# Intel® 64 and IA-32 Architectures Software Developer's Manual

## Documentation Changes

December 2009

Document Number: 252046-026

# Contents

# Revision History

| Revision | Description | Date |
|---|---|---|
| -001 | • Initial release | November 2002 |
| -002 | • Added 1-10 Documentation Changes.<br>• Removed old Documentation Changes items that already have been incorporated in the published Software Developer's manual | December 2002 |
| -003 | • Added 9 -17 Documentation Changes.<br>• Removed Documentation Change #6 - References to bits Gen and Len Deleted.<br>• Removed Documentation Change #4 - VIF Information Added to CLI Discussion | February 2003 |
| -004 | • Removed Documentation changes 1-17.<br>• Added Documentation changes 1-24. | June 2003 |
| -005 | • Removed Documentation Changes 1-24.<br>• Added Documentation Changes 1-15. | September 2003 |
| -006 | • Added Documentation Changes 16- 34. | November 2003 |
| -007 | • Updated Documentation changes 14, 16, 17, and 28.<br>• Added Documentation Changes 35-45. | January 2004 |
| -008 | • Removed Documentation Changes 1-45.<br>• Added Documentation Changes 1-5. | March 2004 |
| -009 | • Added Documentation Changes 7-27. | May 2004 |
| -010 | • Removed Documentation Changes 1-27.<br>• Added Documentation Changes 1. | August 2004 |
| -011 | • Added Documentation Changes 2-28. | November 2004 |
| -012 | • Removed Documentation Changes 1-28.<br>• Added Documentation Changes 1-16. | March 2005 |
| -013 | • Updated title.<br>• There are no Documentation Changes for this revision of the document. | July 2005 |
| -014 | • Added Documentation Changes 1-21. | September 2005 |
| -015 | • Removed Documentation Changes 1-21.<br>• Added Documentation Changes 1-20. | March 9, 2006 |
| -016 | • Added Documentation changes 21-23. | March 27, 2006 |
| -017 | • Removed Documentation Changes 1-23.<br>• Added Documentation Changes 1-36. | September 2006 |
| -018 | • Added Documentation Changes 37-42. | October 2006 |
| -019 | • Removed Documentation Changes 1-42.<br>• Added Documentation Changes 1-19. | March 2007 |
| -020 | • Added Documentation Changes 20-27. | May 2007 |
| -021 | • Removed Documentation Changes 1-27.<br>• Added Documentation Changes 1-6 | November 2007 |
| -022 | • Removed Documentation Changes 1-6<br>• Added Documentation Changes 1-6 | August 2008 |
| -023 | • Removed Documentation Changes 1-6<br>• Added Documentation Changes 1-21 | March 2009 |

| Revision | Description | Date |
|---|---|---|
| -024 | • Removed Documentation Changes 1-21<br>• Added Documentation Changes 1-16 | June 2009 |
| -025 | • Removed Documentation Changes 1-16<br>• Added Documentation Changes 1-18 | September 2009 |
| -026 | • Removed Documentation Changes 1-18<br>• Added Documentation Changes 1-15 | December 2009 |

§

# 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 |
|---|---|
| *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture* | 253665 |
| *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A: Instruction Set Reference, A-M* | 253666 |
| *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B: Instruction Set Reference, N-Z* | 253667 |
| *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A: System Programming Guide, Part 1* | 253668 |
| *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B: System Programming Guide, Part 2* | 253669 |

## 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 3, Volume 2A |
| 2 | Updates to Chapter 4, Volume 2B |
| 3 | Updates to Chapter 4, Volume 3A |
| 4 | Updates to Chapter 5, Volume 3A |
| 5 | Updates to Chapter 8, Volume 3A |
| 6 | Updates to Chapter 10, Volume 3A |
| 7 | Updates to Chapter 15, Volume 3A |
| 8 | Updates to Chapter 21, Volume 3B |
| 9 | Updates to Chapter 22, Volume 3B |
| 10 | Updates to Chapter 25, Volume 3B |
| 11 | Updates to Chapter 27, Volume 3B |
| 12 | Updates to Chapter 30, Volume 3B |
| 13 | Updates to Appendix A, Volume 3B |
| 14 | Updates to Appendix B, Volume 3B |
| 15 | Updates to Appendix G, Volume 3B |

# Documentation Changes

## 1. 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-M.

------------------------------------------------------------------------------------------

...

### 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-bit Mode** | **Compat/ Leg Mode** | **Description** |
|---|---|---|---|---|---|
| F5 | CMC | A | Valid | Valid | Complement carry flag. |

**Instruction Operand Encoding**

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | NA | NA | NA | NA |

...

#### 3.1.1.3 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 definition table is organized according to the order of operand in Intel assembly syntax. The encoding method for each operand in the instruction byte stream is expressed via modR/M:reg, modR/M:r/m, imm8/16/32/64, etc (cross reference).

**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.

### 3.1.1.4 64-bit Mode Column in the Instruction Summary Table

The "64-bit Mode" column indicates whether the opcode sequence is supported in 64-bit mode. The column uses the following notation:

- **Valid** — Supported.
- **Invalid** — 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.

...

## AAA—ASCII Adjust After Addition

| Opcode | Instruction | Op/ En | 64-bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 37 | AAA | A | Invalid | Valid | ASCII adjust AL after addition. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | NA | NA | NA | NA |

...

### 64-Bit Mode Exceptions

| | |
|---|---|
| #UD | If in 64-bit mode. |

...

## AAD—ASCII Adjust AX Before Division

| Opcode | Instruction | Op/ En | 64-bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| D5 0A | AAD | A | Invalid | Valid | ASCII adjust AX before division. |
| D5 *ib* | (No mnemonic) | A | Invalid | Valid | Adjust AX before division to number base *imm8*. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | NA | NA | NA | NA |

...

## AAM—ASCII Adjust AX After Multiply

| Opcode | Instruction | Op/ En | 64-bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| D4 0A | AAM | A | Invalid | Valid | ASCII adjust AX after multiply. |
| D4 *ib* | (No mnemonic) | A | Invalid | Valid | Adjust AX after multiply to number base *imm8*. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | NA | NA | NA | NA |

...

## AAS—ASCII Adjust AL After Subtraction

| **Opcode** | **Instruction** | **Op/ En** | **64-bit Mode** | **Compat/ Leg Mode** | **Description** |
|---|---|---|---|---|---|
| 3F | AAS | A | Invalid | Valid | ASCII adjust AL after subtraction. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | NA | NA | NA | NA |

...

## ADC—Add with Carry

| **Opcode** | **Instruction** | **Op/ En** | **64-bit Mode** | **Compat/ Leg Mode** | **Description** |
|---|---|---|---|---|---|
| 14 *ib* | ADC AL, *imm8* | C | Valid | Valid | Add with carry *imm8* to AL. |
| 15 *iw* | ADC AX, *imm16* | C | Valid | Valid | Add with carry *imm16* to AX. |
| 15 *id* | ADC EAX, *imm32* | C | Valid | Valid | Add with carry *imm32* to EAX. |
| REX.W + 15 *id* | ADC RAX, *imm32* | C | Valid | N.E. | Add with carry *imm32 sign extended to 64-bits* to RAX. |
| 80 /2 *ib* | ADC *r/m8*, *imm8* | B | Valid | Valid | Add with carry *imm8* to *r/m8.* |
| REX + 80 /2 *ib* | ADC *r/m8**, *imm8* | B | Valid | N.E. | Add with carry *imm8* to *r/m8.* |
| 81 /2 *iw* | ADC *r/m16*, *imm16* | B | Valid | Valid | Add with carry *imm16* to *r/m16.* |
| 81 /2 *id* | ADC *r/m32*, *imm32* | B | Valid | Valid | Add with CF *imm32* to *r/m32.* |
| REX.W + 81 /2 *id* | ADC *r/m64*, *imm32* | B | Valid | N.E. | Add with CF *imm32* sign extended to 64-bits to *r/m64.* |
| 83 /2 *ib* | ADC *r/m16*, *imm8* | B | Valid | Valid | Add with CF sign-extended *imm8* to *r/m16.* |
| 83 /2 *ib* | ADC *r/m32*, *imm8* | B | Valid | Valid | Add with CF sign-extended *imm8* into *r/m32.* |
| REX.W + 83 /2 *ib* | ADC *r/m64*, *imm8* | B | Valid | N.E. | Add with CF sign-extended *imm8* into *r/m64.* |
| 10 /*r* | ADC *r/m8*, *r8* | A | Valid | Valid | Add with carry byte register to *r/m8.* |
| REX + 10 /*r* | ADC *r/m8**, *r8** | A | Valid | N.E. | Add with carry byte register to *r/m64.* |
| 11 /*r* | ADC *r/m16*, *r16* | A | Valid | Valid | Add with carry *r16* to *r/m16.* |
| 11 /*r* | ADC *r/m32*, *r32* | A | Valid | Valid | Add with CF *r32* to *r/m32.* |

| Opcode | Instruction | Op/ En | 64-bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| REX.W + 11 /*r* | ADC *r/m64, r64* | A | Valid | N.E. | Add with CF *r64* to *r/m64.* |
| 12 /*r* | ADC *r8, r/m8* | A | Valid | Valid | Add with carry *r/m8* to byte register. |
| REX + 12 /*r* | ADC *r8**, *r/m8** | A | Valid | N.E. | Add with carry *r/m64* to byte register. |
| 13 /*r* | ADC *r16, r/m16* | A | Valid | Valid | Add with carry *r/m16* to *r16.* |
| 13 /*r* | ADC *r32, r/m32* | A | Valid | Valid | Add with CF *r/m32* to *r32.* |
| REX.W + 13 /*r* | ADC *r64, r/m64* | A | Valid | N.E. | Add with CF *r/m64* to *r64.* |

**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 |
|---|---|---|---|---|
| A | ModRM:r/m (r, w) | ModRM:reg (r) | NA | NA |
| B | ModRM:r/m (r, w) | imm8 | NA | NA |
| C | AL/AX/EAX/RAX | imm8 | NA | NA |

...

## ADD—Add

| Opcode | Instruction | Op/ En | 64-bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 04 *ib* | ADD AL, *imm8* | C | Valid | Valid | Add *imm8* to AL. |
| 05 *iw* | ADD AX, *imm16* | C | Valid | Valid | Add *imm16* to AX. |
| 05 *id* | ADD EAX, *imm32* | C | Valid | Valid | Add *imm32* to EAX. |
| REX.W + 05 *id* | ADD RAX, *imm32* | C | Valid | N.E. | Add *imm32 sign-extended to 64-bits* to RAX. |
| 80 /0 *ib* | ADD *r/m8, imm8* | B | Valid | Valid | Add *imm8* to *r/m8.* |
| REX + 80 /0 *ib* | ADD *r/m8**, *imm8* | B | Valid | N.E. | Add *sign*-extended *imm8* to *r/m64.* |
| 81 /0 *iw* | ADD *r/m16, imm16* | B | Valid | Valid | Add *imm16* to *r/m16.* |
| 81 /0 *id* | ADD *r/m32, imm32* | B | Valid | Valid | Add *imm32* to *r/m32.* |
| REX.W + 81 /0 *id* | ADD *r/m64, imm32* | B | Valid | N.E. | Add *imm32 sign-extended to 64-bits* to *r/m64*. |
| 83 /0 *ib* | ADD *r/m16, imm8* | B | Valid | Valid | Add *sign*-extended *imm8* to *r/m16.* |
| 83 /0 *ib* | ADD *r/m32, imm8* | B | Valid | Valid | Add *sign*-extended *imm8* to *r/m32.* |

| Opcode | Instruction | Op/ En | 64-bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| REX.W + 83 /0 *ib* | ADD *r/m64, imm8* | B | Valid | N.E. | Add *sign*-extended *imm8* to *r/m64.* |
| 00 */r* | ADD *r/m8, r8* | A | Valid | Valid | Add *r8* to *r/m8.* |
| REX + 00 */r* | ADD *r/m8**, *r8** | A | Valid | N.E. | Add *r8* to *r/m8.* |
| 01 */r* | ADD *r/m16, r16* | A | Valid | Valid | Add *r16* to *r/m16.* |
| 01 */r* | ADD *r/m32, r32* | A | Valid | Valid | Add r32 to *r/m32.* |
| REX.W + 01 */r* | ADD *r/m64, r64* | A | Valid | N.E. | Add r64 to *r/m64.* |
| 02 */r* | ADD *r8, r/m8* | A | Valid | Valid | Add *r/m8* to *r8.* |
| REX + 02 */r* | ADD *r8**, *r/m8** | A | Valid | N.E. | Add *r/m8* to *r8.* |
| 03 */r* | ADD *r16, r/m16* | A | Valid | Valid | Add *r/m16* to *r16.* |
| 03 */r* | ADD *r32, r/m32* | A | Valid | Valid | Add *r/m32* to *r32.* |
| REX.W + 03 */r* | ADD *r64, r/m64* | A | Valid | N.E. | Add *r/m64* to *r64.* |

**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 |
|---|---|---|---|---|
| A | ModRM:reg (r, w) | ModRM:r/m (r) | NA | NA |
| B | ModRM:r/m (r, w) | imm8 | NA | NA |
| C | AL/AX/EAX/RAX | imm8 | NA | NA |

...

## ADDPD—Add Packed Double-Precision Floating-Point Values

| Opcode | Instruction | Op/ En | 64-bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 66 0F 58 /r | ADDPD *xmm1, xmm2/m128* | A | Valid | Valid | Add packed double-precision floating-point values from *xmm2/m128* to *xmm1*. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (r, w) | ModRM:r/m (r) | NA | NA |

...

## ADDPS—Add Packed Single-Precision Floating-Point Values

| Opcode | Instruction | Op/ En | 64-bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F 58 /r | ADDPS *xmm1, xmm2/m128* | A | Valid | Valid | Add packed single-precision floating-point values from *xmm2/m128* to *xmm1*. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (r, w) | ModRM:r/m (r) | NA | NA |

...

## ADDSD—Add Scalar Double-Precision Floating-Point Values

| Opcode | Instruction | Op/ En | 64-bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| F2 0F 58 /r | ADDSD *xmm1, xmm2/m64* | A | Valid | Valid | Add the low double-precision floating-point value from *xmm2/m64* to *xmm1*. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (r, w) | ModRM:r/m (r) | NA | NA |

...

## ADDSS—Add Scalar Single-Precision Floating-Point Values

| Opcode | Instruction | Op/ En | 64-bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| F3 0F 58 /r | ADDSS *xmm1, xmm2/m32* | A | Valid | Valid | Add the low single-precision floating-point value from *xmm2/m32* to *xmm1*. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (r, w) | ModRM:r/m (r) | NA | NA |

...

## ADDSUBPD—Packed Double-FP Add/Subtract

| Opcode | Instruction | Op/ En | 64-bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 66 0F D0 /r | ADDSUBPD *xmm1*, *xmm2/m128* | A | Valid | Valid | Add/subtract double-precision floating-point values from *xmm2/m128* to *xmm1*. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (r, w) | ModRM:r/m (r) | NA | NA |

...

## ADDSUBPS—Packed Single-FP Add/Subtract

| Opcode | Instruction | Op/ En | 64-bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| F2 0F D0 /r | ADDSUBPS *xmm1*, *xmm2/m128* | A | Valid | Valid | Add/subtract single-precision floating-point values from *xmm2/m128* to *xmm1*. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (r, w) | ModRM:r/m (r) | NA | NA |

...

## AND—Logical AND

| Opcode | Instruction | Op/ En | 64-bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 24 *ib* | AND AL, *imm8* | C | Valid | Valid | AL AND *imm8.* |
| 25 *iw* | AND AX, *imm16* | C | Valid | Valid | AX AND *imm16.* |
| 25 *id* | AND EAX, *imm32* | C | Valid | Valid | EAX AND *imm32.* |
| REX.W + 25 *id* | AND RAX, *imm32* | C | Valid | N.E. | RAX AND *imm32 sign-extended to 64-bits.* |
| 80 /4 *ib* | AND *r/m8, imm8* | B | Valid | Valid | *r/m8* AND *imm8.* |
| REX + 80 /4 *ib* | AND *r/m8**, *imm8* | B | Valid | N.E. | *r/m64* AND *imm8 (sign-extended).* |
| 81 /4 *iw* | AND *r/m16, imm16* | B | Valid | Valid | *r/m16* AND *imm16.* |
| 81 /4 *id* | AND *r/m32, imm32* | B | Valid | Valid | *r/m32* AND *imm32.* |
| REX.W + 81 /4 *id* | AND *r/m64, imm32* | B | Valid | N.E. | *r/m64* AND *imm32 sign extended to 64-bits.* |

| Opcode | Instruction | Op/ En | 64-bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 83 /4 *ib* | AND *r/m16, imm8* | B | Valid | Valid | *r/m16* AND *imm8 (sign-extended).* |
| 83 /4 *ib* | AND *r/m32, imm8* | B | Valid | Valid | *r/m32* AND *imm8 (sign-extended).* |
| REX.W + 83 /4 *ib* | AND *r/m64, imm8* | B | Valid | N.E. | *r/m64* AND *imm8 (sign-extended).* |
| 20 */r* | AND *r/m8, r8* | A | Valid | Valid | *r/m8* AND *r8.* |
| REX + 20 */r* | AND *r/m8**, *r8** | A | Valid | N.E. | *r/m64* AND *r8 (sign-extended).* |
| 21 */r* | AND *r/m16, r16* | A | Valid | Valid | *r/m16* AND *r16.* |
| 21 */r* | AND *r/m32, r32* | A | Valid | Valid | *r/m32* AND *r32.* |
| REX.W + 21 */r* | AND *r/m64, r64* | A | Valid | N.E. | *r/m64* AND *r32.* |
| 22 */r* | AND *r8, r/m8* | A | Valid | Valid | *r8* AND *r/m8.* |
| REX + 22 */r* | AND *r8**, *r/m8** | A | Valid | N.E. | *r/m64* AND *r8 (sign-extended).* |
| 23 */r* | AND *r16, r/m16* | A | Valid | Valid | *r16* AND *r/m16.* |
| 23 */r* | AND *r32, r/m32* | A | Valid | Valid | *r32* AND *r/m32.* |
| REX.W + 23 */r* | AND *r64, r/m64* | A | Valid | N.E. | *r64* AND *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 |
|---|---|---|---|---|
| A | ModRM:reg (r, w) | ModRM:r/m (r) | NA | NA |
| B | ModRM:r/m (r, w) | imm8 | NA | NA |
| C | AL/AX/EAX/RAX | imm8 | NA | NA |

...

## ANDPD—Bitwise Logical AND of Packed Double-Precision Floating-Point Values

| Opcode | Instruction | Op/ En | 64-bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 66 0F 54 /r | ANDPD *xmm1*, *xmm2/m128* | A | Valid | Valid | Bitwise logical AND of *xmm2/m128* and *xmm1*. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (r, w) | ModRM:r/m (r) | NA | NA |

...

## ANDPS—Bitwise Logical AND of Packed Single-Precision Floating-Point Values

| Opcode | Instruction | Op/ En | 64-bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F 54 /r | ANDPS *xmm1*, *xmm2/m128* | A | Valid | Valid | Bitwise logical AND of *xmm2/m128* and *xmm1*. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (r, w) | ModRM:r/m (r) | NA | NA |

...

## ANDNPD—Bitwise Logical AND NOT of Packed Double-Precision Floating-Point Values

| Opcode | Instruction | Op/ En | 64-bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 66 0F 55 /r | ANDNPD *xmm1*, *xmm2/m128* | A | Valid | Valid | Bitwise logical AND NOT of *xmm2/m128* and *xmm1*. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (r, w) | ModRM:r/m (r) | NA | NA |

...

## ANDNPS—Bitwise Logical AND NOT of Packed Single-Precision Floating-Point Values

| Opcode | Instruction | Op/ En | 64-bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F 55 /r | ANDNPS *xmm1*, *xmm2/m128* | A | Valid | Valid | Bitwise logical AND NOT of *xmm2/m128* and *xmm1*. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (r, w) | ModRM:r/m (r) | NA | NA |

...

## ARPL—Adjust RPL Field of Segment Selector

| Opcode | Instruction | Op/ En | 64-bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 63 /*r* | ARPL *r/m16, r16* | A | N. E. | Valid | Adjust RPL of *r/m16* to not less than RPL of *r16*. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:r/m (w) | ModRM:reg (r) | NA | NA |

### Description

Compares the RPL fields of two segment selectors. The first operand (the destination operand) contains one segment selector and the second operand (source operand) contains the other. (The RPL field is located in bits 0 and 1 of each operand.) If the RPL field of the destination operand is less than the RPL field of the source operand, the ZF flag is set and the RPL field of the destination operand is increased to match that of the source operand. Otherwise, the ZF flag is cleared and no change is made to the destination operand. (The destination operand can be a word register or a memory location; the source operand must be a word register.)

The ARPL instruction is provided for use by operating-system procedures (however, it can also be used by applications). It is generally used to adjust the RPL of a segment selector that has been passed to the operating system by an application program to match the privilege level of the application program. Here the segment selector passed to the operating system is placed in the destination operand and segment selector for the application program's code segment is placed in the source operand. (The RPL field in the source operand represents the privilege level of the application program.) Execution of the ARPL instruction then ensures that the RPL of the segment selector received by the operating system is no lower (does not have a higher privilege) than the privilege level of the application program (the segment selector for the application program's code segment can be read from the stack following a procedure call).

This instruction executes as described in compatibility mode and legacy mode. It is not encodable in 64-bit mode.

See “Checking Caller Access Privileges” in Chapter 3, “Protected-Mode Memory Management,” of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*, for more information about the use of this instruction.

...

## BLENDPD — Blend Packed Double Precision Floating-Point Values

| **Opcode** | **Instruction** | Op/ En | **64-bit Mode** | **Compat/ Leg Mode** | **Description** |
|---|---|---|---|---|---|
| 66 0F 3A 0D /r ib | BLENDPD *xmm1, xmm2/m128, imm8* | A | Valid | Valid | Select packed DP-FP values from *xmm1* and *xmm2/m128* from mask specified in imm8 and store the values into *xmm1*. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (r, w) | ModRM:r/m (r) | imm8 | NA |

...

## BLENDPS — Blend Packed Single Precision Floating-Point Values

| **Opcode** | **Instruction** | Op/ En | **64-bit Mode** | **Compat/ Leg Mode** | **Description** |
|---|---|---|---|---|---|
| 66 0F 3A 0C /r ib | BLENDPS *xmm1, xmm2/m128, imm8* | A | Valid | Valid | Select packed single precision floating-point values from *xmm1* and *xmm2/m128* from mask specified in *imm8* and store the values into *xmm1*. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (r, w) | ModRM:r/m (r) | imm8 | NA |

...

## BLENDVPD — Variable Blend Packed Double Precision Floating-Point Values

| **Opcode** | **Instruction** | Op/ En | **64-bit Mode** | **Compat/ Leg Mode** | **Description** |
|---|---|---|---|---|---|
| 66 0F 38 15 /r | BLENDVPD *xmm1, xmm2/m128* , *<XMM0>* | A | Valid | Valid | Select packed DP FP values from *xmm1* and *xmm2* from mask specified in *XMM0* and store the values in *xmm1*. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (r, w) | ModRM:r/m (r) | implicit XMM0 | NA |

...

## BLENDVPS — Variable Blend Packed Single Precision Floating-Point Values

| **Opcode** | **Instruction** | Op/ En | **64-bit Mode** | **Compat/ Leg Mode** | **Description** |
|---|---|---|---|---|---|
| 66 0F 38 14 /r | BLENDVPS *xmm1, xmm2/m128, <XMM0>* | A | Valid | Valid | Select packed single precision floating-point values from *xmm1* and *xmm2/m128* from mask specified in *XMM0* and store the values into *xmm1*. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (r, w) | ModRM:r/m (r) | implicit XMM0 | NA |

...

## BOUND—Check Array Index Against Bounds

| Opcode | Instruction | Op/ En | 64-bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 62 */r* | BOUND *r16, m16&16* | A | Invalid | Valid | Check if *r16* (array index) is within bounds specified by *m16&16.* |
| 62 */r* | BOUND *r32, m32&32* | A | Invalid | Valid | Check if *r32* (array index) is within bounds specified by *m16&16.* |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (r) | ModRM:r/m (r) | NA | NA |

...

## BSF—Bit Scan Forward

| Opcode | Instruction | Op/ En | 64-bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F BC /r | BSF *r16, r/m16* | A | Valid | Valid | Bit scan forward on *r/m16.* |
| 0F BC /r | BSF *r32, r/m32* | A | Valid | Valid | Bit scan forward on *r/m32.* |
| REX.W + 0F BC | BSF *r64, r/m64* | A | Valid | N.E. | Bit scan forward on *r/m64.* |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (w) | ModRM:r/m (r) | NA | NA |

...

## BSR—Bit Scan Reverse

| Opcode | Instruction | Op/ En | 64-bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F BD /r | BSR *r16, r/m16* | A | Valid | Valid | Bit scan reverse on *r/m16.* |
| 0F BD /r | BSR *r32, r/m32* | A | Valid | Valid | Bit scan reverse on *r/m32.* |
| REX.W + 0F BD | BSR *r64, r/m64* | A | Valid | N.E. | Bit scan reverse on *r/m64.* |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (w) | ModRM:r/m (r) | NA | NA |

...

## BSWAP—Byte Swap

| Opcode | Instruction | Op/ En | 64-bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F C8+*rd* | BSWAP *r32* | A | Valid* | Valid | Reverses the byte order of a 32-bit register. |
| REX.W + 0F C8+*rd* | BSWAP *r64* | A | Valid | N.E. | Reverses the byte order of a 64-bit register. |

**NOTES:**

* See IA-32 Architecture Compatibility section below.

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | reg (r, w) | NA | NA | NA |

...

## BT—Bit Test

| Opcode | Instruction | Op/ En | 64-bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F A3 | BT *r/m16, r16* | A | Valid | Valid | Store selected bit in CF flag. |
| 0F A3 | BT *r/m32, r32* | A | Valid | Valid | Store selected bit in CF flag. |
| REX.W + 0F A3 | BT *r/m64, r64* | A | Valid | N.E. | Store selected bit in CF flag. |
| 0F BA /4 *ib* | BT *r/m16, imm8* | B | Valid | Valid | Store selected bit in CF flag. |
| 0F BA /4 *ib* | BT *r/m32, imm8* | B | Valid | Valid | Store selected bit in CF flag. |
| REX.W + 0F BA /4 *ib* | BT *r/m64, imm8* | B | Valid | N.E. | Store selected bit in CF flag. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:r/m (r) | ModRM:reg (r) | NA | NA |
| B | ModRM:r/m (r) | imm8 | NA | NA |

...

## BTC—Bit Test and Complement

| **Opcode** | **Instruction** | **Op/ En** | **64-bit Mode** | **Compat/ Leg Mode** | **Description** |
|---|---|---|---|---|---|
| 0F BB | BTC *r/m16, r16* | A | Valid | Valid | Store selected bit in CF flag and complement. |
| 0F BB | BTC *r/m32, r32* | A | Valid | Valid | Store selected bit in CF flag and complement. |
| REX.W + 0F BB | BTC *r/m64, r64* | A | Valid | N.E. | Store selected bit in CF flag and complement. |
| 0F BA /7 *ib* | BTC *r/m16, imm8* | B | Valid | Valid | Store selected bit in CF flag and complement. |
| 0F BA /7 *ib* | BTC *r/m32, imm8* | B | Valid | Valid | Store selected bit in CF flag and complement. |
| REX.W + 0F BA /7 *ib* | BTC *r/m64, imm8* | B | Valid | N.E. | Store selected bit in CF flag and complement. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:r/m (r, w) | ModRM:reg (r) | NA | NA |
| B | ModRM:r/m (r, w) | imm8 | NA | NA |

...

## BTR—Bit Test and Reset

| **Opcode** | **Instruction** | **Op/ En** | **64-bit Mode** | **Compat/ Leg Mode** | **Description** |
|---|---|---|---|---|---|
| 0F B3 | BTR *r/m16, r16* | A | Valid | Valid | Store selected bit in CF flag and clear. |
| 0F B3 | BTR *r/m32, r32* | A | Valid | Valid | Store selected bit in CF flag and clear. |
| REX.W + 0F B3 | BTR *r/m64, r64* | A | Valid | N.E. | Store selected bit in CF flag and clear. |
| 0F BA /6 *ib* | BTR *r/m16, imm8* | B | Valid | Valid | Store selected bit in CF flag and clear. |
| 0F BA /6 *ib* | BTR *r/m32, imm8* | B | Valid | Valid | Store selected bit in CF flag and clear. |
| REX.W + 0F BA /6 *ib* | BTR *r/m64, imm8* | B | Valid | N.E. | Store selected bit in CF flag and clear. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:r/m (r, w) | ModRM:reg (r) | NA | NA |
| B | ModRM:r/m (r, w) | imm8 | NA | NA |

...

## BTS—Bit Test and Set

| Opcode | Instruction | Op/ En | 64-bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F AB | BTS *r/m16, r16* | A | Valid | Valid | Store selected bit in CF flag and set. |
| 0F AB | BTS *r/m32, r32* | A | Valid | Valid | Store selected bit in CF flag and set. |
| REX.W + 0F AB | BTS *r/m64, r64* | A | Valid | N.E. | Store selected bit in CF flag and set. |
| 0F BA /5 *ib* | BTS *r/m16, imm8* | B | Valid | Valid | Store selected bit in CF flag and set. |
| 0F BA /5 *ib* | BTS *r/m32, imm8* | B | Valid | Valid | Store selected bit in CF flag and set. |
| REX.W + 0F BA /5 *ib* | BTS *r/m64, imm8* | B | Valid | N.E. | Store selected bit in CF flag and set. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:r/m (r, w) | ModRM:reg (r) | NA | NA |
| B | ModRM:r/m (r, w) | imm8 | NA | NA |

...

## CALL—Call Procedure

| Opcode | Instruction | Op/ En | 64-bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| E8 *cw* | CALL *rel16* | B | N.S. | Valid | Call near, relative, displacement relative to next instruction. |
| E8 *cd* | CALL *rel32* | B | Valid | Valid | Call near, relative, displacement relative to next instruction. 32-bit displacement sign extended to 64-bits in 64-bit mode. |
| FF /2 | CALL *r/m16* | B | N.E. | Valid | Call near, absolute indirect, address given in *r/m16*. |
| FF /2 | CALL *r/m32* | B | N.E. | Valid | Call near, absolute indirect, address given in *r/m32*. |

| Opcode | Instruction | Op/ En | 64-bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| FF /2 | CALL *r/m64* | B | Valid | N.E. | Call near, absolute indirect, address given in *r/m*64. |
| 9A *cd* | CALL *ptr16:16* | A | Invalid | Valid | Call far, absolute, address given in operand. |
| 9A *cp* | CALL *ptr16:32* | A | Invalid | Valid | Call far, absolute, address given in operand. |
| FF /3 | CALL *m16:16* | B | Valid | Valid | Call far, absolute indirect address given in *m16:16*. In 32-bit mode: if selector points to a gate, then RIP = 32-bit zero extended displacement taken from gate; else RIP = zero extended 16-bit offset from far pointer referenced in the instruction. |
| FF /3 | CALL *m16:32* | B | Valid | Valid | In 64-bit mode: If selector points to a gate, then RIP = 64-bit displacement taken from gate; else RIP = zero extended 32-bit offset from far pointer referenced in the instruction. |
| REX.W + FF /3 | CALL *m16:64* | B | Valid | N.E. | In 64-bit mode: If selector points to a gate, then RIP = 64-bit displacement taken from gate; else RIP = 64-bit offset from far pointer referenced in the instruction. |

## Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | Offset | NA | NA | NA |
| B | ModRM:r/m (r) | NA | NA | NA |

...

## BW/CWDE/CDQE—Convert Byte to Word/Convert Word to Doubleword/ Convert Doubleword to Quadword

| Opcode | Instruction | Op/ En | 64-bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 98 | CBW | A | Valid | Valid | AX ← sign-extend of AL. |
| 98 | CWDE | A | Valid | Valid | EAX ← sign-extend of AX. |
| REX.W + 98 | CDQE | A | Valid | N.E. | RAX ← sign-extend of EAX. |

**Instruction Operand Encoding**

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | NA | NA | NA | NA |

...

## CLC—Clear Carry Flag

| Opcode | Instruction | Op/ En | 64-bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| F8 | CLC | A | Valid | Valid | Clear CF flag. |

**Instruction Operand Encoding**

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | NA | NA | NA | NA |

...

## CLD—Clear Direction Flag

| Opcode | Instruction | Op/ En | 64-bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| FC | CLD | A | Valid | Valid | Clear DF flag. |

**Instruction Operand Encoding**

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | NA | NA | NA | NA |

...

## CLFLUSH—Flush Cache Line

| Opcode | Instruction | Op/ En | 64-bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F AE /7 | CLFLUSH *m8* | A | Valid | Valid | Flushes cache line containing *m8*. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:r/m (w) | NA | NA | NA |

### Description

Invalidates the cache line that contains the linear address specified with the source operand from all levels of the processor cache hierarchy (data and instruction). The invalidation is broadcast throughout the cache coherence domain. If, at any level of the cache hierarchy, the line is inconsistent with memory (dirty) it is written to memory before invalidation. The source operand is a byte memory location.

The availability of CLFLUSH is indicated by the presence of the CPUID feature flag CLFSH (bit 19 of the EDX register, see "CPUID—CPU Identification" in this chapter). The aligned cache line size affected is also indicated with the CPUID instruction (bits 8 through 15 of the EBX register when the initial value in the EAX register is 1).

The memory attribute of the page containing the affected line has no effect on the behavior of this instruction. It should be noted that processors are free to speculatively fetch and cache data from system memory regions assigned a memory-type allowing for speculative reads (such as, the WB, WC, and WT memory types). PREFETCH*h* instructions can be used to provide the processor with hints for this speculative behavior. Because this speculative fetching can occur at any time and is not tied to instruction execution, the CLFLUSH instruction is not ordered with respect to PREFETCH*h* instructions or any of the speculative fetching mechanisms (that is, data can be speculatively loaded into a cache line just before, during, or after the execution of a CLFLUSH instruction that references the cache line).

CLFLUSH is only ordered by the MFENCE instruction. It is not guaranteed to be ordered by any other fencing or serializing instructions or by another CLFLUSH instruction. For example, software can use an MFENCE instruction to ensure that previous stores are included in the write-back.

The CLFLUSH instruction can be used at all privilege levels and is subject to all permission checking and faults associated with a byte load (and in addition, a CLFLUSH instruction is allowed to flush a linear address in an execute-only segment). Like a load, the CLFLUSH instruction sets the A bit but not the D bit in the page tables.

The CLFLUSH instruction was introduced with the SSE2 extensions; however, because it has its own CPUID feature flag, it can be implemented in IA-32 processors that do not include the SSE2 extensions. Also, detecting the presence of the SSE2 extensions with the CPUID instruction does not guarantee that the CLFLUSH instruction is implemented in the processor.

CLFLUSH operation is the same in non-64-bit modes and 64-bit mode.

...

## CLI — Clear Interrupt Flag

| Opcode | Instruction | Op/ En | 64-bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| FA | CLI | A | Valid | Valid | Clear interrupt flag; interrupts disabled when interrupt flag cleared. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | NA | NA | NA | NA |

...

## CLTS—Clear Task-Switched Flag in CR0

| Opcode | Instruction | Op/ En | 64-bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F 06 | CLTS | A | Valid | Valid | Clears TS flag in CR0. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | NA | NA | NA | NA |

...

## CMC—Complement Carry Flag

| Opcode | Instruction | Op/ En | 64-bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| F5 | CMC | A | Valid | Valid | Complement CF flag. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | NA | NA | NA | NA |

...

## CMOV*cc*—Conditional Move

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F 47 */r* | CMOVA *r16, r/m16* | A | Valid | Valid | Move if above (CF=0 and ZF=0). |
| 0F 47 */r* | CMOVA *r32, r/m32* | A | Valid | Valid | Move if above (CF=0 and ZF=0). |
| REX.W + 0F 47 */r* | CMOVA *r64, r/m64* | A | Valid | N.E. | Move if above (CF=0 and ZF=0). |
| 0F 43 */r* | CMOVAE *r16, r/m16* | A | Valid | Valid | Move if above or equal (CF=0). |
| 0F 43 */r* | CMOVAE *r32, r/m32* | A | Valid | Valid | Move if above or equal (CF=0). |
| REX.W + 0F 43 */r* | CMOVAE *r64, r/m64* | A | Valid | N.E. | Move if above or equal (CF=0). |
| 0F 42 */r* | CMOVB *r16, r/m16* | A | Valid | Valid | Move if below (CF=1). |
| 0F 42 */r* | CMOVB *r32, r/m32* | A | Valid | Valid | Move if below (CF=1). |
| REX.W + 0F 42 */r* | CMOVB *r64, r/m64* | A | Valid | N.E. | Move if below (CF=1). |
| 0F 46 */r* | CMOVBE *r16, r/m16* | A | Valid | Valid | Move if below or equal (CF=1 or ZF=1). |
| 0F 46 */r* | CMOVBE *r32, r/m32* | A | Valid | Valid | Move if below or equal (CF=1 or ZF=1). |
| REX.W + 0F 46 */r* | CMOVBE *r64, r/m64* | A | Valid | N.E. | Move if below or equal (CF=1 or ZF=1). |
| 0F 42 */r* | CMOVC *r16, r/m16* | A | Valid | Valid | Move if carry (CF=1). |
| 0F 42 */r* | CMOVC *r32, r/m32* | A | Valid | Valid | Move if carry (CF=1). |
| REX.W + 0F 42 */r* | CMOVC *r64, r/m64* | A | Valid | N.E. | Move if carry (CF=1). |
| 0F 44 */r* | CMOVE *r16, r/m16* | A | Valid | Valid | Move if equal (ZF=1). |
| 0F 44 */r* | CMOVE *r32, r/m32* | A | Valid | Valid | Move if equal (ZF=1). |
| REX.W + 0F 44 */r* | CMOVE *r64, r/m64* | A | Valid | N.E. | Move if equal (ZF=1). |
| 0F 4F */r* | CMOVG *r16, r/m16* | A | Valid | Valid | Move if greater (ZF=0 and SF=OF). |
| 0F 4F */r* | CMOVG *r32, r/m32* | A | Valid | Valid | Move if greater (ZF=0 and SF=OF). |
| REX.W + 0F 4F */r* | CMOVG *r64, r/m64* | A | Valid | N.E. | Move if greater (ZF=0 and SF=OF). |
| 0F 4D */r* | CMOVGE *r16, r/m16* | A | Valid | Valid | Move if greater or equal (SF=OF). |
| 0F 4D */r* | CMOVGE *r32, r/m32* | A | Valid | Valid | Move if greater or equal (SF=OF). |
| REX.W + 0F 4D */r* | CMOVGE *r64, r/m64* | A | Valid | N.E. | Move if greater or equal (SF=OF). |

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F 4C /r | CMOVL *r16, r/m16* | A | Valid | Valid | Move if less (SF≠ OF). |
| 0F 4C /r | CMOVL *r32, r/m32* | A | Valid | Valid | Move if less (SF≠ OF). |
| REX.W + 0F 4C /r | CMOVL *r64, r/m64* | A | Valid | N.E. | Move if less (SF≠ OF). |
| 0F 4E /r | CMOVLE *r16, r/m16* | A | Valid | Valid | Move if less or equal (ZF=1 or SF≠ OF). |
| 0F 4E /r | CMOVLE *r32, r/m32* | A | Valid | Valid | Move if less or equal (ZF=1 or SF≠ OF). |
| REX.W + 0F 4E /r | CMOVLE *r64, r/m64* | A | Valid | N.E. | Move if less or equal (ZF=1 or SF≠ OF). |
| 0F 46 /r | CMOVNA *r16, r/m16* | A | Valid | Valid | Move if not above (CF=1 or ZF=1). |
| 0F 46 /r | CMOVNA *r32, r/m32* | A | Valid | Valid | Move if not above (CF=1 or ZF=1). |
| REX.W + 0F 46 /r | CMOVNA *r64, r/m64* | A | Valid | N.E. | Move if not above (CF=1 or ZF=1). |
| 0F 42 /r | CMOVNAE *r16, r/m16* | A | Valid | Valid | Move if not above or equal (CF=1). |
| 0F 42 /r | CMOVNAE *r32, r/m32* | A | Valid | Valid | Move if not above or equal (CF=1). |
| REX.W + 0F 42 /r | CMOVNAE *r64, r/m64* | A | Valid | N.E. | Move if not above or equal (CF=1). |
| 0F 43 /r | CMOVNB *r16, r/m16* | A | Valid | Valid | Move if not below (CF=0). |
| 0F 43 /r | CMOVNB *r32, r/m32* | A | Valid | Valid | Move if not below (CF=0). |
| REX.W + 0F 43 /r | CMOVNB *r64, r/m64* | A | Valid | N.E. | Move if not below (CF=0). |
| 0F 47 /r | CMOVNBE *r16, r/m16* | A | Valid | Valid | Move if not below or equal (CF=0 and ZF=0). |
| 0F 47 /r | CMOVNBE *r32, r/m32* | A | Valid | Valid | Move if not below or equal (CF=0 and ZF=0). |
| REX.W + 0F 47 /r | CMOVNBE *r64, r/m64* | A | Valid | N.E. | Move if not below or equal (CF=0 and ZF=0). |
| 0F 43 /r | CMOVNC *r16, r/m16* | A | Valid | Valid | Move if not carry (CF=0). |
| 0F 43 /r | CMOVNC *r32, r/m32* | A | Valid | Valid | Move if not carry (CF=0). |
| REX.W + 0F 43 /r | CMOVNC *r64, r/m64* | A | Valid | N.E. | Move if not carry (CF=0). |
| 0F 45 /r | CMOVNE *r16, r/m16* | A | Valid | Valid | Move if not equal (ZF=0). |
| 0F 45 /r | CMOVNE *r32, r/m32* | A | Valid | Valid | Move if not equal (ZF=0). |
| REX.W + 0F 45 /r | CMOVNE *r64, r/m64* | A | Valid | N.E. | Move if not equal (ZF=0). |
| 0F 4E /r | CMOVNG *r16, r/m16* | A | Valid | Valid | Move if not greater (ZF=1 or SF≠ OF). |

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F 4E /*r* | CMOVNG *r32, r/m32* | A | Valid | Valid | Move if not greater (ZF=1 or SF≠ OF). |
| REX.W + 0F 4E /*r* | CMOVNG *r64, r/m64* | A | Valid | N.E. | Move if not greater (ZF=1 or SF≠ OF). |
| 0F 4C /*r* | CMOVNGE *r16, r/m16* | A | Valid | Valid | Move if not greater or equal (SF≠ OF). |
| 0F 4C /*r* | CMOVNGE *r32, r/m32* | A | Valid | Valid | Move if not greater or equal (SF≠ OF). |
| REX.W + 0F 4C /*r* | CMOVNGE *r64, r/m64* | A | Valid | N.E. | Move if not greater or equal (SF≠ OF). |
| 0F 4D /*r* | CMOVNL *r16, r/m16* | A | Valid | Valid | Move if not less (SF=OF). |
| 0F 4D /*r* | CMOVNL *r32, r/m32* | A | Valid | Valid | Move if not less (SF=OF). |
| REX.W + 0F 4D /*r* | CMOVNL *r64, r/m64* | A | Valid | N.E. | Move if not less (SF=OF). |
| 0F 4F /*r* | CMOVNLE *r16, r/m16* | A | Valid | Valid | Move if not less or equal (ZF=0 and SF=OF). |
| 0F 4F /*r* | CMOVNLE *r32, r/m32* | A | Valid | Valid | Move if not less or equal (ZF=0 and SF=OF). |
| REX.W + 0F 4F /*r* | CMOVNLE *r64, r/m64* | A | Valid | N.E. | Move if not less or equal (ZF=0 and SF=OF). |
| 0F 41 /*r* | CMOVNO *r16, r/m16* | A | Valid | Valid | Move if not overflow (OF=0). |
| 0F 41 /*r* | CMOVNO *r32, r/m32* | A | Valid | Valid | Move if not overflow (OF=0). |
| REX.W + 0F 41 /*r* | CMOVNO *r64, r/m64* | A | Valid | N.E. | Move if not overflow (OF=0). |
| 0F 4B /*r* | CMOVNP *r16, r/m16* | A | Valid | Valid | Move if not parity (PF=0). |
| 0F 4B /*r* | CMOVNP *r32, r/m32* | A | Valid | Valid | Move if not parity (PF=0). |
| REX.W + 0F 4B /*r* | CMOVNP *r64, r/m64* | A | Valid | N.E. | Move if not parity (PF=0). |
| 0F 49 /*r* | CMOVNS *r16, r/m16* | A | Valid | Valid | Move if not sign (SF=0). |
| 0F 49 /*r* | CMOVNS *r32, r/m32* | A | Valid | Valid | Move if not sign (SF=0). |
| REX.W + 0F 49 /*r* | CMOVNS *r64, r/m64* | A | Valid | N.E. | Move if not sign (SF=0). |
| 0F 45 /*r* | CMOVNZ *r16, r/m16* | A | Valid | Valid | Move if not zero (ZF=0). |
| 0F 45 /*r* | CMOVNZ *r32, r/m32* | A | Valid | Valid | Move if not zero (ZF=0). |
| REX.W + 0F 45 /*r* | CMOVNZ *r64, r/m64* | A | Valid | N.E. | Move if not zero (ZF=0). |
| 0F 40 /*r* | CMOVO *r16, r/m16* | A | Valid | Valid | Move if overflow (OF=0). |
| 0F 40 /*r* | CMOVO *r32, r/m32* | A | Valid | Valid | Move if overflow (OF=0). |
| REX.W + 0F 40 /*r* | CMOVO *r64, r/m64* | A | Valid | N.E. | Move if overflow (OF=0). |

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F 4A */r* | CMOVP *r16, r/m16* | A | Valid | Valid | Move if parity (PF=1). |
| 0F 4A */r* | CMOVP *r32, r/m32* | A | Valid | Valid | Move if parity (PF=1). |
| REX.W + 0F 4A */r* | CMOVP *r64, r/m64* | A | Valid | N.E. | Move if parity (PF=1). |
| 0F 4A */r* | CMOVPE *r16, r/m16* | A | Valid | Valid | Move if parity even (PF=1). |
| 0F 4A */r* | CMOVPE *r32, r/m32* | A | Valid | Valid | Move if parity even (PF=1). |
| REX.W + 0F 4A */r* | CMOVPE *r64, r/m64* | A | Valid | N.E. | Move if parity even (PF=1). |
| 0F 4B */r* | CMOVPO *r16, r/m16* | A | Valid | Valid | Move if parity odd (PF=0). |
| 0F 4B */r* | CMOVPO *r32, r/m32* | A | Valid | Valid | Move if parity odd (PF=0). |
| REX.W + 0F 4B */r* | CMOVPO *r64, r/m64* | A | Valid | N.E. | Move if parity odd (PF=0). |
| 0F 48 */r* | CMOVS *r16, r/m16* | A | Valid | Valid | Move if sign (SF=1). |
| 0F 48 */r* | CMOVS *r32, r/m32* | A | Valid | Valid | Move if sign (SF=1). |
| REX.W + 0F 48 */r* | CMOVS *r64, r/m64* | A | Valid | N.E. | Move if sign (SF=1). |
| 0F 44 */r* | CMOVZ *r16, r/m16* | A | Valid | Valid | Move if zero (ZF=1). |
| 0F 44 */r* | CMOVZ *r32, r/m32* | A | Valid | Valid | Move if zero (ZF=1). |
| REX.W + 0F 44 */r* | CMOVZ *r64, r/m64* | A | Valid | N.E. | Move if zero (ZF=1). |

**Instruction Operand Encoding**

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (r, w) | ModRM:r/m (r) | NA | NA |

...

## CMP—Compare Two Operands

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 3C *ib* | CMP AL, *imm8* | D | Valid | Valid | Compare *imm8* with AL. |
| 3D *iw* | CMP AX, *imm16* | D | Valid | Valid | Compare *imm16* with AX. |
| 3D *id* | CMP EAX, *imm32* | D | Valid | Valid | Compare *imm32* with EAX. |
| REX.W + 3D *id* | CMP RAX, *imm32* | D | Valid | N.E. | Compare *imm32 sign-extended to 64-bits* with RAX. |
| 80 /7 *ib* | CMP *r/m8, imm8* | C | Valid | Valid | Compare *imm8* with *r/m8.* |
| REX + 80 /7 *ib* | CMP *r/m8**, *imm8* | C | Valid | N.E. | Compare *imm8* with *r/m8.* |
| 81 /7 *iw* | CMP *r/m16, imm16* | C | Valid | Valid | Compare *imm16* with *r/m16* . |

| **Opcode** | **Instruction** | **Op/ En** | **64-Bit Mode** | **Compat/ Leg Mode** | **Description** |
|---|---|---|---|---|---|
| 81 /7 *id* | CMP *r/m32, imm32* | C | Valid | Valid | Compare *imm32* with *r/m32.* |
| REX.W + 81 /7 *id* | CMP *r/m64, imm32* | C | Valid | N.E. | Compare *imm32 sign-extended to 64-bits* with *r/m64.* |
| 83 /7 *ib* | CMP *r/m16, imm8* | C | Valid | Valid | Compare *imm8* with *r/m16.* |
| 83 /7 *ib* | CMP *r/m32, imm8* | C | Valid | Valid | Compare *imm8* with *r/m32.* |
| REX.W + 83 /7 *ib* | CMP *r/m64, imm8* | C | Valid | N.E. | Compare *imm8* with *r/m64.* |
| 38 */r* | CMP *r/m8, r8* | B | Valid | Valid | Compare *r8* with *r/m8.* |
| REX + 38 */r* | CMP *r/m8**, *r8** | B | Valid | N.E. | Compare *r8* with *r/m8.* |
| 39 */r* | CMP *r/m16, r16* | B | Valid | Valid | Compare *r16* with *r/m16.* |
| 39 */r* | CMP *r/m32, r32* | B | Valid | Valid | Compare *r32* with *r/m32.* |
| REX.W + 39 */r* | CMP *r/m64,r64* | B | Valid | N.E. | Compare *r64* with *r/m64.* |
| 3A */r* | CMP *r8, r/m8* | A | Valid | Valid | Compare *r/m8* with *r8.* |
| REX + 3A */r* | CMP *r8**, *r/m8** | A | Valid | N.E. | Compare *r/m8 with r8.* |
| 3B */r* | CMP *r16, r/m16* | A | Valid | Valid | Compare *r/m16* with *r16.* |
| 3B */r* | CMP *r32, r/m32* | A | Valid | Valid | Compare *r/m32* with *r32.* |
| REX.W + 3B */r* | CMP *r64, r/m64* | A | Valid | N.E. | Compare *r/m64* with *r64.* |

**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 |
|---|---|---|---|---|
| A | ModRM:reg (r, w) | ModRM:r/m (r) | NA | NA |
| B | ModRM:r/m (r, w) | ModRM:reg (w) | NA | NA |
| C | ModRM:r/m (r, w) | imm8 | NA | NA |
| D | AL/AX/EAX/RAX | imm8 | NA | NA |

...

## CMPPD—Compare Packed Double-Precision Floating-Point Values

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 66 0F C2 /*r* ib | CMPPD *xmm1*, *xmm2/m128*, *imm8* | A | Valid | Valid | Compare packed double-precision floating-point values in *xmm2/m128* and *xmm1* using imm8 as comparison predicate. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (r, w) | ModRM:r/m (r) | imm8 | NA |

...

## CMPPS—Compare Packed Single-Precision Floating-Point Values

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F C2 /*r* ib | CMPPS *xmm1*, *xmm2/m128, imm8* | A | Valid | Valid | Compare packed single-precision floating-point values in *xmm2/mem* and *xmm1* using *imm8* as comparison predicate. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (r, w) | ModRM:r/m (r) | imm8 | NA |

...

## CMPS/CMPSB/CMPSW/CMPSD/CMPSQ—Compare String Operands

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| A6 | CMPS *m8, m8* | A | Valid | Valid | For legacy mode, compare byte at address DS:(E)SI with byte at address ES:(E)DI; For 64-bit mode compare byte at address (R\|E)SI to byte at address (R\|E)DI. The status flags are set accordingly. |
| A7 | CMPS *m16*, *m16* | A | Valid | Valid | For legacy mode, compare word at address DS:(E)SI with word at address ES:(E)DI; For 64-bit mode compare word at address (R\|E)SI with word at address (R\|E)DI. The status flags are set accordingly. |
| A7 | CMPS *m32, m32* | A | Valid | Valid | For legacy mode, compare dword at address DS:(E)SI at dword at address ES:(E)DI; For 64-bit mode compare dword at address (R\|E)SI at dword at address (R\|E)DI. The status flags are set accordingly. |
| REX.W + A7 | CMPS *m64, m64* | A | Valid | N.E. | Compares quadword at address (R\|E)SI with quadword at address (R\|E)DI and sets the status flags accordingly. |
| A6 | CMPSB | A | Valid | Valid | For legacy mode, compare byte at address DS:(E)SI with byte at address ES:(E)DI; For 64-bit mode compare byte at address (R\|E)SI with byte at address (R\|E)DI. The status flags are set accordingly. |
| A7 | CMPSW | A | Valid | Valid | For legacy mode, compare word at address DS:(E)SI with word at address ES:(E)DI; For 64-bit mode compare word at address (R\|E)SI with word at address (R\|E)DI. The status flags are set accordingly. |

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| A7 | CMPSD | A | Valid | Valid | For legacy mode, compare dword at address DS:(E)SI with dword at address ES:(E)DI; For 64-bit mode compare dword at address (R\|E)SI with dword at address (R\|E)DI. The status flags are set accordingly. |
| REX.W + A7 | CMPSQ | A | Valid | N.E. | Compares quadword at address (R\|E)SI with quadword at address (R\|E)DI and sets the status flags accordingly. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | NA | NA | NA | NA |

...

## CMPSD—Compare Scalar Double-Precision Floating-Point Values

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| F2 0F C2 /*r* ib | CMPSD *xmm1*, *xmm2/m64*, *imm8* | A | Valid | Valid | Compare low double-precision floating-point value in *xmm2/m64* and *xmm1* using *imm8* as comparison predicate. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (r, w) | ModRM:r/m (r) | imm8 | NA |

...

## CMPSS—Compare Scalar Single-Precision Floating-Point Values

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| F3 0F C2 */r ib* | CMPSS *xmm1, xmm2/m32, imm8* | A | Valid | Valid | Compare low single-precision floating-point value in *xmm2/m32* and *xmm1* using *imm8* as comparison predicate. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (r, w) | ModRM:r/m (r) | imm8 | NA |

### Description

Compares the low single-precision floating-point values in the source operand (second operand) and the destination operand (first operand) and returns the results of the comparison to the destination operand. The comparison predicate operand (third operand) specifies the type of comparison performed. The comparison result is a doubleword mask of all 1s (comparison true) or all 0s (comparison false).

The source operand can be an XMM register or a 32-bit memory location. The destination operand is an XMM register. The result is stored in the low doubleword of the destination operand; the 3 high-order doublewords remain unchanged. The comparison predicate operand is an 8-bit immediate, the first 3 bits of which define the type of comparison to be made (see Table 3-15). 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 a fault, since a mask of all 0s corresponds to a floating-point value of +0.0 and a mask of all 1s corresponds to a QNaN.

Some of the comparisons listed in Table 3-15 can be achieved only through software emulation. For these comparisons the program must swap the operands (copying registers when necessary to protect the data that will now be in the destination operand), and then perform the compare using a different predicate. The predicate to be used for these emulations is listed in Table 3-15 under the heading Emulation.

Compilers and assemblers may implement the following two-operand pseudo-ops in addition to the three-operand CMPSS instruction. See Table 3-19.

...

## CMPXCHG—Compare and Exchange

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F B0/*r* | CMPXCHG *r/m8, r8* | A | Valid | Valid* | Compare AL with *r/m8*. If equal, ZF is set and *r8* is loaded into *r/m8*. Else, clear ZF and load *r/m8* into AL. |
| REX + 0F B0/*r* | CMPXCHG *r/m8***,*r8* | A | Valid | N.E. | Compare AL with *r/m8*. If equal, ZF is set and *r8* is loaded into *r/m8*. Else, clear ZF and load *r/m8* into AL. |
| 0F B1/*r* | CMPXCHG *r/m16, r16* | A | Valid | Valid* | Compare AX with *r/m16*. If equal, ZF is set and *r16* is loaded into *r/m16*. Else, clear ZF and load *r/m16* into AX. |
| 0F B1/*r* | CMPXCHG *r/m32, r32* | A | Valid | Valid* | Compare EAX with *r/m32*. If equal, ZF is set and *r32* is loaded into *r/m32*. Else, clear ZF and load *r/m32* into EAX. |
| REX.W + 0F B1/*r* | CMPXCHG *r/m64, r64* | A | Valid | N.E. | Compare RAX with *r/m64*. If equal, ZF is set and *r64* is loaded into *r/m64*. Else, clear ZF and load *r/m64* into RAX. |

**NOTES:**

* See the IA-32 Architecture Compatibility section below.

** 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 |
|---|---|---|---|---|
| A | ModRM:r/m (r, w) | ModRM:reg (r) | NA | NA |

...

## CMPXCHG8B/CMPXCHG16B—Compare and Exchange Bytes

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F C7 /*1 m64* | CMPXCHG8B *m64* | A | Valid | Valid* | Compare EDX:EAX with *m64*. If equal, set ZF and load ECX:EBX into *m64*. Else, clear ZF and load *m64* into EDX:EAX. |
| REX.W + 0F C7 /*1 m128* | CMPXCHG16B *m128* | A | Valid | N.E. | Compare RDX:RAX with *m128*. If equal, set ZF and load RCX:RBX into *m128*. Else, clear ZF and load *m128* into RDX:RAX. |

**NOTES:**

*See IA-32 Architecture Compatibility section below.

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:r/m (r, w) | NA | NA | NA |

...

## COMISD—Compare Scalar Ordered Double-Precision Floating-Point Values and Set EFLAGS

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 66 0F 2F /*r* | COMISD *xmm1*, *xmm2/m64* | A | Valid | Valid | 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 |
|---|---|---|---|---|
| A | ModRM:reg (r) | ModRM:r/m (r) | NA | NA |

...

## COMISS—Compare Scalar Ordered Single-Precision Floating-Point Values and Set EFLAGS

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F 2F /*r* | COMISS *xmm1*, *xmm2/m32* | A | Valid | Valid | 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 |
|---|---|---|---|---|
| A | ModRM:reg (r) | ModRM:r/m (r) | NA | NA |

...

## CPUID—CPU Identification

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F A2 | CPUID | A | 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 |
|---|---|---|---|---|
| A | NA | NA | NA | NA |

...

Table 3-20. Information Returned by CPUID Instruction (Continued)

| Initial EAX Value | Information Provided about the Processor | |
|---|---|---|
| | *Basic CPUID Information* | |
| ... | | |
| 80000001H | EAX<br><br>EBX<br><br>ECX | Extended Processor Signature and Feature Bits.<br><br>Reserved<br><br>Bit 0: LAHF/SAHF available in 64-bit mode<br>Bits 31-1 Reserved |
| | EDX | Bits 10-0: Reserved<br>Bit 11: SYSCALL/SYSRET available (when in 64-bit mode)<br>Bits 19-12: Reserved = 0<br>Bit 20: Execute Disable Bit available<br>Bits 25-21: Reserved = 0<br>Bit 26: 1-GByte pages are available if 1<br>Bit 27: RDTSCP and IA32_TSC_AUX are available if 1<br>Bits 28: Reserved = 0<br>Bit 29: Intel® 64 Architecture available if 1<br>Bits 31-30: Reserved = 0 |
| ... | | |

...

Table 3-24. More on Feature Information Returned in the EDX Register

| Bit # | Mnemonic | Description |
|---|---|---|
| ... | | |
| 13 | PGE | **Page Global Bit.** 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. |
| ... | | |

...

## CRC32 — Accumulate CRC32 Value

| **Opcode** | **Instruction** | Op/ En | **64-Bit Mode** | **Compat/ Leg Mode** | **Description** |
|---|---|---|---|---|---|
| F2 0F 38 F0 */r* | CRC32 *r32, r/m8* | A | Valid | Valid | Accumulate CRC32 on *r/m8.* |
| F2 REX 0F 38 F0 */r* | CRC32 *r32, r/m8** | A | Valid | N.E. | Accumulate CRC32 on *r/m8.* |
| F2 0F 38 F1 */r* | CRC32 *r32, r/m16* | A | Valid | Valid | Accumulate CRC32 on *r/m16.* |
| F2 0F 38 F1 */r* | CRC32 *r32, r/m32* | A | Valid | Valid | Accumulate CRC32 on *r/m32.* |
| F2 REX.W 0F 38 F0 */r* | CRC32 *r64, r/m8* | A | Valid | N.E. | Accumulate CRC32 on *r/m8.* |
| F2 REX.W 0F 38 F1 */r* | CRC32 *r64, r/m64* | A | Valid | N.E. | Accumulate CRC32 on *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 |
|---|---|---|---|---|
| A | ModRM:reg (r, w) | ModRM:r/m (r) | NA | NA |

...

## CVTDQ2PD—Convert Packed Dword Integers to Packed Double-Precision FP Values

| **Opcode** | **Instruction** | Op/ En | **64-Bit Mode** | **Compat/ Leg Mode** | **Description** |
|---|---|---|---|---|---|
| F3 0F E6 | CVTDQ2PD *xmm1, xmm2/m64* | A | Valid | Valid | Convert two packed signed doubleword integers from *xmm2/m128* to two packed double-precision floating-point values in *xmm1*. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (w) | ModRM:r/m (r) | NA | NA |

...

## CVTDQ2PS—Convert Packed Dword Integers to Packed Single-Precision FP Values

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F 5B /*r* | CVTDQ2PS *xmm1*, *xmm2/m128* | A | Valid | Valid | Convert four packed signed doubleword integers from *xmm2/m128* to four packed single-precision floating-point values in *xmm1*. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (w) | ModRM:r/m (r) | NA | NA |

...

## CVTPD2DQ—Convert Packed Double-Precision FP Values to Packed Dword Integers

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| F2 0F E6 | CVTPD2DQ *xmm1*, *xmm2/m128* | A | Valid | Valid | Convert two packed double-precision floating-point values from *xmm2/m128* to two packed signed doubleword integers in *xmm1*. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (w) | ModRM:r/m (r) | NA | NA |

...

## CVTPD2PI—Convert Packed Double-Precision FP Values to Packed Dword Integers

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 66 0F 2D /*r* | CVTPD2PI *mm*, *xmm/m128* | A | Valid | Valid | Convert two packed double-precision floating-point values from *xmm/m128* to two packed signed doubleword integers in *mm*. |

**Instruction Operand Encoding**

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (w) | ModRM:r/m (r) | NA | NA |

...

## CVTPD2PS—Convert Packed Double-Precision FP Values to Packed Single-Precision FP Values

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 66 0F 5A /r | CVTPD2PS *xmm1*, *xmm2/m128* | A | Valid | Valid | Convert two packed double-precision floating-point values in *xmm2/m128* to two packed single-precision floating-point values in *xmm1*. |

**Instruction Operand Encoding**

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (w) | ModRM:r/m (r) | NA | NA |

...

## CVTPI2PD—Convert Packed Dword Integers to Packed Double-Precision FP Values

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 66 0F 2A /r | CVTPI2PD *xmm*, *mm/m64** | A | Valid | Valid | Convert two packed signed doubleword integers from *mm/mem64* to two packed double-precision floating-point values in *xmm*. |

**NOTES:**

*Operation is different for different operand sets; see the Description section.

**Instruction Operand Encoding**

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (w) | ModRM:r/m (r) | NA | NA |

...

## CVTPI2PS—Convert Packed Dword Integers to Packed Single-Precision FP Values

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F 2A /*r* | CVTPI2PS *xmm*, *mm*/*m64* | A | Valid | Valid | Convert two signed doubleword integers from *mm*/*m64* to two single-precision floating-point values in *xmm*. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (w) | ModRM:r/m (r) | NA | NA |

...

## CVTPS2DQ—Convert Packed Single-Precision FP Values to Packed Dword Integers

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 66 0F 5B /*r* | CVTPS2DQ *xmm1*, *xmm2*/*m128* | A | Valid | Valid | Convert four packed single-precision floating-point values from *xmm2*/*m128* to four packed signed doubleword integers in *xmm1*. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (w) | ModRM:r/m (r) | NA | NA |

...

## CVTPS2PD—Convert Packed Single-Precision FP Values to Packed Double-Precision FP Values

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F 5A /*r* | CVTPS2PD *xmm1*, *xmm2*/*m64* | A | Valid | Valid | Convert two packed single-precision floating-point values in *xmm2*/*m64* to two packed double-precision floating-point values in *xmm1*. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (w) | ModRM:r/m (r) | NA | NA |

...

## CVTPS2PI—Convert Packed Single-Precision FP Values to Packed Dword Integers

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F 2D /*r* | CVTPS2PI *mm, xmm/m64* | A | Valid | Valid | Convert two packed single-precision floating-point values from *xmm/m64* to two packed signed doubleword integers in *mm*. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (w) | ModRM:r/m (r) | NA | NA |

...

## CVTSD2SI—Convert Scalar Double-Precision FP Value to Integer

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| F2 0F 2D /*r* | CVTSD2SI *r32, xmm/m64* | A | Valid | Valid | Convert one double-precision floating-point value from *xmm/m64* to one signed doubleword integer *r32*. |
| F2 REX.W 0F 2D /*r* | CVTSD2SI *r64, xmm/m64* | A | Valid | N.E. | Convert one double-precision floating-point value from *xmm/m64* to one signed quadword integer sign-extended into *r64*. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (w) | ModRM:r/m (r) | NA | NA |

...

## CVTSD2SS—Convert Scalar Double-Precision FP Value to Scalar Single-Precision FP Value

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| F2 0F 5A /r | CVTSD2SS *xmm1*, *xmm2/m64* | A | Valid | Valid | Convert one double-precision floating-point value in *xmm2/m64* to one single-precision floating-point value in *xmm1*. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (w) | ModRM:r/m (r) | NA | NA |

...

## CVTSI2SD—Convert Dword Integer to Scalar Double-Precision FP Value

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| F2 0F 2A /r | CVTSI2SD *xmm*, *r/m32* | A | Valid | Valid | Convert one signed doubleword integer from *r/m32* to one double-precision floating-point value in *xmm*. |
| F2 REX.W 0F 2A /r | CVTSI2SD *xmm*, *r/m64* | A | Valid | N.E. | Convert one signed quadword integer from *r/m64* to one double-precision floating-point value in *xmm*. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (w) | ModRM:r/m (r) | NA | NA |

...

## CVTSI2SS—Convert Dword Integer to Scalar Single-Precision FP Value

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| F3 0F 2A /*r* | CVTSI2SS *xmm*, *r/m32* | A | Valid | Valid | Convert one signed doubleword integer from *r/m32* to one single-precision floating-point value in *xmm*. |
| F3 REX.W 0F 2A /*r* | CVTSI2SS *xmm*, *r/m64* | A | Valid | N.E. | Convert one signed quadword integer from *r/m64* to one single-precision floating-point value in *xmm*. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (w) | ModRM:r/m (r) | NA | NA |

...

## CVTSS2SD—Convert Scalar Single-Precision FP Value to Scalar Double-Precision FP Value

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| F3 0F 5A /*r* | CVTSS2SD *xmm1*, *xmm2/m32* | A | Valid | Valid | Convert one single-precision floating-point value in *xmm2/m32* to one double-precision floating-point value in *xmm1*. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (w) | ModRM:r/m (r) | NA | NA |

...

## VTSS2SI—Convert Scalar Single-Precision FP Value to Dword Integer

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| F3 0F 2D /*r* | CVTSS2SI *r32*, *xmm/m32* | A | Valid | Valid | Convert one single-precision floating-point value from *xmm/m32* to one signed doubleword integer in *r32*. |
| F3 REX.W 0F 2D /*r* | CVTSS2SI *r64*, *xmm/m32* | A | Valid | N.E. | Convert one single-precision floating-point value from *xmm/m32* to one signed quadword integer in *r64*. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (w) | ModRM:r/m (r) | NA | NA |

...

## CVTTPD2DQ—Convert with Truncation Packed Double-Precision FP Values to Packed Dword Integers

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 66 0F E6 | CVTTPD2DQ *xmm1*, *xmm2/m128* | A | Valid | Valid | Convert two packed double-precision floating-point values from *xmm2/m128* to two packed signed doubleword integers in *xmm1* using truncation. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (w) | ModRM:r/m (r) | NA | NA |

...

## CVTTPD2PI—Convert with Truncation Packed Double-Precision FP Values to Packed Dword Integers

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 66 0F 2C /*r* | CVTTPD2PI *mm*, *xmm/m128* | A | Valid | Valid | Convert two packer double-precision floating-point values from *xmm/m128* to two packed signed doubleword integers in *mm* using truncation. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (w) | ModRM:r/m (r) | NA | NA |

...

## CVTTPS2DQ—Convert with Truncation Packed Single-Precision FP Values to Packed Dword Integers

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| F3 0F 5B /*r* | CVTTPS2DQ *xmm1*, *xmm2/m128* | A | Valid | Valid | Convert four single-precision floating-point values from *xmm2/m128* to four signed doubleword integers in *xmm1* using truncation. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (w) | ModRM:r/m (r) | NA | NA |

...

## CVTTPS2PI—Convert with Truncation Packed Single-Precision FP Values to Packed Dword Integers

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F 2C /r | CVTTPS2PI *mm, xmm/m64* | A | Valid | Valid | Convert two single-precision floating-point values from *xmm/m64* to two signed doubleword signed integers in *mm* using truncation. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (w) | ModRM:r/m (r) | NA | NA |

...

## CVTTSD2SI—Convert with Truncation Scalar Double-Precision FP Value to Signed Integer

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| F2 0F 2C /r | CVTTSD2SI *r32*, *xmm/m64* | A | Valid | Valid | Convert one double-precision floating-point value from *xmm/m64* to one signed doubleword integer in *r32* using truncation. |
| F2 REX.W 0F 2C /r | CVTTSD2SI *r64*, *xmm/m64* | A | Valid | N.E. | Convert one double precision floating-point value from *xmm/m64* to one signedquadword integer in *r64* using truncation. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (w) | ModRM:r/m (r) | NA | NA |

...

## CVTTSS2SI—Convert with Truncation Scalar Single-Precision FP Value to Dword Integer

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| F3 0F 2C /r | CVTTSS2SI *r32*, *xmm/m32* | A | Valid | Valid | Convert one single-precision floating-point value from *xmm/m32* to one signed doubleword integer in *r32* using truncation. |
| F3 REX.W 0F 2C /r | CVTTSS2SI *r64*, *xmm/m32* | A | Valid | N.E. | Convert one single-precision floating-point value from *xmm/m32* to one signed quadword integer in *r64* using truncation. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (w) | ModRM:r/m (r) | NA | NA |

...

## CWD/CDQ/CQO—Convert Word to Doubleword/Convert Doubleword to Quadword

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 99 | CWD | A | Valid | Valid | DX:AX ← sign-extend of AX. |
| 99 | CDQ | A | Valid | Valid | EDX:EAX ← sign-extend of EAX. |
| REX.W + 99 | CQO | A | Valid | N.E. | RDX:RAX← sign-extend of RAX. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | NA | NA | NA | NA |

...

## DAA—Decimal Adjust AL after Addition

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 27 | DAA | A | Invalid | Valid | Decimal adjust AL after addition. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | NA | NA | NA | NA |

...

## DAS—Decimal Adjust AL after Subtraction

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 2F | DAS | A | Invalid | Valid | Decimal adjust AL after subtraction. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | NA | NA | NA | NA |

...

## DEC—Decrement by 1

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| FE /1 | DEC *r/m8* | A | Valid | Valid | Decrement *r/m8* by 1. |
| REX + FE /1 | DEC *r/m8*[*] | A | Valid | N.E. | Decrement *r/m8* by 1. |
| FF /1 | DEC *r/m16* | A | Valid | Valid | Decrement *r/m16* by 1. |
| FF /1 | DEC *r/m32* | A | Valid | Valid | Decrement *r/m32* by 1. |
| REX.W + FF /1 | DEC *r/m64* | A | Valid | N.E. | Decrement *r/m64* by 1. |
| 48+rw | DEC *r16* | B | N.E. | Valid | Decrement *r16* by 1. |
| 48+rd | DEC *r32* | B | N.E. | Valid | Decrement *r32* by 1. |

**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 |
|---|---|---|---|---|
| A | ModRM:r/m (r, w) | NA | NA | NA |
| B | reg (r, w) | NA | NA | NA |

...

## DIV—Unsigned Divide

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| F6 /6 | DIV *r/m8* | A | Valid | Valid | Unsigned divide AX by *r/m8*, with result stored in AL ← Quotient, AH ← Remainder. |
| REX + F6 /6 | DIV *r/m8*[*] | A | Valid | N.E. | Unsigned divide AX by *r/m8*, with result stored in AL ← Quotient, AH ← Remainder. |
| F7 /6 | DIV *r/m16* | A | Valid | Valid | Unsigned divide DX:AX by *r/m16*, with result stored in AX ← Quotient, DX ← Remainder. |
| F7 /6 | DIV *r/m32* | A | Valid | Valid | Unsigned divide EDX:EAX by *r/m32*, with result stored in EAX ← Quotient, EDX ← Remainder. |
| REX.W + F7 /6 | DIV *r/m64* | A | Valid | N.E. | Unsigned divide RDX:RAX by *r/m64*, with result stored in RAX ← Quotient, RDX ← Remainder. |

**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 |
|---|---|---|---|---|
| A | ModRM:r/m (w) | NA | NA | NA |

...

## DIVPD—Divide Packed Double-Precision Floating-Point Values

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 66 0F 5E /*r* | DIVPD *xmm1*, *xmm2/m128* | A | Valid | Valid | Divide packed double-precision floating-point values in *xmm1* by packed double-precision floating-point values *xmm2/m128*. |

**Instruction Operand Encoding**

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (r, w) | ModRM:r/m (r) | NA | NA |

...

## DIVPS—Divide Packed Single-Precision Floating-Point Values

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F 5E /*r* | DIVPS *xmm1*, *xmm2/m128* | A | Valid | Valid | Divide packed single-precision floating-point values in *xmm1* by packed single-precision floating-point values *xmm2/m128*. |

**Instruction Operand Encoding**

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (r, w) | ModRM:r/m (r) | NA | NA |

...

## DIVSD—Divide Scalar Double-Precision Floating-Point Values

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| F2 0F 5E /*r* | DIVSD *xmm1*, *xmm2/m64* | A | Valid | Valid | Divide low double-precision floating-point value n *xmm1* by low double-precision floating-point value in *xmm2/mem64*. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (r, w) | ModRM:r/m (r) | NA | NA |

...

## DIVSS—Divide Scalar Single-Precision Floating-Point Values

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| F3 0F 5E /*r* | DIVSS *xmm1*, *xmm2/m32* | A | Valid | Valid | Divide low single-precision floating-point value in *xmm1* by low single-precision floating-point value in *xmm2/m32.* |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (r, w) | ModRM:r/m (r) | NA | NA |

...

## DPPD — Dot Product of Packed Double Precision Floating-Point Values

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 66 0F 3A 41 /r ib | DPPD *xmm1, xmm2/m128, imm8* | A | Valid | Valid | Selectively multiply packed DP floating-point values from *xmm1* with packed DP floating-point values from *xmm2*, add and selectively store the packed DP floating-point values to *xmm1*. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (r, w) | ModRM:r/m (r) | imm8 | NA |

...

## DPPS — Dot Product of Packed Single Precision Floating-Point Values

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 66 0F 3A 40 /r ib | DPPS *xmm1, xmm2/m128, imm8* | A | Valid | Valid | Selectively multiply packed SP floating-point values from *xmm1* with packed SP floating-point values from *xmm2*, add and selectively store the packed SP floating-point values or zero values to *xmm1*. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (r, w) | ModRM:r/m (r) | imm8 | NA |

...

## EMMS—Empty MMX Technology State

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F 77 | EMMS | A | Valid | Valid | Set the x87 FPU tag word to empty. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | NA | NA | NA | NA |

...

## ENTER—Make Stack Frame for Procedure Parameters

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| C8 *iw* 00 | ENTER *imm16*, 0 | A | Valid | Valid | Create a stack frame for a procedure. |
| C8 *iw* 01 | ENTER *imm16*,1 | A | Valid | Valid | Create a nested stack frame for a procedure. |
| C8 *iw* ib | ENTER *imm16, imm8* | A | Valid | Valid | Create a nested stack frame for a procedure. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | iw | imm8 | NA | NA |

...

## EXTRACTPS — Extract Packed Single Precision Floating-Point Value

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 66 0F 3A 17 /r ib | EXTRACTPS *reg/m32, xmm2, imm8* | A | Valid | Valid | Extract a single-precision floating-point value from *xmm2* at the source offset specified by *imm8* and store the result to *reg or m32*. The upper 32 bits of r64 is zeroed if reg is r64. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:r/m (w) | ModRM:reg (r) | imm8 | NA |

...

## FSAVE/FNSAVE—Store x87 FPU State

...

### IA-32 Architecture Compatibility

For Intel math coprocessors and FPUs prior to the Intel Pentium processor, an FWAIT instruction should be executed before attempting to read from the memory image stored with a prior FSAVE/FNSAVE instruction. This FWAIT instruction helps ensure that the storage operation has been completed.

When operating a Pentium or Intel486 processor in MS-DOS compatibility mode, it is possible (under unusual circumstances) for an FNSAVE instruction to be interrupted prior to being executed to handle a pending FPU exception. See the section titled “No-Wait FPU Instructions Can Get FPU Interrupt in Window” in Appendix D of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for a description of these circumstances. An FNSAVE instruction cannot be interrupted in this way on a Pentium 4, Intel Xeon, or P6 family processor.

...

## FXRSTOR—Restore x87 FPU, MMX , XMM, and MXCSR State

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F AE /1 | FXRSTOR *m512byte* | A | Valid | Valid | Restore the x87 FPU, MMX, XMM, and MXCSR register state from *m512byte*. |
| REX.W+ 0F AE / 1 | FXRSTOR64 *m512byte* | A | Valid | N.E. | Restore the x87 FPU, MMX, XMM, and MXCSR register state from *m512byte*. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:r/m (r) | NA | NA | NA |

...

## FXSAVE—Save x87 FPU, MMX Technology, and SSE State

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F AE /0 | FXSAVE *m512byte* | A | Valid | Valid | Save the x87 FPU, MMX, XMM, and MXCSR register state to *m512byte*. |
| REX.W+ 0F AE / 0 | FXSAVE64 *m512byte* | A | Valid | N.E. | Save the x87 FPU, MMX, XMM, and MXCSR register state to *m512byte*. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:r/m (w) | NA | NA | NA |

...

## HADDPD—Packed Double-FP Horizontal Add

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 66 0F 7C /r | HADDPD *xmm1, xmm2/m128* | A | Valid | Valid | Horizontal add packed double-precision floating-point values from *xmm2/m128* to *xmm1*. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (r, w) | ModRM:r/m (r) | NA | NA |

...

## HADDPS—Packed Single-FP Horizontal Add

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| F2 0F 7C /r | HADDPS *xmm1, xmm2/m128* | A | Valid | Valid | Horizontal add packed single-precision floating-point values from *xmm2/m128* to *xmm1*. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (r, w) | ModRM:r/m (r) | NA | NA |

...

## HLT—Halt

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| F4 | HLT | A | Valid | Valid | Halt |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | NA | NA | NA | NA |

...

## HSUBPD—Packed Double-FP Horizontal Subtract

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 66 0F 7D /r | HSUBPD *xmm1, xmm2/m128* | A | Valid | Valid | Horizontal subtract packed double-precision floating-point values from *xmm2/m128* to *xmm1*. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (r, w) | ModRM:r/m (r) | NA | NA |

...

## HSUBPS—Packed Single-FP Horizontal Subtract

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| F2 0F 7D /r | HSUBPS *xmm1, xmm2/m128* | A | Valid | Valid | Horizontal subtract packed single-precision floating-point values from *xmm2/m128* to *xmm1*. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (r, w) | ModRM:r/m (r) | NA | NA |

...

## IDIV—Signed Divide

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| F6 /7 | IDIV *r/m8* | A | Valid | Valid | Signed divide AX by *r/m8*, with result stored in: AL ← Quotient, AH ← Remainder. |
| REX + F6 /7 | IDIV *r/m8** | A | Valid | N.E. | Signed divide AX by *r/m8*, with result stored in AL ← Quotient, AH ← Remainder. |
| F7 /7 | IDIV *r/m16* | A | Valid | Valid | Signed divide DX:AX by *r/m16*, with result stored in AX ← Quotient, DX ← Remainder. |
| F7 /7 | IDIV *r/m32* | A | Valid | Valid | Signed divide EDX:EAX by *r/m32*, with result stored in EAX ← Quotient, EDX ← Remainder. |
| REX.W + F7 /7 | IDIV *r/m64* | A | Valid | N.E. | Signed divide RDX:RAX by *r/m64*, with result stored in RAX ← Quotient, RDX ← Remainder. |

**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 |
|---|---|---|---|---|
| A | ModRM:r/m (r) | NA | NA | NA |

...

## IMUL—Signed Multiply

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| F6 /5 | IMUL *r/m8** | A | Valid | Valid | AX← AL ∗ *r/m* byte. |
| F7 /5 | IMUL *r/m16* | A | Valid | Valid | DX:AX ← AX ∗ *r/m* word. |
| F7 /5 | IMUL *r/m32* | A | Valid | Valid | EDX:EAX ← EAX ∗ *r/m*32. |
| REX.W + F7 /5 | IMUL *r/m64* | A | Valid | N.E. | RDX:RAX ← RAX ∗ *r/m*64. |
| 0F AF /*r* | IMUL *r16, r/m16* | B | Valid | Valid | word register ← word register ∗ *r/m*16. |
| 0F AF /*r* | IMUL *r32, r/m32* | B | Valid | Valid | doubleword register ← doubleword register ∗ *r/m32.* |
| REX.W + 0F AF /*r* | IMUL *r64, r/m64* | B | Valid | N.E. | Quadword register ← Quadword register ∗ *r/m64*. |
| 6B /*r ib* | IMUL *r16, r/m16, imm8* | C | Valid | Valid | word register ← *r/m16* ∗ sign-extended immediate byte. |
| 6B /*r ib* | IMUL *r32, r/m32, imm8* | C | Valid | Valid | doubleword register ← *r/m32* ∗ sign-extended immediate byte. |
| REX.W + 6B /*r ib* | IMUL *r64, r/m64, imm8* | C | Valid | N.E. | Quadword register ← *r/m64* ∗ sign-extended immediate byte. |
| 69 /*r iw* | IMUL *r16, r/m16, imm16* | C | Valid | Valid | word register ← *r/m16* ∗ immediate word. |
| 69 /*r id* | IMUL *r32, r/m32, imm32* | C | Valid | Valid | doubleword register ← *r/m32* ∗ immediate doubleword. |
| REX.W + 69 /*r id* | IMUL *r64, r/m64, imm32* | C | Valid | N.E. | Quadword register ← *r/m64* ∗ 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 |
|---|---|---|---|---|
| A | ModRM:r/m (r, w) | NA | NA | NA |
| B | ModRM:reg (r, w) | ModRM:r/m (r) | NA | NA |
| C | ModRM:reg (r, w) | ModRM:r/m (r) | imm8/16/32 | NA |

...

## IN—Input from Port

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| E4 *ib* | IN AL, *imm8* | A | Valid | Valid | Input byte from *imm8* I/O port address into AL. |
| E5 *ib* | IN AX, *imm8* | A | Valid | Valid | Input word from *imm8* I/O port address into AX. |
| E5 *ib* | IN EAX, *imm8* | A | Valid | Valid | Input dword from *imm8* I/O port address into EAX. |
| EC | IN AL,DX | B | Valid | Valid | Input byte from I/O port in DX into AL. |
| ED | IN AX,DX | B | Valid | Valid | Input word from I/O port in DX into AX. |
| ED | IN EAX,DX | B | Valid | Valid | Input doubleword from I/O port in DX into EAX. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | imm8 | NA | NA | NA |
| B | NA | NA | NA | NA |

...

## INC—Increment by 1

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| FE /0 | INC *r/m8* | A | Valid | Valid | Increment *r/m* byte by 1. |
| REX + FE /0 | INC *r/m8** | A | Valid | N.E. | Increment *r/m* byte by 1. |
| FF /0 | INC *r/m16* | A | Valid | Valid | Increment *r/m* word by 1. |
| FF /0 | INC *r/m32* | A | Valid | Valid | Increment *r/m* doubleword by 1. |
| REX.W + FF /0 | INC *r/m64* | A | Valid | N.E. | Increment *r/m* quadword by 1. |
| 40+ *rw*** | INC *r16* | B | N.E. | Valid | Increment word register by 1. |
| 40+ *rd* | INC *r32* | B | N.E. | Valid | Increment doubleword register by 1. |

**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.

** 40H through 47H are REX prefixes in 64-bit mode.

**Instruction Operand Encoding**

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:r/m (r, w) | NA | NA | NA |
| B | reg (r, w) | NA | NA | NA |

...

## INS/INSB/INSW/INSD—Input from Port to String

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 6C | INS *m8*, DX | A | Valid | Valid | Input byte from I/O port specified in DX into memory location specified in ES:(E)DI or RDI.* |
| 6D | INS *m16*, DX | A | Valid | Valid | Input word from I/O port specified in DX into memory location specified in ES:(E)DI or RDI.[1] |
| 6D | INS *m32*, DX | A | Valid | Valid | Input doubleword from I/O port specified in DX into memory location specified in ES:(E)DI or RDI.[1] |
| 6C | INSB | A | Valid | Valid | Input byte from I/O port specified in DX into memory location specified with ES:(E)DI or RDI.[1] |
| 6D | INSW | A | Valid | Valid | Input word from I/O port specified in DX into memory location specified in ES:(E)DI or RDI.[1] |
| 6D | INSD | A | Valid | Valid | Input doubleword from I/O port specified in DX into memory location specified in ES:(E)DI or RDI.[1] |

**NOTES:**

* In 64-bit mode, only 64-bit (RDI) and 32-bit (EDI) address sizes are supported. In non-64-bit mode, only 32-bit (EDI) and 16-bit (DI) address sizes are supported.

**Instruction Operand Encoding**

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | NA | NA | NA | NA |

...

## INSERTPS — Insert Packed Single Precision Floating-Point Value

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 66 0F 3A 21 /r ib | INSERTPS *xmm1, xmm2/m32, imm8* | A | Valid | Valid | 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*. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (w) | ModRM:r/m (r) | imm8 | NA |

...

## INT n/INTO/INT 3—Call to Interrupt Procedure

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| CC | INT 3 | A | Valid | Valid | Interrupt 3—trap to debugger. |
| CD *ib* | INT *imm8* | B | Valid | Valid | Interrupt vector number specified by immediate byte. |
| CE | INTO | A | Invalid | Valid | Interrupt 4—if overflow flag is 1. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | NA | NA | NA | NA |
| B | imm8 | NA | NA | NA |

...

### Operation

The following operational description applies not only to the INT *n* and INTO instructions, but also to external interrupts and exceptions.

```
IF PE = 0
   THEN
      GOTO REAL-ADDRESS-MODE;
   ELSE (* PE = 1 *)
```

```
            IF (VM = 1 and IOPL < 3 AND INT n)
                THEN
                    #GP(0);
                ELSE (* Protected mode, IA-32e mode, or virtual-8086 mode interrupt *)
                    IF (IA32_EFER.LMA = 0)
                        THEN (* Protected mode, or virtual-8086 mode interrupt *)
                            GOTO PROTECTED-MODE;
                    ELSE (* IA-32e mode interrupt *)
                        GOTO IA-32e-MODE;
                    FI;
            FI;
FI;
REAL-ADDRESS-MODE:
    IF ((vector_number ∗ 4) + 3) is not within IDT limit
        THEN #GP; FI;
    IF stack not large enough for a 6-byte return information
        THEN #SS; FI;
    Push (EFLAGS[15:0]);
    IF ← 0; (* Clear interrupt flag *)
    TF ← 0; (* Clear trap flag *)
    AC ← 0; (* Clear AC flag *)
    Push(CS);
    Push(IP);
    (* No error codes are pushed *)
    CS ← IDT(Descriptor (vector_number ∗ 4), selector));
    EIP ← IDT(Descriptor (vector_number ∗ 4), offset)); (* 16 bit offset AND 0000FFFFH *)
END;
PROTECTED-MODE:
    IF ((vector_number ∗ 8) + 7) is not within IDT limits
    or selected IDT descriptor is not an interrupt-, trap-, or task-gate type
        THEN #GP((vector_number ∗ 8) + 2 + EXT); FI;
        (* EXT is bit 0 in error code *)
    IF software interrupt (* Generated by INT n, INT 3, or INTO *)
        THEN
            IF gate descriptor DPL < CPL
                THEN #GP((vector_number ∗ 8) + 2 ); FI;
                (* PE = 1, DPL<CPL, software interrupt *)
    FI;
    IF gate not present
        THEN #NP((vector_number ∗ 8) + 2 + EXT); FI;
    IF task gate (* Specified in the selected interrupt table descriptor *)
        THEN GOTO TASK-GATE;
        ELSE GOTO TRAP-OR-INTERRUPT-GATE; (* PE = 1, trap/interrupt gate *)
    FI;
END;
IA-32e-MODE:
    IF ((vector_number ∗ 16) + 15) is not in IDT limits
    or selected IDT descriptor is not an interrupt-, or trap-gate type
        THEN #GP((vector_number « 3) + 2 + EXT);
        (* EXT is bit 0 in error code *)
    FI;
    IF software interrupt (* Generated by INT n, INT 3, but not INTO *)
        THEN
            IF gate descriptor DPL < CPL
                THEN #GP((vector_number « 3) + 2 );
                (* PE = 1, DPL < CPL, software interrupt *)
            FI;
        ELSE (* Generated by INTO *)
            #UD;
    FI;
    IF gate not present
        THEN #NP((vector_number « 3) + 2 + EXT);
    FI;
    IF ((vector_number * 16)[IST] ≠ 0)
        NewRSP ← TSS[ISTx]; FI;
    GOTO TRAP-OR-INTERRUPT-GATE; (* Trap/interrupt gate *)
END;
```

...

## INVD—Invalidate Internal Caches

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F 08 | INVD | A | Valid | Valid | Flush internal caches; initiate flushing of external caches. |

**NOTES:**

* See the IA-32 Architecture Compatibility section below.

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | NA | NA | NA | NA |

...

## INVLPG—Invalidate TLB Entry

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F 01/7 | INVLPG *m* | A | Valid | Valid | Invalidate TLB Entry for page that contains *m.* |

**NOTES:**

* See the IA-32 Architecture Compatibility section below.

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:r/m (r) | NA | NA | NA |

...

## IRET/IRETD—Interrupt Return

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| CF | IRET | A | Valid | Valid | Interrupt return (16-bit operand size). |
| CF | IRETD | A | Valid | Valid | Interrupt return (32-bit operand size). |
| REX.W + CF | IRETQ | A | Valid | N.E. | Interrupt return (64-bit operand size). |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | NA | NA | NA | NA |

...

## Jcc—Jump if Condition Is Met

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 77 *cb* | JA *rel8* | A | Valid | Valid | Jump short if above (CF=0 and ZF=0). |
| 73 *cb* | JAE *rel8* | A | Valid | Valid | Jump short if above or equal (CF=0). |
| 72 *cb* | JB *rel8* | A | Valid | Valid | Jump short if below (CF=1). |
| 76 *cb* | JBE *rel8* | A | Valid | Valid | Jump short if below or equal (CF=1 or ZF=1). |
| 72 *cb* | JC *rel8* | A | Valid | Valid | Jump short if carry (CF=1). |
| E3 *cb* | JCXZ *rel8* | A | N.E. | Valid | Jump short if CX register is 0. |
| E3 *cb* | JECXZ *rel8* | A | Valid | Valid | Jump short if ECX register is 0. |
| E3 *cb* | JRCXZ *rel8* | A | Valid | N.E. | Jump short if RCX register is 0. |
| 74 *cb* | JE *rel8* | A | Valid | Valid | Jump short if equal (ZF=1). |
| 7F *cb* | JG *rel8* | A | Valid | Valid | Jump short if greater (ZF=0 and SF=OF). |
| 7D *cb* | JGE *rel8* | A | Valid | Valid | Jump short if greater or equal (SF=OF). |
| 7C *cb* | JL *rel8* | A | Valid | Valid | Jump short if less (SF≠ OF). |
| 7E *cb* | JLE *rel8* | A | Valid | Valid | Jump short if less or equal (ZF=1 or SF≠ OF). |
| 76 *cb* | JNA *rel8* | A | Valid | Valid | Jump short if not above (CF=1 or ZF=1). |
| 72 *cb* | JNAE *rel8* | A | Valid | Valid | Jump short if not above or equal (CF=1). |
| 73 *cb* | JNB *rel8* | A | Valid | Valid | Jump short if not below (CF=0). |
| 77 *cb* | JNBE *rel8* | A | Valid | Valid | Jump short if not below or equal (CF=0 and ZF=0). |
| 73 *cb* | JNC *rel8* | A | Valid | Valid | Jump short if not carry (CF=0). |
| 75 *cb* | JNE *rel8* | A | Valid | Valid | Jump short if not equal (ZF=0). |
| 7E *cb* | JNG *rel8* | A | Valid | Valid | Jump short if not greater (ZF=1 or SF≠ OF). |
| 7C *cb* | JNGE *rel8* | A | Valid | Valid | Jump short if not greater or equal (SF≠ OF). |
| 7D *cb* | JNL *rel8* | A | Valid | Valid | Jump short if not less (SF=OF). |
| 7F *cb* | JNLE *rel8* | A | Valid | Valid | Jump short if not less or equal (ZF=0 and SF=OF). |

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 71 *cb* | JNO *rel8* | A | Valid | Valid | Jump short if not overflow (OF=0). |
| 7B *cb* | JNP *rel8* | A | Valid | Valid | Jump short if not parity (PF=0). |
| 79 *cb* | JNS *rel8* | A | Valid | Valid | Jump short if not sign (SF=0). |
| 75 *cb* | JNZ *rel8* | A | Valid | Valid | Jump short if not zero (ZF=0). |
| 70 *cb* | JO *rel8* | A | Valid | Valid | Jump short if overflow (OF=1). |
| 7A *cb* | JP *rel8* | A | Valid | Valid | Jump short if parity (PF=1). |
| 7A *cb* | JPE *rel8* | A | Valid | Valid | Jump short if parity even (PF=1). |
| 7B *cb* | JPO *rel8* | A | Valid | Valid | Jump short if parity odd (PF=0). |
| 78 *cb* | JS *rel8* | A | Valid | Valid | Jump short if sign (SF=1). |
| 74 *cb* | JZ *rel8* | A | Valid | Valid | Jump short if zero (ZF ← 1). |
| 0F 87 *cw* | JA *rel16* | A | N.S. | Valid | Jump near if above (CF=0 and ZF=0). Not supported in 64-bit mode. |
| 0F 87 *cd* | JA *rel32* | A | Valid | Valid | Jump near if above (CF=0 and ZF=0). |
| 0F 83 *cw* | JAE *rel16* | A | N.S. | Valid | Jump near if above or equal (CF=0). Not supported in 64-bit mode. |
| 0F 83 *cd* | JAE *rel32* | A | Valid | Valid | Jump near if above or equal (CF=0). |
| 0F 82 *cw* | JB *rel16* | A | N.S. | Valid | Jump near if below (CF=1). Not supported in 64-bit mode. |
| 0F 82 *cd* | JB *rel32* | A | Valid | Valid | Jump near if below (CF=1). |
| 0F 86 *cw* | JBE *rel16* | A | N.S. | Valid | Jump near if below or equal (CF=1 or ZF=1). Not supported in 64-bit mode. |
| 0F 86 *cd* | JBE *rel32* | A | Valid | Valid | Jump near if below or equal (CF=1 or ZF=1). |
| 0F 82 *cw* | JC *rel16* | A | N.S. | Valid | Jump near if carry (CF=1). Not supported in 64-bit mode. |
| 0F 82 *cd* | JC *rel32* | A | Valid | Valid | Jump near if carry (CF=1). |
| 0F 84 *cw* | JE *rel16* | A | N.S. | Valid | Jump near if equal (ZF=1). Not supported in 64-bit mode. |

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F 84 *cd* | JE *rel32* | A | Valid | Valid | Jump near if equal (ZF=1). |
| 0F 84 *cw* | JZ *rel16* | A | N.S. | Valid | Jump near if 0 (ZF=1). Not supported in 64-bit mode. |
| 0F 84 *cd* | JZ *rel32* | A | Valid | Valid | Jump near if 0 (ZF=1). |
| 0F 8F *cw* | JG *rel16* | A | N.S. | Valid | Jump near if greater (ZF=0 and SF=OF). Not supported in 64-bit mode. |
| 0F 8F *cd* | JG *rel32* | A | Valid | Valid | Jump near if greater (ZF=0 and SF=OF). |
| 0F 8D *cw* | JGE *rel16* | A | N.S. | Valid | Jump near if greater or equal (SF=OF). Not supported in 64-bit mode. |
| 0F 8D *cd* | JGE *rel32* | A | Valid | Valid | Jump near if greater or equal (SF=OF). |
| 0F 8C *cw* | JL *rel16* | A | N.S. | Valid | Jump near if less (SF≠ OF). Not supported in 64-bit mode. |
| 0F 8C *cd* | JL *rel32* | A | Valid | Valid | Jump near if less (SF≠ OF). |
| 0F 8E *cw* | JLE *rel16* | A | N.S. | Valid | Jump near if less or equal (ZF=1 or SF≠ OF). Not supported in 64-bit mode. |
| 0F 8E *cd* | JLE *rel32* | A | Valid | Valid | Jump near if less or equal (ZF=1 or SF≠ OF). |
| 0F 86 *cw* | JNA *rel16* | A | N.S. | Valid | Jump near if not above (CF=1 or ZF=1). Not supported in 64-bit mode. |
| 0F 86 *cd* | JNA *rel32* | A | Valid | Valid | Jump near if not above (CF=1 or ZF=1). |
| 0F 82 *cw* | JNAE *rel16* | A | N.S. | Valid | Jump near if not above or equal (CF=1). Not supported in 64-bit mode. |
| 0F 82 *cd* | JNAE *rel32* | A | Valid | Valid | Jump near if not above or equal (CF=1). |
| 0F 83 *cw* | JNB *rel16* | A | N.S. | Valid | Jump near if not below (CF=0). Not supported in 64-bit mode. |
| 0F 83 *cd* | JNB *rel32* | A | Valid | Valid | Jump near if not below (CF=0). |
| 0F 87 *cw* | JNBE *rel16* | A | N.S. | Valid | Jump near if not below or equal (CF=0 and ZF=0). Not supported in 64-bit mode. |

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F 87 *cd* | JNBE *rel32* | A | Valid | Valid | Jump near if not below or equal (CF=0 and ZF=0). |
| 0F 83 *cw* | JNC *rel16* | A | N.S. | Valid | Jump near if not carry (CF=0). Not supported in 64-bit mode. |
| 0F 83 *cd* | JNC *rel32* | A | Valid | Valid | Jump near if not carry (CF=0). |
| 0F 85 *cw* | JNE *rel16* | A | N.S. | Valid | Jump near if not equal (ZF=0). Not supported in 64-bit mode. |
| 0F 85 *cd* | JNE *rel32* | A | Valid | Valid | Jump near if not equal (ZF=0). |
| 0F 8E *cw* | JNG *rel16* | A | N.S. | Valid | Jump near if not greater (ZF=1 or SF≠ OF). Not supported in 64-bit mode. |
| 0F 8E *cd* | JNG *rel32* | A | Valid | Valid | Jump near if not greater (ZF=1 or SF≠ OF). |
| 0F 8C *cw* | JNGE *rel16* | A | N.S. | Valid | Jump near if not greater or equal (SF≠ OF). Not supported in 64-bit mode. |
| 0F 8C *cd* | JNGE *rel32* | A | Valid | Valid | Jump near if not greater or equal (SF≠ OF). |
| 0F 8D *cw* | JNL *rel16* | A | N.S. | Valid | Jump near if not less (SF=OF). Not supported in 64-bit mode. |
| 0F 8D *cd* | JNL *rel32* | A | Valid | Valid | Jump near if not less (SF=OF). |
| 0F 8F *cw* | JNLE *rel16* | A | N.S. | Valid | Jump near if not less or equal (ZF=0 and SF=OF). Not supported in 64-bit mode. |
| 0F 8F *cd* | JNLE *rel32* | A | Valid | Valid | Jump near if not less or equal (ZF=0 and SF=OF). |
| 0F 81 *cw* | JNO *rel16* | A | N.S. | Valid | Jump near if not overflow (OF=0). Not supported in 64-bit mode. |
| 0F 81 *cd* | JNO *rel32* | A | Valid | Valid | Jump near if not overflow (OF=0). |
| 0F 8B *cw* | JNP *rel16* | A | N.S. | Valid | Jump near if not parity (PF=0). Not supported in 64-bit mode. |
| 0F 8B *cd* | JNP *rel32* | A | Valid | Valid | Jump near if not parity (PF=0). |

| **Opcode** | **Instruction** | **Op/ En** | **64-Bit Mode** | **Compat/ Leg Mode** | **Description** |
|---|---|---|---|---|---|
| 0F 89 *cw* | JNS *rel16* | A | N.S. | Valid | Jump near if not sign (SF=0). Not supported in 64-bit mode. |
| 0F 89 *cd* | JNS *rel32* | A | Valid | Valid | Jump near if not sign (SF=0). |
| 0F 85 *cw* | JNZ *rel16* | A | N.S. | Valid | Jump near if not zero (ZF=0). Not supported in 64-bit mode. |
| 0F 85 *cd* | JNZ *rel32* | A | Valid | Valid | Jump near if not zero (ZF=0). |
| 0F 80 *cw* | JO *rel16* | A | N.S. | Valid | Jump near if overflow (OF=1). Not supported in 64-bit mode. |
| 0F 80 *cd* | JO *rel32* | A | Valid | Valid | Jump near if overflow (OF=1). |
| 0F 8A *cw* | JP *rel16* | A | N.S. | Valid | Jump near if parity (PF=1). Not supported in 64-bit mode. |
| 0F 8A *cd* | JP *rel32* | A | Valid | Valid | Jump near if parity (PF=1). |
| 0F 8A *cw* | JPE *rel16* | A | N.S. | Valid | Jump near if parity even (PF=1). Not supported in 64-bit mode. |
| 0F 8A *cd* | JPE *rel32* | A | Valid | Valid | Jump near if parity even (PF=1). |
| 0F 8B *cw* | JPO *rel16* | A | N.S. | Valid | Jump near if parity odd (PF=0). Not supported in 64-bit mode. |
| 0F 8B *cd* | JPO *rel32* | A | Valid | Valid | Jump near if parity odd (PF=0). |
| 0F 88 *cw* | JS *rel16* | A | N.S. | Valid | Jump near if sign (SF=1). Not supported in 64-bit mode. |
| 0F 88 *cd* | JS *rel32* | A | Valid | Valid | Jump near if sign (SF=1). |
| 0F 84 *cw* | JZ *rel16* | A | N.S. | Valid | Jump near if 0 (ZF=1). Not supported in 64-bit mode. |
| 0F 84 *cd* | JZ *rel32* | A | Valid | Valid | Jump near if 0 (ZF=1). |

**Instruction Operand Encoding**

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | Offset | NA | NA | NA |

...

## JMP—Jump

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| EB *cb* | JMP *rel8* | A | Valid | Valid | Jump short, RIP = RIP + 8-bit displacement sign extended to 64-bits |
| E9 *cw* | JMP *rel16* | A | N.S. | Valid | Jump near, relative, displacement relative to next instruction. Not supported in 64-bit mode. |
| E9 *cd* | JMP *rel32* | A | Valid | Valid | Jump near, relative, RIP = RIP + 32-bit displacement sign extended to 64-bits |
| FF /4 | JMP *r/m16* | B | N.S. | Valid | Jump near, absolute indirect, address = sign-extended *r/m16*. Not supported in 64-bit mode. |
| FF /4 | JMP *r/m32* | B | N.S. | Valid | Jump near, absolute indirect, address = sign-extended *r/m32*. Not supported in 64-bit mode. |
| FF /4 | JMP *r/m64* | B | Valid | N.E. | Jump near, absolute indirect, RIP = 64-Bit offset from register or memory |
| EA *cd* | JMP *ptr16:16* | A | Inv. | Valid | Jump far, absolute, address given in operand |
| EA *cp* | JMP *ptr16:32* | A | Inv. | Valid | Jump far, absolute, address given in operand |
| FF /5 | JMP *m16:16* | A | Valid | Valid | Jump far, absolute indirect, address given in *m16:16* |
| FF /5 | JMP *m16:32* | A | Valid | Valid | Jump far, absolute indirect, address given in *m16:32*. |
| REX.W + FF /5 | JMP *m16:64* | A | Valid | N.E. | Jump far, absolute indirect, address given in *m16:64*. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | Offset | NA | NA | NA |
| B | ModRM:r/m (r) | NA | NA | NA |

...

## LAHF—Load Status Flags into AH Register

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 9F | LAHF | A | Invalid* | Valid | Load: AH ← EFLAGS(SF:ZF:0:AF:0:PF:1:CF). |

**NOTES:**
*Valid in specific steppings. See Description section.

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | NA | NA | NA | NA |

...

## LAR—Load Access Rights Byte

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F 02 /r | LAR *r16*, *r16/m16* | A | Valid | Valid | *r16* ← *r16/m16* masked by FF00H. |
| 0F 02 /r | LAR *r32*, *r32/m16*[1] | A | Valid | Valid | *r32* ← *r32/m16* masked by 00FxFF00H |
| REX.W + 0F 02 /r | LAR *r64*, *r32/m16*[1] | A | Valid | N.E. | *r64* ← *r32/m16* masked by 00FxFF00H and zero extended |

**NOTES:**
1. For all loads (regardless of source or destination sizing) only bits 16-0 are used. Other bits are ignored.

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (w) | ModRM:r/m (r) | NA | NA |

...

## LDDQU—Load Unaligned Integer 128 Bits

| **Opcode** | **Instruction** | Op/ En | **64-Bit Mode** | **Compat/ Leg Mode** | **Description** |
|---|---|---|---|---|---|
| F2 0F F0 /*r* | LDDQU *xmm1*, *mem* | A | Valid | Valid | Load unaligned data from *mem* and return double quadword in *xmm1*. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (w) | ModRM:r/m (r) | NA | NA |

...

## LDMXCSR—Load MXCSR Register

| **Opcode** | **Instruction** | Op/ En | **64-Bit Mode** | **Compat/ Leg Mode** | **Description** |
|---|---|---|---|---|---|
| 0F,AE,/2 | LDMXCSR *m32* | A | Valid | Valid | Load MXCSR register from *m32*. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:r/m (r) | NA | NA | NA |

...

## LDS/LES/LFS/LGS/LSS—Load Far Pointer

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| C5 /*r* | LDS *r16,m16:16* | A | Invalid | Valid | Load DS:*r16* with far pointer from memory. |
| C5 /*r* | LDS *r32,m16:32* | A | Invalid | Valid | Load DS:*r32* with far pointer from memory. |
| 0F B2 /*r* | LSS *r16,m16:16* | A | Valid | Valid | Load SS:*r16* with far pointer from memory. |
| 0F B2 /*r* | LSS *r32,m16:32* | A | Valid | Valid | Load SS:*r32* with far pointer from memory. |
| REX + 0F B2 /*r* | LSS *r64,m16:64* | A | Valid | N.E. | Load SS:*r64* with far pointer from memory. |
| C4 /*r* | LES *r16,m16:16* | A | Invalid | Valid | Load ES:*r16* with far pointer from memory. |
| C4 /*r* | LES *r32,m16:32* | A | Invalid | Valid | Load ES:*r32* with far pointer from memory. |
| 0F B4 /*r* | LFS *r16,m16:16* | A | Valid | Valid | Load FS:*r16* with far pointer from memory. |
| 0F B4 /*r* | LFS *r32,m16:32* | A | Valid | Valid | Load FS:*r32* with far pointer from memory. |
| REX + 0F B4 /*r* | LFS *r64,m16:64* | A | Valid | N.E. | Load FS:*r64* with far pointer from memory. |
| 0F B5 /*r* | LGS *r16,m16:16* | A | Valid | Valid | Load GS:*r16* with far pointer from memory. |
| 0F B5 /*r* | LGS *r32,m16:32* | A | Valid | Valid | Load GS:*r32* with far pointer from memory. |
| REX + 0F B5 /*r* | LGS *r64,m16:64* | A | Valid | N.E. | Load GS:*r64* with far pointer from memory. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (w) | ModRM:r/m (r) | NA | NA |

...

## LEA—Load Effective Address

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 8D */r* | LEA *r16,m* | A | Valid | Valid | Store effective address for *m* in register *r16.* |
| 8D */r* | LEA *r32,m* | A | Valid | Valid | Store effective address for *m* in register *r32.* |
| REX.W + 8D */r* | LEA *r64,m* | A | Valid | N.E. | Store effective address for *m* in register *r64.* |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (w) | ModRM:r/m (r) | NA | NA |

...

## LEAVE—High Level Procedure Exit

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| C9 | LEAVE | A | Valid | Valid | Set SP to BP, then pop BP. |
| C9 | LEAVE | A | N.E. | Valid | Set ESP to EBP, then pop EBP. |
| C9 | LEAVE | A | Valid | N.E. | Set RSP to RBP, then pop RBP. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | NA | NA | NA | NA |

...

## LFENCE—Load Fence

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F AE /5 | LFENCE | A | Valid | Valid | Serializes load operations. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | NA | NA | NA | NA |

...

## LGDT/LIDT—Load Global/Interrupt Descriptor Table Register

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F 01 /2 | LGDT *m16&32* | A | N.E. | Valid | Load *m* into GDTR. |
| 0F 01 /3 | LIDT *m16&32* | A | N.E. | Valid | Load *m* into IDTR. |
| 0F 01 /2 | LGDT *m16&64* | A | Valid | N.E. | Load *m* into GDTR. |
| 0F 01 /3 | LIDT *m16&64* | A | Valid | N.E. | Load *m* into IDTR. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:r/m (r) | NA | NA | NA |

...

## LLDT—Load Local Descriptor Table Register

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F 00 /2 | LLDT *r/m*16 | A | Valid | Valid | Load segment selector *r/m*16 into LDTR. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:r/m (r) | NA | NA | NA |

...

## LMSW—Load Machine Status Word

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F 01 /6 | LMSW *r/m*16 | A | Valid | Valid | Loads *r/m*16 in machine status word of CR0. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:r/m (r) | NA | NA | NA |

...

## LOCK—Assert LOCK# Signal Prefix

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| F0 | LOCK | A | Valid | Valid | Asserts LOCK# signal for duration of the accompanying instruction. |

**NOTES:**

* See IA-32 Architecture Compatibility section below.

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | NA | NA | NA | NA |

### Description

Causes the processor's LOCK# signal to be asserted during execution of the accompanying instruction (turns the instruction into an atomic instruction). In a multiprocessor environment, the LOCK# signal ensures that the processor has exclusive use of any shared memory while the signal is asserted.

Note that, in later Intel 64 and IA-32 processors (including the Pentium 4, Intel Xeon, and P6 family processors), locking may occur without the LOCK# signal being asserted. See the "IA-32 Architecture Compatibility" section below.

The LOCK prefix can be prepended only to the following instructions and only to those forms of the instructions where the destination operand is a memory operand: ADD, ADC, AND, BTC, BTR, BTS, CMPXCHG, CMPXCH8B, DEC, INC, NEG, NOT, OR, SBB, SUB, XOR, XADD, and XCHG. If the LOCK prefix is used with one of these instructions and the source operand is a memory operand, an undefined opcode exception (#UD) may be generated. An undefined opcode exception will also be generated if the LOCK prefix is used with any instruction not in the above list. The XCHG instruction always asserts the LOCK# signal regardless of the presence or absence of the LOCK prefix.

The LOCK prefix is typically used with the BTS instruction to perform a read-modify-write operation on a memory location in shared memory environment.

The integrity of the LOCK prefix is not affected by the alignment of the memory field. Memory locking is observed for arbitrarily misaligned fields.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

### IA-32 Architecture Compatibility

Beginning with the P6 family processors, when the LOCK prefix is prefixed to an instruction and the memory area being accessed is cached internally in the processor, the LOCK# signal is generally not asserted. Instead, only the processor's cache is locked. Here, the processor's cache coherency mechanism ensures that the operation is carried out atomically with regards to memory. See "Effects of a Locked Operation on Internal Processor Caches" in Chapter 8 of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*, the for more information on locking of caches.

...

## LODS/LODSB/LODSW/LODSD/LODSQ—Load String

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| AC | LODS *m8* | A | Valid | Valid | For legacy mode, Load byte at address DS:(E)SI into AL. For 64-bit mode load byte at address (R)SI into AL. |
| AD | LODS *m16* | A | Valid | Valid | For legacy mode, Load word at address DS:(E)SI into AX. For 64-bit mode load word at address (R)SI into AX. |
| AD | LODS m32 | A | Valid | Valid | For legacy mode, Load dword at address DS:(E)SI into EAX. For 64-bit mode load dword at address (R)SI into EAX. |
| REX.W + AD | LODS *m64* | A | Valid | N.E. | Load qword at address (R)SI into RAX. |
| AC | LODSB | A | Valid | Valid | For legacy mode, Load byte at address DS:(E)SI into AL. For 64-bit mode load byte at address (R)SI into AL. |
| AD | LODSW | A | Valid | Valid | For legacy mode, Load word at address DS:(E)SI into AX. For 64-bit mode load word at address (R)SI into AX. |
| AD | LODSD | A | Valid | Valid | For legacy mode, Load dword at address DS:(E)SI into EAX. For 64-bit mode load dword at address (R)SI into EAX. |
| REX.W + AD | LODSQ | A | Valid | N.E. | Load qword at address (R)SI into RAX. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | NA | NA | NA | NA |

...

## LOOP/LOOP*cc*—Loop According to ECX Counter

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| E2 *cb* | LOOP *rel8* | A | Valid | Valid | Decrement count; jump short if count ≠ 0. |
| E1 *cb* | LOOPE *rel8* | A | Valid | Valid | Decrement count; jump short if count ≠ 0 and ZF = 1. |
| E0 *cb* | LOOPNE *rel8* | A | Valid | Valid | Decrement count; jump short if count ≠ 0 and ZF = 0. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | Offset | NA | NA | NA |

...

## LSL—Load Segment Limit

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F 03 /*r* | LSL *r16*, *r16/m16* | A | Valid | Valid | Load: *r16* ← segment limit, selector *r16/m16.* |
| 0F 03 /*r* | LSL *r32*, *r32/m16** | A | Valid | Valid | Load: *r32* ← segment limit, selector *r32/m16.* |
| REX.W + 0F 03 /*r* | LSL *r64*, *r32/m16** | A | Valid | Valid | Load: *r64* ← segment limit, selector *r32/m16* |

**NOTES:**

* For all loads (regardless of destination sizing), only bits 16-0 are used. Other bits are ignored.

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (w) | ModRM:r/m (r) | NA | NA |

...

## LTR—Load Task Register

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F 00 /3 | LTR *r/m*16 | A | Valid | Valid | Load *r/m*16 into task register. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:r/m (r) | NA | NA | NA |

...

## MASKMOVDQU—Store Selected Bytes of Double Quadword

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 66 0F F7 /*r* | MASKMOVDQU *xmm1*, *xmm2* | A | Valid | Valid | Selectively write bytes from *xmm1* to memory location using the byte mask in *xmm2*. The default memory location is specified by DS:EDI. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (r) | ModRM:r/m (r) | NA | NA |

### Description

Stores selected bytes from the source operand (first operand) into an 128-bit memory location. The mask operand (second operand) selects which bytes from the source operand are written to memory. The source and mask operands are XMM registers. The location of the first byte of the memory location is specified by DI/EDI and DS registers. The memory location does not need to be aligned on a natural boundary. (The size of the store address depends on the address-size attribute.)

The most significant bit in each byte of the mask operand determines whether the corresponding byte in the source operand is written to the corresponding byte location in memory: 0 indicates no write and 1 indicates write.

The MASKMOVDQU instruction generates a non-temporal hint to the processor to minimize cache pollution. The non-temporal hint is implemented by using a write combining (WC) memory type protocol (see “Caching of Temporal vs. Non-Temporal Data” in Chapter 10, of the *Intel® 64 and IA-32 Architectures 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 MASKMOVDQU instructions if multiple processors might use different memory types to read/write the destination memory locations.

...

## MASKMOVQ—Store Selected Bytes of Quadword

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F F7 /r | MASKMOVQ *mm1*, *mm2* | A | Valid | Valid | Selectively write bytes from *mm1* to memory location using the byte mask in *mm2*. The default memory location is specified by DS:EDI. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (r) | ModRM:r/m (r) | NA | NA |

...

## MAXPD—Return Maximum Packed Double-Precision Floating-Point Values

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 66 0F 5F /r | MAXPD *xmm1*, *xmm2/m128* | A | Valid | Valid | Return the maximum double-precision floating-point values between *xmm2/m128* and *xmm1*. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (r, w) | ModRM:r/m (r) | NA | NA |

...

## MAXPS—Return Maximum Packed Single-Precision Floating-Point Values

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F 5F /r | MAXPS *xmm1*, *xmm2/m128* | A | Valid | Valid | Return the maximum single-precision floating-point values between *xmm2/m128* and *xmm1*. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (r, w) | ModRM:r/m (r) | NA | NA |

...

## MAXSD—Return Maximum Scalar Double-Precision Floating-Point Value

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| F2 0F 5F /*r* | MAXSD *xmm1*, *xmm2/m64* | A | Valid | Valid | Return the maximum scalar double-precision floating-point value between *xmm2/mem64* and *xmm1*. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (r, w) | ModRM:r/m (r) | NA | NA |

...

## MAXSS—Return Maximum Scalar Single-Precision Floating-Point Value

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| F3 0F 5F /*r* | MAXSS *xmm1*, *xmm2/m32* | A | Valid | Valid | Return the maximum scalar single-precision floating-point value between *xmm2/mem32* and *xmm1*. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (r, w) | ModRM:r/m (r) | NA | NA |

...

## MFENCE—Memory Fence

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F AE /6 | MFENCE | A | Valid | Valid | Serializes load and store operations. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | NA | NA | NA | NA |

### Description

Performs a serializing operation on all load-from-memory and store-to-memory instructions that were issued prior the MFENCE instruction. This serializing operation guarantees that every load and store instruction that precedes the MFENCE instruction in program order becomes globally visible before any load or store instruction that follows the MFENCE instruction.[1] The MFENCE instruction is ordered with respect to all load and store instructions, other MFENCE instructions, any LFENCE and SFENCE instructions, and any serializing instructions (such as the CPUID instruction). MFENCE does not serialize the instruction stream.

Weakly ordered memory types can be used to achieve higher processor performance through such techniques as out-of-order issue, speculative reads, write-combining, and write-collapsing. The degree to which a consumer of data recognizes or knows that the data is weakly ordered varies among applications and may be unknown to the producer of this data. The MFENCE instruction provides a performance-efficient way of ensuring load and store ordering between routines that produce weakly-ordered results and routines that consume that data.

Processors are free to fetch and cache data speculatively from regions of system memory that use the WB, WC, and WT memory types. This speculative fetching can occur at any time and is not tied to instruction execution. Thus, it is not ordered with respect to executions of the MFENCE instruction; data can be brought into the caches speculatively just before, during, or after the execution of an MFENCE instruction.Processors are free to fetch and cache data speculatively from regions of system memory that use the WB, WC, and WT memory types. This speculative fetching can occur at any time and is not tied to instruction execution. Thus, it is not ordered with respect to executions of the MFENCE instruction; data can be brought into the caches speculatively just before, during, or after the execution of an MFENCE instruction.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

...

## MINPD—Return Minimum Packed Double-Precision Floating-Point Values

| **Opcode** | **Instruction** | **Op/ En** | **64-Bit Mode** | **Compat/ Leg Mode** | **Description** |
|---|---|---|---|---|---|
| 66 0F 5D /*r* | MINPD *xmm1*, *xmm2/m128* | A | Valid | Valid | Return the minimum double-precision floating-point values between *xmm2/m128* and *xmm1*. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (r, w) | ModRM:r/m (r) | NA | NA |

...

1. A load instruction is considered to become globally visible when the value to be loaded into its destination register is determined.

## MINPS—Return Minimum Packed Single-Precision Floating-Point Values

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F 5D /*r* | MINPS *xmm1*, *xmm2/m128* | A | Valid | Valid | Return the minimum single-precision floating-point values between *xmm2/m128* and *xmm1*. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (r, w) | ModRM:r/m (r) | NA | NA |

...

## MINSD—Return Minimum Scalar Double-Precision Floating-Point Value

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| F2 0F 5D /*r* | MINSD *xmm1*, *xmm2/m64* | A | Valid | Valid | Return the minimum scalar double-precision floating-point value between *xmm2/mem64* and *xmm1*. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (r, w) | ModRM:r/m (r) | NA | NA |

...

## MINSS—Return Minimum Scalar Single-Precision Floating-Point Value

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| F3 0F 5D /*r* | MINSS *xmm1*, *xmm2/m32* | A | Valid | Valid | Return the minimum scalar single-precision floating-point value between *xmm2/mem32* and *xmm1*. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (r, w) | ModRM:r/m (r) | NA | NA |

...

## MONITOR—Set Up Monitor Address

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F 01 *C8* | MONITOR | A | Valid | Valid | Sets up a linear address range to be monitored by hardware and activates the monitor. The address range should be a write-back memory caching type. The address is DS:EAX (DS:RAX in 64-bit mode). |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | NA | NA | NA | NA |

...

## MOV—Move

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 88 */r* | MOV *r/m8,r8* | A | Valid | Valid | Move *r8* to *r/m8.* |
| REX + 88 */r* | MOV *r/m8****,*r8**** | A | Valid | N.E. | Move *r8* to *r/m8.* |
| 89 */r* | MOV *r/m16,r16* | A | Valid | Valid | Move *r16* to *r/m16.* |
| 89 */r* | MOV *r/m32,r32* | A | Valid | Valid | Move *r32* to *r/m32.* |
| REX.W + 89 */r* | MOV *r/m64,r64* | A | Valid | N.E. | Move *r64* to *r/m64.* |
| 8A */r* | MOV *r8,r/m8* | B | Valid | Valid | Move *r/m8* to *r8.* |
| REX + 8A */r* | MOV *r8***,r/m8**** | B | Valid | N.E. | Move *r/m8* to *r8.* |
| 8B */r* | MOV *r16,r/m16* | B | Valid | Valid | Move *r/m16* to *r16.* |
| 8B */r* | MOV *r32,r/m32* | B | Valid | Valid | Move *r/m32* to *r32.* |
| REX.W + 8B */r* | MOV *r64,r/m64* | B | Valid | N.E. | Move *r/m64* to *r64.* |
| 8C */r* | MOV *r/m16,Sreg*** | A | Valid | Valid | Move segment register to *r/m16.* |
| REX.W + 8C */r* | MOV *r/m64,Sreg*** | A | Valid | Valid | Move zero extended 16-bit segment register to *r/m64.* |
| 8E */r* | MOV *Sreg,r/m16*** | B | Valid | Valid | Move *r/m16* to segment register. |
| REX.W + 8E */r* | MOV *Sreg,r/m64*** | B | Valid | Valid | Move *lower 16 bits of r/m64* to segment register. |
| A0 | MOV AL,*moffs8** | C | Valid | Valid | Move byte at (*seg:offset*) to AL. |
| REX.W + A0 | MOV AL,*moffs8** | C | Valid | N.E. | Move byte at (*offset*) to AL. |

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| A1 | MOV AX,*moffs16** | C | Valid | Valid | Move word at (*seg:offset*) to AX. |
| A1 | MOV EAX,*moffs32** | C | Valid | Valid | Move doubleword at (*seg:offset*) to EAX. |
| REX.W + A1 | MOV RAX,*moffs64** | C | Valid | N.E. | Move quadword at (*offset*) to RAX. |
| A2 | MOV *moffs8*,AL | D | Valid | Valid | Move AL to (*seg:offset*). |
| REX.W + A2 | MOV *moffs8****,AL | D | Valid | N.E. | Move AL to (*offset*). |
| A3 | MOV *moffs16**,AX | D | Valid | Valid | Move AX to (*seg:offset*). |
| A3 | MOV *moffs32**,EAX | D | Valid | Valid | Move EAX to (*seg:offset*). |
| REX.W + A3 | MOV *moffs64**,RAX | D | Valid | N.E. | Move RAX to (*offset*). |
| B0+ *rb* | MOV *r8, imm8* | E | Valid | Valid | Move *imm8* to *r8.* |
| REX + B0+ *rb* | MOV *r8****, imm8* | E | Valid | N.E. | Move *imm8* to *r8.* |
| B8+ *rw* | MOV *r16, imm16* | E | Valid | Valid | Move *imm16* to *r16.* |
| B8+ *rd* | MOV *r32, imm32* | E | Valid | Valid | Move *imm32* to *r32.* |
| REX.W + B8+ *rd* | MOV *r64, imm64* | E | Valid | N.E. | Move *imm64* to *r64.* |
| C6 */0* | MOV *r/m8, imm8* | F | Valid | Valid | Move *imm8* to *r/m8.* |
| REX + C6 */0* | MOV *r/m8***, imm8* | F | Valid | N.E. | Move *imm8* to *r/m8.* |
| C7 */0* | MOV *r/m16, imm16* | F | Valid | Valid | Move *imm16* to *r/m16.* |
| C7 */0* | MOV *r/m32, imm32* | F | Valid | Valid | Move *imm32* to *r/m32.* |
| REX.W + C7 */0* | MOV *r/m64, imm32* | F | Valid | N.E. | Move *imm32 sign extended to 64-bits* to *r/m64.* |

**NOTES:**

* The *moffs8*, *moffs16*, *moffs32* and *moffs64* operands specify a simple offset relative to the segment base, where 8, 16, 32 and 64 refer to the size of the data. The address-size attribute of the instruction determines the size of the offset, either 16, 32 or 64 bits.

** In 32-bit mode, the assembler may insert the 16-bit operand-size prefix with this instruction (see the following “Description” section for further information).

***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 |
|---|---|---|---|---|
| A | ModRM:r/m (w) | ModRM:reg (r) | NA | NA |
| B | ModRM:reg (w) | ModRM:r/m (r) | NA | NA |
| C | AL/AX/EAX/RAX | Displacement | NA | NA |
| D | Displacement | AL/AX/EAX/RAX | NA | NA |
| E | reg (w) | imm8/16/32/64 | NA | NA |
| F | ModRM:r/m (w) | imm8/16/32/64 | NA | NA |

...

## MOV—Move to/from Control Registers

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F 20/*r* | MOV *r32*, CR0–CR7 | A | N.E. | Valid | Move control register to *r32* |
| 0F 20/*r* | MOV *r64*, CR0–CR7 | A | Valid | N.E. | Move extended control register to *r64*. |
| REX.R + 0F 20 /0 | MOV *r64*, CR8 | A | Valid | N.E. | Move extended CR8 to *r64*.[1] |
| 0F 22 /*r* | MOV CR0–CR7, *r32* | A | N.E. | Valid | Move *r32* to control register |
| 0F 22 /*r* | MOV CR0–CR7, *r64* | A | Valid | N.E. | Move *r64* to extended control register. |
| REX.R + 0F 22 /0 | MOV CR8, *r64* | A | Valid | N.E. | Move *r64* to extended CR8.[1] |

**NOTE:**

1. MOV CR* instructions, except for MOV CR8, are serializing instructions. MOV CR8 is not architecturally defined as a serializing instruction. For more information, see Chapter 8 in *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 |
|---|---|---|---|---|
| A | ModRM:reg (w) | ModRM:r/m (r) | NA | NA |

...

## MOV—Move to/from Debug Registers

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F 21/*r* | MOV *r32*, DR0–DR7 | A | N.E. | Valid | Move debug register to *r32* |
| 0F 21/*r* | MOV *r64*, DR0–DR7 | A | Valid | N.E. | Move extended debug register to *r64*. |
| 0F 23 /*r* | MOV DR0–DR7, *r32* | A | N.E. | Valid | Move *r32* to debug register |
| 0F 23 /*r* | MOV DR0–DR7, *r64* | A | Valid | N.E. | Move *r64* to extended debug register. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (w) | ModRM:r/m (r) | NA | NA |

...

## MOVAPD—Move Aligned Packed Double-Precision Floating-Point Values

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 66 0F 28 /*r* | MOVAPD *xmm1*, *xmm2/m128* | A | Valid | Valid | Move packed double-precision floating-point values from *xmm2/m128* to *xmm1*. |
| 66 0F 29 /*r* | MOVAPD *xmm2/m128*, *xmm1* | B | Valid | Valid | Move packed double-precision floating-point values from *xmm1* to *xmm2/m128*. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (w) | ModRM:r/m (r) | NA | NA |
| B | ModRM:r/m (w) | ModRM:reg (r) | NA | NA |

...

## MOVAPS—Move Aligned Packed Single-Precision Floating-Point Values

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F 28 */r* | MOVAPS *xmm1*, *xmm2/m128* | A | Valid | Valid | Move packed single-precision floating-point values from *xmm2/m128* to *xmm1*. |
| 0F 29 */r* | MOVAPS *xmm2/m128*, *xmm1* | B | Valid | Valid | Move packed single-precision floating-point values from *xmm1* to *xmm2/m128*. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (w) | ModRM:r/m (r) | NA | NA |
| B | ModRM:r/m (w) | ModRM:reg (r) | NA | NA |

...

## MOVBE—Move Data After Swapping Bytes

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F 38 F0 */r* | MOVBE *r16*, m16 | A | Valid | Valid | Reverse byte order in *m16 and move to r16* |
| 0F 38 F0 */r* | MOVBE *r32*, m32 | A | Valid | Valid | Reverse byte order in *m32 and move to r32* |
| REX.W + 0F 38 F0 */r* | MOVBE *r64*, m64 | A | Valid | N.E. | Reverse byte order in *m64 and move to r64*. |
| 0F 38 F1 */r* | MOVBE *m16*, r16 | B | Valid | Valid | Reverse byte order in *r16 and move to m16* |
| 0F 38 F1 */r* | MOVBE *m32*, r32 | B | Valid | Valid | Reverse byte order in *r32 and move to m32* |
| REX.W + 0F 38 F1 */r* | MOVBE *m64*, r64 | B | Valid | N.E. | Reverse byte order in *r64 and move to m64*. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (w) | ModRM:r/m (r) | NA | NA |
| B | ModRM:r/m (w) | ModRM:reg (r) | NA | NA |

...

## MOVD/MOVQ—Move Doubleword/Move Quadword

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F 6E */r* | MOVD *mm, r/m32* | A | Valid | Valid | Move doubleword from *r/m32* to *mm.* |
| REX.W + 0F 6E */r* | MOVQ *mm, r/m64* | A | Valid | N.E. | Move quadword from *r/m64* to *mm.* |
| 0F 7E */r* | MOVD *r/m32, mm* | B | Valid | Valid | Move doubleword from *mm* to *r/m32.* |
| REX.W + 0F 7E */r* | MOVQ *r/m64, mm* | B | Valid | N.E. | Move quadword from *mm* to *r/m64.* |
| 66 0F 6E */r* | MOVD *xmm, r/m32* | A | Valid | Valid | Move doubleword from *r/m32* to *xmm.* |
| 66 REX.W 0F 6E */r* | MOVQ *xmm, r/m64* | A | Valid | N.E. | Move quadword from *r/m64* to *xmm.* |
| 66 0F 7E */r* | MOVD *r/m32, xmm* | B | Valid | Valid | Move doubleword from *xmm* register to *r/m32.* |
| 66 REX.W 0F 7E */r* | MOVQ *r/m64, xmm* | B | Valid | N.E. | Move quadword from *xmm* register to *r/m64.* |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (w) | ModRM:r/m (r) | NA | NA |
| B | ModRM:r/m (w) | ModRM:reg (r) | NA | NA |

...

## MOVDDUP—Move One Double-FP and Duplicate

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| F2 0F 12 */r* | MOVDDUP *xmm1, xmm2/m64* | A | Valid | Valid | Move one double-precision floating-point value from the lower 64-bit operand in *xmm2/m64* to *xmm1* and duplicate. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (w) | ModRM:r/m (r) | NA | NA |

...

## MOVDQA—Move Aligned Double Quadword

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 66 0F 6F /*r* | MOVDQA *xmm1*, *xmm2/m128* | A | Valid | Valid | Move aligned double quadword from *xmm2/m128* to *xmm1*. |
| 66 0F 7F /*r* | MOVDQA *xmm2/m128*, *xmm1* | B | Valid | Valid | Move aligned double quadword from *xmm1* to *xmm2/m128*. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (w) | ModRM:r/m (r) | NA | NA |
| B | ModRM:r/m (w) | ModRM:reg (r) | NA | NA |

...

## MOVDQU—Move Unaligned Double Quadword

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| F3 0F 6F /*r* | MOVDQU *xmm1*, *xmm2/m128* | A | Valid | Valid | Move unaligned double quadword from *xmm2/m128* to *xmm1*. |
| F3 0F 7F /*r* | MOVDQU *xmm2/m128*, *xmm1* | B | Valid | Valid | Move unaligned double quadword from *xmm1* to *xmm2/m128*. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (w) | ModRM:r/m (r) | NA | NA |
| B | ModRM:r/m (w) | ModRM:reg (r) | NA | NA |

...

## MOVDQ2Q—Move Quadword from XMM to MMX Technology Register

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| F2 0F D6 | MOVDQ2Q *mm*, *xmm* | A | Valid | Valid | Move low quadword from *xmm* to *mmx* register. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (w) | ModRM:reg (r) | NA | NA |

...

## MOVHLPS— Move Packed Single-Precision Floating-Point Values High to Low

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F 12 /*r* | MOVHLPS *xmm1, xmm2* | A | Valid | Valid | Move two packed single-precision floating-point values from high quadword of *xmm2* to low quadword of *xmm1*. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (w) | ModRM:reg (r) | NA | NA |

...

## MOVHPD—Move High Packed Double-Precision Floating-Point Value

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 66 0F 16 /*r* | MOVHPD *xmm*, *m64* | A | Valid | Valid | Move double-precision floating-point value from *m64* to high quadword of *xmm*. |
| 66 0F 17 /*r* | MOVHPD *m64*, *xmm* | B | Valid | Valid | Move double-precision floating-point value from high quadword of *xmm* to *m64*. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (r, w) | ModRM:r/m (r) | NA | NA |
| B | ModRM:r/m (w) | ModRM:reg (r) | NA | NA |

...

## MOVHPS—Move High Packed Single-Precision Floating-Point Values

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F 16 /*r* | MOVHPS *xmm*, *m64* | A | Valid | Valid | Move two packed single-precision floating-point values from *m64* to high quadword of *xmm*. |
| 0F 17 /*r* | MOVHPS *m64*, *xmm* | B | Valid | Valid | Move two packed single-precision floating-point values from high quadword of *xmm* to *m64*. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (r, w) | ModRM:r/m (r) | NA | NA |
| B | ModRM:r/m (w) | ModRM:reg (r) | NA | NA |

...

## MOVLHPS—Move Packed Single-Precision Floating-Point Values Low to High

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F 16 /*r* | MOVLHPS *xmm1*, *xmm2* | A | Valid | Valid | Move two packed single-precision floating-point values from low quadword of *xmm2* to high quadword of *xmm1*. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (w) | ModRM:reg (r) | NA | NA |

...

## MOVLPD—Move Low Packed Double-Precision Floating-Point Value

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 66 0F 12 */r* | MOVLPD *xmm*, *m64* | A | Valid | Valid | Move double-precision floating-point value from *m64* to low quadword of *xmm* register. |
| 66 0F 13 */r* | MOVLPD *m64*, *xmm* | B | Valid | Valid | Move double-precision floating-point nvalue from low quadword of *xmm* register to *m64*. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (r, w) | ModRM:r/m (r) | NA | NA |
| B | ModRM:r/m (w) | ModRM:reg (r) | NA | NA |

...

## MOVLPS—Move Low Packed Single-Precision Floating-Point Values

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F 12 */r* | MOVLPS *xmm*, *m64* | A | Valid | Valid | Move two packed single-precision floating-point values from *m64* to low quadword of *xmm*. |
| 0F 13 */r* | MOVLPS *m64*, *xmm* | B | Valid | Valid | Move two packed single-precision floating-point values from low quadword of *xmm* to *m64*. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (r, w) | ModRM:r/m (r) | NA | NA |
| B | ModRM:r/m (w) | ModRM:reg (r) | NA | NA |

...

## MOVMSKPD—Extract Packed Double-Precision Floating-Point Sign Mask

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 66 0F 50 /*r* | MOVMSKPD *reg*, *xmm* | A | Valid | Valid | Extract 2-bit sign mask from *xmm* and store in *reg*. The upper bits of r32 or r64 are filled with zeros. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (w) | ModRM:reg (r) | NA | NA |

...

## MOVMSKPS—Extract Packed Single-Precision Floating-Point Sign Mask

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F 50 /*r* | MOVMSKPS *reg*, *xmm* | A | Valid | Valid | Extract 4-bit sign mask from *xmm* and store in *reg*. The upper bits of r32 or r64 are filled with zeros. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (w) | ModRM:reg (r) | NA | NA |

...

## MOVNTDQA — Load Double Quadword Non-Temporal Aligned Hint

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 66 0F 38 2A /r | MOVNTDQA *xmm1, m128* | A | Valid | Valid | Move double quadword from *m128* to *xmm* using non-temporal hint if WC memory type. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (w) | ModRM:r/m (r) | NA | NA |

...

## MOVNTDQ—Store Double Quadword Using Non-Temporal Hint

| **Opcode** | **Instruction** | **Op/ En** | **64-Bit Mode** | **Compat/ Leg Mode** | **Description** |
|---|---|---|---|---|---|
| 66 0F E7 /*r* | MOVNTDQ *m128*, xmm | A | Valid | Valid | Move double quadword from *xmm* to *m128* using non-temporal hint. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:r/m (w) | ModRM:reg (r) | NA | NA |

...

## MOVNTI—Store Doubleword Using Non-Temporal Hint

| **Opcode** | **Instruction** | **Op/ En** | **64-Bit Mode** | **Compat/ Leg Mode** | **Description** |
|---|---|---|---|---|---|
| 0F C3 /*r* | MOVNTI *m32*, *r32* | A | Valid | Valid | Move doubleword from *r32* to *m32* using non-temporal hint. |
| REX.W + 0F C3 /*r* | MOVNTI *m64*, *r64* | A | Valid | N.E. | Move quadword from *r64* to *m64* using non-temporal hint. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:r/m (w) | ModRM:reg (r) | NA | NA |

...

## MOVNTPD—Store Packed Double-Precision Floating-Point Values Using Non-Temporal Hint

| **Opcode** | **Instruction** | **Op/ En** | **64-Bit Mode** | **Compat/ Leg Mode** | **Description** |
|---|---|---|---|---|---|
| 66 0F 2B /*r* | MOVNTPD *m128*, *xmm* | A | Valid | Valid | Move packed double-precision floating-point values from *xmm* to *m128* using non-temporal hint. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:r/m (w) | ModRM:reg (r) | NA | NA |

...

## MOVNTPS—Store Packed Single-Precision Floating-Point Values Using Non-Temporal Hint

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F 2B */r* | MOVNTPS *m128*, *xmm* | A | Valid | Valid | Move packed single-precision floating-point values from *xmm* to *m128* using non-temporal hint. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:r/m (w) | ModRM:reg (r) | NA | NA |

...

## MOVNTQ—Store of Quadword Using Non-Temporal Hint

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F E7 */r* | MOVNTQ *m64*, *mm* | A | Valid | Valid | Move quadword from *mm* to *m64* using non-temporal hint. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:r/m (w) | ModRM:reg (r) | NA | NA |

...

## MOVQ—Move Quadword

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F 6F /*r* | MOVQ *mm, mm/m64* | A | Valid | Valid | Move quadword from *mm/m64* to *mm*. |
| 0F 7F /*r* | MOVQ *mm/m64, mm* | B | Valid | Valid | Move quadword from *mm* to *mm/m64*. |
| F3 0F 7E | MOVQ *xmm1*, *xmm2/m64* | A | Valid | Valid | Move quadword from *xmm2/mem64* to *xmm1*. |
| 66 0F D6 | MOVQ *xmm2/m64*, *xmm1* | B | Valid | Valid | Move quadword from *xmm1* to *xmm2/mem64*. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (w) | ModRM:r/m (r) | NA | NA |
| B | ModRM:r/m (w) | ModRM:reg (r) | NA | NA |

...

## MOVQ2DQ—Move Quadword from MMX Technology to XMM Register

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| F3 0F D6 | MOVQ2DQ *xmm*, *mm* | A | Valid | Valid | Move quadword from *mmx* to low quadword of *xmm*. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (w) | ModRM:reg (r) | NA | NA |

...

## MOVS/MOVSB/MOVSW/MOVSD/MOVSQ—Move Data from String to String

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| A4 | MOVS *m8*, *m8* | A | Valid | Valid | For legacy mode, Move byte from address DS:(E)SI to ES:(E)DI. For 64-bit mode move byte from address (R\|E)SI to (R\|E)DI. |
| A5 | MOVS *m16*, *m16* | A | Valid | Valid | For legacy mode, move word from address DS:(E)SI to ES:(E)DI. For 64-bit mode move word at address (R\|E)SI to (R\|E)DI. |
| A5 | MOVS *m32*, *m32* | A | Valid | Valid | For legacy mode, move dword from address DS:(E)SI to ES:(E)DI. For 64-bit mode move dword from address (R\|E)SI to (R\|E)DI. |
| REX.W + A5 | MOVS *m64*, *m64* | A | Valid | N.E. | Move qword from address (R\|E)SI to (R\|E)DI. |
| A4 | MOVSB | A | Valid | Valid | For legacy mode, Move byte from address DS:(E)SI to ES:(E)DI. For 64-bit mode move byte from address (R\|E)SI to (R\|E)DI. |
| A5 | MOVSW | A | Valid | Valid | For legacy mode, move word from address DS:(E)SI to ES:(E)DI. For 64-bit mode move word at address (R\|E)SI to (R\|E)DI. |
| A5 | MOVSD | A | Valid | Valid | For legacy mode, move dword from address DS:(E)SI to ES:(E)DI. For 64-bit mode move dword from address (R\|E)SI to (R\|E)DI. |
| REX.W + A5 | MOVSQ | A | Valid | N.E. | Move qword from address (R\|E)SI to (R\|E)DI. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | NA | NA | NA | NA |

...

## MOVSD—Move Scalar Double-Precision Floating-Point Value

| **Opcode** | **Instruction** | **Op/ En** | **64-Bit Mode** | **Compat/ Leg Mode** | **Description** |
|---|---|---|---|---|---|
| F2 0F 10 /*r* | MOVSD *xmm1*, *xmm2/m64* | A | Valid | Valid | Move scalar double-precision floating-point value from *xmm2/m64* to *xmm1* register. |
| F2 0F 11 /*r* | MOVSD *xmm2/m64*, *xmm1* | B | Valid | Valid | Move scalar double-precision floating-point value from *xmm1* register to *xmm2/m64*. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (w) | ModRM:r/m (r) | NA | NA |
| B | ModRM:r/m (w) | ModRM:reg (r) | NA | NA |

...

## MOVSHDUP—Move Packed Single-FP High and Duplicate

| **Opcode** | **Instruction** | **Op/ En** | **64-Bit Mode** | **Compat/ Leg Mode** | **Description** |
|---|---|---|---|---|---|
| F3 0F 16 /*r* | MOVSHDUP *xmm1*, *xmm2/m128* | A | Valid | Valid | Move two single-precision floating-point values from the higher 32-bit operand of each qword in *xmm2/m128* to *xmm1* and duplicate each 32-bit operand to the lower 32-bits of each qword. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (w) | ModRM:r/m (r) | NA | NA |

...

## MOVSLDUP—Move Packed Single-FP Low and Duplicate

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| F3 0F 12 /*r* | MOVSLDUP *xmm1*, *xmm2/m128* | A | Valid | Valid | Move two single-precision floating-point values from the lower 32-bit operand of each qword in *xmm2/m128* to *xmm1* and duplicate each 32-bit operand to the higher 32-bits of each qword. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (w) | ModRM:r/m (r) | NA | NA |

...

## MOVSS—Move Scalar Single-Precision Floating-Point Values

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| F3 0F 10 /*r* | MOVSS *xmm1*, *xmm2/m32* | A | Valid | Valid | Move scalar single-precision floating-point value from *xmm2/m32* to *xmm1* register. |
| F3 0F 11 /*r* | MOVSS *xmm2/m32*, *xmm* | B | Valid | Valid | Move scalar single-precision floating-point value from *xmm1* register to *xmm2/m32*. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (w) | ModRM:r/m (r) | NA | NA |
| B | ModRM:r/m (w) | ModRM:reg (r) | NA | NA |

...

## MOVSX/MOVSXD—Move with Sign-Extension

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F BE /r | MOVSX *r16, r/m8* | A | Valid | Valid | Move byte to word with sign-extension. |
| 0F BE /r | MOVSX *r32, r/m8* | A | Valid | Valid | Move byte to doubleword with sign-extension. |
| REX + 0F BE /r | MOVSX *r64, r/m8** | A | Valid | N.E. | Move byte to quadword with sign-extension. |
| 0F BF /r | MOVSX *r32, r/m16* | A | Valid | Valid | Move word to doubleword, with sign-extension. |
| REX.W + 0F BF /r | MOVSX *r64, r/m16* | A | Valid | N.E. | Move word to quadword with sign-extension. |
| REX.W** + 63 /r | MOVSXD *r64, r/m32* | A | Valid | N.E. | Move doubleword to quadword with sign-extension. |

**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.

** The use of MOVSXD without REX.W in 64-bit mode is discouraged, Regular MOV should be used instead of using MOVSXD without REX.W.

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (w) | ModRM:r/m (r) | NA | NA |

...

## MOVUPD—Move Unaligned Packed Double-Precision Floating-Point Values

| **Opcode** | **Instruction** | Op/ En | **64-Bit Mode** | **Compat/ Leg Mode** | **Description** |
|---|---|---|---|---|---|
| 66 0F 10 /*r* | MOVUPD *xmm1*, *xmm2/m128* | A | Valid | Valid | Move packed double-precision floating-point values from *xmm2/m128* to xmm1. |
| 66 0F 11 /*r* | MOVUPD *xmm2/m128*, *xmm* | B | Valid | Valid | Move packed double-precision floating-point values from xmm1 to *xmm2/m128*. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (w) | ModRM:r/m (r) | NA | NA |
| B | ModRM:r/m (w) | ModRM:reg (r) | NA | NA |

...

## MOVUPS—Move Unaligned Packed Single-Precision Floating-Point Values

| **Opcode** | **Instruction** | Op/ En | **64-Bit Mode** | **Compat/ Leg Mode** | **Description** |
|---|---|---|---|---|---|
| 0F 10 /*r* | MOVUPS *xmm1*, *xmm2/m128* | A | Valid | Valid | Move packed single-precision floating-point values from *xmm2/m128* to xmm1. |
| 0F 11 /*r* | MOVUPS *xmm2/m128*, *xmm1* | B | Valid | Valid | Move packed single-precision floating-point values from xmm1 to *xmm2/m128*. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (w) | ModRM:r/m (r) | NA | NA |
| B | ModRM:r/m (w) | ModRM:reg (r) | NA | NA |

...

## MOVZX—Move with Zero-Extend

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F B6 /*r* | MOVZX *r16, r/m8* | A | Valid | Valid | Move byte to word with zero-extension. |
| 0F B6 /*r* | MOVZX *r32, r/m8* | A | Valid | Valid | Move byte to doubleword, zero-extension. |
| REX.W + 0F B6 /*r* | MOVZX *r64, r/m8** | A | Valid | N.E. | Move byte to quadword, zero-extension. |
| 0F B7 /*r* | MOVZX *r32, r/m16* | A | Valid | Valid | Move word to doubleword, zero-extension. |
| REX.W + 0F B7 /*r* | MOVZX *r64, r/m16* | A | Valid | N.E. | Move word to quadword, zero-extension. |

**NOTES:**

* In 64-bit mode, r/m8 can not be encoded to access the following byte registers if the REX prefix is used: AH, BH, CH, DH.

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (w) | ModRM:r/m (r) | NA | NA |

...

## MPSADBW — Compute Multiple Packed Sums of Absolute Difference

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 66 0F 3A 42 /r ib | MPSADBW *xmm1, xmm2/m128, imm8* | A | Valid | Valid | Sums absolute 8-bit integer difference of adjacent groups of 4 byte integers in *xmm1* and *xmm2/m128* and writes the results in *xmm1*. Starting offsets within *xmm1* and *xmm2/m128* are determined by *imm8*. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (r, w) | ModRM:r/m (r) | imm8 | NA |

...

## MUL—Unsigned Multiply

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| F6 /4 | MUL *r/m8* | A | Valid | Valid | Unsigned multiply (AX ← AL ∗ *r/m8*). |
| REX + F6 /4 | MUL *r/m8** | A | Valid | N.E. | Unsigned multiply (AX ← AL ∗ *r/m8*). |
| F7 /4 | MUL *r/m16* | A | Valid | Valid | Unsigned multiply (DX:AX ← AX ∗ *r/m16*). |
| F7 /4 | MUL *r/m32* | A | Valid | Valid | Unsigned multiply (EDX:EAX ← EAX ∗ *r/m32*). |
| REX.W + F7 /4 | MUL *r/m64* | A | Valid | N.E. | Unsigned multiply (RDX:RAX ← RAX ∗ *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 |
|---|---|---|---|---|
| A | ModRM:r/m (r) | NA | NA | NA |

...

## MULPD—Multiply Packed Double-Precision Floating-Point Values

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 66 0F 59 /*r* | MULPD *xmm1*, *xmm2/m128* | A | Valid | Valid | Multiply packed double-precision floating-point values in *xmm2/m128* by *xmm1*. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (r, w) | ModRM:r/m (r) | NA | NA |

...

## MULPS—Multiply Packed Single-Precision Floating-Point Values

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F 59 /*r* | MULPS *xmm1*, *xmm2/m128* | A | Valid | Valid | Multiply packed single-precision floating-point values in *xmm2/mem* by *xmm1*. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (r, w) | ModRM:r/m (r) | NA | NA |

...

## MULSD—Multiply Scalar Double-Precision Floating-Point Values

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| F2 0F 59 /*r* | MULSD *xmm1*, *xmm2/m64* | A | Valid | Valid | Multiply the low double-precision floating-point value in *xmm2/mem64* by low double-precision floating-point value in *xmm1*. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (r, w) | ModRM:r/m (r) | NA | NA |

...

## MULSS—Multiply Scalar Single-Precision Floating-Point Values

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| F3 0F 59 /*r* | MULSS *xmm1*, *xmm2/m32* | A | Valid | Valid | Multiply the low single-precision floating-point value in *xmm2/mem* by the low single-precision floating-point value in *xmm1*. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (r, w) | ModRM:r/m (r) | NA | NA |

...

## MWAIT—Monitor Wait

| Opcode | Instruction | Op/En | 64-Bit Mode | Compat/Leg Mode | Description |
|---|---|---|---|---|---|
| 0F 01 *C9* | MWAIT | A | Valid | Valid | A hint that allow the processor to stop instruction execution and enter an implementation-dependent optimized state until occurrence of a class of events. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | NA | NA | NA | NA |

...

## 2. 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, N-Z.

------------------------------------------------------------------------------------------

...

## NEG—Two's Complement Negation

| Opcode | Instruction | Op/En | 64-Bit Mode | Compat/Leg Mode | Description |
|---|---|---|---|---|---|
| F6 /3 | NEG *r/m8* | A | Valid | Valid | Two's complement negate *r/m8*. |
| REX + F6 /3 | NEG *r/m8** | A | Valid | N.E. | Two's complement negate *r/m8*. |
| F7 /3 | NEG *r/m16* | A | Valid | Valid | Two's complement negate *r/m16*. |
| F7 /3 | NEG *r/m32* | A | Valid | Valid | Two's complement negate *r/m32*. |
| REX.W + F7 /3 | NEG *r/m64* | A | Valid | N.E. | Two's complement negate *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 |
|---|---|---|---|---|
| A | ModRM:r/m (r, w) | NA | NA | NA |

...

## NOP—No Operation

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 90 | NOP | A | Valid | Valid | One byte no-operation instruction. |
| 0F 1F /0 | NOP r/m16 | B | Valid | Valid | Multi-byte no-operation instruction. |
| 0F 1F /0 | NOP r/m32 | B | Valid | Valid | Multi-byte no-operation instruction. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | NA | NA | NA | NA |
| B | ModRM:r/m (r) | NA | NA | NA |

...

## NOT—One's Complement Negation

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| F6 /2 | NOT *r/m8* | A | Valid | Valid | Reverse each bit of *r/m8.* |
| REX + F6 /2 | NOT *r/m8** | A | Valid | N.E. | Reverse each bit of *r/m8.* |
| F7 /2 | NOT *r/m16* | A | Valid | Valid | Reverse each bit of *r/m16.* |
| F7 /2 | NOT *r/m32* | A | Valid | Valid | Reverse each bit of *r/m32.* |
| REX.W + F7 /2 | NOT *r/m64* | A | Valid | N.E. | Reverse each bit of *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 |
|---|---|---|---|---|
| A | ModRM:r/m (r, w) | NA | NA | NA |

...

## OR—Logical Inclusive OR

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0C *ib* | OR AL, *imm8* | A | Valid | Valid | AL OR *imm8.* |
| 0D *iw* | OR AX, *imm16* | A | Valid | Valid | AX OR *imm16.* |
| 0D *id* | OR EAX, *imm32* | A | Valid | Valid | EAX OR *imm32.* |
| REX.W + 0D *id* | OR RAX, *imm32* | A | Valid | N.E. | RAX OR *imm32 (sign-extended).* |
| 80 /1 *ib* | OR *r/m8, imm8* | B | Valid | Valid | *r/m8* OR *imm8.* |
| REX + 80 /1 *ib* | OR *r/m8**, *imm8* | B | Valid | N.E. | *r/m8* OR *imm8.* |
| 81 /1 *iw* | OR *r/m16, imm16* | B | Valid | Valid | *r/m16* OR *imm16.* |
| 81 /1 *id* | OR *r/m32, imm32* | B | Valid | Valid | *r/m32* OR *imm32.* |
| REX.W + 81 /1 *id* | OR *r/m64, imm32* | B | Valid | N.E. | *r/m64* OR *imm32 (sign-extended).* |
| 83 /1 *ib* | OR *r/m16, imm8* | B | Valid | Valid | *r/m16* OR *imm8 (sign-extended).* |
| 83 /1 *ib* | OR *r/m32, imm8* | B | Valid | Valid | *r/m32* OR *imm8 (sign-extended).* |
| REX.W + 83 /1 *ib* | OR *r/m64, imm8* | B | Valid | N.E. | *r/m64* OR *imm8 (sign-extended).* |
| 08 /*r* | OR *r/m8, r8* | C | Valid | Valid | *r/m8* OR *r8.* |
| REX + 08 /*r* | OR *r/m8**, *r8** | C | Valid | N.E. | *r/m8* OR *r8.* |
| 09 /*r* | OR *r/m16, r16* | C | Valid | Valid | *r/m16* OR *r16.* |
| 09 /*r* | OR *r/m32, r32* | C | Valid | Valid | *r/m32* OR *r32.* |
| REX.W + 09 /*r* | OR *r/m64, r64* | C | Valid | N.E. | *r/m64* OR *r64.* |
| 0A /*r* | OR *r8, r/m8* | D | Valid | Valid | *r8* OR *r/m8.* |
| REX + 0A /*r* | OR *r8**, *r/m8** | D | Valid | N.E. | *r8* OR *r/m8.* |
| 0B /*r* | OR *r16, r/m16* | D | Valid | Valid | *r16* OR *r/m16.* |
| 0B /*r* | OR *r32, r/m32* | D | Valid | Valid | *r32* OR *r/m32.* |
| REX.W + 0B /*r* | OR *r64, r/m64* | D | Valid | N.E. | *r64* OR *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 |
|---|---|---|---|---|
| A | AL/AX/EAX/RAX | imm8/16/32 | NA | NA |
| B | ModRM:r/m (r, w) | imm8/16/32 | NA | NA |
| C | ModRM:r/m (r, w) | ModRM:reg (r) | NA | NA |
| D | ModRM:reg (r, w) | ModRM:r/m (r) | NA | NA |

...

## ORPD—Bitwise Logical OR of Double-Precision Floating-Point Values

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 66 0F 56 */r* | ORPD *xmm1*, *xmm2/m128* | A | Valid | Valid | Bitwise OR of *xmm2/m128* and *xmm1*. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (r, w) | ModRM:r/m (r) | NA | NA |

...

## ORPS—Bitwise Logical OR of Single-Precision Floating-Point Values

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F 56 */r* | ORPS *xmm1*, *xmm2/m128* | A | Valid | Valid | Bitwise OR of *xmm2/m128* and *xmm1*. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (r, w) | ModRM:r/m (r) | NA | NA |

...

## OUT—Output to Port

| Opcode* | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| E6 *ib* | OUT *imm8*, AL | A | Valid | Valid | Output byte in AL to I/O port address *imm8*. |
| E7 *ib* | OUT *imm8*, AX | A | Valid | Valid | Output word in AX to I/O port address *imm8*. |
| E7 *ib* | OUT *imm8*, EAX | A | Valid | Valid | Output doubleword in EAX to I/O port address *imm8*. |
| EE | OUT DX, AL | B | Valid | Valid | Output byte in AL to I/O port address in DX. |
| EF | OUT DX, AX | B | Valid | Valid | Output word in AX to I/O port address in DX. |
| EF | OUT DX, EAX | B | Valid | Valid | Output doubleword in EAX to I/O port address in DX. |

**NOTES:**

* See IA-32 Architecture Compatibility section below.

**Instruction Operand Encoding**

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | imm8 | NA | NA | NA |
| B | NA | NA | NA | NA |

...

### IA-32 Architecture Compatibility

After executing an OUT instruction, the Pentium® processor ensures that the EWBE# pin has been sampled active before it begins to execute the next instruction. (Note that the instruction can be prefetched if EWBE# is not active, but it will not be executed until the EWBE# pin is sampled active.) Only the Pentium processor family has the EWBE# pin.

...

## OUTS/OUTSB/OUTSW/OUTSD—Output String to Port

| Opcode* | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 6E | OUTS DX, *m8* | A | Valid | Valid | Output byte from memory location specified in DS:(E)SI or RSI to I/O port specified in DX**. |
| 6F | OUTS DX, *m16* | A | Valid | Valid | Output word from memory location specified in DS:(E)SI or RSI to I/O port specified in DX**. |
| 6F | OUTS DX, *m32* | A | Valid | Valid | Output doubleword from memory location specified in DS:(E)SI or RSI to I/O port specified in DX**. |
| 6E | OUTSB | A | Valid | Valid | Output byte from memory location specified in DS:(E)SI or RSI to I/O port specified in DX**. |
| 6F | OUTSW | A | Valid | Valid | Output word from memory location specified in DS:(E)SI or RSI to I/O port specified in DX**. |
| 6F | OUTSD | A | Valid | Valid | Output doubleword from memory location specified in DS:(E)SI or RSI to I/O port specified in DX**. |

**NOTES:**

* See IA-32 Architecture Compatibility section below.

** In 64-bit mode, only 64-bit (RSI) and 32-bit (ESI) address sizes are supported. In non-64-bit mode, only 32-bit (ESI) and 16-bit (SI) address sizes are supported.

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | NA | NA | NA | NA |

...

### IA-32 Architecture Compatibility

After executing an OUTS, OUTSB, OUTSW, or OUTSD instruction, the Pentium processor ensures that the EWBE# pin has been sampled active before it begins to execute the next instruction. (Note that the instruction can be prefetched if EWBE# is not active, but it will not be executed until the EWBE# pin is sampled active.) Only the Pentium processor family has the EWBE# pin.

...

## PABSB/PABSW/PABSD — Packed Absolute Value

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F 38 1C /r | PABSB mm1, mm2/m64 | A | Valid | Valid | Compute the absolute value of bytes in mm2/m64 and store UNSIGNED result in mm1. |
| 66 0F 38 1C /r | PABSB xmm1, xmm2/m128 | A | Valid | Valid | Compute the absolute value of bytes in xmm2/m128 and store UNSIGNED result in xmm1. |
| 0F 38 1D /r | PABSW mm1, mm2/m64 | A | Valid | Valid | Compute the absolute value of 16-bit integers in mm2/m64 and store UNSIGNED result in mm1. |
| 66 0F 38 1D /r | PABSW xmm1, xmm2/m128 | A | Valid | Valid | Compute the absolute value of 16-bit integers in xmm2/m128 and store UNSIGNED result in xmm1. |
| 0F 38 1E /r | PABSD mm1, mm2/m64 | A | Valid | Valid | Compute the absolute value of 32-bit integers in mm2/m64 and store UNSIGNED result in mm1. |
| 66 0F 38 1E /r | PABSD xmm1, xmm2/m128 | A | Valid | Valid | Compute the absolute value of 32-bit integers in xmm2/m128 and store UNSIGNED result in xmm1. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (w) | ModRM:r/m (r) | NA | NA |

...

## PACKSSWB/PACKSSDW—Pack with Signed Saturation

| **Opcode** | **Instruction** | **Op/ En** | **64-Bit Mode** | **Compat/ Leg Mode** | **Description** |
|---|---|---|---|---|---|
| 0F 63 /r | PACKSSWB *mm1, mm2/m64* | A | Valid | Valid | Converts 4 packed signed word integers from *mm1* and from *mm2/m64* into 8 packed signed byte integers in *mm1* using signed saturation. |
| 66 0F 63 */r* | PACKSSWB *xmm1, xmm2/m128* | A | Valid | Valid | Converts 8 packed signed word integers from *xmm1* and from *xxm2/m128* into 16 packed signed byte integers in *xxm1* using signed saturation. |
| 0F 6B */r* | PACKSSDW *mm1, mm2/m64* | A | Valid | Valid | Converts 2 packed signed doubleword integers from *mm1* and from *mm2/m64* into 4 packed signed word integers in *mm1* using signed saturation. |
| 66 0F 6B */r* | PACKSSDW *xmm1, xmm2/m128* | A | Valid | Valid | Converts 4 packed signed doubleword integers from *xmm1* and from *xxm2/m128* into 8 packed signed word integers in *xxm1* using signed saturation. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (r, w) | ModRM:r/m (r) | NA | NA |

...

## PACKUSDW — Pack with Unsigned Saturation

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 66 0F 38 2B /r | PACKUSDW *xmm1, xmm2/m128* | A | Valid | Valid | Convert 4 packed signed doubleword integers from *xmm1* and 4 packed signed doubleword integers from *xmm2/m128* into 8 packed unsigned word integers in *xmm1* using unsigned saturation. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (r, w) | ModRM:r/m (r) | NA | NA |

...

## PACKUSWB—Pack with Unsigned Saturation

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F 67 /r | PACKUSWB *mm, mm/m64* | A | Valid | Valid | Converts 4 signed word integers from *mm* and 4 signed word integers from *mm/m64* into 8 unsigned byte integers in *mm* using unsigned saturation. |
| 66 0F 67 /r | PACKUSWB *xmm1, xmm2/m128* | A | Valid | Valid | Converts 8 signed word integers from *xmm1* and 8 signed word integers from *xmm2/m128* into 16 unsigned byte integers in *xmm1* using unsigned saturation. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (r, w) | ModRM:r/m (r) | NA | NA |

...

## PADDB/PADDW/PADDD—Add Packed Integers

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F FC /*r* | PADDB *mm, mm/m64* | A | Valid | Valid | Add packed byte integers from *mm/m64* and *mm.* |
| 66 0F FC /*r* | PADDB *xmm1, xmm2/m128* | A | Valid | Valid | Add packed byte integers from *xmm2/m128* and *xmm1*. |
| 0F FD /*r* | PADDW *mm, mm/m64* | A | Valid | Valid | Add packed word integers from *mm/m64* and *mm.* |
| 66 0F FD /*r* | PADDW *xmm1, xmm2/m128* | A | Valid | Valid | Add packed word integers from *xmm2/m128* and *xmm1*. |
| 0F FE /*r* | PADDD *mm, mm/m64* | A | Valid | Valid | Add packed doubleword integers from *mm/m64* and *mm*. |
| 66 0F FE /*r* | PADDD *xmm1, xmm2/m128* | A | Valid | Valid | Add packed doubleword integers from *xmm2/m128* and *xmm1*. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (r, w) | ModRM:r/m (r) | NA | NA |

...

## PADDQ—Add Packed Quadword Integers

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F D4 /*r* | PADDQ *mm1, mm2/m64* | A | Valid | Valid | Add quadword integer *mm2/m64* to *mm1.* |
| 66 0F D4 /*r* | PADDQ *xmm1, xmm2/m128* | A | Valid | Valid | Add packed quadword integers *xmm2/m128* to *xmm1.* |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (r, w) | ModRM:r/m (r) | NA | NA |

...

## PADDSB/PADDSW—Add Packed Signed Integers with Signed Saturation

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F EC /*r* | PADDSB *mm, mm/m64* | A | Valid | Valid | Add packed signed byte integers from *mm/m64 and mm* and saturate the results. |
| 66 0F EC /*r* | PADDSB *xmm1, xmm2/m128* | A | Valid | Valid | Add packed signed byte integers from *xmm2/m128* and *xmm1* saturate the results. |
| 0F ED /*r* | PADDSW *mm, mm/m64* | A | Valid | Valid | Add packed signed word integers from *mm/m64 and mm* and saturate the results. |
| 66 0F ED /*r* | PADDSW *xmm1, xmm2/m128* | A | Valid | Valid | Add packed signed word integers from *xmm2/m128* and *xmm1* and saturate the results. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (r, w) | ModRM:r/m (r) | NA | NA |

...

## PADDUSB/PADDUSW—Add Packed Unsigned Integers with Unsigned Saturation

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F DC /*r* | PADDUSB *mm, mm/m64* | A | Valid | Valid | Add packed unsigned byte integers from *mm/m64 and mm* and saturate the results. |
| 66 0F DC /*r* | PADDUSB xmm1, xmm2/m128 | A | Valid | Valid | Add packed unsigned byte integers from *xmm2/m128* and *xmm1* saturate the results. |
| 0F DD /*r* | PADDUSW *mm, mm/m64* | A | Valid | Valid | Add packed unsigned word integers from *mm/m64 and mm* and saturate the results. |
| 66 0F DD /*r* | PADDUSW xmm1, xmm2/m128 | A | Valid | Valid | Add packed unsigned word integers from *xmm2/m128* to *xmm1* and saturate the results. |

**Instruction Operand Encoding**

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (r, w) | ModRM:r/m (r) | NA | NA |

...

## PALIGNR — Packed Align Right

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F 3A 0F | PALIGNR mm1, mm2/m64, imm8 | A | Valid | Valid | Concatenate destination and source operands, extract byte-aligned result shifted to the right by constant value in imm8 into mm1. |
| 66 0F 3A 0F | PALIGNR xmm1, xmm2/m128, imm8 | A | Valid | Valid | Concatenate destination and source operands, extract byte-aligned result shifted to the right by constant value in imm8 into xmm1 |

**Instruction Operand Encoding**

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (r, w) | ModRM:r/m (r) | imm8 | NA |

...

## PAND—Logical AND

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F DB /*r* | PAND *mm, mm/m64* | A | Valid | Valid | Bitwise AND *mm/m64* and *mm.* |
| 66 0F DB /*r* | PAND *xmm1, xmm2/m128* | A | Valid | Valid | Bitwise AND of *xmm2/m128* and *xmm1*. |

**Instruction Operand Encoding**

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (r, w) | ModRM:r/m (r) | NA | NA |

...

## PANDN—Logical AND NOT

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F DF /*r* | PANDN *mm, mm/m64* | A | Valid | Valid | Bitwise AND NOT of *mm/m64* and *mm*. |
| 66 0F DF /*r* | PANDN *xmm1, xmm2/m128* | A | Valid | Valid | Bitwise AND NOT of *xmm2/m128* and *xmm1*. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (r, w) | ModRM:r/m (r) | NA | NA |

...

## PAUSE—Spin Loop Hint

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| F3 90 | PAUSE | A | Valid | Valid | Gives hint to processor that improves performance of spin-wait loops. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | NA | NA | NA | NA |

...

## PAVGB/PAVGW—Average Packed Integers

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F E0 /*r* | PAVGB *mm1, mm2/m64* | A | Valid | Valid | Average packed unsigned byte integers from mm2/m64 and mm1 with rounding. |
| 66 0F E0, /*r* | PAVGB *xmm1, xmm2/m128* | A | Valid | Valid | Average packed unsigned byte integers from *xmm2/m128* and *xmm1* with rounding. |
| 0F E3 /*r* | PAVGW *mm1, mm2/m64* | A | Valid | Valid | Average packed unsigned word integers from mm2/m64 and mm1 with rounding. |
| 66 0F E3 /*r* | PAVGW *xmm1, xmm2/m128* | A | Valid | Valid | Average packed unsigned word integers from *xmm2/m128* and *xmm1* with rounding. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (r, w) | ModRM:r/m (r) | NA | NA |

...

## PBLENDVB — Variable Blend Packed Bytes

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 66 0F 38 10 /r | PBLENDVB *xmm1, xmm2/m128, <XMM0>* | A | Valid | Valid | Select byte values from *xmm1* and *xmm2/m128* from mask specified in the high bit of each byte in *XMM0* and store the values into *xmm1*. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (r, w) | ModRM:r/m (r) | <XMM0> | NA |

...

## PBLENDW — Blend Packed Words

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 66 0F 3A 0E /r ib | PBLENDW *xmm1, xmm2/m128, imm8* | A | Valid | Valid | Select words from *xmm1* and *xmm2/m128* from mask specified in *imm8* and store the values into *xmm1*. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (r, w) | ModRM:r/m (r) | imm8 | NA |

...

## PCMPEQB/PCMPEQW/PCMPEQD— Compare Packed Data for Equal

| **Opcode** | **Instruction** | Op/ En | **64-Bit Mode** | **Compat/ Leg Mode** | **Description** |
|---|---|---|---|---|---|
| 0F 74 /*r* | PCMPEQB *mm, mm/m64* | A | Valid | Valid | Compare packed bytes in *mm/m64* and *mm* for equality. |
| 66 0F 74 /*r* | PCMPEQB *xmm1, xmm2/m128* | A | Valid | Valid | Compare packed bytes in *xmm2/m128* and xmm1 for equality. |
| 0F 75 /*r* | PCMPEQW *mm, mm/m64* | A | Valid | Valid | Compare packed words in *mm/m64* and *mm* for equality. |
| 66 0F 75 /*r* | PCMPEQW *xmm1, xmm2/m128* | A | Valid | Valid | Compare packed words in *xmm2/m128* and xmm1 for equality. |
| 0F 76 /*r* | PCMPEQD *mm, mm/m64* | A | Valid | Valid | Compare packed doublewords in *mm/m64* and *mm* for equality. |
| 66 0F 76 /*r* | PCMPEQD *xmm1, xmm2/m128* | A | Valid | Valid | Compare packed doublewords in *xmm2/m128* and xmm1 for equality. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (r, w) | ModRM:r/m (r) | NA | NA |

...

## PCMPEQQ — Compare Packed Qword Data for Equal

| **Opcode** | **Instruction** | Op/ En | **64-Bit Mode** | **Compat/ Leg Mode** | **Description** |
|---|---|---|---|---|---|
| 66 0F 38 29 /r | PCMPEQQ *xmm1, xmm2/m128* | A | Valid | Valid | Compare packed qwords in *xmm2/m128* and *xmm1* for equality. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (r, w) | ModRM:r/m (r) | NA | NA |

...

## PCMPESTRI — Packed Compare Explicit Length Strings, Return Index

| Opcode | Instruction | Op/En | 64-Bit Mode | Compat/Leg Mode | Description |
|---|---|---|---|---|---|
| 66 0F 3A 61 */r* *imm8* | PCMPESTRI *xmm1*, *xmm2/m128*, *imm8* | A | Valid | Valid | Perform a packed comparison of string data with explicit lengths, generating an index, and storing the result in ECX. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (r) | ModRM:r/m (r) | imm8 | NA |

...

## PCMPESTRM — Packed Compare Explicit Length Strings, Return Mask

| Opcode | Instruction | Op/En | 64-Bit Mode | Compat/Leg Mode | Description |
|---|---|---|---|---|---|
| 66 0F 3A 60 /r imm8 | PCMPESTRM *xmm1*, *xmm2/m128*, *imm8* | A | Valid | Valid | Perform a packed comparison of string data with explicit lengths, generating a mask, and storing the result in *XMM0* |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (r) | ModRM:r/m (r) | imm8 | NA |

...

## PCMPISTRI — Packed Compare Implicit Length Strings, Return Index

| Opcode | Instruction | Op/En | 64-Bit Mode | Compat/Leg Mode | Description |
|---|---|---|---|---|---|
| 66 0F 3A 63 */r* *imm8* | PCMPISTRI *xmm1*, *xmm2/m128*, *imm8* | A | Valid | Valid | Perform a packed comparison of string data with implicit lengths, generating an index, and storing the result in ECX. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (r) | ModRM:r/m (r) | imm8 | NA |

...

## PCMPISTRM — Packed Compare Implicit Length Strings, Return Mask

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 66 0F 3A 62 /r *imm8* | PCMPISTRM *xmm1, xmm2/m128, imm8* | A | Valid | Valid | Perform a packed comparison of string data with implicit lengths, generating a mask, and storing the result in *XMM0.* |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (r) | ModRM:r/m (r) | imm8 | NA |

...

## PCMPGTB/PCMPGTW/PCMPGTD—Compare Packed Signed Integers for Greater Than

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F 64 /r | PCMPGTB *mm, mm/m64* | A | Valid | Valid | Compare packed signed byte integers in *mm* and *mm/m64* for greater than. |
| 66 0F 64 /r | PCMPGTB *xmm1, xmm2/m128* | A | Valid | Valid | Compare packed signed byte integers in *xmm1* and *xmm2/m128* for greater than. |
| 0F 65 /r | PCMPGTW *mm, mm/m64* | A | Valid | Valid | Compare packed signed word integers in *mm* and *mm/m64* for greater than. |
| 66 0F 65 /r | PCMPGTW *xmm1, xmm2/m128* | A | Valid | Valid | Compare packed signed word integers in *xmm1* and *xmm2/m128* for greater than. |
| 0F 66 /r | PCMPGTD *mm, mm/m64* | A | Valid | Valid | Compare packed signed doubleword integers in *mm* and *mm/m64* for greater than. |
| 66 0F 66 /r | PCMPGTD *xmm1, xmm2/m128* | A | Valid | Valid | Compare packed signed doubleword integers in *xmm1* and *xmm2/m128* for greater than. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (r, w) | ModRM:r/m (r) | NA | NA |

...

## PCMPGTQ — Compare Packed Data for Greater Than

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 66 0F 38 37 /r | PCMPGTQ *xmm1,xmm2/m128* | A | Valid | Valid | Compare packed qwords in *xmm2/m128* and *xmm1* for greater than. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (r, w) | ModRM:r/m (r) | NA | NA |

...

## PEXTRB/PEXTRD/PEXTRQ — Extract Byte/Dword/Qword

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 66 0F 3A 14 /r ib | PEXTRB *reg/m8, xmm2, imm8* | A | Valid | Valid | Extract a byte integer value from *xmm2* at the source byte offset specified by *imm8* into *rreg or m8. The upper bits of r32 or r64 are zeroed.* |
| 66 0F 3A 16 /r ib | PEXTRD *r/m32, xmm2, imm8* | A | Valid | Valid | Extract a dword integer value from *xmm2* at the source dword offset specified by *imm8* into *r/m32*. |
| 66 REX.W 0F 3A 16 /r ib | PEXTRQ *r/m64, xmm2, imm8* | A | Valid | N. E. | Extract a qword integer value from *xmm2* at the source qword offset specified by *imm8* into r/m64. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:r/m (w) | ModRM:reg (r) | imm8 | NA |

### Description

Copies a data element (byte, dword, quadword) in the source operand (second operand) specified by the count operand (third operand) to the destination operand (first operand). The source operand is an XMM register. The destination operand can be a general-purpose register or a memory address. The count operand is an 8-bit immediate. When specifying a quadword [dword, byte] element, the [2, 4] least-significant bit(s) of the count operand specify the location.

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). PEXTRQ requires REX.W. If the destination operand is a general-purpose register, the default operand size of PEXTRB/ PEXTRW is 64 bits.

...

## PEXTRW—Extract Word

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F C5 /*r* ib | PEXTRW *reg*, *mm*, *imm8* | A | Valid | Valid | Extract the word specified by *imm8* from *mm* and move it to *reg*, bits 15-0. The upper bits of r32 or r64 is zeroed. |
| 66 0F C5 /*r* ib | PEXTRW *reg*, *xmm*, *imm8* | A | Valid | Valid | Extract the word specified by *imm8* from *xmm* and move it to *reg*, bits 15-0. The upper bits of r32 or r64 is zeroed. |
| 66 0F 3A 15 /r ib | PEXTRW *reg/m16*, *xmm*, *imm8* | B | Valid | Valid | Extract the word specified by *imm8* from *xmm* and copy it to lowest 16 bits of *reg or m16*. Zero-extend the result in the destination, r32 or r64. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (w) | ModRM:reg (r) | imm8 | NA |
| B | ModRM:r/m (w) | ModRM:reg (r) | imm8 | NA |

...

## PHADDW/PHADDD — Packed Horizontal Add

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F 38 01 /r | PHADDW mm1, mm2/m64 | A | Valid | Valid | Add 16-bit signed integers horizontally, pack to MM1. |
| 66 0F 38 01 /r | PHADDW xmm1, xmm2/m128 | A | Valid | Valid | Add 16-bit signed integers horizontally, pack to XMM1. |
| 0F 38 02 /r | PHADDD mm1, mm2/m64 | A | Valid | Valid | Add 32-bit signed integers horizontally, pack to MM1. |
| 66 0F 38 02 /r | PHADDD xmm1, xmm2/m128 | A | Valid | Valid | Add 32-bit signed integers horizontally, pack to XMM1. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (r, w) | ModRM:r/m (r) | NA | NA |

...

## PHADDSW — Packed Horizontal Add and Saturate

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F 38 03 /r | PHADDSW mm1, mm2/m64 | A | Valid | Valid | Add 16-bit signed integers horizontally, pack saturated integers to MM1. |
| 66 0F 38 03 /r | PHADDSW xmm1, xmm2/m128 | A | Valid | Valid | Add 16-bit signed integers horizontally, pack saturated integers to XMM1. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (r, w) | ModRM:r/m (r) | NA | NA |

...

## PHMINPOSUW — Packed Horizontal Word Minimum

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 66 0F 38 41 /r | PHMINPOSUW *xmm1, xmm2/m128* | A | Valid | Valid | Find the minimum unsigned word in *xmm2/m128* and place its value in the low word of *xmm1* and its index in the second-lowest word of *xmm1*. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (w) | ModRM:r/m (r) | NA | NA |

...

## PHSUBW/PHSUBD — Packed Horizontal Subtract

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F 38 05 /r | PHSUBW mm1, mm2/m64 | A | Valid | Valid | Subtract 16-bit signed integers horizontally, pack to MM1. |
| 66 0F 38 05 /r | PHSUBW xmm1, xmm2/m128 | A | Valid | Valid | Subtract 16-bit signed integers horizontally, pack to XMM1. |
| 0F 38 06 /r | PHSUBD mm1, mm2/m64 | A | Valid | Valid | Subtract 32-bit signed integers horizontally, pack to MM1. |
| 66 0F 38 06 /r | PHSUBD xmm1, xmm2/m128 | A | Valid | Valid | Subtract 32-bit signed integers horizontally, pack to XMM1. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (r, w) | ModRM:r/m (r) | NA | NA |

...

## PHSUBSW — Packed Horizontal Subtract and Saturate

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F 38 07 /r | PHSUBSW mm1, mm2/m64 | A | Valid | Valid | Subtract 16-bit signed integer horizontally, pack saturated integers to MM1. |
| 66 0F 38 07 /r | PHSUBSW xmm1, xmm2/m128 | A | Valid | Valid | Subtract 16-bit signed integer horizontally, pack saturated integers to XMM1 |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (r, w) | ModRM:r/m (r) | NA | NA |

...

## PINSRB/PINSRD/PINSRQ — Insert Byte/Dword/Qword

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 66 0F 3A 20 /r ib | PINSRB *xmm1, r32/m8, imm8* | A | Valid | Valid | Insert a byte integer value from *r32/m8* into *xmm1* at the destination element in *xmm1* specified by *imm8.* |
| 66 0F 3A 22 /r ib | PINSRD *xmm1, r/m32, imm8* | A | Valid | Valid | Insert a dword integer value from *r/m32* into the *xmm1* at the destination element specified by *imm8.* |
| 66 REX.W 0F 3A 22 /r ib | PINSRQ *xmm1, r/m64, imm8* | A | N. E. | Valid | Insert a qword integer value from *r/m32* into the *xmm1* at the destination element specified by *imm8.* |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (w) | ModRM:r/m (r) | imm8 | NA |

...

## PINSRW—Insert Word

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F C4 /*r* ib | PINSRW *mm, r32/m16, imm8* | A | Valid | Valid | Insert the low word from *r32* or from *m16* into *mm* at the word position specified by *imm8* |
| 66 0F C4 /*r* ib | PINSRW xmm, *r32/m16*, imm8 | A | Valid | Valid | Move the low word of *r32* or from *m16* into xmm at the word position specified by *imm8*. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (w) | ModRM:r/m (r) | imm8 | NA |

...

## PMADDUBSW — Multiply and Add Packed Signed and Unsigned Bytes

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F 38 04 /r | PMADDUBSW mm1, mm2/m64 | A | Valid | Valid | Multiply signed and unsigned bytes, add horizontal pair of signed words, pack saturated signed-words to MM1. |
| 66 0F 38 04 /r | PMADDUBSW xmm1, xmm2/m128 | A | Valid | Valid | Multiply signed and unsigned bytes, add horizontal pair of signed words, pack saturated signed-words to XMM1. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (r, w) | ModRM:r/m (r) | NA | NA |

...

## PMADDWD—Multiply and Add Packed Integers

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F F5 /*r* | PMADDWD *mm, mm/m64* | A | Valid | Valid | Multiply the packed words in *mm* by the packed words in *mm/m64*, add adjacent doubleword results, and store in *mm*. |
| 66 0F F5 /*r* | PMADDWD *xmm1*, *xmm2/m128* | A | Valid | Valid | Multiply the packed word integers in *xmm1* by the packed word integers in *xmm2/m128*, add adjacent doubleword results, and store in *xmm1*. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (r, w) | ModRM:r/m (r) | NA | NA |

...

## PMAXSB — Maximum of Packed Signed Byte Integers

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 66 0F 38 3C /r | PMAXSB *xmm1, xmm2/m128* | A | Valid | Valid | Compare packed signed byte integers in *xmm1* and *xmm2/m128* and store packed maximum values in *xmm1*. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (r, w) | ModRM:r/m (r) | NA | NA |

...

## PMAXSD — Maximum of Packed Signed Dword Integers

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 66 0F 38 3D /r | PMAXSD *xmm1, xmm2/m128* | A | Valid | Valid | Compare packed signed dword integers in *xmm1* and *xmm2/m128* and store packed maximum values in *xmm1*. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (r, w) | ModRM:r/m (r) | NA | NA |

...

## PMAXSW—Maximum of Packed Signed Word Integers

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F EE */r* | PMAXSW *mm1, mm2/m64* | A | Valid | Valid | Compare signed word integers in *mm2/m64* and *mm1* and return maximum values. |
| 66 0F EE */r* | PMAXSW *xmm1, xmm2/m128* | A | Valid | Valid | Compare signed word integers in *xmm2/m128* and *xmm1* and return maximum values. |

Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (r, w) | ModRM:r/m (r) | NA | NA |

...

## PMAXUB—Maximum of Packed Unsigned Byte Integers

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F DE /*r* | PMAXUB *mm1, mm2/m64* | A | Valid | Valid | Compare unsigned byte integers in *mm2/m64* and *mm1* and returns maximum values. |
| 66 0F DE /*r* | PMAXUB *xmm1, xmm2/m128* | A | Valid | Valid | Compare unsigned byte integers in *xmm2/m128* and *xmm1* and returns maximum values. |

Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (r, w) | ModRM:r/m (r) | NA | NA |

...

## PMAXUD — Maximum of Packed Unsigned Dword Integers

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 66 0F 38 3F /r | PMAXUD *xmm1, xmm2/m128* | A | Valid | Valid | Compare packed unsigned dword integers in *xmm1* and *xmm2/m128* and store packed maximum values in *xmm1*. |

Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (r, w) | ModRM:r/m (r) | NA | NA |

...

## PMAXUW — Maximum of Packed Word Integers

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 66 0F 38 3E /r | PMAXUW *xmm1, xmm2/m128* | A | Valid | Valid | Compare packed unsigned word integers in *xmm1* and *xmm2/m128* and store packed maximum values in *xmm1*. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (r, w) | ModRM:r/m (r) | NA | NA |

...

## PMINSB — Minimum of Packed Signed Byte Integers

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 66 0F 38 38 /r | PMINSB *xmm1, xmm2/m128* | A | Valid | Valid | Compare packed signed byte integers in *xmm1* and *xmm2/m128* and store packed minimum values in *xmm1*. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (r, w) | ModRM:r/m (r) | NA | NA |

...

## PMINSD — Minimum of Packed Dword Integers

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 66 0F 38 39 /r | PMINSD *xmm1, xmm2/m128* | A | Valid | Valid | Compare packed signed dword integers in *xmm1* and *xmm2/m128* and store packed minimum values in *xmm1*. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (r, w) | ModRM:r/m (r) | NA | NA |

...

## PMINSW—Minimum of Packed Signed Word Integers

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F EA /*r* | PMINSW *mm1, mm2/m64* | A | Valid | Valid | Compare signed word integers in *mm2/m64* and *mm1* and return minimum values. |
| 66 0F EA /*r* | PMINSW *xmm1*, *xmm2/m128* | A | Valid | Valid | Compare signed word integers in *xmm2/m128* and *xmm1* and return minimum values. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (r, w) | ModRM:r/m (r) | NA | NA |

...

## PMINUB—Minimum of Packed Unsigned Byte Integers

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F DA /*r* | PMINUB *mm1, mm2/m64* | A | Valid | Valid | Compare unsigned byte integers in *mm2/m64* and *mm1* and returns minimum values. |
| 66 0F DA /*r* | PMINUB *xmm1*, *xmm2/m128* | A | Valid | Valid | Compare unsigned byte integers in *xmm2/m128* and *xmm1* and returns minimum values. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (r, w) | ModRM:r/m (r) | NA | NA |

...

## PMINUD — Minimum of Packed Dword Integers

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 66 0F 38 3B /r | PMINUD *xmm1, xmm2/m128* | A | Valid | Valid | Compare packed unsigned dword integers in *xmm1* and *xmm2/m128* and store packed minimum values in *xmm1*. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (r, w) | ModRM:r/m (r) | NA | NA |

...

## PMINUW — Minimum of Packed Word Integers

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 66 0F 38 3A /r | PMINUW *xmm1, xmm2/m128* | A | Valid | Valid | Compare packed unsigned word integers in *xmm1* and *xmm2/m128* and store packed minimum values in *xmm1*. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (r, w) | ModRM:r/m (r) | NA | NA |

...

## PMOVMSKB—Move Byte Mask

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F D7 */r* | PMOVMSKB *r32*, *mm* | A | Valid | Valid | Move a byte mask of *mm* to *r32*. |
| REX.W + 0F D7 */r* | PMOVMSKB *r64*, *mm* | A | Valid | N.E. | Move a byte mask of mm to the lower 32-bits of r64 and zero-fill the upper 32-bits. |
| 66 0F D7 */r* | PMOVMSKB *reg*, *xmm* | A | Valid | Valid | Move a byte mask of *xmm* to *reg*. The upper bits of r32 or r64 are zeroed |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (w) | ModRM:reg (r) | NA | NA |

...

## PMOVSX — Packed Move with Sign Extend

| Opcode | Instruction | Op/ En | 64-bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 66 0f 38 20 /r | PMOVSXBW *xmm1, xmm2/m64* | A | Valid | Valid | Sign extend 8 packed signed 8-bit integers in the low 8 bytes of *xmm2/m64* to 8 packed signed 16-bit integers in *xmm1.* |
| 66 0f 38 21 /r | PMOVSXBD *xmm1, xmm2/m32* | A | Valid | Valid | Sign extend 4 packed signed 8-bit integers in the low 4 bytes of *xmm2/m32* to 4 packed signed 32-bit integers in *xmm1*. |
| 66 0f 38 22 /r | PMOVSXBQ *xmm1, xmm2/m16* | A | Valid | Valid | Sign extend 2 packed signed 8-bit integers in the low 2 bytes of *xmm2/m16* to 2 packed signed 64-bit integers in *xmm1*. |
| 66 0f 38 23 /r | PMOVSXWD *xmm1, xmm2/m64* | A | Valid | Valid | Sign extend 4 packed signed 16-bit integers in the low 8 bytes of *xmm2/m64* to 4 packed signed 32-bit integers in *xmm1*. |
| 66 0f 38 24 /r | PMOVSXWQ *xmm1, xmm2/m32* | A | Valid | Valid | Sign extend 2 packed signed 16-bit integers in the low 4 bytes of *xmm2/m32* to 2 packed signed 64-bit integers in *xmm1*. |
| 66 0f 38 25 /r | PMOVSXDQ *xmm1, xmm2/m64* | A | Valid | Valid | Sign extend 2 packed signed 32-bit integers in the low 8 bytes of *xmm2/m64* to 2 packed signed 64-bit integers in *xmm1*. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (w) | ModRM:r/m (r) | NA | NA |

...

## PMOVZX — Packed Move with Zero Extend

| Opcode | Instruction | Op/ En | 64-bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 66 0f 38 30 /r | PMOVZXBW *xmm1*, *xmm2/m64* | A | Valid | Valid | 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 | PMOVZXBD *xmm1*, *xmm2/m32* | A | Valid | Valid | 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* | A | Valid | Valid | 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* | A | Valid | Valid | 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* | A | Valid | Valid | 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* | A | Valid | Valid | Zero extend 2 packed 32-bit integers in the low 8 bytes of *xmm2/m64* to 2 packed 64-bit integers in *xmm1*. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (w) | ModRM:r/m (r) | NA | NA |

...

## PMULDQ — Multiply Packed Signed Dword Integers

| **Opcode** | **Instruction** | Op/ En | **64-Bit Mode** | **Compat/ Leg Mode** | **Description** |
|---|---|---|---|---|---|
| 66 0F 38 28 /r | PMULDQ *xmm1*, *xmm2/m128* | A | Valid | Valid | Multiply the packed signed dword integers in *xmm1* and *xmm2/m128* and store the quadword product in *xmm1*. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (r, w) | ModRM:r/m (r) | NA | NA |

...

## PMULHRSW — Packed Multiply High with Round and Scale

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F 38 0B /r | PMULHRSW mm1, mm2/m64 | A | Valid | Valid | Multiply 16-bit signed words, scale and round signed doublewords, pack high 16 bits to MM1. |
| 66 0F 38 0B /r | PMULHRSW xmm1, xmm2/m128 | A | Valid | Valid | Multiply 16-bit signed words, scale and round signed doublewords, pack high 16 bits to XMM1. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (r, w) | ModRM:r/m (r) | NA | NA |

...

## PMULHUW—Multiply Packed Unsigned Integers and Store High Result

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F E4 */r* | PMULHUW *mm1*, *mm2/m64* | A | Valid | Valid | Multiply the packed unsigned word integers in *mm1* register and *mm2/m64*, and store the high 16 bits of the results in *mm1*. |
| 66 0F E4 */r* | PMULHUW *xmm1*, *xmm2/m128* | A | Valid | Valid | Multiply the packed unsigned word integers in *xmm1* and *xmm2/m128*, and store the high 16 bits of the results in *xmm1*. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (r, w) | ModRM:r/m (r) | NA | NA |

...

## PMULHW—Multiply Packed Signed Integers and Store High Result

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F E5 /*r* | PMULHW *mm, mm/m64* | A | Valid | Valid | Multiply the packed signed word integers in *mm1* register and *mm2/m64*, and store the high 16 bits of the results in *mm1*. |
| 66 0F E5 /*r* | PMULHW *xmm1, xmm2/m128* | A | Valid | Valid | Multiply the packed signed word integers in *xmm1* and *xmm2/m128*, and store the high 16 bits of the results in *xmm1*. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (r, w) | ModRM:r/m (r) | NA | NA |

...

## PMULLD — Multiply Packed Signed Dword Integers and Store Low Result

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 66 0F 38 40 /r | PMULLD *xmm1, xmm2/m128* | A | Valid | Valid | Multiply the packed dword signed integers in *xmm1* and *xmm2/m128* and store the low 32 bits of each product in *xmm1*. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (r, w) | ModRM:r/m (r) | NA | NA |

...

## PMULLW—Multiply Packed Signed Integers and Store Low Result

| **Opcode** | **Instruction** | **Op/ En** | **64-Bit Mode** | **Compat/ Leg Mode** | **Description** |
|---|---|---|---|---|---|
| 0F D5 */r* | PMULLW *mm, mm/m64* | A | Valid | Valid | Multiply the packed signed word integers in *mm1* register and *mm2/m64*, and store the low 16 bits of the results in *mm1*. |
| 66 0F D5 */r* | PMULLW *xmm1*, *xmm2/m128* | A | Valid | Valid | Multiply the packed signed word integers in *xmm1* and *xmm2/m128*, and store the low 16 bits of the results in *xmm1*. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (r, w) | ModRM:r/m (r) | NA | NA |

...

## PMULUDQ—Multiply Packed Unsigned Doubleword Integers

| **Opcode** | **Instruction** | **Op/ En** | **64-Bit Mode** | **Compat/ Leg Mode** | **Description** |
|---|---|---|---|---|---|
| 0F F4 */r* | PMULUDQ *mm1*, *mm2/m64* | A | Valid | Valid | 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* | A | Valid | Valid | Multiply packed unsigned doubleword integers in *xmm1* by packed unsigned doubleword integers in *xmm2/m128*, and store the quadword results in *xmm1*. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (r, w) | ModRM:r/m (r) | NA | NA |

...

## POP—Pop a Value from the Stack

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 8F /0 | POP r/*m16* | A | Valid | Valid | Pop top of stack into *m16*; increment stack pointer. |
| 8F /0 | POP r/*m32* | A | N.E. | Valid | Pop top of stack into *m32*; increment stack pointer. |
| 8F /0 | POP r/*m64* | A | Valid | N.E. | Pop top of stack into *m64*; increment stack pointer. Cannot encode 32-bit operand size. |
| 58+ *rw* | POP *r16* | B | Valid | Valid | Pop top of stack into *r16*; increment stack pointer. |
| 58+ *rd* | POP *r32* | B | N.E. | Valid | Pop top of stack into *r32*; increment stack pointer. |
| 58+ *rd* | POP *r64* | B | Valid | N.E. | Pop top of stack into *r64*; increment stack pointer. Cannot encode 32-bit operand size. |
| 1F | POP DS | C | Invalid | Valid | Pop top of stack into DS; increment stack pointer. |
| 07 | POP ES | C | Invalid | Valid | Pop top of stack into ES; increment stack pointer. |
| 17 | POP SS | C | Invalid | Valid | Pop top of stack into SS; increment stack pointer. |
| 0F A1 | POP FS | C | Valid | Valid | Pop top of stack into FS; increment stack pointer by 16 bits. |
| 0F A1 | POP FS | C | N.E. | Valid | Pop top of stack into FS; increment stack pointer by 32 bits. |
| 0F A1 | POP FS | C | Valid | N.E. | Pop top of stack into FS; increment stack pointer by 64 bits. |
| 0F A9 | POP GS | C | Valid | Valid | Pop top of stack into GS; increment stack pointer by 16 bits. |
| 0F A9 | POP GS | C | N.E. | Valid | Pop top of stack into GS; increment stack pointer by 32 bits. |
| 0F A9 | POP GS | C | Valid | N.E. | Pop top of stack into GS; increment stack pointer by 64 bits. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:r/m (w) | NA | NA | NA |
| B | reg (w) | NA | NA | NA |
| C | NA | NA | NA | NA |

...

## POPA/POPAD—Pop All General-Purpose Registers

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 61 | POPA | A | Invalid | Valid | Pop DI, SI, BP, BX, DX, CX, and AX. |
| 61 | POPAD | A | Invalid | Valid | Pop EDI, ESI, EBP, EBX, EDX, ECX, and EAX. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | NA | NA | NA | NA |

...

## POPCNT — Return the Count of Number of Bits Set to 1

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| F3 0F B8 /r | POPCNT *r16, r/m16* | A | Valid | Valid | POPCNT on *r/m16* |
| F3 0F B8 /r | POPCNT *r32, r/m32* | A | Valid | Valid | POPCNT on *r/m32* |
| F3 REX.W 0F B8 /r | POPCNT *r64, r/m64* | A | Valid | N.E. | POPCNT on *r/m64* |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (w) | ModRM:r/m (r) | NA | NA |

...

## POPF/POPFD/POPFQ—Pop Stack into EFLAGS Register

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 9D | POPF | A | Valid | Valid | Pop top of stack into lower 16 bits of EFLAGS. |
| 9D | POPFD | A | N.E. | Valid | Pop top of stack into EFLAGS. |
| REX.W + 9D | POPFQ | A | Valid | N.E. | Pop top of stack and zero-extend into RFLAGS. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | NA | NA | NA | NA |

...

## POR—Bitwise Logical OR

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F EB /*r* | POR *mm, mm/m64* | A | Valid | Valid | Bitwise OR of *mm/m64* and *mm*. |
| 66 0F EB /*r* | POR *xmm1, xmm2/m128* | A | Valid | Valid | Bitwise OR of *xmm2/m128* and *xmm1*. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (r, w) | ModRM:r/m (r) | NA | NA |

...

## PREFETCH*h*—Prefetch Data Into Caches

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F 18 /1 | PREFETCHT0 *m8* | A | Valid | Valid | Move data from *m8* closer to the processor using T0 hint. |
| 0F 18 /2 | PREFETCHT1 *m8* | A | Valid | Valid | Move data from *m8* closer to the processor using T1 hint. |
| 0F 18 /3 | PREFETCHT2 *m8* | A | Valid | Valid | Move data from *m8* closer to the processor using T2 hint. |
| 0F 18 /0 | PREFETCHNTA *m8* | A | Valid | Valid | Move data from *m8* closer to the processor using NTA hint. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:r/m (r) | NA | NA | NA |

...

## PSADBW—Compute Sum of Absolute Differences

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F F6 /*r* | PSADBW *mm1, mm2/m64* | A | Valid | Valid | 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* | A | Valid | Valid | 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. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (r, w) | ModRM:r/m (r) | NA | NA |

...

## PSHUFB — Packed Shuffle Bytes

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F 38 00 /r | PSHUFB mm1, mm2/m64 | A | Valid | Valid | Shuffle bytes in mm1 according to contents of mm2/m64. |
| 66 0F 38 00 /r | PSHUFB xmm1, xmm2/m128 | A | Valid | Valid | Shuffle bytes in xmm1 according to contents of xmm2/m128. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (r, w) | ModRM:r/m (r) | NA | NA |

...

## PSHUFD—Shuffle Packed Doublewords

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 66 0F 70 /*r* ib | PSHUFD *xmm1*, *xmm2/m128*, *imm8* | A | Valid | Valid | Shuffle the doublewords in *xmm2/m128* based on the encoding in *imm8* and store the result in *xmm1*. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (w) | ModRM:r/m (r) | imm8 | NA |

...

## PSHUFHW—Shuffle Packed High Words

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| F3 0F 70 /*r* ib | PSHUFHW *xmm1*, *xmm2/ m128*, *imm8* | A | Valid | Valid | Shuffle the high words in *xmm2/m128* based on the encoding in *imm8* and store the result in *xmm1*. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (w) | ModRM:r/m (r) | imm8 | NA |

...

## PSHUFLW—Shuffle Packed Low Words

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| F2 0F 70 /*r* ib | PSHUFLW *xmm1*, *xmm2/m128*, *imm8* | A | Valid | Valid | Shuffle the low words in *xmm2/m128* based on the encoding in *imm8* and store the result in *xmm1*. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (w) | ModRM:r/m (r) | imm8 | NA |

...

## PSHUFW—Shuffle Packed Words

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F 70 /*r* ib | PSHUFW *mm1*, *mm2/m64*, *imm8* | A | Valid | Valid | Shuffle the words in *mm2/m64* based on the encoding in *imm8* and store the result in *mm1*. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (w) | ModRM:r/m (r) | imm8 | NA |

...

## PSIGNB/PSIGNW/PSIGND — Packed SIGN

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F 38 08 /r | PSIGNB mm1, mm2/m64 | A | Valid | Valid | Negate/zero/preserve packed byte integers in mm1 depending on the corresponding sign in mm2/m64 |
| 66 0F 38 08 /r | PSIGNB xmm1, xmm2/m128 | A | Valid | Valid | Negate/zero/preserve packed byte integers in xmm1 depending on the corresponding sign in xmm2/m128. |
| 0F 38 09 /r | PSIGNW mm1, mm2/m64 | A | Valid | Valid | Negate/zero/preserve packed word integers in mm1 depending on the corresponding sign in mm2/m128. |
| 66 0F 38 09 /r | PSIGNW xmm1, xmm2/m128 | A | Valid | Valid | Negate/zero/preserve packed word integers in xmm1 depending on the corresponding sign in xmm2/m128. |
| 0F 38 0A /r | PSIGND mm1, mm2/m64 | A | Valid | Valid | Negate/zero/preserve packed doubleword integers in mm1 depending on the corresponding sign in mm2/m128. |
| 66 0F 38 0A /r | PSIGND xmm1, xmm2/m128 | A | Valid | Valid | Negate/zero/preserve packed doubleword integers in xmm1 depending on the corresponding sign in xmm2/m128. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (r, w) | ModRM:r/m (r) | NA | NA |

...

## PSLLDQ—Shift Double Quadword Left Logical

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 66 0F 73 /7 ib | PSLLDQ *xmm1*, *imm8* | A | Valid | Valid | Shift *xmm1* left by *imm8* bytes while shifting in 0s. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:r/m (r, w) | imm8 | NA | NA |

...

## PSLLW/PSLLD/PSLLQ—Shift Packed Data Left Logical

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F F1 /*r* | PSLLW *mm*, *mm/m64* | A | Valid | Valid | Shift words in *mm* left *mm/m64* while shifting in 0s. |
| 66 0F F1 /*r* | PSLLW *xmm1*, *xmm2/m128* | A | Valid | Valid | Shift words in *xmm1* left by *xmm2/m128* while shifting in 0s. |
| 0F 71 /6 ib | PSLLW *xmm1*, *imm8* | B | Valid | Valid | Shift words in *mm* left by *imm8* while shifting in 0s. |
| 66 0F 71 /6 ib | PSLLW *xmm1*, *imm8* | B | Valid | Valid | Shift words in *xmm1* left by *imm8* while shifting in 0s. |
| 0F F2 /*r* | PSLLD *mm*, *mm/m64* | A | Valid | Valid | Shift doublewords in *mm* left by *mm/m64* while shifting in 0s. |
| 66 0F F2 /*r* | PSLLD *xmm1*, *xmm2/m128* | A | Valid | Valid | Shift doublewords in *xmm1* left by *xmm2/m128* while shifting in 0s. |
| 0F 72 /6 ib | PSLLD *mm, imm8* | B | Valid | Valid | Shift doublewords in *mm* left by *imm8* while shifting in 0s. |
| 66 0F 72 /6 ib | PSLLD *xmm1*, *imm8* | B | Valid | Valid | Shift doublewords in *xmm1* left by *imm8* while shifting in 0s. |
| 0F F3 /*r* | PSLLQ *mm*, *mm/m64* | A | Valid | Valid | Shift quadword in *mm* left by *mm/m64* while shifting in 0s. |
| 66 0F F3 /*r* | PSLLQ *xmm1*, *xmm2/m128* | A | Valid | Valid | Shift quadwords in *xmm1* left by *xmm2/m128* while shifting in 0s. |
| 0F 73 /6 ib | PSLLQ *mm, imm8* | B | Valid | Valid | Shift quadword in *mm* left by *imm8* while shifting in 0s. |

| **Opcode** | **Instruction** | **Op/ En** | **64-Bit Mode** | **Compat/ Leg Mode** | **Description** |
|---|---|---|---|---|---|
| 66 0F 73 /6 ib | PSLLQ *xmm1, imm8* | B | Valid | Valid | Shift quadwords in *xmm1* left by *imm8* while shifting in 0s. |

**Instruction Operand Encoding**

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (r, w) | ModRM:r/m (r) | NA | NA |
| B | ModRM:r/m (r, w) | imm8 | NA | NA |

...

## PSRAW/PSRAD—Shift Packed Data Right Arithmetic

| **Opcode** | **Instruction** | **Op/ En** | **64-Bit Mode** | **Compat/ Leg Mode** | **Description** |
|---|---|---|---|---|---|
| 0F E1 /*r* | PSRAW *mm, mm/m64* | A | Valid | Valid | Shift words in *mm* right by *mm/m64* while shifting in sign bits. |
| 66 0F E1 /*r* | PSRAW *xmm1, xmm2/m128* | A | Valid | Valid | Shift words in *xmm1* right by *xmm2/m128* while shifting in sign bits. |
| 0F 71 /4 ib | PSRAW *mm, imm8* | B | Valid | Valid | Shift words in *mm* right by *imm8* while shifting in sign bits |
| 66 0F 71 /4 ib | PSRAW *xmm1,* imm8 | B | Valid | Valid | Shift words in *xmm1* right by imm8 while shifting in sign bits |
| 0F E2 /*r* | PSRAD *mm, mm/m64* | A | Valid | Valid | Shift doublewords in *mm* right by *mm/m64* while shifting in sign bits. |
| 66 0F E2 /*r* | PSRAD *xmm1, xmm2/m128* | A | Valid | Valid | Shift doubleword in *xmm1* right by *xmm2 /m128* while shifting in sign bits. |
| 0F 72 /4 ib | PSRAD *mm, imm8* | B | Valid | Valid | Shift doublewords in *mm* right by *imm8* while shifting in sign bits. |
| 66 0F 72 /4 ib | PSRAD *xmm1,* imm8 | B | Valid | Valid | Shift doublewords in *xmm1* right by *imm8* while shifting in sign bits. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (r, w) | ModRM:r/m (r) | NA | NA |
| B | ModRM:r/m (r, w) | imm8 | NA | NA |

...

## PSRLDQ—Shift Double Quadword Right Logical

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 66 0F 73 /3 ib | PSRLDQ *xmm1, imm8* | A | Valid | Valid | Shift *xmm1* right by *imm8* while shifting in 0s. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:r/m (r, w) | imm8 | NA | NA |

...

## PSRLW/PSRLD/PSRLQ—Shift Packed Data Right Logical

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F D1 /*r* | PSRLW *mm, mm/m64* | A | Valid | Valid | Shift words in *mm* right by amount specified in *mm/m64* while shifting in 0s. |
| 66 0F D1 /*r* | PSRLW *xmm1, xmm2/m128* | A | Valid | Valid | Shift words in *xmm1* right by amount specified in *xmm2/m128* while shifting in 0s. |
| 0F 71 /2 ib | PSRLW *mm, imm8* | B | Valid | Valid | Shift words in *mm* right by *imm8* while shifting in 0s. |
| 66 0F 71 /2 ib | PSRLW *xmm1, imm8* | B | Valid | Valid | Shift words in *xmm1* right by *imm8* while shifting in 0s. |
| 0F D2 /*r* | PSRLD *mm, mm/m64* | A | Valid | Valid | Shift doublewords in *mm* right by amount specified in *mm/m64* while shifting in 0s. |
| 66 0F D2 /*r* | PSRLD *xmm1, xmm2/m128* | A | Valid | Valid | Shift doublewords in *xmm1* right by amount specified in *xmm2 /m128* while shifting in 0s. |
| 0F 72 /2 ib | PSRLD *mm, imm8* | B | Valid | Valid | Shift doublewords in *mm* right by *imm8* while shifting in 0s. |

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 66 0F 72 /2 ib | PSRLD *xmm1*, imm8 | B | Valid | Valid | Shift doublewords in *xmm1* right by *imm8* while shifting in 0s. |
| 0F D3 /*r* | PSRLQ *mm*, *mm/m64* | A | Valid | Valid | Shift *mm* right by amount specified in *mm/m64* while shifting in 0s. |
| 66 0F D3 /*r* | PSRLQ *xmm1*, *xmm2/m128* | A | Valid | Valid | Shift quadwords in *xmm1* right by amount specified in *xmm2/m128* while shifting in 0s. |
| 0F 73 /2 ib | PSRLQ *mm*, *imm8* | B | Valid | Valid | Shift *mm* right by *imm8* while shifting in 0s. |
| 66 0F 73 /2 ib | PSRLQ *xmm1*, *imm8* | B | Valid | Valid | Shift quadwords in *xmm1* right by *imm8* while shifting in 0s. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (r, w) | ModRM:r/m (r) | NA | NA |
| B | ModRM:r/m (r, w) | imm8 | NA | NA |

...

## PSUBB/PSUBW/PSUBD—Subtract Packed Integers

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F F8 /*r* | PSUBB *mm*, *mm/m64* | A | Valid | Valid | Subtract packed byte integers in *mm/m64* from packed byte integers in *mm*. |
| 66 0F F8 /*r* | PSUBB *xmm1*, *xmm2/m128* | A | Valid | Valid | Subtract packed byte integers in *xmm2/m128* from packed byte integers in *xmm1*. |
| 0F F9 /*r* | PSUBW *mm*, *mm/m64* | A | Valid | Valid | Subtract packed word integers in *mm/m64* from packed word integers in *mm*. |
| 66 0F F9 /*r* | PSUBW *xmm1*, *xmm2/m128* | A | Valid | Valid | Subtract packed word integers in *xmm2/m128* from packed word integers in *xmm1*. |
| 0F FA /*r* | PSUBD *mm*, *mm/m64* | A | Valid | Valid | Subtract packed doubleword integers in *mm/m64* from packed doubleword integers in *mm*. |

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 66 0F FA /*r* | PSUBD *xmm1*, *xmm2/m128* | A | Valid | Valid | Subtract packed doubleword integers in *xmm2/mem128* from packed doubleword integers in *xmm1*. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (r, w) | ModRM:r/m (r) | NA | NA |

...

## PSUBQ—Subtract Packed Quadword Integers

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F FB /*r* | PSUBQ *mm1*, *mm2/m64* | A | Valid | Valid | Subtract quadword integer in *mm1* from *mm2 /m64*. |
| 66 0F FB /*r* | PSUBQ *xmm1*, *xmm2/m128* | A | Valid | Valid | Subtract packed quadword integers in *xmm1* from *xmm2 /m128*. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (r, w) | ModRM:r/m (r) | NA | NA |

...

## PSUBSB/PSUBSW—Subtract Packed Signed Integers with Signed Saturation

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F E8 /*r* | PSUBSB *mm*, *mm/m64* | A | Valid | Valid | Subtract signed packed bytes in *mm/m64* from signed packed bytes in *mm* and saturate results. |
| 66 0F E8 /*r* | PSUBSB *xmm1*, *xmm2/m128* | A | Valid | Valid | Subtract packed signed byte integers in *xmm2/m128* from packed signed byte integers in *xmm1* and saturate results. |
| 0F E9 /*r* | PSUBSW *mm*, *mm/m64* | A | Valid | Valid | Subtract signed packed words in *mm/m64* from signed packed words in *mm* and saturate results. |

| **Opcode** | **Instruction** | Op/ En | **64-Bit Mode** | **Compat/ Leg Mode** | **Description** |
|---|---|---|---|---|---|
| 66 0F E9 /*r* | PSUBSW *xmm1*, *xmm2/m128* | A | Valid | Valid | Subtract packed signed word integers in *xmm2/m128* from packed signed word integers in *xmm1* and saturate results. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (r, w) | ModRM:r/m (r) | NA | NA |

...

## PSUBUSB/PSUBUSW—Subtract Packed Unsigned Integers with Unsigned Saturation

| **Opcode** | **Instruction** | Op/ En | **64-Bit Mode** | **Compat/ Leg Mode** | **Description** |
|---|---|---|---|---|---|
| 0F D8 /*r* | PSUBUSB *mm*, *mm/m64* | A | Valid | Valid | Subtract unsigned packed bytes in *mm/m64* from unsigned packed bytes in *mm* and saturate result. |
| 66 0F D8 /*r* | PSUBUSB *xmm1*, *xmm2/m128* | A | Valid | Valid | Subtract packed unsigned byte integers in *xmm2/m128* from packed unsigned byte integers in xmm1 and saturate result. |
| 0F D9 /*r* | PSUBUSW *mm*, *mm/m64* | A | Valid | Valid | Subtract unsigned packed words in *mm/m64* from unsigned packed words in *mm* and saturate result. |
| 66 0F D9 /*r* | PSUBUSW *xmm1*, *xmm2/m128* | A | Valid | Valid | Subtract packed unsigned word integers in *xmm2/m128* from packed unsigned word integers in xmm1 and saturate result. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (r, w) | ModRM:r/m (r) | NA | NA |

...

## PTEST- Logical Compare

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 66 0F 38 17 /r | PTEST *xmm1, xmm2/m128* | A | Valid | Valid | Set ZF if *xmm2/m128* AND *xmm1* result is all 0s. Set CF if *xmm2/m128* AND NOT *xmm1* result is all 0s. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (r) | ModRM:r/m (r) | NA | NA |

...

## PUNPCKHBW/PUNPCKHWD/PUNPCKHDQ/PUNPCKHQDQ— Unpack High Data

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F 68 /*r* | PUNPCKHBW *mm, mm/m64* | A | Valid | Valid | Unpack and interleave high-order bytes from *mm* and *mm/m64* into *mm.* |
| 66 0F 68 /*r* | PUNPCKHBW *xmm1*, *xmm2/m128* | A | Valid | Valid | Unpack and interleave high-order bytes from *xmm1* and *xmm2/m128* into *xmm1.* |
| 0F 69 /*r* | PUNPCKHWD *mm, mm/m64* | A | Valid | Valid | Unpack and interleave high-order words from *mm* and *mm/m64* into *mm.* |
| 66 0F 69 /*r* | PUNPCKHWD *xmm1*, *xmm2/m128* | A | Valid | Valid | Unpack and interleave high-order words from *xmm1* and *xmm2/m128* into *xmm1.* |
| 0F 6A /*r* | PUNPCKHDQ *mm, mm/m64* | A | Valid | Valid | Unpack and interleave high-order doublewords from *mm* and *mm/m64* into *mm.* |
| 66 0F 6A /*r* | PUNPCKHDQ *xmm1*, *xmm2/m128* | A | Valid | Valid | Unpack and interleave high-order doublewords from *xmm1* and *xmm2/m128* into *xmm1.* |
| 66 0F 6D /*r* | PUNPCKHQDQ *xmm1*, *xmm2/m128* | A | Valid | Valid | Unpack and interleave high-order quadwords from *xmm1* and *xmm2/m128* into *xmm1.* |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (r, w) | ModRM:r/m (r) | NA | NA |

...

## PUNPCKLBW/PUNPCKLWD/PUNPCKLDQ/PUNPCKLQDQ— Unpack Low Data

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F 60 /*r* | PUNPCKLBW *mm, mm/m32* | A | Valid | Valid | Interleave low-order bytes from *mm* and *mm/m32* into *mm*. |
| 66 0F 60 /*r* | PUNPCKLBW *xmm1*, *xmm2/m128* | A | Valid | Valid | Interleave low-order bytes from *xmm1* and *xmm2/m128* into *xmm1*. |
| 0F 61 /*r* | PUNPCKLWD *mm, mm/m32* | A | Valid | Valid | Interleave low-order words from *mm* and *mm/m32* into *mm*. |
| 66 0F 61 /*r* | PUNPCKLWD *xmm1*, *xmm2/m128* | A | Valid | Valid | Interleave low-order words from *xmm1* and *xmm2/m128* into *xmm1*. |
| 0F 62 /*r* | PUNPCKLDQ *mm, mm/m32* | A | Valid | Valid | Interleave low-order doublewords from *mm* and *mm/m32* into *mm*. |
| 66 0F 62 /*r* | PUNPCKLDQ *xmm1*, *xmm2/m128* | A | Valid | Valid | Interleave low-order doublewords from *xmm1* and *xmm2/m128* into xmm1. |
| 66 0F 6C /*r* | PUNPCKLQDQ *xmm1*, *xmm2/m128* | A | Valid | Valid | Interleave low-order quadword from *xmm1* and *xmm2/m128* into *xmm1* register. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (r, w) | ModRM:r/m (r) | NA | NA |

...

## PUSH—Push Word, Doubleword or Quadword Onto the Stack

| Opcode* | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| FF /6 | PUSH *r/m16* | A | Valid | Valid | Push *r/m16.* |
| FF /6 | PUSH *r/m32* | A | N.E. | Valid | Push *r/m32.* |
| FF /6 | PUSH *r/m64* | A | Valid | N.E. | Push *r/m64.* Default operand size 64-bits. |
| 50+*rw* | PUSH *r16* | B | Valid | Valid | Push *r16.* |
| 50+*rd* | PUSH *r32* | B | N.E. | Valid | Push *r32.* |
| 50+*rd* | PUSH *r64* | B | Valid | N.E. | Push *r64.* Default operand size 64-bits. |
| 6A | PUSH *imm8* | C | Valid | Valid | Push sign-extended *imm8. Stack pointer is incremented by the size of stack pointer.* |
| 68 | PUSH *imm16* | C | Valid | Valid | Push sign-extended *imm16. Stack pointer is incremented by the size of stack pointer.* |
| 68 | PUSH *imm32* | C | Valid | Valid | Push sign-extended *imm32. Stack pointer is incremented by the size of stack pointer.* |
| 0E | PUSH CS | D | Invalid | Valid | Push CS. |
| 16 | PUSH SS | D | Invalid | Valid | Push SS. |
| 1E | PUSH DS | D | Invalid | Valid | Push DS. |
| 06 | PUSH ES | D | Invalid | Valid | Push ES. |
| 0F A0 | PUSH FS | D | Valid | Valid | Push FS and decrement stack pointer by 16 bits. |
| 0F A0 | PUSH FS | D | N.E. | Valid | Push FS and decrement stack pointer by 32 bits. |
| 0F A0 | PUSH FS | D | Valid | N.E. | Push FS. Default operand size 64-bits. (66H override causes 16-bit operation). |
| 0F A8 | PUSH GS | D | Valid | Valid | Push GS and decrement stack pointer by 16 bits. |
| 0F A8 | PUSH GS | D | N.E. | Valid | Push GS and decrement stack pointer by 32 bits. |
| 0F A8 | PUSH GS | D | Valid | N.E. | Push GS, default operand size 64-bits. (66H override causes 16-bit operation). |

**NOTES:**

* See IA-32 Architecture Compatibility section below.

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:r/m (r) | NA | NA | NA |
| B | reg (r) | NA | NA | NA |
| C | imm8/16/32 | NA | NA | NA |
| D | NA | NA | NA | NA |

...

## PUSHA/PUSHAD—Push All General-Purpose Registers

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 60 | PUSHA | A | Invalid | Valid | Push AX, CX, DX, BX, original SP, BP, SI, and DI. |
| 60 | PUSHAD | A | Invalid | Valid | Push EAX, ECX, EDX, EBX, original ESP, EBP, ESI, and EDI. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | NA | NA | NA | NA |

...

## PUSHF/PUSHFD—Push EFLAGS Register onto the Stack

| Opcode* | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 9C | PUSHF | A | Valid | Valid | Push lower 16 bits of EFLAGS. |
| 9C | PUSHFD | A | N.E. | Valid | Push EFLAGS. |
| 9C | PUSHFQ | A | Valid | N.E. | Push RFLAGS. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | NA | NA | NA | NA |

...

## PXOR—Logical Exclusive OR

| Opcode* | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F EF /*r* | PXOR *mm, mm/m64* | A | Valid | Valid | Bitwise XOR of *mm/m64* and *mm*. |
| 66 0F EF /*r* | PXOR *xmm1, xmm2/m128* | A | Valid | Valid | Bitwise XOR of *xmm2/m128* and *xmm1*. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (r, w) | ModRM:r/m (r) | NA | NA |

...

## RCL/RCR/ROL/ROR-—Rotate

| Opcode** | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| D0 /2 | RCL *r/m8*, 1 | A | Valid | Valid | Rotate 9 bits (CF, *r/m8*) left once. |
| REX + D0 /2 | RCL *r/m8**, 1 | A | Valid | N.E. | Rotate 9 bits (CF, *r/m8*) left once. |
| D2 /2 | RCL *r/m8*, CL | B | Valid | Valid | Rotate 9 bits (CF, *r/m8*) left CL times. |
| REX + D2 /2 | RCL *r/m8**, CL | B | Valid | N.E. | Rotate 9 bits (CF, *r/m8*) left CL times. |
| C0 /2 *ib* | RCL *r/m8, imm8* | C | Valid | Valid | Rotate 9 bits (CF, *r/m8*) left *imm8* times. |
| REX + C0 /2 *ib* | RCL *r/m8*, imm8* | C | Valid | N.E. | Rotate 9 bits (CF, *r/m8*) left *imm8* times. |
| D1 /2 | RCL *r/m16*, 1 | A | Valid | Valid | Rotate 17 bits (CF, *r/m16*) left once. |
| D3 /2 | RCL *r/m16*, CL | B | Valid | Valid | Rotate 17 bits (CF, *r/m16*) left CL times. |
| C1 /2 *ib* | RCL *r/m16, imm8* | C | Valid | Valid | Rotate 17 bits (CF, *r/m16*) left *imm8* times. |
| D1 /2 | RCL *r/m32*, 1 | A | Valid | Valid | Rotate 33 bits (CF, *r/m32*) left once. |
| REX.W + D1 /2 | RCL *r/m64*, 1 | A | Valid | N.E. | Rotate 65 bits (CF, *r/m64*) left once. Uses a 6 bit count. |
| D3 /2 | RCL *r/m32*, CL | B | Valid | Valid | Rotate 33 bits (CF, *r/m32*) left CL times. |
| REX.W + D3 /2 | RCL *r/m64*, CL | B | Valid | N.E. | Rotate 65 bits (CF, *r/m64*) left CL times. Uses a 6 bit count. |

| Opcode** | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| C1 /2 *ib* | RCL *r/m32, imm8* | C | Valid | Valid | Rotate 33 bits (CF, *r/m32*) left *imm8* times. |
| REX.W + C1 /2 *ib* | RCL *r/m64, imm8* | C | Valid | N.E. | Rotate 65 bits (CF, *r/m64*) left *imm8* times. Uses a 6 bit count. |
| D0 /3 | RCR *r/m8*, 1 | A | Valid | Valid | Rotate 9 bits (CF, *r/m8*) right once. |
| REX + D0 /3 | RCR *r/m8**, 1 | A | Valid | N.E. | Rotate 9 bits (CF, *r/m8*) right once. |
| D2 /3 | RCR *r/m8*, CL | B | Valid | Valid | Rotate 9 bits (CF, *r/m8*) right CL times. |
| REX + D2 /3 | RCR *r/m8**, CL | B | Valid | N.E. | Rotate 9 bits (CF, *r/m8*) right CL times. |
| C0 /3 *ib* | RCR *r/m8, imm8* | C | Valid | Valid | Rotate 9 bits (CF, *r/m8*) right *imm8* times. |
| REX + C0 /3 *ib* | RCR *r/m8*, imm8* | C | Valid | N.E. | Rotate 9 bits (CF, *r/m8*) right *imm8* times. |
| D1 /3 | RCR *r/m16*, 1 | A | Valid | Valid | Rotate 17 bits (CF, *r/m16*) right once. |
| D3 /3 | RCR *r/m16*, CL | B | Valid | Valid | Rotate 17 bits (CF, *r/m16*) right CL times. |
| C1 /3 *ib* | RCR *r/m16, imm8* | C | Valid | Valid | Rotate 17 bits (CF, *r/m16*) right *imm8* times. |
| D1 /3 | RCR *r/m32*, 1 | A | Valid | Valid | Rotate 33 bits (CF, *r/m32*) right once. Uses a 6 bit count. |
| REX.W + D1 /3 | RCR *r/m64*, 1 | A | Valid | N.E. | Rotate 65 bits (CF, *r/m64*) right once. Uses a 6 bit count. |
| D3 /3 | RCR *r/m32*, CL | B | Valid | Valid | Rotate 33 bits (CF, *r/m32*) right CL times. |
| REX.W + D3 /3 | RCR *r/m64*, CL | B | Valid | N.E. | Rotate 65 bits (CF, *r/m64*) right CL times. Uses a 6 bit count. |
| C1 /3 *ib* | RCR *r/m32, imm8* | C | Valid | Valid | Rotate 33 bits (CF, *r/m32*) right *imm8* times. |
| REX.W + C1 /3 *ib* | RCR *r/m64, imm8* | C | Valid | N.E. | Rotate 65 bits (CF, *r/m64*) right *imm8* times. Uses a 6 bit count. |
| D0 /0 | ROL *r/m8*, 1 | A | Valid | Valid | Rotate 8 bits *r/m8* left once. |
| REX + D0 /0 | ROL *r/m8**, 1 | A | Valid | N.E. | Rotate 8 bits *r/m8* left once |
| D2 /0 | ROL *r/m8*, CL | B | Valid | Valid | Rotate 8 bits *r/m8* left CL times. |
| REX + D2 /0 | ROL *r/m8**, CL | B | Valid | N.E. | Rotate 8 bits *r/m8* left CL times. |

| Opcode** | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| C0 /0 *ib* | ROL *r/m8, imm8* | C | Valid | Valid | Rotate 8 bits *r/m8* left *imm8* times. |
| REX + C0 /0 *ib* | ROL *r/m8**, *imm8* | C | Valid | N.E. | Rotate 8 bits *r/m8* left *imm8* times. |
| D1 /0 | ROL *r/m16*, 1 | A | Valid | Valid | Rotate 16 bits *r/m16* left once. |
| D3 /0 | ROL *r/m16*, CL | B | Valid | Valid | Rotate 16 bits *r/m16* left CL times. |
| C1 /0 *ib* | ROL *r/m16, imm8* | C | Valid | Valid | Rotate 16 bits *r/m16* left *imm8* times. |
| D1 /0 | ROL *r/m32*, 1 | A | Valid | Valid | Rotate 32 bits *r/m32* left once. |
| REX.W + D1 /0 | ROL *r/m64*, 1 | A | Valid | N.E. | Rotate 64 bits *r/m64* left once. Uses a 6 bit count. |
| D3 /0 | ROL *r/m32*, CL | B | Valid | Valid | Rotate 32 bits *r/m32* left CL times. |
| REX.W + D3 /0 | ROL *r/m64*, CL | B | Valid | N.E. | Rotate 64 bits *r/m64* left CL times. Uses a 6 bit count. |
| C1 /0 *ib* | ROL *r/m32, imm8* | C | Valid | Valid | Rotate 32 bits *r/m32* left *imm8* times. |
| C1 /0 *ib* | ROL *r/m64, imm8* | C | Valid | N.E. | Rotate 64 bits *r/m64* left *imm8* times. Uses a 6 bit count. |
| D0 /1 | ROR *r/m8*, 1 | A | Valid | Valid | Rotate 8 bits *r/m8* right once. |
| REX + D0 /1 | ROR *r/m8**, 1 | A | Valid | N.E. | Rotate 8 bits *r/m8* right once. |
| D2 /1 | ROR *r/m8*, CL | B | Valid | Valid | Rotate 8 bits *r/m8* right CL times. |
| REX + D2 /1 | ROR *r/m8**, CL | B | Valid | N.E. | Rotate 8 bits *r/m8* right CL times. |
| C0 /1 *ib* | ROR *r/m8, imm8* | C | Valid | Valid | Rotate 8 bits *r/m16* right *imm8* times. |
| REX + C0 /1 *ib* | ROR *r/m8**, *imm8* | C | Valid | N.E. | Rotate 8 bits *r/m16* right *imm8* times. |
| D1 /1 | ROR *r/m16*, 1 | A | Valid | Valid | Rotate 16 bits *r/m16* right once. |
| D3 /1 | ROR *r/m16*, CL | B | Valid | Valid | Rotate 16 bits *r/m16* right CL times. |
| C1 /1 *ib* | ROR *r/m16, imm8* | C | Valid | Valid | Rotate 16 bits *r/m16* right *imm8* times. |
| D1 /1 | ROR *r/m32*, 1 | A | Valid | Valid | Rotate 32 bits *r/m32* right once. |

| Opcode** | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| REX.W + D1 /1 | ROR *r/m64*, 1 | A | Valid | N.E. | Rotate 64 bits *r/m64* right once. Uses a 6 bit count. |
| D3 /1 | ROR *r/m32*, CL | B | Valid | Valid | Rotate 32 bits *r/m32* right CL times. |
| REX.W + D3 /1 | ROR *r/m64*, CL | B | Valid | N.E. | Rotate 64 bits *r/m64* right CL times. Uses a 6 bit count. |
| C1 /1 *ib* | ROR *r/m32, imm8* | C | Valid | Valid | Rotate 32 bits *r/m32* right *imm8* times. |
| REX.W + C1 /1 *ib* | ROR *r/m64, imm8* | C | Valid | N.E. | Rotate 64 bits *r/m64* right *imm8* 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 |
|---|---|---|---|---|
| A | ModRM:r/m (w) | 1 | NA | NA |
| B | ModRM:r/m (w) | CL (r) | NA | NA |
| C | ModRM:r/m (w) | imm8 | NA | NA |

...

## RCPPS—Compute Reciprocals of Packed Single-Precision Floating-Point Values

| Opcode* | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F 53 /*r* | RCPPS *xmm1*, *xmm2/m128* | A | Valid | Valid | Computes the approximate reciprocals of the packed single-precision floating-point values in *xmm2/m128* and stores the results in *xmm1*. |

**Instruction Operand Encoding**

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (w) | ModRM:r/m (r) | NA | NA |

...

## RCPSS—Compute Reciprocal of Scalar Single-Precision Floating-Point Values

| Opcode* | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| F3 0F 53 /*r* | RCPSS *xmm1*, xmm2/m32 | A | Valid | Valid | Computes the approximate reciprocal of the scalar single-precision floating-point value in *xmm2/m32* and stores the result in *xmm1*. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (w) | ModRM:r/m (r) | NA | NA |

...

## RDMSR—Read from Model Specific Register

| Opcode* | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F 32 | RDMSR | A | Valid | Valid | Read MSR specified by ECX into EDX:EAX. |

**NOTES:**

* See IA-32 Architecture Compatibility section below.

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | NA | NA | NA | NA |

...

## RDPMC—Read Performance-Monitoring Counters

| Opcode* | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F 33 | RDPMC | A | Valid | Valid | Read performance-monitoring counter specified by ECX into EDX:EAX. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | NA | NA | NA | NA |

## Description

The EAX register is loaded with the low-order 32 bits. The EDX register is loaded with the supported high-order bits of the counter. The number of high-order bits loaded into EDX is implementation specific on processors that do no support architectural performance monitoring. The width of fixed-function and general-purpose performance counters on processors supporting architectural performance monitoring are reported by CPUID 0AH leaf. See below for the treatment of the EDX register for "fast" reads.

The ECX register selects one of two type of performance counters, specifies the index relative to the base of each counter type, and selects "fast" read mode if supported. The two counter types are :

- General-purpose or special-purpose performance counters: The number of general-purpose counters is model specific if the processor does not support architectural performance monitoring, see Chapter 30 of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*. Special-purpose counters are available only in selected processor members, see Section 30.13, 30.14 of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*. This counter type is selected if ECX[30] is clear.
- Fixed-function performance counter. The number fixed-function performance counters is enumerated by CPUID 0AH leaf. See Chapter 30 of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*. This counter type is selected if ECX[30] is set.

ECX[29:0] specifies the index. The width of general-purpose performance counters are 40-bits for processors that do not support architectural performance monitoring counters.The width of special-purpose performance counters are implementation specific. The width of fixed-function performance counters and general-purpose performance counters on processor supporting architectural performance monitoring are reported by CPUID 0AH leaf.

Table 4-2 lists valid indices of the general-purpose and special-purpose performance counters according to the derived displayed_family/displayed_model values of CPUID encoding for each processor family.

### Table 4-2 Valid General and Special Purpose Performance Counter Index Range for RDPMC

| Processor Family | Displayed_Family_Displayed_Model/ Other Signatures | Valid PMC Index Range | General-purpose Counters |
|---|---|---|---|
| P6 | 06H_01H, 06H_03H, 06H_05H, 06H_06H, 06H_07H, 06H_08H, 06H_0AH, 06H_0BH | 0, 1 | 0, 1 |
| Pentium® 4, Intel® Xeon processors | 0FH_00H, 0FH_01H, 0FH_02H | ≥ 0 and ≤ 17 | ≥ 0 and ≤ 17 |
| Pentium 4, Intel Xeon processors | (0FH_03H, 0FH_04H, 0FH_06H) and (L3 is absent) | ≥ 0 and ≤ 17 | ≥ 0 and ≤ 17 |

**Table 4-2 Valid General and Special Purpose Performance Counter Index Range for RDPMC (Continued)**

| Processor Family | Displayed_Family_Displayed_Model/ Other Signatures | Valid PMC Index Range | General-purpose Counters |
|---|---|---|---|
| Pentium M processors | 06H_09H, 06H_0DH | 0, 1 | 0, 1 |
| 64-bit Intel Xeon processors with L3 | 0FH_03H, 0FH_04H) and (L3 is present) | ≥ 0 and ≤ 25 | ≥ 0 and ≤ 17 |
| Intel® Core™ Solo and Intel® Core™ Duo processors, Dual-core Intel® Xeon® processor LV | 06H_0EH | 0, 1 | 0, 1 |
| Intel® Core™2 Duo processor, Intel Xeon processor 3000, 5100, 5300, 7300 Series - general-purpose PMC | 06H_0FH | 0, 1 | 0, 1 |
| Intel Xeon processors 7100 series with L3 | (0FH_06H) and (L3 is present) | ≥ 0 and ≤ 25 | ≥ 0 and ≤ 17 |
| Intel® Core™2 Duo processor family, Intel Xeon processor family - general-purpose PMC | 06H_17H | 0, 1 | 0, 1 |
| Intel Xeon processors 7400 series | (06H_1DH) | ≥ 0 and ≤ 9 | 0, 1 |
| Intel® Atom™ processor family | 06H_1CH | 0, 1 | 0, 1 |
| Intel® Core™i7 processor, Intel Xeon processors 5500 series | 06H_1AH, 06H_1EH, 06H_1FH, 06H_2EH | 0-3 | 0, 1, 2, 3 |

The Pentium 4 and Intel Xeon processors also support "fast" (32-bit) and "slow" (40-bit) reads on the first 18 performance counters. Selected this option using ECX[31]. If bit 31 is set, RDPMC reads only the low 32 bits of the selected performance counter. If bit 31 is clear, all 40 bits are read. A 32-bit result is returned in EAX and EDX is set to 0. A 32-bit read executes faster on Pentium 4 processors and Intel Xeon processors than a full 40-bit read.

On 64-bit Intel Xeon processors with L3, performance counters with indices 18-25 are 32-bit counters. EDX is cleared after executing RDPMC for these counters. On Intel Xeon processor 7100 series with L3, performance counters with indices 18-25 are also 32-bit counters.

In Intel Core 2 processor family, Intel Xeon processor 3000, 5100, 5300 and 7400 series, the fixed-function performance counters are 40-bits wide; they can be accessed by RDMPC with ECX between from 4000_0000H and 4000_0002H.

On Intel Xeon processor 7400 series, there are eight 32-bit special-purpose counters addressable with indices 2-9, ECX[30]=0.

When in protected or virtual 8086 mode, the performance-monitoring counters enabled (PCE) flag in register CR4 restricts the use of the RDPMC instruction as follows. When the PCE flag is set, the RDPMC instruction can be executed at any privilege level; when the flag is clear, the instruction can only be executed at privilege level 0. (When in real-address mode, the RDPMC instruction is always enabled.)

The performance-monitoring counters can also be read with the RDMSR instruction, when executing at privilege level 0.

The performance-monitoring counters are event counters that can be programmed to count events such as the number of instructions decoded, number of interrupts received, or number of cache loads. Appendix A, “Performance Monitoring Events,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B*, lists the events that can be counted for various processors in the Intel 64 and IA-32 architecture families.

The RDPMC instruction is not a serializing instruction; that is, it does not imply that all the events caused by the preceding instructions have been completed or that events caused by subsequent instructions have not begun. If an exact event count is desired, software must insert a serializing instruction (such as the CPUID instruction) before and/or after the RDPMC instruction.

In the Pentium 4 and Intel Xeon processors, performing back-to-back fast reads are not guaranteed to be monotonic. To guarantee monotonicity on back-to-back reads, a serializing instruction must be placed between the two RDPMC instructions.

The RDPMC instruction can execute in 16-bit addressing mode or virtual-8086 mode; however, the full contents of the ECX register are used to select the counter, and the event count is stored in the full EAX and EDX registers. The RDPMC instruction was introduced into the IA-32 Architecture in the Pentium Pro processor and the Pentium processor with MMX technology. The earlier Pentium processors have performance-monitoring counters, but they must be read with the RDMSR instruction.

## Operation

```
(* Intel Core i7 processor family and Intel Xeon processor 3400, 5500 series*)

Most significant counter bit (MSCB) = 47

IF ((CR4.PCE = 1) or (CPL = 0) or (CR0.PE = 0))
    THEN IF (ECX[30] = 1 and ECX[29:0] in valid fixed-counter range)
        EAX ← IA32_FIXED_CTR(ECX)[30:0];
        EDX ← IA32_FIXED_CTR(ECX)[MSCB:32];
    ELSE IF (ECX[30] = 0 and ECX[29:0] in valid general-purpose counter range)
        EAX ← PMC(ECX[30:0])[31:0];
        EDX ← PMC(ECX[30:0])[MSCB:32];
    ELSE (* ECX is not valid or CR4.PCE is 0 and CPL is 1, 2, or 3 and CR0.PE is 1 *)
        #GP(0);
FI;


(* Intel Core 2 Duo processor family and Intel Xeon processor 3000, 5100, 5300, 7400 series*)

Most significant counter bit (MSCB) = 39

IF ((CR4.PCE = 1) or (CPL = 0) or (CR0.PE = 0))
    THEN IF (ECX[30] = 1 and ECX[29:0] in valid fixed-counter range)
        EAX ← IA32_FIXED_CTR(ECX)[30:0];
        EDX ← IA32_FIXED_CTR(ECX)[MSCB:32];
    ELSE IF (ECX[30] = 0 and ECX[29:0] in valid general-purpose counter range)
        EAX ← PMC(ECX[30:0])[31:0];
        EDX ← PMC(ECX[30:0])[MSCB:32];
    ELSE IF (ECX[30] = 0 and ECX[29:0] in valid special-purpose counter range)
        EAX ← PMC(ECX[30:0])[31:0]; (* 32-bit read *)
    ELSE (* ECX is not valid or CR4.PCE is 0 and CPL is 1, 2, or 3 and CR0.PE is 1 *)
        #GP(0);
FI;

(* P6 family processors and Pentium processor with MMX technology *)

IF (ECX = 0 or 1) and ((CR4.PCE = 1) or (CPL = 0) or (CR0.PE = 0))
    THEN
        EAX ← PMC(ECX)[31:0];
        EDX ← PMC(ECX)[39:32];
    ELSE (* ECX is not 0 or 1 or CR4.PCE is 0 and CPL is 1, 2, or 3 and CR0.PE is 1 *)
        #GP(0);
FI;
(* Processors with CPUID family 15 *)
IF ((CR4.PCE = 1) or (CPL = 0) or (CR0.PE = 0))
    THEN IF (ECX[30:0] = 0:17)
        THEN IF ECX[31] = 0
            THEN
                EAX ← PMC(ECX[30:0])[31:0]; (* 40-bit read *)
                EDX ← PMC(ECX[30:0])[39:32];
        ELSE (* ECX[31] = 1*)
            THEN
                EAX ← PMC(ECX[30:0])[31:0]; (* 32-bit read *)
                EDX ← 0;
        FI;
    ELSE IF (*64-bit Intel Xeon processor with L3 *)
        THEN IF (ECX[30:0] = 18:25 )
            EAX ← PMC(ECX[30:0])[31:0]; (* 32-bit read *)
            EDX ← 0;
        FI;
    ELSE IF (*Intel Xeon processor 7100 series with L3 *)
        THEN IF (ECX[30:0] = 18:25 )
            EAX ← PMC(ECX[30:0])[31:0]; (* 32-bit read *)
            EDX ← 0;
        FI;
    ELSE  (* Invalid PMC index in ECX[30:0], see Table 4-5. *)
        GP(0);
    FI;
ELSE  (* CR4.PCE = 0 and (CPL = 1, 2, or 3) and CR0.PE = 1 *)
    #GP(0);
FI;

...
```

## RDTSC—Read Time-Stamp Counter

| Opcode* | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F 31 | RDTSC | A | Valid | Valid | Read time-stamp counter into EDX:EAX. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | NA | NA | NA | NA |

### Description

Loads the current value of the processor's time-stamp counter (a 64-bit MSR) into the EDX:EAX registers. The EDX register is loaded with the high-order 32 bits of the MSR and the EAX register is loaded with the low-order 32 bits. (On processors that support the Intel 64 architecture, the high-order 32 bits of each of RAX and RDX are cleared.)

The processor monotonically increments the time-stamp counter MSR every clock cycle and resets it to 0 whenever the processor is reset. See "Time Stamp Counter" in Chapter 16 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*, for specific details of the time stamp counter behavior.

When in protected or virtual 8086 mode, the time stamp disable (TSD) flag in register CR4 restricts the use of the RDTSC instruction as follows. When the TSD flag is clear, the RDTSC instruction can be executed at any privilege level; when the flag is set, the instruction can only be executed at privilege level 0. (When in real-address mode, the RDTSC instruction is always enabled.)

The time-stamp counter can also be read with the RDMSR instruction, when executing at privilege level 0.

The RDTSC instruction is not a serializing instruction. It does not necessarily wait until all previous instructions have been executed before reading the counter. Similarly, subsequent instructions may begin execution before the read operation is performed. If software requires RDTSC to be executed only after all previous instructions have completed locally, it can either use RDTSCP (if the processor supports that instruction) or execute the sequence LFENCE;RDTSC.

This instruction was introduced by the Pentium processor.

See "Changes to Instruction Behavior in VMX Non-Root Operation" in Chapter 22 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*, for more information about the behavior of this instruction in VMX non-root operation.

...

## RDTSCP—Read Time-Stamp Counter and Processor ID

| Opcode* | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F 01 F9 | RDTSCP | A | Valid | Valid | Read 64-bit time-stamp counter and 32-bit IA32_TSC_AUX value into EDX:EAX and ECX. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | NA | NA | NA | NA |

...

## REP/REPE/REPZ/REPNE/REPNZ—Repeat String Operation Prefix

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| F3 6C | REP INS *m8*, DX | A | Valid | Valid | Input (E)CX bytes from port DX into ES:[(E)DI]. |
| F3 6C | REP INS *m8*, DX | A | Valid | N.E. | Input RCX bytes from port DX into [RDI]. |
| F3 6D | REP INS *m16*, DX | A | Valid | Valid | Input (E)CX words from port DX into ES:[(E)DI.] |
| F3 6D | REP INS *m32*, DX | A | Valid | Valid | Input (E)CX doublewords from port DX into ES:[(E)DI]. |
| F3 6D | REP INS *r/m32*, DX | A | Valid | N.E. | Input RCX default size from port DX into [RDI]. |
| F3 A4 | REP MOVS *m8, m8* | A | Valid | Valid | Move (E)CX bytes from DS:[(E)SI] to ES:[(E)DI]. |
| F3 REX.W A4 | REP MOVS *m8, m8* | A | Valid | N.E. | Move RCX bytes from [RSI] to [RDI]. |
| F3 A5 | REP MOVS *m16, m16* | A | Valid | Valid | Move (E)CX words from DS:[(E)SI] to ES:[(E)DI]. |
| F3 A5 | REP MOVS *m32, m32* | A | Valid | Valid | Move (E)CX doublewords from DS:[(E)SI] to ES:[(E)DI]. |
| F3 REX.W A5 | REP MOVS *m64, m64* | A | Valid | N.E. | Move RCX quadwords from [RSI] to [RDI]. |
| F3 6E | REP OUTS DX, *r/m8* | A | Valid | Valid | Output (E)CX bytes from DS:[(E)SI] to port DX. |
| F3 REX.W 6E | REP OUTS DX, *r/m8** | A | Valid | N.E. | Output RCX bytes from [RSI] to port DX. |
| F3 6F | REP OUTS DX, *r/m16* | A | Valid | Valid | Output (E)CX words from DS:[(E)SI] to port DX. |
| F3 6F | REP OUTS DX, *r/m32* | A | Valid | Valid | Output (E)CX doublewords from DS:[(E)SI] to port DX. |

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| F3 REX.W 6F | REP OUTS DX, *r/m32* | A | Valid | N.E. | Output RCX default size from [RSI] to port DX. |
| F3 AC | REP LODS AL | A | Valid | Valid | Load (E)CX bytes from DS:[(E)SI] to AL. |
| F3 REX.W AC | REP LODS AL | A | Valid | N.E. | Load RCX bytes from [RSI] to AL. |
| F3 AD | REP LODS AX | A | Valid | Valid | Load (E)CX words from DS:[(E)SI] to AX. |
| F3 AD | REP LODS EAX | A | Valid | Valid | Load (E)CX doublewords from DS:[(E)SI] to EAX. |
| F3 REX.W AD | REP LODS RAX | A | Valid | N.E. | Load RCX quadwords from [RSI] to RAX. |
| F3 AA | REP STOS *m8* | A | Valid | Valid | Fill (E)CX bytes at ES:[(E)DI] with AL. |
| F3 REX.W AA | REP STOS *m8* | A | Valid | N.E. | Fill RCX bytes at [RDI] with AL. |
| F3 AB | REP STOS *m16* | A | Valid | Valid | Fill (E)CX words at ES:[(E)DI] with AX. |
| F3 AB | REP STOS *m32* | A | Valid | Valid | Fill (E)CX doublewords at ES:[(E)DI] with EAX. |
| F3 REX.W AB | REP STOS *m64* | A | Valid | N.E. | Fill RCX quadwords at [RDI] with RAX. |
| F3 A6 | REPE CMPS *m8*, *m8* | A | Valid | Valid | Find nonmatching bytes in ES:[(E)DI] and DS:[(E)SI]. |
| F3 REX.W A6 | REPE CMPS *m8*, *m8* | A | Valid | N.E. | Find non-matching bytes in [RDI] and [RSI]. |
| F3 A7 | REPE CMPS *m16*, *m16* | A | Valid | Valid | Find nonmatching words in ES:[(E)DI] and DS:[(E)SI]. |
| F3 A7 | REPE CMPS *m32*, *m32* | A | Valid | Valid | Find nonmatching doublewords in ES:[(E)DI] and DS:[(E)SI]. |
| F3 REX.W A7 | REPE CMPS *m64*, *m64* | A | Valid | N.E. | Find non-matching quadwords in [RDI] and [RSI]. |
| F3 AE | REPE SCAS *m8* | A | Valid | Valid | Find non-AL byte starting at ES:[(E)DI]. |
| F3 REX.W AE | REPE SCAS *m8* | A | Valid | N.E. | Find non-AL byte starting at [RDI]. |
| F3 AF | REPE SCAS *m16* | A | Valid | Valid | Find non-AX word starting at ES:[(E)DI]. |
| F3 AF | REPE SCAS *m32* | A | Valid | Valid | Find non-EAX doubleword starting at ES:[(E)DI]. |
| F3 REX.W AF | REPE SCAS *m64* | A | Valid | N.E. | Find non-RAX quadword starting at [RDI]. |

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| F2 A6 | REPNE CMPS *m8, m8* | A | Valid | Valid | Find matching bytes in ES:[(E)DI] and DS:[(E)SI]. |
| F2 REX.W A6 | REPNE CMPS *m8, m8* | A | Valid | N.E. | Find matching bytes in [RDI] and [RSI]. |
| F2 A7 | REPNE CMPS *m16, m16* | A | Valid | Valid | Find matching words in ES:[(E)DI] and DS:[(E)SI]. |
| F2 A7 | REPNE CMPS *m32, m32* | A | Valid | Valid | Find matching doublewords in ES:[(E)DI] and DS:[(E)SI]. |
| F2 REX.W A7 | REPNE CMPS *m64, m64* | A | Valid | N.E. | Find matching doublewords in [RDI] and [RSI]. |
| F2 AE | REPNE SCAS *m8* | A | Valid | Valid | Find AL, starting at ES:[(E)DI]. |
| F2 REX.W AE | REPNE SCAS *m8* | A | Valid | N.E. | Find AL, starting at [RDI]. |
| F2 AF | REPNE SCAS *m16* | A | Valid | Valid | Find AX, starting at ES:[(E)DI]. |
| F2 AF | REPNE SCAS *m32* | A | Valid | Valid | Find EAX, starting at ES:[(E)DI]. |
| F2 REX.W AF | REPNE SCAS *m64* | A | Valid | N.E. | Find RAX, starting at [RDI]. |

**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 |
|---|---|---|---|---|
| A | NA | NA | NA | NA |

...

## RET—Return from Procedure

| Opcode* | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| C3 | RET | A | Valid | Valid | Near return to calling procedure. |
| CB | RET | A | Valid | Valid | Far return to calling procedure. |
| C2 *iw* | RET *imm16* | B | Valid | Valid | Near return to calling procedure and pop *imm16* bytes from stack. |
| CA *iw* | RET *imm16* | B | Valid | Valid | Far return to calling procedure and pop *imm16* bytes from stack. |

**Instruction Operand Encoding**

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | NA | NA | NA | NA |
| B | imm16 | NA | NA | NA |

...

## ROUNDPD — Round Packed Double Precision Floating-Point Values

| Opcode* | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 66 0F 3A 09 /r ib | ROUNDPD *xmm1, xmm2/m128, imm8* | A | Valid | Valid | Round packed double precision floating-point values in *xmm2/m128* and place the result in *xmm1*. The rounding mode is determined by *imm8*. |

**Instruction Operand Encoding**

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (w) | ModRM:r/m (r) | imm8 | NA |

...

## ROUNDPS — Round Packed Single Precision Floating-Point Values

| Opcode* | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 66 0F 3A 08 /r ib | ROUNDPS *xmm1, xmm2/m128, imm8* | A | Valid | Valid | Round packed single precision floating-point values in *xmm2/m128* and place the result in *xmm1*. The rounding mode is determined by *imm8*. |

**Instruction Operand Encoding**

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (w) | ModRM:r/m (r) | imm8 | NA |

...

## ROUNDSD — Round Scalar Double Precision Floating-Point Values

| Opcode* | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 66 0F 3A 0B /r ib | ROUNDSD *xmm1, xmm2/m64, imm8* | A | Valid | Valid | Round the low packed double precision floating-point value in *xmm2/m64* and place the result in *xmm1*. The rounding mode is determined by *imm8*. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (w) | ModRM:r/m (r) | imm8 | NA |

...

## ROUNDSS — Round Scalar Single Precision Floating-Point Values

| Opcode* | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 66 0F 3A 0A /r ib | ROUNDSS *xmm1, xmm2/m32, imm8* | A | Valid | Valid | Round the low packed single precision floating-point value in *xmm2/m32* and place the result in *xmm1*. The rounding mode is determined by *imm8*. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (w) | ModRM:r/m (r) | imm8 | NA |

...

## RSM—Resume from System Management Mode

| Opcode* | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F AA | RSM | A | Invalid | Valid | Resume operation of interrupted program. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | NA | NA | NA | NA |

...

## RSQRTPS—Compute Reciprocals of Square Roots of Packed Single-Precision Floating-Point Values

| Opcode* | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F 52 /*r* | RSQRTPS *xmm1*, *xmm2/m128* | A | Valid | Valid | Computes the approximate reciprocals of the square roots of the packed single-precision floating-point values in *xmm2/m128* and stores the results in *xmm1*. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (w) | ModRM:r/m (r) | NA | NA |

...

## RSQRTSS—Compute Reciprocal of Square Root of Scalar Single-Precision Floating-Point Value

| Opcode* | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| F3 0F 52 /*r* | RSQRTSS *xmm1*, *xmm2/m32* | A | Valid | Valid | Computes the approximate reciprocal of the square root of the low single-precision floating-point value in *xmm2/m32* and stores the results in *xmm1*. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (w) | ModRM:r/m (r) | NA | NA |

...

## SAHF—Store AH into Flags

| Opcode* | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 9E | SAHF | A | Invalid* | Valid | Loads SF, ZF, AF, PF, and CF from AH into EFLAGS register. |

**NOTES:**

* Valid in specific steppings. See Description section.

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | NA | NA | NA | NA |

...

## SAL/SAR/SHL/SHR—Shift

| **Opcode***** | **Instruction** | **Op/ En** | **64-Bit Mode** | **Compat/ Leg Mode** | **Description** |
|---|---|---|---|---|---|
| D0 /4 | SAL *r/m8*, 1 | A | Valid | Valid | Multiply *r/m8* by 2, once. |
| REX + D0 /4 | SAL *r/m8***, 1 | A | Valid | N.E. | Multiply *r/m8* by 2, once. |
| D2 /4 | SAL *r/m8*, CL | B | Valid | Valid | Multiply *r/m8* by 2, CL times. |
| REX + D2 /4 | SAL *r/m8***, CL | B | Valid | N.E. | Multiply *r/m8* by 2, CL times. |
| C0 /4 *ib* | SAL *r/m8, imm8* | C | Valid | Valid | Multiply *r/m8* by 2, *imm8* times. |
| REX + C0 /4 *ib* | SAL *r/m8**, imm8* | C | Valid | N.E. | Multiply *r/m8* by 2, *imm8* times. |
| D1 /4 | SAL *r/m16*, 1 | A | Valid | Valid | Multiply *r/m16* by 2, once. |
| D3 /4 | SAL *r/m16*, CL | B | Valid | Valid | Multiply *r/m16* by 2, CL times. |
| C1 /4 *ib* | SAL *r/m16, imm8* | C | Valid | Valid | Multiply *r/m16* by 2, *imm8* times. |
| D1 /4 | SAL *r/m32*, 1 | A | Valid | Valid | Multiply *r/m32* by 2, once. |
| REX.W + D1 /4 | SAL *r/m64*, 1 | A | Valid | N.E. | Multiply *r/m64* by 2, once. |
| D3 /4 | SAL *r/m32*, CL | B | Valid | Valid | Multiply *r/m32* by 2, CL times. |
| REX.W + D3 /4 | SAL *r/m64*, CL | B | Valid | N.E. | Multiply *r/m64* by 2, CL times. |
| C1 /4 *ib* | SAL *r/m32, imm8* | C | Valid | Valid | Multiply *r/m32* by 2, *imm8* times. |
| REX.W + C1 /4 *ib* | SAL *r/m64, imm8* | C | Valid | N.E. | Multiply *r/m64* by 2, *imm8* times. |
| D0 /7 | SAR *r/m8*, 1 | A | Valid | Valid | Signed divide* *r/m8* by 2, once. |
| REX + D0 /7 | SAR *r/m8***, 1 | A | Valid | N.E. | Signed divide* *r/m8* by 2, once. |
| D2 /7 | SAR *r/m8*, CL | B | Valid | Valid | Signed divide* *r/m8* by 2, CL times. |
| REX + D2 /7 | SAR *r/m8***, CL | B | Valid | N.E. | Signed divide* *r/m8* by 2, CL times. |
| C0 /7 *ib* | SAR *r/m8, imm8* | C | Valid | Valid | Signed divide* *r/m8* by 2, *imm8* time. |
| REX + C0 /7 *ib* | SAR *r/m8**, imm8* | C | Valid | N.E. | Signed divide* *r/m8* by 2, *imm8* times. |

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| D1 /7 | SAR *r/m16*,1 | A | Valid | Valid | Signed divide* *r/m16* by 2, once. |
| D3 /7 | SAR *r/m16*, CL | B | Valid | Valid | Signed divide* *r/m16* by 2, CL times. |
| C1 /7 *ib* | SAR *r/m16, imm8* | C | Valid | Valid | Signed divide* *r/m16* by 2, *imm8* times. |
| D1 /7 | SAR *r/m32*, 1 | A | Valid | Valid | Signed divide* *r/m32* by 2, once. |
| REX.W + D1 /7 | SAR *r/m64*, 1 | A | Valid | N.E. | Signed divide* *r/m64* by 2, once. |
| D3 /7 | SAR *r/m32*, CL | B | Valid | Valid | Signed divide* *r/m32* by 2, CL times. |
| REX.W + D3 /7 | SAR *r/m64*, CL | B | Valid | N.E. | Signed divide* *r/m64* by 2, CL times. |
| C1 /7 *ib* | SAR *r/m32, imm8* | C | Valid | Valid | Signed divide* *r/m32* by 2, *imm8* times. |
| REX.W + C1 /7 *ib* | SAR *r/m64, imm8* | C | Valid | N.E. | Signed divide* *r/m64* by 2, *imm8* times |
| D0 /4 | SHL *r/m8*, 1 | A | Valid | Valid | Multiply *r/m8* by 2, once. |
| REX + D0 /4 | SHL *r/m8***, 1 | A | Valid | N.E. | Multiply *r/m8* by 2, once. |
| D2 /4 | SHL *r/m8*, CL | B | Valid | Valid | Multiply *r/m8* by 2, CL times. |
| REX + D2 /4 | SHL *r/m8***, CL | B | Valid | N.E. | Multiply *r/m8* by 2, CL times. |
| C0 /4 *ib* | SHL *r/m8, imm8* | C | Valid | Valid | Multiply *r/m8* by 2, *imm8* times. |
| REX + C0 /4 *ib* | SHL *r/m8***, *imm8* | C | Valid | N.E. | Multiply *r/m8* by 2, *imm8* times. |
| D1 /4 | SHL *r/m16*,1 | A | Valid | Valid | Multiply *r/m16* by 2, once. |
| D3 /4 | SHL *r/m16*, CL | B | Valid | Valid | Multiply *r/m16* by 2, CL times. |
| C1 /4 *ib* | SHL *r/m16, imm8* | C | Valid | Valid | Multiply *r/m16* by 2, *imm8* times. |
| D1 /4 | SHL *r/m32*,1 | A | Valid | Valid | Multiply *r/m32* by 2, once. |
| REX.W + D1 /4 | SHL *r/m64*,1 | A | Valid | N.E. | Multiply *r/m64* by 2, once. |
| D3 /4 | SHL *r/m32*, CL | B | Valid | Valid | Multiply *r/m32* by 2, CL times. |
| REX.W + D3 /4 | SHL *r/m64*, CL | B | Valid | N.E. | Multiply *r/m64* by 2, CL times. |
| C1 /4 *ib* | SHL *r/m32, imm8* | C | Valid | Valid | Multiply *r/m32* by 2, *imm8* times. |
| REX.W + C1 /4 *ib* | SHL *r/m64, imm8* | C | Valid | N.E. | Multiply *r/m64* by 2, *imm8* times. |
| D0 /5 | SHR *r/m8*,1 | A | Valid | Valid | Unsigned divide *r/m8* by 2, once. |

| **Opcode*** ** | **Instruction** | **Op/ En** | **64-Bit Mode** | **Compat/ Leg Mode** | **Description** |
|---|---|---|---|---|---|
| REX + D0 /5 | SHR *r/m8***, 1 | A | Valid | N.E. | Unsigned divide *r/m8* by 2, once. |
| D2 /5 | SHR *r/m8*, CL | B | Valid | Valid | Unsigned divide *r/m8* by 2, CL times. |
| REX + D2 /5 | SHR *r/m8***, CL | B | Valid | N.E. | Unsigned divide *r/m8* by 2, CL times. |
| C0 /5 *ib* | SHR *r/m8, imm8* | C | Valid | Valid | Unsigned divide *r/m8* by 2, *imm8* times. |
| REX + C0 /5 *ib* | SHR *r/m8***, *imm8* | C | Valid | N.E. | Unsigned divide *r/m8* by 2, *imm8* times. |
| D1 /5 | SHR *r/m16*, 1 | A | Valid | Valid | Unsigned divide *r/m16* by 2, once. |
| D3 /5 | SHR *r/m16*, CL | B | Valid | Valid | Unsigned divide *r/m16* by 2, CL times |
| C1 /5 *ib* | SHR *r/m16, imm8* | C | Valid | Valid | Unsigned divide *r/m16* by 2, *imm8* times. |
| D1 /5 | SHR *r/m32*, 1 | A | Valid | Valid | Unsigned divide *r/m32* by 2, once. |
| REX.W + D1 /5 | SHR *r/m64*, 1 | A | Valid | N.E. | Unsigned divide *r/m64* by 2, once. |
| D3 /5 | SHR *r/m32*, CL | B | Valid | Valid | Unsigned divide *r/m32* by 2, CL times. |
| REX.W + D3 /5 | SHR *r/m64*, CL | B | Valid | N.E. | Unsigned divide *r/m64* by 2, CL times. |
| C1 /5 *ib* | SHR *r/m32, imm8* | C | Valid | Valid | Unsigned divide *r/m32* by 2, *imm8* times. |
| REX.W + C1 /5 *ib* | SHR *r/m64, imm8* | C | Valid | N.E. | Unsigned divide *r/m64* by 2, *imm8* times. |

**NOTES:**

* Not the same form of division as IDIV; rounding is toward negative infinity.

** 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 |
|---|---|---|---|---|
| A | ModRM:r/m (r, w) | 1 | NA | NA |
| B | ModRM:r/m (r, w) | CL (r) | NA | NA |
| C | ModRM:r/m (r, w) | imm8 | NA | NA |

...

## SBB—Integer Subtraction with Borrow

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 1C *ib* | SBB AL, *imm8* | A | Valid | Valid | Subtract with borrow *imm8* from AL. |
| 1D *iw* | SBB AX, *imm16* | A | Valid | Valid | Subtract with borrow *imm16* from AX. |
| 1D *id* | SBB EAX, *imm32* | A | Valid | Valid | Subtract with borrow *imm32* from EAX. |
| REX.W + 1D *id* | SBB RAX, *imm32* | A | Valid | N.E. | Subtract with borrow sign-extended *imm.32 to 64-bits* from RAX. |
| 80 /3 *ib* | SBB *r/m8, imm8* | B | Valid | Valid | Subtract with borrow *imm8* from *r/m8.* |
| REX + 80 /3 *ib* | SBB *r/m8**, *imm8* | B | Valid | N.E. | Subtract with borrow *imm8* from *r/m8.* |
| 81 /3 *iw* | SBB *r/m16, imm16* | B | Valid | Valid | Subtract with borrow *imm16* from *r/m16.* |
| 81 /3 *id* | SBB *r/m32, imm32* | B | Valid | Valid | Subtract with borrow *imm32* from *r/m32.* |
| REX.W + 81 /3 *id* | SBB *r/m64, imm32* | B | Valid | N.E. | Subtract with borrow sign-extended *imm32 to 64-bits* from *r/m64.* |
| 83 /3 *ib* | SBB *r/m16, imm8* | B | Valid | Valid | Subtract with borrow sign-extended *imm8* from *r/m16.* |
| 83 /3 *ib* | SBB *r/m32, imm8* | B | Valid | Valid | Subtract with borrow sign-extended *imm8* from *r/m32.* |
| REX.W + 83 /3 *ib* | SBB *r/m64, imm8* | B | Valid | N.E. | Subtract with borrow sign-extended *imm8* from *r/m64.* |
| 18 /*r* | SBB *r/m8, r8* | C | Valid | Valid | Subtract with borrow *r8* from *r/m8.* |
| REX + 18 /*r* | SBB *r/m8**, *r8* | C | Valid | N.E. | Subtract with borrow *r8* from *r/m8.* |
| 19 /*r* | SBB *r/m16, r16* | C | Valid | Valid | Subtract with borrow *r16* from *r/m16.* |
| 19 /*r* | SBB *r/m32, r32* | C | Valid | Valid | Subtract with borrow *r32* from *r/m32.* |
| REX.W + 19 /*r* | SBB *r/m64, r64* | C | Valid | N.E. | Subtract with borrow *r64* from *r/m64.* |
| 1A /*r* | SBB *r8, r/m8* | D | Valid | Valid | Subtract with borrow *r/m8* from *r8.* |
| REX + 1A /*r* | SBB *r8**, *r/m8** | D | Valid | N.E. | Subtract with borrow *r/m8* from *r8.* |
| 1B /*r* | SBB *r16, r/m16* | D | Valid | Valid | Subtract with borrow *r/m16* from *r16.* |

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 1B /r | SBB *r32, r/m32* | D | Valid | Valid | Subtract with borrow *r/m32* from *r32.* |
| REX.W + 1B /r | SBB *r64, r/m64* | D | Valid | N.E. | Subtract with borrow *r/m64* from *r64.* |

**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 |
|---|---|---|---|---|
| A | AL/AX/EAX/RAX | imm8/16/32 | NA | NA |
| B | ModRM:r/m (w) | imm8/16/32 | NA | NA |
| C | ModRM:r/m (w) | ModRM:reg (r) | NA | NA |
| D | ModRM:reg (w) | ModRM:r/m (r) | NA | NA |

...

## SCAS/SCASB/SCASW/SCASD—Scan String

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| AE | SCAS *m8* | A | Valid | Valid | Compare AL with byte at ES:(E)DI or RDI, then set status flags.* |
| AF | SCAS *m16* | A | Valid | Valid | Compare AX with word at ES:(E)DI or RDI, then set status flags.* |
| AF | SCAS *m32* | A | Valid | Valid | Compare EAX with doubleword at ES(E)DI or RDI then set status flags.* |
| REX.W + AF | SCAS *m64* | A | Valid | N.E. | Compare RAX with quadword at RDI or EDI then set status flags. |
| AE | SCASB | A | Valid | Valid | Compare AL with byte at ES:(E)DI or RDI then set status flags.* |
| AF | SCASW | A | Valid | Valid | Compare AX with word at ES:(E)DI or RDI then set status flags.* |
| AF | SCASD | A | Valid | Valid | Compare EAX with doubleword at ES:(E)DI or RDI then set status flags.* |
| REX.W + AF | SCASQ | A | Valid | N.E. | Compare RAX with quadword at RDI or EDI then set status flags. |

**NOTES:**

* In 64-bit mode, only 64-bit (RDI) and 32-bit (EDI) address sizes are supported. In non-64-bit mode, only 32-bit (EDI) and 16-bit (DI) address sizes are supported.

**Instruction Operand Encoding**

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | NA | NA | NA | NA |

...

## SETcc—Set Byte on Condition

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F 97 | SETA *r/m8* | A | Valid | Valid | Set byte if above (CF=0 and ZF=0). |
| REX + 0F 97 | SETA *r/m8** | A | Valid | N.E. | Set byte if above (CF=0 and ZF=0). |
| 0F 93 | SETAE *r/m8* | A | Valid | Valid | Set byte if above or equal (CF=0). |
| REX + 0F 93 | SETAE *r/m8** | A | Valid | N.E. | Set byte if above or equal (CF=0). |
| 0F 92 | SETB *r/m8* | A | Valid | Valid | Set byte if below (CF=1). |
| REX + 0F 92 | SETB *r/m8** | A | Valid | N.E. | Set byte if below (CF=1). |
| 0F 96 | SETBE *r/m8* | A | Valid | Valid | Set byte if below or equal (CF=1 or ZF=1). |
| REX + 0F 96 | SETBE *r/m8** | A | Valid | N.E. | Set byte if below or equal (CF=1 or ZF=1). |
| 0F 92 | SETC *r/m8* | A | Valid | Valid | Set byte if carry (CF=1). |
| REX + 0F 92 | SETC *r/m8** | A | Valid | N.E. | Set byte if carry (CF=1). |
| 0F 94 | SETE *r/m8* | A | Valid | Valid | Set byte if equal (ZF=1). |
| REX + 0F 94 | SETE *r/m8** | A | Valid | N.E. | Set byte if equal (ZF=1). |
| 0F 9F | SETG *r/m8* | A | Valid | Valid | Set byte if greater (ZF=0 and SF=OF). |
| REX + 0F 9F | SETG *r/m8** | A | Valid | N.E. | Set byte if greater (ZF=0 and SF=OF). |
| 0F 9D | SETGE *r/m8* | A | Valid | Valid | Set byte if greater or equal (SF=OF). |
| REX + 0F 9D | SETGE *r/m8** | A | Valid | N.E. | Set byte if greater or equal (SF=OF). |
| 0F 9C | SETL *r/m8* | A | Valid | Valid | Set byte if less (SF≠ OF). |
| REX + 0F 9C | SETL *r/m8** | A | Valid | N.E. | Set byte if less (SF≠ OF). |
| 0F 9E | SETLE *r/m8* | A | Valid | Valid | Set byte if less or equal (ZF=1 or SF≠ OF). |

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| REX + 0F 9E | SETLE *r/m8** | A | Valid | N.E. | Set byte if less or equal (ZF=1 or SF≠ OF). |
| 0F 96 | SETNA *r/m8* | A | Valid | Valid | Set byte if not above (CF=1 or ZF=1). |
| REX + 0F 96 | SETNA *r/m8** | A | Valid | N.E. | Set byte if not above (CF=1 or ZF=1). |
| 0F 92 | SETNAE *r/m8* | A | Valid | Valid | Set byte if not above or equal (CF=1). |
| REX + 0F 92 | SETNAE *r/m8** | A | Valid | N.E. | Set byte if not above or equal (CF=1). |
| 0F 93 | SETNB *r/m8* | A | Valid | Valid | Set byte if not below (CF=0). |
| REX + 0F 93 | SETNB *r/m8** | A | Valid | N.E. | Set byte if not below (CF=0). |
| 0F 97 | SETNBE *r/m8* | A | Valid | Valid | Set byte if not below or equal (CF=0 and ZF=0). |
| REX + 0F 97 | SETNBE *r/m8** | A | Valid | N.E. | Set byte if not below or equal (CF=0 and ZF=0). |
| 0F 93 | SETNC *r/m8* | A | Valid | Valid | Set byte if not carry (CF=0). |
| REX + 0F 93 | SETNC *r/m8** | A | Valid | N.E. | Set byte if not carry (CF=0). |
| 0F 95 | SETNE *r/m8* | A | Valid | Valid | Set byte if not equal (ZF=0). |
| REX + 0F 95 | SETNE *r/m8** | A | Valid | N.E. | Set byte if not equal (ZF=0). |
| 0F 9E | SETNG *r/m8* | A | Valid | Valid | Set byte if not greater (ZF=1 or SF≠ OF) |
| REX + 0F 9E | SETNG *r/m8** | A | Valid | N.E. | Set byte if not greater (ZF=1 or SF≠ OF). |
| 0F 9C | SETNGE *r/m8* | A | Valid | Valid | Set byte if not greater or equal (SF≠ OF). |
| REX + 0F 9C | SETNGE *r/m8** | A | Valid | N.E. | Set byte if not greater or equal (SF≠ OF). |
| 0F 9D | SETNL *r/m8* | A | Valid | Valid | Set byte if not less (SF=OF). |
| REX + 0F 9D | SETNL *r/m8** | A | Valid | N.E. | Set byte if not less (SF=OF). |
| 0F 9F | SETNLE *r/m8* | A | Valid | Valid | Set byte if not less or equal (ZF=0 and SF=OF). |
| REX + 0F 9F | SETNLE *r/m8** | A | Valid | N.E. | Set byte if not less or equal (ZF=0 and SF=OF). |
| 0F 91 | SETNO *r/m8* | A | Valid | Valid | Set byte if not overflow (OF=0). |
| REX + 0F 91 | SETNO *r/m8** | A | Valid | N.E. | Set byte if not overflow (OF=0). |
| 0F 9B | SETNP *r/m8* | A | Valid | Valid | Set byte if not parity (PF=0). |
| REX + 0F 9B | SETNP *r/m8** | A | Valid | N.E. | Set byte if not parity (PF=0). |
| 0F 99 | SETNS *r/m8* | A | Valid | Valid | Set byte if not sign (SF=0). |

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| REX + 0F 99 | SETNS *r/m8** | A | Valid | N.E. | Set byte if not sign (SF=0). |
| 0F 95 | SETNZ *r/m8* | A | Valid | Valid | Set byte if not zero (ZF=0). |
| REX + 0F 95 | SETNZ *r/m8** | A | Valid | N.E. | Set byte if not zero (ZF=0). |
| 0F 90 | SETO *r/m8* | A | Valid | Valid | Set byte if overflow (OF=1) |
| REX + 0F 90 | SETO *r/m8** | A | Valid | N.E. | Set byte if overflow (OF=1). |
| 0F 9A | SETP *r/m8* | A | Valid | Valid | Set byte if parity (PF=1). |
| REX + 0F 9A | SETP *r/m8** | A | Valid | N.E. | Set byte if parity (PF=1). |
| 0F 9A | SETPE *r/m8* | A | Valid | Valid | Set byte if parity even (PF=1). |
| REX + 0F 9A | SETPE *r/m8** | A | Valid | N.E. | Set byte if parity even (PF=1). |
| 0F 9B | SETPO *r/m8* | A | Valid | Valid | Set byte if parity odd (PF=0). |
| REX + 0F 9B | SETPO *r/m8** | A | Valid | N.E. | Set byte if parity odd (PF=0). |
| 0F 98 | SETS *r/m8* | A | Valid | Valid | Set byte if sign (SF=1). |
| REX + 0F 98 | SETS *r/m8** | A | Valid | N.E. | Set byte if sign (SF=1). |
| 0F 94 | SETZ *r/m8* | A | Valid | Valid | Set byte if zero (ZF=1). |
| REX + 0F 94 | SETZ *r/m8** | A | Valid | N.E. | Set byte if zero (ZF=1). |

**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 |
|---|---|---|---|---|
| A | ModRM:r/m (r) | NA | NA | NA |

...

## SFENCE—Store Fence

| Opcode* | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F AE /7 | SFENCE | A | Valid | Valid | Serializes store operations. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | NA | NA | NA | NA |

### Description

Performs a serializing operation on all store-to-memory instructions that were issued prior the SFENCE instruction. This serializing operation guarantees that every store instruction that precedes the SFENCE instruction in program order becomes globally visible before any store instruction that follows the SFENCE instruction. The SFENCE instruction is ordered with respect to store instructions, other SFENCE instructions, any LFENCE and MFENCE instructions, and any serializing instructions (such as the CPUID instruction). It is not ordered with respect to load instructions.

Weakly ordered memory types can be used to achieve higher processor performance through such techniques as out-of-order issue, write-combining, and write-collapsing. The degree to which a consumer of data recognizes or knows that the data is weakly ordered varies among applications and may be unknown to the producer of this data. The SFENCE instruction provides a performance-efficient way of ensuring store ordering between routines that produce weakly-ordered results and routines that consume this data.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

...

## SGDT—Store Global Descriptor Table Register

| **Opcode*** | **Instruction** | **Op/ En** | **64-Bit Mode** | **Compat/ Leg Mode** | **Description** |
|---|---|---|---|---|---|
| 0F 01 /0 | SGDT *m* | A | Valid | Valid | Store GDTR to *m.* |

**NOTES:**

* See IA-32 Architecture Compatibility section below.

**Instruction Operand Encoding**

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:r/m (w) | NA | NA | NA |

...

## SHLD—Double Precision Shift Left

| **Opcode*** | **Instruction** | **Op/ En** | **64-Bit Mode** | **Compat/ Leg Mode** | **Description** |
|---|---|---|---|---|---|
| 0F A4 | SHLD *r/m16, r16, imm8* | A | Valid | Valid | Shift *r/m16* to left *imm8* places while shifting bits from *r16* in from the right. |
| 0F A5 | SHLD *r/m16, r16,* CL | B | Valid | Valid | Shift *r/m16* to left CL places while shifting bits from *r16* in from the right. |
| 0F A4 | SHLD *r/m32, r32, imm8* | A | Valid | Valid | Shift *r/m32* to left *imm8* places while shifting bits from *r32* in from the right. |
| REX.W + 0F A4 | SHLD *r/m64, r64, imm8* | A | Valid | N.E. | Shift *r/m64* to left *imm8* places while shifting bits from *r64* in from the right. |

| Opcode* | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F A5 | SHLD *r/m32*, *r32*, CL | B | Valid | Valid | Shift *r/m32* to left CL places while shifting bits from *r32* in from the right. |
| REX.W + 0F A5 | SHLD *r/m64*, *r64*, CL | B | Valid | N.E. | Shift *r/m64* to left CL places while shifting bits from *r64* in from the right. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:r/m (w) | ModRM:reg (r) | imm8 | NA |
| B | ModRM:r/m (w) | ModRM:reg (r) | CL | NA |

...

## SHRD—Double Precision Shift Right

| Opcode* | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F AC | SHRD *r/m16*, *r16*, *imm8* | A | Valid | Valid | Shift *r/m16* to right *imm8* places while shifting bits from *r16* in from the left. |
| 0F AD | SHRD *r/m16*, *r16*, CL | B | Valid | Valid | Shift *r/m16* to right CL places while shifting bits from *r16* in from the left. |
| 0F AC | SHRD *r/m32*, *r32*, *imm8* | A | Valid | Valid | Shift *r/m32* to right *imm8* places while shifting bits from *r32* in from the left. |
| REX.W + 0F AC | SHRD *r/m64*, *r64*, *imm8* | A | Valid | N.E. | Shift *r/m64* to right *imm8* places while shifting bits from *r64* in from the left. |
| 0F AD | SHRD *r/m32*, *r32*, CL | B | Valid | Valid | Shift *r/m32* to right CL places while shifting bits from *r32* in from the left. |
| REX.W + 0F AD | SHRD *r/m64*, *r64*, CL | B | Valid | N.E. | Shift *r/m64* to right CL places while shifting bits from *r64* in from the left. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:r/m (w) | ModRM:reg (r) | imm8 | NA |
| B | ModRM:r/m (w) | ModRM:reg (r) | CL | NA |

...

## SHUFPD—Shuffle Packed Double-Precision Floating-Point Values

| **Opcode*** | **Instruction** | **Op/ En** | **64-Bit Mode** | **Compat/ Leg Mode** | **Description** |
|---|---|---|---|---|---|
| 66 0F C6 /*r* ib | SHUFPD *xmm1*, *xmm2/m128*, *imm8* | A | Valid | Valid | Shuffle packed double-precision floating-point values selected by *imm8* from *xmm1* and *xmm2/m128* to *xmm1*. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (r, w) | ModRM:r/m (r) | imm8 | NA |

...

## SHUFPS—Shuffle Packed Single-Precision Floating-Point Values

| **Opcode*** | **Instruction** | **Op/ En** | **64-Bit Mode** | **Compat/ Leg Mode** | **Description** |
|---|---|---|---|---|---|
| 0F C6 /*r* ib | SHUFPS *xmm1*, *xmm2/m128*, *imm8* | A | Valid | Valid | Shuffle packed single-precision floating-point values selected by *imm8* from *xmm1* and *xmm1/m128* to *xmm1*. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (r, w) | ModRM:r/m (r) | imm8 | NA |

...

## SIDT—Store Interrupt Descriptor Table Register

| **Opcode*** | **Instruction** | **Op/ En** | **64-Bit Mode** | **Compat/ Leg Mode** | **Description** |
|---|---|---|---|---|---|
| 0F 01 /1 | SIDT *m* | A | Valid | Valid | Store IDTR to *m.* |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:r/m (w) | NA | NA | NA |

...

## SLDT—Store Local Descriptor Table Register

| Opcode* | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F 00 /0 | SLDT *r/m16* | A | Valid | Valid | Stores segment selector from LDTR in *r/m16*. |
| REX.W + 0F 00 /0 | SLDT *r64/m16* | A | Valid | Valid | Stores segment selector from LDTR in *r64/m16*. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:r/m (w) | NA | NA | NA |

...

## SMSW—Store Machine Status Word

| Opcode* | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F 01 /4 | SMSW *r/m16* | A | Valid | Valid | Store machine status word to *r/m16*. |
| 0F 01 /4 | SMSW *r32/m16* | A | Valid | Valid | Store machine status word in low-order 16 bits of *r32/m16*; high-order 16 bits of *r32* are undefined. |
| REX.W + 0F 01 /4 | SMSW *r64/m16* | A | Valid | Valid | Store machine status word in low-order 16 bits of *r64/m16*; high-order 16 bits of *r32* are undefined. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:r/m (w) | NA | NA | NA |

...

## SQRTPD—Compute Square Roots of Packed Double-Precision Floating-Point Values

| Opcode* | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 66 0F 51 /*r* | SQRTPD *xmm1*, *xmm2/m128* | A | Valid | Valid | Computes square roots of the packed double-precision floating-point values in *xmm2/m128* and stores the results in *xmm1*. |

**Instruction Operand Encoding**

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (w) | ModRM:r/m (r) | NA | NA |

...

## SQRTPS—Compute Square Roots of Packed Single-Precision Floating-Point Values

| Opcode* | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F 51 /*r* | SQRTPS *xmm1*, *xmm2/m128* | A | Valid | Valid | Computes square roots of the packed single-precision floating-point values in *xmm2/m128* and stores the results in *xmm1*. |

**Instruction Operand Encoding**

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (w) | ModRM:r/m (r) | NA | NA |

...

## SQRTSD—Compute Square Root of Scalar Double-Precision Floating-Point Value

| Opcode* | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| F2 0F 51 /*r* | SQRTSD *xmm1*, *xmm2/m64* | A | Valid | Valid | Computes square root of the low double-precision floating-point value in *xmm2/m64* and stores the results in *xmm1*. |

**Instruction Operand Encoding**

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (w) | ModRM:r/m (r) | NA | NA |

...

## SQRTSS—Compute Square Root of Scalar Single-Precision Floating-Point Value

| Opcode* | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| F3 0F 51 /*r* | SQRTSS *xmm1*, *xmm2/m32* | A | Valid | Valid | Computes square root of the low single-precision floating-point value in *xmm2/m32* and stores the results in *xmm1*. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (w) | ModRM:r/m (r) | NA | NA |

...

## STC—Set Carry Flag

| Opcode* | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| F9 | STC | A | Valid | Valid | Set CF flag. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | NA | NA | NA | NA |

...

## STD—Set Direction Flag

| Opcode* | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| FD | STD | A | Valid | Valid | Set DF flag. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | NA | NA | NA | NA |

...

## STI—Set Interrupt Flag

| Opcode* | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| FB | STI | A | Valid | Valid | Set interrupt flag; external, maskable interrupts enabled at the end of the next instruction. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | NA | NA | NA | NA |

...

## STMXCSR—Store MXCSR Register State

| Opcode* | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F AE /3 | STMXCSR *m32* | A | Valid | Valid | Store contents of MXCSR register to *m32*. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:r/m (w) | NA | NA | NA |

...

## STOS/STOSB/STOSW/STOSD/STOSQ—Store String

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| AA | STOS *m8* | A | Valid | Valid | For legacy mode, store AL at address ES:(E)DI; For 64-bit mode store AL at address RDI or EDI. |
| AB | STOS *m16* | A | Valid | Valid | For legacy mode, store AX at address ES:(E)DI; For 64-bit mode store AX at address RDI or EDI. |
| AB | STOS *m32* | A | Valid | Valid | For legacy mode, store EAX at address ES:(E)DI; For 64-bit mode store EAX at address RDI or EDI. |
| REX.W + AB | STOS *m64* | A | Valid | N.E. | Store RAX at address RDI or EDI. |

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| AA | STOSB | A | Valid | Valid | For legacy mode, store AL at address ES:(E)DI; For 64-bit mode store AL at address RDI or EDI. |
| AB | STOSW | A | Valid | Valid | For legacy mode, store AX at address ES:(E)DI; For 64-bit mode store AX at address RDI or EDI. |
| AB | STOSD | A | Valid | Valid | For legacy mode, store EAX at address ES:(E)DI; For 64-bit mode store EAX at address RDI or EDI. |
| REX.W + AB | STOSQ | A | Valid | N.E. | Store RAX at address RDI or EDI. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | NA | NA | NA | NA |

...

## STR—Store Task Register

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F 00 /1 | STR *r/m16* | A | Valid | Valid | Stores segment selector from TR in *r/m16*. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:r/m (w) | NA | NA | NA |

...

## SUB—Subtract

| **Opcode** | **Instruction** | Op/ En | **64-Bit Mode** | **Compat/ Leg Mode** | **Description** |
|---|---|---|---|---|---|
| 2C *ib* | SUB AL, *imm8* | A | Valid | Valid | Subtract *imm8* from AL. |
| 2D *iw* | SUB AX, *imm16* | A | Valid | Valid | Subtract *imm16* from AX. |
| 2D *id* | SUB EAX, *imm32* | A | Valid | Valid | Subtract *imm32* from EAX. |
| REX.W + 2D *id* | SUB RAX, *imm32* | A | Valid | N.E. | Subtract *imm32* sign-extended to 64-bits from RAX. |
| 80 /5 *ib* | SUB *r/m8, imm8* | B | Valid | Valid | Subtract *imm8* from *r/m8.* |
| REX + 80 /5 *ib* | SUB *r/m8*, imm8* | B | Valid | N.E. | Subtract *imm8* from *r/m8.* |
| 81 /5 *iw* | SUB *r/m16, imm16* | B | Valid | Valid | Subtract *imm16* from *r/m16.* |
| 81 /5 *id* | SUB *r/m32, imm32* | B | Valid | Valid | Subtract *imm32* from *r/m32.* |
| REX.W + 81 /5 *id* | SUB *r/m64, imm32* | B | Valid | N.E. | Subtract *imm32* sign-extended to 64-bits from *r/m64.* |
| 83 /5 *ib* | SUB *r/m16, imm8* | B | Valid | Valid | Subtract sign-extended *imm8* from *r/m16.* |
| 83 /5 *ib* | SUB *r/m32, imm8* | B | Valid | Valid | Subtract sign-extended *imm8* from *r/m32.* |
| REX.W + 83 /5 *ib* | SUB *r/m64, imm8* | B | Valid | N.E. | Subtract sign-extended *imm8* from *r/m64.* |
| 28 */r* | SUB *r/m8, r8* | C | Valid | Valid | Subtract *r8* from *r/m8.* |
| REX + 28 */r* | SUB *r/m8*, r8** | C | Valid | N.E. | Subtract *r8* from *r/m8.* |
| 29 */r* | SUB *r/m16, r16* | C | Valid | Valid | Subtract *r16* from *r/m16.* |
| 29 */r* | SUB *r/m32, r32* | C | Valid | Valid | Subtract *r32* from *r/m32.* |
| REX.W + 29 */r* | SUB *r/m64, r32* | C | Valid | N.E. | Subtract *r64* from *r/m64.* |
| 2A */r* | SUB *r8, r/m8* | D | Valid | Valid | Subtract *r/m8* from *r8.* |
| REX + 2A */r* | SUB *r8*, r/m8** | D | Valid | N.E. | Subtract *r/m8* from *r8.* |
| 2B */r* | SUB *r16, r/m16* | D | Valid | Valid | Subtract *r/m16* from *r16.* |
| 2B */r* | SUB *r32, r/m32* | D | Valid | Valid | Subtract *r/m32* from *r32.* |
| REX.W + 2B */r* | SUB *r64, r/m64* | D | Valid | N.E. | Subtract *r/m64* from *r64.* |

**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 |
|---|---|---|---|---|
| A | AL/AX/EAX/RAX | imm8/26/32 | NA | NA |
| B | ModRM:r/m (r, w) | imm8/26/32 | NA | NA |
| C | ModRM:r/m (r, w) | ModRM:reg (r) | NA | NA |
| D | ModRM:reg (r, w) | ModRM:r/m (r) | NA | NA |

...

## SUBPD—Subtract Packed Double-Precision Floating-Point Values

| **Opcode** | **Instruction** | Op/ En | **64-Bit Mode** | **Compat/ Leg Mode** | **Description** |
|---|---|---|---|---|---|
| 66 0F 5C /*r* | SUBPD *xmm1*, *xmm2/m128* | A | Valid | Valid | Subtract packed double-precision floating-point values in *xmm2/m128* from *xmm1*. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (r, w) | ModRM:r/m (r) | NA | NA |

...

## SUBPS—Subtract Packed Single-Precision Floating-Point Values

| **Opcode** | **Instruction** | Op/ En | **64-Bit Mode** | **Compat/ Leg Mode** | **Description** |
|---|---|---|---|---|---|
| 0F 5C /*r* | SUBPS *xmm1* *xmm2/m128* | A | Valid | Valid | Subtract packed single-precision floating-point values in *xmm2/mem* from *xmm1*. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (r, w) | ModRM:r/m (r) | NA | NA |

...

## SUBSD—Subtract Scalar Double-Precision Floating-Point Values

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| F2 0F 5C /*r* | SUBSD *xmm1*, *xmm2/m64* | A | Valid | Valid | Subtracts the low double-precision floating-point values in *xmm2/mem64* from *xmm1*. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (r, w) | ModRM:r/m (r) | NA | NA |

...

## SUBSS—Subtract Scalar Single-Precision Floating-Point Values

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| F3 0F 5C /*r* | SUBSS *xmm1*, *xmm2/m32* | A | Valid | Valid | Subtract the lower single-precision floating-point values in *xmm2/m32* from *xmm1*. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (r, w) | ModRM:r/m (r) | NA | NA |

...

## SWAPGS—Swap GS Base Register

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F 01 /7 | SWAPGS | A | Valid | Invalid | Exchanges the current GS base register value with the value contained in MSR address C0000102H. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | NA | NA | NA | NA |

...

## SYSCALL—Fast System Call

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F 05 | SYSCALL | A | Valid | Invalid | Fast call to privilege level 0 system procedures. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | NA | NA | NA | NA |

...

## SYSENTER—Fast System Call

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F 34 | SYSENTER | A | Valid | Valid | Fast call to privilege level 0 system procedures. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | NA | NA | NA | NA |

...

### Operation

```
IF CR0.PE = 0 THEN #GP(0); FI;
IF SYSENTER_CS_MSR[15:2] = 0 THEN #GP(0); FI;
EFLAGS.VM ← 0;                        (* ensures protected mode execution *)
EFLAGS.IF ← 0;                        (* Mask interrupts *)
EFLAGS.RF ← 0;

CS.SEL ← SYSENTER_CS_MSR              (* Operating system provides CS *)
(* Set rest of CS to a fixed value *)
CS.BASE ← 0;                          (* Flat segment *)
CS.LIMIT ← FFFFFH;                    (* 4-GByte limit *)
CS.ARbyte.G ← 1;                      (* 4-KByte granularity *)
CS.ARbyte.S ← 1;
CS.ARbyte.TYPE ← 1011B;               (* Execute + Read, Accessed *)
CS.ARbyte.D ← 1;                  (* 32-bit code segment*)
CS.ARbyte.DPL ← 0;
CS.SEL.RPL ← 0;
CS.ARbyte.P ← 1;
CPL ← 0;

SS.SEL ← CS.SEL + 8;
(* Set rest of SS to a fixed value *)
```

```
SS.BASE ← 0;                        (* Flat segment *)
SS.LIMIT ← FFFFFH;                  (* 4-GByte limit *)
SS.ARbyte.G ← 1;                    (* 4-KByte granularity *)
SS.ARbyte.S ←;
SS.ARbyte.TYPE ← 0011B;             (* Read/Write, Accessed *)
SS.ARbyte.D ← 1;                    (* 32-bit stack segment*)
SS.ARbyte.DPL ← 0;
SS.SEL.RPL ← 0;
SS.ARbyte.P ← 1;

ESP ← SYSENTER_ESP_MSR;
EIP ← SYSENTER_EIP_MSR;
```

...

## SYSEXIT—Fast Return from Fast System Call

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F 35 | SYSEXIT | A | Valid | Valid | Fast return to privilege level 3 user code. |
| REX.W + 0F 35 | SYSEXIT | A | Valid | Valid | Fast return to 64-bit mode privilege level 3 user code. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | NA | NA | NA | NA |

...

## SYSRET—Return From Fast System Call

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F 07 | SYSRET | A | Valid | Invalid | Return from fast system call |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | NA | NA | NA | NA |

...

## TEST—Logical Compare

| **Opcode** | **Instruction** | Op/ En | **64-Bit Mode** | **Compat/ Leg Mode** | **Description** |
|---|---|---|---|---|---|
| A8 *ib* | TEST AL, *imm8* | A | Valid | Valid | AND *imm8* with AL; set SF, ZF, PF according to result. |
| A9 *iw* | TEST AX, *imm16* | A | Valid | Valid | AND *imm16* with AX; set SF, ZF, PF according to result. |
| A9 *id* | TEST EAX, *imm32* | A | Valid | Valid | AND *imm32* with EAX; set SF, ZF, PF according to result. |
| REX.W + A9 *id* | TEST RAX, *imm32* | A | Valid | N.E. | AND *imm32* sign-extended to 64-bits with RAX; set SF, ZF, PF according to result. |
| F6 /0 *ib* | TEST *r/m8, imm8* | B | Valid | Valid | AND *imm8* with *r/m8*; set SF, ZF, PF according to result. |
| REX + F6 /0 *ib* | TEST *r/m8**, *imm8* | B | Valid | N.E. | AND *imm8* with *r/m8*; set SF, ZF, PF according to result. |
| F7 /0 *iw* | TEST *r/m16, imm16* | B | Valid | Valid | AND *imm16* with *r/m16*; set SF, ZF, PF according to result. |
| F7 /0 *id* | TEST *r/m32, imm32* | B | Valid | Valid | AND *imm32* with *r/m32*; set SF, ZF, PF according to result. |
| REX.W + F7 /0 *id* | TEST *r/m64, imm32* | B | Valid | N.E. | AND *imm32* sign-extended to 64-bits with *r/m64*; set SF, ZF, PF according to result. |
| 84 */r* | TEST *r/m8, r8* | C | Valid | Valid | AND *r8* with *r/m8*; set SF, ZF, PF according to result. |
| REX + 84 */r* | TEST *r/m8**, *r8** | C | Valid | N.E. | AND *r8* with *r/m8*; set SF, ZF, PF according to result. |
| 85 */r* | TEST *r/m16, r16* | C | Valid | Valid | AND *r16* with *r/m16*; set SF, ZF, PF according to result. |
| 85 */r* | TEST *r/m32, r32* | C | Valid | Valid | AND *r32* with *r/m32*; set SF, ZF, PF according to result. |
| REX.W + 85 */r* | TEST *r/m64, r64* | C | Valid | N.E. | AND *r64* with *r/m64*; set SF, ZF, PF according to result. |

**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 |
|---|---|---|---|---|
| A | AL/AX/EAX/RAX | imm8/16/32 | NA | NA |
| B | ModRM:r/m (r) | imm8/16/32 | NA | NA |
| C | ModRM:r/m (r) | ModRM:reg (r) | NA | NA |

...

## UCOMISD—Unordered Compare Scalar Double-Precision Floating-Point Values and Set EFLAGS

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 66 0F 2E /*r* | UCOMISD *xmm1*, *xmm2/m64* | A | Valid | Valid | Compares (unordered) the low double-precision floating-point values in *xmm1* and *xmm2/m64* and set the EFLAGS accordingly. |

**Instruction Operand Encoding**

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (r) | ModRM:r/m (r) | NA | NA |

...

## UCOMISS—Unordered Compare Scalar Single-Precision Floating-Point Values and Set EFLAGS

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F 2E /*r* | UCOMISS *xmm1*, *xmm2/m32* | A | Valid | Valid | Compare lower single-precision floating-point value in *xmm1* register with lower single-precision floating-point value in *xmm2/mem* and set the status flags accordingly. |

**Instruction Operand Encoding**

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (r) | ModRM:r/m (r) | NA | NA |

...

## UD2—Undefined Instruction

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F 0B | UD2 | A | Valid | Valid | Raise invalid opcode exception. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | NA | NA | NA | NA |

...

## UNPCKHPD—Unpack and Interleave High Packed Double-Precision Floating-Point Values

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 66 0F 15 /*r* | UNPCKHPD *xmm1, xmm2/m128* | A | Valid | Valid | Unpacks and Interleaves double-precision floating-point values from high quadwords of *xmm1* and *xmm2/m128*. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (r, w) | ModRM:r/m (r) | NA | NA |

...

## UNPCKHPS—Unpack and Interleave High Packed Single-Precision Floating-Point Values

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F 15 /*r* | UNPCKHPS *xmm1, xmm2/m128* | A | Valid | Valid | Unpacks and Interleaves single-precision floating-point values from high quadwords of *xmm1* and *xmm2/mem* into *xmm1*. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (r, w) | ModRM:r/m (r) | NA | NA |

...

## UNPCKLPD—Unpack and Interleave Low Packed Double-Precision Floating-Point Values

| **Opcode** | **Instruction** | **Op/ En** | **64-Bit Mode** | **Compat/ Leg Mode** | **Description** |
|---|---|---|---|---|---|
| 66 0F 14 /*r* | UNPCKLPD *xmm1*, *xmm2/m128* | A | Valid | Valid | Unpacks and Interleaves double-precision floating-point values from low quadwords of *xmm1* and *xmm2/m128*. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (r, w) | ModRM:r/m (r) | NA | NA |

...

## UNPCKLPS—Unpack and Interleave Low Packed Single-Precision Floating-Point Values

| **Opcode** | **Instruction** | **Op/ En** | **64-Bit Mode** | **Compat/ Leg Mode** | **Description** |
|---|---|---|---|---|---|
| 0F 14 /*r* | UNPCKLPS *xmm1*, *xmm2/m128* | A | Valid | Valid | Unpacks and Interleaves single-precision floating-point values from low quadwords of *xmm1* and *xmm2/mem* into *xmm1*. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (r, w) | ModRM:r/m (r) | NA | NA |

...

## VERR/VERW—Verify a Segment for Reading or Writing

| **Opcode** | **Instruction** | **Op/ En** | **64-Bit Mode** | **Compat/ Leg Mode** | **Description** |
|---|---|---|---|---|---|
| 0F 00 /4 | VERR *r/m16* | A | Valid | Valid | Set ZF=1 if segment specified with *r/m16* can be read. |
| 0F 00 /5 | VERW *r/m16* | B | Valid | Valid | Set ZF=1 if segment specified with *r/m16* can be written. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:r/m (r) | NA | NA | NA |
| B | NA | NA | NA | NA |

...

## WAIT/FWAIT—Wait

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 9B | WAIT | A | Valid | Valid | Check pending unmasked floating-point exceptions. |
| 9B | FWAIT | A | Valid | Valid | Check pending unmasked floating-point exceptions. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | NA | NA | NA | NA |

### Description

Causes the processor to check for and handle pending, unmasked, floating-point exceptions before proceeding. (FWAIT is an alternate mnemonic for WAIT.)

This instruction is useful for synchronizing exceptions in critical sections of code. Coding a WAIT instruction after a floating-point instruction ensures that any unmasked floating-point exceptions the instruction may raise are handled before the processor can modify the instruction's results. See the section titled "Floating-Point Exception Synchronization" in Chapter 8 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for more information on using the WAIT/FWAIT instruction.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

...

## WBINVD—Write Back and Invalidate Cache

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F 09 | WBINVD | A | Valid | Valid | Write back and flush Internal caches; initiate writing-back and flushing of external caches. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | NA | NA | NA | NA |

...

## WRMSR—Write to Model Specific Register

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F 30 | WRMSR | A | Valid | Valid | Write the value in EDX:EAX to MSR specified by ECX. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | NA | NA | NA | NA |

...

## XADD—Exchange and Add

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F C0 /*r* | XADD *r/m8, r8* | A | Valid | Valid | Exchange *r8* and *r/m8*; load sum into *r/m8*. |
| REX + 0F C0 /*r* | XADD *r/m8**, *r8** | A | Valid | N.E. | Exchange *r8* and *r/m8*; load sum into *r/m8*. |
| 0F C1 /*r* | XADD *r/m16, r16* | A | Valid | Valid | Exchange *r16* and *r/m16*; load sum into *r/m16*. |
| 0F C1 /*r* | XADD *r/m32, r32* | A | Valid | Valid | Exchange *r32* and *r/m32*; load sum into *r/m32*. |
| REX.W + 0F C1 /*r* | XADD *r/m64, r64* | A | 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 |
|---|---|---|---|---|
| A | ModRM:r/m (r, w) | ModRM:reg (r) | NA | NA |

...

## XCHG—Exchange Register/Memory with Register

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 90+*rw* | XCHG AX, *r16* | A | Valid | Valid | Exchange *r16* with AX. |
| 90+*rw* | XCHG *r16*, AX | B | Valid | Valid | Exchange AX with *r16.* |
| 90+*rd* | XCHG EAX, *r32* | A | Valid | Valid | Exchange *r32* with EAX. |
| REX.W + 90+*rd* | XCHG RAX, *r64* | A | Valid | N.E. | Exchange *r64* with RAX. |
| 90+*rd* | XCHG *r32*, EAX | B | Valid | Valid | Exchange EAX with *r32.* |
| REX.W + 90+*rd* | XCHG *r64*, RAX | B | Valid | N.E. | Exchange RAX with *r64.* |
| 86 /*r* | XCHG *r/m8, r8* | C | Valid | Valid | Exchange *r8* (byte register) with byte from *r/m8.* |
| REX + 86 /*r* | XCHG *r/m8*, r8** | C | Valid | N.E. | Exchange *r8* (byte register) with byte from *r/m8.* |
| 86 /*r* | XCHG *r8, r/m8* | D | Valid | Valid | Exchange byte from *r/m8* with *r8* (byte register). |
| REX + 86 /*r* | XCHG *r8*, r/m8** | D | Valid | N.E. | Exchange byte from *r/m8* with *r8* (byte register). |
| 87 /*r* | XCHG *r/m16, r16* | C | Valid | Valid | Exchange *r16* with word from *r/m16.* |
| 87 /*r* | XCHG *r16, r/m16* | D | Valid | Valid | Exchange word from *r/m16* with *r16.* |
| 87 /*r* | XCHG *r/m32, r32* | C | Valid | Valid | Exchange *r32* with doubleword from *r/m32.* |
| REX.W + 87 /*r* | XCHG *r/m64, r64* | C | Valid | N.E. | Exchange *r64* with quadword from *r/m64.* |
| 87 /*r* | XCHG *r32, r/m32* | D | Valid | Valid | Exchange doubleword from *r/m32* with *r32.* |
| REX.W + 87 /*r* | XCHG *r64, r/m64* | D | Valid | N.E. | Exchange quadword from *r/m64* with *r64.* |

**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 |
|---|---|---|---|---|
| A | AX/EAX/RAX (r, w) | reg (r, w) | NA | NA |
| B | reg (r, w) | AX/EAX/RAX (r, w) | NA | NA |
| C | ModRM:r/m (r, w) | ModRM:reg (r, w) | NA | NA |
| D | ModRM:reg (r, w) | ModRM:r/m (r, w) | NA | NA |

...

## XGETBV—Get Value of Extended Control Register

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F 01 D0 | XGETBV | A | Valid | Valid | Reads an XCR specified by ECX into EDX:EAX. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | NA | NA | NA | NA |

...

## XLAT/XLATB—Table Look-up Translation

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| D7 | XLAT *m8* | A | Valid | Valid | Set AL to memory byte DS:[(E)BX + unsigned AL]. |
| D7 | XLATB | A | Valid | Valid | Set AL to memory byte DS:[(E)BX + unsigned AL]. |
| REX.W + D7 | XLATB | A | Valid | N.E. | Set AL to memory byte [RBX + unsigned AL]. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | NA | NA | NA | NA |

...

## XOR—Logical Exclusive OR

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 34 *ib* | XOR AL, *imm8* | A | Valid | Valid | AL XOR *imm8.* |
| 35 *iw* | XOR AX, *imm16* | A | Valid | Valid | AX XOR *imm16.* |
| 35 *id* | XOR EAX, *imm32* | A | Valid | Valid | EAX XOR *imm32.* |
| REX.W + 35 *id* | XOR RAX, *imm32* | A | Valid | N.E. | RAX XOR *imm32 (sign-extended).* |
| 80 /6 *ib* | XOR *r/m8, imm8* | B | Valid | Valid | *r/m8* XOR *imm8.* |
| REX + 80 /6 *ib* | XOR *r/m8*, imm8* | B | Valid | N.E. | *r/m8* XOR *imm8.* |
| 81 /6 *iw* | XOR *r/m16, imm16* | B | Valid | Valid | *r/m16* XOR *imm16.* |
| 81 /6 *id* | XOR *r/m32, imm32* | B | Valid | Valid | *r/m32* XOR *imm32.* |
| REX.W + 81 /6 *id* | XOR *r/m64, imm32* | B | Valid | N.E. | *r/m64* XOR *imm32 (sign-extended).* |
| 83 /6 *ib* | XOR *r/m16, imm8* | B | Valid | Valid | *r/m16* XOR *imm8 (sign-extended).* |
| 83 /6 *ib* | XOR *r/m32, imm8* | B | Valid | Valid | *r/m32* XOR *imm8 (sign-extended).* |
| REX.W + 83 /6 *ib* | XOR *r/m64, imm8* | B | Valid | N.E. | *r/m64* XOR *imm8 (sign-extended).* |
| 30 /*r* | XOR *r/m8, r8* | C | Valid | Valid | *r/m8* XOR *r8.* |
| REX + 30 /*r* | XOR *r/m8*, r8** | C | Valid | N.E. | *r/m8* XOR *r8.* |
| 31 /*r* | XOR *r/m16, r16* | C | Valid | Valid | *r/m16* XOR *r16.* |
| 31 /*r* | XOR *r/m32, r32* | C | Valid | Valid | *r/m32* XOR *r32.* |
| REX.W + 31 /*r* | XOR *r/m64, r64* | C | Valid | N.E. | *r/m64* XOR *r64.* |
| 32 /*r* | XOR *r8, r/m8* | D | Valid | Valid | *r8* XOR *r/m8.* |
| REX + 32 /*r* | XOR *r8*, r/m8** | D | Valid | N.E. | *r8* XOR *r/m8.* |
| 33 /*r* | XOR *r16, r/m16* | D | Valid | Valid | *r16* XOR *r/m16.* |
| 33 /*r* | XOR *r32, r/m32* | D | Valid | Valid | *r32* XOR *r/m32.* |
| REX.W + 33 /*r* | XOR *r64, r/m64* | D | Valid | N.E. | *r64* XOR *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 |
|---|---|---|---|---|
| A | AL/AX/EAX/RAX | imm8/16/32 | NA | NA |
| B | ModRM:r/m (r, w) | imm8/16/32 | NA | NA |
| C | ModRM:r/m (r, w) | ModRM:reg (r) | NA | NA |
| D | ModRM:reg (r, w) | ModRM:r/m (r) | NA | NA |

...

## XORPD—Bitwise Logical XOR for Double-Precision Floating-Point Values

| **Opcode** | **Instruction** | Op/ En | **64-Bit Mode** | **Compat/ Leg Mode** | **Description** |
|---|---|---|---|---|---|
| 66 0F 57 /*r* | XORPD *xmm1, xmm2/m128* | A | Valid | Valid | Bitwise exclusive-OR of *xmm2/m128* and *xmm1*. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (r, w) | ModRM:r/m (r) | NA | NA |

...

## XORPS—Bitwise Logical XOR for Single-Precision Floating-Point Values

| **Opcode** | **Instruction** | Op/ En | **64-Bit Mode** | **Compat/ Leg Mode** | **Description** |
|---|---|---|---|---|---|
| 0F 57 /*r* | XORPS *xmm1, xmm2/m128* | A | Valid | Valid | Bitwise exclusive-OR of *xmm2/m128* and *xmm1*. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:reg (r, w) | ModRM:r/m (r) | NA | NA |

...

## XRSTOR—Restore Processor Extended States

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F AE /5 | XRSTOR *mem* | A | Valid | Valid | Restore processor extended states from *memory*. The states are specified by EDX:EAX |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:r/m (r) | NA | NA | NA |

...

## XSAVE—Save Processor Extended States

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F AE /4 | XSAVE *mem* | A | Valid | Valid | Save processor extended states to *memory*. The states are specified by EDX:EAX |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | ModRM:r/m (w) | NA | NA | NA |

...

## XSETBV—Set Extended Control Register

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F 01 D1 | XSETBV | A | Valid | Valid | Write the value in EDX:EAX to the XCR specified by ECX. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| A | NA | NA | NA | NA |

...

## 3. Updates to Chapter 4, Volume 3A

Change bars show changes to Chapter 4 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A:* System Programming Guide, Part 1.

------------------------------------------------------------------------------------

...

Table 4-1 illustrates the key differences between the three paging modes.

**Table 4-1 Properties of Different Paging Modes**

| Paging Mode | CR0.PG | CR4.PAE | LME in IA32_EFER | Linear-Address Width | Physical-Address Width[1] | Page Size(s) | Supports Execute-Disable? |
|---|---|---|---|---|---|---|---|
| None | 0 | N/A | N/A | 32 | 32 | N/A | No |
| 32-bit | 1 | 0 | 0[2] | 32 | Up to 40[3] | 4-KByte<br>4-MByte[4] | No |
| PAE | 1 | 1 | 0 | 32 | Up to 52 | 4-KByte<br>2-MByte | Yes[5] |
| IA-32e | 1 | 1 | 2 | 48 | Up to 52 | 4-KByte<br>2-MByte<br>1-GByte[6] | Yes[5] |

**NOTES:**

1. The physical-address width is always bounded by MAXPHYADDR; see Section 4.1.4.
2. The processor ensures that IA32_EFER.LME must be 0 if CR0.PG = 1 and CR4.PAE = 0.
3. 32-bit paging supports physical-address widths of more than 32 bits only for 4-MByte pages and only if the PSE-36 mechanism is supported; see Section 4.1.4 and Section 4.3.
4. 4-MByte pages are used with 32-bit paging only if CR4.PSE = 1; see Section 4.3.
5. Execute-disable access rights are applied only if IA32_EFER.NXE = 1; see Section 4.6.
6. Not all processors that support IA-32e paging support 1-GByte pages; see Section 4.1.4.

Because they are used only if IA32_EFER.LME = 0, 32-bit paging and PAE paging is used only in legacy protected mode. Because legacy protected mode cannot produce

...

### 4.1.4 Enumeration of Paging Features by CPUID

Software can discover support for different paging features using the CPUID instruction:

- PSE: page-size extensions for 32-bit paging.
  If CPUID.01H:EDX.PSE [bit 3] = 1, CR4.PSE may be set to 1, enabling support for 4-MByte pages with 32-bit paging (see Section 4.3).
- PAE: physical-address extension.
  If CPUID.01H:EDX.PAE [bit 6] = 1, CR4.PAE may be set to 1, enabling PAE paging (this setting is also required for IA-32e paging).
- PGE: global-page support.
  If CPUID.01H:EDX.PGE [bit 13] = 1, CR4.PGE may be set to 1, enabling the global-page feature (see Section 4.10.1.4).

- PAT: page-attribute table.
  If CPUID.01H:EDX.PAT [bit 16] = 1, the 8-entry page-attribute table (PAT) is supported. When the PAT is supported, three bits in certain paging-structure entries select a memory type (used to determine type of caching used) from the PAT (see Section 4.9).
- PSE-36: 36-Bit page size extension.
  If CPUID.01H:EDX.PSE-36 [bit 17] = 1, the PSE-36 mechanism is supported, indicating that translations using 4-MByte pages with 32-bit paging may produce physical addresses with more than 32 bits (see Section 4.3).
- NX: execute disable.
  If CPUID.80000001H:EDX.NX [bit 20] = 1, IA32_EFER.NXE may be set to 1, allowing PAE paging and IA-32e paging to disable execute access to selected pages (see Section 4.6). (Processors that do not support CPUID function 80000001H do not allow IA32_EFER.NXE to be set to 1.)
- Page1GB: 1-GByte pages.
  If CPUID.80000001H:EDX.Page1GB [bit 26] = 1, 1-GByte pages are supported with IA-32e paging (see Section 4.5).
- LM: IA-32e mode support.
  If CPUID.80000001H:EDX.LM [bit 29] = 1, IA32_EFER.LME may be set to 1, enabling IA-32e paging. (Processors that do not support CPUID function 80000001H do not allow IA32_EFER.LME to be set to 1.)
- CPUID.80000008H:EAX[7:0] reports the physical-address width supported by the processor. (For processors that do not support CPUID function 80000008H, the width is generally 36 if CPUID.01H:EDX.PAE [bit 6] = 1 and 32 otherwise.) This width is referred to as MAXPHYADDR. MAXPHYADDR is at most 52.
- CPUID.80000008H:EAX[15:8] reports the linear-address width supported by the processor. Generally, this value is 48 if CPUID.80000001H:EDX.LM [bit 29] = 1 and 32 otherwise. (Processors that do not support CPUID function 80000008H, support a linear-address width of 32.)

...

## 4.2 HIERARCHICAL PAGING STRUCTURES: AN OVERVIEW

All three paging modes translate linear addresses use **hierarchical paging structures**. This section provides an overview of their operation. Section 4.3, Section 4.4, and Section 4.5 provide details for the three paging modes.

Every paging structure is 4096 Bytes in size and comprises a number of individual **entries**. With 32-bit paging, each entry is 32 bits (4 bytes); there are thus 1024 entries in each structure. With PAE paging and IA-32e paging, each entry is 64 bits (8 bytes); there are thus 512 entries in each structure. (PAE paging includes one exception, a paging structure that is 32 bytes in size, containing 4 64-bit entries.)

The processor uses the upper portion of a linear address to identify a series of paging-structure entries. The last of these entries identifies the physical address of the region to which the linear address translates (called the **page frame**). The lower portion of the linear address (called the **page offset**) identifies the specific address within that region to which the linear address translates.

Each paging-structure entry contains a physical address, which is either the address of another paging structure or the address of a page frame. In the first case, the entry is said to **reference** the other paging structure; in the latter, the entry is said to **map a page**.

The first paging structure used for any translation is located at the physical address in CR3. A linear address is translated using the following iterative procedure. A portion of the linear address (initially the uppermost bits) select an entry in a paging structure (initially the one located using CR3). If that entry references another paging structure, the process continues with that paging structure and with the portion of the linear address immediately below that just used. If instead the entry maps a page, the process completes: the physical address in the entry is that of the page frame and the remaining lower portion of the linear address is the page offset.

The following items give an example for each of the three paging modes (each example locates a 4-KByte page frame):

- With 32-bit paging, each paging structure comprises 1024 = $2^{10}$ entries. For this reason, the translation process uses 10 bits at a time from a 32-bit linear address. Bits 31:22 identify the first paging-structure entry and bits 21:12 identify a second. The latter identifies the page frame. Bits 11:0 of the linear address are the page offset within the 4-KByte page frame. (See Figure 4-2 for an illustration.)
- With PAE paging, the first paging structure comprises only 4 = $2^2$ entries. Translation thus begins by using bits 31:30 from a 32-bit linear address to identify the first paging-structure entry. Other paging structures comprise 512 =$2^9$ entries, so the process continues by using 9 bits at a time. Bits 29:21 identify a second paging-structure entry and bits 20:12 identify a third. This last identifies the page frame. (See Figure 4-5 for an illustration.)
- With IA-32e paging, each paging structure comprises 512 = $2^9$ entries and translation uses 9 bits at a time from a 48-bit linear address. Bits 47:39 identify the first paging-structure entry, bits 38:30 identify a second, bits 29:21 a third, and bits 20:12 identify a fourth. Again, the last identifies the page frame. (See Figure 4-8 for an illustration.)

The translation process in each of the examples above completes by identifying a page frame. However, the paging structures may be configured so that translation terminates before doing so. This occurs if process encounters a paging-structure entry that is marked “not present” (because its P flag — bit 0 — is clear) or in which a reserved bit is set. In this case, there is no translation for the linear address; an access to that address causes a page-fault exception (see Section 4.7).

In the examples above, a paging-structure entry maps a page with 4-KByte page frame when only 12 bits remain in the linear address; entries identified earlier always reference other paging structures. That may not apply in other cases. The following items identify when an entry maps a page and when it references another paging structure:

- If more than 12 bits remain in the linear address, bit 7 (PS — page size) of the current paging-structure entry is consulted. If the bit is 0, the entry references another paging structure; if the bit is 1, the entry maps a page.
- If only 12 bits remain in the linear address, the current paging-structure entry always maps a page (bit 7 is used for other purposes).

If a paging-structure entry maps a page when more than 12 bits remain in the linear address, the entry identifies a page frame larger than 4 KBytes. For example, 32-bit paging uses the upper 10 bits of a linear address to locate the first paging-structure entry; 22 bits remain. If that entry maps a page, the page frame is $2^{22}$ Bytes = 4 MBytes. 32-bit paging supports 4-MByte pages if CR4.PSE = 1. PAE paging and IA-32e paging support 2-MByte pages (regardless of the value of CR4.PSE). IA-32e paging may support 1-GByte pages (see Section 4.1.4).

Paging structures are given different names based their uses in the translation process. Table 4-2 gives the names of the different paging structures. It also provides, for each structure, the source of the physical address used to locate it (CR3 or a different paging-structure entry); the bits in the linear address used to select an entry from the structure; and details of about whether and how such an entry can map a page.

...

**Table 4-2 Paging Structures in the Different Paging Modes**

| Paging Structure | Entry Name | Paging Mode | Physical Address of Structure | Bits Selecting Entry | Page Mapping |
|---|---|---|---|---|---|
| PML4 table | PML4E | 32-bit, PAE | N/A | | |
| | | IA-32e | CR3 | 47:39 | N/A (PS must be 0) |
| Page-directory-pointer table | PDPTE | 32-bit | N/A | | |
| | | PAE | CR3 | 31:30 | N/A (PS must be 0) |
| | | IA-32e | PML4E | 38:30 | 1-GByte page if PS=1[1] |
| Page directory | PDE | 32-bit | CR3 | 31:22 | 4-MByte page if PS=1[2] |
| | | PAE, IA-32e | PDPTE | 29:21 | 2-MByte page if PS=1 |
| Page table | PTE | 32-bit | PDE | 21:12 | 4-KByte page |
| | | PAE, IA-32e | | 20:12 | 4-KByte page |

**NOTES:**

1. Not all processors allow the PS flag to be 1 in PDPTEs; see Section 4.1.4 for how to determine whether 1-GByte pages are supported.
2. 32-bit paging ignores the PS flag in a PDE (and uses the entry to reference a page table) unless CR4.PSE = 1. Not all processors allow CR4.PSE to be 1; see Section 4.1.4 for how to determine whether 4-MByte pages are supported with 32-bit paging.

...

## 4.4.1 PDPTE Registers

When PAE paging is used, CR3 references the base of a 32-Byte **page-directory-pointer table**. Table 4-8 illustrates how CR3 is used with PAE paging.

**Table 4-8 Use of CR3 with PAE Paging**

| Bit Position(s) | Contents |
|---|---|
| 4:0 | Ignored |
| 31:5 | Physical address of the 32-Byte aligned page-directory-pointer table used for linear-address translation |
| 63:32 | Ignored (these bits exist only on processors supporting the Intel-64 architecture) |

The page-directory-pointer-table comprises four (4) 64-bit entries called PDPTEs. Each PDPTE controls access to a 1-GByte region of the linear-address space. Corresponding to the PDPTEs, the logical processor maintains a set of four (4) internal, non-architectural PDPTE registers, called PDPTE0, PDPTE1, PDPTE2, and PDPTE3. The logical processor loads these registers from the PDPTEs in memory as part of certain executions the MOV to CR instruction:

- If PAE paging would be in use following an execution of MOV to CR0 or MOV to CR4 (see Section 4.1.1) and the instruction is modifying any of CR0.CD, CR0.NW, CR0.PG, CR4.PAE, CR4.PGE, or CR4.PSE; then the PDPTEs are loaded from the address in CR3.
- If MOV to CR3 is executed while the logical processor is using PAE paging, the PDPTEs are loaded from the address being loaded into CR3.
- If PAE paging is in use and a task switch changes the value of CR3, the PDPTEs are loaded from the address in the new CR3 value.
- Certain VMX transitions load the PDPTE registers. See Section 4.11.1.

...

.

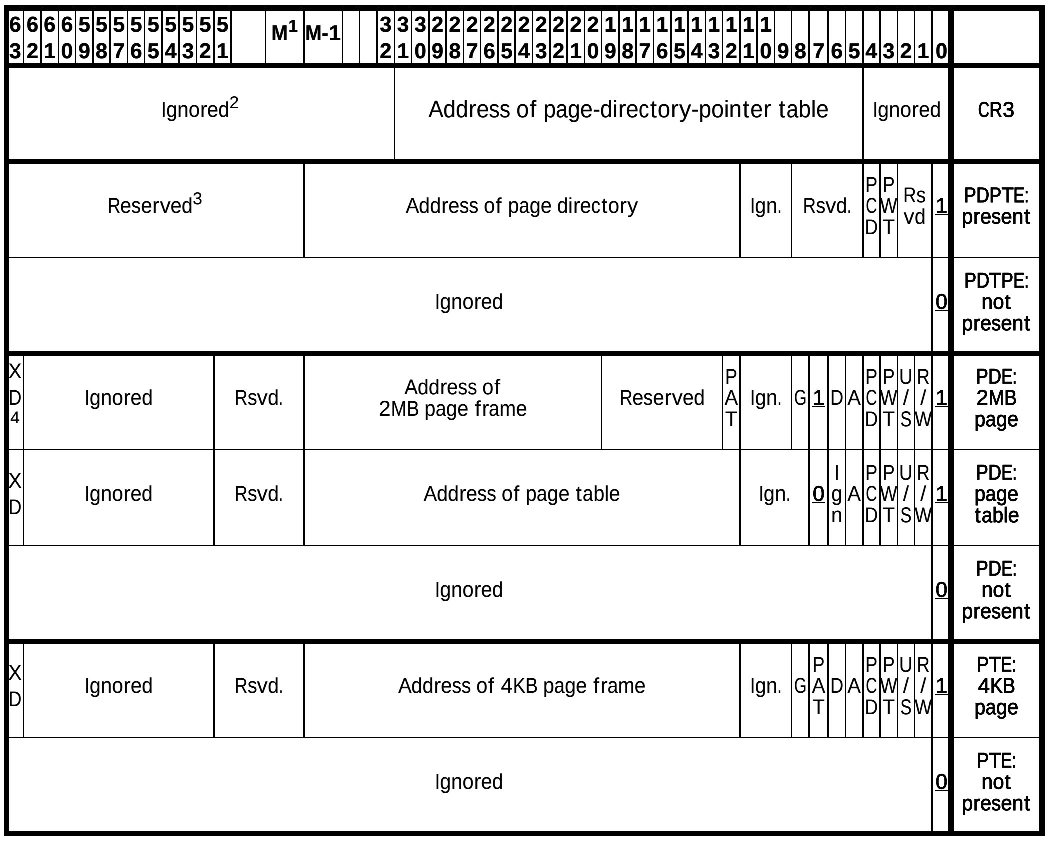


**Figure 4-7. Formats of CR3 and Paging-Structure Entries with PAE Paging**

**NOTES:**

1. M is an abbreviation for MAXPHYADDR.
2. CR3 has 64 bits only on processors supporting the Intel-64 architecture. These bits are ignored with PAE paging.

3. Reserved fields must be 0.
4. If IA32_EFER.NXE = 0 and the P flag of a PDE or a PTE is 1, the XD flag (bit 63) is reserved.

...

**Table 4-8. Format of a PAE Page-Directory-Pointer-Table Entry (PDPTE)**

| Bit Position(s) | Contents |
|---|---|
| 0 (P) | Present; must be 1 to reference a page directory |
| 2:1 | Reserved (must be 0) |
| 3 (PWT) | Page-level write-through; indirectly determines the memory type used to access the page directory referenced by this entry (see Section 4.9) |
| 4 (PCD) | Page-level cache disable; indirectly determines the memory type used to access the page directory referenced by this entry (see Section 4.9) |
| 8:5 | Reserved (must be 0) |
| 11:9 | Ignored |
| (M–1):12 | Physical address of 4-KByte aligned page directory referenced by this entry[1] |
| 63:M | Reserved (must be 0) |

**NOTES:**
1. M is an abbreviation for MAXPHYADDR, which is at most 52; see Section 4.1.4.

...

## 4.5 IA-32E PAGING

A logical processor uses IA-32e paging if CR0.PG = 1, CR4.PAE = 1, and IA32_EFER.LME = 1. With IA-32e paging, linear address are translated using a hierarchy of in-memory paging structures located using the contents of CR3. IA-32e paging translates 48-bit linear addresses to 52-bit physical addresses.[1] Although 52 bits corresponds to 4 PBytes, linear addresses are limited to 48 bits; at most 256 TBytes of linear-address space may be accessed at any given time.

IA-32e paging uses a hierarchy of paging structures to produce a translation for a linear address. CR3 is used to locate the first paging-structure, the PML4 table. Table 4-12 illustrates how CR3 is used with IA-32e paging.

**Table 4-12 Use of CR3 with IA-32e Paging**

| Bit Position(s) | Contents |
|---|---|
| 2:0 | Ignored |
| 3 (PWT) | Page-level write-through; indirectly determines the memory type used to access the PML4 table during linear-address translation (see Section 4.9) |
| 4 (PCD) | Page-level cache disable; indirectly determines the memory type used to access the PML4 table during linear-address translation (see Section 4.9) |
| 11:5 | Ignored |
| (M–1):12 | Physical address of the 4-KByte aligned PML4 table used for linear-address translation[1] |
| 63:M | Reserved (must be 0) |

**NOTES:**
1. M is an abbreviation for MAXPHYADDR, which is at most 52; see Section 4.1.4.

IA-32e paging may map linear addresses to 4-KByte pages, 2-MByte pages, or 1-GByte pages.[2] Figure 4-8 illustrates the translation process when it produces a 4-KByte page; Figure 4-9 covers the case of a 2-MByte page, and Figure 4-10 the case of a 1-GByte page. The following items describe the IA-32e paging process in more detail as well has how the page size is determined:

- A 4-KByte naturally aligned page-directory-pointer table is located at the physical address specified in bits 51:12 of the PML4E (see Table 4-13). A page-directory-pointer table comprises 512 64-bit entries (PDPTEs). A PDPTE is selected using the physical address defined as follows:
  - — Bits 51:12 are from the PML4E.
  - — Bits 11:3 are bits 38:30 of the linear address.
  - — Bits 2:0 are all 0.

...

1. If MAXPHYADDR < 52, bits in the range 51:MAXPHYADDR will be 0 in any physical address used by IA-32e paging. (The corresponding bits are reserved in the paging-structure entries.) See Section 4.1.4 for how to determine MAXPHYADDR.
2. Not all processors support 1-GByte pages; see Section 4.1.4.

Because a PDPTE is identified using bits 47:30 of the linear address, it controls access to a 1-GByte region of the linear-address space. Use of the PDPTE depends on its PS flag (bit 7):[1]

...

- If the PDPTE's PS flag is 1, the PDPTE maps a 1-GByte page (see Table 4-14). The final physical address is computed as follows:

**Table 4-14 Format of an IA-32e Page-Directory-Pointer-Table Entry (PDPTE) that Maps a 1-GByte Page**

| Bit Position(s) | Contents |
|---|---|
| 0 (P) | Present; must be 1 to map a 1-GByte page |
| 1 (R/W) | Read/write; if 0, writes may not be allowed to the 1-GByte page referenced by this entry (depends on CPL and CR0.WP; see Section 4.6) |
| 2 (U/S) | User/supervisor; if 0, accesses with CPL=3 are not allowed to the 1-GByte page referenced by this entry (see Section 4.6) |
| 3 (PWT) | Page-level write-through; indirectly determines the memory type used to access the 1-GByte page referenced by this entry (see Section 4.9) |
| 4 (PCD) | Page-level cache disable; indirectly determines the memory type used to access the 1-GByte page referenced by this entry (see Section 4.9) |
| 5 (A) | Accessed; indicates whether software has accessed the 1-GByte page referenced by this entry (see Section 4.8) |
| 6 (D) | Dirty; indicates whether software has written to the 1-GByte page referenced by this entry (see Section 4.8) |
| 7 (PS) | Page size; must be 1 (otherwise, this entry references a page directory; see Table Table 4-15.) |
| 8 (G) | Global; if CR4.PGE = 1, determines whether the translation is global (see Section 4.10); ignored otherwise |
| 11:9 | Ignored |
| 12 (PAT) | Indirectly determines the memory type used to access the 1-GByte page referenced by this entry (see Section 4.9)[1] |
| 29:13 | Reserved (must be 0) |
| (M–1):30 | Physical address of the 1-GByte page referenced by this entry |
| 51:M | Reserved (must be 0) |
| 62:52 | Ignored |

1. The PS flag of a PDPTE is reserved and must be 0 (if the P flag is 1) if 1-GByte pages are not supported. See Section 4.1.4 for how to determine whether 1-GByte pages are supported.

### Table 4-14 Format of an IA-32e Page-Directory-Pointer-Table Entry (PDPTE) that Maps a 1-GByte Page (Continued)

| Bit Position(s) | Contents |
|---|---|
| 63 (XD) | If IA32_EFER.NXE = 1, execute-disable (if 1, instruction fetches are not allowed from the 1-GByte page controlled by this entry; see Section 4.6); otherwise, reserved (must be 0) |

**NOTES:**
1. The PAT is supported on all processors that support IA-32e paging.

  — Bits 51:30 are from the PDPTE.
  — Bits 29:0 are from the original linear address.

- If the PDE's PS flag is 0, a 4-KByte naturally aligned page directory is located at the physical address specified in bits 51:12 of the PDPTE (see Table 4-15). A page directory comprises 512 64-bit entries (PDEs). A PDE is selected using the physical address defined as follows:

### Table 4-15 Format of an IA-32e Page-Directory-Pointer-Table Entry (PDPTE) that References a Page Directory

| Bit Position(s) | Contents |
|---|---|
| 0 (P) | Present; must be 1 to reference a page directory |
| 1 (R/W) | Read/write; if 0, writes may not be allowed to the 1-GByte region controlled by this entry (depends on CPL and CR0.WP; see Section 4.6) |
| 2 (U/S) | User/supervisor; if 0, accesses with CPL=3 are not allowed to the 1-GByte region controlled by this entry (see Section 4.6) |
| 3 (PWT) | Page-level write-through; indirectly determines the memory type used to access the page directory referenced by this entry (see Section 4.9) |
| 4 (PCD) | Page-level cache disable; indirectly determines the memory type used to access the page directory referenced by this entry (see Section 4.9) |
| 5 (A) | Accessed; indicates whether this entry has been used for linear-address translation (see Section 4.8) |
| 6 | Ignored |
| 7 (PS) | Page size; must be 0 (otherwise, this entry maps a 1-GByte page; see Table 4-14) |
| 11:8 | Ignored |
| (M–1):12 | Physical address of 4-KByte aligned page directory referenced by this entry |
| 51:M | Reserved (must be 0) |
| 62:52 | Ignored |
| 63 (XD) | If IA32_EFER.NXE = 1, execute-disable (if 1, instruction fetches are not allowed from the 1-GByte region controlled by this entry; see Section 4.6); otherwise, reserved (must be 0) |

- — Bits 51:12 are from the PDPTE.
- — Bits 11:3 are bits 29:21 of the linear address.
- — Bits 2:0 are all 0

...

If a paging-structure entry's P flag (bit 0) is 0 or if the entry sets any reserved bit, the entry is used neither to reference another paging-structure entry nor to map a page. A reference using a linear address whose translation would use such a paging-structure entry causes a page-fault exception (see Section 4.7).

The following bits are reserved with IA-32e paging:

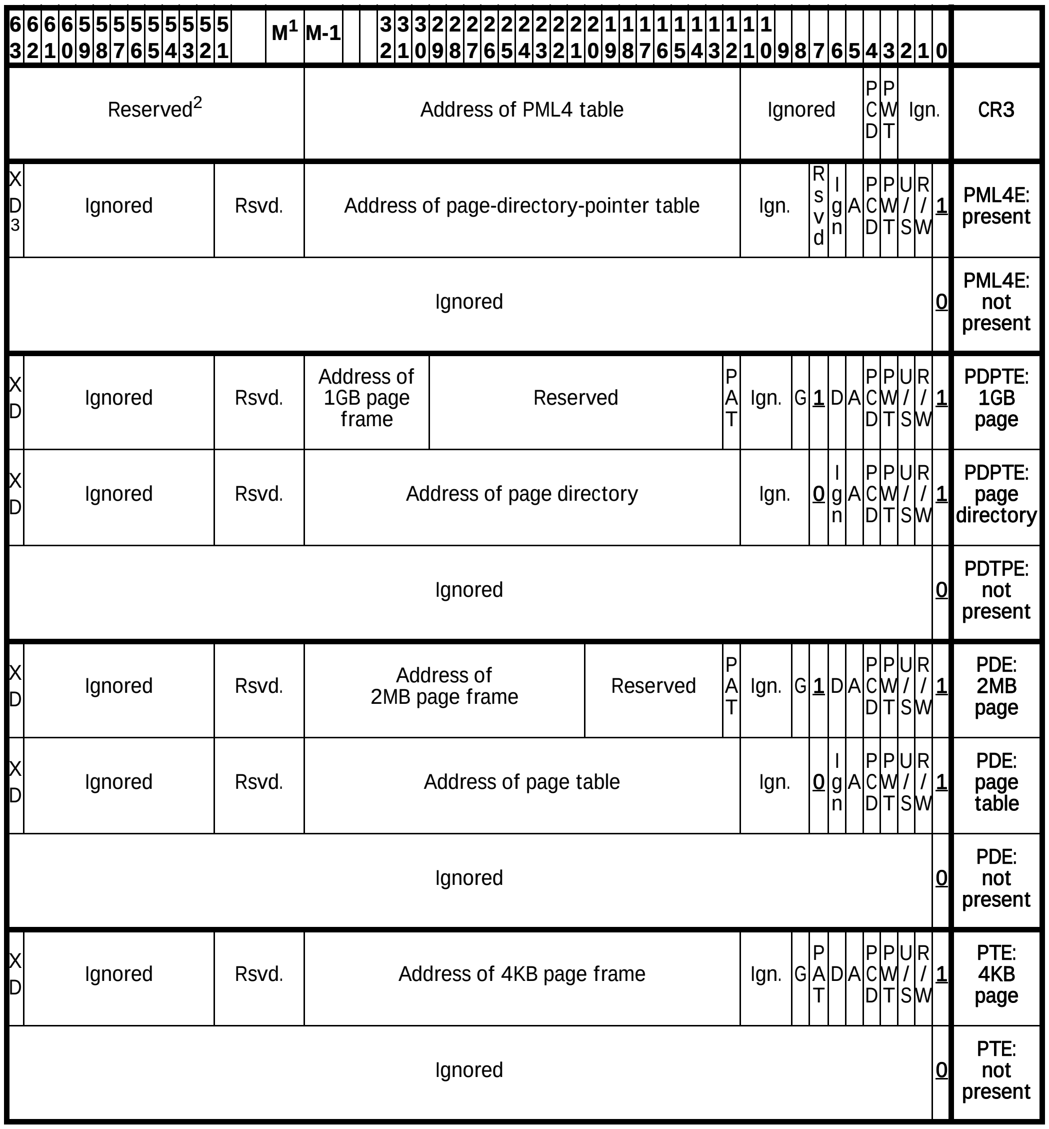
- If the P flag of a paging-structure entry is 1, bits 51:MAXPHYADDR are reserved.
- If the P flag of a PML4E is 1, the PS flag is reserved.
- If 1-GByte pages are not supported and the P flag of a PDPTE is 1, the PS flag is reserved.[1]
- If the P flag and the PS flag of a PDPTE are both 1, bits 29:13 are reserved.
- If the P flag and the PS flag of a PDE are both 1, bits 20:13 are reserved.

If IA32_EFER.NXE = 0 and the P flag of a paging-structure entry is 1, the XD flag (bit 63) is reserved.

...

1. See Section 4.1.4 for how to determine whether 1-GByte pages are supported.

## Figure 4-11. Formats of CR3 and Paging-Structure Entries with IA-32e Paging


6 6 6 5 5 5 5 5 5 5 5 5 5 / 3 2 1 0 9 8 7 6 5 4 3 2 1
M1 M-1
3 3 3 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1 / 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
Reserved2 | Address of PML4 table | Ignored | PCD | PWT | Ign. | CR3
XD3 | Ignored | Rsvd. | Address of page-directory-pointer table | Ign. | Rsvd | Ign | A | PCD | PWT | U/S | R/W | 1 | PML4E: present
Ignored | 0 | PML4E: not present
XD | Ignored | Rsvd. | Address of 1GB page frame | Reserved | PAT | Ign. | G | 1 | D | A | PCD | PWT | U/S | R/W | 1 | PDPTE: 1GB page
XD | Ignored | Rsvd. | Address of page directory | Ign. | 0 | Ign | A | PCD | PWT | U/S | R/W | 1 | PDPTE: page directory
Ignored | 0 | PDTPE: not present
XD | Ignored | Rsvd. | Address of 2MB page frame | Reserved | PAT | Ign. | G | 1 | D | A | PCD | PWT | U/S | R/W | 1 | PDE: 2MB page
XD | Ignored | Rsvd. | Address of page table | Ign. | 0 | Ign | A | PCD | PWT | U/S | R/W | 1 | PDE: page table
Ignored | 0 | PDE: not present
XD | Ignored | Rsvd. | Address of 4KB page frame | Ign. | G | PAT | D | A | PCD | PWT | U/S | R/W | 1 | PTE: 4KB page
Ignored | 0 | PTE: not present

**NOTES:**

1. M is an abbreviation for MAXPHYADDR.
2. Reserved fields must be 0.
3. If IA32_EFER.NXE = 0 and the P flag of a paging-structure entry is 1, the XD flag (bit 63) is reserved.

...

## 4.7 PAGE-FAULT EXCEPTIONS

Accesses using linear addresses may cause **page-fault exceptions** (#PF; exception 14). An access to a linear address may cause page-fault exception for either of two reasons: (1) there is no valid translation for the linear address; or (2) there is a valid translation for the linear address, but its access rights do not permit the access.

As noted in Section 4.3, Section 4.4.2, and Section 4.5, there is no valid translation for a linear address if the translation process for that address would use a paging-structure entry in which the P flag (bit 0) is 0 or one that sets a reserved bit. If there is a valid translation for a linear address, its access rights are determined as specified in Section 4.6.

Figure 4-12 illustrates the error code that the processor provides on delivery of a page-fault exception. The following items explain how the bits in the error code describe the nature of the page-fault exception:

- P flag (bit 0).
  This flag is 0 if there is no valid translation for the linear address because the P flag was 0 in one of the paging-structure entries used to translate that address.
- W/R (bit 1).
  If the access causing the page-fault exception was a write, this flag is 1; otherwise, it is 0. This flag describes the access causing the page-fault exception, not the access rights specified by paging.
- U/S (bit 2).
  If a user-mode (CPL= 3) access caused the page-fault exception, this flag is 1; it is 0 if a supervisor-mode (CPL < 3) access did so. This bit describes the access causing the page-fault exception, not the access rights specified by paging.

...

## 4.8 ACCESSED AND DIRTY FLAGS

For any paging-structure entry that is used during linear-address translation, bit 5 is the **accessed** flag. For paging-structure entries that map a page (as opposed to referencing another paging structure), bit 6 is the **dirty** flag. These flags are provided for use by memory-management software to manage the transfer of pages and paging structures into and out of physical memory.

Whenever the processor uses a paging-structure entry as part of linear-address translation, it sets the accessed flag in that entry (if it is not already set).

Whenever there is a write to a linear address, the processor sets the dirty flag (if it is not already set) in the paging-structure entry that identifies the final physical address for the linear address (either a PTE or a paging-structure entry in which the PS flag is 1).

...

### 4.9.2 Paging and Memory Typing When the PAT is Supported (Pentium III and More Recent Processor Families)

If the PAT is supported, paging contributes to memory typing in conjunction with the PAT and the memory-type range registers (MTRRs) as specified in Table 11-7 in Section 11.5.2.2.

The PAT is a 64-bit MSR (IA32_PAT; MSR index 277H) comprising eight (8) 8-bit entries (entry *i* comprises bits 8*i*+7:8*i* of the MSR).

For any access to a physical address, the table combines the memory type specified for that physical address by the MTRRs with a memory type selected from the PAT. Table 11-11 in Section 11.12.3 specifies how a memory type is selected from the PAT. Specifically, it comes from entry *i* of the PAT, where *i* is defined as follows:

- For an access to an entry in a paging structure whose address is in CR3 (e.g., the PML4 table with IA-32e paging), *i* = 2*PCD+PWT, where the PCD and PWT values come from CR3.
- For an access to a PDE with PAE paging, *i* = 2*PCD+PWT, where the PCD and PWT values come from the relevant PDPTE register.
- For an access to a paging-structure entry X whose address is in another paging-structure entry Y, *i* = 2*PCD+PWT, where the PCD and PWT values come from Y.
- For an access to the physical address that is the translation of a linear address, *i* = 4*PAT+2*PCD+PWT, where the PAT, PCD, and PWT values come from the relevant PTE (if the translation uses a 4-KByte page), the relevant PDE (if the translation uses a 2-MByte page or a 4-MByte page), or the relevant PDPTE (if the translation uses a 1-GByte page).

...

#### 4.10.1.1 Page Numbers, Page Frames, and Page Offsets

Section 4.3, Section 4.4.2, and Section 4.5 give details of how the different paging modes translate linear addresses to physical addresses. Specifically, the upper bits of a linear address (called the **page number**) determine the upper bits of the physical address (called the **page frame**); the lower bits of the linear address (called the **page offset**) determine the lower bits of the physical address. The boundary between the page number and the page offset is determined by the **page size**. Specifically:

- 32-bit paging:
  - — If the translation does not use a PTE (because CR4.PSE = 1 and the PS flag is 1 in the PDE used), the page size is 4 MBytes and the page number comprises bits 31:22 of the linear address.
  - — If the translation does use a PTE, the page size is 4 KBytes and the page number comprises bits 31:12 of the linear address.
- PAE paging:
  - — If the translation does not use a PTE (because the PS flag is 1 in the PDE used), the page size is 2 MBytes and the page number comprises bits 31:21 of the linear address.
  - — If the translation does uses a PTE, the page size is 4 KBytes and the page number comprises bits 31:12 of the linear address.
- IA-32e paging:
  - — If the translation does not use a PDE (because the PS flag is 1 in the PDPTE used), the page size is 1 GBytes and the page number comprises bits 47:30 of the linear address.
  - — If the translation does use a PDE but does not uses a PTE (because the PS flag is 1 in the PDE used), the page size is 2 MBytes and the page number comprises bits 47:21 of the linear address.

— If the translation does use a PTE, the page size is 4 KBytes and the page number comprises bits 47:12 of the linear address.

...

#### 4.10.1.2 Caching Translations in TLBs

The processor may accelerate the paging process by caching individual translations in **translation lookaside buffers** (**TLBs**). Each entry in a TLB is an individual translation. Each translation is referenced by a page number. It contains the following information from the paging-structure entries used to translate linear addresses with the page number:

- The physical address corresponding to the page number (the page frame).
- The access rights from the paging-structure entries used to translate linear addresses with the page number (see Section 4.6):
  - — The logical-AND of the R/W flags.
  - — The logical-AND of the U/S flags.
  - — The logical-OR of the XD flags (necessary only if IA32_EFER.NXE = 1).
- Attributes from a paging-structure entry that identifies the final page frame for the page number (either a PTE or a paging-structure entry in which the PS flag is 1):
  - — The dirty flag (see Section 4.8).
  - — The memory type (see Section 4.9).

...

#### 4.10.1.3 Details of TLB Use

Because the TLBs cache only valid translations, there can be a TLB entry for a page number only if the P flag is 1 and the reserved bits are 0 in each of the paging-structure entries used to translate that page number. In addition, the processor does not cache a translation for a page number unless the accessed flag is 1 in each of the paging-structure entries used during translation; before caching a translation, the processor sets any of these accessed flags that is not already 1.

The processor may cache translations required for prefetches and for accesses that are a result of speculative execution that would never actually occur in the executed code path.

If the page number of a linear address corresponds to a TLB entry, the processor may use that TLB entry to determine the page frame, access rights, and other attributes for accesses to that linear address. In this case, the processor may not actually consult the paging structures in memory. The processor may retain a TLB entry unmodified even if software subsequently modifies the relevant paging-structure entries in memory. See Section 4.10.3.2 for how software can ensure that the processor uses the modified paging-structure entries.

If the paging structures specify a translation using a page larger than 4 KBytes, some processors may choose to cache multiple smaller-page TLB entries for that translation. Each such TLB entry would be associated with a page number corresponding to the smaller page size (e.g., bits 47:12 of a linear address with IA-32e paging), even though part of that page number (e.g., bits 20:12) are part of the offset with respect to the page specified by the paging structures. The upper bits of the physical address in such a TLB entry are derived from the physical address in the PDE used to create the translation, while the lower bits come from the linear address of the access for which the translation is created. There is no way for software to be aware that multiple translations for smaller pages have been used for a large page.

If software modifies the paging structures so that the page size used for a 4-KByte range of linear addresses changes, the TLBs may subsequently contain multiple translations for the address range (one for each page size). A reference to a linear address in the address range may use any of these translations. Which translation is used may vary from one execution to another, and the choice may be implementation-specific.

#### 4.10.1.4 Global Pages

The Intel-64 and IA-32 architectures also allow for **global pages** when the PGE flag (bit 7) is 1 in CR4. If the G flag (bit 8) is 1 in a paging-structure entry that maps a page (either a PTE or a paging-structure entry in which the PS flag is 1), any TLB entry cached for a linear address using that paging-structure entry is considered to be **global**. Because the G flag is used only in paging-structure entries that map a page, and because information from such entries are not cached in the paging-structure caches, the global-page feature does not affect the behavior of the paging-structure caches.

...

#### 4.10.2.1 Caches for Paging Structures

A processor may support any or of all the following paging-structure caches:

- **PML4 cache** (IA-32e paging only). Each PML4-cache entry is referenced by a 9-bit value and is used for linear addresses for which bits 47:39 have that value. The entry contains information from the PML4E used to translate such linear addresses:
  - — The physical address from the PML4E (the address of the page-directory-pointer table).
  - — The value of the R/W flag of the PML4E.
  - — The value of the U/S flag of the PML4E.
  - — The value of the XD flag of the PML4E.
  - — The values of the PCD and PWT flags of the PML4E.

  The following items detail how a processor may use the PML4 cache:
  - — If the processor has a PML4-cache entry for a linear address, it may use that entry when translating the linear address (instead of the PML4E in memory).
  - — The processor does not create a PML4-cache entry unless the P flag is 1 and all reserved bits are 0 in the PML4E in memory.
  - — The processor does not create a PML4-cache entry unless the accessed flag is 1 in the PML4E in memory; before caching a translation, the processor sets the accessed flag if it is not already 1.
  - — The processor may create a PML4-cache entry even if there are no translations for any linear address that might use that entry (e.g., because the P flags are 0 in all entries in the referenced page-directory-pointer table).
  - — If the processor creates a PML4-cache entry, the processor may retain it unmodified even if software subsequently modifies the corresponding PML4E in memory.

- **PDPTE cache** (IA-32e paging only).[1] Each PDPTE-cache entry is referenced by an 18-bit value and is used for linear addresses for which bits 47:30 have that value. The entry contains information from the PML4E and PDPTE used to translate such linear addresses:
  - — The physical address from the PDPTE (the address of the page directory). (No PDPTE-cache entry is created for a PDPTE that maps a 1-GByte page.)
  - — The logical-AND of the R/W flags in the PML4E and the PDPTE.
  - — The logical-AND of the U/S flags in the PML4E and the PDPTE.
  - — The logical-OR of the XD flags in the PML4E and the PDPTE.
  - — The values of the PCD and PWT flags of the PDPTE.

  The following items detail how a processor may use the PDPTE cache:
  - — If the processor has a PDPTE-cache entry for a linear address, it may use that entry when translating the linear address (instead of the PML4E and the PDPTE in memory).
  - — The processor does not create a PDPTE-cache entry unless the P flag is 1, the PS flag is 0, and the reserved bits are 0 in the PML4E and the PDPTE in memory.

...

#### 4.10.3.2 Recommended Invalidation

The following items provide some recommendations regarding when software should perform invalidations:

- If software modifies a paging-structure entry that identifies the final page frame for a page number (either a PTE or a paging-structure entry in which the PS flag is 1), it should execute INVLPG for any linear address with a page number whose translation uses that PTE.[2] (If the paging-structure entry may be used in the translation of different page numbers — see Section 4.10.2.3 — software should execute INVLPG for linear addresses with each of those page numbers; alternatively, it could use MOV to CR3 or MOV to CR4.)
- If software modifies a paging-structure entry that references another paging structure, it may use one of the following approaches depending upon the types and number of translations controlled by the modified entry:
  - — Execute INVLPG for linear addresses with each of the page numbers with translations that would use the entry. However, if no page numbers that would use the entry have translations (e.g., because the P flags are 0 in all entries in the paging structure referenced by the modified entry), it remains necessary to execute INVLPG at least once.
  - — Execute MOV to CR3 if the modified entry controls no global pages.
  - — Execute MOV to CR4 to modify CR4.PGE.
- If software using PAE paging modifies a PDPTE, it should reload CR3 with the register's current value to ensure that the modified PDPTE is loaded into the corresponding PDPTE register (see Section 4.4.1).

1. With PAE paging, the PDPTEs are stored in internal, non-architectural registers. The operation of these registers is described in Section 4.4.1 and differs from that described here.
2. One execution of INVLPG is sufficient even for a page with size greater than 4 KBytes.

- If the nature of the paging structures is such that a single entry may be used for multiple purposes (see Section 4.10.2.3), software should perform invalidations for all of these purposes. For example, if a single entry might serve as both a PDE and PTE, it may be necessary to execute INVLPG with two (or more) linear addresses, one that uses the entry as a PDE and one that uses it as a PTE. (Alternatively, software could use MOV to CR3 or MOV to CR4.)
- As noted in Section 4.10.1, the TLBs may subsequently contain multiple translations for the address range if software modifies the paging structures so that the page size used for a 4-KByte range of linear addresses changes. A reference to a linear address in the address range may use any of these translations.

  Software wishing to prevent this uncertainty should not write to a paging-structure entry in a way that would change, for any linear address, both the page size and either the page frame, access rights, or other attributes. It can instead use the following algorithm: first clear the P flag in the relevant paging-structure entry (e.g., PDE); then invalidate any translations for the affected linear addresses (see Section 4.10.3.2); and then modify the relevant paging-structure entry to set the P flag and establish modified translation(s) for the new page size.

...

### 4.10.3.3 Optional Invalidation

The following items describe cases in which software may choose not to invalidate and the potential consequences of that choice:

- If a paging-structure entry is modified to change the P flag from 0 to 1, no invalidation is necessary. This is because no TLB entry or paging-structure cache entry is created with information from a paging-structure entry in which the P flag is 0.[1]
- If a paging-structure entry is modified to change the accessed flag from 0 to 1, no invalidation is necessary (assuming that an invalidation was performed the last time the accessed flag was changed from 1 to 0). This is because no TLB entry or paging-structure cache entry is created with information from a paging-structure entry in which the accessed flag is 0.
- If a paging-structure entry is modified to change the R/W flag from 0 to 1, failure to perform an invalidation may result in a “spurious” page-fault exception (e.g., in response to an attempted write access) but no other adverse behavior. Such an exception will occur at most once for each affected linear address (see Section 4.10.3.1).
- If a paging-structure entry is modified to change the U/S flag from 0 to 1, failure to perform an invalidation may result in a “spurious” page-fault exception (e.g., in response to an attempted user-mode access) but no other adverse behavior. Such an exception will occur at most once for each affected linear address (see Section 4.10.3.1).
- If a paging-structure entry is modified to change the XD flag from 1 to 0, failure to perform an invalidation may result in a “spurious” page-fault exception (e.g., in response to an attempted instruction fetch) but no other adverse behavior. Such an exception will occur at most once for each affected linear address (see Section 4.10.3.1).

1. If it is also the case that no invalidation was performed the last time the P flag was changed from 1 to 0, the processor may use a TLB entry or paging-structure cache entry that was created when the P flag had earlier been 1.

- If a paging-structure entry is modified to change the accessed flag from 1 to 0, failure to perform an invalidation may result in the processor not setting that bit in response to a subsequent access to a linear address whose translation uses the entry. Software cannot interpret the bit being clear as an indication that such an access has not occurred.
- If software modifies a paging-structure entry that identifies the final physical address for a linear address (either a PTE or a paging-structure entry in which the PS flag is 1) to change the dirty flag from 1 to 0, failure to perform an invalidation may result in the processor not setting that bit in response to a subsequent write to a linear address whose translation uses the entry. Software cannot interpret the bit being clear as an indication that such a write has not occurred.
- The read of a paging-structure entry in translating an address being used to fetch an instruction may appear to execute before an earlier write to that paging-structure entry if there is no serializing instruction between the write and the instruction fetch. Note that the invalidating instructions identified in Section 4.10.3.1 are all serializing instructions.
- Section 4.10.2.3 describes situations in which a single paging-structure entry may contain information cached in multiple entries in the paging-structure caches. Because all entries in these caches are invalidated by any execution of INVLPG, it is not necessary to follow the modification of such a paging-structure entry by executing INVLPG multiple times solely for the purpose of invalidating these multiple cached entries. (It may be necessary to do so to invalidate multiple TLB entries.)

#### 4.10.3.4 Delayed Invalidation

Required invalidations may be delayed under some circumstances. Software developers should understand that, between the modification of a paging-structure entry and execution of the invalidation instruction recommended in Section 4.10.3.2, the processor may use translations based on either the old value or the new value of the paging-structure entry. The following items describe some of the potential consequences of delayed invalidation:

- If a paging-structure entry is modified to change from 1 to 0 the P flag from 1 to 0, an access to a linear address whose translation is controlled by this entry may or may not cause a page-fault exception.
- If a paging-structure entry is modified to change the R/W flag from 0 to 1, write accesses to linear addresses whose translation is controlled by this entry may or may not cause a page-fault exception.
- If a paging-structure entry is modified to change the U/S flag from 0 to 1, user-mode accesses to linear addresses whose translation is controlled by this entry may or may not cause a page-fault exception.
- If a paging-structure entry is modified to change the XD flag from 1 to 0, instruction fetches from linear addresses whose translation is controlled by this entry may or may not cause a page-fault exception.

As noted in Section 8.1.1, an x87 instruction or an SSE instruction that accesses data larger than a quadword may be implemented using multiple memory accesses. If such an instruction stores to memory and invalidation has been delayed, some of the accesses may complete (writing to memory) while another causes a page-fault exception.[1] In this case, the effects of the completed accesses may be visible to software even though the overall instruction caused a fault.

1. If the accesses are to different pages, this may occur even if invalidation has not been delayed.

In some cases, the consequences of delayed invalidation may not affect software adversely. For example, when freeing a portion of the linear-address space (by marking paging-structure entries “not present”), invalidation using INVLPG may be delayed if software does not re-allocate that portion of the linear-address space or the memory that had been associated with it. However, because of speculative execution (or errant software), there may be accesses to the freed portion of the linear-address space before the invalidations occur. In this case, the following can happen:

- Reads can occur to the freed portion of the linear-address space. Therefore, invalidation should not be delayed for an address range that has read side effects.
- The processor may retain entries in the TLBs and paging-structure caches for an extended period of time. Software should not assume that the processor will not use entries associated with a linear address simply because time has passed.
- As noted in Section 4.10.2.1, the processor may create an entry in a paging-structure cache even if there are no translations for any linear address that might use that entry. Thus, if software has marked “not present” all entries in page table, the processor may subsequently create a PDE-cache entry for the PDE that references that page table (assuming that the PDE itself is marked “present”).
- If software attempts to write to the freed portion of the linear-address space, the processor might not generate a page fault. (Such an attempt would likely be the result of a software error.) For that reason, the page frames previously associated with the freed portion of the linear-address space should not be reallocated for another purpose until the appropriate invalidations have been performed.

...

## 4. Updates to Chapter 5, Volume 3A

Change bars show changes to Chapter 5 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A:* System Programming Guide, Part 1.

------------------------------------------------------------------------------------------

...

## 5.3 LIMIT CHECKING

The limit field of a segment descriptor prevents programs or procedures from addressing memory locations outside the segment. The effective value of the limit depends on the setting of the G (granularity) flag (see Figure 5-1). For data segments, the limit also depends on the E (expansion direction) flag and the B (default stack pointer size and/or upper bound) flag. The E flag is one of the bits in the type field when the segment descriptor is for a data-segment type.

When the G flag is clear (byte granularity), the effective limit is the value of the 20-bit limit field in the segment descriptor. Here, the limit ranges from 0 to FFFFFH (1 MByte). When the G flag is set (4-KByte page granularity), the processor scales the value in the limit field by a factor of $2^{12}$ (4 KBytes). In this case, the effective limit ranges from FFFH (4 KBytes) to FFFFFFFFH (4 GBytes). Note that when scaling is used (G flag is set), the lower 12 bits of a segment offset (address) are not checked against the limit; for example, note that if the segment limit is 0, offsets 0 through FFFH are still valid.

For all types of segments except expand-down data segments, the effective limit is the last address that is allowed to be accessed in the segment, which is one less than the size, in bytes, of the segment. The processor causes a general-protection exception (or, if the segment is SS, a stack-fault exception) any time an attempt is made to access the following addresses in a segment:

- A byte at an offset greater than the effective limit
- A word at an offset greater than the (effective-limit – 1)
- A doubleword at an offset greater than the (effective-limit – 3)
- A quadword at an offset greater than the (effective-limit – 7)
- A double quadword at an offset greater than the (effective limit – 15)

When the effective limit is FFFFFFFFH (4 GBytes), these accesses may or may not cause the indicated exceptions. Behavior is implementation-specific and may vary from one execution to another.

...

### 5.8.8 Fast System Calls in 64-bit Mode

The SYSCALL and SYSRET instructions are designed for operating systems that use a flat memory model (segmentation is not used). The instructions, along with SYSENTER and SYSEXIT, are suited for IA-32e mode operation. SYSCALL and SYSRET, however, are not supported in compatibility mode. Use CPUID to check if SYSCALL and SYSRET are available (CPUID.80000001H.EDX[bit 11] = 1).

SYSCALL is intended for use by user code running at privilege level 3 to access operating system or executive procedures running at privilege level 0. SYSRET is intended for use by privilege level 0 operating system or executive procedures for fast returns to privilege level 3 user code.

Stack pointers for SYSCALL/SYSRET are not specified through model specific registers. The clearing of bits in RFLAGS is programmable rather than fixed. SYSCALL/SYSRET save and restore the RFLAGS register.

For SYSCALL, the processor saves RFLAGS into R11 and the RIP of the next instruction into RCX; it then gets the privilege-level 0 target instruction and stack pointer from:

- **Target code segment** — Reads a non-NULL selector from IA32_STAR[47:32].
- **Target instruction** — Reads a 64-bit canonical address from IA32_LSTAR.
- **Stack segment** — Computed by adding 8 to the value in IA32_STAR[47:32].
- **System flags** — The processor sets RFLAGS to the logical-AND of its current value with the complement of the value in the IA32_FMASK MSR.

...

## 5. 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.

------------------------------------------------------------------------------------------

...

## 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.

...

### 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 bus widths, cache lines, and page boundaries are not guaranteed to be atomic by the Intel Core 2 Duo, Intel 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 4.10.3.4), such page faults may occur even if all accesses are to the same page.

...

### 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.

...

### 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 recommend 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.

...

### 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 it's 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.

...

## 8.2.1 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 HyperThreading 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:
  - — writes executed with the CLFLUSH instruction;
  - — 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).
- 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 cannot pass earlier LFENCE, SFENCE, and MFENCE instructions.

- LFENCE instructions cannot pass earlier reads.
- SFENCE instructions cannot pass earlier writes.
- MFENCE instructions cannot pass earlier reads or writes.

...

### 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 [ _x] | mov r1, [ _z]<br>mov r2, [ _y] |
| Initially on processor 0: EAX == 1, ECX==128, ES:EDI ==_x<br>Initially [_x] to 511[_x]== 0, _x <= _y < _z < _x+512<br>r1 == 1 and r2 == 0 is allowed | |

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.

...

## 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.[1]
- **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

1. 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.

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.

## 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[1]), MOV (to debug register), WBINVD, and WRMSR.
- **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:[2]

...

### 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 Intel Xeon Processors”), 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.

...

### 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. MOV CR8 is not defined architecturally as a serializing instruction.
2. 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.

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.

...

### 8.7.11 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.

Operating system microcode update drivers that adhere to Intel’s guidelines do not need to be modified to run on processors supporting Intel Hyper-Threading Technology.

...

## 6. Updates to Chapter 10, Volume 3A

Change bars show changes to Chapter 10 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A:* System Programming Guide, Part 1.

------------------------------------------------------------------------------------------

...

## 10.3 THE INTEL® 82489DX EXTERNAL APIC, THE APIC, THE XAPIC, AND THE X2APIC

The local APIC in the P6 family and Pentium processors is an architectural subset of the Intel® 82489DX external APIC. See Section 19.27.1, "Software Visible Differences Between the Local APIC and the 82489DX."

The APIC architecture used in the Pentium 4 and Intel Xeon processors (called the xAPIC architecture) is an extension of the APIC architecture found in the P6 family processors. The primary difference between the APIC and xAPIC architectures is that with the xAPIC architecture, the local APICs and the I/O APIC communicate through the system bus. With the APIC architecture, they communication through the APIC bus (see Section 10.2, "System Bus Vs. APIC Bus"). Also, some APIC architectural features have been extended and/or modified in the xAPIC architecture. These extensions and modifications are described in Section 10.4 through Section 10.10.

The x2APIC architecture is an extension of the xAPIC architecture, primarily to increase processor addressability. The x2APIC architecture provides backward compatibility to the xAPIC architecture and forward extendability for future Intel platform innovations. These extensions and modifications are supported by a new mode of execution (**x2APIC mode**) are detailed in Section 10.12.

...

### 10.4.1 The Local APIC Block Diagram

Figure 10-4 gives a functional block diagram for the local APIC. Software interacts with the local APIC by reading and writing its registers. APIC registers are memory-mapped to a 4-KByte region of the processor's physical address space with an initial starting address of FEE00000H. For correct APIC operation, this address space must be mapped to an area of memory that has been designated as strong uncacheable (UC). See Section 11.3, "Methods of Caching Available."

In MP system configurations, the APIC registers for Intel 64 or IA-32 processors on the system bus are initially mapped to the same 4-KByte region of the physical address space. Software has the option of changing initial mapping to a different 4-KByte region for all the local APICs or of mapping the APIC registers for each local APIC to its own 4-KByte region. Section 10.4.5, "Relocating the Local APIC Registers," describes how to relocate the base address for APIC registers.

On processors supporting x2APIC architecture (indicated by CPUID.01H:ECX[21] = 1), the local APIC supports operation in the xAPIC mode (as described in Section 10.4. Additionally, software can enable the local APIC to operate in x2APIC mode for extended processor addressability (see Section 10.12).

...

### NOTE

In processors based on Intel Microarchitecture (Nehalem) the Local APIC ID Register is no longer Read/Write; it is Read Only.

### Table 10-1 Local APIC Register Address Map

| Address | Register Name | Software Read/Write |
|---|---|---|
| FEE0 0000H | Reserved | |
| FEE0 0010H | Reserved | |
| FEE0 0020H | Local APIC ID Register | Read/Write. |
| FEE0 0030H | Local APIC Version Register | Read Only. |
| FEE0 0040H | Reserved | |
| FEE0 0050H | Reserved | |
| FEE0 0060H | Reserved | |
| FEE0 0070H | Reserved | |
| FEE0 0080H | Task Priority Register (TPR) | Read/Write. |
| FEE0 0090H | Arbitration Priority Register[1] (APR) | Read Only. |
| FEE0 00A0H | Processor Priority Register (PPR) | Read Only. |
| FEE0 00B0H | EOI Register | Write Only. |
| FEE0 00C0H | Remote Read Register[1] (RRD) | Read Only |
| FEE0 00D0H | Logical Destination Register | Read/Write. |
| FEE0 00E0H | Destination Format Register | Read/Write (see Section 10.6.2.2). |
| FEE0 00F0H | Spurious Interrupt Vector Register | Read/Write (see Section 10.9. |
| FEE0 0100H | In-Service Register (ISR); bits 31:0 | Read Only. |
| FEE0 0110H | In-Service Register (ISR); bits 63:32 | Read Only. |
| FEE0 0120H | In-Service Register (ISR); bits 95:64 | Read Only. |
| FEE0 0130H | In-Service Register (ISR); bits 127:96 | Read Only. |
| FEE0 0140H | In-Service Register (ISR); bits 159:128 | Read Only. |
| FEE0 0150H | In-Service Register (ISR); bits 191:160 | Read Only. |
| FEE0 0160H | In-Service Register (ISR); bits 223:192 | Read Only. |
| FEE0 0170H | In-Service Register (ISR); bits 255:224 | Read Only. |
| FEE0 0180H | Trigger Mode Register (TMR); bits 31:0 | Read Only. |
| FEE0 0190H | Trigger Mode Register (TMR); bits 63:32 | Read Only. |
| FEE0 01A0H | Trigger Mode Register (TMR); bits 95:64 | Read Only. |
| FEE0 01B0H | Trigger Mode Register (TMR); bits 127:96 | Read Only. |
| FEE0 01C0H | Trigger Mode Register (TMR); bits 159:128 | Read Only. |
| FEE0 01D0H | Trigger Mode Register (TMR); bits 191:160 | Read Only. |
| FEE0 01E0H | Trigger Mode Register (TMR); bits 223:192 | Read Only. |

**Table 10-1 Local APIC Register Address Map (Continued)**

| Address | Register Name | Software Read/Write |
|---|---|---|
| FEE0 01F0H | Trigger Mode Register (TMR); bits 255:224 | Read Only. |
| FEE0 0200H | Interrupt Request Register (IRR); bits 31:0 | Read Only. |
| FEE0 0210H | Interrupt Request Register (IRR); bits 63:32 | Read Only. |
| FEE0 0220H | Interrupt Request Register (IRR); bits 95:64 | Read Only. |
| FEE0 0230H | Interrupt Request Register (IRR); bits 127:96 | Read Only. |
| FEE0 0240H | Interrupt Request Register (IRR); bits 159:128 | Read Only. |
| FEE0 0250H | Interrupt Request Register (IRR); bits 191:160 | Read Only. |
| FEE0 0260H | Interrupt Request Register (IRR); bits 223:192 | Read Only. |
| FEE0 0270H | Interrupt Request Register (IRR); bits 255:224 | Read Only. |
| FEE0 0280H | Error Status Register | Read Only. |
| FEE0 0290H through FEE0 02E0H | Reserved | |
| FEE0 02F0H | LVT CMCI Registers | Read/Write. |
| FEE0 0300H | Interrupt Command Register (ICR); bits 0-31 | Read/Write. |
| FEE0 0310H | Interrupt Command Register (ICR); bits 32-63 | Read/Write. |
| FEE0 0320H | LVT Timer Register | Read/Write. |
| FEE0 0330H | LVT Thermal Sensor Register[2] | Read/Write. |
| FEE0 0340H | LVT Performance Monitoring Counters Register[3] | Read/Write. |
| FEE0 0350H | LVT LINT0 Register | Read/Write. |
| FEE0 0360H | LVT LINT1 Register | Read/Write. |
| FEE0 0370H | LVT Error Register | Read/Write. |
| FEE0 0380H | Initial Count Register (for Timer) | Read/Write. |
| FEE0 0390H | Current Count Register (for Timer) | Read Only. |
| FEE0 03A0H through FEE0 03D0H | Reserved | |
| FEE0 03E0H | Divide Configuration Register (for Timer) | Read/Write. |
| FEE0 03F0H | Reserved | |

**NOTES:**

1. Not supported in the Pentium 4 and Intel Xeon processors. The Illegal Register Access bit (7) of the ESR will not be set when writing to these registers.
2. Introduced in the Pentium 4 and Intel Xeon processors. This APIC register and its associated function are implementation dependent and may not be present in future IA-32 or Intel 64 processors.
3. Introduced in the Pentium Pro processor. This APIC register and its associated function are implementation dependent and may not be present in future IA-32 or Intel 64 processors.

...

**Suppress EOI-broadcasts**
Indicates whether software can inhibit the broadcast of EOI message by setting bit 12 of the Spurious Interrupt Vector Register; see Section 10.8.5 and Section 10.9.

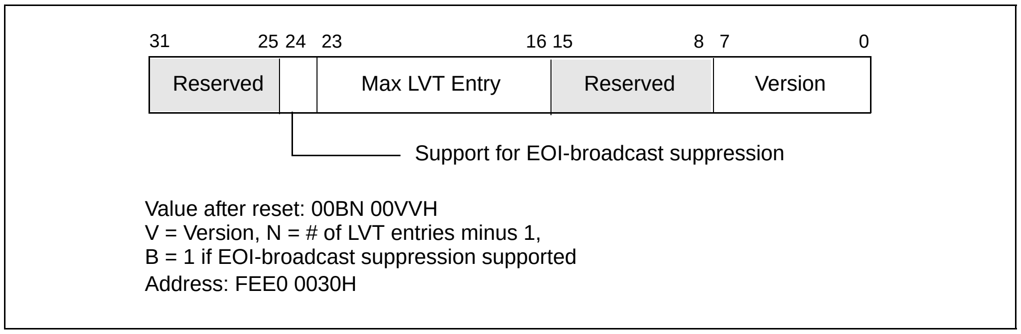

31
25 24
23
16 15
8 7
0
Reserved
Max LVT Entry
Reserved
Version
Support for EOI-broadcast suppression
Value after reset: 00BN 00VVH
V = Version, N = # of LVT entries minus 1,
B = 1 if EOI-broadcast suppression supported
Address: FEE0 0030H

**Figure 10-7. Local APIC Version Register**

...

## 10.5.1 Local Vector Table

The local vector table (LVT) allows software to specify the manner in which the local interrupts are delivered to the processor core. It consists of the following 32-bit APIC registers (see Figure 10-8), one for each local interrupt:

- **LVT Timer Register (FEE0 0320H)** — Specifies interrupt delivery when the APIC timer signals an interrupt (see Section 10.5.4, “APIC Timer”).
- **LVT Thermal Monitor Register (FEE0 0330H)** — Specifies interrupt delivery when the thermal sensor generates an interrupt (see Section 14.5.2, “Thermal Monitor”). This LVT entry is implementation specific, not architectural. If implemented, it will always be at base address FEE0 0330H.
- **LVT Performance Counter Register (FEE0 0340H)** — Specifies interrupt delivery when a performance counter generates an interrupt on overflow (see Section 30.8.5.8, “Generating an Interrupt on Overflow”). This LVT entry is implementation specific, not architectural. If implemented, it is not guaranteed to be at base address FEE0 0340H.
- **LVT LINT0 Register (FEE0 0350H)** — Specifies interrupt delivery when an interrupt is signaled at the LINT0 pin.
- **LVT LINT1 Register (FEE0 0360H)** — Specifies interrupt delivery when an interrupt is signaled at the LINT1 pin.
- **LVT Error Register (FEE0 0370H)** — Specifies interrupt delivery when the APIC detects an internal error (see Section 10.5.3, “Error Handling”).
- **CMCI LVT Register (FEE0 02F0H)** — Specifies interrupt delivery when an overflow condition of corrected machine check error count reaching a threshold value occurred in a machine check bank supporting CMCI (see Section 15.5.1, “CMCI Local APIC Interface”).

The LVT performance counter register and its associated interrupt were introduced in the P6 processors and are also present in the Pentium 4 and Intel Xeon processors. The LVT thermal monitor register and its associated interrupt were introduced in the Pentium 4 and Intel Xeon processors.

As shown in Figure 10-8, some of these fields and flags are not available (and reserved) for some entries.

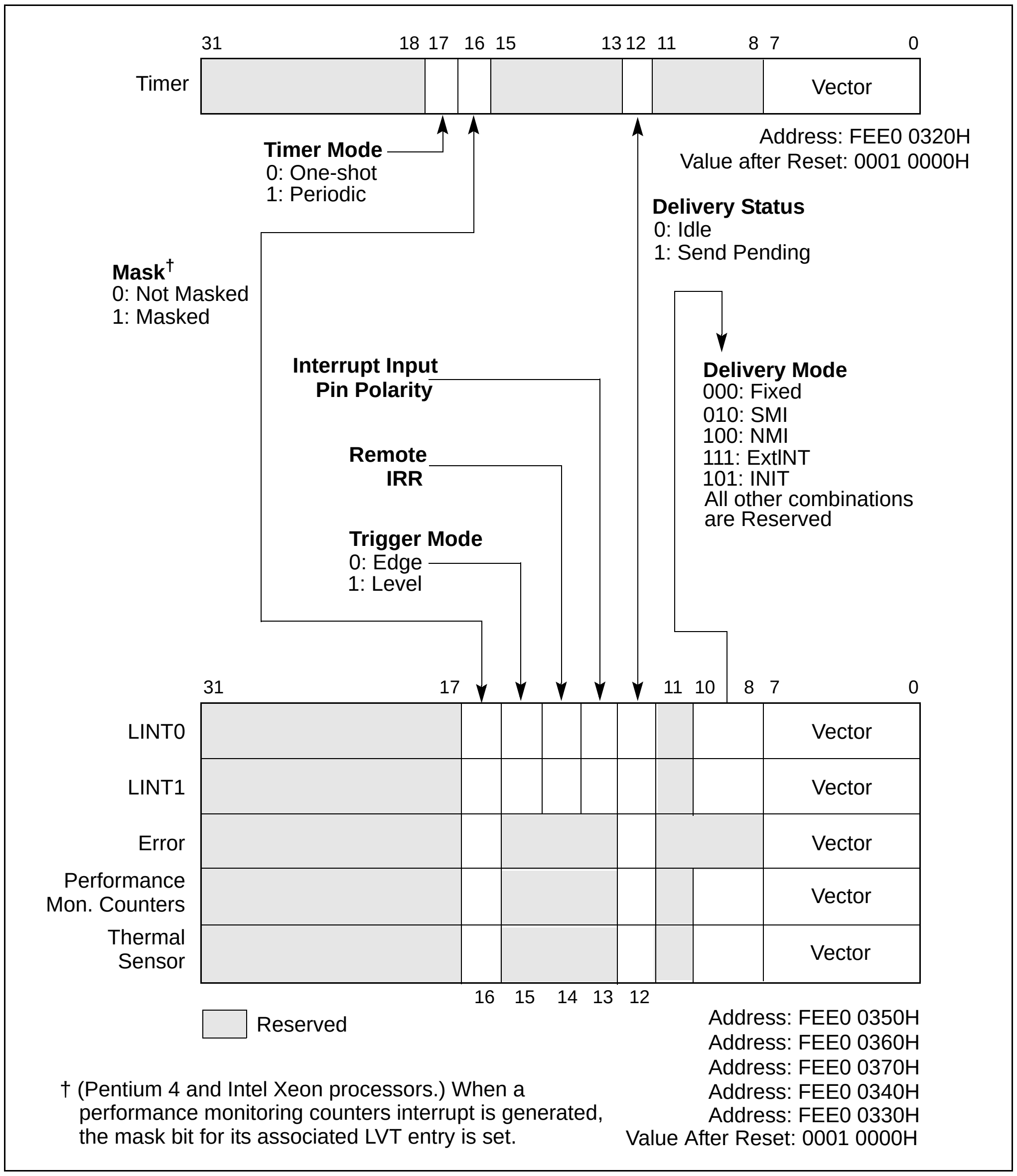


**Figure 10-8 Local Vector Table (LVT)**

...

## 10.5.3 Error Handling

The local APIC provides an error status register (ESR) that it uses to record errors that it detects when handling interrupts (see Figure 10-9). An APIC error interrupt is generated when the local APIC sets one of the error bits in the ESR. The LVT error register allows selection of the interrupt vector to be delivered to the processor core when APIC error is detected. The LVT error register also provides a means of masking an APIC error interrupt.

The ESR is a write/read register. A write (of any value) to the ESR must be done to update the register before attempting to read it. This write clears any previously logged errors and updates the ESR with any errors detected since the last write to the ESR. Errors are collected regardless of LVT Error mask bit, but the APIC will only issue an interrupt due to the error if the LVT Error mask bit is cleared.

The functions of the ESR are listed in Table 10-2.

Error handling in x2APIC mode is discussed in Section 10.12.8.

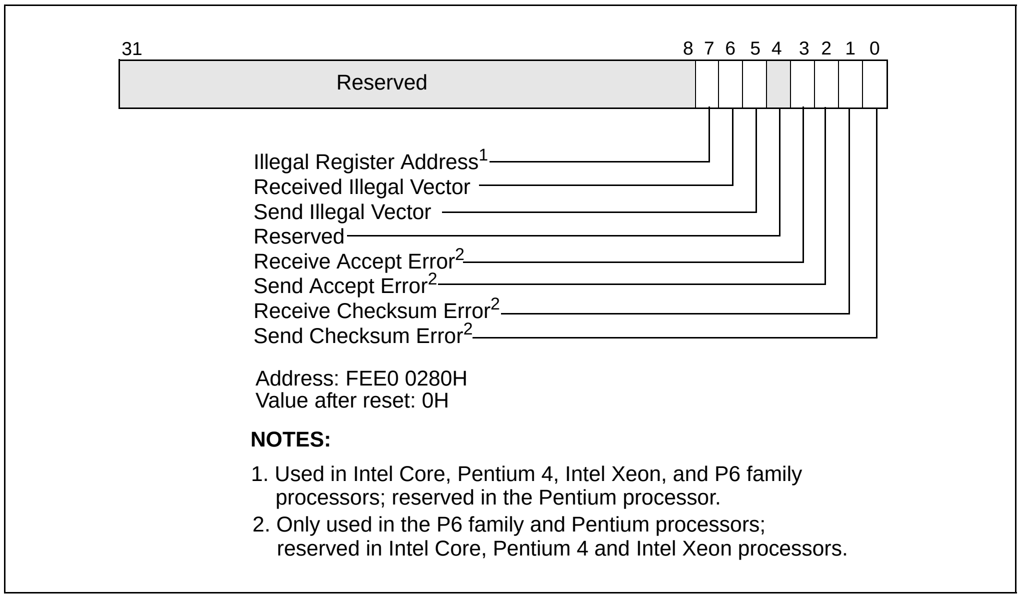

31
8 7 6 5 4 3 2 1 0
Reserved
Illegal Register Address1
Received Illegal Vector
Send Illegal Vector
Reserved
Receive Accept Error2
Send Accept Error2
Receive Checksum Error2
Send Checksum Error2
Address: FEE0 0280H
Value after reset: 0H
NOTES:
1. Used in Intel Core, Pentium 4, Intel Xeon, and P6 family processors; reserved in the Pentium processor.
2. Only used in the P6 family and Pentium processors; reserved in Intel Core, Pentium 4 and Intel Xeon processors.

**Figure 10-9 Error Status Register (ESR)**

**Table 10-2. ESR Flags**

| FLAG | Function |
|---|---|
| Send Checksum Error | (P6 family and Pentium processors only) Set when the local APIC detects a checksum error for a message that it sent on the APIC bus. |
| Receive Checksum Error | (P6 family and Pentium processors only) Set when the local APIC detects a checksum error for a message that it received on the APIC bus. |
| Send Accept Error | (P6 family and Pentium processors only) Set when the local APIC detects that a message it sent was not accepted by any APIC on the APIC bus. |
| Receive Accept Error | (P6 family and Pentium processors only) Set when the local APIC detects that the message it received was not accepted by any APIC on the APIC bus, including itself. |

Table 10-2. ESR Flags

| FLAG | Function |
|---|---|
| Send Checksum Error | (P6 family and Pentium processors only) Set when the local APIC detects a checksum error for a message that it sent on the APIC bus. |
| Receive Checksum Error | (P6 family and Pentium processors only) Set when the local APIC detects a checksum error for a message that it received on the APIC bus. |
| Send Illegal Vector | Set when the local APIC detects an illegal vector in the message that it is sending. |
| Receive Illegal Vector | Set when the local APIC detects an illegal vector in the message it received, including an illegal vector code in the local vector table interrupts or in a self-interrupt. |
| Illegal Reg. Address | (Intel Core, Intel Atom, Pentium 4, Intel Xeon, and P6 family processors only) Set when the processor is trying to access a register in the processor's local APIC register address space that is reserved (see Table 10-1). Addresses in one of the 0x10 byte regions marked reserved are illegal register addresses. The Local APIC Register Map is the address range of the APIC register base address (specified in the IA32_APIC_BASE MSR) plus 4 KBytes. |

...

### 10.5.4 APIC Timer

The local APIC unit contains a 32-bit programmable timer that is available to software to time events or operations. This timer is set up by programming four registers: the divide configuration register (see Figure 10-10), the initial-count and current-count registers (see Figure 10-11), and the LVT timer register (see Figure 10-8).

If CPUID.06H:EAX.ARAT[bit 2] = 1, the processor’s APIC timer runs at a constant rate regardless of P-state transitions and it continues to run at the same rate in deep C-states.

If CPUID.06H:EAX.ARAT[bit 2] = 0 or if CPUID 06H is not supported, the APIC timer may temporarily stop while the processor is in deep C-states or during transitions caused by Enhanced Intel SpeedStep® Technology.

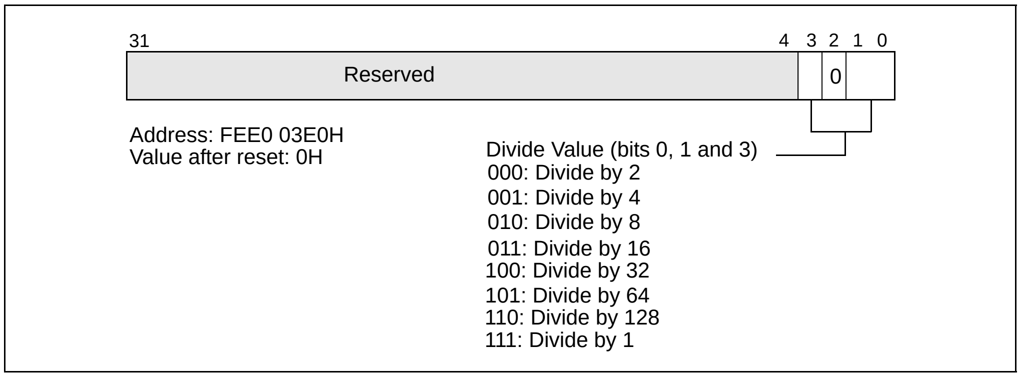


Figure 10-10 Divide Configuration Register

...

...

### 10.6.1 Interrupt Command Register (ICR)

The interrupt command register (ICR) is a 64-bit local APIC register (see Figure 10-12) that allows software running on the processor to specify and send interprocessor interrupts (IPIs) to other processors in the system.

To send an IPI, software must set up the ICR to indicate the type of IPI message to be sent and the destination processor or processors. (All fields of the ICR are read-write by software with the exception of the delivery status field, which is read-only.) The act of writing to the low doubleword of the ICR causes the IPI to be sent.

...

**Delivery Status (Read Only)**

Indicates the IPI delivery status, as follows:

**0 (Idle)** Indicates that this local APIC has completed sending any previous IPIs.

**1 (Send Pending)** Indicates that this local APIC has not completed sending the last IPI.

...

**Destination** Specifies the target processor or processors. This field is only used when the destination shorthand field is set to 00B. If the destination mode is set to physical, then bits 56 through 59 contain the APIC ID of the target processor for Pentium and P6 family processors and bits 56 through 63 contain the APIC ID of the target processor the for Pentium 4 and Intel Xeon processors. If the destination mode is set to logical, the interpretation of the 8-bit destination field depends on the settings of the DFR and LDR registers of the local APICs in all the processors in the system (see Section 10.6.2, “Determining IPI Destination”).

Not all combinations of options for the ICR are valid. Table 10-3 shows the valid combinations for the fields in the ICR for the Pentium 4 and Intel Xeon processors; Table 10-4 shows the valid combinations for the fields in the ICR for the P6 family processors. Also note that the lower half of the ICR may not be preserved over transitions to the deepest C-States.

ICR operation in x2APIC mode is discussed in Section 10.12.9.

...

### 10.6.2 Determining IPI Destination

The destination of an IPI can be one, all, or a subset (group) of the processors on the system bus. The sender of the IPI specifies the destination of an IPI with the following APIC registers and fields within the registers:

- **ICR Register** — The following fields in the ICR register are used to specify the destination of an IPI:

- **Destination Mode** — Selects one of two destination modes (physical or logical).
- **Destination Field** — In physical destination mode, used to specify the APIC ID of the destination processor; in logical destination mode, used to specify a message destination address (MDA) that can be used to select specific processors in clusters.
- **Destination Shorthand** — A quick method of specifying all processors, all excluding self, or self as the destination.
- **Delivery mode, Lowest Priority** — Architecturally specifies that a lowest-priority arbitration mechanism be used to select a destination processor from a specified group of processors. The ability of a processor to send a lowest priority IPI is model specific and should be avoided by BIOS and operating system software.

• **Local destination register (LDR)** — Used in conjunction with the logical destination mode and MDAs to select the destination processors.

• **Destination format register (DFR)** — Used in conjunction with the logical destination mode and MDAs to select the destination processors.

How the ICR, LDR, and DFR are used to select an IPI destination depends on the destination mode used: physical, logical, broadcast/self, or lowest-priority delivery mode. These destination modes are described in the following sections.

Determination of IPI destinations in x2APIC mode is discussed in Section 10.12.10.

...

**NOTES**

All processors that have their APIC software enabled (using the spurious vector enable/disable bit) must have their DFRs (Destination Format Registers) programmed identically.

The default mode for DFR is flat mode. If you are using cluster mode, DFRs must be programmed before the APIC is software enabled. Since some chipsets do not accurately track a system view of the logical mode, program DFRs as soon as possible after starting the processor.

#### 10.6.2.3 Broadcast/Self Delivery Mode

The destination shorthand field of the ICR allows the delivery mode to be by-passed in favor of broadcasting the IPI to all the processors on the system bus and/or back to itself (see Section 10.6.1, "Interrupt Command Register (ICR)"). Three destination shorthands are supported: self, all excluding self, and all including self. The destination mode is ignored when a destination shorthand is used.

...

## 10.8.5 Signaling Interrupt Servicing Completion

For all interrupts except those delivered with the NMI, SMI, INIT, ExtINT, the start-up, or INIT-Deassert delivery mode, the interrupt handler must include a write to the end-of-interrupt (EOI) register (see Figure 10-21). This write must occur at the end of the handler routine, sometime before the IRET instruction. This action indicates that the servicing of the current interrupt is complete and the local APIC can issue the next interrupt from the ISR.

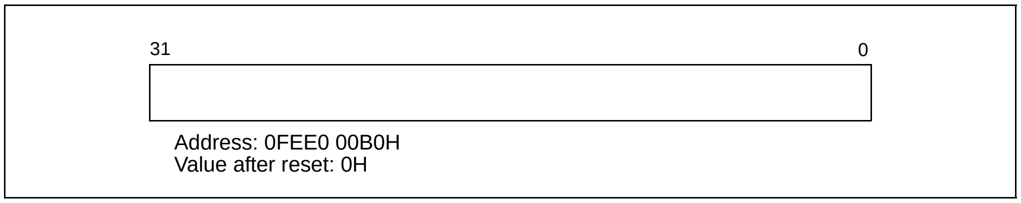


**Figure 10-21 EOI Register**

Upon receiving and EOI, the APIC clears the highest priority bit in the ISR and dispatches the next highest priority interrupt to the processor. If the terminated interrupt was a level-triggered interrupt, the local APIC also sends an end-of-interrupt message to all I/O APICs.

System software may prefer to direct EOIs to specific I/O APICs rather than having the local APIC send end-of-interrupt messages to all I/O APICs.

Software can inhibit the broadcast of EOI message by setting bit 12 of the Spurious Interrupt Vector Register (see Section 10.9). If this bit is set, a broadcast EOI is not generated on an EOI cycle even if the associated TMR bit indicates that the current interrupt was level-triggered. The default value for the bit is 0, indicating that EOI broadcasts are performed.

Bit 12 of the Spurious Interrupt Vector Register is reserved to 0 if the processor does not support suppression of EOI broadcasts. Support for EOI-broadcast suppression is reported in bit 24 in the Local APIC Version Register (see Section 10.4.8); the feature is supported if that bit is set to 1. When supported, the feature is available in both xAPIC mode and x2APIC mode.

System software desiring to perform directed EOIs for level-triggered interrupts should set bit 12 of the Spurious Interrupt Vector Register and follow each the EOI to the local xAPIC for a level triggered interrupt with a directed EOI to the I/O APIC generating the interrupt (this is done by writing to the I/O APIC's EOI register). System software performing directed EOIs must retain a mapping associating level-triggered interrupts with the I/O APICs in the system.

...

## 10.8.6 Task Priority in IA-32e Mode

In IA-32e mode, operating systems can manage the 16 priority classes of external interrupts (see Section 10.8.3, "Interrupt, Task, and Processor Priority") explicitly using the task priority register (TPR). Operating systems can use the TPR to temporarily block specific (low-priority) interrupts from interrupting a high-priority task. This is done by loading TPR with a value corresponding to the highest-priority interrupt that is to be blocked. For example:

- Loading the TPR with a value of 8 (01000B) blocks all interrupts with a priority of 8 or less while allowing all interrupts with a priority of nine or more to be recognized.
- Loading the TPR with zero enables all external interrupts.
- Loading the TPR with 0F (01111B) disables all external interrupts.

The TPR (shown in Figure 10-18) is cleared to 0 on reset. In 64-bit mode, software can read and write the TPR using an alternate interface, MOV CR8 instruction. The new priority level is established when the MOV CR8 instruction completes execution. Software does not need to force serialization after loading the TPR using MOV CR8.

Use of the MOV CRn instruction requires a privilege level of 0. Programs running at privilege level greater than 0 cannot read or write the TPR. An attempt to do so causes a general-protection exception. The TPR is abstracted from the interrupt controller (IC), which prioritizes and manages external interrupt delivery to the processor. The IC can be an external device, such as an APIC or 8259. Typically, the IC provides a priority mechanism similar or identical to the TPR. The IC, however, is considered implementation-dependent with the under-lying priority mechanisms subject to change. CR8, by contrast, is part of the Intel 64 architecture. Software can depend on this definition remaining unchanged.

Figure 10-22 shows the layout of CR8; only the low four bits are used. The remaining 60 bits are reserved and must be written with zeros. Failure to do this causes a general-protection exception.

...

## 10.9 SPURIOUS INTERRUPT

A special situation may occur when a processor raises its task priority to be greater than or equal to the level of the interrupt for which the processor INTR signal is currently being asserted. If at the time the INTA cycle is issued, the interrupt that was to be dispensed has become masked (programmed by software), the local APIC will deliver a spurious-interrupt vector. Dispensing the spurious-interrupt vector does not affect the ISR, so the handler for this vector should return without an EOI.

The vector number for the spurious-interrupt vector is specified in the spurious-interrupt vector register (see Figure 10-23). The functions of the fields in this register are as follows:

**Spurious Vector** Determines the vector number to be delivered to the processor when the local APIC generates a spurious vector.

(Pentium 4 and Intel Xeon processors.) Bits 0 through 7 of the this field are programmable by software.

(P6 family and Pentium processors). Bits 4 through 7 of the this field are programmable by software, and bits 0 through 3 are hardwired to logical ones. Software writes to bits 0 through 3 have no effect.

**APIC Software Enable / Disable**

Allows software to temporarily enable (1) or disable (0) the local APIC (see Section 10.4.3, “Enabling or Disabling the Local APIC”).

**Focus Processor Checking**

Determines if focus processor checking is enabled (0) or disabled (1) when using the lowest-priority delivery mode. In Pentium 4 and Intel Xeon processors, this bit is reserved and should be cleared to 0.

**Suppress EOI Broadcasts**

Determines whether an EOI for a level-triggered interrupt causes EOI messages to be broadcast to the I/O APICs (0) or not (1). See Section 10.8.5. The default value for this bit is 0, indicating that EOI broadcasts are performed. This bit is reserved to 0 if the processor does not support EOI-broadcast suppression.

**NOTE**

Do not program an LVT or IOAPIC RTE with a spurious vector even if you set the mask bit. A spurious vector ISR does not do an EOI. If for some reason an interrupt is generated by an LVT or RTE entry, the bit in the in-service register will be left set for the spurious vector. This will mask all interrupts at the same or lower priority

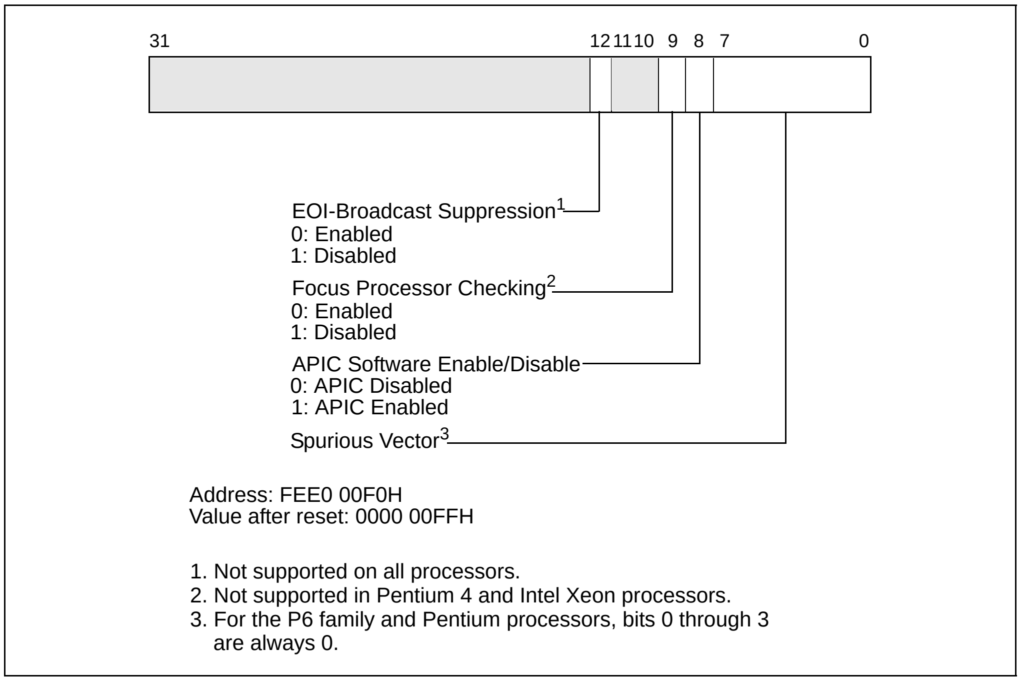

31
12
11
10
9
8
7
0
EOI-Broadcast Suppression[1]
0: Enabled
1: Disabled
Focus Processor Checking[2]
0: Enabled
1: Disabled
APIC Software Enable/Disable
0: APIC Disabled
1: APIC Enabled
Spurious Vector[3]
Address: FEE0 00F0H
Value after reset: 0000 00FFH
1. Not supported on all processors.
2. Not supported in Pentium 4 and Intel Xeon processors.
3. For the P6 family and Pentium processors, bits 0 through 3 are always 0.

**Figure 10-23 Spurious-Interrupt Vector Register (SVR)**

...

## 10.12 EXTENDED XAPIC (X2APIC)

The x2APIC architecture extends the xAPIC architecture (described in Section 9.4) in a backward compatible manner and provides forward extendability for future Intel platform innovations. Specifically, the x2APIC architecture does the following:

- Retains all key elements of compatibility to the xAPIC architecture:
  - delivery modes,
  - interrupt and processor priorities,
  - interrupt sources,
  - interrupt destination types;
- Provides extensions to scale processor addressability for both the logical and physical destination modes;
- Adds new features to enhance performance of interrupt delivery;
- Reduces complexity of logical destination mode interrupt delivery on link based platform architectures.

- Uses MSR programming interface to access APIC registers in x2APIC mode instead of memory-mapped interfaces. Memory-mapped interface is supported when operating in xAPIC mode.

## 10.12.1 Detecting and Enabling x2APIC Mode

Processor support for x2APIC mode can be detected by executing CPUID with EAX=1 and then checking ECX, bit 21 ECX. If CPUID.(EAX=1):ECX.21 is set , the processor supports the x2APIC capability and can be placed into the x2APIC mode.

System software can place the local APIC in the x2APIC mode by setting the x2APIC mode enable bit (bit 10) in the IA32_APIC_BASE MSR at MSR address 01BH. The layout for the IA32_APIC_BASE MSR is shown in Figure 10-26.

Table 10-5, “x2APIC operating mode configurations” describe the possible combinations of the enable bit (EN - bit 11) and the extended mode bit (EXTD - bit 10) in the IA32_APIC_BASE MSR.

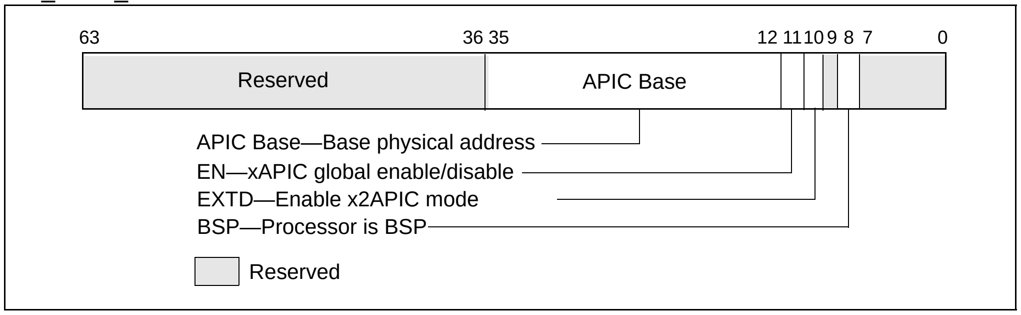


Figure 10-26 IA32_APIC_BASE MSR Supporting x2APIC

Table 10-5 x2APIC Operating Mode Configurations

| xAPIC global enable (IA32_APIC_BASE[11]) | x2APIC enable (IA32_APIC_BASE[10]) | Description |
|---|---|---|
| 0 | 0 | local APIC is disabled |
| 0 | 1 | Invalid |
| 1 | 0 | local APIC is enabled in xAPIC mode |
| 1 | 1 | local APIC is enabled in x2APIC mode |

Once the local APIC has been switched to x2APIC mode (EN = 1, EXTD = 1), switching back to xAPIC mode would require system software to disable the local APIC unit. Specifically, attempting to write a value to the IA32_APIC_BASE MSR that has (EN= 1, EXTD = 0) when the local APIC is enabled and in x2APIC mode causes a general-protection exception. Once bit 10 in IA32_APIC_BASE MSR is set, the only way to leave x2APIC mode using IA32_APIC_BASE would require a WRMSR to set both bit 11 and bit 10 to zero. Section 10.12.5, “x2APIC State Transitions” provides a detailed state diagram for the state transitions allowed for the local APIC.

...

The MSR address range 800H through BFFH is architecturally reserved and dedicated for accessing APIC registers in x2APIC mode. Table 10-6 lists the APIC registers that are available in x2APIC mode. When appropriate, the table also gives the offset at which each register is available on the page referenced by IA32_APIC_BASE[35:12] in xAPIC mode.

There is a one-to-one mapping between the x2APIC MSRs and the legacy xAPIC register offsets with the following exceptions:

- The Destination Format Register (DFR): The DFR, supported at offset 0E0H in x2APIC mode, is not supported in x2APIC mode. There is no MSR with address 80EH.
- The Interrupt Command Register (ICR): The two 32-bit registers in xAPIC mode (at offsets 300H and 310H) are merged into a single 64-bit MSR in x2APIC mode (with MSR address 830H). There is no MSR with address 831H.
- The SELF IPI register. This register is available only in x2APIC mode at address 83FH. In xAPIC mode, there is no register defined at offset 3F0H.

Addresses in the range 800H–BFFH that are not listed in Table 10-6 (including 80EH and 831H) are reserved. Executions of RDMSR and WRMSR that attempt to access such addresses cause general-protection exceptions.

The MSR address space is compressed to allow for future growth. Every 32 bit register on a 128-bit boundary in the legacy MMIO space is mapped to a single MSR in the local x2APIC MSR address space. The upper 32-bits of all x2APIC MSRs (except for the ICR) are reserved.

**Table 10-6 Local APIC Register Address Map Supported by x2APIC**

| MSR Address (x2APIC mode) | MMIO Offset (xAPIC mode) | Register Name | MSR R/W Semantics | Comments |
|---|---|---|---|---|
| 802H | 020H | Local APIC ID register | Read-only[1] | See Section 10.12.5.1 for initial values. |
| 803H | 030H | Local APIC Version register | Read-only | Same version used in xAPIC mode and x2APIC mode. |
| 808H | 080H | Task Priority Register (TPR) | Read/write | Bits 31:8 are reserved.[2] |
| 80AH | 0A0H | Processor Priority Register (PPR) | Read-only | |
| 80BH | 0B0H | EOI register | Write-only[3] | WRMSR of a non-zero value causes #GP(0). |
| 80DH | 0D0H | Logical Destination Register (LDR) | Read-only | Read/write in xAPIC mode. |
| 80FH | 0F0H | Spurious Interrupt Vector Register (SVR) | Read/write | See Section 10.9 for reserved bits. |
| 810H | 100H | In-Service Register (ISR); bits 31:0 | Read-only | |
| 811H | 110H | ISR bits 63:32 | Read-only | |
| 812H | 120H | ISR bits 95:64 | Read-only | |
| 813H | 130H | ISR bits 127:96 | Read-only | |
| 814H | 140H | ISR bits 159:128 | Read-only | |

| MSR Address (x2APIC mode) | MMIO Offset (xAPIC mode) | Register Name | MSR R/W Semantics | Comments |
|---|---|---|---|---|
| 815H | 150H | ISR bits 191:160 | Read-only | |
| 816H | 160H | ISR bits 223:192 | Read-only | |
| 817H | 170H | ISR bits 255:224 | Read-only | |
| 818H | 180H | Trigger Mode Register (TMR); bits 31:0 | Read-only | |
| 819H | 190H | TMR bits 63:32 | Read-only | |
| 81AH | 1A0H | TMR bits 95:64 | Read-only | |
| 81BH | 1B0H | TMR bits 127:96 | Read-only | |
| 81CH | 1C0H | TMR bits 159:128 | Read-only | |
| 81DH | 1D0H | TMR bits 191:160 | Read-only | |
| 81EH | 1E0H | TMR bits 223:192 | Read-only | |
| 81FH | 1F0H | TMR bits 255:224 | Read-only | |
| 820H | 200H | Interrupt Request Register (IRR); bits 31:0 | Read-only | |
| 821H | 210H | IRR bits 63:32 | Read-only | |
| 822H | 220H | IRR bits 95:64 | Read-only | |
| 823H | 230H | IRR bits 127:96 | Read-only | |
| 824H | 240H | IRR bits 159:128 | Read-only | |
| 825H | 250H | IRR bits 191:160 | Read-only | |
| 826H | 260H | IRR bits 223:192 | Read-only | |
| 827H | 270H | IRR bits 255:224 | Read-only | |
| 828H | 280H | Error Status Register (ESR) | Read/write | WRMSR of a non-zero value causes #GP(0). See Section 10.5.3 and Section 10.12.8. |
| 82FH | 2F0H | LVT CMCI register | Read/write | See Figure 15-10 for reserved bits. |
| 830H[4] | 300H and 310H | Interrupt Command Register (ICR) | Read/write | See Figure 10-29 for reserved bits |
| 832H | 320H | LVT Timer register | Read/write | See Figure 10-8 for reserved bits. |
| 833H | 330H | LVT Thermal Sensor register | Read/write | See Figure 10-8 for reserved bits. |
| 834H | 340H | LVT Performance Monitoring register | Read/write | See Figure 10-8 for reserved bits. |
| 835H | 350H | LVT LINT0 register | Read/write | See Figure 10-8 for reserved bits. |
| 836H | 360H | LVT LINT1 register | Read/write | See Figure 10-8 for reserved bits. |

| MSR Address (x2APIC mode) | MMIO Offset (xAPIC mode) | Register Name | MSR R/W Semantics | Comments |
|---|---|---|---|---|
| 837H | 370H | LVT Error register | Read/write | See Figure 10-8 for reserved bits. |
| 838H | 380H | Initial Count register (for Timer) | Read/write | |
| 839H | 390H | Current Count register (for Timer) | Read-only | |
| 83EH | 3E0H | Divide Configuration Register (DCR; for Timer) | Read/write | See Figure 10-10 for reserved bits. |
| 83FH | Not available | SELF IPI[5] | Write-only | Available only in x2APIC mode. |

**NOTES:**

1. WRMSR causes #GP(0) for read-only registers.
2. WRMSR causes #GP(0) for attempts to set a reserved bit to 1 in a read/write register (including bits 63:32 of each register).
3. RDMSR causes #GP(0) for write-only registers.
4. MSR 831H is reserved; read/write operations cause general-protection exceptions. The contents of the APIC register at MMIO offset 310H are accessible in x2APIC mode through the MSR at address 830H.
5. SELF IPI register is supported only in x2APIC mode.

### 10.12.1.3 Reserved Bit Checking

Section 10.12.1.2 and Table 10-6 specifies the reserved bit definitions for the APIC registers in x2APIC mode. Non-zero writes (by WRMSR instruction) to reserved bits to these registers will raise a general protection fault exception while reads return zeros (RsvdZ semantics).

In x2APIC mode, the local APIC ID register is increased to 32 bits wide. This enables $2^{32}$–1 processors to be addressable in physical destination mode. This 32-bit value is referred to as “x2APIC ID”. A processor implementation may choose to support less than 32 bits in its hardware. System software should be agnostic to the actual number of bits that are implemented. All non-implemented bits will return zeros on reads by software.

The APIC ID value of FFFF_FFFFH and the highest value corresponding to the implemented bit-width of the local APIC ID register in the system are reserved and cannot be assigned to any logical processor.

In x2APIC mode, the local APIC ID register is a read-only register to system software and will be initialized by hardware. It is accessed via the RDMSR instruction reading the MSR at address 0802H.

Each logical processor in the system (including clusters with a communication fabric) must be configured with an unique x2APIC ID to avoid collisions of x2APIC IDs. On DP and high-end MP processors targeted to specific market segments and depending on the system configuration, it is possible that logical processors in different and “unconnected” clusters power up initialized with overlapping x2APIC IDs. In these configurations, a model-specific means may be provided in those product segments to enable BIOS and/or platform firmware to re-configure the x2APIC IDs in some clusters to provide for unique and non-overlapping system wide IDs before configuring the disconnected components into a single system.

### 10.12.2 x2APIC Register Availability

The local APIC registers can be accessed via the MSR interface only when the local APIC has been switched to the x2APIC mode as described in Section 10.12.1. Accessing any APIC register in the MSR address range 0800H through 0BFFH via RDMSR or WRMSR when the local APIC is not in x2APIC mode causes a general-protection exception. In x2APIC mode, the memory mapped interface is not available and any access to the MMIO interface will behave similar to that of a legacy xAPIC in globally disabled state. Table 10-7 provides the interactions between the legacy & extended modes and the legacy and register interfaces.

Table 10-7 MSR/MMIO Interface of a Local x2APIC in Different Modes of Operation

| | MMIO Interface | MSR Interface |
|---|---|---|
| xAPIC mode | Available | General-protection exception |
| x2APIC mode | Behavior identical to xAPIC in globally disabled state | Available |

### 10.12.3 MSR Access in x2APIC Mode

To allow for efficient access to the APIC registers in x2APIC mode, the serializing semantics of WRMSR are relaxed when writing to the APIC registers. Thus, system software should not use “WRMSR to APIC registers in x2APIC mode” as a serializing instruction. Read and write accesses to the APIC registers will occur in program order. A WRMSR to an APIC register may complete before all preceding stores are globally visible; software can prevent this by inserting a serializing instruction, an SFENCE, or an MFENCE before the WRMSR.

The RDMSR instruction is not serializing and this behavior is unchanged when reading APIC registers in x2APIC mode. System software accessing the APIC registers using the RDMSR instruction should not expect a serializing behavior. (Note: The MMIO-based xAPIC interface is mapped by system software as an un-cached region. Consequently, read/writes to the xAPIC-MMIO interface have serializing semantics in the xAPIC mode.)

### 10.12.4 VM-Exit Controls for MSRs and x2APIC Registers

The VMX architecture allows a VMM to specify lists of MSRs to be loaded or stored on VMX transitions using the VMX-transition MSR areas (see VM-exit MSR-store address field, VM-exit MSR-load address filed, and VM-entry MSR-load address field in *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B*).

The X2APIC MSRs cannot to be loaded and stored on VMX transitions. A VMX transition fails if the VMM has specified that the transition should access any MSRs in the address range from 0000_0800H to 0000_08FFH (the range used for accessing the X2APIC registers). Specifically, processing of an 128-bit entry in any of the VMX-transition MSR areas fails if bits 31:0 of that entry (represented as ENTRY_LOW_DW) satisfies the expression: “ENTRY_LOW_DW & FFFFF800H = 00000800H”. Such a failure causes an associated VM entry to fail (by reloading host state) and causes an associated VM exit to lead to VMX abort.

### 10.12.5 x2APIC State Transitions

This section provides a detailed description of the x2APIC states of a local x2APIC unit, transitions between these states as well as interactions of these states with INIT and RESET.

#### 10.12.5.1 x2APIC States

The valid states for a local x2APIC unit is listed in Table 10-5:

- APIC disabled: IA32_APIC_BASE[EN]=0 and IA32_APIC_BASE[EXTD]=0
- xAPIC mode: IA32_APIC_BASE[EN]=1 and IA32_APIC_BASE[EXTD]=0
- x2APIC mode: IA32_APIC_BASE[EN]=1 and IA32_APIC_BASE[EXTD]=1
- Invalid: IA32_APIC_BASE[EN]=0 and IA32_APIC_BASE[EXTD]=1

The state corresponding to EXTD=1 and EN=0 is not valid and it is not possible to get into this state. An execution of WRMSR to the IA32_APIC_BASE_MSR that attempts a transition from a valid state to this invalid state causes a general-protection exception. Figure 10-27 shows the comprehensive state transition diagram for a local x2APIC unit.

...

#### x2APIC Transitions From x2APIC Mode

From the x2APIC mode, the only valid x2APIC transition using IA32_APIC_BASE is to the state where the x2APIC is disabled by setting EN to 0 and EXTD to 0. The x2APIC ID (32 bits) and the legacy local xAPIC ID (8 bits) are preserved across this transition. A transition from the x2APIC mode to xAPIC mode is not valid, and the corresponding WRMSR to the IA32_APIC_BASE MSR causes a general-protection exception.

A RESET in this state places the x2APIC in xAPIC mode. All APIC registers (including the local APIC ID register) are initialized as described in Section 10.12.5.1.

An INIT in this state keeps the x2APIC in the x2APIC mode. The state of the local APIC ID register is preserved (all 32 bits). However, all the other APIC registers are initialized as a result of the INIT transition.

...

### 10.12.7 CPUID Extensions And Topology Enumeration

For Intel 64 and IA-32 processors that support x2APIC, a value of 1 reported by CPUID.01H:ECX[21] indicates that the processor supports x2APIC and the extended topology enumeration leaf (CPUID.0BH).

The extended topology enumeration leaf can be accessed by executing CPUID with EAX = 0BH. Processors that do not support x2APIC may support CPUID leaf 0BH. Software can detect the availability of the 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 or 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. If available, the extended topology enumeration leaf is the preferred mechanism for enumerating topology. The presence of CPUID leaf 0BH in a processor does not guarantee support for 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.

The extended topology enumeration leaf is intended to assist software with enumerating processor topology on systems that requires 32-bit x2APIC IDs to address individual logical processors. Details of CPUID leaf 0BH can be found in the reference pages of CPUID in Chapter 3 of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*.

Processor topology enumeration algorithm for processors supporting the extended topology enumeration leaf of CPUID and processors that do not support CPUID leaf 0BH are treated in Section 8.9.4, "Algorithm for Three-Level Mappings of APIC_ID".

...

### 10.12.8 Error Handling in x2APIC Mode

RDMSR and WRMSR operations to reserved addresses in x2APIC mode cause general-protection exceptions, as do reserved-bit violations (see Section 10.12.1.3). Beyond illegal register access and reserved bit violations, other APIC errors are logged in Error Status Register. Writes of a non-zero value to the Error Status Register in x2APIC mode cause general-protection exceptions. Figure 10-28 illustrates the Error Status Register in x2APIC mode.

Write to the ICR (in xAPIC and x2APIC modes) or to SELF IPI register (x2APIC mode only) with an illegal vector (vector ≤ 0FH) will set the "Send Illegal Vector" bit.

On receiving an IPI with an illegal vector (vector ≤ 0FH), the "Receive Illegal Vector" bit will be set. On receiving an interrupt with illegal vector in the range 0H – 0FH, the interrupt will not be delivered to the processor nor will an IRR bit be set in that range. Only the ESR "Receive Illegal Vector" bit will be set.

If the ICR is programmed with lowest priority delivery mode then the "Re-directible IPI" bit will be set in x2APIC modes (same as legacy xAPIC behavior) and the interrupt will not be processed.

Write to the ICR with both lowest priority delivery mode and illegal vector, will set the "re-directible IPI" error bit. The interrupt will not be processed and hence the "Send Illegal Vector" error bit will not be set.

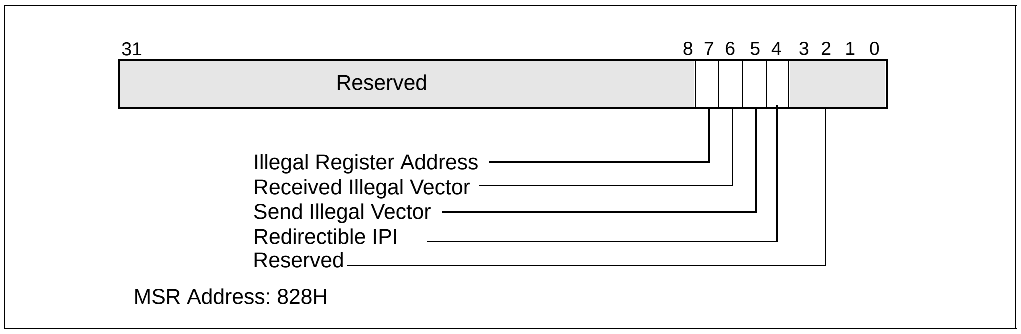


**Figure 10-28 Error Status Register (ESR) in x2APIC Mode**

### 10.12.9 ICR Operation in x2APIC Mode

In x2APIC mode, the layout of the Interrupt Command Register is shown in Figure 10-12. The lower 32 bits of ICR in x2APIC mode is identical to the lower half of the ICR in xAPIC mode, except the Delivery Status bit is removed since it is not needed in x2APIC mode. The destination ID field is expanded to 32 bits in x2APIC mode.

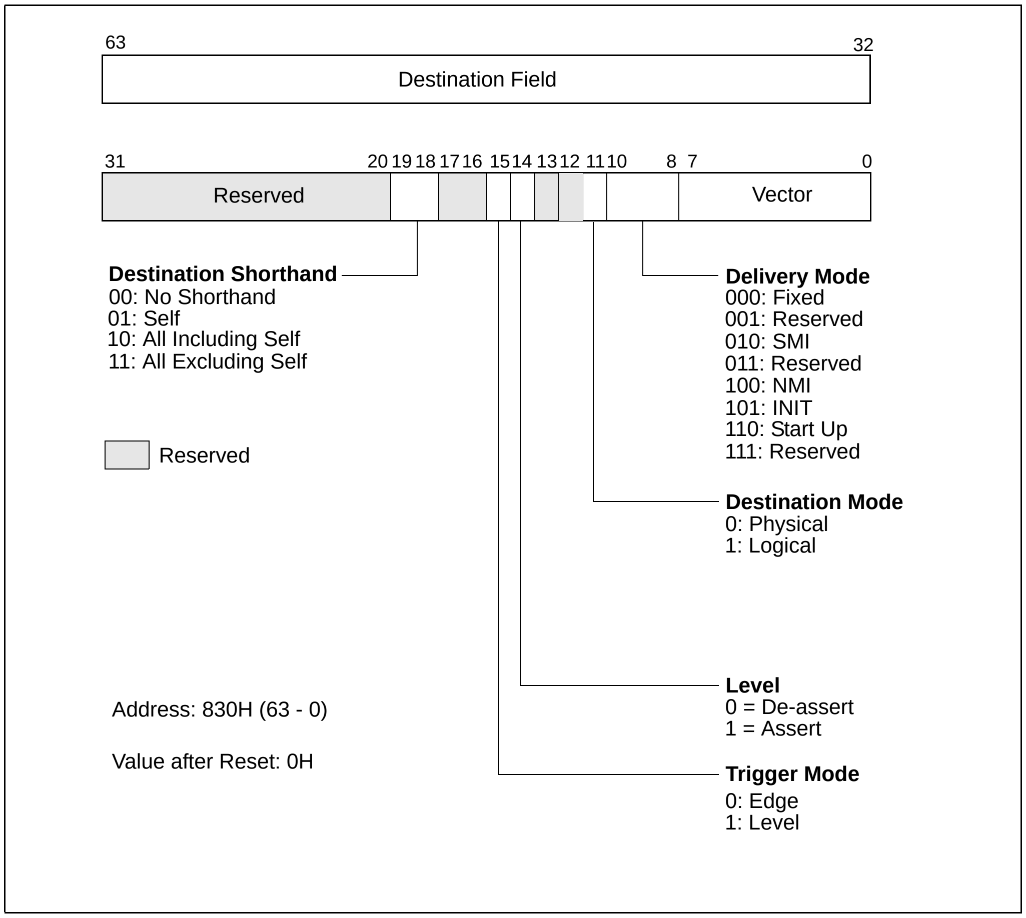


**Figure 10-29. Interrupt Command Register (ICR) in x2APIC Mode**

To send an IPI using the ICR, software must set up the ICR to indicate the type of IPI message to be sent and the destination processor or processors. Self IPIs can also be sent using the SELF IPI register (see Section 10.12.11).

A single MSR write to the Interrupt Command Register is required for dispatching an interrupt in x2APIC mode. With the removal of the Delivery Status bit, system software no longer has a reason to read the ICR. It remains readable only to aid in debugging; however, software should not assume the value returned by reading the ICR is the last written value

A destination ID value of FFFF_FFFFH is used for broadcast of interrupts in both logical destination and physical destination modes.

## 10.12.10 Determining IPI Destination in x2APIC Mode

### 10.12.10.1 Logical Destination Mode in x2APIC Mode

In x2APIC mode, the Logical Destination Register (LDR) is increased to 32 bits wide. It is a read-only register to system software. This 32-bit value is referred to as "logical x2APIC ID". System software accesses this register via the RDMSR instruction reading the MSR at address 80DH. Figure 10-30 provides the layout of the Logical Destination Register in x2APIC mode.

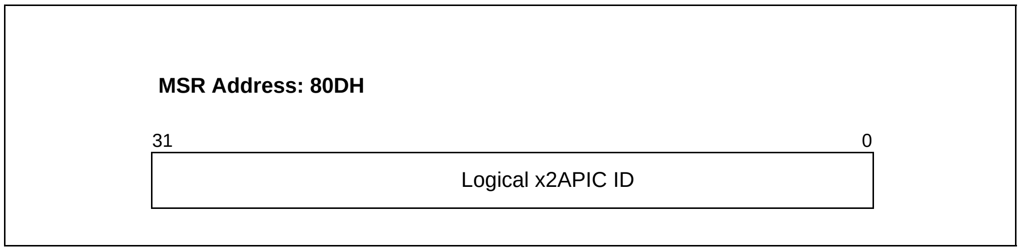


**Figure 10-30 Logical Destination Register in x2APIC Mode**

In the xAPIC mode, the Destination Format Register (DFR) through MMIO interface determines the choice of a flat logical mode or a clustered logical mode. Flat logical mode is not supported in the x2APIC mode. Hence the Destination Format Register (DFR) is eliminated in x2APIC mode.

The 32-bit logical x2APIC ID field of LDR is partitioned into two sub-fields:

- Cluster ID (LDR[31:16]): is the address of the destination cluster
- Logical ID (LDR[15:0]): defines a logical ID of the individual local x2APIC within the cluster specified by LDR[31:16].

This layout enables 2^16-1 clusters each with up to 16 unique logical IDs - effectively providing an addressability of ((2^20) - 16) processors in logical destination mode.

It is likely that processor implementations may choose to support less than 16 bits of the cluster ID or less than 16-bits of the Logical ID in the Logical Destination Register. However system software should be agnostic to the number of bits implemented in the cluster ID and logical ID sub-fields. The x2APIC hardware initialization will ensure that the appropriately initialized logical x2APIC IDs are available to system software and reads of non-implemented bits return zero. This is a read-only register that software must read to determine the logical x2APIC ID of the processor. Specifically, software can apply a 16-bit mask to the lowest 16 bits of the logical x2APIC ID to identify the logical address of a processor within a cluster without needing to know the number of implemented bits in cluster ID and Logical ID sub-fields. Similarly, software can create a message destination address for cluster model, by bit-Oring the Logical X2APIC ID (31:0) of processors that have matching Cluster ID(31:16).

To enable cluster ID assignment in a fashion that matches the system topology characteristics and to enable efficient routing of logical mode lowest priority device interrupts in link based platform interconnects, the LDR are initialized by hardware based on the value of x2APIC ID upon x2APIC state transitions. Details of this initialization are provided in Section 10.12.10.2.

### 10.12.10.2 Deriving Logical x2APIC ID from the Local x2APIC ID

In x2APIC mode, the 32-bit logical x2APIC ID, which can be read from LDR, is derived from the 32-bit local x2APIC ID. Specifically, the 16-bit logical ID sub-field is derived by shifting 1 by the lowest 4 bits of the x2APIC ID, i.e. Logical ID = 1 « x2APIC ID[3:0]. The remaining bits of the x2APIC ID then form the cluster ID portion of the logical x2APIC ID:

Logical x2APIC ID = [(x2APIC ID[19:4] « 16) | (1 « x2APIC ID[3:0])]

The use of the lowest 4 bits in the x2APIC ID implies that at least 16 APIC IDs are reserved for logical processors within a socket in multi-socket configurations. If more than 16 APIC IDS are reserved for logical processors in a socket/package then multiple cluster IDs can exist within the package.

The LDR initialization occurs whenever the x2APIC mode is enabled (see Section 10.12.5).

## 10.12.11 SELF IPI Register

SELF IPIs are used extensively by some system software. The x2APIC architecture introduces a new register interface. This new register is dedicated to the purpose of sending self-IPIs with the intent of enabling a highly optimized path for sending self-IPIs.

Figure 10-31 provides the layout of the SELF IPI register. System software only specifies the vector associated with the interrupt to be sent. The semantics of sending a self-IPI via the SELF IPI register are identical to sending a self targeted edge triggered fixed interrupt with the specified vector. Specifically the semantics are identical to the following settings for an inter-processor interrupt sent via the ICR - Destination Shorthand (ICR[19:18] = 01 (Self)), Trigger Mode (ICR[15] = 0 (Edge)), Delivery Mode (ICR[10:8] = 000 (Fixed)), Vector (ICR[7:0] = Vector).

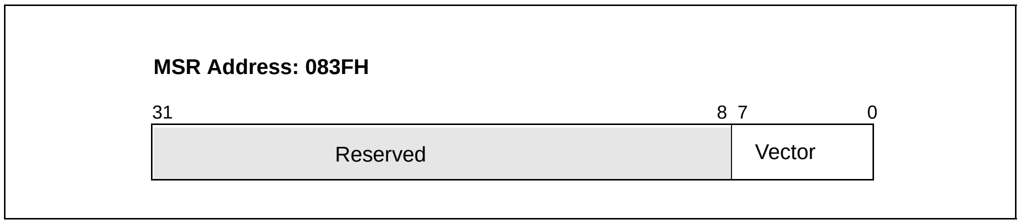


Figure 10-31 SELF IPI register

The SELF IPI register is a write-only register. A RDMSR instruction with address of the SELF IPI register causes a general-protection exception.

The handling and prioritization of a self-IPI sent via the SELF IPI register is architecturally identical to that for an IPI sent via the ICR from a legacy xAPIC unit. Specifically the state of the interrupt would be tracked via the Interrupt Request Register (IRR) and In Service Register (ISR) and Trigger Mode Register (TMR) as if it were received from the system bus. Also sending the IPI via the Self Interrupt Register ensures that interrupt is delivered to the processor core. Specifically completion of the WRMSR instruction to the SELF IPI register implies that the interrupt has been logged into the IRR. As expected for edge triggered interrupts, depending on the processor priority and readiness to accept interrupts, it is possible that interrupts sent via the SELF IPI register or via the ICR with identical vectors can be combined.

## 7. Updates to Chapter 15, Volume 3A

Change bars show changes to Chapter 15 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A:* System Programming Guide, Part 1.

------------------------------------------------------------------------------------

...

Table 15-7 lists overwrite rules for uncorrected errors, corrected errors, and uncorrected recoverable errors.

**Table 15-7 Overwrite Rules for UC, CE, and UCR Errors**

| First Event | Second Event | UC | PCC | S | AR | MCA Bank | Reset System |
|---|---|---|---|---|---|---|---|
| CE | UCR | 1 | 0 | 0 if UCNA, else 1 | 1 if SRAR, else 0 | second | yes, if AR=1 |
| UCR | CE | 1 | 0 | 0 if UCNA, else 1 | 1 if SRAR, else 0 | first | yes, if AR=1 |
| UCNA | UCNA | 1 | 0 | 0 | 0 | first | no |
| UCNA | SRAO | 1 | 0 | 1 | 0 | first | no |
| UCNA | SRAR | 1 | 0 | 1 | 1 | first | yes |
| SRAO | UCNA | 1 | 0 | 1 | 0 | first | no |
| SRAO | SRAO | 1 | 0 | 1 | 0 | first | no |
| SRAO | SRAR | 1 | 0 | 1 | 1 | first | yes |
| SRAR | UCNA | 1 | 0 | 1 | 1 | first | yes |
| SRAR | SRAO | 1 | 0 | 1 | 1 | first | yes |
| SRAR | SRAR | 1 | 0 | 1 | 1 | first | yes |
| UCR | UC | 1 | 1 | undefined | undefined | second | yes |
| UC | UCR | 1 | 1 | undefined | undefined | first | yes |

...

## 8. Updates to Chapter 21, Volume 3B

Change bars show changes to Chapter 21 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B:* System Programming Guide, Part 2.

------------------------------------------------------------------------------------------

...

## 21.1 OVERVIEW

A logical processor uses **virtual-machine control data structures** (**VMCSs**) while it is in VMX operation. These manage transitions into and out of VMX non-root operation (VM entries and VM exits) as well as processor behavior in VMX non-root operation. This structure is manipulated by the new instructions VMCLEAR, VMPTRLD, VMREAD, and VMWRITE.

A VMM can use a different VMCS for each virtual machine that it supports. For a virtual machine with multiple logical processors (virtual processors), the VMM can use a different VMCS for each virtual processor.

A logical processor associates a region in memory with each VMCS. This region is called the **VMCS region**.[1] Software references a specific VMCS using the 64-bit physical address of the region (a **VMCS pointer**). VMCS pointers must be aligned on a 4-KByte boundary (bits 11:0 must be zero). On processors that support Intel 64 architecture, these pointers must not set bits beyond the processor's physical-address width.[2] On processors that do not support Intel 64 architecture, they must not set any bits in the range 63:32.

A logical processor may maintain a number of VMCSs that are **active**. The processor may optimize VMX operation by maintaining the state of an active VMCS in memory, on the processor, or both. At any given time, at most one of the active VMCSs is the **current** VMCS. (This document frequently uses the term "the VMCS" to refer to the current VMCS.) The VMLAUNCH, VMREAD, VMRESUME, and VMWRITE instructions operate only on the current VMCS.

The following items describe how a logical processor determines which VMCSs are active and which is current:

- The memory operand of the VMPTRLD instruction is the address of a VMCS. After execution of the instruction, that VMCS is both active and current on the logical processor. Any other VMCS that had been active remains so, but no other VMCS is current.
- The memory operand of the VMCLEAR instruction is also the address of a VMCS. After execution of the instruction, that VMCS is neither active nor current on the logical processor. If the VMCS had been current on the logical processor, the logical processor no longer has a current VMCS.

1. The amount of memory required for a VMCS region is at most 4 KBytes. The exact size is implementation specific and can be determined by consulting the VMX capability MSR IA32_VMX_BASIC to determine the size of the VMCS region (see Appendix G.1).
2. 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.

The VMPTRST instruction stores the address of the logical processor's current VMCS into a specified memory location (it stores the value FFFFFFFF_FFFFFFFFH if there is no current VMCS).

The **launch state** of a VMCS determines which VM-entry instruction should be used with that VMCS: the VMLAUNCH instruction requires a VMCS whose launch state is "clear"; the VMRESUME instruction requires a VMCS whose launch state is "launched". A logical processor maintains a VMCS's launch state in the corresponding VMCS region. The following items describe how a logical processor manages the launch state of a VMCS:

- If the launch state of the current VMCS is "clear", successful execution of the VMLAUNCH instruction changes the launch state to "launched".
- The memory operand of the VMCLEAR instruction is the address of a VMCS. After execution of the instruction, the launch state of that VMCS is "clear".
- There are no other ways to modify the launch state of a VMCS (it cannot be modified using VMWRITE) and there is no direct way to discover it (it cannot be read using VMREAD).

Figure 21-1 illustrates the different states of a VMCS. It uses "X" to refer to the VMCS and "Y" to refer to any other VMCS. Thus: "VMPTRLD X" always makes X current and active; "VMPTRLD Y" always makes X not current (because it makes Y current); VMLAUNCH makes the launch state of X "launched" if X was current and its launch state was "clear"; and VMCLEAR X always makes X inactive and not current and makes its launch state "clear".

The figure does not illustrate operations that do not modify the VMCS state relative to these parameters (e.g., execution of VMPTRLD X when X is already current). Note that VMCLEAR X makes X "inactive, not current, and clear," even if X's current state is not defined (e.g., even if X has not yet been initialized). See Section 21.11.

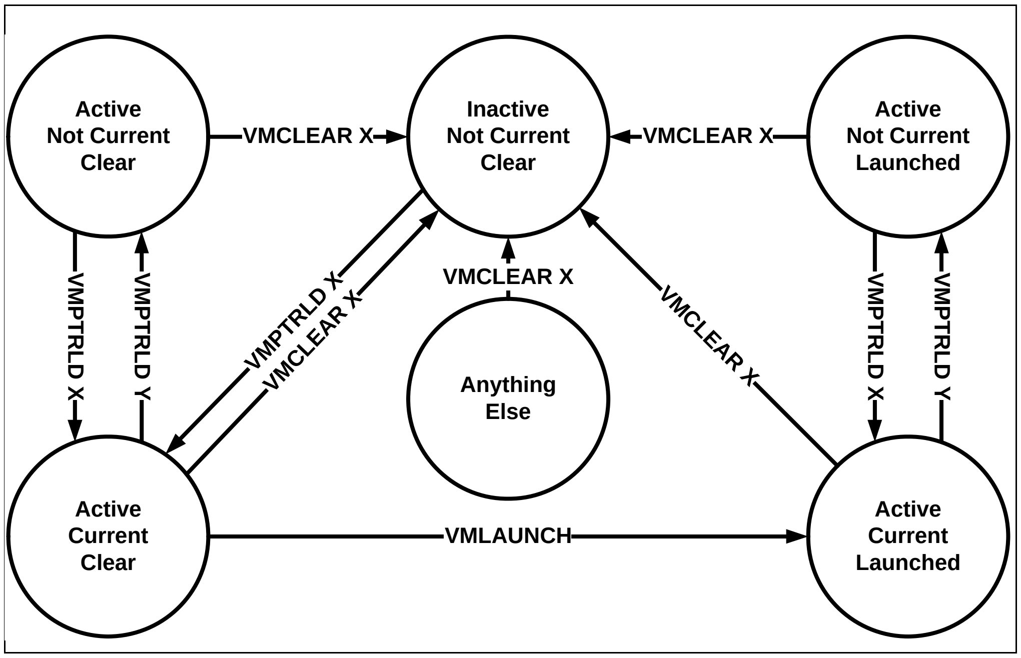


Figure 21-1 States of VMCS X

...

## 21.10 SOFTWARE USE OF THE VMCS AND RELATED STRUCTURES

This section details guidelines that software should observe when using a VMCS and related structures. It also provides descriptions of consequences for failing to follow guidelines.

### 21.10.1 Software Use of Virtual-Machine Control Structures

To ensure proper processor behavior, software should observe certain guidelines when using an active VMCS.

No VMCS should ever be active on more than one logical processor. If a VMCS is to be “migrated” from one logical processor to another, the first logical processor should execute VMCLEAR for the VMCS (to make it inactive on that logical processor and to ensure that all VMCS data are in memory) before the other logical processor executes VMPTRLD for the VMCS (to make it active on the second logical processor). A VMCS that is made active on more than one logical processor may become **corrupted** (see below).

Software should use the VMREAD and VMWRITE instructions to access the different fields in the current VMCS (see Section 21.10.2). Software should never access or modify the VMCS data of an active VMCS using ordinary memory operations, in part because the format used to store the VMCS data is implementation-specific and not architecturally defined, and also because a logical processor may maintain some VMCS data of an active VMCS on the processor and not in the VMCS region. The following items detail some of the hazards of accessing VMCS data using ordinary memory operations:

- Any data read from a VMCS with an ordinary memory read does not reliably reflect the state of the VMCS. Results may vary from time to time or from logical processor to logical processor.
- Writing to a VMCS with an ordinary memory write is not guaranteed to have a deterministic effect on the VMCS. Doing so may cause the VMCS to become corrupted (see below).

(Software can avoid these hazards by removing any linear-address mappings to a VMCS region before executing a VMPTRLD for that region and by not remapping it until after executing VMCLEAR for that region.)

If a logical processor leaves VMX operation, any VMCSs active on that logical processor may be corrupted (see below). To prevent such corruption of a VMCS that may be used either after a return to VMX operation or on another logical processor, software should VMCLEAR that VMCS before executing the VMXOFF instruction or removing power from the processor (e.g., as part of a transition to the S3 and S4 power states).

This section has identified operations that may cause a VMCS to become corrupted. These operations may cause the VMCS's data to become undefined. Behavior may be unpredictable if that VMCS used subsequently on any logical processor. The following items detail some hazards of VMCS corruption:

- VM entries may fail for unexplained reasons or may load undesired processor state.
- The processor may not correctly support VMX non-root operation as documented in Chapter 21 and may generate unexpected VM exits.
- VM exits may load undesired processor state, save incorrect state into the VMCS, or cause the logical processor to transition to a shutdown state.

...

### 21.10.3 Initializing a VMCS

Software should initialize fields in a VMCS (using VMWRITE) before using the VMCS for VM entry. Failure to do so may result in unpredictable behavior; for example, a VM entry may fail for unexplained reasons, or a successful transition (VM entry or VM exit) may load processor state with unexpected values.

It is not necessary to initialize fields that the logical processor will not use. (For example, it is not necessary to initialize the MSR-bitmap address if the "use MSR bitmaps" VM-execution control is 0.)

A processor maintains some VMCS information that cannot be modified with the VMWRITE instruction; this includes a VMCS's launch state (see Section 21.1). Such information may be stored in the VMCS data portion of a VMCS region. Because the format of this information is implementation-specific, there is no way for software to know, when it first allocates a region of memory for use as a VMCS region, how the processor will determine this information from the contents of the memory region.

In addition to its other functions, the VMCLEAR instruction initializes any implementation-specific information in the VMCS region referenced by its operand. To avoid the uncertainties of implementation-specific behavior, software should execute VMCLEAR on a VMCS region before making the corresponding VMCS active with VMPTRLD for the first time. (Figure 21-1 illustrates how execution of VMCLEAR puts a VMCS into a well-defined state.)

The following software usage is consistent with these limitations:

- VMCLEAR should be executed for a VMCS before it is used for VM entry for the first time.
- VMLAUNCH should be used for the first VM entry using a VMCS after VMCLEAR has been executed for that VMCS.
- VMRESUME should be used for any subsequent VM entry using a VMCS (until the next execution of VMCLEAR for the VMCS).

It is expected that, in general, VMRESUME will have lower latency than VMLAUNCH. Since “migrating” a VMCS from one logical processor to another requires use of VMCLEAR (see Section 21.10.1), which sets the launch state of the VMCS to “clear”, such migration requires the next VM entry to be performed using VMLAUNCH. Software developers can avoid the performance cost of increased VM-entry latency by avoiding unnecessary migration of a VMCS from one logical processor to another.

...

## 9. 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.

------------------------------------------------------------------------------------------

...

### 22.1.1 Relative Priority of Faults and VM Exits

The following principles describe the ordering between existing faults and VM exits:

- Certain exceptions have priority over VM exits. These include invalid-opcode exceptions, faults based on privilege level,[1] and general-protection exceptions that are based on checking I/O permission bits in the task-state segment (TSS). For example, execution of RDMSR with CPL = 3 generates a general-protection exception and not a VM exit.[2]

...

1. These include faults generated by attempts to execute, in virtual-8086 mode, privileged instructions that are not recognized in that mode.
2. MOV DR is an exception to this rule; see Section 22.1.3.

## 10. Updates to Chapter 25, Volume 3B

Change bars show changes to Chapter 25 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B:* System Programming Guide, Part 2.

------------------------------------------------------------------------------------------

...

### 25.2.2 EPT Translation Mechanism

...

Because a PDPTE is identified using bits 47:30 of the guest-physical address, it controls access to a 1-GByte region of the guest-physical-address space. Use of the PDPTE depends on the value of bit 7 in that entry:[1]

- If bit 7 of the EPT PDPTE is 1, the EPT PDPTE maps a 1-GByte page (see Table 25-2). The final physical address is computed as follows:

**Table 25-2 Format of an EPT Page-Directory-Pointer-Table Entry (PDPTE) that Maps a 1-GByte Page**

| Bit Position(s) | Contents |
|---|---|
| 0 | Read access; indicates whether reads are allowed from the 1-GByte page referenced by this entry |
| 1 | Write access; indicates whether writes are allowed to the 1-GByte page referenced by this entry |
| 2 | Execute access; indicates whether instruction fetches are allowed from the 1-GByte page referenced by this entry |
| 5:3 | EPT memory type for this 1-GByte page (see Section 25.2.4) |
| 6 | Ignore PAT memory type for this 1-GByte page (see Section 25.2.4) |
| 7 | Must be 1 (otherwise, this entry references an EPT page directory) |
| 11:8 | Ignored |
| 29:12 | Reserved (must be 0) |
| (N–1):30 | Physical address of the 1-GByte page referenced by this entry[1] |
| 51:N | Reserved (must be 0) |
| 63:52 | Ignored |

**NOTES:**
1. N is the physical-address width supported by the logical processor.

1. Not all processors allow bit 7 of an EPT PDPTE to be set to 1. Software should read the VMX capability MSR IA32_VMX_EPT_VPID_CAP (see Appendix G.10) to determine whether this is allowed.

- Bits 63:52 are all 0.
- Bits 51:30 are from the EPT PDPTE.
- Bits 29:0 are from the original guest-physical address.

- If bit 7 of the EPT PDPTE is 0, a 4-KByte naturally aligned EPT page directory is located at the physical address specified in bits 51:12 of the EPT PDPTE (see Table 25-3). An EPT page-directory comprises 512 64-bit entries (PDEs). An EPT PDE is selected using the physical address defined as follows:
  - Bits 63:52 are all 0.
  - Bits 51:12 are from the EPT PDPTE.
  - Bits 11:3 are bits 29:21 of the guest-physical address.

## 11. Updates to Chapter 27, Volume 3B

Change bars show changes to Chapter 27 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B:* System Programming Guide, Part 2.

------------------------------------------------------------------------------------------

...

## 27.3 MANAGING VMCS REGIONS AND POINTERS

A VMM must observe necessary procedures when working with a VMCS, the associated VMCS pointer, and the VMCS region. It must also not assume the state of persistency for VMCS regions in memory or cache.

Before entering VMX operation, the host VMM allocates a VMXON region. A VMM can host several virtual machines and have many VMCSs active under its management. A unique VMCS region is required for each virtual machine; a VMXON region is required for the VMM itself.

A VMM determines the VMCS region size by reading IA32_VMX_BASIC MSR; it creates VMCS regions of this size using a 4-KByte-aligned area of physical memory. Each VMCS region needs to be initialized with a VMCS revision identifier (at byte offset 0) identical to the revision reported by the processor in the VMX capability MSR.

### NOTE

Software must not read or write directly to the VMCS data region as the format is not architecturally defined. Consequently, Intel recommends that the VMM remove any linear-address mappings to VMCS regions before loading.

System software does not need to do special preparation to the VMXON region before entering into VMX operation. The address of the VMXON region for the VMM is provided as an operand to VMXON instruction. Once in VMX root operation, the VMM needs to prepare data fields in the VMCS that control the execution of a VM upon a VM entry. The VMM can make a VMCS the current VMCS by using the VMPTRLD instruction. VMCS data fields must be read or written only through VMREAD and VMWRITE commands respectively.

Every component of the VMCS is identified by a 32-bit encoding that is provided as an operand to VMREAD and VMWRITE. Appendix H provides the encodings. A VMM must properly initialize all fields in a VMCS before using the current VMCS for VM entry.

A VMCS is referred to as a controlling VMCS if it is the current VMCS on a logical processor in VMX non-root operation. A current VMCS for controlling a logical processor in VMX non-root operation may be referred to as a working VMCS if the logical processor is not in VMX non-root operation. The relationship of active, current (i.e. working) and controlling VMCS during VMX operation is shown in Figure 27-1.

### NOTE

As noted in Section 21.1, the processor may optimize VMX operation by maintaining the state of an active VMCS (one for which VMPTRLD has been executed) on the processor. Before relinquishing control to other system software that may, without informing the VMM, remove power from the processor (e.g., for transitions to S3 or S4) or leave VMX operation, a VMM must VMCLEAR all active VMCSs. This ensures that all

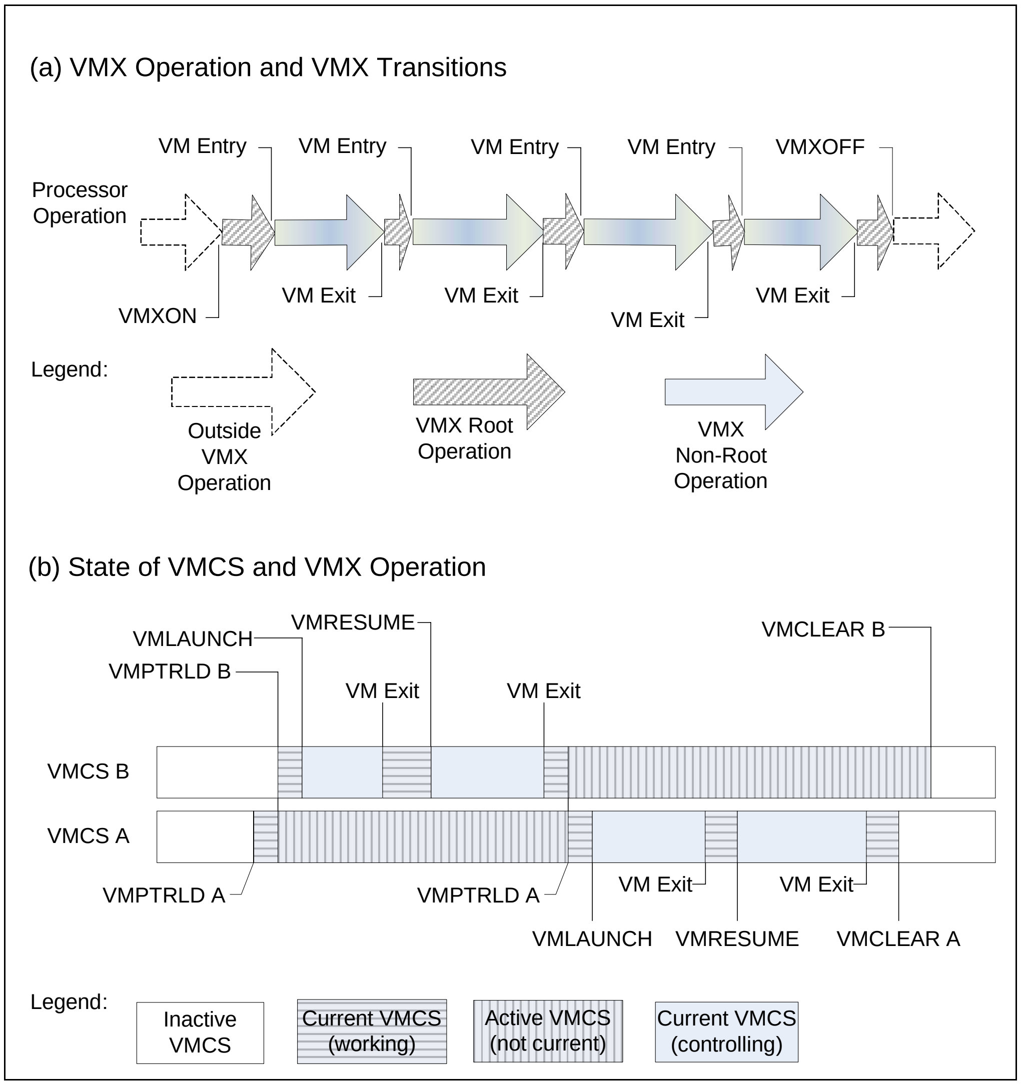


**Figure 27-1 VMX Transitions and States of VMCS in a Logical Processor**

VMCS data cached by the processor are flushed to memory and that no other software can corrupt the current VMM's VMCS data. It is also recommended that the VMM execute VMXOFF after such executions of VMCLEAR.

The VMX capability MSR IA32_VMX_BASIC reports the memory type used by the processor for accessing a VMCS or any data structures referenced through pointers in the VMCS. Software must maintain the VMCS structures in cache-coherent memory. Software must always map the regions hosting the I/O bitmaps, MSR bitmaps, VM-exit MSR-store area, VM-exit MSR-load area, and VM-entry MSR-load area to the write-back (WB) memory type. Mapping these regions to uncacheable (UC) memory type is supported, but strongly discouraged due to negative impact on performance.

...

## 12. Updates to Chapter 30, Volume 3B

Change bars show changes to Chapter 30 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B:* System Programming Guide, Part 2.

------------------------------------------------------------------------------------

...

### 30.2.3 Pre-defined Architectural Performance Events

...

A processor that supports architectural performance monitoring may not support all the predefined architectural performance events (Table 30-1). The non-zero bits in CPUID.0AH:EBX indicate the events that are not available.

...

## 30.6 PERFORMANCE MONITORING FOR PROCESSORS BASED ON INTEL® MICROARCHITECTURE (NEHALEM)

Intel Core i7 processor family[1] supports architectural performance monitoring capability with version ID 3 (see Section 30.2.2.2) and a host of non-architectural monitoring capabilities. The Intel Core i7 processor family is based on Intel® Microarchitecture (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.

...

#### 30.6.1.1 Precise 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_ENABLES[7] and IA32_MISC_ENABLES[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 is shown in Figure 30-13.

...

**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 30-10. In addition to using IA32_PERFEVTSELx to specify event unit/mask settings and setting the EN_PMCx bit in

1. Intel Xeon processor 5500 series and 3400 series are also based on Intel microarchitecture (Nehalem), so the performance monitoring facilities described in this section generally also apply.

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.

...

### 30.14.1 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 GBSQ 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.

...

## 13. Updates to Appendix A, Volume 3B

Change bars show changes to Appendix A of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B:* System Programming Guide, Part 2.

------------------------------------------------------------------------------------------

...

## A.2 PERFORMANCE MONITORING EVENTS FOR INTEL® CORE™I7 PROCESSOR FAMILY AND XEON PROCESSOR FAMILY

Processors based on the Intel microarchitecture (Nehalem) support the architectural and non-architectural performance-monitoring events listed in Table A-1 and Table A-2. The events in Table A-2 generally applies to processors with CPUID signature of DisplayFamily_DisplayModel encoding with the following values: 06_1AH, 06_1EH, 06_1FH, and 06_2EH. However, Intel Xeon processors with CPUID signature of DisplayFamily_DisplayModel 06_2EH have a small number of events that are not supported in processors with CPUID signature 06_1AH, 06_1EH, and 06_1FH. These events are noted in the comment column.

In addition, these processors (CPUID signature of DisplayFamily_DisplayModel 06_1AH, 06_1EH, 06_1FH) also support the following non-architectural, product-specific uncore performance-monitoring events listed in Table A-3.

Fixed counters in the core PMU support the architecture events defined in Table A-7.

**Table A-2 Non-Architectural Performance Events In the Processor Core for Intel Core i7 Processor and Intel Xeon Processor 5500 Series**

| Event Num. | Umask Value | Event Mask Mnemonic | Description | Comment |
|---|---|---|---|---|
| 04H | 07H | SB_DRAIN.ANY | Counts the number of store buffer drains. | |
| ... | | | | |
| 0FH | 01H | MEM_UNCORE_RETIRED.L3_DATA_MISS_UNKNOWN | Counts number of memory load instructions retired where the memory reference missed L3 and data source is unknown. | Available only for CPUID signature 06_2EH |
| ... | | | | |
| 0FH | 80H | MEM_UNCORE_RETIRED.UNCACHEABLE | Counts number of memory load instructions retired where the memory reference missed the L1, L2 and L3 caches and to perform I/O. | Available only for CPUID signature 06_2EH |
| ... | | | | |
| B1H | 1FH | UOPS_EXECUTED.CORE_ACTIVE_CYCLES_NO_PORT5 | Counts cycles when the Uops executed were issued from any ports except port 5. Use Cmask=1 for active cycles; Cmask=0 for weighted cycles; Use CMask=1, Invert=1 to count P0-4 stalled cycles Use Cmask=1, Edge=1, Invert=1 to count P0-4 stalls. | |
| ... | | | | |
| B1H | 3FH | UOPS_EXECUTED.CORE_ACTIVE_CYCLES | Counts cycles when the Uops are executing . Use Cmask=1 for active cycles; Cmask=0 for weighted cycles; Use CMask=1, Invert=1 to count P0-4 stalled cycles Use Cmask=1, Edge=1, Invert=1 to count P0-4 stalls. | |
| ... | | | | |
| B7H | 01H | OFF_CORE_RESPONSE_0 | see Section 30.6.1.3, “Off-core Response Performance Monitoring in the Processor Core” | Requires programming MSR 01A6H |
| ... | | | | |
| BBH | 01H | OFF_CORE_RESPONSE_1 | see Section 30.6.1.3, “Off-core Response Performance Monitoring in the Processor Core” | Requires programming MSR 01A7H |
| ... | | | | |

Non-architectural Performance monitoring events that are located in the uncore sub-system are implementation specific between different platforms using processors based on Intel microarchitecture (Nehalem). Processors with CPUID signature of DisplayFamily_DisplayModel 06_1AH, 06_1EH, and 06_1FH support performance events listed in Table A-3.

**Table A-3 Non-Architectural Performance Events In the Processor Uncore for Intel Core i7 Processor and Intel Xeon Processor 5500 Series**

| Event Num. | Umask Value | Event Mask Mnemonic | Description | Comment |
|---|---|---|---|---|
| ... | | | | |

Intel Xeon processors with CPUID signature of DisplayFamily_DisplayModel 06_2EH have a distinct uncore sub-system that is significantly different from the uncore found in processors with CPUID signature 06_1AH, 06_1EH, and 06_1FH. Non-architectural Performance monitoring events for its uncore will be available in future documentation.

...

**Table A-4 Non-Architectural Performance Events In Next Generation Processor Core (Codenamed Westmere)**

| Event Num. | Umask Value | Event Mask Mnemonic | Description | Comment |
|---|---|---|---|---|
| ... | | | | |
| 0FH | 10H | MEM_UNCORE_RETIRED.LOCAL_DRAM | Load instructions retired with a data source of local DRAM or locally homed remote cache HITM (Precise Event) | |
| ... | | | | |
| 20H | 01H | LSD_OVERFLOW | Number of loops that can not stream from the instruction queue. | |
| ... | | | | |
| 24H | 0CH | L2_RQSTS.RFOS | Counts all L2 store RFO requests. L2 RFO requests include both L1D demand RFO misses as well as L1D RFO prefetches.. | |
| ... | | | | |
| B1H | 1FH | UOPS_EXECUTED.CORE_ACTIVE_CYCLES_NO_PORT5 | Counts number of cycles there are one or more uops being executed and were issued on ports 0-4. This is a core count only and can not be collected per thread. | |
| ... | | | | |

## Non-Architectural Performance Events In Next Generation Processor Core (Codenamed Westmere) (Continued)

| | | | | |
|---|---|---|---|---|
| B1H | 3FH | UOPS_EXECUTED.CORE_ACTIVE_CYCLES | Counts number of cycles there are one or more uops being executed on any ports. This is a core count only and can not be collected per thread. | |
| ... | | | | |
| B7H | 01H | OFF_CORE_RESPONSE_0 | see Section 30.6.1.3, "Off-core Response Performance Monitoring in the Processor Core" | Requires programming MSR 01A6H |
| ... | | | | |
| ECH | 01H | THREAD_ACTIVE | Counts cycles threads are active. | |
| ... | | | | |

Non-architectural Performance monitoring events of the uncore sub-system for Processors with CPUID signature of DisplayFamily_DisplayModel 06_25H, 06_2CH, and 06_1FH support performance events listed in Table A-5.

### Table A-5 Non-Architectural Performance Events In the Processor Uncore for Next Generation Intel Processor (Codenamed Wesmere)

| Event Num. | Umask Value | Event Mask Mnemonic | Description | Comment |
|---|---|---|---|---|
| ... | | | | |
| 02H | 01H | UNC_GQ_OCCUPANCY.READ_TRACKER | Increments the number of queue entries (code read, data read, and RFOs) in the tread tracker. The GQ read tracker allocate to deallocate occupancy count is divided by the count to obtain the average read tracker latency. | |
| ... | | | | |
| 0CH | 01H | UNC_GQ_SNOOP.GOTO_S | Counts the number of remote snoops that have requested a cache line be set to the S state. | |
| 0CH | 02H | UNC_GQ_SNOOP.GOTO_I | Counts the number of remote snoops that have requested a cache line be set to the I state. | |
| 0CH | 04H | UNC_GQ_SNOOP.GOTO_S_HIT_E | Counts the number of remote snoops that have requested a cache line be set to the S state from E state. | Requires writing MSR 301H with mask = 2H |
| 0CH | 04H | UNC_GQ_SNOOP.GOTO_S_HIT_F | Counts the number of remote snoops that have requested a cache line be set to the S state from F (forward) state. | Requires writing MSR 301H with mask = 8H |

| | | | | |
|---|---|---|---|---|
| 0CH | 04H | UNC_GQ_SNOOP.GOTO_S_HIT_M | Counts the number of remote snoops that have requested a cache line be set to the S state from M state. | Requires writing MSR 301H with mask = 1H |
| 0CH | 04H | UNC_GQ_SNOOP.GOTO_S_HIT_S | Counts the number of remote snoops that have requested a cache line be set to the S state from S state. | Requires writing MSR 301H with mask = 4H |
| 0CH | 08H | UNC_GQ_SNOOP.GOTO_I_HIT_E | Counts the number of remote snoops that have requested a cache line be set to the I state from E state. | Requires writing MSR 301H with mask = 2H |
| 0CH | 08H | UNC_GQ_SNOOP.GOTO_I_HIT_F | Counts the number of remote snoops that have requested a cache line be set to the I state from F (forward) state. | Requires writing MSR 301H with mask = 8H |
| 0CH | 08H | UNC_GQ_SNOOP.GOTO_I_HIT_M | Counts the number of remote snoops that have requested a cache line be set to the I state from M state. | Requires writing MSR 301H with mask = 1H |
| 0CH | 08H | UNC_GQ_SNOOP.GOTO_I_HIT_S | Counts the number of remote snoops that have requested a cache line be set to the I state from S state. | Requires writing MSR 301H with mask = 4H |
| ... | | | | |
| 2AH | 07H | UNC_QMC_OCCUPANCY.ANY | Normal read request occupancy for any channel. | |
| ... | | | | |
| 32H | 01H | UNC_IMC_RETRY.CH0 | Counts number of IMC DRAM channel 0 retries. DRAM retry only occurs when configured in RAS mode. | |
| 32H | 02H | UNC_IMC_RETRY.CH1 | Counts number of IMC DRAM channel 1 retries. DRAM retry only occurs when configured in RAS mode. | |
| 32H | 04H | UNC_IMC_RETRY.CH2 | Counts number of IMC DRAM channel 2 retries. DRAM retry only occurs when configured in RAS mode. | |
| 32H | 07H | UNC_IMC_RETRY.ANY | Counts number of IMC DRAM retries from any channel. DRAM retry only occurs when configured in RAS mode. | |
| 33H | 01H | UNC_QHL_FRC_ACK_CNFLTS.IOH | Counts number of Force Acknowledge Conflict messages sent by the Quickpath Home Logic to the IOH. | |
| 33H | 02H | UNC_QHL_FRC_ACK_CNFLTS.REMOTE | Counts number of Force Acknowledge Conflict messages sent by the Quickpath Home Logic to the remote home. | |
| ... | | | | |

| | | | | |
|---|---|---|---|---|
| 33H | 07H | UNC_QHL_FRC_ACK_CNFLTS.ANY | Counts number of Force Acknowledge Conflict messages sent by the Quickpath Home Logic. | |
| 34H | 01H | UNC_QHL_SLEEPS.IOH_ORDER | Counts number of occurrences a request was put to sleep due to IOH ordering (write after read) conflicts. While in the sleep state, the request is not eligible to be scheduled to the QMC | |
| 34H | 02H | UNC_QHL_SLEEPS.REMOTE_ORDER | Counts number of occurrences a request was put to sleep due to remote socket ordering (write after read) conflicts. While in the sleep state, the request is not eligible to be scheduled to the QMC | |
| 34H | 04H | UNC_QHL_SLEEPS.LOCAL_ORDER | Counts number of occurrences a request was put to sleep due to local socket ordering (write after read) conflicts. While in the sleep state, the request is not eligible to be scheduled to the QMC | |
| 34H | 08H | UNC_QHL_SLEEPS.IOH_CONFLICT | Counts number of occurrences a request was put to sleep due to IOH address conflicts. While in the sleep state, the request is not eligible to be scheduled to the QMC | |
| 34H | 10H | UNC_QHL_SLEEPS.REMOTE_CONFLICT | Counts number of occurrences a request was put to sleep due to remote socket address conflicts. While in the sleep state, the request is not eligible to be scheduled to the QMC | |
| 34H | 20H | UNC_QHL_SLEEPS.LOCAL_CONFLICT | Counts number of occurrences a request was put to sleep due to local socket address conflicts. While in the sleep state, the request is not eligible to be scheduled to the QMC | |
| 35H | 01H | UNC_ADDR_OPCODE_MATCH.IOH | Counts number of requests from the IOH, address/opcode of request is qualified by mask value written to MSR 396H. The following mask values are supported:<br>0: NONE<br>40000000_00000000H:RSPFWDI<br>40001A00_00000000H:RSPFWDS<br>40001D00_00000000H:RSPIWB | Match opcode/ address by writing MSR 396H with mask supported mask value |

| | | | | |
|---|---|---|---|---|
| 35H | 02H | UNC_ADDR_OPCODE_MATCH.REMOTE | Counts number of requests from the remote socket, address/opcode of request is qualified by mask value written to MSR 396H. The following mask values are supported:<br>0: NONE<br>40000000_00000000H:RSPFWDI<br>40001A00_00000000H:RSPFWDS<br>40001D00_00000000H:RSPIWB | Match opcode/ address by writing MSR 396H with mask supported mask value |
| 35H | 04H | UNC_ADDR_OPCODE_MATCH.LOCAL | Counts number of requests from the local socket, address/opcode of request is qualified by mask value written to MSR 396H. The following mask values are supported:<br>0: NONE<br>40000000_00000000H:RSPFWDI<br>40001A00_00000000H:RSPFWDS<br>40001D00_00000000H:RSPIWB | Match opcode/ address by writing MSR 396H with mask supported mask value |
| ... | | | | |
| 42H | 01H | UNC_QPI_TX_HEADER.FULL.LINK_0 | Number of cycles that the header buffer in the Quickpath Interface outbound link 0 is full. | |
| ... | | | | |
| 42H | 04H | UNC_QPI_TX_HEADER.FULL.LINK_1 | Number of cycles that the header buffer in the Quickpath Interface outbound link 1 is full. | |
| ... | | | | |
| 67H | 01H | UNC_DRAM_THERMAL_THROTTLED | Uncore cycles DRAM was throttled due to its temperature being above the thermal throttling threshold. | |
| 80H | 01H | UNC_THERMAL_THROTTLING_TEMP.CORE_0 | Cycles that the PCU records that core 0 is above the thermal throttling threshold temperature. | |
| 80H | 02H | UNC_THERMAL_THROTTLING_TEMP.CORE_1 | Cycles that the PCU records that core 1 is above the thermal throttling threshold temperature. | |
| 80H | 04H | UNC_THERMAL_THROTTLING_TEMP.CORE_2 | Cycles that the PCU records that core 2 is above the thermal throttling threshold temperature. | |
| 80H | 08H | UNC_THERMAL_THROTTLING_TEMP.CORE_3 | Cycles that the PCU records that core 3 is above the thermal throttling threshold temperature. | |
| 81H | 01H | UNC_THERMAL_THROTTLED_TEMP.CORE_0 | Cycles that the PCU records that core 0 is in the power throttled state due to core's temperature being above the thermal throttling threshold. | |

| | | | | |
|---|---|---|---|---|
| 81H | 02H | UNC_THERMAL_THROTTLED_TEMP.CORE_1 | Cycles that the PCU records that core 1 is in the power throttled state due to core's temperature being above the thermal throttling threshold. | |
| 81H | 04H | UNC_THERMAL_THROTTLED_TEMP.CORE_2 | Cycles that the PCU records that core 2 is in the power throttled state due to core's temperature being above the thermal throttling threshold. | |
| 81H | 08H | UNC_THERMAL_THROTTLED_TEMP.CORE_3 | Cycles that the PCU records that core 3 is in the power throttled state due to core's temperature being above the thermal throttling threshold. | |
| 82H | 01H | UNC_PROCHOT_ASSERTION | Number of system assertions of PROCHOT indicating the entire processor has exceeded the thermal limit. | |
| 83H | 01H | UNC_THERMAL_THROTTLING_PROCHOT.CORE_0 | Cycles that the PCU records that core 0 is a low power state due to the system asserting PROCHOT the entire processor has exceeded the thermal limit. | |
| 83H | 02H | UNC_THERMAL_THROTTLING_PROCHOT.CORE_1 | Cycles that the PCU records that core 1 is a low power state due to the system asserting PROCHOT the entire processor has exceeded the thermal limit. | |
| 83H | 04H | UNC_THERMAL_THROTTLING_PROCHOT.CORE_2 | Cycles that the PCU records that core 2 is a low power state due to the system asserting PROCHOT the entire processor has exceeded the thermal limit. | |
| 83H | 08H | UNC_THERMAL_THROTTLING_PROCHOT.CORE_3 | Cycles that the PCU records that core 3 is a low power state due to the system asserting PROCHOT the entire processor has exceeded the thermal limit. | |
| 84H | 01H | UNC_TURBO_MODE.CORE_0 | Uncore cycles that core 0 is operating in turbo mode. | |
| 84H | 02H | UNC_TURBO_MODE.CORE_1 | Uncore cycles that core 1 is operating in turbo mode. | |
| 84H | 04H | UNC_TURBO_MODE.CORE_2 | Uncore cycles that core 2 is operating in turbo mode. | |
| 84H | 08H | UNC_TURBO_MODE.CORE_3 | Uncore cycles that core 3 is operating in turbo mode. | |
| 85H | 02H | UNC_CYCLES_UNHALTED_L3_FLL_ENABLE | Uncore cycles that at least one core is unhalted and all L3 ways are enabled. | |

| | | | | |
|---|---|---|---|---|
| 86H | 01H | UNC_CYCLES_UNHALTED_L3_FLL_DISABLE | Uncore cycles that at least one core is unhalted and all L3 ways are disabled. | |
| ... | | | | |

...

**Table A-7 Fixed-Function Performance Counter and Pre-defined Performance Events**

| Fixed-Function Performance Counter | Address | Event Mask Mnemonic | Description |
|---|---|---|---|
| MSR_PERF_FIXED_CTR0/ IA32_PERF_FIXED_CTR0 | 309H | Inst_Retired.Any | This event counts the number of instructions that retire execution. For instructions that consist of multiple micro-ops, this event counts the retirement of the last micro-op of the instruction. The counter continue counting during hardware interrupts, traps, and inside interrupt handlers |
| MSR_PERF_FIXED_CTR1/ IA32_PERF_FIXED_CTR1 | 30AH | CPU_CLK_UNHALTED.CORE | This event counts the number of core cycles while the core is not in a halt state. The core enters the halt state when it is running the HLT instruction. This event is a component in many key event ratios. |
| ... | | | |
| MSR_PERF_FIXED_CTR2/ IA32_PERF_FIXED_CTR2 | 30BH | CPU_CLK_UNHALTED.REF | This event counts the number of reference cycles when the core is not in a halt state and not in a TM stop-clock state. The core enters the halt state when it is running the HLT instruction or the MWAIT instruction. |
| ... | | | |

...

## 14. Updates to Appendix B, Volume 3B

Change bars show changes to Appendix B of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B:* System Programming Guide, Part 2.

------------------------------------------------------------------------------------

...

Table B-1. CPUID Signature Values of DisplayFamily_DisplayModel

| DisplayFamily_DisplayModel | Processor Families/Processor Number Series |
|---|---|
| **06_1AH** | Intel Core i7 Processor, Intel Xeon Processor 5500 series |
| **06_1EH, 06_1FH** | Intel Core i7 and i5 Processor, |
| **06_2EH** | Intel Xeon Processors based on Intel Microarchitecture (Nehalem) |
| **06_25H, 06_2CH** | Next Generation Intel Processor (Westmere) |
| **06_1DH** | Intel Xeon Processor MP 7400 series |
| **06_17H** | Intel Xeon Processor 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_1CH** | Intel Atom processor |
| **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 |

Table B-2. IA-32 Architectural MSRs

| Register Address | | Architectural MSR Name and bit fields (Former MSR Name) | MSR/Bit Description | Introduced as Architectural MSR |
|---|---|---|---|---|
| Hex | Decimal | | | |
| ... | | | | |
| 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_1AH |
| | | 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. | |
| | | 63:25 | Reserved | |
| ... | | | | |
| 1B0H | 432 | IA32_ENERGY_PERF_BIAS | Performance Energy Bias Hint (R/W) | if CPUID.6H:ECX[3] = 1 |
| ... | | | | |
| 280H | 640 | IA32_MC0_CTL2 | (R/W) | 06_1AH |
| | | 14:0 | Corrected error count threshold | |
| | | 29:15 | Reserved | |
| | | 30 | CMCI_EN | |
| | | 63:31 | Reserved | |
| ... | | | | |
| 414H | 1044 | IA32_MC5_CTL | MC5_CTL | 06_0FH |
| 415H | 1045 | IA32_MC5_STATUS | MC5_STATUS | 06_0FH |
| 416H | 1046 | IA32_MC5_ADDR[1] | MC5_ADDR | 06_0FH |

| Register Address | | Architectural MSR Name and bit fields (Former MSR Name) | MSR/Bit Description | Introduced as Architectural MSR |
|---|---|---|---|---|
| Hex | Decimal | | | |
| 417H | 1047 | IA32_MC5_MISC | MC5_MISC | 06_0FH |
| 418H | 1048 | IA32_MC6_CTL | MC6_CTL | 06_1DH |
| 419H | 1049 | IA32_MC6_STATUS | MC6_STATUS | 06_1DH |
| 41AH | 1050 | IA32_MC6_ADDR[1] | MC6_ADDR | 06_1DH |
| 41BH | 1051 | IA32_MC6_MISC | MC6_MISC | 06_1DH |
| 41CH | 1052 | IA32_MC7_CTL | MC7_CTL | 06_1AH |
| 41DH | 1053 | IA32_MC7_STATUS | MC7_STATUS | 06_1AH |
| 41EH | 1054 | IA32_MC7_ADDR[1] | MC7_ADDR | 06_1AH |
| 41FH | 1055 | IA32_MC7_MISC | MC7_MISC | 06_1AH |
| 420H | 1056 | IA32_MC8_CTL | MC8_CTL | 06_1AH |
| 421H | 1057 | IA32_MC8_STATUS | MC8_STATUS | 06_1AH |
| 422H | 1058 | IA32_MC8_ADDR[1] | MC8_ADDR | 06_1AH |
| 423H | 1059 | IA32_MC8_MISC | MC8_MISC | 06_1AH |
| 424H | 1060 | IA32_MC9_CTL | MC9_CTL | 06_2EH |
| 425H | 1061 | IA32_MC9_STATUS | MC9_STATUS | 06_2EH |
| 426H | 1062 | IA32_MC9_ADDR[1] | MC9_ADDR | 06_2EH |
| 427H | 1063 | IA32_MC9_MISC | MC9_MISC | 06_2EH |
| 428H | 1064 | IA32_MC10_CTL | MC10_CTL | 06_2EH |
| 429H | 1065 | IA32_MC10_STATUS | MC10_STATUS | 06_2EH |
| 42AH | 1066 | IA32_MC10_ADDR[1] | MC10_ADDR | 06_2EH |
| 42BH | 1067 | IA32_MC10_MISC | MC10_MISC | 06_2EH |
| 42CH | 1068 | IA32_MC11_CTL | MC11_CTL | 06_2EH |
| 42DH | 1069 | IA32_MC11_STATUS | MC11_STATUS | 06_2EH |
| 42EH | 1070 | IA32_MC11_ADDR[1] | MC11_ADDR | 06_2EH |
| 42FH | 1071 | IA32_MC11_MISC | MC11_MISC | 06_2EH |
| 430H | 1072 | IA32_MC12_CTL | MC12_CTL | 06_2EH |
| 431H | 1073 | IA32_MC12_STATUS | MC12_STATUS | 06_2EH |
| 432H | 1074 | IA32_MC12_ADDR[1] | MC12_ADDR | 06_2EH |
| 433H | 1075 | IA32_MC12_MISC | MC12_MISC | 06_2EH |
| 434H | 1076 | IA32_MC13_CTL | MC13_CTL | 06_2EH |
| 435H | 1077 | IA32_MC13_STATUS | MC13_STATUS | 06_2EH |
| 436H | 1078 | IA32_MC13_ADDR[1] | MC13_ADDR | 06_2EH |
| 437H | 1079 | IA32_MC13_MISC | MC13_MISC | 06_2EH |
| 438H | 1080 | IA32_MC14_CTL | MC14_CTL | 06_2EH |
| 439H | 1081 | IA32_MC14_STATUS | MC14_STATUS | 06_2EH |

| Register Address | | Architectural MSR Name and bit fields (Former MSR Name) | MSR/Bit Description | Introduced as Architectural MSR |
|---|---|---|---|---|
| **Hex** | **Decimal** | | | |
| 43AH | 1082 | IA32_MC14_ADDR[1] | MC14_ADDR | 06_2EH |
| 43BH | 1083 | IA32_MC14_MISC | MC14_MISC | 06_2EH |
| 43CH | 1084 | IA32_MC15_CTL | MC15_CTL | 06_2EH |
| 43DH | 1085 | IA32_MC15_STATUS | MC15_STATUS | 06_2EH |
| 43EH | 1086 | IA32_MC15_ADDR[1] | MC15_ADDR | 06_2EH |
| 43FH | 1087 | IA32_MC15_MISC | MC15_MISC | 06_2EH |
| 440H | 1088 | IA32_MC16_CTL | MC16_CTL | 06_2EH |
| 441H | 1089 | IA32_MC16_STATUS | MC16_STATUS | 06_2EH |
| 442H | 1090 | IA32_MC16_ADDR[1] | MC16_ADDR | 06_2EH |
| 443H | 1091 | IA32_MC16_MISC | MC16_MISC | 06_2EH |
| 444H | 1092 | IA32_MC17_CTL | MC17_CTL | 06_2EH |
| 445H | 1093 | IA32_MC17_STATUS | MC17_STATUS | 06_2EH |
| 446H | 1094 | IA32_MC17_ADDR[1] | MC17_ADDR | 06_2EH |
| 447H | 1095 | IA32_MC17_MISC | MC17_MISC | 06_2EH |
| 448H | 1096 | IA32_MC18_CTL | MC18_CTL | 06_2EH |
| 449H | 1097 | IA32_MC18_STATUS | MC18_STATUS | 06_2EH |
| 44AH | 1098 | IA32_MC18_ADDR[1] | MC18_ADDR | 06_2EH |
| 44BH | 1099 | IA32_MC18_MISC | MC18_MISC | 06_2EH |
| 44CH | 1100 | IA32_MC19_CTL | MC19_CTL | 06_2EH |
| 44DH | 1101 | IA32_MC19_STATUS | MC19_STATUS | 06_2EH |
| 44EH | 1102 | IA32_MC19_ADDR[1] | MC19_ADDR | 06_2EH |
| 44FH | 1103 | IA32_MC19_MISC | MC19_MISC | 06_2EH |
| 450H | 1104 | IA32_MC20_CTL | MC20_CTL | 06_2EH |
| 451H | 1105 | IA32_MC20_STATUS | MC20_STATUS | 06_2EH |
| 452H | 1106 | IA32_MC20_ADDR[1] | MC20_ADDR | 06_2EH |
| 453H | 1107 | IA32_MC20_MISC | MC20_MISC | 06_2EH |
| 454H | 1108 | IA32_MC21_CTL | MC21_CTL | 06_2EH |
| 455H | 1109 | IA32_MC21_STATUS | MC21_STATUS | 06_2EH |
| 456H | 1110 | IA32_MC21_ADDR[1] | MC21_ADDR | 06_2EH |
| 457H | 1111 | IA32_MC21_MISC | MC21_MISC | 06_2EH |
| ... | | | | |

...

Table B-5 MSRs in Processors Based on Intel Microarchitecture (Continued)(Nehalem)

| Register Address | | Register Name | Scope | Bit Description |
|---|---|---|---|---|
| Hex | Dec | | | |
| ... | | | | |
| 1C8H | 456 | MSR_LBR_SELECT | Core | **Last Branch Record Filtering Select Register** (R/W) see Section 16.6.2, "Filtering of Last Branch Records." |
| ... | | | | |
| 3B0H | 960 | MSR_UNCORE_PMC0 | Package | See Section 30.6.2.2, "Uncore Performance Event Configuration Facility." |
| 3B1H | 961 | MSR_UNCORE_PMC1 | Package | See Section 30.6.2.2, "Uncore Performance Event Configuration Facility." |
| 3B2H | 962 | MSR_UNCORE_PMC2 | Package | See Section 30.6.2.2, "Uncore Performance Event Configuration Facility." |
| 3B3H | 963 | MSR_UNCORE_PMC3 | Package | See Section 30.6.2.2, "Uncore Performance Event Configuration Facility." |
| 3B4H | 964 | MSR_UNCORE_PMC4 | Package | See Section 30.6.2.2, "Uncore Performance Event Configuration Facility." |
| 3B5H | 965 | MSR_UNCORE_PMC5 | Package | See Section 30.6.2.2, "Uncore Performance Event Configuration Facility." |
| 3B6H | 966 | MSR_UNCORE_PMC6 | Package | See Section 30.6.2.2, "Uncore Performance Event Configuration Facility." |
| 3B7H | 967 | MSR_UNCORE_PMC7 | Package | See Section 30.6.2.2, "Uncore Performance Event Configuration Facility." |
| 3C0H | 944 | MSR_UNCORE_PERFEVTSEL0 | Package | See Section 30.6.2.2, "Uncore Performance Event Configuration Facility." |
| 3C1H | 945 | MSR_UNCORE_PERFEVTSEL1 | Package | See Section 30.6.2.2, "Uncore Performance Event Configuration Facility." |
| 3C2H | 946 | MSR_UNCORE_PERFEVTSEL2 | Package | See Section 30.6.2.2, "Uncore Performance Event Configuration Facility." |
| 3C3H | 947 | MSR_UNCORE_PERFEVTSEL3 | Package | See Section 30.6.2.2, "Uncore Performance Event Configuration Facility." |
| 3C4H | 948 | MSR_UNCORE_PERFEVTSEL4 | Package | See Section 30.6.2.2, "Uncore Performance Event Configuration Facility." |
| 3C5H | 949 | MSR_UNCORE_PERFEVTSEL5 | Package | See Section 30.6.2.2, "Uncore Performance Event Configuration Facility." |
| 3C6H | 950 | MSR_UNCORE_PERFEVTSEL6 | Package | See Section 30.6.2.2, "Uncore Performance Event Configuration Facility." |
| 3C7H | 951 | MSR_UNCORE_PERFEVTSEL7 | Package | See Section 30.6.2.2, "Uncore Performance Event Configuration Facility." |
| ... | | | | |
| 403H | 1027 | MSR_MC0_MISC | Package | See Section 15.3.2.4, "IA32_MCi_MISC MSRs." |
| ... | | | | |

| Register Address | | Register Name | Scope | Bit Description |
|---|---|---|---|---|
| Hex | Dec | | | |
| 407H | 1031 | MSR_MC1_MISC | Package | See Section 15.3.2.4, "IA32_MCi_MISC MSRs." |
| ... | | | | |
| 40BH | 1035 | MSR_MC2_MISC | Core | See Section 15.3.2.4, "IA32_MCi_MISC MSRs." |
| 40CH | 1036 | MSR_MC3_CTL | Core | See Section 15.3.2.1, "IA32_MCi_CTL MSRs." |
| 40DH | 1037 | MSR_MC3_ STATUS | Core | See Section 15.3.2.2, "IA32_MCi_STATUS MSRS." |
| 40EH | 1038 | MSR_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 ADDRV 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 | MSR_MC3_MISC | Core | See Section 15.3.2.4, "IA32_MCi_MISC MSRs." |
| 410H | 1040 | MSR_MC4_CTL | Core | See Section 15.3.2.1, "IA32_MCi_CTL MSRs." |
| 411H | 1041 | MSR_MC4_ STATUS | Core | See Section 15.3.2.2, "IA32_MCi_STATUS MSRS." |
| 412H | 1042 | MSR_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 ADDRV 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_MC4_MISC | Core | See Section 15.3.2.4, "IA32_MCi_MISC MSRs." |
| 414H | 1044 | MSR_MC5_CTL | Core | See Section 15.3.2.1, "IA32_MCi_CTL MSRs." |
| 415H | 1045 | MSR_MC5_ STATUS | Core | See Section 15.3.2.2, "IA32_MCi_STATUS MSRS." |
| 416H | 1046 | MSR_MC5_ADDR | Core | See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." |
| 417H | 1047 | MSR_MC5_MISC | Core | See Section 15.3.2.4, "IA32_MCi_MISC MSRs." |
| 418H | 1048 | MSR_MC6_CTL | Package | See Section 15.3.2.1, "IA32_MCi_CTL MSRs." |
| 419H | 1049 | MSR_MC6_ STATUS | Package | See Section 15.3.2.2, "IA32_MCi_STATUS MSRS." and Appendix E. |
| 41AH | 1050 | MSR_MC6_ADDR | Package | See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." |
| 41BH | 1051 | MSR_MC6_MISC | Package | See Section 15.3.2.4, "IA32_MCi_MISC MSRs." |
| 41CH | 1052 | MSR_MC7_CTL | Package | See Section 15.3.2.1, "IA32_MCi_CTL MSRs." |
| 41DH | 1053 | MSR_MC7_ STATUS | Package | See Section 15.3.2.2, "IA32_MCi_STATUS MSRS." and Appendix E. |

| Register Address | | Register Name | Scope | Bit Description |
|---|---|---|---|---|
| Hex | Dec | | | |
| 41EH | 1054 | MSR_MC7_ADDR | Package | See Section 15.3.2.3, “IA32_MCi_ADDR MSRs.” |
| 41FH | 1055 | MSR_MC7_MISC | Package | See Section 15.3.2.4, “IA32_MCi_MISC MSRs.” |
| 420H | 1056 | MSR_MC8_CTL | Package | See Section 15.3.2.1, “IA32_MCi_CTL MSRs.” |
| 421H | 1057 | MSR_MC8_ STATUS | Package | See Section 15.3.2.2, “IA32_MCi_STATUS MSRS.” and Appendix E. |
| 422H | 1058 | MSR_MC8_ADDR | Package | See Section 15.3.2.3, “IA32_MCi_ADDR MSRs.” |
| 423H | 1059 | MSR_MC8_MISC | Package | See Section 15.3.2.4, “IA32_MCi_MISC MSRs.” |
| 424H | 1060 | MSR_MC9_CTL | Package | See Section 15.3.2.1, “IA32_MCi_CTL MSRs.” |
| 425H | 1061 | MSR_MC9_ STATUS | Package | See Section 15.3.2.2, “IA32_MCi_STATUS MSRS.” and Appendix E. |
| 426H | 1062 | MSR_MC9_ADDR | Package | See Section 15.3.2.3, “IA32_MCi_ADDR MSRs.” |
| 427H | 1063 | MSR_MC9_MISC | Package | See Section 15.3.2.4, “IA32_MCi_MISC MSRs.” |
| 428H | 1064 | MSR_MC10_CTL | Package | See Section 15.3.2.1, “IA32_MCi_CTL MSRs.” |
| 429H | 1065 | MSR_MC10_ STATUS | Package | See Section 15.3.2.2, “IA32_MCi_STATUS MSRS.” and Appendix E. |
| 42AH | 1066 | MSR_MC10_ADDR | Package | See Section 15.3.2.3, “IA32_MCi_ADDR MSRs.” |
| 42BH | 1067 | MSR_MC10_MISC | Package | See Section 15.3.2.4, “IA32_MCi_MISC MSRs.” |
| 42CH | 1068 | MSR_MC11_CTL | Package | See Section 15.3.2.1, “IA32_MCi_CTL MSRs.” |
| 42DH | 1069 | MSR_MC11_ STATUS | Package | See Section 15.3.2.2, “IA32_MCi_STATUS MSRS.” and Appendix E. |
| 42EH | 1070 | MSR_MC11_ADDR | Package | See Section 15.3.2.3, “IA32_MCi_ADDR MSRs.” |
| 42FH | 1071 | MSR_MC11_MISC | Package | See Section 15.3.2.4, “IA32_MCi_MISC MSRs.” |
| 430H | 1072 | MSR_MC12_CTL | Package | See Section 15.3.2.1, “IA32_MCi_CTL MSRs.” |
| 431H | 1073 | MSR_MC12_ STATUS | Package | See Section 15.3.2.2, “IA32_MCi_STATUS MSRS.” and Appendix E. |
| 432H | 1074 | MSR_MC12_ADDR | Package | See Section 15.3.2.3, “IA32_MCi_ADDR MSRs.” |
| 433H | 1075 | MSR_MC12_MISC | Package | See Section 15.3.2.4, “IA32_MCi_MISC MSRs.” |
| 434H | 1076 | MSR_MC13_CTL | Package | See Section 15.3.2.1, “IA32_MCi_CTL MSRs.” |
| 435H | 1077 | MSR_MC13_ STATUS | Package | See Section 15.3.2.2, “IA32_MCi_STATUS MSRS.” and Appendix E. |
| 436H | 1078 | MSR_MC13_ADDR | Package | See Section 15.3.2.3, “IA32_MCi_ADDR MSRs.” |
| 437H | 1079 | MSR_MC13_MISC | Package | See Section 15.3.2.4, “IA32_MCi_MISC MSRs.” |
| 438H | 1080 | MSR_MC14_CTL | Package | See Section 15.3.2.1, “IA32_MCi_CTL MSRs.” |
| 439H | 1081 | MSR_MC14_ STATUS | Package | See Section 15.3.2.2, “IA32_MCi_STATUS MSRS.” and Appendix E. |
| 43AH | 1082 | MSR_MC14_ADDR | Package | See Section 15.3.2.3, “IA32_MCi_ADDR MSRs.” |
| 43BH | 1083 | MSR_MC14_MISC | Package | See Section 15.3.2.4, “IA32_MCi_MISC MSRs.” |

| Register Address | | Register Name | Scope | Bit Description |
|---|---|---|---|---|
| Hex | Dec | | | |
| 43CH | 1084 | MSR_MC15_CTL | Package | See Section 15.3.2.1, “IA32_MCi_CTL MSRs.” |
| 43DH | 1085 | MSR_MC15_STATUS | Package | See Section 15.3.2.2, “IA32_MCi_STATUS MSRS.” and Appendix E. |
| 43EH | 1086 | MSR_MC15_ADDR | Package | See Section 15.3.2.3, “IA32_MCi_ADDR MSRs.” |
| 43FH | 1087 | MSR_MC15_MISC | Package | See Section 15.3.2.4, “IA32_MCi_MISC MSRs.” |
| 440H | 1088 | MSR_MC16_CTL | Package | See Section 15.3.2.1, “IA32_MCi_CTL MSRs.” |
| 441H | 1089 | MSR_MC16_STATUS | Package | See Section 15.3.2.2, “IA32_MCi_STATUS MSRS.” and Appendix E. |
| 442H | 1090 | MSR_MC16_ADDR | Package | See Section 15.3.2.3, “IA32_MCi_ADDR MSRs.” |
| 443H | 1091 | MSR_MC16_MISC | Package | See Section 15.3.2.4, “IA32_MCi_MISC MSRs.” |
| 444H | 1092 | MSR_MC17_CTL | Package | See Section 15.3.2.1, “IA32_MCi_CTL MSRs.” |
| 445H | 1093 | MSR_MC17_STATUS | Package | See Section 15.3.2.2, “IA32_MCi_STATUS MSRS.” and Appendix E. |
| 446H | 1094 | MSR_MC17_ADDR | Package | See Section 15.3.2.3, “IA32_MCi_ADDR MSRs.” |
| 447H | 1095 | MSR_MC17_MISC | Package | See Section 15.3.2.4, “IA32_MCi_MISC MSRs.” |
| 448H | 1096 | MSR_MC18_CTL | Package | See Section 15.3.2.1, “IA32_MCi_CTL MSRs.” |
| 449H | 1097 | MSR_MC18_STATUS | Package | See Section 15.3.2.2, “IA32_MCi_STATUS MSRS.” and Appendix E. |
| 44AH | 1098 | MSR_MC18_ADDR | Package | See Section 15.3.2.3, “IA32_MCi_ADDR MSRs.” |
| 44BH | 1099 | MSR_MC18_MISC | Package | See Section 15.3.2.4, “IA32_MCi_MISC MSRs.” |
| 44CH | 1100 | MSR_MC19_CTL | Package | See Section 15.3.2.1, “IA32_MCi_CTL MSRs.” |
| 44DH | 1101 | MSR_MC19_STATUS | Package | See Section 15.3.2.2, “IA32_MCi_STATUS MSRS.” and Appendix E. |
| 44EH | 1102 | MSR_MC19_ADDR | Package | See Section 15.3.2.3, “IA32_MCi_ADDR MSRs.” |
| 44FH | 1103 | MSR_MC19_MISC | Package | See Section 15.3.2.4, “IA32_MCi_MISC MSRs.” |
| 450H | 1104 | MSR_MC20_CTL | Package | See Section 15.3.2.1, “IA32_MCi_CTL MSRs.” |
| 451H | 1105 | MSR_MC20_STATUS | Package | See Section 15.3.2.2, “IA32_MCi_STATUS MSRS.” and Appendix E. |
| 452H | 1106 | MSR_MC20_ADDR | Package | See Section 15.3.2.3, “IA32_MCi_ADDR MSRs.” |
| 453H | 1107 | MSR_MC20_MISC | Package | See Section 15.3.2.4, “IA32_MCi_MISC MSRs.” |
| 454H | 1108 | MSR_MC21_CTL | Package | See Section 15.3.2.1, “IA32_MCi_CTL MSRs.” |
| 455H | 1109 | MSR_MC21_STATUS | Package | See Section 15.3.2.2, “IA32_MCi_STATUS MSRS.” and Appendix E. |
| 456H | 1110 | MSR_MC21_ADDR | Package | See Section 15.3.2.3, “IA32_MCi_ADDR MSRs.” |
| 457H | 1111 | MSR_MC21_MISC | Package | See Section 15.3.2.4, “IA32_MCi_MISC MSRs.” |
| ... | | | | |

## B-5 MSRS IN THE NEXT GENERATION INTEL PROCESSOR (CODENAMED WESMERE)

Next Generation Intel 64 processors (codenamed Wesmere) supports the MSR interfaces listed in Table B-5, plus additional MSR listed in Table B-6.

**Table B-6 Additional MSRs supported by Next Generation Intel Processors (Codenamed Wesmere)**

| Register Address | | Register Name | Scope | Bit Description |
|---|---|---|---|---|
| **Hex** | **Dec** | | | |
| 1A7H | 423 | MSR_OFFCORE_RSP1 | Thread | Offcore Response Event Select Register (R/W) |
| 1B0H | 432 | IA32_ENERGY_PERF_BIAS | Package | see Table B-2. |

...

## 15. Updates to Appendix G, Volume 3B

Change bars show changes to Appendix G of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B:* System Programming Guide, Part 2.

------------------------------------------------------------------------------------------

...

## G.10 VPID AND EPT CAPABILITIES

The IA32_VMX_EPT_VPID_CAP MSR (index 48CH) reports information about the capabilities of the logical processor with regard to virtual-processor identifiers (VPIDs, Section 25.1) and extended page tables (EPT, Section 25.2):

- If bit 0 is read as 1, the logical processor allows software to configure EPT paging-structure entries in which bits 2:0 have value 100b (indicating an execute-only translation).
- Bit 6 indicates support for a page-walk length of 4.
- If bit 8 is read as 1, the logical processor allows software to configure the EPT paging-structure memory type to be uncacheable (UC); see Section 21.6.11.
- If bit 14 is read as 1, the logical processor allows software to configure the EPT paging-structure memory type to be write-back (WB).
- If bit 16 is read as 1, the logical processor allows software to configure a EPT PDE to map a 2-Mbyte page (by setting bit 7 in the EPT PDE).
- If bit 17 is read as 1, the logical processor allows software to configure a EPT PDPTE to map a 1-Gbyte page (by setting bit 7 in the EPT PDPTE).
- Support for the INVEPT instruction (see Chapter 6 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A* and Section 25.3.3.1).
  - — If bit 20 is read as 1, the INVEPT instruction is supported.
  - — If bit 25 is read as 1, the single-context INVEPT type is supported.
  - — If bit 26 is read as 1, the all-context INVEPT type is supported.