) Any attempt to set IA32_RTIT_CTL.PSBFreq, to set IA32_RTIT_CTL.CYCEn, or write a non-zero value to IA32_RTIT_STATUS.PacketByteCnt any access to IA32_RTIT_CR3_MATCH, will #GP fault. (b) If trace packet generation is enabled by setting TraceEn, a PSB is always generated. (c) Any attempt to set IA32_RTIT_CTL.CYCEn will #GP fault.
	2	IP Filtering and TraceStop supported, and Preserve Intel PT MSRs across warm reset	1: (a) IA32_RTIT_CTL provides at one or more ADDRn_CFG field to configure the corresponding address range MSRs for IP Filtering or IP TraceStop. Each ADDRn_CFG field accepts a value in the range of 0:2 inclusive. The number of ADDRn_CFG fields is reported by CPUID.(EAX=14H, ECX=1):EAX.RANGECNT[2:0]. (b) At least one register pair IA32_RTIT_ADDRn_A and IA32_RTIT_ADDRn_B are provided to configure address ranges for IP filtering or IP TraceStop. (c) On warm reset, all Intel PT MSRs will retain their pre-reset values, though IA32_RTIT_CTL.TraceEn will be cleared. The Intel PT MSRs are listed in Section 36.2.7. 0: (a) An Attempt to write IA32_RTIT_CTL.ADDRn_CFG with non-zero encoding values will cause #GP. (b) Any access to IA32_RTIT_ADDRn_A and IA32_RTIT_ADDRn_B, will #GP fault. (c) On warm reset, all Intel PT MSRs will be cleared.
	3	MTC Supported	1: IA32_RTIT_CTL.MTCEn can be set to 1, and MTC packets will be generated. See Section 36.2.7. 0: An attempt to set IA32_RTIT_CTL.MTCEn or IA32_RTIT_CTL.MTCFreq to a non-zero value will #GP fault.
	4	PTWRITE Supported	1: Writes can set IA32_RTIT_CTL[12] (PTWEn) and IA32_RTIT_CTL[5] (FUPonPTW), and PTWRITE can generate packets. 0: Writes that set IA32_RTIT_CTL[12] or IA32_RTIT_CTL[5] will #GP, and PTWRITE will #UD fault.
	5	Power Event Trace Supported	1: Writes can set IA32_RTIT_CTL[4] (PwrEvtEn), enabling Power Event Trace packet generation. 0: Writes that set IA32_RTIT_CTL[4] will #GP.
		31:6	Reserved

Table 36-11. CPUID Leaf 14H Enumeration of Intel Processor Trace Capabilities (Contd.)

CPUID.(EAX=14H,ECX=0)		Name	Description Behavior
Register	Bits		
ECX	0	ToPA Output Supported	1: Tracing can be enabled with IA32_RTIT_CTL.ToPA = 1, hence utilizing the ToPA output scheme (Section 36.2.6.2) IA32_RTIT_OUTPUT_BASE and IA32_RTIT_OUTPUT_MASK_PTRS MSRs can be accessed. 0: Unless CPUID.(EAX=14H, ECX=0);ECX.SNGLRNGOUT[bit 2] = 1. writes to IA32_RTIT_OUTPUT_BASE or IA32_RTIT_OUTPUT_MASK_PTRS. MSRs will #GP fault.
	1	ToPA Tables Allow Multiple Output Entries	1: ToPA tables can hold any number of output entries, up to the maximum allowed by the MaskOffsetTableOffset field of IA32_RTIT_OUTPUT_MASK_PTRS. 0: ToPA tables can hold only one output entry, which must be followed by an END=1 entry which points back to the base of the table. Further, ToPA PMIs will be delivered before the region is filled. See ToPA PMI in Section 36.2.6.2. If there is more than one output entry before the END entry, or if the END entry has the wrong base address, an operational error will be signaled (see “ToPA Errors” in Section 36.2.6.2).
	2	Single-Range Output Supported	1: Enabling tracing (TraceEn=1) with IA32_RTIT_CTL.ToPA=0 is supported. 0: Unless CPUID.(EAX=14H, ECX=0);ECX.TOPAOUT[bit 0] = 1. writes to IA32_RTIT_OUTPUT_BASE or IA32_RTIT_OUTPUT_MASK_PTRS. MSRs will #GP fault.
	3	Output to Trace Transport Subsystem Supported	1: Setting IA32_RTIT_CTL.FabricEn to 1 is supported. 0: IA32_RTIT_CTL.FabricEn is reserved. Write 1 to IA32_RTIT_CTL.FabricEn will #GP fault.
	30:4	Reserved	
	31	IP Payloads are LIP	1: Generated packets which contain IP payloads have LIP values, which include the CS base component. 0: Generated packets which contain IP payloads have RIP values, which are the offset from CS base.
EDX	31:0	Reserved	

If CPUID.(EAX=14H, ECX=0):EAX reports a non-zero value, additional capabilities of Intel Processor Trace are described in the sub-leaves of CPUID leaf 14H.

Table 36-12. CPUID Leaf 14H, sub-leaf 1H Enumeration of Intel Processor Trace Capabilities

CPUID.(EAX=14H,ECX=1)		Name	Description Behavior
Register	Bits		
EAX	2:0	Number of Address Ranges	A non-zero value specifies the number ADDRn_CFG field supported in IA32_RTIT_CTL and the number of register pair IA32_RTIT_ADDRn_A/IA32_RTIT_ADDRn_B supported for IP filtering and IP TraceStop. <b>NOTE:</b> Currently, no processors support more than 4 address ranges.
	15:3	Reserved	
	31:16	Bitmap of supported MTC Period Encodings	The non-zero bit positions indicate the map of supported encoding values for the IA32_RTIT_CTL.MTCFreq field. This applies only if CPUID.(EAX=14H, ECX=0);EBX.MTC[bit 3] = 1 (MTC Packet generation is supported), otherwise the MTCFreq field is reserved to 0. Each bit position in this field represents 1 encoding value in the 4-bit MTCFreq field (ie, bit 0 is associated with encoding value 0). For each bit: 1: MTCFreq can be assigned the associated encoding value. 0: MTCFreq cannot be assigned to the associated encoding value. A write to IA32_RTIT_CTL.MTCFreq with unsupported encoding will cause #GP fault.
EBX	15:0	Bitmap of supported Cycle Threshold values	The non-zero bit positions indicate the map of supported encoding for the IA32_RTIT_CTL.CycThresh field. This applies only if CPUID.(EAX=14H, ECX=0);EBX.CPSB_CAM[bit 1] = 1 (Cycle-Accurate Mode is Supported), otherwise the CycThresh field is reserved to 0. See Section 36.2.7. Each bit position in this field represents 1 encoding value in the 4-bit CycThresh field (ie, bit 0 is associated with encoding value 0). For each bit: 1: CycThresh can be assigned the associated encoding value. 0: CycThresh cannot be assigned to the associated encoding value. A write to CycThresh with unsupported encoding will cause #GP fault.
	31:16	Bitmap of supported Configurable PSB Frequency encoding	The non-zero bit positions indicate the map of supported encoding for the IA32_RTIT_CTL.PSBFreq field. This applies only if CPUID.(EAX=14H, ECX=0);EBX.CPSB_CAM[bit 1] = 1 (Configurable PSB is supported), otherwise the PSBFreq field is reserved to 0. See Section 36.2.7. Each bit position in this field represents 1 encoding value in the 4-bit PSBFreq field (ie, bit 0 is associated with encoding value 0). For each bit: 1: PSBFreq can be assigned the associated encoding value. 0: PSBFreq cannot be assigned to the associated encoding value. A write to PSBFreq with unsupported encoding will cause #GP fault.
ECX	31:0	Reserved	
EDX	31:0	Reserved	

### 36.3.1.1 Packet Decoding of RIP versus LIP

FUP, TIP, TIP.PGE, and TIP.PGE packets can contain an instruction pointer (IP) payload. On some processor generations, this payload will be an effective address (RIP), while on others this will be a linear address (LIP). In the former case, the payload is the offset from the current CS base address, while in the latter it is the sum of the offset and the CS base address (Note that in real mode, the CS base address is the value of CS<<4, while in protected mode the CS base address is the base linear address of the segment indicated by the CS register.). Which IP type is in use is indicated by enumeration (see CPUID.(EAX=14H, ECX=0):ECX.LIP[bit 31] in Table 36-11).

For software that executes while the CS base address is 0 (including all software executing in 64-bit mode), the difference is indistinguishable. A trace decoder must account for cases where the CS base address is not 0 and the resolved LIP will not be evident in a trace generated on a CPU that enumerates use of RIP. This is likely to cause problems when attempting to link the trace with the associated binaries.

Note that IP comparison logic, for IP filtering and TraceStop range calculation, is based on the same IP type as these IP packets. For processors that output RIP, the IP comparison mechanism is also based on RIP, and hence on those processors RIP values should be written to IA32\_RTIT\_ADDRn\_[AB] MSRs. This can produce differing behavior if the same trace configuration setting is run on processors reporting different IP types, i.e. CPUID.(EAX=14H, ECX=0):ECX.LIP[bit 31]. Care should be taken to check CPUID when configuring IP filters.

### 36.3.1.2 Model Specific Capability Restrictions

Some processor generations impose restrictions that prevent use of LBRs/BTS/BTM/LERs when software has enabled tracing with Intel Processor Trace. On these processors, when TraceEn is set, updates of LBR, BTS, BTM, LERs are suspended but the states of the corresponding IA32\_DEBUGCTL control fields remained unchanged as if it were still enabled. When TraceEn is cleared, the LBR array is reset, and LBR/BTS/BTM/LERs updates will resume. Further, reads of these registers will return 0, and writes will be dropped.

The list of MSRs whose updates/accesses are restricted follows.

- MSR\_LASTBRANCH\_x\_TO\_IP, MSR\_LASTBRANCH\_x\_FROM\_IP, MSR\_LBR\_INFO\_x, MSR\_LASTBRANCH\_TOS
- MSR\_LER\_FROM\_IP, MSR\_LER\_TO\_IP
- MSR\_LBR\_SELECT

For processor with CPUID DisplayFamily\_DisplayModel signature of 06\_3DH, 06\_47H, 06\_4EH, 06\_4FH, 06\_56H and 06\_5EH, the use of Intel PT and LBRs are mutually exclusive.

## 36.3.2 Enabling and Configuration of Trace Packet Generation

To configure trace packets, enable packet generation, and capture packets, software starts with using CPUID instruction to detect its feature flag, CPUID.(EAX=07H, ECX=0H):EBX[bit 25] = 1; followed by enumerating the capabilities described in Section 36.3.1.

Based on the capability queried from Section 36.3.1, software must configure a number of model-specific registers. This section describes programming considerations related to those MSRs.

### 36.3.2.1 Enabling Packet Generation

When configuring and enabling packet generation, the IA32\_RTIT\_CTL MSR should be written after any other Intel PT MSRs have been written, since writes to the other configuration MSRs cause a general-protection fault (#GP) if TraceEn = 1. If a prior trace collection context is not being restored, then software should first clear IA32\_RTIT\_STATUS. This is important since the Stopped, and Error fields are writable; clearing the MSR clears any values that may have persisted from prior trace packet collection contexts. See Section 36.2.7.2 for details of packets generated by setting TraceEn to 1.

If setting TraceEn to 1 causes an operational error (see Section 36.3.9), there may be a delay after the WRMSR completes before the error is signaled in the IA32\_RTIT\_STATUS MSR.

While packet generation is enabled, the values of some configuration MSRs (e.g., IA32\_RTIT\_STATUS and IA32\_RTIT\_OUTPUT\_\*) are transient, and reads may return values that are out of date. Only after packet generation is disabled (by clearing TraceEn) do reads of these MSRs return reliable values.

### 36.3.2.2 Disabling Packet Generation

After disabling packet generation by clearing IA32\_RTIT\_CTL, it is advisable to read the IA32\_RTIT\_STATUS MSR (Section 36.2.7.4):

- If the Error bit is set, an operational error was encountered, and the trace is most likely compromised. Software should check the source of the error (by examining the output MSR values), correct the source of the problem, and then attempt to gather the trace again. For details on operational errors, see Section 36.3.9. Software should clear IA32\_RTIT\_STATUS.Error before re-enabling packet generation.
- If the Stopped bit is set, software execution encountered an IP TraceStop (see Section 36.2.4.3) or the ToPA Stop condition (see “ToPA STOP” in Section 36.2.6.2) before packet generation was disabled.

### 36.3.3 Flushing Trace Output

Packets are first buffered internally and then written out asynchronously. To collect packet output for post-processing, a collector needs first to ensure that all packet data has been flushed from internal buffers. Software can ensure this by stopping packet generation by clearing IA32\_RTIT\_CTL.TraceEn (see “Disabling Packet Generation” in Section 36.2.7.2).

When software clears IA32\_RTIT\_CTL.TraceEn to flush out internally buffered packets, the logical processor issues an SFENCE operation which ensures that WC trace output stores will be ordered with respect to the next store, or serializing operation. A subsequent read from the same logical processor will see the flushed trace data, while a read from another logical processor should be preceded by a store, fence, or architecturally serializing operation on the tracing logical processor.

When the flush operations complete, the IA32\_RTIT\_OUTPUT\_\* MSR values indicate where the trace ended. While TraceEn is set, these MSRs may hold stale values.

### 36.3.4 Warm Reset

The MSRs software uses to program Intel Processor Trace are cleared after a power-on RESET (or cold RESET). On a warm RESET, the contents of those MSRs can retain their values from before the warm RESET with the exception that IA32\_RTIT\_CTL.TraceEn will be cleared (which may have the side effect of clearing some bits in IA32\_RTIT\_STATUS).

### 36.3.5 Context Switch Consideration

To facilitate construction of instruction execution traces at the granularity of a software process or thread context, software can save and restore the states of the trace configuration MSRs across the process or thread context switch boundary. The principle is the same as saving and restoring the typical architectural processor states across context switches.

#### 36.3.5.1 Manual Trace Configuration Context Switch

The configuration can be saved and restored through a sequence of instructions of RDMSR, management of MSR content and WRMSR. To stop tracing and to ensure that all configuration MSRs contain stable values, software must clear IA32\_RTIT\_CTL.TraceEn before reading any other trace configuration MSRs. The recommended method for saving trace configuration context manually follows:

1. RDMSR IA32\_RTIT\_CTL, save value to memory
2. WRMSR IA32\_RTIT\_CTL with saved value from RDMSR above and TraceEn cleared
3. RDMSR all other configuration MSRs whose values had changed from previous saved value, save changed values to memory

When restoring the trace configuration context, IA32\_RTIT\_CTL should be restored last:

1. Read saved configuration MSR values, aside from IA32\_RTIT\_CTL, from memory, and restore them with WRMSR
2. Read saved IA32\_RTIT\_CTL value from memory, and restore with WRMSR.

### 36.3.5.2 Trace Configuration Context Switch Using XSAVES/XRSTORS

On processors whose XSAVE feature set supports XSAVES and XRSTORS, the Trace configuration state can be saved using XSAVES and restored by XRSTORS, in conjunction with the bit field associated with supervisory state component in IA32\_XSS. See Chapter 13, “Managing State Using the XSAVE Feature Set” of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*.

The layout of the trace configuration component state in the XSAVE area is shown in Table 36-13.<sup>1</sup>

**Table 36-13. Memory Layout of the Trace Configuration State Component**

Offset within Component Area	Field	Offset within Component Area	Field
0H	IA32_RTIT_CTL	08H	IA32_RTIT_OUTPUT_BASE
10H	IA32_RTIT_OUTPUT_MASK_PTRS	18H	IA32_RTIT_STATUS
20H	IA32_RTIT_CR3_MATCH	28H	IA32_RTIT_ADDR0_A
30H	IA32_RTIT_ADDR0_B	38H	IA32_RTIT_ADDR1_A
40H	IA32_RTIT_ADDR1_B	48H-End	Reserved

The IA32\_XSS MSR is zero coming out of RESET. Once IA32\_XSS[bit 8] is set, system software operating at CPL=0 can use XSAVES/XRSTORS with the appropriate requested-feature bitmap (RFBM) to manage supervisor state components in the XSAVE map. See Chapter 13, “Managing State Using the XSAVE Feature Set” of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*.

## 36.3.6 Cycle-Accurate Mode

Intel PT can be run in a cycle-accurate mode which enables CYC packets (see Section 36.4.2.14) that provide low-level information in the processor core clock domain. This cycle counter data in CYC packets can be used to compute IPC (Instructions Per Cycle), or to track wall-clock time on a fine-grain level.

To enable cycle-accurate mode packet generation, software should set IA32\_RTIT\_CTL.CYCEn=1. It is recommended that software also set TSCEn=1 anytime cycle-accurate mode is in use. With this, all CYC-eligible packets will be preceded by a CYC packet, the payload of which indicates the number of core clock cycles since the last CYC packet. In cases where multiple CYC-eligible packets are generated in a single cycle, only a single CYC will be generated before the CYC-eligible packets, otherwise each CYC-eligible packet will be preceded by its own CYC. The CYC-eligible packets are:

- TNT, TIP, TIP.PGE, TIP.PGD, MODE.EXEC, MODE.TSX, PIP, VMCS, OVF, MTC, TSC, PTWRITE, EXSTOP

TSC packets are generated when there is insufficient information to reconstruct wall-clock time, due to tracing being disabled (TriggerEn=0), or power down scenarios like a transition to a deep-sleep MWAIT C-state. In this case, the CYC that is generated along with the TSC will indicate the number of cycles actively tracing (those powered up, with TriggerEn=1) executed between the last CYC packet and the TSC packet. And hence the amount of time spent while tracing is inactive can be inferred from the difference in time between that expected based on the CYC value, and the actual time indicated by the TSC.

Additional CYC packets may be sent stand-alone, so that the processor can ensure that the decoder is aware of the number of cycles that have passed before the internal hardware counter wraps, or is reset due to other micro-architectural condition. There is no guarantee at what intervals these standalone CYC packets will be sent, except that they will be sent before the wrap occurs. An illustration is given below.

1. Table 36-13 documents support for the MSRs defining address ranges 0 and 1. Processors that provide XSAVE support for Intel Processor Trace support only those address ranges.



**Example 36-1. An Illustrative CYC Packet Example**

Time (cycles)	Instruction Snapshot	Generated Packets	Comment
x	call %eax	CYC(?), TIP	?Elapsed cycles from the previous CYC unknown
x + 2	call %ebx	CYC(2), TIP	1 byte CYC packet; 2 cycles elapsed from the previous CYC
x + 8	jnz Foo (not taken)	CYC(6)	1 byte CYC packet
x + 9	ret (compressed)		
x + 12	jnz Bar (taken)		
x + 16	ret (uncompressed)	TNT, CYC(8), TIP	1 byte CYC packet
x + 4111		CYC(4095)	2 byte CYC packet
x + 12305		CYC(8194)	3 byte CYC packet
x + 16332	mov cr3, %ebx	CYC(4027), PIP	2 byte CYC packet

**36.3.6.1 Cycle Counter**

The cycle counter is implemented in hardware (independent of the time stamp counter or performance monitoring counters), and is a simple incrementing counter that does not saturate, but rather wraps. The size of the counter is implementation specific.

The cycle counter is reset to zero any time that TriggerEn is cleared, and when a CYC packet is sent. The cycle counter will continue to count when ContextEn or FilterEn are cleared, and cycle packets will still be generated. It will not count during sleep states that result in Intel PT logic being powered-down, but will count up to the point where clocks are disabled, and resume counting once they are re-enabled.

**36.3.6.2 Cycle Packet Semantics**

Cycle-accurate mode adheres to the following protocol:

- All packets that precede a CYC packet represent instructions or events that took place before the CYC time.
- All packets that follow a CYC packet represent instructions or events that took place at the same time as, or after, the CYC time.
- The CYC-eligible packet that immediately follows a CYC packet represents an instruction or event that took place at the same time as the CYC time.

These items above give the decoder a means to apply CYC packets to a specific instruction in the assembly stream. Most packets represent a single instruction or event, and hence the CYC packet that precedes each of those packets represents the retirement time of that instruction or event. In the case of TNT packets, up to 6 conditional branches and/or compressed RETs may be contained in the packet. In this case, the preceding CYC packet provides the retirement time of the first branch in the packet. It is possible that multiple branches retired in the same cycle as that first branch in the TNT, but the protocol will not make that obvious. Also note that a MTC packet could be generated in the same cycle as the first JCC in the TNT packet. In this case, the CYC would precede both the MTC and the TNT, and apply to both.

Note that there are times when the cycle counter will stop counting, though cycle-accurate mode is enabled. After any such scenario, a CYC packet followed by TSC packet will be sent. See Section 36.8.3.2 to understand how to interpret the payload values

**Multi-packet Instructions or Events**

Some operations, such as interrupts or task switches, generate multiple packets. In these cases, multiple CYC packets may be sent for the operation, preceding each CYC-eligible packet in the operation. An example, using a task switch on a software interrupt, is shown below.



**Example 36-2. An Example of CYC in the Presence of Multi-Packet Operations**

Time (cycles)	Instruction Snapshot	Generated Packets
x	jnz Foo (not taken)	CYC(?),
x + 2	ret (compressed)	
x + 8	jnz Bar (taken)	
x + 9	jmp %eax	TNT, CYC(9), TIP
x + 12	jnz Bar (not taken)	CYC(3)
x + 32	int3 (task gate)	TNT, FUP, CYC(10), PIP, CYC(20), MODE.Exec, TIP

**36.3.6.3 Cycle Thresholds**

Software can opt to reduce the frequency of cycle packets, a trade-off to save bandwidth and intrusion at the expense of precision. This is done by utilizing a cycle threshold (see Section 36.2.7.2).

IA32\_RTIT\_CTL.CycThresh indicates to the processor the minimum number of cycles that must pass before the next CYC packet should be sent. If this value is 0, no threshold is used, and CYC packets can be sent every cycle in which a CYC-eligible packet is generated. If this value is greater than 0, the hardware will wait until the associated number of cycles have passed since the last CYC packet before sending another. CPUID provides the threshold options for CycThresh, see Section 36.3.1.

Note that the cycle threshold does not dictate how frequently a CYC packet will be posted, it merely assigns the maximum frequency. If the cycle threshold is 16, a CYC packet can be posted no more frequently than every 16 cycles. However, once that threshold of 16 cycles has passed, it still requires a new CYC-eligible packet to be generated before a CYC will be inserted. Table 36-14 illustrates the threshold behavior.

**Table 36-14. An Illustrative CYC Packet Example**

Time (cycles)	Instruction Snapshot	Threshold			
		0	16	32	64
x	jmp %eax	CYC, TIP	CYC, TIP	CYC, TIP	CYC, TIP
x + 9	call %ebx	CYC, TIP	TIP	TIP	TIP
x + 15	call %ecx	CYC, TIP	TIP	TIP	TIP
x + 30	jmp %edx	CYC, TIP	CYC, TIP	TIP	TIP
x + 38	mov cr3, %eax	CYC, PIP	PIP	CYC, PIP	PIP
x + 46	jmp [%eax]	CYC, TIP	CYC, TIP	TIP	TIP
x + 64	call %edx	CYC, TIP	CYC, TIP	TIP	CYC, TIP
x + 71	jmp %edx	CYC, TIP	TIP	CYC, TIP	TIP

**36.3.7 Decoder Synchronization (PSB+)**

The PSB packet (Section 36.4.2.17) serves as a synchronization point for a trace-packet decoder. It is a pattern in the trace log for which the decoder can quickly scan to align packet boundaries. No legal packet combination can result in such a byte sequence. As such, it serves as the starting point for packet decode. To decode a trace log properly, the decoder needs more than simply to be aligned: it needs to know some state and potentially some timing information as well. The decoder should never need to retain any information (e.g., LastIP, call stack, compound packet event) across a PSB; all compound packet events will be completed before a PSB, and any compression state will be reset.

When a PSB packet is generated, it is followed by a PSBEND packet (Section 36.4.2.18). One or more packets may be generated in between those two packets, and these inform the decoder of the current state of the processor. These packets, known collectively as PSB+, should be interpreted as “status only”, since they do not imply any change of state at the time of the PSB, nor are they associated directly with any instruction or event. Thus, the

normal binding and ordering rules that apply to these packets outside of PSB+ can be ignored when these packets are between a PSB and PSBEND. They inform the decoder of the state of the processor at the time of the PSB.

PSB+ can include:

- Timestamp (TSC), if IA32\_RTIT\_CTL.TSCEn=1.
- Timestamp-MTC Align (TMA), if IA32\_RTIT\_CTL.TSCEn=1 && IA32\_RTIT\_CTL.MTCEn=1.
- Paging Info Packet (PIP), if ContextEn=1 and IA32\_RTIT\_CTL.OS=1. The non-root bit (NR) is set if the logical processor is in VMX non-root operation and the “conceal VMX non-root operation from Intel PT”. VM-execution control is 0.
- VMCS packet, if either the logical is in VMX root operation or the logical processor is in VMX non-root operation and the “conceal VMX non-root operation from Intel PT” VM-execution control is 0.
- Core Bus Ratio (CBR).
- MODE.TSX, if ContextEn=1 and BranchEn = 1.
- MODE.Exec, if PacketEn=1.
- Flow Update Packet (FUP), if PacketEn=1.

PSB is generated only when TriggerEn=1; hence PSB+ has the same dependencies. The ordering of packets within PSB+ is not fixed. Timing packets such as CYC and MTC may be generated between PSB and PSBEND, and their meanings are the same as outside PSB+.

Note that an overflow can occur during PSB+, and this could cause the PSBEND packet to be lost. For this reason, the OVF packet should also be viewed as terminating PSB+.

### 36.3.8 Internal Buffer Overflow

In the rare circumstances when new packets need to be generated but the processor’s dedicated internal buffers are all full, an “internal buffer overflow” occurs. On such an overflow packet generation ceases (as packets would need to enter the processor’s internal buffer) until the overflow resolves. Once resolved, packet generation resumes.

When the buffer overflow is cleared, an OVF packet (Section 36.4.2.16) is generated, and the processor ensures that packets which follow the OVF are not compressed (IP compression or RET compression) against packets that were lost.

If IA32\_RTIT\_CTL.BranchEn = 1, the OVF packet will be followed by a FUP if the overflow resolves while PacketEn=1. If the overflow resolves while PacketEn = 0 no packet is generated, but a TIP.PGE will naturally be generated later, once PacketEn = 1. The payload of the FUP or TIP.PGE will be the Current IP of the first instruction upon which tracing resumes after the overflow is cleared. Between the OVF and following FUP or TIP.PGE, there may be packets that do not depend on PacketEn, such as timing packets. If the overflow resolves while PacketEn=0, other packets that are not dependent on PacketEn may come before the TIP.PGE.

#### 36.3.8.1 Overflow Impact on Enables

The address comparisons to ADDRn ranges, for IP filtering and TraceStop (Section 36.2.4.3), continue during a buffer overflow, and TriggerEn, ContextEn, and FilterEn may change during a buffer overflow. Like other packets, however, any TIP.PGE or TIP.PGD packets that would have been generated will be lost. Further, IA32\_RTIT\_STATUS.PacketByteCnt will not increment, since it is only incremented when packets are generated.

If a TraceStop event occurs during the buffer overflow, IA32\_RTIT\_STATUS.Stopped will still be set, tracing will cease as a result. However, the TraceStop packet, and any TIP.PGD that result from the TraceStop, may be dropped.

#### 36.3.8.2 Overflow Impact on Timing Packets

Any timing packets that are generated during a buffer overflow will be dropped. If only a few MTC packets are dropped, a decoder should be able to detect this by noticing that the time value in the first MTC packet after the buffer overflow incremented by more than one. If the buffer overflow lasted long enough that 256 MTC packets are lost (and thus the MTC packet ‘wraps’ its 8-bit CTC value), then the decoder may be unable to properly understand

the trace. This is not an expected scenario. No CYC packets are generated during overflow, even if the cycle counter wraps.

Note that, if cycle-accurate mode is enabled, the OVF packet will generate a CYC packet. Because the cycle counter counts during overflows, this CYC packet can provide the duration of the overflow. However, there is a risk that the cycle counter wrapped during the overflow, which could render this CYC misleading.

### 36.3.9 Operational Errors

Errors are detected as a result of packet output configuration problems, which can include output alignment issues, ToPA reserved bit violations, or overlapping packet output with restricted memory. See “ToPA Errors” in Section 36.2.6.2 for details on ToPA errors, and Section 36.2.6.4 for details on restricted memory errors. Operational errors are only detected and signaled when TraceEn=1.

When an operational error is detected, tracing is disabled and the error is logged. Specifically, IA32\_RTIT\_STATUS.Error is set, which will cause IA32\_RTIT\_STATUS.TriggerEn to be 0. This will disable generation of all packets. Some causes of operational errors may lead to packet bytes being dropped.

It should be noted that the timing of error detection may not be predictable. Errors are signaled when the processor encounters the problematic configuration. This could be as soon as packet generation is enabled but could also be later when the problematic entry or field needs to be used.

Once an error is signaled, software should disable packet generation by clearing TraceEn, diagnose and fix the error condition, and clear IA32\_RTIT\_STATUS.Error. At this point, packet generation can be re-enabled.

## 36.4 TRACE PACKETS AND DATA TYPES

This section details the data packets generated by Intel Processor Trace. It is useful for developers writing the interpretation code that will decode the data packets and apply it to the traced source code.

### 36.4.1 Packet Relationships and Ordering

This section introduces the concept of packet “binding”, which involves determining the IP in a binary disassembly at which the change indicated by a given packet applies. Some packets have the associated IP as the payload (FUP, TIP), while for others the decoder need only search for the next instance of a particular instruction (or instructions) to bind the packet (TNT). However, in many cases, the decoder will need to consider the relationship between packets, and to use this packet context to determine how to bind the packet.

Section 36.4.2 below provides detailed descriptions of the packets, including how packets bind to IPs in the disassembly, to other packets, or to nothing at all. Many packets listed are simple to bind, because they are generated in only a few scenarios. Those that require more consideration are typically part of “compound packet events”, such as interrupts, exceptions, and some instructions, where multiple packets are generated by a single operation (instruction or event). These compound packet events frequently begin with a FUP to indicate the source address (if it is not clear from the disassembly), and are concluded by a TIP or TIP.PGD packet that indicates the destination address (if one is provided). In this scenario, the FUP is said to be “coupled” with the TIP packet.

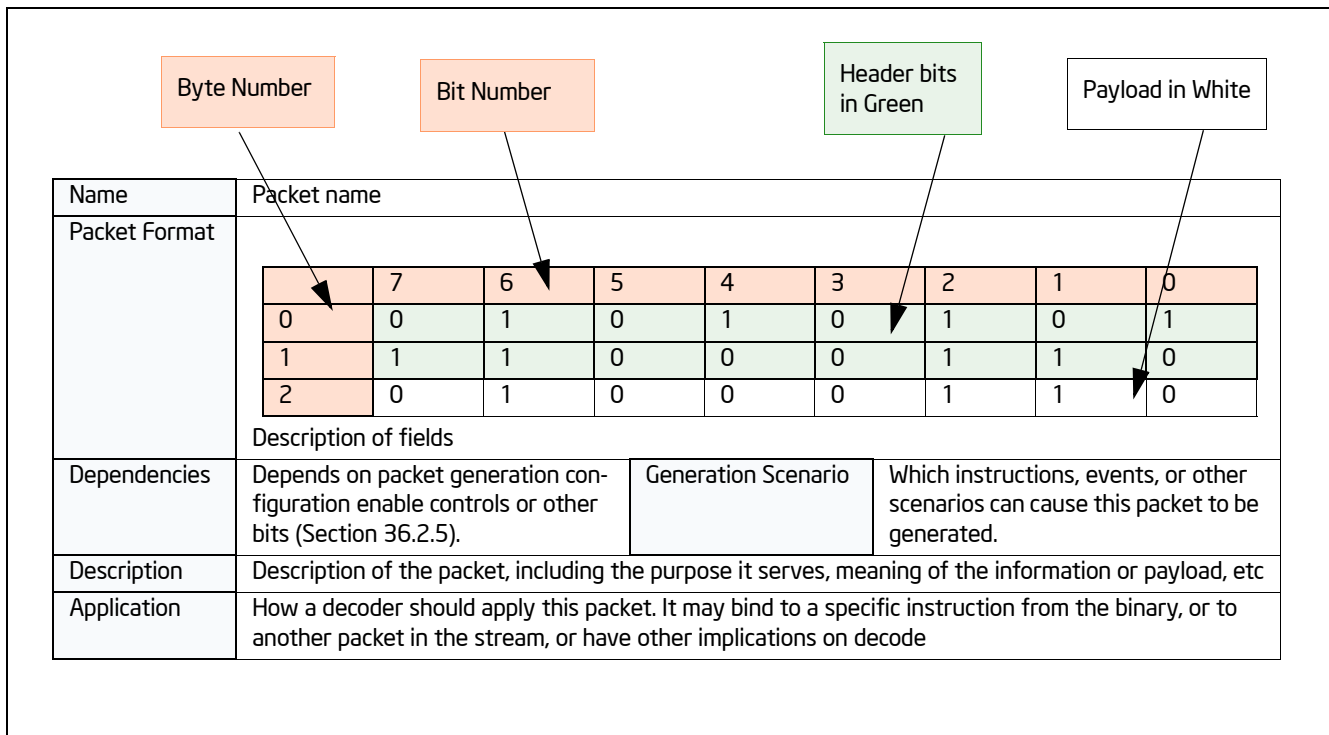
Other packets could be in between the coupled FUP and TIP packet. Timing packets, such as TSC, MTC, CYC, or CBR, could arrive at any time, and hence could intercede in a compound packet event. If an operation changes CR3 or the processor’s mode of execution, a state update packet (i.e., PIP or MODE) is generated. The state changes indicated by these intermediate packets should be applied at the IP of the TIP\* packet. A summary of compound packet events is provided in Table 36-15; see Section 36.4.2 for more per-packet details and Section 36.7 for more detailed packet generation examples.

**Table 36-15. Compound Packet Event Summary**

Event Type	Beginning	Middle	End	Comment
Unconditional, uncompressed control-flow transfer	FUP or none	Any combination of PIP, VMCS, MODE.Exec, or none	TIP or TIP.PGD	FUP only for asynchronous events. Order of middle packets may vary. PIP/VMCS/MODE only if the operation modifies the state tracked by these respective packets
TSX Update	MODE.TSX, and (FUP or none)	None	TIP, TIP.PGD, or none	FUP TIP/TIP.PGD only for TSX abort cases
Overflow	OVF	PSB, PSBEND, or none	FUP or TIP.PGE	FUP if overflow resolves while ContextEn=1, else TIP.PGE.

### 36.4.2 Packet Definitions

The following description of packet definitions are in tabular format. Figure 36-3 explains how to interpret them. Packet bits listed as "RSVD" are not guaranteed to be 0.



**Figure 36-3. Interpreting Tabular Definition of Packet Format**

#### 36.4.2.1 Taken/Not-taken (TNT) Packet

**Table 36-16. TNT Packet Definition**

Name	Taken/Not-taken (TNT) Packet																													
Packet Format	<table border="1"> <thead> <tr> <th></th> <th>7</th> <th>6</th> <th>5</th> <th>4</th> <th>3</th> <th>2</th> <th>1</th> <th>0</th> <th></th> </tr> </thead> <tbody> <tr> <td>0</td> <td>1</td> <td>B<sub>1</sub></td> <td>B<sub>2</sub></td> <td>B<sub>3</sub></td> <td>B<sub>4</sub></td> <td>B<sub>5</sub></td> <td>B<sub>6</sub></td> <td>0</td> <td>Short TNT</td> </tr> </tbody> </table>											7	6	5	4	3	2	1	0		0	1	B <sub>1</sub>	B <sub>2</sub>	B <sub>3</sub>	B <sub>4</sub>	B <sub>5</sub>	B <sub>6</sub>	0	Short TNT
	7	6	5	4	3	2	1	0																						
0	1	B <sub>1</sub>	B <sub>2</sub>	B <sub>3</sub>	B <sub>4</sub>	B <sub>5</sub>	B <sub>6</sub>	0	Short TNT																					

Table 36-16. TNT Packet Definition (Contd.)

	B1...BN represent the last N conditional branch or compressed RET (Section 36.4.2.2) results, such that B1 is oldest and BN is youngest. The short TNT packet can contain from 1 to 6 TNT bits. The long TNT packet can contain up from 1 to 47 TNT bits.									
		7	6	5	4	3	2	1	0	
	0	0	0	0	0	0	0	1	0	Long TNT
	1	1	0	1	0	0	0	1	1	
	2	B <sub>40</sub>	B <sub>41</sub>	B <sub>42</sub>	B <sub>43</sub>	B <sub>44</sub>	B <sub>45</sub>	B <sub>46</sub>	B <sub>47</sub>	
	3	B <sub>32</sub>	B <sub>33</sub>	B <sub>34</sub>	B <sub>35</sub>	B <sub>36</sub>	B <sub>37</sub>	B <sub>38</sub>	B <sub>39</sub>	
	4	B <sub>24</sub>	B <sub>25</sub>	B <sub>26</sub>	B <sub>27</sub>	B <sub>28</sub>	B <sub>29</sub>	B <sub>30</sub>	B <sub>31</sub>	
	5	B <sub>16</sub>	B <sub>17</sub>	B <sub>18</sub>	B <sub>19</sub>	B <sub>20</sub>	B <sub>21</sub>	B <sub>22</sub>	B <sub>23</sub>	
	6	B <sub>8</sub>	B <sub>9</sub>	B <sub>10</sub>	B <sub>11</sub>	B <sub>12</sub>	B <sub>13</sub>	B <sub>14</sub>	B <sub>15</sub>	
	7	1	B <sub>1</sub>	B <sub>2</sub>	B <sub>3</sub>	B <sub>4</sub>	B <sub>5</sub>	B <sub>6</sub>	B <sub>7</sub>	
	Irrespective of how many TNT bits is in a packet, the last valid TNT bit is followed by a trailing 1, or Stop bit, as shown above. If the TNT packet is not full (fewer than 6 TNT bits for the Short TNT, or fewer than 47 TNT bits for the Long TNT), the Stop bit moves up, and the trailing bits of the packet are filled with 0s. Examples of these "partial TNTs" are shown below.									
		7	6	5	4	3	2	1	0	
	0	0	0	1	B <sub>1</sub>	B <sub>2</sub>	B <sub>3</sub>	B <sub>4</sub>	0	Short TNT
		7	6	5	4	3	2	1	0	
	0	0	0	0	0	0	0	1	0	Long TNT
	1	1	0	1	0	0	0	1	1	
	2	B <sub>24</sub>	B <sub>25</sub>	B <sub>26</sub>	B <sub>27</sub>	B <sub>28</sub>	B <sub>29</sub>	B <sub>30</sub>	B <sub>31</sub>	
	3	B <sub>16</sub>	B <sub>17</sub>	B <sub>18</sub>	B <sub>19</sub>	B <sub>20</sub>	B <sub>21</sub>	B <sub>22</sub>	B <sub>23</sub>	
	4	B <sub>8</sub>	B <sub>9</sub>	B <sub>10</sub>	B <sub>11</sub>	B <sub>12</sub>	B <sub>13</sub>	B <sub>14</sub>	B <sub>15</sub>	
	5	1	B <sub>1</sub>	B <sub>2</sub>	B <sub>3</sub>	B <sub>4</sub>	B <sub>5</sub>	B <sub>6</sub>	B <sub>7</sub>	
	6	0	0	0	0	0	0	0	0	
	7	0	0	0	0	0	0	0	0	
Dependencies	PacketEn			Generation Scenario		On a conditional branch or compressed RET, if it fills the TNT. Also, partial TNTs may be generated at any time, as a result of other packets being generated, or certain micro-architectural conditions occurring, before the TNT is full.				
Description	Provides the taken/not-taken results for the last 1–N conditional branches (Jcc, J*CXZ, or LOOP) or compressed RETs (Section 36.4.2.2). The TNT payload bits should be interpreted as follows: <ul style="list-style-type: none"> <li>▪ 1 indicates a taken conditional branch, or a compressed RET</li> <li>▪ 0 indicates a not-taken conditional branch</li> </ul>									
Application	Each valid payload bit (that is, bits between the header bits and the trailing Stop bit) applies to an upcoming conditional branch or RET instruction. Once a decoder consumes a TNT packet with N valid payload bits, these bits should be applied to (and hence provide the destination for) the next N conditional branches or RETs									

### 36.4.2.2 Target IP (TIP) Packet

Table 36-17. IP Packet Definition

Name	Target IP (TIP) Packet								
Packet Format		7	6	5	4	3	2	1	0
	0	IPBytes			0	1	1	0	1
	1	TargetIP[7:0]							
	2	TargetIP[15:8]							
	3	TargetIP[23:16]							
	4	TargetIP[31:24]							
	5	TargetIP[39:32]							
	6	TargetIP[47:40]							
	7	TargetIP[55:48]							
	8	TargetIP[63:56]							
Dependencies	PacketEn	Generation Scenario	Indirect branch (including un-compressed RET), far branch, interrupt, exception, INIT, SIPI, (VM exit, VM entry), <sup>1</sup> TSX abort, (EENTER, EEXIT, ERESUME, AEX) <sup>2</sup> .						
Description	Provides the target for some control flow transfers								
Application	Anytime a TIP is encountered, it indicates that control was transferred to the IP provided in the payload.								
	The source of this control flow change, and hence the IP or instruction to which it binds, depends on the packets that precede the TIP. If a TIP is encountered and all preceding packets have already been bound, then the TIP will apply to the upcoming indirect branch, far branch, or VMRESUME. However, if there was a preceding FUP that remains unbound, it will bind to the TIP. Here, the TIP provides the target of an asynchronous event or TSX abort that occurred at the IP given in the FUP payload. Note that there may be other packets, in addition to the FUP, which will bind to the TIP packet. See the packet application descriptions for other packets for details.								

**NOTES:**

1. If IA32\_VMX\_MISC[bit 14] reports 1.
2. In a debug enclave.

#### IP Compression

The IP payload in a TIP, FUP, TIP.PGE, or TIP.PGD packet can vary in size, based on the mode of execution, and the use of IP compression. IP compression is an optional compression technique the processor may choose to employ to reduce bandwidth. With IP compression, the IP to be represented in the payload is compared with the last IP sent out, via any of FUP, TIP, TIP.PGE, or TIP.PGD. If that previous IP had the same upper (most significant) address bytes, those matching bytes may be suppressed in the current packet. The processor maintains an internal state of the “Last IP” that was encoded in trace packets, thus the decoder will need to keep track of the “Last IP” state in software, to match fidelity with packets generated by hardware. “Last IP” is initialized to zero, hence if the first IP in the trace may be compressed if the upper bytes are zeroes.

The “IPBytes” field of the IP packets (FUP, TIP, TIP.PGE, TIP.PGD) serves to indicate how many bytes of payload are provided, and how the decoder should fill in any suppressed bytes. The algorithm for reconstructing the IP for a TIP/FUP packet is shown in the table below.

**Table 36-18. FUP/TIP IP Reconstruction**

IPBytes	Uncompressed IP Value							
	63:56	55:48	47:40	39:32	31:24	23:16	15:8	7:0
000b	None, IP is out of context							
001b	Last IP[63:16]						IP Payload[15:0]	
010b	Last IP[63:32]				IP Payload[31:0]			
011b	IP Payload[47] extended		IP Payload[47:0]					
100b	Last IP [63:48]		IP Payload[47:0]					
101b	Reserved							
110b	IP Payload[63:0]							
111b	Reserved							

The processor-internal Last IP state is guaranteed to be reset to zero when a PSB is sent out. This means that the IP that follows the PSB with either be un-compressed (011b or 110b, see Table 36-18), or compressed against zero.

At times, “IPbytes” will have a value of 0. As shown above, this does not mean that the IP payload matches the full address of the last IP, but rather that the IP for this packet was suppressed. This is used for cases where the IP that applies to the packet is out of context. An example is the TIP.PGD sent on a SYSCALL, when tracing only USR code. In that case, no TargetIP will be included in the packet, since that would expose an instruction point at CPL = 0. When the IP payload is suppressed in this manner, Last IP is not cleared, and instead refers to the last IP packet with a non-zero IPBytes field.

On processors that support a maximum linear address size of 32 bits, IP payloads may never exceed 32 bits (IPBytes <= 010b).

### Indirect Transfer Compression for Returns (RET)

In addition to IP compression, TIP packets for near return (RET) instructions can also be compressed. If the RET target matches the next IP of the corresponding CALL, then the TIP packet is unneeded, since the decoder can deduce the target IP by maintaining a CALL/RET stack of its own.

A CALL/RET stack can be maintained by the decoder by doing the following:

1. Allocate space to store 64 RET targets.
2. For near CALLs, push the Next IP onto the stack. Once the stack is full, new CALLs will force the oldest entry off the end of the stack, such that only the youngest 64 entries are stored. Note that this excludes zero-length CALLs, which are direct near CALLs with displacement zero (to the next IP). These CALLs typically don't have matching RETs.
3. For near RETs, pop the top (youngest) entry off the stack. This will be the target of the RET.

In cases where the RET is compressed, the target is guaranteed to match the value produced in 2) above. If the target is not compressed, a TIP packet will be generated with the RET target, which may differ from 2).

The hardware ensure that packets read by the decoder will always have seen the CALL that corresponds to any compressed RET. The processor will never compress a RET across a PSB, a buffer overflow, or scenario where PacketEn=0. This means that a RET whose corresponding CALL executed while PacketEn=0, or before the last PSB, etc., will not be compressed.

If the CALL/RET stack is manipulated or corrupted by software, and thereby causes a RET to transfer control to a target that is inconsistent with the CALL/RET stack, then the RET will not be compressed, and will produce a TIP packet. This can happen, for example, if software executes a PUSH instruction to push a target onto the stack, and a later RET uses this target.

When a RET is compressed, a Taken indication is added to the TNT buffer. Because it sends no TIP packet, it also does not update the internal Last IP value, and thus the decoder should treat it the same way. If the RET is not compressed, it will generate a TIP packet (just like when RET compression is disabled, via IA32\_RTIT\_CTL.DisRETC). For processors that employ deferred TIPs (Section 36.4.2.3), an uncompressed RET will not be deferred, and hence will force out any accumulated TNTs or TIPs. This serves to avoid ambiguity, and make

clear to the decoder whether the near RET was compressed, and hence a bit in the in-progress TNT should be consumed, or uncompressed, in which case there will be no in-progress TNT and thus a TIP should be consumed.

Note that in the unlikely case that a RET executes in a different execution mode than the associated CALL, the decoder will need to model the same behavior with its CALL stack. For instance, if a CALL executes in 64-bit mode, a 64-bit IP value will be pushed onto the software stack. If the corresponding RET executes in 32-bit mode, then only the lower 32 target bits will be popped off of the stack, which may mean that the RET does not go to the CALL's Next IP. This is architecturally correct behavior, and this RET could be compressed, thus the decoder should match this behavior

### 36.4.2.3 Deferred TIPs

The processor may opt to defer sending out the TNT when TIPs are generated. Thus, rather than sending a partial TNT followed by a TIP, both packets will be deferred while the TNT accumulates more Jcc/RET results. Any number of TIP packets may be accumulated this way, such that only once the TNT is filled, or once another packet (e.g., FUP) is generated, the TNT will be sent, followed by all the deferred TIP packets, and finally terminated by the other packet(s) that forced out the TNT and TIP packets. Generation of many other packets (see list below) will force out the TNT and any accumulated TIP packets. This is an optional optimization in hardware to reduce the bandwidth consumption, and hence the performance impact, incurred by tracing.

**Table 36-19. TNT Examples with Deferred TIPs**

Code Flow	Packets, Non-Deferred TIPs	Packets, Deferred TIPs
0x1000 cmp %rcx, 0 0x1004 jnz Foo // not-taken 0x1008 jmp %rdx	TNT(0b0), TIP(0x1308)	
0x1308 cmp %rcx, 1 0x130c jnz Bar // not-taken 0x1310 cmp %rcx, 2 0x1314 jnz Baz // taken 0x1500 cmp %eax, 7 0x1504 jg Exit // not-taken 0x1508 jmp %r15	TNT(0b010), TIP(0x1100)	
0x1100 cmp %rbx, 1 0x1104 jg Start // not-taken 0x1108 add %rcx, %eax 0x110c ... // <b>an asynchronous interrupt arrives</b> INThandler: 0xcc00 pop %rdx	TNT(0b0), FUP(0x110c), TIP(0xcc00)	TNT(0b00100), TIP(0x1308), TIP(0x1100), FUP(0x110c), TIP(0xcc00)



### 36.4.2.4 Packet Generation Enable (TIP.PGE)

**Table 36-20. TIP.PGE Packet Definition**

Name	Target IP - Packet Generation Enable (TIP.PGE)								
Packet Format		7	6	5	4	3	2	1	0
	0	IPBytes			1	0	0	0	1
	1	TargetIP[7:0]							
	2	TargetIP[15:8]							
	3	TargetIP[23:16]							
	4	TargetIP[31:24]							
	5	TargetIP[39:32]							
	6	TargetIP[47:40]							
	7	TargetIP[55:48]							
	8	TargetIP[63:56]							
Dependencies	PacketEn transitions to 1			Generation Scenario	Any branch instruction, control flow transfer, or MOV CR3 that sets PacketEn, a WRMSR that enables packet generation and sets PacketEn				
Description	<p>Indicates that PacketEn has transitioned to 1. It provides the IP at which the tracing begins. This can occur due to any of the enables that comprise PacketEn transitioning from 0 to 1, as long as all the others are asserted. Examples:</p> <ul style="list-style-type: none"> <li>▪ TriggerEn: This is set on software write to set IA32_RTIT_CTL.TraceEn as long as the Stopped and Error bits in IA32_RTIT_STATUS are clear. The IP payload will be the Next IP of the WRMSR.</li> <li>▪ FilterEn: This is set when software jumps into the tracing region. This region is defined by enabling IP filtering in IA32_RTIT_CTL.ADDRn_CFG, and defining the range in IA32_RTIT_ADDRn_[AB], see. Section 36.2.4.3. The IP payload will be the target of the branch.</li> <li>▪ ContextEn: This is set on a CPL change, a CR3 write or any other means of changing ContextEn. The IP payload will be the Next IP of the instruction that changes context if it is not a branch, otherwise it will be the target of the branch.</li> </ul>								
Application	TIP.PGE packets bind to the instruction at the IP given in the payload.								

### 36.4.2.5 Packet Generation Disable (TIP.PGD)

**Table 36-21. TIP.PGD Packet Definition**

Name	Target IP - Packet Generation Disable (TIP.PGD)								
Packet Format		7	6	5	4	3	2	1	0
	0	IPBytes			0	0	0	0	1
	1	TargetIP[7:0]							
	2	TargetIP[15:8]							
	3	TargetIP[23:16]							
	4	TargetIP[31:24]							
	5	TargetIP[39:32]							
	6	TargetIP[47:40]							
	7	TargetIP[55:48]							
	8	TargetIP[63:56]							
Dependencies	PacketEn transitions to 0	Generation Scenario	Any branch instruction, control flow transfer, or MOV CR3 that clears PacketEn, a WRMSR that disables packet generation and clears PacketEn						
Description	<p>Indicates that PacketEn has transitioned to 0. It will include the IP at which the tracing ends, unless ContextEn = 0 or TraceEn=0 at the conclusion of the instruction or event that cleared PacketEn.</p> <p>PacketEn can be cleared due to any of the enables that comprise PacketEn transitioning from 1 to 0. Examples:</p> <ul style="list-style-type: none"> <li>▪ TriggerEn: This is cleared on software write to clear IA32_RTIT_CTL.TraceEn, or when IA32_RTIT_STATUS.Stopped is set, or on operational error. The IP payload will be suppressed in this case, and the “IPBytes” field will have the value 0.</li> <li>▪ FilterEn: This is set when software jumps out of the tracing region. This region is defined by enabling IP filtering in IA32_RTIT_CTL.ADDRn_CFG, and defining the range in IA32_RTIT_ADDRn_[AB], see. Section 36.2.4.3. The IP payload will depend on the type of the branch. For conditional branches, the payload is suppressed (IPBytes = 0), and in this case the destination can be inferred from the disassembly. For any other type of branch, the IP payload will be the target of the branch.</li> <li>▪ ContextEn: This can happen on a CPL change, a CR3 write or any other means of changing ContextEn. See Section 36.2.4.3 for details. In this case, when ContextEn is cleared, there will be no IP payload. The “IPBytes” field will have value 0.</li> </ul> <p>Note that, in cases where a branch that would normally produce a TIP packet (i.e., far transfer, indirect branch, interrupt, etc) or TNT update (conditional branch or compressed RT) causes PacketEn to transition from 1 to 0, the TIP or TNI bit will be replaced with TIP.PGD. The payload of the TIP.PGD will be the target of the branch, unless the result of the instruction causes TraceEn or ContextEn to be cleared (ie, SYSCALL when IA32_RTIT_CTL.OS=0, In the case where a conditional branch clears FilterEn and hence PacketEn, there will be no TNT bit for this branch, replaced instead by the TIP.PGD.</p>								
Application	<p>TIP.PGD can be produced by any branch instructions, as well as some non-branch instructions, that clear PacketEn. When produced by a branch, it replaces any TIP or TNT update that the branch would normally produce. In cases where there is an unbound FUP preceding the TIP.PGD, then the TIP.PGD is part of compound operation (i.e., asynchronous event or TSX abort) which cleared PacketEn. For most such cases, the TIP.PGD is simply replacing a TIP, and should be treated the same way. The TIP.PGD may or may not have an IP payload, depending on whether the operation cleared ContextEn.</p> <p>If there is not an associated FUP, the binding will depend on whether there is an IP payload. If there is an IP payload, then the TIP.PGD should be applied to either the next direct branch whose target matches the TIP.PGD payload, or the next branch that would normally generate a TIP or TNT packet. If there is no IP payload, then the TIP.PGD should apply to the next branch or MOV CR3 instruction.</p>								

### 36.4.2.6 Flow Update (FUP) Packet

Table 36-22. FUP Packet Definition

Name	Flow Update (FUP) Packet								
Packet Format		7	6	5	4	3	2	1	0
	0	IPBytes			1	1	1	0	1
	1	IP[7:0]							
	2	IP[15:8]							
	3	IP[23:16]							
	4	IP[31:24]							
	5	IP[39:32]							
	6	IP[47:40]							
	7	IP[55:48]							
	8	IP[63:56]							
Dependencies	TriggerEn & ContextEn. (Typically depends on BranchEn and FilterEn as well, see Section 36.2.4 for details.)		Generation Scenario	Asynchronous Events (interrupts, exceptions, INIT, SIPI, SMI, VM exit <sup>1</sup> , #MC), XBEGIN, XEND, XABORT, XACQUIRE, XRELEASE, (EENTRY, EEXIT, ERESUME, EEE, AEX,) <sup>2</sup> , INT 0, INT 3, INT n, a WRMSR that disables packet generation.					
Description	Provides the source address for asynchronous events, and some other instructions. Is never sent alone, always sent with an associated TIP or MODE packet, and potentially others.								
Application	FUP packets provide the IP to which they bind. However, they are never standalone, but are coupled with other packets. In TSX cases, the FUP is immediately preceded by a MODE.TSX, which binds to the same IP. A TIP will follow only in the case of TSX aborts, see Section 36.4.2.8 for details. Otherwise, FUPs are part of compound packet events (see Section 36.4.1). In these compound cases, the FUP provides the source IP for an instruction or event, while a following TIP (or TIP.PGD) uop will provide any destination IP. Other packets may be included in the compound event between the FUP and TIP.								

**NOTES:**

1. If IA32\_VMX\_MISC[bit 14] reports 1.
2. If Intel Software Guard Extensions is supported.

## FUP IP Payload

Flow Update Packet gives the source address of an instruction when it is needed. In general, branch instructions do not need a FUP, because the source address is clear from the disassembly. For asynchronous events, however, the source address cannot be inferred from the source, and hence a FUP will be sent. Table 36-23 illustrates cases where FUPs are sent, and which IP can be expected in those cases.

**Table 36-23. FUP Cases and IP Payload**

Event	Flow Update IP	Comment
External Interrupt, NMI/SMI, Traps, Machine Check (trap-like), INIT/SIPI	Address of next instruction (Next IP) that would have been executed	Functionally, this matches the LBR FROM field value and also the EIP value which is saved onto the stack.
Exceptions/Faults, Machine check (fault-like)	Address of the instruction which took the exception/fault (Current IP)	This matches the similar functionality of LBR FROM field value and also the EIP value which is saved onto the stack.
Software Interrupt	Address of the software interrupt instruction (Current IP)	This matches the similar functionality of LBR FROM field value, but does not match the EIP value which is saved onto the stack (Next Linear Instruction Pointer - NLIP).
EENTER, EEXIT, ERESUME, Enclave Exiting Event (EEE), AEX <sup>1</sup>	Current IP of the instruction	This matches the LBR FROM field value and also the EIP value which is saved onto the stack.
XACQUIRE	Address of the X* instruction	
XRELEASE, XBEGIN, XEND, XABORT, other transactional abort	Current IP	
#SMI	IP that is saved into SMRAM	
WRMSR that clears TraceEn	Current IP	

### NOTES:

1. Information on EENTER, EEXIT, ERESUME, EEE, Asynchronous Enclave eXit (AEX) can be found in *Intel® Software Guard Extensions Programming Reference*.

On a canonical fault due to sequentially fetching an instruction in non-canonical space (as opposed to jumping to non-canonical space), the IP of the fault (and thus the payload of the FUP) will be a non-canonical address. This is consistent with what is pushed on the stack for such faulting cases.

If there are post-commit task switch faults, the IP value of the FUP will be the original IP when the task switch started. This is the same value as would be seen in the LBR\_FROM field. But it is a different value as is saved on the stack or VMCS.

## 36.4.2.7 Paging Information (PIP) Packet

Table 36-24. PIP Packet Definition

Name	Paging Information (PIP) Packet									
Packet Format		7	6	5	4	3	2	1	0	
	0	0	0	0	0	0	0	1	0	
	1	0	1	0	0	0	0	1	1	
	2	CR3[11:5] or 0							RSVD/NR	
	3	CR3[19:12]								
	4	CR3[27:20]								
	5	CR3[35:28]								
	6	CR3[43:36]								
	7	CR3[51:44]								
	Dependencies	TriggerEn && ContextEn && IA32_RTIT_CTL.OS			Generation Scenario		MOV CR3, Task switch, INIT, SIPI, PSB+; If IA32_VMX_MISC[bit 14] reports 1: VM exit, VM entry			
Description	<p>The CR3 payload shown includes only the address portion of the CR3 value. For PAE paging, CR3[11:5] are thus included. For other page modes (32-bit and IA-32e paging), these bits are 0.</p> <p>This packet holds the CR3 address value. It will be generated on operations that modify CR3:</p> <ul style="list-style-type: none"> <li>▪ MOV CR3 operation</li> <li>▪ Task Switch</li> <li>▪ INIT and SIPI</li> <li>▪ VM exit and VM entry, if appropriate controls in the VMCS are clear (see Section 36.5.1)</li> </ul> <p>PIPs are not generated, despite changes to CR3, on SMI and RSM. This is due to the special behavior on these operations, see Section for details. Note that, for some cases of task switch where CR3 is not modified, no PIP will be produced.</p> <p>The purpose of the PIP is to indicate to the decoder which application is running, so that it can apply the proper binaries to the linear addresses that are being traced.</p> <p>The PIP packet contains the new CR3 value when CR3 is written.</p> <p>PIPs generated by VM entries set the NR bit. PIPs generated in VMX non-root operation set the NR bit if the “conceal VMX non-root operation from Intel PT” VM-execution control is 0. All other PIPs clear the NR bit.</p>									
Application	<p>The purpose of the PIP packet is to help the decoder uniquely identify what software is running at any given time. When a PIP is encountered, a decoder should do the following:</p> <ol style="list-style-type: none"> <li>1) If there was a prior unbound FUP (that is, a FUP not preceded by a packet such as MODE.TSX that consumes it, and it hence pairs with a TIP that has not yet been seen), then this PIP is part of a compound packet event (Section 36.4.1). Find the ending TIP and apply the new CR3/NR values to the TIP payload IP.</li> <li>2) Otherwise, look for the next MOV CR3, far branch, or VMRESUME/VMLAUNCH in the disassembly, and apply the new CR3 to the next (or target) IP.</li> </ol> <p>For examples of the packets generated by these flows, see Section 36.7.</p>									

### 36.4.2.8 MODE Packets

MODE packets keep the decoder informed of various processor modes about which it needs to know in order to properly manage the packet output, or to properly disassemble the associated binaries. MODE packets include a header and a mode byte, as shown below.

**Table 36-25. General Form of MODE Packets**

	7	6	5	4	3	2	1	0
0	1	0	0	1	1	0	0	1
1	Leaf ID			Mode				

The MODE Leaf ID indicates which set of mode bits are held in the lower bits.

#### MODE.Exec Packet

**Table 36-26. MODE.Exec Packet Definition**

Name	MODE.Exec Packet																													
Packet Format	<table border="1"> <tr> <td></td> <td>7</td> <td>6</td> <td>5</td> <td>4</td> <td>3</td> <td>2</td> <td>1</td> <td>0</td> </tr> <tr> <td>0</td> <td>1</td> <td>0</td> <td>0</td> <td>1</td> <td>1</td> <td>0</td> <td>0</td> <td>1</td> </tr> <tr> <td>1</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>CS.D</td> <td>(CS.L &amp; LMA)</td> </tr> </table>				7	6	5	4	3	2	1	0	0	1	0	0	1	1	0	0	1	1	0	0	0	0	0	0	CS.D	(CS.L & LMA)
	7	6	5	4	3	2	1	0																						
0	1	0	0	1	1	0	0	1																						
1	0	0	0	0	0	0	CS.D	(CS.L & LMA)																						
Dependencies	PacketEn	Generation Scenario	Far branch, interrupt, exception, (VM exit, VM entry), <sup>1</sup> if the mode changes. PSB+, and any scenario that can generate a TIP.PGE, such that the mode may have changed since the last MODE.Exec.																											
Description	<p>Indicates whether software is in 16, 32, or 64-bit mode, by providing the CS.D and (CS.L &amp; IA32_EFER.LMA) values. Essential for the decoder to properly disassemble the associated binary.</p> <table border="1"> <thead> <tr> <th>CS.D</th> <th>(CS.L &amp; IA32_EFER.LMA)</th> <th>Addressing Mode</th> </tr> </thead> <tbody> <tr> <td>1</td> <td>1</td> <td>N/A</td> </tr> <tr> <td>0</td> <td>1</td> <td>64-bit mode</td> </tr> <tr> <td>1</td> <td>0</td> <td>32-bit mode</td> </tr> <tr> <td>0</td> <td>0</td> <td>16-bit mode</td> </tr> </tbody> </table> <p>MODE.Exec is sent at the time of a mode change, if PacketEn=1 at the time, or when tracing resumes, if necessary. In the former case, the MODE.Exec packet is generated along with other packets that result from the far transfer operation that changes the mode. In cases where the mode changes while PacketEn=0, the processor will send out a MODE.Exec along with the TIP.PGE when tracing resumes. The processor may opt to suppress the MODE.Exec when tracing resumes if the mode matches that from the last MODE.Exec packet, if there was no PSB in between.</p>			CS.D	(CS.L & IA32_EFER.LMA)	Addressing Mode	1	1	N/A	0	1	64-bit mode	1	0	32-bit mode	0	0	16-bit mode												
CS.D	(CS.L & IA32_EFER.LMA)	Addressing Mode																												
1	1	N/A																												
0	1	64-bit mode																												
1	0	32-bit mode																												
0	0	16-bit mode																												
Application	MODE.Exec always immediately precedes a TIP or TIP.PGE. The mode change applies to the IP address in the payload of the next TIP or TIP.PGE.																													

**NOTES:**

1. If IA32\_VMX\_MISC[bit 14] reports 1.

## MODE.TSX Packet

Table 36-27. MODE.TSX Packet Definition

Name	MODE.TSX Packet																																			
Packet Format	<table border="1"> <tr> <td></td> <td>7</td> <td>6</td> <td>5</td> <td>4</td> <td>3</td> <td>2</td> <td>1</td> <td>0</td> </tr> <tr> <td>0</td> <td>1</td> <td>0</td> <td>0</td> <td>1</td> <td>1</td> <td>0</td> <td>0</td> <td>1</td> </tr> <tr> <td>1</td> <td>0</td> <td>0</td> <td>1</td> <td>0</td> <td>0</td> <td>0</td> <td>TXAbort</td> <td>InTX</td> </tr> </table>										7	6	5	4	3	2	1	0	0	1	0	0	1	1	0	0	1	1	0	0	1	0	0	0	TXAbort	InTX
		7	6	5	4	3	2	1	0																											
	0	1	0	0	1	1	0	0	1																											
1	0	0	1	0	0	0	TXAbort	InTX																												
Dependencies	TriggerEn and ContextEn	Generation Scenario	XBEGIN, XEND, XABORT, XACQUIRE, XRELEASE, if INTX changes, Asynchronous TSX Abort, PSB+																																	
Description	Indicates when a TSX transaction (either HLE or RTM) begins, commits, or aborts. Instructions executed transactionally will be “rolled back” if the transaction is aborted.																																			
	<table border="1"> <thead> <tr> <th>TXAbort</th> <th>InTX</th> <th>Implication</th> </tr> </thead> <tbody> <tr> <td>1</td> <td>1</td> <td>N/A</td> </tr> <tr> <td>0</td> <td>1</td> <td>Transaction begins, or executing transactionally</td> </tr> <tr> <td>1</td> <td>0</td> <td>Transaction aborted</td> </tr> <tr> <td>0</td> <td>0</td> <td>Transaction committed, or not executing transactionally</td> </tr> </tbody> </table>									TXAbort	InTX	Implication	1	1	N/A	0	1	Transaction begins, or executing transactionally	1	0	Transaction aborted	0	0	Transaction committed, or not executing transactionally												
	TXAbort	InTX	Implication																																	
	1	1	N/A																																	
	0	1	Transaction begins, or executing transactionally																																	
	1	0	Transaction aborted																																	
0	0	Transaction committed, or not executing transactionally																																		
Application	If PacketEn=1, MODE.TSX always immediately precedes a FUP. If the TXAbort bit is zero, then the mode change applies to the IP address in the payload of the FUP. If TXAbort=1, then the FUP will be followed by a TIP, and the mode change will apply to the IP address in the payload of the TIP. MODE.TSX packets may be generated when PacketEn=0, due to FilterEn=0. In this case, only the last MODE.TSX generated before TIP.PGE need be applied.																																			

### 36.4.2.9 TraceStop Packet

**Table 36-28. TraceStop Packet Definition**

Name	TraceStop Packet																													
Packet Format	<table border="1"> <tr> <td></td> <td>7</td> <td>6</td> <td>5</td> <td>4</td> <td>3</td> <td>2</td> <td>1</td> <td>0</td> </tr> <tr> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>1</td> <td>0</td> </tr> <tr> <td>1</td> <td>1</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>1</td> <td>1</td> </tr> </table>				7	6	5	4	3	2	1	0	0	0	0	0	0	0	0	1	0	1	1	0	0	0	0	0	1	1
	7	6	5	4	3	2	1	0																						
0	0	0	0	0	0	0	1	0																						
1	1	0	0	0	0	0	1	1																						
Dependencies	TriggerEn && ContextEn	Generation Scenario	Taken branch with target in TraceStop IP region, MOV CR3 in TraceStop IP region, or WRMSR that sets TraceEn in TraceStop IP region.																											
Description	<p>Indicates when software has entered a user-configured TraceStop region. When the IP matches a TraceStop range while ContextEn and TriggerEn are set, a TraceStop action occurs. This disables tracing by setting IA32_RTIT_STATUS.Stopped, thereby clearing TriggerEn, and causes a TraceStop packet to be generated.</p> <p>The TraceStop action also forces FilterEn to 0. Note that TraceStop may not force a flush of internally buffered packets, and thus trace packet generation should still be manually disabled by clearing IA32_RTIT_CTL.TraceEn before examining output. See Section 36.2.4.3 for more details.</p>																													
Application	<p>If TraceStop follows a TIP.PGD (before the next TIP.PGE), then it was triggered either by the instruction that cleared PacketEn, or it was triggered by some later instruction that executed while FilterEn=0. In either case, the TraceStop can be applied at the IP of the TIP.PGD (if any).</p> <p>If TraceStop follows a TIP.PGE (before the next TIP.PGD), it should be applied at the last known IP.</p>																													

### 36.4.2.10 Core:Bus Ratio (CBR) Packet

**Table 36-29. CBR Packet Definition**

Name	Core:Bus Ratio (CBR) Packet																																															
Packet Format	<table border="1"> <tr> <td></td> <td>7</td> <td>6</td> <td>5</td> <td>4</td> <td>3</td> <td>2</td> <td>1</td> <td>0</td> </tr> <tr> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>1</td> <td>0</td> </tr> <tr> <td>1</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>1</td> <td>1</td> </tr> <tr> <td>2</td> <td colspan="8">Core:Bus Ratio</td> </tr> <tr> <td>3</td> <td colspan="8">Reserved</td> </tr> </table>				7	6	5	4	3	2	1	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	1	1	2	Core:Bus Ratio								3	Reserved							
	7	6	5	4	3	2	1	0																																								
0	0	0	0	0	0	0	1	0																																								
1	0	0	0	0	0	0	1	1																																								
2	Core:Bus Ratio																																															
3	Reserved																																															
Dependencies	TriggerEn	Generation Scenario	After any frequency change, on C-state wake up, PSB+, and after enabling trace packet generation.																																													
Description	Indicates the core:bus ratio of the processor core. Useful for correlating wall-clock time and cycle time.																																															
Application	All packets following the CBR represent instructions that executed with the new core:bus ratio, while all preceding packets (aside from timing packets) represent instructions that executed with the prior ratio. There is not a precise IP provided, to which to bind the CBR packet.																																															



## 36.4.2.11 Timestamp Counter (TSC) Packet

Table 36-30. TSC Packet Definition

Name	Timestamp Counter (TSC) Packet																																																																																			
Packet Format	<table border="1"> <thead> <tr> <th></th> <th>7</th> <th>6</th> <th>5</th> <th>4</th> <th>3</th> <th>2</th> <th>1</th> <th>0</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>1</td> <td>1</td> <td>0</td> <td>0</td> <td>1</td> </tr> <tr> <td>1</td> <td colspan="8">SW TSC[7:0]</td> </tr> <tr> <td>2</td> <td colspan="8">SW TSC[15:8]</td> </tr> <tr> <td>3</td> <td colspan="8">SW TSC[23:16]</td> </tr> <tr> <td>4</td> <td colspan="8">SW TSC[31:24]</td> </tr> <tr> <td>5</td> <td colspan="8">SW TSC[39:32]</td> </tr> <tr> <td>6</td> <td colspan="8">SW TSC[47:40]</td> </tr> <tr> <td>7</td> <td colspan="8">SW TSC[55:48]</td> </tr> </tbody> </table>				7	6	5	4	3	2	1	0	0	0	0	0	1	1	0	0	1	1	SW TSC[7:0]								2	SW TSC[15:8]								3	SW TSC[23:16]								4	SW TSC[31:24]								5	SW TSC[39:32]								6	SW TSC[47:40]								7	SW TSC[55:48]							
	7	6	5	4	3	2	1	0																																																																												
0	0	0	0	1	1	0	0	1																																																																												
1	SW TSC[7:0]																																																																																			
2	SW TSC[15:8]																																																																																			
3	SW TSC[23:16]																																																																																			
4	SW TSC[31:24]																																																																																			
5	SW TSC[39:32]																																																																																			
6	SW TSC[47:40]																																																																																			
7	SW TSC[55:48]																																																																																			
Dependencies	IA32_RTIT_CTL.TSCEn && TriggerEn	Generation Scenario	Sent after any event that causes the processor clocks or Intel PT timing packets (such as MTC or CYC) to stop, This may include P-state changes, wake from C-state, or clock modulation. Also on transition of TraceEn from 0 to 1.																																																																																	
Description	When enabled by software, a TSC packet provides the lower 7 bytes of the current TSC value, as returned by the RDTSC instruction. This may be useful for tracking wall-clock time, and synchronizing the packets in the log with other timestamped logs.																																																																																			
Application	TSC packet provides a wall-clock proxy of the event which generated it (packet generation enable, sleep state wake, etc). In all cases, TSC does not precisely indicate the time of any control flow packets; however, all preceding packets represent instructions that executed before the indicated TSC time, and all subsequent packets represent instructions that executed after it. There is not a precise IP to which to bind the TSC packet.																																																																																			

### 36.4.2.12 Mini Time Counter (MTC) Packet

**Table 36-31. MTC Packet Definition**

Name	Mini time Counter (MTC) Packet																													
Packet Format	<table border="1" style="width: 100%; text-align: center;"> <tr> <td style="width: 10%;"></td> <td style="width: 10%;">7</td> <td style="width: 10%;">6</td> <td style="width: 10%;">5</td> <td style="width: 10%;">4</td> <td style="width: 10%;">3</td> <td style="width: 10%;">2</td> <td style="width: 10%;">1</td> <td style="width: 10%;">0</td> </tr> <tr> <td>0</td> <td>0</td> <td>1</td> <td>0</td> <td>1</td> <td>1</td> <td>0</td> <td>0</td> <td>1</td> </tr> <tr> <td>1</td> <td colspan="8">CTC[N+7:N]</td> </tr> </table>				7	6	5	4	3	2	1	0	0	0	1	0	1	1	0	0	1	1	CTC[N+7:N]							
	7	6	5	4	3	2	1	0																						
0	0	1	0	1	1	0	0	1																						
1	CTC[N+7:N]																													
Dependencies	IA32_RTIT_CTL.MTCEn && TriggerEn	Generation Scenario	Periodic, based on the core crystal clock, or Always Running Timer (ART).																											
Description	<p>When enabled by software, an MTC packet provides a periodic indication of wall-clock time. The 8-bit CTC (Common Timestamp Copy) payload value is set to <math>(ART \gg N) \&amp; 0xFF</math>. The frequency of the ART is related to the Maximum Non-Turbo frequency, and the ratio can be determined from CPUID leaf 15H, as described in Section 36.8.3. Software can select the threshold N, which determines the MTC frequency by setting the IA32_RTIT_CTL.MTCFreq field (see Section 36.2.7.2) to a supported value using the lookup enumerated by CPUID (see Section 36.3.1). See Section 36.8.3 for details on how to use the MTC payload to track TSC time.</p> <p>MTC provides 8 bits from the ART, starting with the bit selected by MTCFreq to dictate the frequency of the packet. Whenever that 8-bit range being watched changes, an MTC packet will be sent out with the new value of that 8-bit range. This allows the decoder to keep track of how much wall-clock time has elapsed since the last TSC packet was sent, by keeping track of how many MTC packets were sent and what their value was. The decoder can infer the truncated bits, CTC[N-1:0], are 0 at the time of the MTC packet.</p> <p>There are cases in which MTC packet can be dropped, due to overflow or other micro-architectural conditions. The decoder should be able to recover from such cases by checking the 8-bit payload of the next MTC packet, to determine how many MTC packets were dropped. It is not expected that &gt;256 consecutive MTC packets should ever be dropped.</p>																													
Application	MTC does not precisely indicate the time of any other packet, nor does it bind to any IP. However, all preceding packets represent instructions or events that executed before the indicated ART time, and all subsequent packets represent instructions that executed after, or at the same time as, the ART time.																													

## 36.4.2.13 TSC/MTC Alignment (TMA) Packet

Table 36-32. TMA Packet Definition

Name	TSC/MTC Alignment (TMA) Packet																																																																										
Packet Format	<table border="1"> <thead> <tr> <th></th> <th>7</th> <th>6</th> <th>5</th> <th>4</th> <th>3</th> <th>2</th> <th>1</th> <th>0</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>1</td> <td>0</td> </tr> <tr> <td>1</td> <td>0</td> <td>1</td> <td>1</td> <td>1</td> <td>0</td> <td>0</td> <td>1</td> <td>1</td> </tr> <tr> <td>2</td> <td colspan="8">CTC[7:0]</td> </tr> <tr> <td>3</td> <td colspan="8">CTC[15:8]</td> </tr> <tr> <td>4</td> <td colspan="7">Reserved</td> <td>0</td> </tr> <tr> <td>5</td> <td colspan="8">FastCounter[7:0]</td> </tr> <tr> <td>6</td> <td colspan="7">Reserved</td> <td>FC[8]</td> </tr> </tbody> </table>				7	6	5	4	3	2	1	0	0	0	0	0	0	0	0	1	0	1	0	1	1	1	0	0	1	1	2	CTC[7:0]								3	CTC[15:8]								4	Reserved							0	5	FastCounter[7:0]								6	Reserved							FC[8]
	7	6	5	4	3	2	1	0																																																																			
0	0	0	0	0	0	0	1	0																																																																			
1	0	1	1	1	0	0	1	1																																																																			
2	CTC[7:0]																																																																										
3	CTC[15:8]																																																																										
4	Reserved							0																																																																			
5	FastCounter[7:0]																																																																										
6	Reserved							FC[8]																																																																			
Dependencies	IA32_RTIT_CTL.MTCEn && IA32_RTIT_CTL.TSCEn && TriggerEn	Generation Scenario	Sent with any TSC packet.																																																																								
Description	The TMA packet serves to provide the information needed to allow the decoder to correlate MTC packets with TSC packets. With this packet, when a MTC packet is encountered, the decoder can determine how many timestamp counter ticks have passed since the last TSC or MTC packet. See Section 36.8.3.2 for details on how to make this calculation.																																																																										
Application	TMA is always sent immediately following a TSC packet, and the payload values are consistent with the TSC payload value. Thus the application of TMA matches that of TSC.																																																																										

### 36.4.2.14 Cycle Count Packet (CYC) Packet

**Table 36-33. Cycle Count Packet Definition**

Name	Cycle Count (CYC) Packet																																															
Packet Format	<table border="1" style="width: 100%; border-collapse: collapse;"> <tr> <td style="width: 10%;"></td> <td style="width: 10%; text-align: center;">7</td> <td style="width: 10%; text-align: center;">6</td> <td style="width: 10%; text-align: center;">5</td> <td style="width: 10%; text-align: center;">4</td> <td style="width: 10%; text-align: center;">3</td> <td style="width: 10%; text-align: center;">2</td> <td style="width: 10%; text-align: center;">1</td> <td style="width: 10%; text-align: center;">0</td> </tr> <tr> <td style="text-align: center;">0</td> <td colspan="5">Cycle Counter[4:0]</td> <td style="text-align: center;">Exp</td> <td style="text-align: center;">1</td> <td style="text-align: center;">1</td> </tr> <tr> <td style="text-align: center;">1</td> <td colspan="7">Cycle Counter[11:5]</td> <td style="text-align: center;">Exp</td> </tr> <tr> <td style="text-align: center;">2</td> <td colspan="7">Cycle Counter[18:12]</td> <td style="text-align: center;">Exp</td> </tr> <tr> <td style="text-align: center;">...</td> <td colspan="8">... (if Exp = 1 in the previous byte)</td> </tr> </table>				7	6	5	4	3	2	1	0	0	Cycle Counter[4:0]					Exp	1	1	1	Cycle Counter[11:5]							Exp	2	Cycle Counter[18:12]							Exp	...	... (if Exp = 1 in the previous byte)							
	7	6	5	4	3	2	1	0																																								
0	Cycle Counter[4:0]					Exp	1	1																																								
1	Cycle Counter[11:5]							Exp																																								
2	Cycle Counter[18:12]							Exp																																								
...	... (if Exp = 1 in the previous byte)																																															
Dependencies	IA32_RTIT_CTL.CYCEn && TriggerEn	Generation Scenario	Can be sent at any time, though a maximum of one CYC packet is sent per core clock cycle. See Section 36.3.6 for CYC-eligible packets.																																													
Description	<p>The Cycle Counter field increments at the same rate as the processor core clock ticks, but with a variable length format (using a trailing EXP bit field) and a range-capped byte length.</p> <p>If the CYC value is less than 32, a 1-byte CYC will be generated, with Exp=0. If the CYC value is between 32 and 4095 inclusive, a 2-byte CYC will be generated, with byte 0 Exp=1 and byte 1 Exp=0. And so on.</p> <p>CYC provides the number of core clocks that have passed since the last CYC packet. CYC can be configured to be sent in every cycle in which an eligible packet is generated, or software can opt to use a threshold to limit the number of CYC packets, at the expense of some precision. These settings are configured using the IA32_RTIT_CTL.CycThresh field (see Section 36.2.7.2). For details on Cycle-Accurate Mode, IPC calculation, etc, see Section 36.3.6.</p> <p>When CycThresh=0, and hence no threshold is in use, then a CYC packet will be generated in any cycle in which any CYC-eligible packet is generated. The CYC packet will precede the other packets generated in the cycle, and provides the precise cycle time of the packets that follow.</p> <p>In addition to these CYC packets generated with other packets, CYC packets can be sent stand-alone. These packets serve simply to update the decoder with the number of cycles passed, and are used to ensure that a wrap of the processor's internal cycle counter doesn't cause cycle information to be lost. These stand-alone CYC packets do not indicate the cycle time of any other packet or operation, and will be followed by another CYC packet before any other CYC-eligible packet is seen.</p> <p>When CycThresh&gt;0, CYC packets are generated only after a minimum number of cycles have passed since the last CYC packet. Once this threshold has passed, the behavior above resumes, where CYC will either be sent in the next cycle that produces other CYC-eligible packets, or could be sent stand-alone.</p> <p>When using CYC thresholds, only the cycle time of the operation (instruction or event) that generates the CYC packet is truly known. Other operations simply have their execution time bounded: they completed at or after the last CYC time, and before the next CYC time.</p>																																															
Application	<p>CYC provides the offset cycle time (since the last CYC packet) for the CYC-eligible packet that follows. If another CYC is encountered before the next CYC-eligible packet, the cycle values should be accumulated and applied to the next CYC-eligible packet.</p> <p>If a CYC packet is generated by a TNT, note that the cycle time provided by the CYC packet applies to the first branch in the TNT packet.</p>																																															

## 36.4.2.15 VMCS Packet

Table 36-34. VMCS Packet Definition

Name	VMCS Packet																																																																										
Packet Format	<table border="1"> <thead> <tr> <th></th> <th>7</th> <th>6</th> <th>5</th> <th>4</th> <th>3</th> <th>2</th> <th>1</th> <th>0</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>1</td> <td>0</td> </tr> <tr> <td>1</td> <td>1</td> <td>1</td> <td>0</td> <td>0</td> <td>1</td> <td>0</td> <td>0</td> <td>0</td> </tr> <tr> <td>2</td> <td colspan="8">VMCS Base Address [19:12]</td> </tr> <tr> <td>3</td> <td colspan="8">VMCS Base Address [27:20]</td> </tr> <tr> <td>4</td> <td colspan="8">VMCS Base Address [35:28]</td> </tr> <tr> <td>5</td> <td colspan="8">VMCS Base Address [43:36]</td> </tr> <tr> <td>6</td> <td colspan="8">VMCS Base Address [51:44]</td> </tr> </tbody> </table>				7	6	5	4	3	2	1	0	0	0	0	0	0	0	0	1	0	1	1	1	0	0	1	0	0	0	2	VMCS Base Address [19:12]								3	VMCS Base Address [27:20]								4	VMCS Base Address [35:28]								5	VMCS Base Address [43:36]								6	VMCS Base Address [51:44]							
	7	6	5	4	3	2	1	0																																																																			
0	0	0	0	0	0	0	1	0																																																																			
1	1	1	0	0	1	0	0	0																																																																			
2	VMCS Base Address [19:12]																																																																										
3	VMCS Base Address [27:20]																																																																										
4	VMCS Base Address [35:28]																																																																										
5	VMCS Base Address [43:36]																																																																										
6	VMCS Base Address [51:44]																																																																										
Dependencies	TriggerEn && ContextEn; Also in VMX operation.	Generation Scenario	Generated on successful VMPTRLD, and optionally on SMM VM exits and VM entries that return from SMM (see Section 36.5).																																																																								
Description	<p>The VMCS packet provides an address related to a VMCS pointer for a decoder to determine the transition of code contexts:</p> <ul style="list-style-type: none"> <li>On a successful VMPTRLD (i.e., a VMPTRLD that doesn't fault, fail, or VM exit), the VMCS packet contains the address of the current working VMCS pointer of the logical processor that will execute a VM guest context.</li> <li>On SMM VM exits, the VMCS packet provides the STM VMCS base address (SMM Transfer VMCS pointer), if VMCS-based controls are clear (see Section 36.5.1). See Section 36.6 on tracing inside and outside STM.</li> <li>On VM entries that return from SMM, the VMCS packet provides the current working VMCS pointer of the guest VM (see Section 36.6), if VMCS-based controls are clear (see Section 36.5.1). Root versus Non-Root operation can be distinguished from the PIP.NR bit.</li> </ul> <p>If a VMCS packet is generated before a VMCS has been loaded, or after it has been cleared, the base address value will be all 1s.</p> <p>VMCS packets will not be seen on processors with IA32_VMX_MISC[bit 14]=0, as these processors do not allow TraceEn to be set in VMX operation.</p>																																																																										
Application	<p>The purpose of the VMCS packet is to help the decoder uniquely identify changes in the executing software context in situations that CR3 may not be unique.</p> <p>When a VMCS is encountered, a decoder should do the following:</p> <ul style="list-style-type: none"> <li>If there was a prior unbound FUP (that is, a FUP not preceded by a packet such as MODE.TSX that consumes it, and it hence pairs with a TIP that has not yet been seen), then this VMCS is part of a compound packet event (Section 36.4.1). Find the ending TIP and apply the new VMCS base pointer value to the TIP payload IP.</li> <li>Otherwise, look for the next VMPTRLD, VMRESUME, or VMLAUNCH in the disassembly, and apply the new VMCS base pointer on the next VM entry.</li> </ul> <p>For examples of the packets generated by these flows, see Section 36.7.</p>																																																																										

### 36.4.2.16 Overflow (OVF) Packet

**Table 36-35. OVF Packet Definition**

Name	Overflow (OVF) Packet																													
Packet Format	<table border="1"> <thead> <tr> <th></th> <th>7</th> <th>6</th> <th>5</th> <th>4</th> <th>3</th> <th>2</th> <th>1</th> <th>0</th> </tr> </thead> <tbody> <tr> <th>0</th> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>1</td> <td>0</td> </tr> <tr> <th>1</th> <td>1</td> <td>1</td> <td>1</td> <td>1</td> <td>0</td> <td>0</td> <td>1</td> <td>1</td> </tr> </tbody> </table>				7	6	5	4	3	2	1	0	0	0	0	0	0	0	0	1	0	1	1	1	1	1	0	0	1	1
	7	6	5	4	3	2	1	0																						
0	0	0	0	0	0	0	1	0																						
1	1	1	1	1	0	0	1	1																						
Dependencies	TriggerEn	Generation Scenario	On resolution of internal buffer overflow																											
Description	OVF simply indicates to the decoder that an internal buffer overflow occurred, and packets were likely lost. If BranchEN= 1, OVF is followed by a FUP or TIP.PGE which will provide the IP at which packet generation resumes. See Section 36.3.8.																													
Application	When an OVF packet is encountered, the decoder should skip to the IP given in the subsequent FUP or TIP.PGE. The cycle counter for the CYC packet will be reset at the time the OVF packet is sent. Software should reset its call stack depth on overflow, since no RET compression is allowed across an overflow. Similarly, any IP compression that follows the OVF is guaranteed to use as a reference LastIP the IP payload of an IP packet that preceded the overflow.																													

### 36.4.2.17 Packet Stream Boundary (PSB) Packet

**Table 36-36. PSB Packet Definition**

Name	Packet Stream Boundary (PSB) Packet																																																																																																																																																																
Packet Format	<table border="1"> <thead> <tr> <th></th> <th>7</th> <th>6</th> <th>5</th> <th>4</th> <th>3</th> <th>2</th> <th>1</th> <th>0</th> </tr> </thead> <tbody> <tr> <th>0</th> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>1</td> <td>0</td> </tr> <tr> <th>1</th> <td>1</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>1</td> <td>0</td> </tr> <tr> <th>2</th> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>1</td> <td>0</td> </tr> <tr> <th>3</th> <td>1</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>1</td> <td>0</td> </tr> <tr> <th>4</th> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>1</td> <td>0</td> </tr> <tr> <th>5</th> <td>1</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>1</td> <td>0</td> </tr> <tr> <th>6</th> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>1</td> <td>0</td> </tr> <tr> <th>7</th> <td>1</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>1</td> <td>0</td> </tr> <tr> <th>8</th> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>1</td> <td>0</td> </tr> <tr> <th>9</th> <td>1</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>1</td> <td>0</td> </tr> <tr> <th>10</th> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>1</td> <td>0</td> </tr> <tr> <th>11</th> <td>1</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>1</td> <td>0</td> </tr> <tr> <th>12</th> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>1</td> <td>0</td> </tr> <tr> <th>13</th> <td>1</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>1</td> <td>0</td> </tr> <tr> <th>14</th> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>1</td> <td>0</td> </tr> <tr> <th>15</th> <td>1</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>1</td> <td>0</td> </tr> </tbody> </table>									7	6	5	4	3	2	1	0	0	0	0	0	0	0	0	1	0	1	1	0	0	0	0	0	1	0	2	0	0	0	0	0	0	1	0	3	1	0	0	0	0	0	1	0	4	0	0	0	0	0	0	1	0	5	1	0	0	0	0	0	1	0	6	0	0	0	0	0	0	1	0	7	1	0	0	0	0	0	1	0	8	0	0	0	0	0	0	1	0	9	1	0	0	0	0	0	1	0	10	0	0	0	0	0	0	1	0	11	1	0	0	0	0	0	1	0	12	0	0	0	0	0	0	1	0	13	1	0	0	0	0	0	1	0	14	0	0	0	0	0	0	1	0	15	1	0	0	0	0	0	1	0
	7	6	5	4	3	2	1	0																																																																																																																																																									
0	0	0	0	0	0	0	1	0																																																																																																																																																									
1	1	0	0	0	0	0	1	0																																																																																																																																																									
2	0	0	0	0	0	0	1	0																																																																																																																																																									
3	1	0	0	0	0	0	1	0																																																																																																																																																									
4	0	0	0	0	0	0	1	0																																																																																																																																																									
5	1	0	0	0	0	0	1	0																																																																																																																																																									
6	0	0	0	0	0	0	1	0																																																																																																																																																									
7	1	0	0	0	0	0	1	0																																																																																																																																																									
8	0	0	0	0	0	0	1	0																																																																																																																																																									
9	1	0	0	0	0	0	1	0																																																																																																																																																									
10	0	0	0	0	0	0	1	0																																																																																																																																																									
11	1	0	0	0	0	0	1	0																																																																																																																																																									
12	0	0	0	0	0	0	1	0																																																																																																																																																									
13	1	0	0	0	0	0	1	0																																																																																																																																																									
14	0	0	0	0	0	0	1	0																																																																																																																																																									
15	1	0	0	0	0	0	1	0																																																																																																																																																									

**Table 36-36. PSB Packet Definition (Contd.)**

Dependencies	TriggerEn	Generation Scenario	Periodic, based on the number of output bytes generated while tracing. PSB is sent when IA32_RTIT_STATUS.PacketByteCnt=0, and each time it crosses the software selected threshold after that. May be sent for other micro-architectural conditions as well.
Description	PSB is a unique pattern in the packet output log, and hence serves as a sync point for the decoder. It is a pattern that the decoder can search for in order to get aligned on packet boundaries. This packet is periodic, based on the number of output bytes, as indicated by IA32_RTIT_STATUS.PacketByteCnt. The period is chosen by software, via IA32_RTIT_CTL.PSBFreq (see Section 36.2.7.2). Note, however, that the PSB period is not precise, it simply reflects the average number of output bytes that should pass between PSBs. The processor will make a best effort to insert PSB as quickly after the selected threshold is reached as possible. The processor also may send extra PSB packets for some micro-architectural conditions. PSB also serves as the leading packet for a set of “status-only” packets collectively known as PSB+ (Section 36.3.7).		
Application	When a PSB is seen, the decoder should interpret all following packets as “status only”, until either a PSBEND or OVF packet is encountered. “Status only” implies that the binding and ordering rules to which these packets normally adhere are ignored, and the state they carry can instead be applied to the IP payload in the FUP packet that is included.		

### 36.4.2.18 PSBEND Packet

**Table 36-37. PSBEND Packet Definition**

Name	PSBEND Packet																																		
Packet Format	<table border="1"> <thead> <tr> <th></th> <th>7</th> <th>6</th> <th>5</th> <th>4</th> <th>3</th> <th>2</th> <th>1</th> <th>0</th> </tr> </thead> <tbody> <tr> <th>0</th> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>1</td> <td>0</td> </tr> <tr> <th>1</th> <td>0</td> <td>0</td> <td>1</td> <td>0</td> <td>0</td> <td>0</td> <td>1</td> <td>1</td> </tr> </tbody> </table>									7	6	5	4	3	2	1	0	0	0	0	0	0	0	0	1	0	1	0	0	1	0	0	0	1	1
	7	6	5	4	3	2	1	0																											
0	0	0	0	0	0	0	1	0																											
1	0	0	1	0	0	0	1	1																											
Dependencies	TriggerEn	Generation Scenario	Always follows PSB packet, separated by PSB+ packets																																
Description	PSBEND is simply a terminator for the series of “status only” (PSB+) packets that follow PSB (Section 36.3.7).																																		
Application	When a PSBEND packet is seen, the decoder should cease to treat packets as “status only”.																																		

### 36.4.2.19 Maintenance (MNT) Packet

**Table 36-38. MNT Packet Definition**

Name	Maintenance (MNT) Packet																																																																																																														
Packet Format	<table border="1"> <thead> <tr> <th></th> <th>7</th> <th>6</th> <th>5</th> <th>4</th> <th>3</th> <th>2</th> <th>1</th> <th>0</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>1</td> <td>0</td> </tr> <tr> <td>1</td> <td>1</td> <td>1</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>1</td> <td>1</td> </tr> <tr> <td>2</td> <td>1</td> <td>0</td> <td>0</td> <td>0</td> <td>1</td> <td>0</td> <td>0</td> <td>0</td> </tr> <tr> <td>3</td> <td colspan="8">Payload[7:0]</td> </tr> <tr> <td>4</td> <td colspan="8">Payload[15:8]</td> </tr> <tr> <td>5</td> <td colspan="8">Payload[23:16]</td> </tr> <tr> <td>6</td> <td colspan="8">Payload[31:24]</td> </tr> <tr> <td>7</td> <td colspan="8">Payload[39:32]</td> </tr> <tr> <td>8</td> <td colspan="8">Payload[47:40]</td> </tr> <tr> <td>9</td> <td colspan="8">Payload[55:48]</td> </tr> <tr> <td>10</td> <td colspan="8">Payload[63:56]</td> </tr> </tbody> </table>				7	6	5	4	3	2	1	0	0	0	0	0	0	0	0	1	0	1	1	1	0	0	0	0	1	1	2	1	0	0	0	1	0	0	0	3	Payload[7:0]								4	Payload[15:8]								5	Payload[23:16]								6	Payload[31:24]								7	Payload[39:32]								8	Payload[47:40]								9	Payload[55:48]								10	Payload[63:56]							
	7	6	5	4	3	2	1	0																																																																																																							
0	0	0	0	0	0	0	1	0																																																																																																							
1	1	1	0	0	0	0	1	1																																																																																																							
2	1	0	0	0	1	0	0	0																																																																																																							
3	Payload[7:0]																																																																																																														
4	Payload[15:8]																																																																																																														
5	Payload[23:16]																																																																																																														
6	Payload[31:24]																																																																																																														
7	Payload[39:32]																																																																																																														
8	Payload[47:40]																																																																																																														
9	Payload[55:48]																																																																																																														
10	Payload[63:56]																																																																																																														
Dependencies	TriggerEn	Generation Scenario	Implementation specific.																																																																																																												
Description	This packet is generated by hardware, the payload meaning is model-specific.																																																																																																														
Application	Unless a decoder has been extended for a particular family/model/stepping to interpret MNT packet payloads, this packet should simply be ignored. It does not bind to any IP.																																																																																																														

### 36.4.2.20 PAD Packet

**Table 36-39. PAD Packet Definition**

Name	PAD Packet																				
Packet Format	<table border="1"> <thead> <tr> <th></th> <th>7</th> <th>6</th> <th>5</th> <th>4</th> <th>3</th> <th>2</th> <th>1</th> <th>0</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> </tr> </tbody> </table>				7	6	5	4	3	2	1	0	0	0	0	0	0	0	0	0	0
	7	6	5	4	3	2	1	0													
0	0	0	0	0	0	0	0	0													
Dependencies	TriggerEn	Generation Scenario	Implementation specific																		
Description	PAD is simply a NOP packet. Processor implementations may choose to add pad packets to improve packet alignment or for implementation-specific reasons.																				
Application	Ignore PAD packets.																				



## 36.4.2.21 PTWRITE Packet

Table 36-40. PTW Packet Definition

Name	PTW Packet																																																																																																										
Packet Format	<table border="1"> <tr> <td></td> <td>7</td> <td>6</td> <td>5</td> <td>4</td> <td>3</td> <td>2</td> <td>1</td> <td>0</td> </tr> <tr> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>1</td> <td>0</td> </tr> <tr> <td>1</td> <td>IP</td> <td colspan="2">PayloadBytes</td> <td>1</td> <td>0</td> <td>0</td> <td>1</td> <td>0</td> </tr> <tr> <td>2</td> <td colspan="8">Payload[7:0]</td> </tr> <tr> <td>3</td> <td colspan="8">Payload[15:8]</td> </tr> <tr> <td>4</td> <td colspan="8">Payload[23:16]</td> </tr> <tr> <td>5</td> <td colspan="8">Payload[31:24]</td> </tr> <tr> <td>6</td> <td colspan="8">Payload[39:32]</td> </tr> <tr> <td>7</td> <td colspan="8">Payload[47:40]</td> </tr> <tr> <td>8</td> <td colspan="8">Payload[55:48]</td> </tr> <tr> <td>9</td> <td colspan="8">Payload[63:56]</td> </tr> </table>									7	6	5	4	3	2	1	0	0	0	0	0	0	0	0	1	0	1	IP	PayloadBytes		1	0	0	1	0	2	Payload[7:0]								3	Payload[15:8]								4	Payload[23:16]								5	Payload[31:24]								6	Payload[39:32]								7	Payload[47:40]								8	Payload[55:48]								9	Payload[63:56]							
	7	6	5	4	3	2	1	0																																																																																																			
0	0	0	0	0	0	0	1	0																																																																																																			
1	IP	PayloadBytes		1	0	0	1	0																																																																																																			
2	Payload[7:0]																																																																																																										
3	Payload[15:8]																																																																																																										
4	Payload[23:16]																																																																																																										
5	Payload[31:24]																																																																																																										
6	Payload[39:32]																																																																																																										
7	Payload[47:40]																																																																																																										
8	Payload[55:48]																																																																																																										
9	Payload[63:56]																																																																																																										
	<p>The PayloadBytes field indicates the number of bytes of payload that follow the header bytes. Encodings are as follows:</p> <table border="1"> <thead> <tr> <th>PayloadBytes</th> <th>Bytes of Payload</th> </tr> </thead> <tbody> <tr> <td>'00</td> <td>4</td> </tr> <tr> <td>'01</td> <td>8</td> </tr> <tr> <td>'10</td> <td>Reserved</td> </tr> <tr> <td>'11</td> <td>Reserved</td> </tr> </tbody> </table> <p>IP bit indicates if a FUP, whose payload will be the IP of the PTWRITE instruction, will follow.</p>								PayloadBytes	Bytes of Payload	'00	4	'01	8	'10	Reserved	'11	Reserved																																																																																									
PayloadBytes	Bytes of Payload																																																																																																										
'00	4																																																																																																										
'01	8																																																																																																										
'10	Reserved																																																																																																										
'11	Reserved																																																																																																										
Dependencies	TriggerEn & ContextEn & FilterEn & PTWEn	Generation Scenario	PTWRITE Instruction																																																																																																								
Description	Contains the value held in the PTWRITE operand. This packet is CYC-eligible, and hence will generate a CYC packet if IA32_RTIT_CTL.CYCEn=1 and any CYC Threshold has been reached.																																																																																																										
Application	Binds to the associated PTWRITE instruction. The IP of the PTWRITE will be provided by a following FUP, when PTW.IP=1.																																																																																																										

### 36.4.2.22 Execution Stop (EXSTOP) Packet

**Table 36-41. EXSTOP Packet Definition**

Name	EXSTOP Packet																													
Packet Format	<table border="1" style="margin-left: 20px;"> <tr> <td></td> <td>7</td> <td>6</td> <td>5</td> <td>4</td> <td>3</td> <td>2</td> <td>1</td> <td>0</td> </tr> <tr> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>1</td> <td>0</td> </tr> <tr> <td>1</td> <td>IP</td> <td>1</td> <td>1</td> <td>0</td> <td>0</td> <td>0</td> <td>1</td> <td>0</td> </tr> </table> <p>IP bit indicates if a FUP will follow.</p>				7	6	5	4	3	2	1	0	0	0	0	0	0	0	0	1	0	1	IP	1	1	0	0	0	1	0
	7	6	5	4	3	2	1	0																						
0	0	0	0	0	0	0	1	0																						
1	IP	1	1	0	0	0	1	0																						
Dependencies	TriggerEn & PwrEvtEn	Generation Scenario	C-state entry, P-state change, or other processor clock power-down. Includes : <ul style="list-style-type: none"> <li>▪ Entry to C-state deeper than C0.0</li> <li>▪ TM1/2</li> <li>▪ STPCLK#</li> <li>▪ Frequency change due to IA32_CLOCK_MODULATION, Turbo</li> </ul>																											
Description	This packet indicates that software execution has stopped due to processor clock powerdown. Later packets will indicate when execution resumes. If EXSTOP is generated while ContextEn is set, the IP bit will be set, and EXSTOP will be followed by a FUP packet containing the IP at which execution stopped. More precisely, this will be the IP of the oldest instruction that has not yet completed. This packet is CYC-eligible, and hence will generate a CYC packet if IA32_RTIT_CTL.CYCEn=1 and any CYC Threshold has been reached.																													
Application	If a FUP follows EXSTOP (hence IP bit set), the EXSTOP can be bound to the FUP IP. Otherwise the IP is not known. Time of powerdown can be inferred from the preceding CYC, if CYCEn=1. Combined with the TSC at the time of wake (if TSCEn=1), this can be used to determine the duration of the powerdown.																													

## 36.4.2.23 MWAIT Packet

Table 36-42. MWAIT Packet Definition

Name	MWAIT Packet																																																																																																											
Packet Format	<table border="1"> <thead> <tr> <th></th> <th>7</th> <th>6</th> <th>5</th> <th>4</th> <th>3</th> <th>2</th> <th>1</th> <th>0</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>1</td> <td>0</td> </tr> <tr> <td>1</td> <td>1</td> <td>1</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>1</td> <td>0</td> </tr> <tr> <td>2</td> <td colspan="8">MWAIT Hints[7:0]</td> </tr> <tr> <td>3</td> <td colspan="8">Reserved</td> </tr> <tr> <td>4</td> <td colspan="8">Reserved</td> </tr> <tr> <td>5</td> <td colspan="8">Reserved</td> </tr> <tr> <td>6</td> <td colspan="6">Reserved</td> <td colspan="2">EXT[1:0]</td> </tr> <tr> <td>7</td> <td colspan="8">Reserved</td> </tr> <tr> <td>8</td> <td colspan="8">Reserved</td> </tr> <tr> <td>9</td> <td colspan="8">Reserved</td> </tr> </tbody> </table>										7	6	5	4	3	2	1	0	0	0	0	0	0	0	0	1	0	1	1	1	0	0	0	0	1	0	2	MWAIT Hints[7:0]								3	Reserved								4	Reserved								5	Reserved								6	Reserved						EXT[1:0]		7	Reserved								8	Reserved								9	Reserved							
	7	6	5	4	3	2	1	0																																																																																																				
0	0	0	0	0	0	0	1	0																																																																																																				
1	1	1	0	0	0	0	1	0																																																																																																				
2	MWAIT Hints[7:0]																																																																																																											
3	Reserved																																																																																																											
4	Reserved																																																																																																											
5	Reserved																																																																																																											
6	Reserved						EXT[1:0]																																																																																																					
7	Reserved																																																																																																											
8	Reserved																																																																																																											
9	Reserved																																																																																																											
Dependencies	TriggerEn & PwrEvtEn & ContextEn	Generation Scenario	MWAIT instruction, or I/O redirection to MWAIT, that complete without fault or VMexit.																																																																																																									
Description	Indicates that an MWAIT operation to C-state deeper than C0.0 completed. The MWAIT hints and extensions passed in by software are exposed in the payload. This packet is CYC-eligible, and hence will generate a CYC packet if IA32_RTIT_CTL.CYCEn=1 and any CYC Threshold has been reached.																																																																																																											
Application	The MWAIT packet should bind to the IP of the next FUP, which will be the IP of the instruction that caused the MWAIT. This FUP will be shared with EXSTOP.																																																																																																											

### 36.4.2.24 Power Entry (PWRE) Packet

**Table 36-43. PWRE Packet Definition**

Name	PWRE Packet																																															
Packet Format	<table border="1"> <tr> <td></td> <td>7</td> <td>6</td> <td>5</td> <td>4</td> <td>3</td> <td>2</td> <td>1</td> <td>0</td> </tr> <tr> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>1</td> <td>0</td> </tr> <tr> <td>1</td> <td>0</td> <td>0</td> <td>1</td> <td>0</td> <td>0</td> <td>0</td> <td>1</td> <td>0</td> </tr> <tr> <td>2</td> <td>HW</td> <td colspan="7">Reserved</td> </tr> <tr> <td>3</td> <td colspan="4">Resolved Thread C-State</td> <td colspan="4">Resolved Thread Sub C-State</td> </tr> </table>				7	6	5	4	3	2	1	0	0	0	0	0	0	0	0	1	0	1	0	0	1	0	0	0	1	0	2	HW	Reserved							3	Resolved Thread C-State				Resolved Thread Sub C-State			
	7	6	5	4	3	2	1	0																																								
0	0	0	0	0	0	0	1	0																																								
1	0	0	1	0	0	0	1	0																																								
2	HW	Reserved																																														
3	Resolved Thread C-State				Resolved Thread Sub C-State																																											
Dependencies	TriggerEn & PwrEvtEn	Generation Scenario	Transition to a C-state deeper than C0.0.																																													
Description	<p>Indicates processor entry to the resolved thread C-state and sub C-state indicated. The processor will remain in this C-state until either another PWRE indicates the processor has moved to a C-state deeper than C0.0, or a PWRX packet indicates a return to C0.</p> <p>Note that some CPUs may allow MWAIT to request a deeper C-state than is supported by the core. These deeper C-states may have platform-level implications that differentiate them. However, the PWRE packet will provide only the resolved thread C-state, which will not exceed that supported by the core.</p> <p>If the C-state entry was initiated by hardware, rather than a direct software request (such as MWAIT, HLT, or shut-down), the HW bit will be set to indicate this. Hardware Duty Cycling (see Section 14.5, "Hardware Duty Cycling (HDC)" in the <i>Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B</i>) is an example of such a case.</p>																																															
Application	<p>When transitioning from C0.0 to a deeper C-state, the PWRE packet will be followed by an EXSTOP. If that EXSTOP packet has the IP bit set, then the following FUP will provide the IP at which the C-state entry occurred. Subsequent PWRE packets generated before the next PWRX should bind to the same IP.</p>																																															

### 36.4.2.25 Power Exit (PWRX) Packet

Table 36-44. PWRX Packet Definition

Name	PWRX Packet																																																																										
Packet Format	<table border="1"> <tr> <td></td> <td>7</td> <td>6</td> <td>5</td> <td>4</td> <td>3</td> <td>2</td> <td>1</td> <td>0</td> </tr> <tr> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>1</td> <td>0</td> </tr> <tr> <td>1</td> <td>1</td> <td>0</td> <td>1</td> <td>0</td> <td>0</td> <td>0</td> <td>1</td> <td>0</td> </tr> <tr> <td>2</td> <td colspan="4">Last Core C-State</td> <td colspan="4">Deepest Core C-State</td> </tr> <tr> <td>3</td> <td colspan="4">Reserved</td> <td colspan="4">Wake Reason</td> </tr> <tr> <td>4</td> <td colspan="8">Reserved</td> </tr> <tr> <td>5</td> <td colspan="8">Reserved</td> </tr> <tr> <td>6</td> <td colspan="8">Reserved</td> </tr> </table>				7	6	5	4	3	2	1	0	0	0	0	0	0	0	0	1	0	1	1	0	1	0	0	0	1	0	2	Last Core C-State				Deepest Core C-State				3	Reserved				Wake Reason				4	Reserved								5	Reserved								6	Reserved							
	7	6	5	4	3	2	1	0																																																																			
0	0	0	0	0	0	0	1	0																																																																			
1	1	0	1	0	0	0	1	0																																																																			
2	Last Core C-State				Deepest Core C-State																																																																						
3	Reserved				Wake Reason																																																																						
4	Reserved																																																																										
5	Reserved																																																																										
6	Reserved																																																																										
Dependencies	TriggerEn & PwrEvtEn	Generation Scenario	Transition from a C-state deeper than C0.0 to C0.																																																																								
Description	<p>Indicates processor return to thread C0 from a C-state deeper than C0.0. The Last Core C-State field provides the MWAIT encoding for the core C-state at the time of the wake. The Deepest Core C-State provides the MWAIT encoding for the deepest core C-state achieved during the sleep session, or since leaving thread C0. MWAIT encodings for C-states can be found in Table 4-11 in the <i>Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B</i>. Note that these values reflect only the core C-state, and hence will not exceed the maximum supported core C-state, even if deeper C-states can be requested. The Wake Reason field is one-hot, encoded as follows:</p> <table border="1"> <thead> <tr> <th>Bit</th> <th>Field</th> <th>Meaning</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>Interrupt</td> <td>Wake due to external interrupt received.</td> </tr> <tr> <td>1</td> <td>Reserved</td> <td></td> </tr> <tr> <td>2</td> <td>Store to Monitored Address</td> <td>Wake due to store to monitored address.</td> </tr> <tr> <td>3</td> <td>HW Wake</td> <td>Wake due to hardware autonomous condition, such as HDC.</td> </tr> </tbody> </table>			Bit	Field	Meaning	0	Interrupt	Wake due to external interrupt received.	1	Reserved		2	Store to Monitored Address	Wake due to store to monitored address.	3	HW Wake	Wake due to hardware autonomous condition, such as HDC.																																																									
Bit	Field	Meaning																																																																									
0	Interrupt	Wake due to external interrupt received.																																																																									
1	Reserved																																																																										
2	Store to Monitored Address	Wake due to store to monitored address.																																																																									
3	HW Wake	Wake due to hardware autonomous condition, such as HDC.																																																																									
Application	PWRX will always apply to the same IP as the PWRE. The time of wake can be discerned from (optional) timing packets that precede PWRX.																																																																										

## 36.5 TRACING IN VMX OPERATION

On processors that IA32\_VMX\_MISC[bit 14] reports 1, TraceEn can be set in VMX operation. A series of mechanisms exist to allow the VMM to configure tracing based on the desired trace domain, and on the consumer of the trace output. The VMM can configure specific VM-execution controls to control what virtualization-specific data are included within the trace packets (see Section 36.5.1 for details). MSR save and load lists can be employed by the VMM to restrict tracing to the desired context (see Section 36.5.2 for details). These configuration options are summarized in Table 36-45. Table 36-45 covers common Intel PT usages while SMIs are handled by the default SMM treatment. Tracing with SMM Transfer Monitor is described in Section 36.6.

**Table 36-45. Common Usages of Intel PT and VMX**

Target Domain	Output Consumer	Virtualize Output	Configure VMCS Controls	TraceEN Configuration	Save/Restore MSR states of Trace Configuration
System-Wide (VMM + VMs)	Host	NA	Default Setting (no suppression)	WRMSR or XRSTORS by Host	NA
VMM Only	Intel PT Aware VMM	NA	Enable suppression	MSR load list to disable tracing in VM, enable tracing on VM exits	NA
VM Only	Intel PT Aware VMM	NA	Enable suppression	MSR load list to enable tracing in VM, disable tracing on VM exits	NA
Intel PT Aware Guest(s)	Per Guest	VMM adds trace output virtualization	Enable suppression	MSR load list to enable tracing in VM, disable tracing on VM exits	VMM Update guest state on XRSTORS-exiting VM exits

### 36.5.1 VMX-Specific Packets and VMCS Controls

In all of the usages of VMX and Intel PT, the decoder in the host or VMM context can identify the occurrences of VMX transitions with the aid of VMX-specific packets. Packets relevant to VMX fall into the follow two kinds:

- **VMCS Packet:** The VMX transitions of individual VM can be distinguished by a decoder using the base address field in a VMCS packet. The base address field stores the VMCS pointer address of a successful VMPTRLD. A VMCS packet is sent on a successful execution of VMPTRLD. See Section 36.4.2.15 for details.
- **NonRoot (NR) bit field in PIP packet:** PIP packets are generated with each VM entry/exit. The NR bit in a PIP packet is set when in VMX non-Root operation. Thus a transition of the NR bit from 0 to 1 indicates the occurrence of a VM entry, and a transition of 1 to 0 indicates the occurrence of a VM exit.

Processors with IA32\_VMX\_MISC[bit 14]= 1 also provides VMCS controls that a VMM can configure to prevent VMX-specific information from leaking across virtualization boundaries.

**Table 36-46. VMCS Controls For Intel Processor Trace**

Name	Type	Bit Position	Value	Behavior
Conceal VMX non-root operation from Intel PT	VM-execution control	19	0	PIPs generated in VM non-root operation will set the PIP.NR bit. PSB+ in VMX non-root operation will include the VMCS packet, to ensure that the decoder knows which guest is currently in use.
			1	PIPs generated in VMX non-root operation will not set the PIP.NR bit. PSB+ in VMX non-root operation will not include the VMCS packet.
Conceal VM exits from Intel PT	VM-exit control	24	0	PIPs are generated on VM exit, with NonRoot=0. On VM exit to SMM, VMCS packets are additionally generated.
			1	No PIP is generated on VM exit, and no VMCS packet is generated on VM exit to SMM.
Conceal VM entries from Intel PT	VM-entry control	17	0	PIPs are generated on VM entry, with NonRoot=1 if the destination of the VM entry is VMX non-root operation. On VM entry to SMM, VMCS packets are additionally generated.
			1	No PIP is generated on VM entry, and no VMCS packet is generated on VM entry to SMM.

The default setting for the VMCS controls that interacts with Intel PT is to enable all VMX-specific packet information. The scenarios that would use the default setting also do not require the VMM to use MSR load list to manage the configuration of turning-on/off of trace packet generation across VM exits.

If IA32\_VMX\_MISC[bit 14] reports 0, any attempt to set the VMCS control bits in Table 36-46 will result in a failure on guest entry.

## 36.5.2 Managing Trace Packet Generation Across VMX Transitions

In tracing scenarios that collect packets for both VMX root operation and VMX non-root operation, a host executive can manage the MSR associated with trace packet generation directly. The states of these MSRs need not be modified using MSR load list or MSR save list across VMX transitions.

For tracing scenarios that collect only packets within either VMX root operation or VMX non-root operation, the VMM can use the MSR load list and/or MSR save list to toggle IA32\_RTIT\_CTL.TraceEn.

### 36.5.2.1 System-Wide Tracing

When a host or VMM configures Intel PT to collect trace packets of the entire system, it can leave the VMCS controls clear to allow VMX-specific packets to provide information across VMX transitions. MSR load list is not used across VM exits or VM entries, nor is VM-exit MSR save list.

The decoder will desire to identify the occurrence of VMX transitions. The packets of interests to a decoder are shown in Table 36-47.

**Table 36-47. Packets on VMX Transitions (System-Wide Tracing)**

Event	Packets	Description
VM exit	FUP(GuestIP)	The FUP indicates at which point in the guest flow the VM exit occurred. This is important, since VM exit can be an asynchronous event. The IP will match that written into the VMCS.
	PIP(HostCR3, NR=0)	The PIP packet provides the new host CR3 value, as well as indication that the logical processor is entering VMX root operation. This allows the decoder to identify the change of executing context from guest to host and load the appropriate set of binaries to continue decode.
	TIP(HostIP)	The TIP indicates the destination IP, the IP of the first instruction to be executed in VMX root operation.  Note, this packet could be preceded by a MODE.Exec packet (Section 36.4.2.8). This is generated only in cases where CS.D or (CS.L & EFER.LMA) change during the transition.
VM entry	PIP(GuestCR3, NR=1)	The PIP packet provides the new guest CR3 value, as well as indication that the logical processor is entering VMX non-root operation. This allows the decoder to identify the change of executing context from host to guest and load the appropriate set of binaries to continue decode.
	TIP(GuestIP)	The TIP indicates the destination IP, the IP of the first instruction to be executed in VMX non-root operation. This should match the IP value read out from the VMCS.  Note, this packet could be preceded by a MODE.Exec packet (Section 36.4.2.8). This is generated only in cases where CS.D or (CS.L & EFER.LMA) change during the transition.

Since the packet suppression controls are cleared, the VMCS packet will be included in all PSB+ for this usage scenario. Thus the decoder can distinguish the execution context of different VMs. Additionally, it will be generated on VMPTRLD. Thus the decoder can distinguish the execution context of different VMs.

When the host VMM configures a system to collect trace packets in this scenario, it should emulate CPUID to report CPUID.(EAX=07H, ECX=0):EBX[bit 26] with 0 to guests, indicating to guests that Intel PT is not available.

### VMX TSC Manipulation

The TSC packets generated while in VMX non-root operation will include any changes resulting from the use of a VMM's use of the TSC offsetting or TSC scaling VMCS control (see Chapter 25, "VMX Non-Root Operation"). In this system-wide usage model, the decoder may need to account for the effect of per-VM adjustments in the TSC packets generated in VMX non-root operation and the absence of TSC adjustments in TSC packets generated in VMX root operation. The VMM can supply this information to the decoder.

### 36.5.2.2 Host-Only Tracing

When trace packets in VMX non-root operation are not desired, the VMM can use VM-entry MSR load list with IA32\_RTIT\_CTL.TraceEn=0 to disable trace packet generation in guests, set IA32\_RTIT\_CTL.TraceEn=1 via VM-exit MSR load list.

When tracing only the host, the decoder does not need information about the guests, the VMCS controls for suppressing VMX-specific packets can be set to reduce the packets generated. VMCS packets will still be generated on successful VMPTRLD and in PSB+ generated in the Host, but these will be unused by the decoder.

The packets of interests to a decoder when trace packets are collected for host-only tracing are shown in Table 36-48.

**Table 36-48. Packets on VMX Transitions (Host-Only Tracing)**

Event	Packets	Description
VM exit	TIP.PGE(HostIP)	The TIP.PGE indicates that trace packet generation is enabled and gives the IP of the first instruction to be executed in VMX root operation. Note, this packet could be preceded by a MODE.Exec packet (Section 36.4.2.8). This is generated only in cases where CS.D or (CS.L & EFER.LMA) change during the transition.
VM entry	TIP.PGD()	The TIP indicates that trace packet generation was disabled. This ensure that all buffered packets are flushed out.

### 36.5.2.3 Guest-Only Tracing

A VMM can configure trace packet generation while in non-root operation for guests executing normally. This is accomplished by utilizing the MSR load lists across VM exit and VM entry to confine trace packet generation to stay within the guest environment.

For this usage, the VM-entry MSR load list is programmed to turn on trace packet generation. The VM-exit MSR load list is used to clear TraceEn=0 to disable trace packet generation in the host. Further, if it is preferred that the guest packet stream contain no indication that execution was in VMX non-root operation, the VMM should set the VMCS controls described in Table 36-46.

### 36.5.2.4 Virtualization of Guest Output Packet Streams

Each Intel PT aware guest OS can produce one or more output packet streams to destination addresses specified as guest physical address (GPA) using context-switched IA32\_RTIT\_OUTPUT\_BASE within the guest. The processor generates trace packets to the platform physical address specified in IA32\_RTIT\_OUTPUT\_BASE, and those specified in the ToPA tables. Thus, a VMM that supports Intel PT aware guest OS may wish to virtualize the output configurations of IA32\_RTIT\_OUTPUT\_BASE and ToPA for each trace configuration state of all the guests.

### 36.5.2.5 Emulation of Intel PT Traced State

If a VMM emulates an element of processor state by taking a VM exit on reads and/or writes to that piece of state, and the state element impacts Intel PT packet generation or values, it may be incumbent upon the VMM to insert or modify the output trace data.

If a VM exit is taken on a guest write to CR3 (including "MOV CR3" as well as task switches), the PIP packet normally generated on the CR3 write will be missing.

To avoid decoder confusion when the guest trace is decoded, the VMM should emulate the missing PIP by writing it into the guest output buffer. If the guest CR3 value is manipulated, the VMM may also need to manipulate the IA32\_RTIT\_CR3\_MATCH value, in order to ensure the trace behavior matches the guest's expectation.

Similarly, if a VMM emulates the TSC value by taking a VM exit on RDTSC, the TSC packets generated in the trace may mismatch the TSC values returned by the VMM on RDTSC. To ensure that the trace can be properly aligned with software logs based on RDTSC, the VMM should either make corresponding modifications to the TSC packet values in the guest trace, or use mechanisms such as TSC offsetting or TSC scaling in place of exiting.

### 36.5.2.6 TSC Scaling

When TSC scaling is enabled for a guest using Intel PT, the VMM should ensure that the value of Maximum Non-Turbo Ratio[15:8] in MSR\_PLATFORM\_INFO (MSR 0CEH) and the TSC/"core crystal clock" ratio (EBX/EAX) in CPUID leaf 15H are set in a manner consistent with the resulting TSC rate that will be visible to the VM. This will allow the decoder to properly apply TSC packets, MTC packets (based on the core crystal clock or ART, whose frequency is indicated by CPUID leaf 15H), and CBR packets (which indicate the ratio of the processor frequency to the Max



Non-Turbo frequency). Absent this, or separate indication of the scaling factor, the decoder will be unable to properly track time in the trace. See Section 36.8.3 for details on tracking time within an Intel PT trace.

### 36.5.2.7 Failed VM Entry

The packets generated by a failed VM entry depend both on the VMCS configuration, as well as on the type of failure. The results to expect are summarized in the table below. Note that packets in *italics* may or may not be generated, depending on implementation choice, and the point of failure.

**Table 36-49. Packets on a Failed VM Entry**

Usage Model	Entry Configuration	Early Failure (fall through to Next IP)	Late Failure (VM exit)
System-Wide	No MSR load list	TIP (NextIP)	<i>PIP(Guest CR3, NR=1), TraceEn 0-&gt;1 Packets (See Section 36.2.7.3), PIP(HostCR3, NR=0), TIP(HostIP)</i>
VMM Only	MSR load list disables TraceEn	TIP (NextIP)	<i>TraceEn 0-&gt;1 Packets (See Section 36.2.7.3), TIP(HostIP)</i>
VM Only	MSR load list Enables TraceEn	None	None

### 36.5.2.8 VMX Abort

VMX abort conditions take the processor into a shutdown state. On a VM exit that leads to VMX abort, some packets (FUP, PIP) may be generated, but any expected TIP, TIP.PGE, or TIP.PGD may be dropped.

## 36.6 TRACING AND SMM TRANSFER MONITOR (STM)

SMM Transfer Monitor is a VMM that operates inside SMM while in VMX root operation. An STM operates in conjunction with an executive monitor. The latter operates outside SMM and in VMX root operation. Transitions from the executive monitor or its VMs to the STM are called SMM VM exits. The STM returns from SMM via a VM entry to the VM in VMX non-root operation or the executive monitor in VMX root operation.

Intel PT supports tracing in an STM similar to tracing support for VMX operation as described above in Section 36.7. As a result, on a SMM VM exit resulting from #SMI, TraceEn is not saved and then cleared. Software can save the state of the trace configuration MSRs and clear TraceEn using the MSR load/save lists.

## 36.7 PACKET GENERATION SCENARIOS

Table 36-50 and Table 36-52 illustrate the packets generated in various scenarios. In the heading row, PacketEn is abbreviated as PktEn, ContextEn as CntxEn. Note that this assumes that TraceEn=1 in IA32\_RTIT\_CTL, while TriggerEn=1 and Error=0 in IA32\_RTIT\_STATUS, unless otherwise specified. Entries that do not matter in packet generation are marked "D.C." Packets followed by a "?" imply that these packets depend on additional factors, which are listed in the "Other Dependencies" column.

In Table 36-50, PktEn is evaluated based on TiggerEn & ContextEn & FilterEn & BranchEn.

**Table 36-50. Packet Generation under Different Enable Conditions**

Case	Operation	PktEn Before	PktEn After	CntxEn After	Other Dependencies	Packets Output
1a	Normal non-jump operation	0	0	D.C.		None
1b	Normal non-jump operation	1	1	1		None
2a	WRMSR/XRSTORS/RSM that changes TraceEn 0 -> 1, with PacketByteCnt > 0	0	0	D.C.	*TSC if TSCEn=1; *TMA if TSCEn=MTCEn=1	TSC?, TMA?, CBR

**Table 36-50. Packet Generation under Different Enable Conditions (Contd.)**

Case	Operation	PktEn Before	PktEn After	CntxE After	Other Dependencies	Packets Output
2b	WRMSR/XRSTORS/RSM that changes TraceEn 0 -> 1, with PacketByteCnt =0	0	0	D.C.	*TSC if TSCEn=1; *TMA if TSCEn=MTCEn=1	PSB, PSBEND (see Section 36.4.2.17)
2d	WRMSR/XRSTORS/RSM that changes TraceEn 0 -> 1, with PacketByteCnt >0	0	1	1	TSC if TSCEn=1; TMA if TSCEn=MTCEn=1	TSC?, TMA?, CBR, MODE.Exec, TIP.PGE(NLIP)
2e	WRMSR/XRSTORS/RSM that changes TraceEn 0 -> 1, with PacketByteCnt =0	0	1	1		MODE.Exec, TIP.PGE(NLIP), PSB, PSBEND (see Section 36.4.2.8, 36.4.2.7, 36.4.2.13,36.4.2.15, 36.4.2.17)
3a	WRMSR that changes TraceEn 1 -> 0	0	0	D.C.		None
3b	WRMSR that changes TraceEn 1 -> 0	1	0	D.C.		FUP(CLIP), TIP.PGD()
5a	MOV to CR3	0	0	0		None
5f	MOV to CR3	0	0	1	TraceStop if executed in a TraceStop region	PIP(NewCR3,NR?), TraceStop?
5b	MOV to CR3	0	1	1	*PIP.NR=1 if not in root operation, and "Conceal VMX non-root operation from Intel PT" execution control = 0 *MODE.Exec if the mode has changed since the last MODE.Exec, or if no MODE.Exec since last PSB	PIP(NewCR3, NR?), MODE.Exec?, TIP.PGE(NLIP)
5c	MOV to CR3	1	0	0		TIP.PGD()
5e	MOV to CR3	1	0	1	*PIP.NR=1 if not in root operation, and "Conceal VMX non-root operation from Intel PT" execution control = 0 *TraceStop if executed in a TraceStop region	PIP(NewCR3, NR?), TIP.PGE(NLIP), TraceStop?
5d	MOV to CR3	1	1	1	*PIP.NR=1 if not in root operation, and "Conceal VMX non-root operation from Intel PT" execution control = 0	PIP(NewCR3, NR?)
6a	Unconditional direct near jump	0	0	D.C.		None
6b	Unconditional direct near jump	1	0	1	TraceStop if BLIP is in a TraceStop region	TIP.PGD(BLIP), TraceStop?
6c	Unconditional direct near jump	0	1	1	MODE.Exec if the mode has changed since the last MODE.Exec, or if no MODE.Exec since last PSB	MODE.Exec?, TIP.PGE(BLIP)
6d	Unconditional direct near jump	1	1	1		None
7a	Conditional taken jump or compressed RET that does not fill up the internal TNT buffer	0	0	D.C.		None

**Table 36-50. Packet Generation under Different Enable Conditions (Contd.)**

Case	Operation	PktEn Before	PktEn After	CntxEn After	Other Dependencies	Packets Output
7b	Conditional taken jump or compressed RET	0	1	1	MODE.Exec if the mode has changed since the last MODE.Exec, or if no MODE.Exec since last PSB	MODE.Exec?, TIP.PGE(BLIP)
7e	Conditional taken jump or compressed RET, with empty TNT buffer	1	0	1	TraceStop if BLIP is in a TraceStop region	TIP.PGD(), TraceStop?
7f	Conditional taken jump or compressed RET, with non-empty TNT buffer	1	0	1	TraceStop if BLIP is in a TraceStop region	TNT, TIP.PGD(), TraceStop?
7d	Conditional taken jump or compressed RET that fills up the internal TNT buffer	1	1	1		TNT
8a	Conditional non-taken jump	0	0	D.C.		None
8d	Conditional not-taken jump that fills up the internal TNT buffer	1	1	1		TNT
9a	Near indirect jump (JMP, CALL, or uncompressed RET)	0	0	D.C.		None
9b	Near indirect jump (JMP, CALL, or uncompressed RET)	0	1	1	MODE.Exec if the mode has changed since the last MODE.Exec, or if no MODE.Exec since last PSB	MODE.Exec?, TIP.PGE(BLIP)
9c	Near indirect jump (JMP, CALL, or uncompressed RET)	1	0	1	TraceStop if BLIP is in a TraceStop region	TIP.PGD(BLIP), TraceStop?
9d	Near indirect jump (JMP, CALL, or uncompressed RET)	1	1	1		TIP(BLIP)
10a	Far Branch (CALL/JMP/RET)	0	0	0		None
10f	Far Branch (CALL/JMP/RET)	0	0	1	*PIP if CR3 is updated (i.e., task switch), and OS=1; *PIP.NR=1 if destination is not root operation, and "Conceal VMX non-root operation from Intel PT" execution control = 0; *TraceStop if BLIP is in a TraceStop region	PIP(new CR3, NR?), TraceStop?
10b	Far Branch (CALL/JMP/RET)	0	1	1	*PIP if CR3 is updated (i.e., task switch), and OS=1; *PIP.NR=1 if destination is not root operation, and "Conceal VMX non-root operation from Intel PT" execution control = 0; *MODE.Exec if the mode has changed since the last MODE.Exec, or if no MODE.Exec since last PSB	PIP(new CR3, NR?), MODE.Exec?, TIP.PGE(BLIP)
10c	Far Branch (CALL/JMP/RET)	1	0	0		TIP.PGD()

**Table 36-50. Packet Generation under Different Enable Conditions (Contd.)**

Case	Operation	PktEn Before	PktEn After	CntxEn After	Other Dependencies	Packets Output
10d	Far Branch (CALL/JMP/RET)	1	0	1	*PIP if CR3 is updated (i.e., task switch), and OS=1; *PIP.NR=1 if destination is not root operation, and "Conceal VMX non-root operation from Intel PT" execution control = 0; *TraceStop if BLIP is in a TraceStop region	PIP(new CR3, NR?), TIP.PGD(BLIP), TraceStop?
10e	Far Branch (CALL/JMP/RET)	1	1	1	*PIP if CR3 is updated (i.e., task switch), and OS=1 *PIP.NR=1 if destination is not root operation, and "Conceal VMX non-root operation from Intel PT" execution control = 0; * MODE.Exec if the operation changes CS.L/D or IA32_EFER.LMA	PIP(NewCR3, NR?)?, MODE.Exec?, TIP(BLIP)
11a	HW Interrupt	0	0	0		None
11f	HW Interrupt	0	0	1	*PIP if CR3 is updated (i.e., task switch), and OS=1; *PIP.NR=1 if destination is not root operation, and "Conceal VMX non-root operation from Intel PT" execution control = 0; *TraceStop if BLIP is in a TraceStop region	PIP(new CR3, NR?), TraceStop?
11b	HW Interrupt	0	1	1	*PIP if CR3 is updated (i.e., task switch), and OS=1; *PIP.NR=1 if destination is not root operation, and "Conceal VMX non-root operation from Intel PT" execution control = 0; * MODE.Exec if the mode has changed since the last MODE.Exec, or if no MODE.Exec since last PSB	PIP(new CR3, NR?), MODE.Exec?, TIP.PGE(BLIP)
11c	HW Interrupt	1	0	0		FUP(NLIP), TIP.PGD()
11d	HW Interrupt	1	0	1	* PIP if CR3 is updated (i.e., task switch), and OS=1 *PIP.NR=1 if destination is not root operation, and "Conceal VMX non-root operation from Intel PT" execution control = 0; *TraceStop if BLIP is in a TraceStop region	FUP(NLIP), PIP(NewCR3, NR?)?, TIP.PGD(BLIP), TraceStop

Table 36-50. Packet Generation under Different Enable Conditions (Contd.)

Case	Operation	PktEn Before	PktEn After	CntxEn After	Other Dependencies	Packets Output
11e	HW Interrupt	1	1	1	* PIP if CR3 is updated (i.e., task switch), and OS=1 *PIP.NR=1 if destination is not root operation, and "Conceal VMX non-root operation from Intel PT" execution control = 0; * MODE.Exec if the operation changes CS.L/D or IA32_EFER.LMA	FUP(NLIP), PIP(NewCR3, NR?)?, MODE.Exec?, TIP(BLIP)
12a	SW Interrupt	0	0	0		None
12f	SW Interrupt	0	0	1	* PIP if CR3 is updated (i.e., task switch), and OS=1 *PIP.NR=1 if destination is not root operation, and "Conceal VMX non-root operation from Intel PT" execution control = 0; *TraceStop if BLIP is in a TraceStop region	PIP(NewCR3, NR?)?, TraceStop?
12b	SW Interrupt	0	1	1	* PIP if CR3 is updated (i.e., task switch), and OS=1 *PIP.NR=1 if destination is not root operation, and "Conceal VMX non-root operation from Intel PT" execution control = 0; *MODE.Exec if the mode has changed since the last MODE.Exec, or if no MODE.Exec since last PSB	PIP(NewCR3, NR?)?, MODE.Exec?, TIP.PGE(BLIP)
12c	SW Interrupt	1	0	0		FUP(CLIP), TIP.PGD()
12d	SW Interrupt	1	0	1	* PIP if CR3 is updated (i.e., task switch), and OS=1 *PIP.NR=1 if destination is not root operation, and "Conceal VMX non-root operation from Intel PT" execution control = 0; *TraceStop if BLIP is in a TraceStop region	FUP(CLIP), PIP(NewCR3, NR?)?, TIP.PGD(BLIP), TraceStop?
12e	SW Interrupt	1	1	1	* PIP if CR3 is updated (i.e., task switch), and OS=1 *PIP.NR=1 if destination is not root operation, and "Conceal VMX non-root operation from Intel PT" execution control = 0; * MODE.Exec if the operation changes CS.L/D or IA32_EFER.LMA	FUP(CLIP), PIP(NewCR3, NR?)?, FUP(NLIP), MODE.Exec?, TIP(BLIP)
13a	Exception/Fault	0	0	0		None

**Table 36-50. Packet Generation under Different Enable Conditions (Contd.)**

Case	Operation	PktEn Before	PktEn After	CntxEn After	Other Dependencies	Packets Output
13f	Exception/Fault	0	0	1	* PIP if CR3 is updated (i.e., task switch), and OS=1 *PIP.NR=1 if destination is not root operation, and "Conceal VMX non-root operation from Intel PT" execution control = 0; *TraceStop if BLIP is in a TraceStop region	PIP(NewCR3, NR?)?, TraceStop?
13b	Exception/Fault	0	1	1	* PIP if CR3 is updated (i.e., task switch), and OS=1 *PIP.NR=1 if destination is not root operation, and "Conceal VMX non-root operation from Intel PT" execution control = 0; *MODE.Exec if the mode has changed since the last MODE.Exec, or if no MODE.Exec since last PSB	PIP(NewCR3, NR?)?, MODE.Exec?, TIP.PGE(BLIP)
13c	Exception/Fault	1	0	0		FUP(CLIP), TIP.PGD()
13d	Exception/Fault	1	0	1	* PIP if CR3 is updated (i.e., task switch), and OS=1 *PIP.NR=1 if destination is not root operation, and "Conceal VMX non-root operation from Intel PT" execution control = 0; *TraceStop if BLIP is in a TraceStop region	FUP(CLIP), PIP(NewCR3, NR?)?, TIP.PGD(BLIP), TraceStop?
13e	Exception/Fault	1	1	1	* PIP if CR3 is updated (i.e., task switch), and OS=1 *PIP.NR=1 if destination is not root operation, and "Conceal VMX non-root operation from Intel PT" execution control = 0; * MODE.Exec if the operation changes CS.L/D or IA32_EFER.LMA	FUP(CLIP), PIP(NewCR3, NR?)?, MODE.Exec?, TIP(BLIP)
14a	SMI (TraceEn cleared)	0	0	D.C.		None
14b	SMI (TraceEn cleared)	1	0	0		FUP(SMRAM,LIP), TIP.PGD()
14f	SMI (TraceEn cleared)	1	0	1		NA
14c	SMI (TraceEn cleared)	1	1	1		NA
15a	RSM, TraceEn restored to 0	0	0	0		None
15b	RSM, TraceEn restored to 1	0	0	D.C.		See WRMSR cases for packets on enable

Table 36-50. Packet Generation under Different Enable Conditions (Contd.)

Case	Operation	PktEn Before	PktEn After	CntxEn After	Other Dependencies	Packets Output
15c	RSM, TraceEn restored to 1	0	1	1		See WRMSR cases for packets on enable. FUP/TIP.PGE IP is SMRAM.LIP
15e	RSM (TraceEn=1, goes to shutdown)	1	0	0		None
15f	RSM (TraceEn=1, goes to shutdown)	1	0	1		None
15d	RSM (TraceEn=1, goes to shutdown)	1	1	1		None
16i	Vmext	0	0	0		None
16a	Vmext	0	0	1	*PIP if OF=1, and "Conceal VM exits from Intel PT" execution control = 0; *TraceStop if VMCSH.LIP is in a TraceStop region	PIP(HostCR3, NR=0)?, TraceStop?
16b	VM exit, MSR list sets TraceEn=1	0	0	0		See WRMSR cases for packets on enable. FUP IP is VMCSH.LIP
16c	VM exit, MSR list sets TraceEn=1	0	1	1		See WRMSR cases for packets on enable. FUP/TIP.PGE IP is VMCSH.LIP
16e	VM exit	0	1	1	*PIP if OF=1, and "Conceal VM exits from Intel PT" execution control = 0; *MODE.Exec if the value is different, since last TIP.PGD	PIP(HostCR3, NR=0)?, MODE.Exec?, TIP.PGE(VMCSH.LIP)
16f	VM exit, MSR list clears TraceEn=0	1	0	0	*PIP if OF=1, and "Conceal VM exits from Intel PT" execution control = 0;	FUP(VMCSG.LIP), PIP(HostCR3, NR=0)?, TIP.PGD
16j	VM exit, ContextEN 1->0	1	0	0		FUP(VMCSG.LIP), TIP.PGD
16g	VM exit	1	0	1	*PIP if OF=1, and "Conceal VM exits from Intel PT" execution control = 0; *TraceStop if VMCSH.LIP is in a TraceStop region	FUP(VMCSG.LIP), PIP(HostCR3, NR=0)?, TIP.PGD(VMCSH.LIP), TraceStop?
16h	VM exit	1	1	1	*PIP if OF=1, and "Conceal VM exits from Intel PT" execution control = 0; *MODE.Exec if the value is different, since last TIP.PGD	FUP(VMCSG.LIP), PIP(HostCR3, NR=0)?, MODE.Exec, TIP(VMCSH.LIP)
17a	VM entry	0	0	0		None
17b	VM entry	0	0	1	*PIP if OF=1, and "Conceal VM entries from Intel PT" execution control = 0; *TraceStop if VMCSG.LIP is in a TraceStop region	PIP(GuestCR3, NR=1)?, TraceStop?
17c	VM entry, MSR load list sets TraceEn=1	0	0	1		See WRMSR cases for packets on enable. FUP IP is VMCSG.LIP

**Table 36-50. Packet Generation under Different Enable Conditions (Contd.)**

Case	Operation	PktEn Before	PktEn After	CntxEn After	Other Dependencies	Packets Output
17d	VM entry, MSR load list sets TraceEn=1	0	1	1		See WRMSR cases for packets on enable. FUP/TIP.PGE IP is VMCSg.LIP
17f	VM entry, FilterEN 0->1	0	1	1	*PIP if OF=1, and "Conceal VM entries from Intel PT" execution control = 0; *MODE.Exec if the value is different, since last TIP.PGD	PIP(GuestCR3, NR=1)?, MODE.Exec?, TIP.PGE(VMCSg.LIP)
17j	VM entry, ContextEN 0->1	0	1	1	*MODE.Exec if the value is different, since last TIP.PGD	MODE.Exec, TIP.PGE(VMCSg.LIP)
17g	VM entry, MSR list clears TraceEn=0	1	0	0	*PIP if OF=1, and "Conceal VM entries from Intel PT" execution control = 0;	PIP(GuestCR3, NR=1)?, TIP.PGD
17h	VM entry	1	0	1	*PIP if OF=1, and "Conceal VM entries from Intel PT" execution control = 0; *TraceStop if VMCSg.LIP is in a TraceStop region	PIP(GuestCR3, NR=1)?, TIP.PGD(VMCSg.LIP), TraceStop?
17i	VM entry	1	1	1	*PIP if OF=1, and "Conceal VM entries from Intel PT" execution control = 0; *MODE.Exec if the value is different, since last TIP.PGD	PIP(GuestCR3, NR=1)?, MODE.Exec, TIP(VMCSg.LIP)
20a	EENTER/ERESUME to non-debug enclave	0	0	0		None
20c	EENTER/ERESUME to non-debug enclave	1	0	0		FUP(CLIP), TIP.PGD()
21a	EEXIT from non-debug enclave	0	0	D.C.		None
21b	EEXIT from non-debug enclave	0	1	1	*MODE.Exec if the value is different, since last TIP.PGD	MODE.Exec?, TIP.PGE(BLIP)
22a	AEX/EEE from non-debug enclave	0	0	D.C.		None
22b	AEX/EEE from non-debug enclave	0	1	1	*MODE.Exec if the value is different, since last TIP.PGD	MODE.Exec?, TIP.PGE(AEP.LIP)
23a	EENTER/ERESUME to debug enclave	0	0	D.C.		None
23b	EENTER/ERESUME to debug enclave	0	1	1	*MODE.Exec if the value is different, since last TIP.PGD	MODE.Exec?, TIP.PGE(BLIP)
23c	EENTER/ERESUME to debug enclave	1	0	0		FUP(CLIP), TIP.PGD()
23d	EENTER/ERESUME to debug enclave	0	0	1	*TraceStop if BLIP is in a TraceStop region	FUP(CLIP), TIP.PGD(BLIP), TraceStop?
23e	EENTER/ERESUME to debug enclave	1	1	1		FUP(CLIP), TIP(BLIP)
24f	EEXIT from debug enclave	0	0	D.C.		None
24b	EEXIT from debug enclave	0	1	1	*MODE.Exec if the value is different, since last TIP.PGD	MODE.Exec?, TIP.PGE(BLIP)
24d	EEXIT from debug enclave	1	0	1	*TraceStop if BLIP is in a TraceStop region	FUP(CLIP), TIP.PGD(BLIP), TraceStop?



Table 36-50. Packet Generation under Different Enable Conditions (Contd.)

Case	Operation	PktEn Before	PktEn After	CntxEn After	Other Dependencies	Packets Output
24e	EEXIT from debug enclave	1	1	1		FUP(CLIP), TIP(BLIP)
25a	AEX/EEE from debug enclave	0	0	D.C.		None
25b	AEX/EEE from debug enclave	0	1	1	*MODE.Exec if the value is different, since last TIP.PGD	MODE.Exec?, TIP.PGE(AEP.LIP)
25d	AEX/EEE from debug enclave	1	0	1	*For AEX, FUP IP could be NLIP, for trap-like events	FUP(CLIP), TIP.PGD(AEP.LIP)
25e	AEX/EEE from debug enclave	1	1	1	*MODE.Exec if the value is different, since last TIP.PGD *For AEX, FUP IP could be NLIP, for trap-like events	FUP(CLIP), MODE.Exec?, TIP(AEP.LIP)
26a	XBEGIN/XACQUIRE	0	0	D.C.		None
26d	XBEGIN/XACQUIRE that does not set InTX	1	1	1		None
26e	XBEGIN/XACQUIRE that sets InTX	1	1	1		MODE(InTX=1, TXAbort=0), FUP(CLIP)
27a	XEND/XRELEASE	0	0	D.C.		None
27d	XEND/XRELEASE that does not clear InTX	1	1	1		None
27e	XEND/XRELEASE that clears InTX	1	1	1		MODE(InTX=0, TXAbort=0), FUP(CLIP)
28a	XABORT(Async XAbort, or other)	0	0	0		None
28e	XABORT(Async XAbort, or other)	0	0	1	*TraceStop if BLIP is in a TraceStop region	MODE(InTX=0, TXAbort=1), TraceStop?
28b	XABORT(Async XAbort, or other)	0	1	1		MODE(InTX=0, TXAbort=1), TIP.PGE(BLIP)
28c	XABORT(Async XAbort, or other)	1	0	1	*TraceStop if BLIP is in a TraceStop region	MODE(InTX=0, TXAbort=1), TIP.PGD (BLIP), TraceStop?
28d	XABORT(Async XAbort, or other)	1	1	1		MODE(InTX=0, TXAbort=1), FUP(CLIP), TIP(BLIP)
30a	INIT (BSP)	0	0	0		None
30b	INIT (BSP)	0	0	1	*TraceStop if RESET.LIP is in a TraceStop region	BIP(0), TraceStop?
30c	INIT (BSP)	0	1	1	* MODE.Exec if the value is different, since last TIP.PGD	MODE.Exec?, PIP(0), TIP.PGE(ResetLIP)
30d	INIT (BSP)	1	0	0		FUP(NLIP), TIP.PGD()
30e	INIT (BSP)	1	0	1	* PIP if OS=1 *TraceStop if RESET.LIP is in a TraceStop region	FUP(NLIP), PIP(0), TIP.PGD, TraceStop?

**Table 36-50. Packet Generation under Different Enable Conditions (Contd.)**

Case	Operation	PktEn Before	PktEn After	CntxEn After	Other Dependencies	Packets Output
30f	INIT (BSP)	1	1	1	* MODE.Exec if the mode has changed since the last MODE.Exec, or if no MODE.Exec since last PSB * PIP if OS=1	FUP(NLIP), PIP(0)?, MODE.Exec?, TIP(ResetLIP)
31a	INIT (AP, goes to wait-for-SIPI)	0	D.C.	D.C.		None
31b	INIT (AP, goes to wait-for-SIPI)	1	D.C.	D.C.	* PIP if OS=1	FUP(NLIP), PIP(0)
32a	SIPI	0	0	0		None
32c	SIPI	0	1	1	* MODE.Exec if the mode has changed since the last MODE.Exec, or if no MODE.Exec since last PSB	MODE.Exec?, TIP.PGE(SIPI-LIP)
32d	SIPI	1	0	0		TIP.PGD
32e	SIPI	1	0	1	*TraceStop if SIPI LIP is in a TraceStop region	TIP.PGD(SIPI LIP); TraceStop?
32f	SIPI	1	1	1	* MODE.Exec if the mode has changed since the last MODE.Exec, or if no MODE.Exec since last PSB	MODE.Exec?, TIP(SIPI LIP)
33a	MWAIT (to C0)	D.C.	D.C.	D.C.		None
33b	MWAIT (to higher-numbered C-State, packet sent on wake)	D.C.	D.C.	D.C.	*TSC if TSCEn=1 *TMA if TSCEn=MTCEn=1	TSC?, TMA?, CBR

In Table 36-52, PktEn is evaluated based on (TiggerEn & ContextEn & FilterEn & BranchEn & PwrEvtEn).

**Table 36-51. PwrEvtEn and PTWEn Packet Generation under Different Enable Conditions**

Case	Operation	PktEn Before	PktEn After	CntxEn After	Other Dependencies	Packets Output
16.1	MWAIT or I/O redir to MWAIT, gets #UD or #GP fault	dc	dc	dc		None
16.2	MWAIT or I/O redir to MWAIT, VM exits	dc	dc	dc		See VM exit examples (16[a-z] in Table 36-50) for BranchEn packets.
16.3	MWAIT or I/O redir to MWAIT, requests C0, or monitor not armed, or VMX virtual-interrupt delivery	dc	dc	dc		None
16.4a	MWAIT(X) or I/O redir to MWAIT, goes to C-state Y (Y>0)	dc	0	0		PWRE(Cx), EXSTOP
16.4b	MWAIT(X) or I/O redir to MWAIT, goes to C-state Y (Y>0)	dc	dc	1		MWAIT(Cy), PWRE(Cx), EXSTOP(IP), FUP(CLIP)
16.5a	MWAIT(X) or I/O redir to MWAIT, Pending event after resolving to go to C-state Y (Y>0)	dc	0	0	* TSC if TSCEn=1 * TMA if TSCEn=MTCEn=1	PWRE(Cx), EXSTOP, TSC?, TMA?, CBR, PWRX(LCC, DCC, 0)
16.5b	MWAIT(X) or I/O redir to MWAIT, Pending event after resolving to go to C-state Y (Y>0)	dc	dc	1	* TSC if TSCEn=1 * TMA if TSCEn=MTCEn=1	PWRE(Cx), EXSTOP(IP), FUP(CLIP), TSC?, TMA?, CBR, PWRX(LCC, DCC, 0)

Table 36-51. PwrEvtEn and PTWEn Packet Generation under Different Enable Conditions (Contd.)

Case	Operation	PktEn Before	PktEn After	CntxEn After	Other Dependencies	Packets Output
16.6a	MWAIT(5) or I/O redir to MWAIT, other thread(s) in core in C0/C1	dc	0	0		PWRE(C1), EXSTOP
16.6b	MWAIT(5) or I/O redir to MWAIT, other thread(s) in core in C0/C1	dc	dc	1		MWAIT(5), PWRE(C1), EXSTOP(IP), FUP(CLIP)
16.9a	HLT, Triple-fault shutdown, #MC with CR4.MCE=0, RSM to Cx (x>0)	dc	0	0		PWRE(C1), EXSTOP
16.9b	HLT, Triple-fault shutdown, #MC with CR4.MCE=1, RSM to Cx (x>0)	dc	dc			PWRE(C1), EXSTOP(IP), FUP(CLIP)
16.10a	VMX abort	dc	0	0		See "VMX Abort" (cases 16* and 18* in Table 36-50) for BranchEn packets that precede  PWRE(C1), EXSTOP
16.10b	VMX abort	dc	dc	1		See "VMX Abort" (cases 16* and 18* in Table 36-50) for BranchEn packets that precede  PWRE(C1), EXSTOP(IP), FUP(CLIP)
16.11a	RSM to Shutdown	dc	0	0		See "RSM to Shutdown" (cases 15[def] in Table 36-50) for BranchEn packets that precede  PWRE(C1), EXSTOP
16.11b	RSM to Shutdown	dc	dc	1		See "RSM to Shutdown" (cases 15[def] in Table 36-50) for BranchEn packets that precede  PWRE(C1), EXSTOP(IP), FUP(CLIP)
16.12a	INIT (BSP)	dc	0	0		See "INIT (BSP)" (cases 30[a-z] in Table 36-50) for packets that BranchEn precede  PWRE(C1), EXSTOP
16.12b	INIT (BSP)	dc	dc	1		See "INIT (BSP)" (cases 30[a-z] in Table 36-50) for packets that BranchEn precede  PWRE(C1), EXSTOP(IP), FUP(NLIP)

**Table 36-51. PwrEvtEn and PTWEn Packet Generation under Different Enable Conditions (Contd.)**

Case	Operation	PktEn Before	PktEn After	CntxEn After	Other Dependencies	Packets Output
16.13a	INIT (AP, goes to Wait-for-SIPI)	dc	0	0		See "INIT (AP, goes to Wait-for-SIPI)" (cases 31[a-z] in Table 36-50) for BranchEn packets that precede  PWRE(C1), EXSTOP
16.13b	INIT (AP, goes to Wait-for-SIPI)	dc	dc	1		See "INIT (AP, goes to Wait-for-SIPI)" (cases 31[a-z] in Table 36-50) for BranchEn packets that precede  PWRE(C1), EXSTOP(IP), FUP(NLIP)
16.14a	Hardware Duty Cycling (HDC)	dc	0	0	* TSC if TSCEn=1 * TMA if TSCEn=MTCEn=1	PWRE(HW, C6), EXSTOP, TSC?, TMA?, CBR, PWRX(CC6, CC6, 0x8)
16.14b	Hardware Duty Cycling (HDC)	dc	dc	1	* TSC if TSCEn=1 * TMA if TSCEn=MTCEn=1	PWRE(HW, C6), EXSTOP(IP), FUP(NLIP), TSC?, TMA?, CBR, PWRX(CC6, CC6, 0x8)
16.15a	VM entry to HLT or Shutdown	dc	0	0		See "VM entry" (cases 17[a-z] in Table 36-50) for BranchEn packets that precede.  PWRE(C1), EXSTOP
16.15b	VM entry to HLT or Shutdown	dc	dc	1		See "VM entry" (cases 17[a-z] in Table 36-50) for BranchEn packets that precede.  PWRE(C1), EXSTOP(IP), FUP(CLIP)
16.16a	EIST in C0, S1/TM1/TM2, or STP-CLK#	dc	0	0	* TSC if TSCEn=1 * TMA if TSCEn=MTCEn=1	EXSTOP, TSC?, TMA?, CBR
16.16b	EIST in C0, S1/TM1/TM2, or STP-CLK#	dc	dc	1	* TSC if TSCEn=1 * TMA if TSCEn=MTCEn=1	EXSTOP(IP), FUP(NLIP), TSC?, TMA?, CBR
16.17	EIST in Cx (x>0)	dc	dc	dc		None
16.18	INTR during Cx (x>0)	dc	dc	dc	* TSC if TSCEn=1 * TMA if TSCEn=MTCEn=1	TSC?, TMA?, CBR, PWRX(LCC, DCC, 0x1)  See "HW Interrupt" (cases 11[a-z] in Table 36-50) for BranchEn packets that follow.

Table 36-51. PwrEvtEn and PTWEn Packet Generation under Different Enable Conditions (Contd.)

Case	Operation	PktEn Before	PktEn After	CntxEEn After	Other Dependencies	Packets Output
16.18	SMI during Cx (x>0)	dc	dc	dc	* TSC if TSCEn=1 * TMA if TSCEn=MTCEEn=1	TSC?, TMA?, CBR, PWRX(LCC, DCC, 0)  See “HW Interrupt” (cases 14[a-z] in Table 36-50) for BranchEn packets that follow.
16.19	NMI during Cx (x>0)	dc	dc	dc	* TSC if TSCEn=1 * TMA if TSCEn=MTCEEn=1	TSC?, TMA?, CBR, PWRX(LCC, DCC, 0)  See “HW Interrupt” (cases 11[a-z] in Table 36-50) for BranchEn packets that follow.
16.2	Store to monitored address during Cx (x>0)	dc	dc	dc	* TSC if TSCEn=1 * TMA if TSCEn=MTCEEn=1	TSC?, TMA?, CBR, PWRX(LCC, DCC, 0x4)
16.22	#MC, IERR, TSC deadline timer expiration, or APIC counter under-flow during Cx (x>0)	dc	dc	dc	* TSC if TSCEn=1 * TMA if TSCEn=MTCEEn=1	TSC?, TMA?, CBR, PWRX(LCC, DCC, 0)

In Table 36-52, PktEn is evaluated based on (TiggerEn & ContextEn & FilterEn & BranchEn & PTWEn).

**Table 36-52. PwrEvtEn and PTWEn Packet Generation under Different Enable Conditions**

Case	Operation	PktEn Before	PktEn After	CntxEn After	Other Dependencies	Packets Output
16.24a	PTWRITE rm32/64, no fault	dc	dc	dc		None
16.24b	PTWRITE rm32/64, no fault	dc	0	0		None
16.24d	PTWRITE rm32, no fault	dc	1	1	* FUP, IP=1 if FUPonPTW=1	PTW(IP=1?, 4B, rm32_value), FUP(CLIP)?
16.24e	PTWRITE rm64, no fault	dc	1	1	* FUP, IP=1 if FUPonPTW=1	PTW(IP=1?, 8B, rm64_value), FUP(CLIP)?
16.25a	PTWRITE mem32/64, fault	dc	dc	dc		See "Exception/fault" (cases 13[a-z] in Table 36-50) for BranchEn packets.

## 36.8 SOFTWARE CONSIDERATIONS

### 36.8.1 Tracing SMM Code

Nothing prevents an SMM handler from configuring and enabling packet generation for its own use. As described in Section , SMI will always clear TraceEn, so the SMM handler would have to set TraceEn in order to enable tracing. There are some unique aspects and guidelines involved with tracing SMM code, which follows:

1. SMM should save away the existing values of any configuration MSRs that SMM intends to modify for tracing. This will allow the non-SMM tracing context to be restored before RSM.
2. It is recommended that SMM wait until it sets CSbase to 0 before enabling packet generation, to avoid possible LIP vs RIP confusion.
3. Packet output cannot be directed to SMRR memory, even while tracing in SMM.
4. Before performing RSM, SMM should take care to restore modified configuration MSRs to the values they had immediately after #SMI. This involves first disabling packet generation by clearing TraceEn, then restoring any other configuration MSRs that were modified.
5. RSM
  - Software must ensure that TraceEn=0 at the time of RSM. Tracing RSM is not a supported usage model, and the packets generated by RSM are undefined.
  - For processors on which Intel PT and LBR use are mutually exclusive (see Section 36.3.1.2), any RSM during which TraceEn is restored to 1 will suspend any LBR or BTS logging.

### 36.8.2 Cooperative Transition of Multiple Trace Collection Agents

A third-party trace-collection tool should take into consideration the fact that it may be deployed on a processor that supports Intel PT but may run under any operating system.

In such a deployment scenario, Intel recommends that tool agents follow similar principles of cooperative transition of single-use hardware resources, similar to how performance monitoring tools handle performance monitoring hardware:

- Respect the "in-use" ownership of an agent who already configured the trace configuration MSRs, see architectural MSRs with the prefix "IA32\_RTIT\_" in Chapter 35, "Model-Specific Registers (MSRs)", where "in-use" can be determined by reading the "enable bits" in the configuration MSRs.

- Relinquish ownership of the trace configuration MSRs by clearing the “enabled bits” of those configuration MSRs.

### 36.8.3 Tracking Time

This section describes the relationships of several clock counters whose update frequencies reside in different domains that feed into the timing packets. To track time, the decoder also needs to know the regularity or irregularity of the occurrences of various timing packets that store those clock counters.

Intel PT provides time information for three different but related domains:

- Processor timestamp counter

This counter increments at the max non-turbo or P1 frequency, and its value is returned on a RDTSC. Its frequency is fixed. The TSC packet holds the lower 7 bytes of the timestamp counter value. The TSC packet occurs occasionally and are much less frequent than the frequency of the time stamp counter. The timestamp counter will continue to increment when the processor is in deep C-States, with the exception of processors reporting CPUID.80000007H:EDX.InvariantTSC[bit 8] =0.

- Core crystal clock

The ratio of the core crystal clock to timestamp counter frequency is known as P, and can calculating CPUID.15H:EBX[31:0] / CPUID.15H:EAX[31:0]. The frequency of the core crystal clock is fixed and lower than that of the timestamp counter. The periodic MTC packet is generated based on software-selected multiples of the crystal clock frequency. The MTC packet is expected to occur more frequently than the TSC packet.

- Processor core clock

The processor core clock frequency can vary due to P-state and thermal conditions. The CYC packet provides elapsed time as measured in processor core clock cycles relative to the last CYC packet.

A decoder can use all or some combination of these packets to track time at different resolutions throughout the trace packets.

#### 36.8.3.1 Time Domain Relationships

The three domains are related by the following formula:

$$\text{TimeStampValue} = (\text{CoreCrystalClockValue} * P) + \text{AdjustedProcessorCycles} + \text{Software\_Offset};$$

The CoreCrystalClockValue can provide the coarse-grained component of the TSC value. P, or the TSC/“core crystal clock” ratio, can be derived from CPUID leaf 15H, as described in Section 36.8.3.

The AdjustedProcessorCycles component provides the fine-grained distance from the rising edge of the last core crystal clock. Specifically, it is a cycle count in the same frequency as the timestamp counter from the last crystal clock rising edge. The value is adjusted based on the ratio of the processor core clock frequency to the Maximum Non-Turbo (or P1) frequency.

The Software\_Offsets component includes software offsets that are factored into the timestamp value, such as IA32\_TSC\_ADJUST.

#### 36.8.3.2 Estimating TSC within Intel PT

For many usages, it may be useful to have an estimated timestamp value for all points in the trace. The formula provided in Section 36.8.3.1 above provides the framework for how such an estimate can be calculated from the various timing packets present in the trace.

The TSC packet provides the precise timestamp value at the time it is generated; however, TSC packets are infrequent, and estimates of the current timestamp value based purely on TSC packets are likely to be very inaccurate for this reason. In order to get more precise timing information between TSC packets, CYC packets and/or MTC packets should be enabled.

MTC packets provide incremental updates of the CoreCrystalClockValue. On processors that support CPUID leaf 15H, the frequency of the timestamp counter and the core crystal clock is fixed, thus MTC packets provide a means to update the running timestamp estimate. Between two MTC packets A and B, the number of crystal clock cycles passed is calculated from the 8-bit payloads of respective MTC packets:

$(CTC_B - CTC_A)$ , where  $CTC_i = MTC_i[15:8] \ll IA32\_RTIT\_CTL.MTCFreq$  and  $i = A, B$ .

The time from a TSC packet to the subsequent MTC packet can be calculated using the TMA packet that follows the TSC packet. The TMA packet provides both the crystal clock value (lower 16 bits, in the CTC field) and the AdjustedProcessorCycles value (in the FastCounter field) that can be used in the calculation of the corresponding core crystal clock value of the TSC packet.

When the next MTC after a pair of TSC/TMA is seen, the number of crystal clocks passed since the TSC packet can be calculated by subtracting the TMA.CTC value from the time indicated by the  $MTC_{Next}$  packet by

$CTC_{Delta}[15:0] = (CTC_{Next}[15:0] - TMA.CTC[15:0])$ , where  $CTC_{Next} = MTC_{Payload} \ll IA32\_RTIT\_CTL.MTCFreq$ .

The TMA.FastCounter field provides the fractional component of the TSC packet into the next crystal clock cycle.

CYC packets can provide further precision of an estimated timestamp value to many non-timing packets, by providing an indication of the time passed between other timing packets (MTCs or TSCs).

When enabled, CYC packets are sent preceding each CYC-eligible packet, and provide the number of processor core clock cycles that have passed since the last CYC packet. Thus between MTCs and TSCs, the accumulated CYC values can be used to estimate the adjusted\_processor\_cycles component of the timestamp value. The accumulated CPU cycles will have to be adjusted to account for the difference in frequency between the processor core clock and the P1 frequency. The necessary adjustment can be estimated using the core:bus ratio value given in the CBR packet, by multiplying the accumulated cycle count value by  $P1/CBR_{payload}$ .

A greater level of precision may be achieved by calculating the CPU clock frequency, see Section 36.8.3.4 below for a method to do so using Intel PT packets.

CYCs can be used to estimate time between TSCs even without MTCs, though this will likely result in a reduction in estimated TSC precision.

### 36.8.3.3 VMX TSC Manipulation

When software executes in non-Root operation, additional offset and scaling factors may be applied to the TSC value. These are optional, but may be enabled via VMCS controls on a per-VM basis. See Chapter 25, "VMX Non-Root Operation" for details on VMX TSC offsetting and TSC scaling.

Like the value returned by RDTSC, TSC packets will include these adjustments, but other timing packets (such as MTC, CYC, and CBR) are not impacted. In order to use the algorithm above to estimate the TSC value when TSC scaling is in use, it will be necessary for software to account for the scaling factor. See Section 36.5.2.6 for details.

### 36.8.3.4 Calculating Frequency with Intel PT

Because Intel PT can provide both wall-clock time and processor clock cycle time, it can be used to measure the processor core clock frequency. Either TSC or MTC packets can be used to track the wall-clock time. By using CYC packets to count the number of processor core cycles that pass in between a pair of wall-clock time packets, the ratio between processor core clock frequency and TSC frequency can be derived. If the P1 frequency is known, it can be applied to determine the CPU frequency. See Section 36.8.3.1 above for details on the relationship between TSC, MTC, and CYC.