

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

## Volume 2C: Instruction Set Reference

**NOTE:** The *Intel® 64 and IA-32 Architectures Software Developer's Manual* consists of seven volumes: *Basic Architecture*, Order Number 253665; *Instruction Set Reference A-L*, Order Number 253666; *Instruction Set Reference M-Z*, Order Number 253667; *Instruction Set Reference*, Order Number 326018; *System Programming Guide, Part 1*, Order Number 253668; *System Programming Guide, Part 2*, Order Number 253669; *System Programming Guide, Part 3*, Order Number 326019. Refer to all seven volumes when evaluating your design needs.

Order Number: 326018-040US  
October 2011

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

UNLESS OTHERWISE AGREED IN WRITING BY INTEL, THE INTEL PRODUCTS ARE NOT DESIGNED NOR INTENDED FOR ANY APPLICATION IN WHICH THE FAILURE OF THE INTEL PRODUCT COULD CREATE A SITUATION WHERE PERSONAL INJURY OR DEATH MAY OCCUR.

Intel may make changes to specifications and product descriptions at any time, without notice. Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined." Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them. The information here is subject to change without notice. Do not finalize a design with this information.

The Intel® 64 architecture processors may contain design defects or errors known as errata. Current characterized errata are available on request.

Intel® Hyper-Threading Technology requires a computer system with an Intel® processor supporting Hyper-Threading Technology and an Intel® HT Technology enabled chipset, BIOS and operating system. Performance will vary depending on the specific hardware and software you use. For more information, see <http://www.intel.com/technology/hyperthread/index.htm>; including details on which processors support Intel HT Technology.

Intel® Virtualization Technology requires a computer system with an enabled Intel® processor, BIOS, virtual machine monitor (VMM) and for some uses, certain platform software enabled for it. Functionality, performance or other benefits will vary depending on hardware and software configurations. Intel® Virtualization Technology-enabled BIOS and VMM applications are currently in development.

64-bit computing on Intel architecture requires a computer system with a processor, chipset, BIOS, operating system, device drivers and applications enabled for Intel® 64 architecture. Processors will not operate (including 32-bit operation) without an Intel® 64 architecture-enabled BIOS. Performance will vary depending on your hardware and software configurations. Consult with your system vendor for more information.

Enabling Execute Disable Bit functionality requires a PC with a processor with Execute Disable Bit capability and a supporting operating system. Check with your PC manufacturer on whether your system delivers Execute Disable Bit functionality.

Intel, Pentium, Intel Xeon, Intel NetBurst, Intel Core, Intel Core Solo, Intel Core Duo, Intel Core 2 Duo, Intel Core 2 Extreme, Intel Pentium D, Itanium, Intel SpeedStep, MMX, Intel Atom, and VTune are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

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

Contact your local Intel sales office or your distributor to obtain the latest specifications and before placing your product order.

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

Copyright © 1997-2011 Intel Corporation

# CHAPTER 5

## VMX INSTRUCTION REFERENCE

---

### NOTE

For the current document revision, this chapter is located here as Chapter 5, “VMX Instruction Reference,” and duplicated in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3C* as Chapter 33, “VMX Instruction Reference.”

For all future document revisions, this chapter will be located in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3C* as Chapter 33, “VMX Instruction Reference,” and will not be duplicated elsewhere.

## 5.1 OVERVIEW

This chapter describes the virtual-machine extensions (VMX) for the Intel 64 and IA-32 architectures. VMX is intended to support virtualization of processor hardware and a system software layer acting as a host to multiple guest software environments. The virtual-machine extensions (VMX) includes five instructions that manage the virtual-machine control structure (VMCS), four instructions that manage VMX operation, two TLB-management instructions, and two instructions for use by guest software. Additional details of VMX are described in *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3C*.

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

- **VMPTRLD** — This instruction takes a single 64-bit source operand that is in memory. It makes the referenced VMCS active and current, loading the current-VMCS pointer with this operand and establishes the current VMCS based on the contents of VMCS-data area in the referenced VMCS region. Because this makes the referenced VMCS active, a logical processor may start maintaining on the processor some of the VMCS data for the VMCS.
- **VMPTRST** — This instruction takes a single 64-bit destination operand that is in memory. The current-VMCS pointer is stored into the destination operand.
- **VMCLEAR** — This instruction takes a single 64-bit operand that is in memory. The instruction sets the launch state of the VMCS referenced by the operand to “clear”, renders that VMCS inactive, and ensures that data for the VMCS have been written to the VMCS-data area in the referenced VMCS region. If the operand is the same as the current-VMCS pointer, that pointer is made invalid.
- **VMREAD** — This instruction reads a component from the VMCS (the encoding of that field is given in a register operand) and stores it into a destination operand that may be a register or in memory.

- **VMWRITE** — This instruction writes a component to the VMCS (the encoding of that field is given in a register operand) from a source operand that may be a register or in memory.

The behavior of the VMX management instructions is summarized below:

- **VMLAUNCH** — This instruction launches a virtual machine managed by the VMCS. A VM entry occurs, transferring control to the VM.
- **VMRESUME** — This instruction resumes a virtual machine managed by the VMCS. A VM entry occurs, transferring control to the VM.
- **VMXOFF** — This instruction causes the processor to leave VMX operation.
- **VMXON** — This instruction takes a single 64-bit source operand that is in memory. It causes a logical processor to enter VMX root operation and to use the memory referenced by the operand to support VMX operation.

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

- **INVEPT** — This instruction invalidates entries in the TLBs and paging-structure caches that were derived from extended page tables (EPT).
- **INVVPID** — This instruction invalidates entries in the TLBs and paging-structure caches based on a Virtual-Processor Identifier (VPID).

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

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

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

## 5.2 CONVENTIONS

The operation sections for the VMX instructions in Section 5.3 use the pseudo-function VMexit, which indicates that the logical processor performs a VM exit.

The operation sections also use the pseudo-functions VMsucceed, VMfail, VMfailInvalid, and VMfailValid. These pseudo-functions signal instruction success or failure by setting or clearing bits in RFLAGS and, in some cases, by writing the VM-instruction error field. The following pseudocode fragments detail these functions:

```
VMsucceed:  
    CF ← 0;  
    PF ← 0;  
    AF ← 0;
```

```
ZF ← 0;
SF ← 0;
OF ← 0;
```

```
VMfail(ErrorNumber):
  IF VMCS pointer is valid
    THEN VMfailValid(ErrorNumber);
    ELSE VMfailInvalid;
  FI;
```

```
VMfailInvalid:
  CF ← 1;
  PF ← 0;
  AF ← 0;
  ZF ← 0;
  SF ← 0;
  OF ← 0;
```

```
VMfailValid(ErrorNumber):// executed only if there is a current VMCS
  CF ← 0;
  PF ← 0;
  AF ← 0;
  ZF ← 1;
  SF ← 0;
  OF ← 0;
  Set the VM-instruction error field to ErrorNumber;
```

The different VM-instruction error numbers are enumerated in Section 5.4, “VM Instruction Error Numbers”.

## 5.3 VMX INSTRUCTIONS

This section provides detailed descriptions of the VMX instructions.

## INVEPT— Invalidate Translations Derived from EPT

Opcode	Instruction	Description
66 0F 38 80	INVEPT r64, m128	Invalidates EPT-derived entries in the TLBs and paging-structure caches (in 64-bit mode)
66 0F 38 80	INVEPT r32, m128	Invalidates EPT-derived entries in the TLBs and paging-structure caches (outside 64-bit mode)

### Description

Invalidates mappings in the translation lookaside buffers (TLBs) and paging-structure caches that were derived from extended page tables (EPT). (See Chapter 28, “VMX Support for Address Translation” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3C*.) Invalidation is based on the **INVEPT type** specified in the register operand and the **INVEPT descriptor** specified in the memory operand.

Outside IA-32e mode, the register operand is always 32 bits, regardless of the value of CS.D; in 64-bit mode, the register operand has 64 bits (the instruction cannot be executed in compatibility mode).

The INVEPT types supported by a logical processor are reported in the IA32\_VMX\_EPT\_VPID\_CAP MSR (see Appendix “VMX Capability Reporting Facility” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3C*). There are two INVEPT types currently defined:

- Single-context invalidation. If the INVEPT type is 1, the logical processor invalidates all mappings associated with bits 51:12 of the EPT pointer (EPTP) specified in the INVEPT descriptor. It may invalidate other mappings as well.
- Global invalidation: If the INVEPT type is 2, the logical processor invalidates mappings associated with all EPTPs.

If an unsupported INVEPT type is specified, the instruction fails.

INVEPT invalidates all the specified mappings for the indicated EPTP(s) regardless of the VPID and PCID values with which those mappings may be associated.

The INVEPT descriptor comprises 128 bits and contains a 64-bit EPTP value in bits 63:0 (see Figure 5-1).

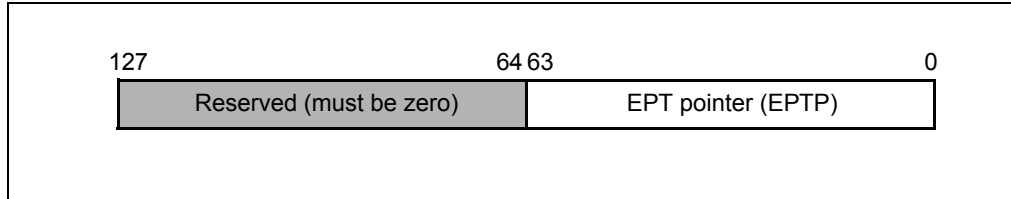


Figure 5-1. INVEPT Descriptor

## Operation

```

IF (not in VMX operation) or (CR0.PE = 0) or (RFLAGS.VM = 1) or (IA32_EFER.LMA = 1 and CS.L = 0)
    THEN #UD;
ELSIF in VMX non-root operation
    THEN VM exit;
ELSIF CPL > 0
    THEN #GP(0);
ELSE
    INVEPT_TYPE ← value of register operand;
    IF IA32_VMX_EPT_VPID_CAP MSR indicates that processor does not support INVEPT_TYPE
        THEN VMfail(Invalid operand to INVEPT/INVVPID);
    ELSE // INVEPT_TYPE must be 1 or 2
        INVEPT_DESC ← value of memory operand;
        EPTP ← INVEPT_DESC[63:0];
        CASE INVEPT_TYPE OF
            1: // single-context invalidation
                IF VM entry with the "enable EPT" VM execution control set to 1
                    would fail due to the EPTP value
                    THEN VMfail(Invalid operand to INVEPT/INVVPID);
                ELSE
                    Invalidate mappings associated with EPTP[51:12];
                    VMSucceed;
            FI;
            BREAK;
            2: // global invalidation
                Invalidate mappings associated with all EPTPs;
                VMSucceed;
                BREAK;
        ESAC;
    FI;
FI;

```

## Flags Affected

See the operation section and Section 5.2.

## Protected Mode Exceptions

#GP(0)	<p>If the current privilege level is not 0.</p> <p>If the memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.</p> <p>If the DS, ES, FS, or GS register contains an unusable segment.</p> <p>If the source operand is located in an execute-only code segment.</p>
#PF(fault-code)	If a page fault occurs in accessing the memory operand.
#SS(0)	<p>If the memory operand effective address is outside the SS segment limit.</p> <p>If the SS register contains an unusable segment.</p>
#UD	<p>If not in VMX operation.</p> <p>If the logical processor does not support EPT (IA32_VMX_PROCBASED_CTLDS2[33]=0).</p> <p>If the logical processor supports EPT (IA32_VMX_PROCBASED_CTLDS2[33]=1) but does not support the INVEPT instruction (IA32_VMX_EPT_VPID_CAP[20]=0).</p>

## Real-Address Mode Exceptions

#UD	A logical processor cannot be in real-address mode while in VMX operation and the INVEPT instruction is not recognized outside VMX operation.
-----	---

## Virtual-8086 Mode Exceptions

#UD	The INVEPT instruction is not recognized in virtual-8086 mode.
-----	--

## Compatibility Mode Exceptions

#UD	The INVEPT instruction is not recognized in compatibility mode.
-----	---

## 64-Bit Mode Exceptions

#GP(0)	<p>If the current privilege level is not 0.</p> <p>If the memory operand is in the CS, DS, ES, FS, or GS segments and the memory address is in a non-canonical form.</p>
#PF(fault-code)	If a page fault occurs in accessing the memory operand.
#SS(0)	If the memory operand is in the SS segment and the memory address is in a non-canonical form.
#UD	If not in VMX operation.



If the logical processor does not support EPT  
(IA32\_VMX\_PROCBASED\_CTL2[33]=0).

If the logical processor supports EPT  
(IA32\_VMX\_PROCBASED\_CTL2[33]=1) but does not support  
the INVEPT instruction (IA32\_VMX\_EPT\_VPID\_CAP[20]=0).

## INVVPID— Invalidate Translations Based on VPID

Opcode	Instruction	Description
66 0F 38 81	INVVPID r64, m128	Invalidates entries in the TLBs and paging-structure caches based on VPID (in 64-bit mode)
66 0F 38 81	INVVPID r32, m128	Invalidates entries in the TLBs and paging-structure caches based on VPID (outside 64-bit mode)

### Description

Invalidates mappings in the translation lookaside buffers (TLBs) and paging-structure caches based on **virtual-processor identifier** (VPID). (See Chapter 28, “VMX Support for Address Translation” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3C*.) Invalidation is based on the **INVVPID type** specified in the register operand and the **INVVPID descriptor** specified in the memory operand.

Outside IA-32e mode, the register operand is always 32 bits, regardless of the value of CS.D; in 64-bit mode, the register operand has 64 bits (the instruction cannot be executed in compatibility mode).

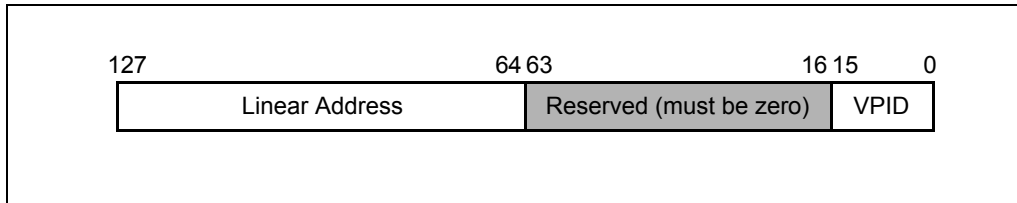
The INVVPID types supported by a logical processors are reported in the IA32\_VMX\_EPT\_VPID\_CAP MSR (see Appendix A, “VMX Capability Reporting Facility” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3C*). There are four INVVPID types currently defined:

- Individual-address invalidation: If the INVVPID type is 0, the logical processor invalidates mappings for the linear address and VPID specified in the INVVPID descriptor. In some cases, it may invalidate mappings for other linear addresses (or other VPIDs) as well.
- Single-context invalidation: If the INVVPID type is 1, the logical processor invalidates all mappings tagged with the VPID specified in the INVVPID descriptor. In some cases, it may invalidate mappings for other VPIDs as well.
- All-contexts invalidation: If the INVVPID type is 2, the logical processor invalidates all mappings tagged with all VPIDs except VPID 0000H. In some cases, it may invalidate translations with VPID 0000H as well.
- Single-context invalidation, retaining global translations: If the INVVPID type is 3, the logical processor invalidates all mappings tagged with the VPID specified in the INVVPID descriptor except global translations. In some cases, it may invalidate global translations (and mappings with other VPIDs) as well. See the “Caching Translation Information” section in Chapter 4 of the *IA-32 Intel Architecture Software Developer’s Manual, Volumes 3A* for information about global translations.

If an unsupported INVVPID type is specified, the instruction fails.

INVVPID invalidates all the specified mappings for the indicated VPID(s) regardless of the EPTP and PCID values with which those mappings may be associated.

The INVVPID descriptor comprises 128 bits and consists of a VPID and a linear address as shown in Figure 5-2.



**Figure 5-2. INVVPID Descriptor**

### Operation

```

IF (not in VMX operation) or (CRO.PE = 0) or (RFLAGS.VM = 1) or (IA32_EFER.LMA = 1 and CS.L = 0)
    THEN #UD;
ELSIF in VMX non-root operation
    THEN VM exit;
ELSIF CPL > 0
    THEN #GP(0);
ELSE
    INVVPID_TYPE ← value of register operand;
    IF IA32_VMX_EPT_VPID_CAP MSR indicates that processor does not support
    INVVPID_TYPE
        THEN VMfail(Invalid operand to INVEPT/INVVPID);
    ELSE // INVVPID_TYPE must be in the range 0-3
        INVVPID_DESC ← value of memory operand;
        IF INVVPID_DESC[63:16] ≠ 0
            THEN VMfail(Invalid operand to INVEPT/INVVPID);
        ELSE
            CASE INVVPID_TYPE OF
                0: // individual-address invalidation
                    VPID ← INVVPID_DESC[15:0];
                    IF VPID = 0
                        THEN VMfail(Invalid operand to INVEPT/INVVPID);
                    ELSE
                        GL_ADDR ← INVVPID_DESC[127:64];
                        IF (GL_ADDR is not in a canonical form)
                            THEN
                                VMfail(Invalid operand to INVEPT/INVVPID);
                        ELSE

```

```

with VPID;
                                Invalidate mappings for GL_ADDR tagged
                                VMsucceed;
                                FI;
                                FI;
                                BREAK;
1:                                // single-context invalidation
    VPID ← INVVPID_DESC[15:0];
    IF VPID = 0
        THEN VMfail(Invalid operand to INVEPT/INVVPID);
        ELSE
            Invalidate all mappings tagged with VPID;
            VMsucceed;
            FI;
            BREAK;
2:                                // all-context invalidation
    Invalidate all mappings tagged with all non-zero VPIDs;
    VMsucceed;
    BREAK;
3:                                // single-context invalidation retaining globals
    VPID ← INVVPID_DESC[15:0];
    IF VPID = 0
        THEN VMfail(Invalid operand to INVEPT/INVVPID);
        ELSE
            Invalidate all mappings tagged with VPID except
global translations;
                                VMsucceed;
                                FI;
                                BREAK;
                                ESAC;
                                FI;
                                FI;
FI;

```

### Flags Affected

See the operation section and Section 5.2.

### Protected Mode Exceptions

- #GP(0) If the current privilege level is not 0.
- If the memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
- If the DS, ES, FS, or GS register contains an unusable segment.

	If the source operand is located in an execute-only code segment.
#PF(fault-code)	If a page fault occurs in accessing the memory operand.
#SS(0)	If the memory operand effective address is outside the SS segment limit.
	If the SS register contains an unusable segment.
#UD	If not in VMX operation.
	If the logical processor does not support VPIDs (IA32_VMX_PROCBASED_CTLSS2[37]=0).
	If the logical processor supports VPIDs (IA32_VMX_PROCBASED_CTLSS2[37]=1) but does not support the INVVPID instruction (IA32_VMX_EPT_VPID_CAP[32]=0).

### Real-Address Mode Exceptions

#UD	A logical processor cannot be in real-address mode while in VMX operation and the INVVPID instruction is not recognized outside VMX operation.
-----	--

### Virtual-8086 Mode Exceptions

#UD	The INVVPID instruction is not recognized in virtual-8086 mode.
-----	---

### Compatibility Mode Exceptions

#UD	The INVVPID instruction is not recognized in compatibility mode.
-----	--

### 64-Bit Mode Exceptions

#GP(0)	If the current privilege level is not 0.
	If the memory operand is in the CS, DS, ES, FS, or GS segments and the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs in accessing the memory operand.
#SS(0)	If the memory destination operand is in the SS segment and the memory address is in a non-canonical form.
#UD	If not in VMX operation.
	If the logical processor does not support VPIDs (IA32_VMX_PROCBASED_CTLSS2[37]=0).
	If the logical processor supports VPIDs (IA32_VMX_PROCBASED_CTLSS2[37]=1) but does not support the INVVPID instruction (IA32_VMX_EPT_VPID_CAP[32]=0).

## VMCALL—Call to VM Monitor

Opcode	Instruction	Description
OF 01 C1	VMCALL	Call to VM monitor by causing VM exit.

### Description

This instruction allows guest software can make a call for service into an underlying VM monitor. The details of the programming interface for such calls are VMM-specific; this instruction does nothing more than cause a VM exit, registering the appropriate exit reason.

Use of this instruction in VMX root operation invokes an SMM monitor (see Section 29.15.2 in *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C*). This invocation will activate the dual-monitor treatment of system-management interrupts (SMIs) and system-management mode (SMM) if it is not already active (see Section 29.15.6 in *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C*).

### Operation

```

IF not in VMX operation
    THEN #UD;
ELSIF in VMX non-root operation
    THEN VM exit;
ELSIF (RFLAGS.VM = 1) or (IA32_EFER.LMA = 1 and CS.L = 0)
    THEN #UD;
ELSIF CPL > 0
    THEN #GP(0);
ELSIF in SMM or the logical processor does not support the dual-monitor treatment of SMIs and SMM or the valid bit in the IA32_SMM_MONITOR_CTL MSR is clear
    THEN VMfail (VMCALL executed in VMX root operation);
ELSIF dual-monitor treatment of SMIs and SMM is active
    THEN perform an SMM VM exit (see Section 32.15.2
        of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C);
ELSIF current-VMCS pointer is not valid
    THEN VMfailInvalid;
ELSIF launch state of current VMCS is not clear
    THEN VMfailValid (VMCALL with non-clear VMCS);
ELSIF VM-exit control fields are not valid (see Section 29.15.6.1 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C)
    THEN VMfailValid (VMCALL with invalid VM-exit control fields);
ELSE
    enter SMM;
  
```

```

read revision identifier in MSEG;
IF revision identifier does not match that supported by processor
    THEN
        leave SMM;
        VMfailValid(VMCALL with incorrect MSEG revision identifier);
    ELSE
        read SMM-monitor features field in MSEG (see Section 29.15.6.2,
        in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C;
        IF features field is invalid
            THEN
                leave SMM;
                VMfailValid(VMCALL with invalid SMM-monitor features);
            ELSE activate dual-monitor treatment of SMIs and SMM (see Section 29.15.6
            in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume
            3C);
        FI;
    FI;
FI;

```

### Flags Affected

See the operation section and Section 5.2.

### Protected Mode Exceptions

#GP(0)            If the current privilege level is not 0 and the logical processor is in VMX root operation.

#UD                If executed outside VMX operation.

### Real-Address Mode Exceptions

#UD                If executed outside VMX operation.

### Virtual-8086 Mode Exceptions

#UD                If executed outside VMX non-root operation.

### Compatibility Mode Exceptions

#UD                If executed outside VMX non-root operation.

### 64-Bit Mode Exceptions

#UD                If executed outside VMX non-root operation.

## VMCLEAR—Clear Virtual-Machine Control Structure

Opcode	Instruction	Description
66 0F C7 /6	VMCLEAR m64	Copy VMCS data to VMCS region in memory.

### Description

This instruction applies to the VMCS whose VMCS region resides at the physical address contained in the instruction operand. The instruction ensures that VMCS data for that VMCS (some of these data may be currently maintained on the processor) are copied to the VMCS region in memory. It also initializes parts of the VMCS region (for example, it sets the launch state of that VMCS to clear). See Chapter 30, “Virtual-Machine Monitor Programming Considerations,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3C*.

The operand of this instruction is always 64 bits and is always in memory. If the operand is the current-VMCS pointer, then that pointer is made invalid (set to FFFFFFFF\_FFFFFFFFH).

Note that the VMCLEAR instruction might not explicitly write any VMCS data to memory; the data may be already resident in memory before the VMCLEAR is executed.

### Operation

IF (register operand) or (not in VMX operation) or (CR0.PE = 0) or (RFLAGS.VM = 1) or (IA32\_EFER.LMA = 1 and CS.L = 0)

THEN #UD;

ELSIF in VMX non-root operation

THEN VM exit;

ELSIF CPL > 0

THEN #GP(0);

ELSE

addr ← contents of 64-bit in-memory operand;

IF addr is not 4KB-aligned OR

addr sets any bits beyond the physical-address width<sup>1</sup>

THEN VMfail(VMCLEAR with invalid physical address);

ELSIF addr = VMXON pointer

THEN VMfail(VMCLEAR with VMXON pointer);

ELSE

ensure that data for VMCS referenced by the operand is in memory;

initialize implementation-specific data in VMCS region;

1. If IA32\_VMX\_BASIC[48] is read as 1, VMfail occurs if addr sets any bits in the range 63:32; see Appendix A.1.



```

launch state of VMCS referenced by the operand ← “clear”
IF operand addr = current-VMCS pointer
    THEN current-VMCS pointer ← FFFFFFFF_FFFFFFFFH;
FI;
VMsucceed;

```

```

FI;

```

```

FI;

```

## Flags Affected

See the operation section and Section 5.2.

## Protected Mode Exceptions

#GP(0)	<p>If the current privilege level is not 0.</p> <p>If the memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.</p> <p>If the DS, ES, FS, or GS register contains an unusable segment.</p> <p>If the operand is located in an execute-only code segment.</p>
#PF(fault-code)	If a page fault occurs in accessing the memory operand.
#SS(0)	<p>If the memory operand effective address is outside the SS segment limit.</p> <p>If the SS register contains an unusable segment.</p>
#UD	<p>If operand is a register.</p> <p>If not in VMX operation.</p>

## Real-Address Mode Exceptions

#UD	A logical processor cannot be in real-address mode while in VMX operation and the VMCLEAR instruction is not recognized outside VMX operation.
-----	--

## Virtual-8086 Mode Exceptions

#UD	The VMCLEAR instruction is not recognized in virtual-8086 mode.
-----	---

## Compatibility Mode Exceptions

#UD	The VMCLEAR instruction is not recognized in compatibility mode.
-----	--

## 64-Bit Mode Exceptions

#GP(0)	<p>If the current privilege level is not 0.</p> <p>If the source operand is in the CS, DS, ES, FS, or GS segments and the memory address is in a non-canonical form.</p>
--------	--

## VMX INSTRUCTION REFERENCE

#PF(fault-code)	If a page fault occurs in accessing the memory operand.
#SS(0)	If the source operand is in the SS segment and the memory address is in a non-canonical form.
#UD	If operand is a register. If not in VMX operation.

## VMFUNC—Invoke VM Function

Opcode	Instruction	Description
0F 01 D4	VMFUNC	Invoke VM function specified in EAX.

### Description

This instruction allows software in VMX non-root operation to invoke a VM function, which is processor functionality enabled and configured by software in VMX root operation. The value of EAX selects the specific VM function being invoked.

The behavior of each VM function (including any additional fault checking) is specified in Section 25.7.4, “VM Functions,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3C*.

### Operation

Perform functionality of the VM function specified in EAX;

### Flags Affected

Depends on the VM function specified in EAX. See Section 25.7.4, “VM Functions,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3C*.

### Protected Mode Exceptions (not including those defined by specific VM functions)

#UD	If executed outside VMX non-root operation. If “enable VM functions” VM-execution control is 0. If $EAX \geq 64$ . If the bit at position EAX is 0 in the VM-function controls.
-----	--

### Real-Address Mode Exceptions

Same exceptions as in protected mode.

### Virtual-8086 Exceptions

Same exceptions as in protected mode.

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

Same exceptions as in protected mode.

## VMLAUNCH/VMRESUME—Launch/Resume Virtual Machine

Opcode	Instruction	Description
0F 01 C2	VMLAUNCH	Launch virtual machine managed by current VMCS.
0F 01 C3	VMRESUME	Resume virtual machine managed by current VMCS.

### Description

Effects a VM entry managed by the current VMCS.

- VMLAUNCH fails if the launch state of current VMCS is not “clear”. If the instruction is successful, it sets the launch state to “launched.”
- VMRESUME fails if the launch state of the current VMCS is not “launched.”

If VM entry is attempted, the logical processor performs a series of consistency checks as detailed in Chapter 32, “Handling Boundary Conditions in a Virtual Machine Monitor,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3C*. Failure to pass checks on the VMX controls or on the host-state area passes control to the instruction following the VMLAUNCH or VMRESUME instruction. If these pass but checks on the guest-state area fail, the logical processor loads state from the host-state area of the VMCS, passing control to the instruction referenced by the RIP field in the host-state area.

VM entry is not allowed when events are blocked by MOV SS or POP SS. Neither VMLAUNCH nor VMRESUME should be used immediately after either MOV to SS or POP to SS.

### Operation

IF (not in VMX operation) or (CR0.PE = 0) or (RFLAGS.VM = 1) or (IA32\_EFER.LMA = 1 and CS.L = 0)  
THEN #UD;

ELSIF in VMX non-root operation

THEN VMexit;

ELSIF CPL > 0

THEN #GP(0);

ELSIF current-VMCS pointer is not valid

THEN VMfailInvalid;

ELSIF events are being blocked by MOV SS

THEN VMfailValid(VM entry with events blocked by MOV SS);

ELSIF (VMLAUNCH and launch state of current VMCS is not “clear”)

THEN VMfailValid(VMLAUNCH with non-clear VMCS);

ELSIF (VMRESUME and launch state of current VMCS is not “launched”)

THEN VMfailValid(VMRESUME with non-launched VMCS);

ELSE

Check settings of VMX controls and host-state area;

```

IF invalid settings
  THEN VMfailValid(VM entry with invalid VMX-control field(s)) or
    VMfailValid(VM entry with invalid host-state field(s)) or
    VMfailValid(VM entry with invalid executive-VMCS pointer) or
    VMfailValid(VM entry with non-launched executive VMCS) or
    VMfailValid(VM entry with executive-VMCS pointer not VMXON pointer) or
    VMfailValid(VM entry with invalid VM-execution control fields in executive
      VMCS)
    as appropriate;
ELSE
  Attempt to load guest state and PDPTRs as appropriate;
  clear address-range monitoring;
  IF failure in checking guest state or PDPTRs
    THEN VM entry fails (see Section 26.7, in the
      Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C);
    ELSE
      Attempt to load MSR from VM-entry MSR-load area;
      IF failure
        THEN VM entry fails (see Section 26.7, in the Intel® 64 and IA-32
          Architectures Software Developer's Manual, Volume 3C);
        ELSE
          IF VMLAUNCH
            THEN launch state of VMCS ← "launched";
          FI;
          IF in SMM and "entry to SMM" VM-entry control is 0
            THEN
              IF "deactivate dual-monitor treatment" VM-entry
                control is 0
                THEN SMM-transfer VMCS pointer ←
                  current-VMCS pointer;
              FI;
              IF executive-VMCS pointer is VMX pointer
                THEN current-VMCS pointer ←
                  VMCS-link pointer;
                ELSE current-VMCS pointer ←
                  executive-VMCS pointer;
              FI;
              leave SMM;
            FI;
          VM entry succeeds;
        FI;
      FI;
    FI;
  FI;
FI;

```

Further details of the operation of the VM-entry appear in Chapter 26 of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3C*.

### Flags Affected

See the operation section and Section 5.2.

### Protected Mode Exceptions

- #GP(0) If the current privilege level is not 0.
- #UD If executed outside VMX operation.

### Real-Address Mode Exceptions

- #UD A logical processor cannot be in real-address mode while in VMX operation and the VMLAUNCH and VMRESUME instructions are not recognized outside VMX operation.

### Virtual-8086 Mode Exceptions

- #UD The VMLAUNCH and VMRESUME instructions are not recognized in virtual-8086 mode.

### Compatibility Mode Exceptions

- #UD The VMLAUNCH and VMRESUME instructions are not recognized in compatibility mode.

### 64-Bit Mode Exceptions

- #GP(0) If the current privilege level is not 0.
- #UD If executed outside VMX operation.

## VMPTRLD—Load Pointer to Virtual-Machine Control Structure

Opcode	Instruction	Description
0F C7 /6	VMPTRLD m64	Loads the current VMCS pointer from memory.

### Description

Marks the current-VMCS pointer valid and loads it with the physical address in the instruction operand. The instruction fails if its operand is not properly aligned, sets unsupported physical-address bits, or is equal to the VMXON pointer. In addition, the instruction fails if the 32 bits in memory referenced by the operand do not match the VMCS revision identifier supported by this processor.<sup>2</sup>

The operand of this instruction is always 64 bits and is always in memory.

### Operation

IF (register operand) or (not in VMX operation) or (CR0.PE = 0) or (RFLAGS.VM = 1) or (IA32\_EFER.LMA = 1 and CS.L = 0)

THEN #UD;

ELSIF in VMX non-root operation

THEN VMexit;

ELSIF CPL > 0

THEN #GP(0);

ELSE

addr ← contents of 64-bit in-memory source operand;

IF addr is not 4KB-aligned OR

addr sets any bits beyond the physical-address width<sup>3</sup>

THEN VMfail(VMPTRLD with invalid physical address);

ELSIF addr = VMXON pointer

THEN VMfail(VMPTRLD with VMXON pointer);

ELSE

rev ← 32 bits located at physical address addr;

IF rev ≠ VMCS revision identifier supported by processor

THEN VMfail(VMPTRLD with incorrect VMCS revision identifier);

ELSE

current-VMCS pointer ← addr;

VMsucceed;

- 
2. Software should consult the VMX capability MSR VMX\_BASIC to discover the VMCS revision identifier supported by this processor (see Appendix A, “VMX Capability Reporting Facility,” in the *Intel 64 and IA-32 Architecture Software Developer’s Manual, Volume 3C*).
  3. If IA32\_VMX\_BASIC[48] is read as 1, VMfail occurs if addr sets any bits in the range 63:32; see Appendix A.1.

FI;  
FI;  
FI;

### Flags Affected

See the operation section and Section 5.2.

### Protected Mode Exceptions

#GP(0)	If the current privilege level is not 0. If the memory source operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register contains an unusable segment. If the source operand is located in an execute-only code segment.
#PF(fault-code)	If a page fault occurs in accessing the memory source operand.
#SS(0)	If the memory source operand effective address is outside the SS segment limit. If the SS register contains an unusable segment.
#UD	If operand is a register. If not in VMX operation.

### Real-Address Mode Exceptions

#UD	A logical processor cannot be in real-address mode while in VMX operation and the VMPTRLD instruction is not recognized outside VMX operation.
-----	--

### Virtual-8086 Mode Exceptions

#UD	The VMPTRLD instruction is not recognized in virtual-8086 mode.
-----	---

### Compatibility Mode Exceptions

#UD	The VMPTRLD instruction is not recognized in compatibility mode.
-----	--

### 64-Bit Mode Exceptions

#GP(0)	If the current privilege level is not 0. If the source operand is in the CS, DS, ES, FS, or GS segments and the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs in accessing the memory source operand.



#SS(0)	If the source operand is in the SS segment and the memory address is in a non-canonical form.
#UD	If operand is a register. If not in VMX operation.

## VMPTRST—Store Pointer to Virtual-Machine Control Structure

Opcode	Instruction	Description
0F C7 77	VMPTRST m64	Stores the current VMCS pointer into memory.

### Description

Stores the current-VMCS pointer into a specified memory address. The operand of this instruction is always 64 bits and is always in memory.

### Operation

IF (register operand) or (not in VMX operation) or (CR0.PE = 0) or (RFLAGS.VM = 1) or (IA32\_EFER.LMA = 1 and CS.L = 0)

THEN #UD;

ELSIF in VMX non-root operation

THEN VMexit;

ELSIF CPL > 0

THEN #GP(0);

ELSE

64-bit in-memory destination operand ← current-VMCS pointer;

VMsucceed;

FI;

### Flags Affected

See the operation section and Section 5.2.

### Protected Mode Exceptions

#GP(0)	<p>If the current privilege level is not 0.</p> <p>If the memory destination operand effective address is outside the CS, DS, ES, FS, or GS segment limit.</p> <p>If the DS, ES, FS, or GS register contains an unusable segment.</p> <p>If the destination operand is located in a read-only data segment or any code segment.</p>
#PF(fault-code)	<p>If a page fault occurs in accessing the memory destination operand.</p>
#SS(0)	<p>If the memory destination operand effective address is outside the SS segment limit.</p> <p>If the SS register contains an unusable segment.</p>
#UD	<p>If operand is a register.</p> <p>If not in VMX operation.</p>

### Real-Address Mode Exceptions

#UD                    A logical processor cannot be in real-address mode while in VMX operation and the VMPTRST instruction is not recognized outside VMX operation.

### Virtual-8086 Mode Exceptions

#UD                    The VMPTRST instruction is not recognized in virtual-8086 mode.

### Compatibility Mode Exceptions

#UD                    The VMPTRST instruction is not recognized in compatibility mode.

### 64-Bit Mode Exceptions

#GP(0)                If the current privilege level is not 0.  
                          If the destination operand is in the CS, DS, ES, FS, or GS segments and the memory address is in a non-canonical form.

#PF(fault-code)     If a page fault occurs in accessing the memory destination operand.

#SS(0)                If the destination operand is in the SS segment and the memory address is in a non-canonical form.

#UD                    If operand is a register.  
                          If not in VMX operation.

## VMREAD—Read Field from Virtual-Machine Control Structure

Opcode	Instruction	Description
0F 78	VMREAD r/m64, r64	Reads a specified VMCS field (in 64-bit mode).
0F 78	VMREAD r/m32, r32	Reads a specified VMCS field (outside 64-bit mode).

### Description

Reads a specified field from the VMCS and stores it into a specified destination operand (register or memory).

The specific VMCS field is identified by the VMCS-field encoding contained in the register source operand. Outside IA-32e mode, the source operand has 32 bits, regardless of the value of CS.D. In 64-bit mode, the source operand has 64 bits; however, if bits 63:32 of the source operand are not zero, VMREAD will fail due to an attempt to access an unsupported VMCS component (see operation section).

The effective size of the destination operand, which may be a register or in memory, is always 32 bits outside IA-32e mode (the setting of CS.D is ignored with respect to operand size) and 64 bits in 64-bit mode. If the VMCS field specified by the source operand is shorter than this effective operand size, the high bits of the destination operand are cleared to 0. If the VMCS field is longer, then the high bits of the field are not read.

Note that any faults resulting from accessing a memory destination operand can occur only after determining, in the operation section below, that the VMCS pointer is valid and that the specified VMCS field is supported.

### Operation

```

IF (not in VMX operation) or (RFLAGS.VM = 1) or (IA32_EFER.LMA = 1 and CS.L = 0)
    THEN #UD;
ELSIF in VMX non-root operation
    THEN VMexit;
ELSIF CPL > 0
    THEN #GP(0);
ELSIF current-VMCS pointer is not valid
    THEN VMfailInvalid;
ELSIF register source operand does not correspond to any VMCS field
    THEN VMfailValid(VMREAD/VMWRITE from/to unsupported VMCS component);
ELSE
    DEST ← contents of VMCS field indexed by register source operand;
    VMSucceed;
FI;

```

## Flags Affected

See the operation section and Section 5.2.

## Protected Mode Exceptions

#GP(0)	<p>If the current privilege level is not 0.</p> <p>If a memory destination operand effective address is outside the CS, DS, ES, FS, or GS segment limit.</p> <p>If the DS, ES, FS, or GS register contains an unusable segment.</p> <p>If the destination operand is located in a read-only data segment or any code segment.</p>
#PF(fault-code)	If a page fault occurs in accessing a memory destination operand.
#SS(0)	<p>If a memory destination operand effective address is outside the SS segment limit.</p> <p>If the SS register contains an unusable segment.</p>
#UD	If not in VMX operation.

## Real-Address Mode Exceptions

#UD	A logical processor cannot be in real-address mode while in VMX operation and the VMREAD instruction is not recognized outside VMX operation.
-----	---

## Virtual-8086 Mode Exceptions

#UD	The VMREAD instruction is not recognized in virtual-8086 mode.
-----	--

## Compatibility Mode Exceptions

#UD	The VMREAD instruction is not recognized in compatibility mode.
-----	---

## 64-Bit Mode Exceptions

#GP(0)	<p>If the current privilege level is not 0.</p> <p>If the memory destination operand is in the CS, DS, ES, FS, or GS segments and the memory address is in a non-canonical form.</p>
#PF(fault-code)	If a page fault occurs in accessing a memory destination operand.
#SS(0)	If the memory destination operand is in the SS segment and the memory address is in a non-canonical form.
#UD	If not in VMX operation.

## **VMRESUME—Resume Virtual Machine**

See VMLAUNCH/VMRESUME—Launch/Resume Virtual Machine.

## VMWRITE—Write Field to Virtual-Machine Control Structure

Opcode	Instruction	Description
0F 79	VMWRITE r64, r/m64	Writes a specified VMCS field (in 64-bit mode)
0F 79	VMWRITE r32, r/m32	Writes a specified VMCS field (outside 64-bit mode)

### Description

Writes to a specified field in the VMCS specified by a secondary source operand (register only) using the contents of a primary source operand (register or memory).

The VMCS field is identified by the VMCS-field encoding contained in the register secondary source operand. Outside IA-32e mode, the secondary source operand is always 32 bits, regardless of the value of CS.D. In 64-bit mode, the secondary source operand has 64 bits; however, if bits 63:32 of the secondary source operand are not zero, VMWRITE will fail due to an attempt to access an unsupported VMCS component (see operation section).

The effective size of the primary source operand, which may be a register or in memory, is always 32 bits outside IA-32e mode (the setting of CS.D is ignored with respect to operand size) and 64 bits in 64-bit mode. If the VMCS field specified by the secondary source operand is shorter than this effective operand size, the high bits of the primary source operand are ignored. If the VMCS field is longer, then the high bits of the field are cleared to 0.

Note that any faults resulting from accessing a memory source operand occur after determining, in the operation section below, that the VMCS pointer is valid but before determining if the destination VMCS field is supported.

### Operation

```

IF (not in VMX operation) or (CR0.PE = 0) or (RFLAGS.VM = 1) or (IA32_EFER.LMA = 1 and CS.L = 0)
    THEN #UD;
ELSIF in VMX non-root operation
    THEN VMExit;
ELSIF CPL > 0
    THEN #GP(0);
ELSIF current-VMCS pointer is not valid
    THEN VMfailInvalid;
ELSIF register destination operand does not correspond to any VMCS field
    THEN VMfailValid(VMREAD/VMWRITE from/to unsupported VMCS component);
ELSIF VMCS field indexed by register destination operand is read-only
    THEN VMfailValid(VMWRITE to read-only VMCS component);
ELSE
    VMCS field indexed by register destination operand ← SRC;
    VMsucceed;

```

FI;

### Flags Affected

See the operation section and Section 5.2.

### Protected Mode Exceptions

#GP(0)	If the current privilege level is not 0. If a memory source operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register contains an unusable segment. If the source operand is located in an execute-only code segment.
#PF(fault-code)	If a page fault occurs in accessing a memory source operand.
#SS(0)	If a memory source operand effective address is outside the SS segment limit. If the SS register contains an unusable segment.
#UD	If not in VMX operation.

### Real-Address Mode Exceptions

#UD	A logical processor cannot be in real-address mode while in VMX operation and the VMWRITE instruction is not recognized outside VMX operation.
-----	--

### Virtual-8086 Mode Exceptions

#UD	The VMWRITE instruction is not recognized in virtual-8086 mode.
-----	---

### Compatibility Mode Exceptions

#UD	The VMWRITE instruction is not recognized in compatibility mode.
-----	--

### 64-Bit Mode Exceptions

#GP(0)	If the current privilege level is not 0. If the memory source operand is in the CS, DS, ES, FS, or GS segments and the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs in accessing a memory source operand.
#SS(0)	If the memory source operand is in the SS segment and the memory address is in a non-canonical form.
#UD	If not in VMX operation.



## VMXOFF—Leave VMX Operation

Opcode	Instruction	Description
OF 01 C4	VMXOFF	Leaves VMX operation.

### Description

Takes the logical processor out of VMX operation, unblocks INIT signals, conditionally re-enables A20M, and clears any address-range monitoring.<sup>4</sup>

### Operation

```

IF (not in VMX operation) or (CR0.PE = 0) or (RFLAGS.VM = 1) or (IA32_EFER.LMA = 1 and CS.L = 0)
    THEN #UD;
ELSIF in VMX non-root operation
    THEN VMexit;
ELSIF CPL > 0
    THEN #GP(0);
ELSIF dual-monitor treatment of SMIs and SMM is active
    THEN VMfail(VMXOFF under dual-monitor treatment of SMIs and SMM);
ELSE
    leave VMX operation;
    unblock INIT;
    IF IA32_SMM_MONITOR_CTL[2] = 05
        THEN unblock SMIs;
    IF outside SMX operation6
        THEN unblock and enable A20M;
    FI;
    clear address-range monitoring;
    VMSucceed;
FI;

```

- 
- See the information on MONITOR/MWAIT in Chapter 8, “Multiple-Processor Management,” of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.
  - Setting IA32\_SMM\_MONITOR\_CTL[bit 2] to 1 prevents VMXOFF from unblocking SMIs regardless of the value of the register’s value bit (bit 0). Not all processors allow this bit to be set to 1. Software should consult the VMX capability MSR IA32\_VMX\_MISC (see Appendix A.6) to determine whether this is allowed.
  - A logical processor is outside SMX operation if GETSEC[SENDER] has not been executed or if GETSEC[SEXIT] was executed after the last execution of GETSEC[SENDER]. See Chapter 6, “Safer Mode Extensions Reference.”

## Flags Affected

See the operation section and Section 5.2.

## Protected Mode Exceptions

- #GP(0) If executed in VMX root operation with CPL > 0.
- #UD If executed outside VMX operation.

## Real-Address Mode Exceptions

- #UD A logical processor cannot be in real-address mode while in VMX operation and the VMXOFF instruction is not recognized outside VMX operation.

## Virtual-8086 Mode Exceptions

- #UD The VMXOFF instruction is not recognized in virtual-8086 mode.

## Compatibility Mode Exceptions

- #UD The VMXOFF instruction is not recognized in compatibility mode.

## 64-Bit Mode Exceptions

- #GP(0) If executed in VMX root operation with CPL > 0.
- #UD If executed outside VMX operation.

## VMXON—Enter VMX Operation

Opcode	Instruction	Description
F3 0F C7 /6	VMXON m64	Enter VMX root operation.

### Description

Puts the logical processor in VMX operation with no current VMCS, blocks INIT signals, disables A20M, and clears any address-range monitoring established by the MONITOR instruction.<sup>7</sup>

The operand of this instruction is a 4KB-aligned physical address (the VMXON pointer) that references the VMXON region, which the logical processor may use to support VMX operation. This operand is always 64 bits and is always in memory.

### Operation

IF (register operand) or (CR0.PE = 0) or (CR4.VMXE = 0) or (RFLAGS.VM = 1) or (IA32\_EFER.LMA = 1 and CS.L = 0)

THEN #UD;

ELSIF not in VMX operation

THEN

IF (CPL > 0) or (in A20M mode) or

(the values of CR0 and CR4 are not supported in VMX operation<sup>8</sup>) or

(bit 0 (lock bit) of IA32\_FEATURE\_CONTROL MSR is clear) or

(in SMX operation<sup>9</sup> and bit 1 of IA32\_FEATURE\_CONTROL MSR is clear) or

(outside SMX operation and bit 2 of IA32\_FEATURE\_CONTROL MSR is clear)

THEN #GP(0);

ELSE

addr ← contents of 64-bit in-memory source operand;

IF addr is not 4KB-aligned or

addr sets any bits beyond the physical-address width<sup>10</sup>

THEN VMfailInvalid;

- 
7. See the information on MONITOR/MWAIT in Chapter 8, “Multiple-Processor Management,” of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.
  8. See Section 19.8 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B*.
  9. A logical processor is in SMX operation if GETSEC[SEXIT] has not been executed since the last execution of GETSEC[SENTER]. A logical processor is outside SMX operation if GETSEC[SENTER] has not been executed or if GETSEC[SEXIT] was executed after the last execution of GETSEC[SENTER]. See Chapter 6, “Safer Mode Extensions Reference.”
  10. If IA32\_VMX\_BASIC[48] is read as 1, VMfailInvalid occurs if addr sets any bits in the range 63:32; see Appendix A.1.

```

ELSE
    rev ← 32 bits located at physical address addr;
    IF rev ≠ VMCS revision identifier supported by processor
        THEN VMfailInvalid;
        ELSE
            current-VMCS pointer ← FFFFFFFF_FFFFFFFFH;
            enter VMX operation;
            block INIT signals;
            block and disable A20M;
            clear address-range monitoring;
            VMSucceed;
    FI;
FI;
FI;
ELSIF in VMX non-root operation
    THEN VMExit;
ELSIF CPL > 0
    THEN #GP(0);
    ELSE VMfail("VMXON executed in VMX root operation");
FI;

```

### Flags Affected

See the operation section and Section 5.2.

### Protected Mode Exceptions

#GP(0)	<p>If executed outside VMX operation with CPL&gt;0 or with invalid CR0 or CR4 fixed bits.</p> <p>If executed in A20M mode.</p> <p>If the memory source operand effective address is outside the CS, DS, ES, FS, or GS segment limit.</p> <p>If the DS, ES, FS, or GS register contains an unusable segment.</p> <p>If the source operand is located in an execute-only code segment.</p>
#PF(fault-code)	If a page fault occurs in accessing the memory source operand.
#SS(0)	<p>If the memory source operand effective address is outside the SS segment limit.</p> <p>If the SS register contains an unusable segment.</p>
#UD	<p>If operand is a register.</p> <p>If executed with CR4.VMXE = 0.</p>

**Real-Address Mode Exceptions**

#UD The VMXON instruction is not recognized in real-address mode.

**Virtual-8086 Mode Exceptions**

#UD The VMXON instruction is not recognized in virtual-8086 mode.

**Compatibility Mode Exceptions**

#UD The VMXON instruction is not recognized in compatibility mode.

**64-Bit Mode Exceptions**

#GP(0) If executed outside VMX operation with CPL > 0 or with invalid CR0 or CR4 fixed bits.  
If executed in A20M mode.  
If the source operand is in the CS, DS, ES, FS, or GS segments and the memory address is in a non-canonical form.

#PF(fault-code) If a page fault occurs in accessing the memory source operand.

#SS(0) If the source operand is in the SS segment and the memory address is in a non-canonical form.

#UD If operand is a register.  
If executed with CR4.VMXE = 0.

## 5.4 VM INSTRUCTION ERROR NUMBERS

For certain error conditions, the VM-instruction error field is loaded with an error number to indicate the source of the error. Table 5-1 lists VM-instruction error numbers.

**Table 5-1. VM-Instruction Error Numbers**

Error Number	Description
1	VMCALL executed in VMX root operation
2	VMCLEAR with invalid physical address
3	VMCLEAR with VMXON pointer
4	VMLAUNCH with non-clear VMCS
5	VMRESUME with non-launched VMCS
6	VMRESUME after VMXOFF (VMXOFF and VMXON between VMLAUNCH and VMRESUME) <sup>a</sup>
7	VM entry with invalid control field(s) <sup>b,c</sup>
8	VM entry with invalid host-state field(s) <sup>b</sup>
9	VMPTRLD with invalid physical address
10	VMPTRLD with VMXON pointer
11	VMPTRLD with incorrect VMCS revision identifier
12	VMREAD/VMWRITE from/to unsupported VMCS component
13	VMWRITE to read-only VMCS component
15	VMXON executed in VMX root operation
16	VM entry with invalid executive-VMCS pointer <sup>b</sup>
17	VM entry with non-launched executive VMCS <sup>b</sup>
18	VM entry with executive-VMCS pointer not VMXON pointer (when attempting to deactivate the dual-monitor treatment of SMIs and SMM) <sup>b</sup>
19	VMCALL with non-clear VMCS (when attempting to activate the dual-monitor treatment of SMIs and SMM)
20	VMCALL with invalid VM-exit control fields
22	VMCALL with incorrect MSEG revision identifier (when attempting to activate the dual-monitor treatment of SMIs and SMM)
23	VMXOFF under dual-monitor treatment of SMIs and SMM
24	VMCALL with invalid SMM-monitor features (when attempting to activate the dual-monitor treatment of SMIs and SMM)

**Table 5-1. VM-Instruction Error Numbers (Contd.)**

Error Number	Description
25	VM entry with invalid VM-execution control fields in executive VMCS (when attempting to return from SMM) <sup>b,c</sup>
26	VM entry with events blocked by MOV SS.
28	Invalid operand to INVEPT/INVVPID.

**NOTES:**

- a. Earlier versions of this manual described this error as “VMRESUME with a corrupted VMCS”.
- b. VM-entry checks on control fields and host-state fields may be performed in any order. Thus, an indication by error number of one cause does not imply that there are not also other errors. Different processors may give different error numbers for the same VMCS.
- c. Error number 7 is not used for VM entries that return from SMM that fail due to invalid VM-execution control fields in the executive VMCS. Error number 25 is used for these cases.





# CHAPTER 6

## SAFER MODE EXTENSIONS REFERENCE

---

### 6.1 OVERVIEW

This chapter describes the Safer Mode Extensions (SMX) for the Intel 64 and IA-32 architectures. Safer Mode Extensions (SMX) provide a programming interface for system software to establish a measured environment within the platform to support trust decisions by end users. The measured environment includes:

- Measured launch of a system executive, referred to as a Measured Launched Environment (MLE)<sup>1</sup>. The system executive may be based on a Virtual Machine Monitor (VMM), a measured VMM is referred to as MVMM<sup>2</sup>.
- Mechanisms to ensure the above measurement is protected and stored in a secure location in the platform.
- Protection mechanisms that allow the VMM to control attempts to modify the VMM

The measurement and protection mechanisms used by a measured environment are supported by the capabilities of an Intel® Trusted Execution Technology (Intel® TXT) platform:

- The SMX are the processor's programming interface in an Intel TXT platform;
- The chipset in an Intel TXT platform provides enforcement of the protection mechanisms;
- Trusted Platform Module (TPM) 1.2 in the platform provides platform configuration registers (PCRs) to store software measurement values.

### 6.2 SMX FUNCTIONALITY

SMX functionality is provided in an Intel 64 processor through the GETSEC instruction via leaf functions. The GETSEC instruction supports multiple leaf functions. Leaf functions are selected by the value in EAX at the time GETSEC is executed. Each GETSEC leaf function is documented separately in the reference pages with a unique mnemonic (even though these mnemonics share the same opcode, 0F 37).

---

1. See *Intel® Trusted Execution Technology Measured Launched Environment Programming Guide*.  
2. An MVMM is sometimes referred to as a measured launched environment (MLE). See Intel® *Trusted Execution Technology Measured Launched Environment Programming Guide*

## 6.2.1 Detecting and Enabling SMX

Software can detect support for SMX operation using the CPUID instruction. If software executes CPUID with 1 in EAX, a value of 1 in bit 6 of ECX indicates support for SMX operation (GETSEC is available), see CPUID instruction for the layout of feature flags of reported by CPUID.01H:ECX.

System software enables SMX operation by setting CR4.SMXE[Bit 14] = 1 before attempting to execute GETSEC. Otherwise, execution of GETSEC results in the processor signaling an invalid opcode exception (#UD).

If the CPUID SMX feature flag is clear (CPUID.01H.ECX[Bit 6] = 0), attempting to set CR4.SMXE[Bit 14] results in a general protection exception.

The IA32\_FEATURE\_CONTROL MSR (at address 03AH) provides feature control bits that configure operation of VMX and SMX. These bits are documented in Table 6-1.

**Table 6-1. Layout of IA32\_FEATURE\_CONTROL**

Bit Position	Description
0	Lock bit (0 = unlocked, 1 = locked). When set to '1' further writes to this MSR are blocked.
1	Enable VMX in SMX operation
2	Enable VMX outside SMX operation
7:3	Reserved
14:8	SENTER Local Function Enables: When set, each bit in the field represents an enable control for a corresponding SENTER function.
15	SENTER Global Enable: Must be set to '1' to enable operation of GETSEC[SENTER]
63:16	Reserved

- Bit 0 is a lock bit. If the lock bit is clear, an attempt to execute VMXON will cause a general-protection exception. Attempting to execute GETSEC[SENTER] when the lock bit is clear will also cause a general-protection exception. If the lock bit is set, WRMSR to the IA32\_FEATURE\_CONTROL MSR will cause a general-protection exception. Once the lock bit is set, the MSR cannot be modified until a power-on reset. System BIOS can use this bit to provide a setup option for BIOS to disable support for VMX, SMX or both VMX and SMX.
- Bit 1 enables VMX in SMX operation (between executing the SENTER and SEXIT leaves of GETSEC). If this bit is clear, an attempt to execute VMXON in SMX will cause a general-protection exception if executed in SMX operation. Attempts to set this bit on logical processors that do not support both VMX operation (Chapter 5, "VMX Instruction Reference") and SMX operation cause general-protection exceptions.

- Bit 2 enables VMX outside SMX operation. If this bit is clear, an attempt to execute VMXON will cause a general-protection exception if executed outside SMX operation. Attempts to set this bit on logical processors that do not support VMX operation cause general-protection exceptions.
- Bits 8 through 14 specify enabled functionality of the SENTER leaf function. Each bit in the field represents an enable control for a corresponding SENTER function. Only enabled SENTER leaf functionality can be used when executing SENTER.
- Bits 15 specify global enable of all SENTER functionalities.

## 6.2.2 SMX Instruction Summary

System software must first query for available GETSEC leaf functions by executing GETSEC[CAPABILITIES]. The CAPABILITIES leaf function returns a bit map of available GETSEC leaves. An attempt to execute an unsupported leaf index results in an undefined opcode (#UD) exception.

### 6.2.2.1 GETSEC[CAPABILITIES]

The SMX functionality provides an architectural interface for newer processor generations to extend SMX capabilities. Specifically, the GETSEC instruction provides a capability leaf function for system software to discover the available GETSEC leaf functions that are supported in a processor. Table 6-2 lists the currently available GETSEC leaf functions.

**Table 6-2. GETSEC Leaf Functions**

Index (EAX)	Leaf function	Description
0	CAPABILITIES	Returns the available leaf functions of the GETSEC instruction
1	Undefined	Reserved
2	ENTERACCS	Enter
3	EXITAC	Exit
4	SENER	Launch an MLE
5	SEXIT	Exit the MLE
6	PARAMETERS	Return SMX related parameter information
7	SMCTRL	SMX mode control
8	WAKEUP	Wake up sleeping processors in safer mode
9 - (4G-1)	Undefined	Reserved

### 6.2.2.2 GETSEC[ENTERACCS]

The GETSEC[ENTERACCS] leaf enables authenticated code execution mode. The ENTERACCS leaf function performs an authenticated code module load using the chipset public key as the signature verification. ENTERACCS requires the existence of an Intel® Trusted Execution Technology capable chipset since it unlocks the chipset private configuration register space after successful authentication of the loaded module. The physical base address and size of the authenticated code module are specified as input register values in EBX and ECX, respectively.

While in the authenticated code execution mode, certain processor state properties change. For this reason, the time in which the processor operates in authenticated code execution mode should be limited to minimize impact on external system events.

Upon entry into , the previous paging context is disabled (since the authenticated code module image is specified with physical addresses and can no longer rely upon external memory-based page-table structures).

Prior to executing the GETSEC[ENTERACCS] leaf, system software must ensure the logical processor issuing GETSEC[ENTERACCS] is the boot-strap processor (BSP), as indicated by IA32\_APIC\_BASE.BSP = 1. System software must ensure other logical processors are in a suitable idle state and not marked as BSP.

The GETSEC[ENTERACCS] leaf may be used by different agents to load different authenticated code modules to perform functions related to different aspects of a measured environment, for example system software and Intel® TXT enabled BIOS may use more than one authenticated code modules.

### 6.2.2.3 GETSEC[EXITAC]

GETSEC[EXITAC] takes the processor out of . When this instruction leaf is executed, the contents of the authenticated code execution area are scrubbed and control is transferred to the non-authenticated context defined by a near pointer passed with the GETSEC[EXITAC] instruction.

The authenticated code execution area is no longer accessible after completion of GETSEC[EXITAC]. RBX (or EBX) holds the address of the near absolute indirect target to be taken.

### 6.2.2.4 GETSEC[SENDER]

The GETSEC[SENDER] leaf function is used by the initiating logical processor (ILP) to launch an MLE. GETSEC[SENDER] can be considered a superset of the ENTERACCS leaf, because it enters as part of the measured environment launch.

Measured environment startup consists of the following steps:

- the ILP rendezvous the responding logical processors (RLPs) in the platform into a controlled state (At the completion of this handshake, all the RLPs except for

the ILP initiating the measured environment launch are placed in a newly defined SENTER sleep state).

- Load and authenticate the authenticated code module required by the measured environment, and enter authenticated code execution mode.
- Verify and lock certain system configuration parameters.
- Measure the dynamic root of trust and store into the PCRs in TPM.
- Transfer control to the MLE with interrupts disabled.

Prior to executing the GETSEC[SENDER] leaf, system software must ensure the platform's TPM is ready for access and the ILP is the boot-strap processor (BSP), as indicated by IA32\_APIC\_BASE.BSP. System software must ensure other logical processors (RLPs) are in a suitable idle state and not marked as BSP.

System software launching a measurement environment is responsible for providing a proper authenticate code module address when executing GETSEC[SENDER]. The AC module responsible for the launch of a measured environment and loaded by GETSEC[SENDER] is referred to as SINIT. See *Intel® Trusted Execution Technology Measured Launched Environment Programming Guide* for additional information on system software requirements prior to executing GETSEC[SENDER].

### 6.2.2.5 GETSEC[SEXIT]

System software exits the measured environment by executing the instruction GETSEC[SEXIT] on the ILP. This instruction rendezvous the responding logical processors in the platform for exiting from the measured environment. External events (if left masked) are unmasked and Intel® TXT-capable chipset's private configuration space is re-locked.

### 6.2.2.6 GETSEC[PARAMETERS]

The GETSEC[PARAMETERS] leaf function is used to report attributes, options and limitations of SMX operation. Software uses this leaf to identify operating limits or additional options.

The information reported by GETSEC[PARAMETERS] may require executing the leaf multiple times using EBX as an index. If the GETSEC[PARAMETERS] instruction leaf or if a specific parameter field is not available, then SMX operation should be interpreted to use the default limits of respective GETSEC leaves or parameter fields defined in the GETSEC[PARAMETERS] leaf.

### 6.2.2.7 GETSEC[SMCTRL]

The GETSEC[SMCTRL] leaf function is used for providing additional control over specific conditions associated with the SMX architecture. An input register is supported for selecting the control operation to be performed. See the specific leaf description for details on the type of control provided.

### 6.2.2.8 GETSEC[WAKEUP]

Responding logical processors (RLPs) are placed in the SENTER sleep state after the initiating logical processor executes GETSEC[SENDER]. The ILP can wake up RLPs to join the measured environment by using GETSEC[WAKEUP]. When the RLPs in SENTER sleep state wake up, these logical processors begin execution at the entry point defined in a data structure held in system memory (pointed to by a chipset register LT.MLE.JOIN) in TXT configuration space.

## 6.2.3 Measured Environment and SMX

This section gives a simplified view of a representative life cycle of a measured environment that is launched by a system executive using SMX leaf functions. *Intel® Trusted Execution Technology Measured Launched Environment Programming Guide* provides more detailed examples of using SMX and chipset resources (including chipset registers, Trusted Platform Module) to launch an MVMM.

The life cycle starts with the system executive (an OS, an OS loader, and so forth) loading the MLE and SINIT AC module into available system memory. The system executive must validate and prepare the platform for the measured launch. When the platform is properly configured, the system executive executes GETSEC[SENDER] on the initiating logical processor (ILP) to rendezvous the responding logical processors into an SENTER sleep state, the ILP then enters into using the SINIT AC module. In a multi-threaded or multi-processing environment, the system executive must ensure that other logical processors are already in an idle loop, or asleep (such as after executing HLT) before executing GETSEC[SENDER].

After the GETSEC[SENDER] rendezvous handshake is performed between all logical processors in the platform, the ILP loads the chipset authenticated code module (SINIT) and performs an authentication check. If the check passes, the processor hashes the SINIT AC module and stores the result into TPM PCR 17. It then switches execution context to the SINIT AC module. The SINIT AC module will perform a number of platform operations, including: verifying the system configuration, protecting the system memory used by the MLE from I/O devices capable of DMA, producing a hash of the MLE, storing the hash value in TPM PCR 18, and various other operations. When SINIT completes execution, it executes the GETSEC[EXITAC] instruction and transfers control the MLE at the designated entry point.

Upon receiving control from the SINIT AC module, the MLE must establish its protection and isolation controls before enabling DMA and interrupts and transferring control to other software modules. It must also wakeup the RLPs from their SENTER sleep state using the GETSEC[WAKEUP] instruction and bring them into its protection and isolation environment.

While executing in a measured environment, the MVMM can access the Trusted Platform Module (TPM) in locality 2. The MVMM has complete access to all TPM commands and may use the TPM to report current measurement values or use the measurement values to protect information such that only when the platform config-

uration registers (PCRs) contain the same value is the information released from the TPM. This protection mechanism is known as sealing.

A measured environment shutdown is ultimately completed by executing GETSEC[SEXIT]. Prior to this step system software is responsible for scrubbing sensitive information left in the processor caches, system memory.

## 6.3 GETSEC LEAF FUNCTIONS

This section provides detailed descriptions of each leaf function of the GETSEC instruction. GETSEC is available only if CPUID.01H:ECX[Bit 6] = 1. This indicates the availability of SMX and the GETSEC instruction. Before GETSEC can be executed, SMX must be enabled by setting CR4.SMXE[Bit 14] = 1.

A GETSEC leaf can only be used if it is shown to be available as reported by the GETSEC[CAPABILITIES] function. Attempts to access a GETSEC leaf index not supported by the processor, or if CR4.SMXE is 0, results in the signaling of an undefined opcode exception.

All GETSEC leaf functions are available in protected mode, including the compatibility sub-mode of IA-32e mode and the 64-bit sub-mode of IA-32e mode. Unless otherwise noted, the behavior of all GETSEC functions and interactions related to the measured environment are independent of IA-32e mode. This also applies to the interpretation of register widths<sup>1</sup> passed as input parameters to GETSEC functions and to register results returned as output parameters.

The GETSEC functions ENTERACCS, SENTER, SEXIT, and WAKEUP require a Intel® TXT capable-chipset to be present in the platform. The GETSEC[CAPABILITIES] returned bit vector in position 0 indicates an Intel® TXT-capable chipset has been sampled present<sup>2</sup> by the processor.

The processor's operating mode also affects the execution of the following GETSEC leaf functions: SMCTRL, ENTERACCS, EXITAC, SENTER, SEXIT, and WAKEUP. These functions are only allowed in protected mode at CPL = 0. They are not allowed while in SMM in order to prevent potential intra-mode conflicts. Further execution qualifications exist to prevent potential architectural conflicts (for example: nesting of the measured environment or authenticated code execution mode). See the definitions of the GETSEC leaf functions for specific requirements.

- 
1. This chapter uses the 64-bit notation RAX, RIP, RSP, RFLAGS, etc. for processor registers because processors that support SMX also support Intel 64 Architecture. The MVM can be launched in IA-32e mode or outside IA-32e mode. The 64-bit notation of processor registers also refer to its 32-bit forms if SMX is used in 32-bit environment. In some places, notation such as EAX is used to refer specifically to lower 32 bits of the indicated register
  2. Sampled present means that the processor sent a message to the chipset and the chipset responded that it (a) knows about the message and (b) is capable of executing SENTER. This means that the chipset CAN support Intel® TXT, and is configured and WILLING to support it.

## SAFER MODE EXTENSIONS REFERENCE

For the purpose of performance monitor counting, the execution of GETSEC functions is counted as a single instruction with respect to retired instructions. The response by a responding logical processor (RLP) to messages associated with GETSEC[SENDER] or GTSEC[SEXIT] is transparent to the retired instruction count on the ILP.



## GETSEC[CAPABILITIES] - Report the SMX Capabilities

Opcode	Instruction	Description
0F 37 (EAX = 0)	GETSEC[CAPABILITIES]	Report the SMX capabilities. The capabilities index is input in EBX with the result returned in EAX.

### Description

The GETSEC[CAPABILITIES] function returns a bit vector of supported GETSEC leaf functions. The CAPABILITIES leaf of GETSEC is selected with EAX set to 0 at entry. EBX is used as the selector for returning the bit vector field in EAX. GETSEC[CAPABILITIES] may be executed at all privilege levels, but the CR4.SMXE bit must be set or an undefined opcode exception (#UD) is returned.

With EBX = 0 upon execution of GETSEC[CAPABILITIES], EAX returns the a bit vector representing status on the presence of a Intel® TXT-capable chipset and the first 30 available GETSEC leaf functions. The format of the returned bit vector is provided in Table 6-3.

If bit 0 is set to 1, then an Intel® TXT-capable chipset has been sampled present by the processor. If bits in the range of 1-30 are set, then the corresponding GETSEC leaf function is available. If the bit value at a given bit index is 0, then the GETSEC leaf function corresponding to that index is unsupported and attempted execution results in a #UD.

Bit 31 of EAX indicates if further leaf indexes are supported. If the Extended Leafs bit 31 is set, then additional leaf functions are accessed by repeating GETSEC[CAPABILITIES] with EBX incremented by one. When the most significant bit of EAX is not set, then additional GETSEC leaf functions are not supported; indexing EBX to a higher value results in EAX returning zero.

**Table 6-3. Getsec Capability Result Encoding (EBX = 0)**

Field	Bit position	Description
Chipset Present	0	Intel® TXT-capable chipset is present
Undefined	1	Reserved
ENTERACCS	2	GETSEC[ENTERACCS] is available
EXITAC	3	GETSEC[EXITAC] is available
SENER	4	GETSEC[SENER] is available
SEXIT	5	GETSEC[SEXIT] is available

**Table 6-3. Getsec Capability Result Encoding (EBX = 0) (Contd.)**

Field	Bit position	Description
PARAMETERS	6	GETSEC[PARAMETERS] is available
SMCTRL	7	GETSEC[SMCTRL] is available
WAKEUP	8	GETSEC[WAKEUP] is available
Undefined	30:9	Reserved
Extended Leafs	31	Reserved for extended information reporting of GETSEC capabilities

**Operation**

```

IF (CR4.SMXE=0)
    THEN #UD;
ELSIF (in VMX non-root operation)
    THEN VM Exit (reason="GETSEC instruction");
IF (EBX=0) THEN
    BitVector ← 0;
    IF (TXT chipset present)
        BitVector[Chipset present] ← 1;
    IF (ENTERACCS Available)
        THEN BitVector[ENTERACCS] ← 1;
    IF (EXITAC Available)
        THEN BitVector[EXITAC] ← 1;
    IF (SENDER Available)
        THEN BitVector[SENDER] ← 1;
    IF (SEXIT Available)
        THEN BitVector[SEXIT] ← 1;
    IF (PARAMETERS Available)
        THEN BitVector[PARAMETERS] ← 1;
    IF (SMCTRL Available)
        THEN BitVector[SMCTRL] ← 1;
    IF (WAKEUP Available)
        THEN BitVector[WAKEUP] ← 1;
    EAX ← BitVector;
ELSE
    EAX ← 0;
END;;

```

**Flags Affected**

None

## Use of Prefixes

LOCK	Causes #UD
REP*	Cause #UD (includes REPNE/REPNZ and REP/REPE/REPZ)
Operand size	Causes #UD
Segment overrides	Ignored
Address size	Ignored
REX	Ignored

## Protected Mode Exceptions

#UD IF CR4.SMXE = 0.

## Real-Address Mode Exceptions

#UD IF CR4.SMXE = 0.

## Virtual-8086 Mode Exceptions

#UD IF CR4.SMXE = 0.

## Compatibility Mode Exceptions

#UD IF CR4.SMXE = 0.

## 64-Bit Mode Exceptions

#UD IF CR4.SMXE = 0.

## VM-exit Condition

Reason (GETSEC) IF in VMX non-root operation.

## GETSEC[ENTERACCS] - Execute Authenticated Chipset Code

Opcode	Instruction	Description
0F 37 (EAX = 2)	GETSEC[ENTERACCS]	Enter authenticated code execution mode. EBX holds the authenticated code module physical base address. ECX holds the authenticated code module size (bytes).

### Description

The GETSEC[ENTERACCS] function loads, authenticates and executes an authenticated code module using an Intel® TXT platform chipset's public key. The ENTERACCS leaf of GETSEC is selected with EAX set to 2 at entry.

There are certain restrictions enforced by the processor for the execution of the GETSEC[ENTERACCS] instruction:

- Execution is not allowed unless the processor is in protected mode or IA-32e mode with CPL = 0 and EFLAGS.VM = 0.
- Processor cache must be available and not disabled, that is, CR0.CD and CR0.NW bits must be 0.
- For processor packages containing more than one logical processor, CR0.CD is checked to ensure consistency between enabled logical processors.
- For enforcing consistency of operation with numeric exception reporting using Interrupt 16, CR0.NE must be set.
- An Intel TXT-capable chipset must be present as communicated to the processor by sampling of the power-on configuration capability field after reset.
- The processor can not already be in authenticated code execution mode as launched by a previous GETSEC[ENTERACCS] or GETSEC[SENDER] instruction without a subsequent exiting using GETSEC[EXITAC]).
- To avoid potential operability conflicts between modes, the processor is not allowed to execute this instruction if it currently is in SMM or VMX operation.
- To insure consistent handling of SIPI messages, the processor executing the GETSEC[ENTERACCS] instruction must also be designated the BSP (boot-strap processor) as defined by A32\_APIC\_BASE.BSP (Bit 8).

Failure to conform to the above conditions results in the processor signaling a general protection exception.

Prior to execution of the ENTERACCS leaf, other logical processors, i.e. RLPs, in the platform must be:

- idle in a wait-for-SIPI state (as initiated by an INIT assertion or through reset for non-BSP designated processors), or
- in the SENTER sleep state as initiated by a GETSEC[SENDER] from the initiating logical processor (ILP).

If other logical processor(s) in the same package are not idle in one of these states, execution of ENTERACCS signals a general protection exception. The same requirement and action applies if the other logical processor(s) of the same package do not have `CR0.CD = 0`.

A successful execution of ENTERACCS results in the ILP entering an authenticated code execution mode. Prior to reaching this point, the processor performs several checks. These include:

- Establish and check the location and size of the specified authenticated code module to be executed by the processor.
- Inhibit the ILP's response to the external events: INIT, A20M, NMI and SMI.
- Broadcast a message to enable protection of memory and I/O from other processor agents.
- Load the designated code module into an authenticated code execution area.
- Isolate the contents of the authenticated code execution area from further state modification by external agents.
- Authenticate the authenticated code module.
- Initialize the initiating logical processor state based on information contained in the authenticated code module header.
- Unlock the Intel® TXT-capable chipset private configuration space and TPM locality 3 space.
- Begin execution in the authenticated code module at the defined entry point.

The `GETSEC[ENTERACCS]` function requires two additional input parameters in the general purpose registers `EBX` and `ECX`. `EBX` holds the authenticated code (AC) module physical base address (the AC module must reside below 4 GBytes in physical address space) and `ECX` holds the AC module size (in bytes). The physical base address and size are used to retrieve the code module from system memory and load it into the internal authenticated code execution area. The base physical address is checked to verify it is on a modulo-4096 byte boundary. The size is verified to be a multiple of 64, that it does not exceed the internal authenticated code execution area capacity (as reported by `GETSEC[CAPABILITIES]`), and that the top address of the AC module does not exceed 32 bits. An error condition results in an abort of the authenticated code execution launch and the signaling of a general protection exception.

As an integrity check for proper processor hardware operation, execution of `GETSEC[ENTERACCS]` will also check the contents of all the machine check status registers (as reported by the MSRs `IA32_MCi_STATUS`) for any valid uncorrectable error condition. In addition, the global machine check status register `IA32_MCG_STATUS` `MCIP` bit must be cleared and the `IERR` processor package pin (or its equivalent) must not be asserted, indicating that no machine check exception processing is currently in progress. These checks are performed prior to initiating the load of the authenticated code module. Any outstanding valid uncorrectable machine check error condition present in these status registers at this point will result in the processor signaling a general protection violation.

The ILP masks the response to the assertion of the external signals INIT#, A20M, NMI#, and SMI#. This masking remains active until optionally unmasked by GETSEC[EXITAC] (this defined unmasking behavior assumes GETSEC[ENTERACCS] was not executed by a prior GETSEC[SENDER]). The purpose of this masking control is to prevent exposure to existing external event handlers that may not be under the control of the authenticated code module.

The ILP sets an internal flag to indicate it has entered authenticated code execution mode. The state of the A20M pin is likewise masked and forced internally to a de-asserted state so that any external assertion is not recognized during authenticated code execution mode.

To prevent other (logical) processors from interfering with the ILP operating in authenticated code execution mode, memory (excluding implicit write-back transactions) access and I/O originating from other processor agents are blocked. This protection starts when the ILP enters into authenticated code execution mode. Only memory and I/O transactions initiated from the ILP are allowed to proceed. Exiting authenticated code execution mode is done by executing GETSEC[EXITAC]. The protection of memory and I/O activities remains in effect until the ILP executes GETSEC[EXITAC].

Prior to launching the authenticated execution module using GETSEC[ENTERACCS] or GETSEC[SENDER], the processor's MTRRs (Memory Type Range Registers) must first be initialized to map out the authenticated RAM addresses as WB (writeback). Failure to do so may affect the ability for the processor to maintain isolation of the loaded authenticated code module. If the processor detected this requirement is not met, it will signal an Intel® TXT reset condition with an error code during the loading of the authenticated code module.

While physical addresses within the load module must be mapped as WB, the memory type for locations outside of the module boundaries must be mapped to one of the supported memory types as returned by GETSEC[PARAMETERS] (or UC as default).

To conform to the minimum granularity of MTRR MSRs for specifying the memory type, authenticated code RAM (ACRAM) is allocated to the processor in 4096 byte granular blocks. If an AC module size as specified in ECX is not a multiple of 4096 then the processor will allocate up to the next 4096 byte boundary for mapping as ACRAM with indeterminate data. This pad area will not be visible to the authenticated code module as external memory nor can it depend on the value of the data used to fill the pad area.

At the successful completion of GETSEC[ENTERACCS], the architectural state of the processor is partially initialized from contents held in the header of the authenticated code module. The processor GDTR, CS, and DS selectors are initialized from fields within the authenticated code module. Since the authenticated code module must be relocatable, all address references must be relative to the authenticated code module base address in EBX. The processor GDTR base value is initialized to the AC module header field GDTBasePtr + module base address held in EBX and the GDTR limit is set to the value in the GDTLimit field. The CS selector is initialized to the AC module header SegSel field, while the DS selector is initialized to CS + 8. The segment

descriptor fields are implicitly initialized to  $\text{BASE}=0$ ,  $\text{LIMIT}=\text{FFFFFh}$ ,  $\text{G}=1$ ,  $\text{D}=1$ ,  $\text{P}=1$ ,  $\text{S}=1$ , read/write access for DS, and execute/read access for CS. The processor begins the authenticated code module execution with the EIP set to the AC module header EntryPoint field + module base address (EBX). The AC module based fields used for initializing the processor state are checked for consistency and any failure results in a shutdown condition.

A summary of the register state initialization after successful completion of GETSEC[ENTERACCS] is given for the processor in Table 6-4. The paging is disabled upon entry into authenticated code execution mode. The authenticated code module is loaded and initially executed using physical addresses. It is up to the system software after execution of GETSEC[ENTERACCS] to establish a new (or restore its previous) paging environment with an appropriate mapping to meet new protection requirements. EBP is initialized to the authenticated code module base physical address for initial execution in the authenticated environment. As a result, the authenticated code can reference EBP for relative address based references, given that the authenticated code module must be position independent.

**Table 6-4. Register State Initialization after GETSEC[ENTERACCS]**

Register State	Initialization Status	Comment
CRO	$\text{PG}\leftarrow 0$ , $\text{AM}\leftarrow 0$ , $\text{WP}\leftarrow 0$ : Others unchanged	Paging, Alignment Check, Write-protection are disabled
CR4	$\text{MCE}\leftarrow 0$ : Others unchanged	Machine Check Exceptions Disabled
EFLAGS	00000002H	
IA32_EFER	0H	IA-32e mode disabled
EIP	AC.base + EntryPoint	AC.base is in EBX as input to GETSEC[ENTERACCS]
[E R]BX	Pre-ENTERACCS state: Next [E R]IP prior to GETSEC[ENTERACCS]	Carry forward 64-bit processor state across GETSEC[ENTERACCS]
ECX	Pre-ENTERACCS state: [31:16]=GDTR.limit; [15:0]=CS.sel	Carry forward processor state across GETSEC[ENTERACCS]
[E R]DX	Pre-ENTERACCS state: GDTR base	Carry forward 64-bit processor state across GETSEC[ENTERACCS]
EBP	AC.base	
CS	Sel=[SegSel], base=0, limit=FFFFFh, G=1, D=1, AR=9BH	
DS	Sel=[SegSel] +8, base=0, limit=FFFFFh, G=1, D=1, AR=93H	

**Table 6-4. Register State Initialization after GETSEC[ENTERACCS] (Contd.)**

Register State	Initialization Status	Comment
GDTR	Base= AC.base (EBX) + [GDTBasePtr], Limit=[GDTLimit]	
DR7	00000400H	
IA32_DEBUGCTL	0H	
IA32_MISC_ENA BLE	see Table 6-5 for example	The number of initialized fields may change due to processor implementation

The segmentation related processor state that has not been initialized by GETSEC[ENTERACCS] requires appropriate initialization before use. Since a new GDT context has been established, the previous state of the segment selector values held in ES, SS, FS, GS, TR, and LDTR might not be valid.

The MSR IA32\_EFER is also unconditionally cleared as part of the processor state initialized by ENTERACCS. Since paging is disabled upon entering authenticated code execution mode, a new paging environment will have to be reestablished in order to establish IA-32e mode while operating in authenticated code execution mode.

Debug exception and trap related signaling is also disabled as part of GETSEC[ENTERACCS]. This is achieved by resetting DR7, TF in EFLAGS, and the MSR IA32\_DEBUGCTL. These debug functions are free to be re-enabled once supporting exception handler(s), descriptor tables, and debug registers have been properly initialized following entry into authenticated code execution mode. Also, any pending single-step trap condition will have been cleared upon entry into this mode.

The IA32\_MISC\_ENABLE MSR is initialized upon entry into authenticated execution mode. Certain bits of this MSR are preserved because preserving these bits may be important to maintain previously established platform settings (See the footnote for Table 6-5.). The remaining bits are cleared for the purpose of establishing a more consistent environment for the execution of authenticated code modules. One of the impacts of initializing this MSR is any previous condition established by the MONITOR instruction will be cleared.

To support the possible return to the processor architectural state prior to execution of GETSEC[ENTERACCS], certain critical processor state is captured and stored in the general-purpose registers at instruction completion. [E|R]BX holds effective address ([E|R]IP) of the instruction that would execute next after GETSEC[ENTERACCS], ECX[15:0] holds the CS selector value, ECX[31:16] holds the GDTR limit field, and [E|R]DX holds the GDTR base field. The subsequent authenticated code can preserve the contents of these registers so that this state can be manually restored if needed, prior to exiting authenticated code execution mode with GETSEC[EXITAC]. For the processor state after exiting authenticated code execution mode, see the description of GETSEC[SEXIT].



**Table 6-5. IA32\_MISC\_ENABLE MSR Initialization<sup>1</sup> by ENTERACCS and SENTER**

Field	Bit position	Description
Fast strings enable	0	Clear to 0
FOPCODE compatibility mode enable	2	Clear to 0
Thermal monitor enable	3	Set to 1 if other thermal monitor capability is not enabled. <sup>2</sup>
Split-lock disable	4	Clear to 0
Bus lock on cache line splits disable	8	Clear to 0
Hardware prefetch disable	9	Clear to 0
GV1/2 legacy enable	15	Clear to 0
MONITOR/MWAIT s/m enable	18	Clear to 0
Adjacent sector prefetch disable	19	Clear to 0

**NOTES:**

1. The number of IA32\_MISC\_ENABLE fields that are initialized may vary due to processor implementations.
2. ENTERACCS (and SENTER) initialize the state of processor thermal throttling such that at least a minimum level is enabled. If thermal throttling is already enabled when executing one of these GETSEC leaves, then no change in the thermal throttling control settings will occur. If thermal throttling is disabled, then it will be enabled via setting of the thermal throttle control bit 3 as a result of executing these GETSEC leaves.

The IDTR will also require reloading with a new IDT context after entering authenticated code execution mode, before any exceptions or the external interrupts INTR and NMI can be handled. Since external interrupts are re-enabled at the completion of authenticated code execution mode (as terminated with EXITAC), it is recommended that a new IDT context be established before this point. Until such a new IDT context is established, the programmer must take care in not executing an INT n instruction or any other operation that would result in an exception or trap signaling.

Prior to completion of the GETSEC[ENTERACCS] instruction and after successful authentication of the AC module, the private configuration space of the Intel TXT chipset is unlocked. The authenticated code module alone can gain access to this normally restricted chipset state for the purpose of securing the platform.

Once the authenticated code module is launched at the completion of GETSEC[ENTERACCS], it is free to enable interrupts by setting EFLAGS.IF and enable NMI by execution of IRET. This presumes that it has re-established interrupt handling support through initialization of the IDT, GDT, and corresponding interrupt handling code.

### Operation in a Uni-Processor Platform

(\* The state of the internal flag ACMODEFLAG persists across instruction boundary \*)

```

IF (CR4.SMXE=0)
    THEN #UD;
ELSIF (in VMX non-root operation)
    THEN VM Exit (reason="GETSEC instruction");
ELSIF (GETSEC leaf unsupported)
    THEN #UD;
ELSIF ((in VMX operation) or
        (CR0.PE=0) or (CR0.CD=1) or (CR0.NW=1) or (CR0.NE=0) or
        (CPL>0) or (EFLAGS.VM=1) or
        (IA32_APIC_BASE.BSP=0) or
        (TXT chipset not present) or
        (ACMODEFLAG=1) or (IN_SMM=1))
    THEN #GP(0);
IF (GETSEC[PARAMETERS].Parameter_Type = 5, MCA_Handling (bit 6) = 0)
    FOR I = 0 to IA32_MCG_CAP.COUNT-1 DO
        IF (IA32_MCG[I]_STATUS = uncorrectable error)
            THEN #GP(0);
    OD;
FI;
IF (IA32_MCG_STATUS.MCIP=1) or (IERR pin is asserted)
    THEN #GP(0);
ACBASE← EBX;
ACSIZE← ECX;
IF (((ACBASE MOD 4096) != 0) or ((ACSIZE MOD 64) != 0) or (ACSIZE < minimum module size) OR
    (ACSIZE > authenticated RAM capacity)) or ((ACBASE+ACSIZE) > (2^32 -1)))
    THEN #GP(0);
IF (secondary thread(s) CR0.CD = 1) or ((secondary thread(s) NOT(wait-for-SIPI)) and
    (secondary thread(s) not in SENTER sleep state)
    THEN #GP(0);
Mask SMI, INIT, A20M, and NMI external pin events;
IA32_MISC_ENABLE← (IA32_MISC_ENABLE & MASK_CONST*)
(* The hexadecimal value of MASK_CONST may vary due to processor implementations *)
A20M← 0;
IA32_DEBUGCTL← 0;
Invalidate processor TLB(s);
Drain Outgoing Transactions;

```

```

ACMODEFLAG← 1;
SignalTXTMessage(ProcessorHold);
Load the internal ACRAM based on the AC module size;
(* Ensure that all ACRAM loads hit Write Back memory space *)
IF (ACRAM memory type != WB)
    THEN TXT-SHUTDOWN(#BadACMMType);
IF (AC module header version isnot supported) OR (ACRAM[ModuleType] <> 2)
    THEN TXT-SHUTDOWN(#UnsupportedACM);
(* Authenticate the AC Module and shutdown with an error if it fails *)
KEY← GETKEY(ACRAM, ACBASE);
KEYHASH← HASH(KEY);
CSKEYHASH← READ(TXT.PUBLIC.KEY);
IF (KEYHASH <> CSKEYHASH)
    THEN TXT-SHUTDOWN(#AuthenticateFail);
SIGNATURE← DECRYPT(ACRAM, ACBASE, KEY);
(* The value of SIGNATURE_LEN_CONST is implementation-specific*)
FOR I=0 to SIGNATURE_LEN_CONST - 1 DO
    ACRAM[SCRATCH.I]← SIGNATURE[I];
COMPUTEDSIGNATURE← HASH(ACRAM, ACBASE, ACSIZE);
FOR I=0 to SIGNATURE_LEN_CONST - 1 DO
    ACRAM[SCRATCH.SIGNATURE_LEN_CONST+I]← COMPUTEDSIGNATURE[I];
IF (SIGNATURE<>COMPUTEDSIGNATURE)
    THEN TXT-SHUTDOWN(#AuthenticateFail);
ACMCONTROL← ACRAM[CodeControl];
IF ((ACMCONTROL.0 = 0) and (ACMCONTROL.1 = 1) and (snoop hit to modified line detected on
ACRAM load))
    THEN TXT-SHUTDOWN(#UnexpectedHITM);
IF (ACMCONTROL reserved bits are set)
    THEN TXT-SHUTDOWN(#BadACMFormat);
IF ((ACRAM[GDTBasePtr] < (ACRAM[HeaderLen] * 4 + Scratch_size)) OR
((ACRAM[GDTBasePtr] + ACRAM[GDTLimit]) >= ACSIZE))
    THEN TXT-SHUTDOWN(#BadACMFormat);
IF ((ACMCONTROL.0 = 1) and (ACMCONTROL.1 = 1) and (snoop hit to modified line detected on
ACRAM load))
    THEN ACEntryPoint← ACBASE+ACRAM[ErrorEntryPoint];
ELSE
    ACEntryPoint← ACBASE+ACRAM[EntryPoint];
IF ((ACEntryPoint >= ACSIZE) OR (ACEntryPoint < (ACRAM[HeaderLen] * 4 + Scratch_size)))THEN
    TXT-SHUTDOWN(#BadACMFormat);
IF (ACRAM[GDTLimit] & FFFF0000h)
    THEN TXT-SHUTDOWN(#BadACMFormat);
IF ((ACRAM[SegSel] > (ACRAM[GDTLimit] - 15)) OR (ACRAM[SegSel] < 8))
    THEN TXT-SHUTDOWN(#BadACMFormat);
IF ((ACRAM[SegSel].TI=1) OR (ACRAM[SegSel].RPLI=0))

```

```

    THEN TXT-SHUTDOWN(#BadACMFormat);
CRO.[PG.AM.WP]← 0;
CR4.MCE← 0;
EFLAGS← 00000002h;
IA32_EFER← 0h;
[E|R]BX← [E|R]IP of the instruction after GETSEC[ENTERACCS];
ECX← Pre-GETSEC[ENTERACCS] GDT.limit:CS.sel;
[E|R]DX← Pre-GETSEC[ENTERACCS] GDT.base;
EBP← ACBASE;
GDTR.BASE← ACBASE+ACRAM[GDTBasePtr];
GDTR.LIMIT← ACRAM[GDTLimit];
CS.SEL← ACRAM[SegSel];
CS.BASE← 0;
CS.LIMIT← FFFFFFFh;
CS.G← 1;
CS.D← 1;
CS.AR← 9Bh;
DS.SEL← ACRAM[SegSel]+8;
DS.BASE← 0;
DS.LIMIT← FFFFFFFh;
DS.G← 1;
DS.D← 1;
DS.AR← 93h;
DR7← 00000400h;
IA32_DEBUGCTL← 0;
SignalTXTMsg(OpenPrivate);
SignalTXTMsg(OpenLocality3);
EIP← ACEntryPoint;
END;

```

### Flags Affected

All flags are cleared.

### Use of Prefixes

LOCK	Causes #UD
REP*	Cause #UD (includes REPNE/REPNZ and REP/REPE/REPZ)
Operand size	Causes #UD
Segment overrides	Ignored
Address size	Ignored
REX	Ignored

## Protected Mode Exceptions

#UD	<p>If CR4.SMXE = 0.</p> <p>If GETSEC[ENTERACCS] is not reported as supported by GETSEC[CAPABILITIES].</p>
#GP(0)	<p>If CR0.CD = 1 or CR0.NW = 1 or CR0.NE = 0 or CR0.PE = 0 or CPL &gt; 0 or EFLAGS.VM = 1.</p> <p>If a Intel® TXT-capable chipset is not present.</p> <p>If in VMX root operation.</p> <p>If the initiating processor is not designated as the bootstrap processor via the MSR bit IA32_APIC_BASE.BSP.</p> <p>If the processor is already in authenticated code execution mode.</p> <p>If the processor is in SMM.</p> <p>If a valid uncorrectable machine check error is logged in IA32_MC[I]_STATUS.</p> <p>If the authenticated code base is not on a 4096 byte boundary.</p> <p>If the authenticated code size &gt; processor internal authenticated code area capacity.</p> <p>If the authenticated code size is not modulo 64.</p> <p>If other enabled logical processor(s) of the same package CR0.CD = 1.</p> <p>If other enabled logical processor(s) of the same package are not in the wait-for-SIPI or SENTER sleep state.</p>

## Real-Address Mode Exceptions

#UD	<p>If CR4.SMXE = 0.</p> <p>If GETSEC[ENTERACCS] is not reported as supported by GETSEC[CAPABILITIES].</p>
#GP(0)	GETSEC[ENTERACCS] is not recognized in real-address mode.

## Virtual-8086 Mode Exceptions

#UD	<p>If CR4.SMXE = 0.</p> <p>If GETSEC[ENTERACCS] is not reported as supported by GETSEC[CAPABILITIES].</p>
#GP(0)	GETSEC[ENTERACCS] is not recognized in virtual-8086 mode.

## Compatibility Mode Exceptions

All protected mode exceptions apply.

#GP	IF AC code module does not reside in physical address below $2^{32} - 1$ .
-----	--

## 64-Bit Mode Exceptions

All protected mode exceptions apply.

#GP IF AC code module does not reside in physical address below  $2^{32} - 1$ .

## VM-exit Condition

Reason (GETSEC) IF in VMX non-root operation.

## GETSEC[EXITAC]—Exit Authenticated Code Execution Mode

Opcode	Instruction	Description
0F 37 (EAX=3)	GETSEC[EXITAC]	Exit authenticated code execution mode. RBX holds the Near Absolute Indirect jump target and EDX hold the exit parameter flags

### Description

The GETSEC[EXITAC] leaf function exits the ILP out of authenticated code execution mode established by GETSEC[ENTERACCS] or GETSEC[SENDER]. The EXITAC leaf of GETSEC is selected with EAX set to 3 at entry. EBX (or RBX, if in 64-bit mode) holds the near jump target offset for where the processor execution resumes upon exiting authenticated code execution mode. EDX contains additional parameter control information. Currently only an input value of 0 in EDX is supported. All other EDX settings are considered reserved and result in a general protection violation.

GETSEC[EXITAC] can only be executed if the processor is in protected mode with CPL = 0 and EFLAGS.VM = 0. The processor must also be in authenticated code execution mode. To avoid potential operability conflicts between modes, the processor is not allowed to execute this instruction if it is in SMM or in VMX operation. A violation of these conditions results in a general protection violation.

Upon completion of the GETSEC[EXITAC] operation, the processor unmask responses to external event signals INIT#, NMI#, and SMI#. This unmasking is performed conditionally, based on whether the authenticated code execution mode was entered via execution of GETSEC[SENDER] or GETSEC[ENTERACCS]. If the processor is in authenticated code execution mode due to the execution of GETSEC[SENDER], then these external event signals will remain masked. In this case, A20M is kept disabled in the measured environment until the measured environment executes GETSEC[SEXIT]. INIT# is unconditionally unmasked by EXITAC. Note that any events that are pending, but have been blocked while in authenticated code execution mode, will be recognized at the completion of the GETSEC[EXITAC] instruction if the pin event is unmasked.

The intent of providing the ability to optionally leave the pin events SMI#, and NMI# masked is to support the completion of a measured environment bring-up that makes use of VMX. In this envisioned security usage scenario, these events will remain masked until an appropriate virtual machine has been established in order to field servicing of these events in a safer manner. Details on when and how events are masked and unmasked in VMX operation are described in *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*. It should be cautioned that if no VMX environment is to be activated following GETSEC[EXITAC], that these events will remain masked until the measured environment is exited with GETSEC[SEXIT]. If this is not desired then the GETSEC function SMCTRL(0) can be used for unmasking SMI# in this context. NMI# can be correspondingly unmasked by execution of IRET.

A successful exit of the authenticated code execution mode requires the ILP to perform additional steps as outlined below:

- Invalidate the contents of the internal authenticated code execution area.
- Invalidate processor TLBs.
- Clear the internal processor AC Mode indicator flag.
- Re-lock the TPM locality 3 space.
- Unlock the Intel® TXT-capable chipset memory and I/O protections to allow memory and I/O activity by other processor agents.
- Perform a near absolute indirect jump to the designated instruction location.

The content of the authenticated code execution area is invalidated by hardware in order to protect it from further use or visibility. This internal processor storage area can no longer be used or relied upon after GETSEC[EXITAC]. Data structures need to be re-established outside of the authenticated code execution area if they are to be referenced after EXITAC. Since addressed memory content formerly mapped to the authenticated code execution area may no longer be coherent with external system memory after EXITAC, processor TLBs in support of linear to physical address translation are also invalidated.

Upon completion of GETSEC[EXITAC] a near absolute indirect transfer is performed with EIP loaded with the contents of EBX (based on the current operating mode size). In 64-bit mode, all 64 bits of RBX are loaded into RIP if REX.W precedes GETSEC[EXITAC]. Otherwise RBX is treated as 32 bits even while in 64-bit mode. Conventional CS limit checking is performed as part of this control transfer. Any exception conditions generated as part of this control transfer will be directed to the existing IDT; thus it is recommended that an IDTR should also be established prior to execution of the EXITAC function if there is a need for fault handling. In addition, any segmentation related (and paging) data structures to be used after EXITAC should be re-established or validated by the authenticated code prior to EXITAC.

In addition, any segmentation related (and paging) data structures to be used after EXITAC need to be re-established and mapped outside of the authenticated RAM designated area by the authenticated code prior to EXITAC. Any data structure held within the authenticated RAM allocated area will no longer be accessible after completion by EXITAC.

### Operation

(\* The state of the internal flag ACMODEFLAG and SENTERFLAG persist across instruction boundary \*)

```
IF (CR4.SMXE=0)
    THEN #UD;
```

```
ELSIF ( in VMX non-root operation)
```

```
    THEN VM Exit (reason="GETSEC instruction");
```

```
ELSIF (GETSEC leaf unsupported)
```

```
    THEN #UD;
```

```
ELSIF ((in VMX operation) or ( (in 64-bit mode) and ( RBX is non-canonical) )
```



```

(CR0.PE=0) or (CPL>0) or (EFLAGS.VM=1) or
(ACMODEFLAG=0) or (IN_SMM=1)) or (EDX != 0))
THEN #GP(0);
IF (OperandSize = 32)
  THEN tempEIP← EBX;
ELSIF (OperandSize = 64)
  THEN tempEIP← RBX;
ELSE
  tempEIP← EBX AND 0000FFFFH;
IF (tempEIP > code segment limit)
  THEN #GP(0);
Invalidate ACRAM contents;
Invalidate processor TLB(s);
Drain outgoing messages;
SignalTXTMsg(CloseLocality3);
SignalTXTMsg(LockSMRAM);
SignalTXTMsg(ProcessorRelease);
Unmask INIT;
IF (SENTERFLAG=0)
  THEN Unmask SMI, INIT, NMI, and A20M pin event;
ELSEIF (IA32_SMM_MONITOR_CTL[0] = 0)
  THEN Unmask SMI pin event;
ACMODEFLAG← 0;
EIP← tempEIP;
END;

```

## Flags Affected

None.

## Use of Prefixes

LOCK	Causes #UD
REP*	Cause #UD (includes REPNE/REPNZ and REP/REPE/REPZ)
Operand size	Causes #UD
Segment overrides	Ignored
Address size	Ignored
REX.W	Sets 64-bit mode Operand size attribute

## Protected Mode Exceptions

#UD	If CR4.SMXE = 0. If GETSEC[EXITAC] is not reported as supported by GETSEC[CAPABILITIES].
-----	---

## SAFER MODE EXTENSIONS REFERENCE

#GP(0)                    If CR0.PE = 0 or CPL>0 or EFLAGS.VM = 1.  
If in VMX root operation.  
If the processor is not currently in authenticated code execution mode.  
If the processor is in SMM.  
If any reserved bit position is set in the EDX parameter register.

### Real-Address Mode Exceptions

#UD                      If CR4.SMXE = 0.  
If GETSEC[EXITAC] is not reported as supported by GETSEC[CAPABILITIES].

#GP(0)                    GETSEC[EXITAC] is not recognized in real-address mode.

### Virtual-8086 Mode Exceptions

#UD                      If CR4.SMXE = 0.  
If GETSEC[EXITAC] is not reported as supported by GETSEC[CAPABILITIES].

#GP(0)                    GETSEC[EXITAC] is not recognized in virtual-8086 mode.

### Compatibility Mode Exceptions

All protected mode exceptions apply.

### 64-Bit Mode Exceptions

All protected mode exceptions apply.

#GP(0)                    If the target address in RBX is not in a canonical form.

### VM-Exit Condition

Reason (GETSEC)      If in VMX non-root operation.

## GETSEC[SENDER]—Enter a Measured Environment

Opcode	Instruction	Description
0F 37 (EAX=4)	GETSEC[SENDER]	<p>Launch a measured environment</p> <p>EBX holds the SINIT authenticated code module physical base address.</p> <p>ECX holds the SINIT authenticated code module size (bytes).</p> <p>EDX controls the level of functionality supported by the measured environment launch.</p>

### Description

The GETSEC[SENDER] instruction initiates the launch of a measured environment and places the initiating logical processor (ILP) into the authenticated code execution mode. The SENDER leaf of GETSEC is selected with EAX set to 4 at execution. The physical base address of the AC module to be loaded and authenticated is specified in EBX. The size of the module in bytes is specified in ECX. EDX controls the level of functionality supported by the measured environment launch. To enable the full functionality of the protected environment launch, EDX must be initialized to zero.

The authenticated code base address and size parameters (in bytes) are passed to the GETSEC[SENDER] instruction using EBX and ECX respectively. The ILP evaluates the contents of these registers according to the rules for the AC module address in GETSEC[ENTERACCS]. AC module execution follows the same rules, as set by GETSEC[ENTERACCS].

The launching software must ensure that the TPM.ACCESS\_0.activeLocality bit is clear before executing the GETSEC[SENDER] instruction.

There are restrictions enforced by the processor for execution of the GETSEC[SENDER] instruction:

- Execution is not allowed unless the processor is in protected mode or IA-32e mode with CPL = 0 and EFLAGS.VM = 0.
- Processor cache must be available and not disabled using the CR0.CD and NW bits.
- For enforcing consistency of operation with numeric exception reporting using Interrupt 16, CR0.NE must be set.
- An Intel TXT-capable chipset must be present as communicated to the processor by sampling of the power-on configuration capability field after reset.
- The processor can not be in authenticated code execution mode or already in a measured environment (as launched by a previous GETSEC[ENTERACCS] or GETSEC[SENDER] instruction).
- To avoid potential operability conflicts between modes, the processor is not allowed to execute this instruction if it currently is in SMM or VMX operation.

- To insure consistent handling of SIPI messages, the processor executing the GETSEC[SENDER] instruction must also be designated the BSP (boot-strap processor) as defined by A32\_APIC\_BASE.BSP (Bit 8).
- EDX must be initialized to a setting supportable by the processor. Unless enumeration by the GETSEC[PARAMETERS] leaf reports otherwise, only a value of zero is supported.

Failure to abide by the above conditions results in the processor signaling a general protection violation.

This instruction leaf starts the launch of a measured environment by initiating a rendezvous sequence for all logical processors in the platform. The rendezvous sequence involves the initiating logical processor sending a message (by executing GETSEC[SENDER]) and other responding logical processors (RLPs) acknowledging the message, thus synchronizing the RLP(s) with the ILP.

In response to a message signaling the completion of rendezvous, RLPs clear the bootstrap processor indicator flag (IA32\_APIC\_BASE.BSP) and enter an SENTER sleep state. In this sleep state, RLPs enter an idle processor condition while waiting to be activated after a measured environment has been established by the system executive. RLPs in the SENTER sleep state can only be activated by the GETSEC leaf function WAKEUP in a measured environment.

A successful launch of the measured environment results in the initiating logical processor entering the authenticated code execution mode. Prior to reaching this point, the ILP performs the following steps internally:

- Inhibit processor response to the external events: INIT, A20M, NMI, and SMI.
- Establish and check the location and size of the authenticated code module to be executed by the ILP.
- Check for the existence of an Intel® TXT-capable chipset.
- Verify the current power management configuration is acceptable.
- Broadcast a message to enable protection of memory and I/O from activities from other processor agents.
- Load the designated AC module into authenticated code execution area.
- Isolate the content of authenticated code execution area from further state modification by external agents.
- Authenticate the AC module.
- Updated the Trusted Platform Module (TPM) with the authenticated code module's hash.
- Initialize processor state based on the authenticated code module header information.
- Unlock the Intel® TXT-capable chipset private configuration register space and TPM locality 3 space.
- Begin execution in the authenticated code module at the defined entry point.

As an integrity check for proper processor hardware operation, execution of GETSEC[SENTER] will also check the contents of all the machine check status registers (as reported by the MSRs IA32\_MCi\_STATUS) for any valid uncorrectable error condition. In addition, the global machine check status register IA32\_MCG\_STATUS MCIP bit must be cleared and the IERR processor package pin (or its equivalent) must be not asserted, indicating that no machine check exception processing is currently in-progress. These checks are performed twice: once by the ILP prior to the broadcast of the rendezvous message to RLPs, and later in response to RLPs acknowledging the rendezvous message. Any outstanding valid uncorrectable machine check error condition present in the machine check status registers at the first check point will result in the ILP signaling a general protection violation. If an outstanding valid uncorrectable machine check error condition is present at the second check point, then this will result in the corresponding logical processor signaling the more severe TXT-shutdown condition with an error code of 12.

Before loading and authentication of the target code module is performed, the processor also checks that the current voltage and bus ratio encodings correspond to known good values supportable by the processor. The MSR IA32\_PERF\_STATUS values are compared against either the processor supported maximum operating target setting, system reset setting, or the thermal monitor operating target. If the current settings do not meet any of these criteria then the SENTER function will attempt to change the voltage and bus ratio select controls in a processor-specific manner. This adjustment may be to the thermal monitor, minimum (if different), or maximum operating target depending on the processor.

This implies that some thermal operating target parameters configured by BIOS may be overridden by SENTER. The measured environment software may need to take responsibility for restoring such settings that are deemed to be safe, but not necessarily recognized by SENTER. If an adjustment is not possible when an out of range setting is discovered, then the processor will abort the measured launch. This may be the case for chipset controlled settings of these values or if the controllability is not enabled on the processor. In this case it is the responsibility of the external software to program the chipset voltage ID and/or bus ratio select settings to known good values recognized by the processor, prior to executing SENTER.

## NOTE

For a mobile processor, an adjustment can be made according to the thermal monitor operating target. For a quad-core processor the SENTER adjustment mechanism may result in a more conservative but non-uniform voltage setting, depending on the pre-SENTER settings per core.

The ILP and RLPs mask the response to the assertion of the external signals INIT#, A20M, NMI#, and SMI#. The purpose of this masking control is to prevent exposure to existing external event handlers until a protected handler has been put in place to directly handle these events. Masked external pin events may be unmasked conditionally or unconditionally via the GETSEC[EXITAC], GETSEC[SEXIT], GETSEC[SMCTRL] or for specific VMX related operations such as a VM entry or the

VMXOFF instruction (see respective GETSEC leaves and *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B* for more details). The state of the A20M pin is masked and forced internally to a de-asserted state so that external assertion is not recognized. A20M masking as set by GETSEC[SENDER] is undone only after taking down the measured environment with the GETSEC[SEXIT] instruction or processor reset. INTR is masked by simply clearing the EFLAGS.IF bit. It is the responsibility of system software to control the processor response to INTR through appropriate management of EFLAGS.

To prevent other (logical) processors from interfering with the ILP operating in authenticated code execution mode, memory (excluding implicit write-back transactions) and I/O activities originating from other processor agents are blocked. This protection starts when the ILP enters into authenticated code execution mode. Only memory and I/O transactions initiated from the ILP are allowed to proceed. Exiting authenticated code execution mode is done by executing GETSEC[EXITAC]. The protection of memory and I/O activities remains in effect until the ILP executes GETSEC[EXITAC].

Once the authenticated code module has been loaded into the authenticated code execution area, it is protected against further modification from external bus snoops. There is also a requirement that the memory type for the authenticated code module address range be WB (via initialization of the MTRRs prior to execution of this instruction). If this condition is not satisfied, it is a violation of security and the processor will force a TXT system reset (after writing an error code to the chipset LT.ERROR-CODE register). This action is referred to as a Intel® TXT reset condition. It is performed when it is considered unreliable to signal an error through the conventional exception reporting mechanism.

To conform to the minimum granularity of MTRR MSRs for specifying the memory type, authenticated code RAM (ACRAM) is allocated to the processor in 4096 byte granular blocks. If an AC module size as specified in ECX is not a multiple of 4096 then the processor will allocate up to the next 4096 byte boundary for mapping as ACRAM with indeterminate data. This pad area will not be visible to the authenticated code module as external memory nor can it depend on the value of the data used to fill the pad area.

Once successful authentication has been completed by the ILP, the computed hash is stored in the TPM at PCR17 after this register is implicitly reset. PCR17 is a dedicated register for holding the computed hash of the authenticated code module loaded and subsequently executed by the GETSEC[SENDER]. As part of this process, the dynamic PCRs 18-22 are reset so they can be utilized by subsequently software for registration of code and data modules. After successful execution of SENTER, PCR17 contains the measurement of AC code and the SENTER launching parameters.

After authentication is completed successfully, the private configuration space of the Intel® TXT-capable chipset is unlocked so that the authenticated code module and measured environment software can gain access to this normally restricted chipset state. The Intel® TXT-capable chipset private configuration space can be locked later by software writing to the chipset LT.CMD.CLOSE-PRIVATE register or unconditionally using the GETSEC[SEXIT] instruction.

The SENTER leaf function also initializes some processor architecture state for the ILP from contents held in the header of the authenticated code module. Since the authenticated code module is relocatable, all address references are relative to the base address passed in via EBX. The ILP GDTR base value is initialized to  $EBX + [GDTBasePtr]$  and GDTR limit set to  $[GDTLimit]$ . The CS selector is initialized to the value held in the AC module header field SegSel, while the DS, SS, and ES selectors are initialized to CS+8. The segment descriptor fields are initialized implicitly with BASE=0, LIMIT=FFFFFh, G=1, D=1, P=1, S=1, read/write/accessed for DS, SS, and ES, while execute/read/accessed for CS. Execution in the authenticated code module for the ILP begins with the EIP set to  $EBX + [EntryPoint]$ . AC module defined fields used for initializing processor state are consistency checked with a failure resulting in an TXT-shutdown condition.

Table 6-6 provides a summary of processor state initialization for the ILP and RLP(s) after successful completion of GETSEC[SENTER]. For both ILP and RLP(s), paging is disabled upon entry to the measured environment. It is up to the ILP to establish a trusted paging environment, with appropriate mappings, to meet protection requirements established during the launch of the measured environment. RLP state initialization is not completed until a subsequent wake-up has been signaled by execution of the GETSEC[WAKEUP] function by the ILP.

**Table 6-6. Register State Initialization after GETSEC[SENTER] and GETSEC[WAKEUP]**

Register State	ILP after GETSEC[SENTER]	RLP after GETSEC[WAKEUP]
CRO	$PG \leftarrow 0, AM \leftarrow 0, WP \leftarrow 0$ ; Others unchanged	$PG \leftarrow 0, CD \leftarrow 0, NW \leftarrow 0, AM \leftarrow 0, WP \leftarrow 0$ ; $PE \leftarrow 1, NE \leftarrow 1$
CR4	00004000H	00004000H
EFLAGS	00000002H	00000002H
IA32_EFER	0H	0
EIP	[EntryPoint from MLE header <sup>1</sup> ]	[LT.MLE.JOIN + 12]
EBX	Unchanged [SINIT.BASE]	Unchanged
EDX	SENTER control flags	Unchanged
EBP	SINIT.BASE	Unchanged
CS	Sel=[SINIT SegSel], base=0, limit=FFFFFh, G=1, D=1, AR=9BH	Sel = [LT.MLE.JOIN + 8], base = 0, limit = FFFFFFFH, G = 1, D = 1, AR = 9BH
DS, ES, SS	Sel=[SINIT SegSel] +8, base=0, limit=FFFFFh, G=1, D=1, AR=93H	Sel = [LT.MLE.JOIN + 8] +8, base = 0, limit = FFFFFFFH, G = 1, D = 1, AR = 93H

**Table 6-6. Register State Initialization after GETSEC[SENTER] and GETSEC[WAKEUP]**

GDTR	Base= SINIT.base (EBX) + [SINIT.GDTBasePtr], Limit=[SINIT.GDTLimit]	Base = [LT.MLE.JOIN + 4], Limit = [LT.MLE.JOIN]
DR7	00000400H	00000400H
IA32_DEBUGCTL	0H	0H
Performance counters and counter control registers	0H	0H
IA32_MISC_ENABLE	See Table 6-5	See Table 6-5
IA32_SMM_MONITOR_CTL	Bit 2←0	Bit 2←0

**NOTES:**

1. See *Intel® Trusted Execution Technology Measured Launched Environment Programming Guide* for MLE header format.

Segmentation related processor state that has not been initialized by GETSEC[SENTER] requires appropriate initialization before use. Since a new GDT context has been established, the previous state of the segment selector values held in FS, GS, TR, and LDTR may no longer be valid. The IDTR will also require reloading with a new IDT context after launching the measured environment before exceptions or the external interrupts INTR and NMI can be handled. In the meantime, the programmer must take care in not executing an INT n instruction or any other condition that would result in an exception or trap signaling.

Debug exception and trap related signaling is also disabled as part of execution of GETSEC[SENTER]. This is achieved by clearing DR7, TF in EFLAGS, and the MSR IA32\_DEBUGCTL as defined in Table 6-6. These can be re-enabled once supporting exception handler(s), descriptor tables, and debug registers have been properly re-initialized following SENTER. Also, any pending single-step trap condition will be cleared at the completion of SENTER for both the ILP and RLP(s).

Performance related counters and counter control registers are cleared as part of execution of SENTER on both the ILP and RLP. This implies any active performance counters at the time of SENTER execution will be disabled. To reactive the processor performance counters, this state must be re-initialized and re-enabled.

Since MCE along with all other state bits (with the exception of SMXE) are cleared in CR4 upon execution of SENTER processing, any enabled machine check error condition that occurs will result in the processor performing the TXT-shutdown action. This also applies to an RLP while in the SENTER sleep state. For each logical processor



CR4.MCE must be reestablished with a valid machine check exception handler to otherwise avoid an TXT-shutdown under such conditions.

The MSR IA32\_EFER is also unconditionally cleared as part of the processor state initialized by SENTER for both the ILP and RLP. Since paging is disabled upon entering authenticated code execution mode, a new paging environment will have to be re-established if it is desired to enable IA-32e mode while operating in authenticated code execution mode.

The miscellaneous feature control MSR, IA32\_MISC\_ENABLE, is initialized as part of the measured environment launch. Certain bits of this MSR are preserved because preserving these bits may be important to maintain previously established platform settings. See the footnote for Table 6-5 The remaining bits are cleared for the purpose of establishing a more consistent environment for the execution of authenticated code modules. Among the impact of initializing this MSR, any previous condition established by the MONITOR instruction will be cleared.

### Effect of MSR IA32\_FEATURE\_CONTROL MSR

Bits 15:8 of the IA32\_FEATURE\_CONTROL MSR affect the execution of GETSEC[SENTER]. These bits consist of two fields:

- Bit 15: a global enable control for execution of SENTER.
- Bits 14:8: a parameter control field providing the ability to qualify SENTER execution based on the level of functionality specified with corresponding EDX parameter bits 6:0.

The layout of these fields in the IA32\_FEATURE\_CONTROL MSR is shown in Table 6-1.

Prior to the execution of GETSEC[SENTER], the lock bit of IA32\_FEATURE\_CONTROL MSR must be bit set to affirm the settings to be used. Once the lock bit is set, only a power-up reset condition will clear this MSR. The IA32\_FEATURE\_CONTROL MSR must be configured in accordance to the intended usage at platform initialization. Note that this MSR is only available on SMX or VMX enabled processors. Otherwise, IA32\_FEATURE\_CONTROL is treated as reserved.

The *Intel® Trusted Execution Technology Measured Launched Environment Programming Guide* provides additional details and requirements for programming measured environment software to launch in an Intel TXT platform.

### Operation in a Uni-Processor Platform

(\* The state of the internal flag ACMODEFLAG and SENTERFLAG persist across instruction boundary \*)

#### GETSEC[SENTER] (ILP only):

```
IF (CR4.SMXE=0)
    THEN #UD;
ELSE IF (in VMX non-root operation)
    THEN VM Exit (reason="GETSEC instruction");
ELSE IF (GETSEC leaf unsupported)
    THEN #UD;
```

```

ELSE IF ((in VMX root operation) or
  (CRO.PE=0) or (CRO.CD=1) or (CRO.NW=1) or (CRO.NE=0) or
  (CPL>0) or (EFLAGS.VM=1) or
  (IA32_APIC_BASE.BSP=0) or (TXT chipset not present) or
  (SENTERFLAG=1) or (ACMODEFLAG=1) or (IN_SMM=1) or
  (TPM interface is not present) or
  (EDX != (SENTER_EDX_support_mask & EDX)) or
  (IA32_CR_FEATURE_CONTROL[0]=0) or (IA32_CR_FEATURE_CONTROL[15]=0) or
  ((IA32_CR_FEATURE_CONTROL[14:8] & EDX[6:0]) != EDX[6:0]))
  THEN #GP(0);
IF (GETSEC[PARAMETERS].Parameter_Type = 5, MCA_Handling (bit 6) = 0)
  FOR I = 0 to IA32_MCG_CAP.COUNT-1 DO
    IF IA32_MC[I]_STATUS = uncorrectable error
      THEN #GP(0);
    FI;
  OD;
FI;
IF (IA32_MCG_STATUS.MCIP=1) or (IERR pin is asserted)
  THEN #GP(0);
ACBASE← EBX;
ACSIZE← ECX;
IF (((ACBASE MOD 4096) != 0) or ((ACSIZE MOD 64) != 0) or (ACSIZE < minimum
  module size) or (ACSIZE > AC RAM capacity) or ((ACBASE+ACSIZE) > (2^32 -1)))
  THEN #GP(0);
Mask SMI, INIT, A20M, and NMI external pin events;
SignalTXTMsg(SENTER);
DO
  WHILE (no SignalSEnTER message);

TXT_SENTER__MSG_EVENT (ILP & RLP):
Mask and clear SignalSEnTER event;
Unmask SignalSEXIT event;
IF (in VMX operation)
  THEN TXT-SHUTDOWN(#IllegalEvent);
FOR I = 0 to IA32_MCG_CAP.COUNT-1 DO
  IF IA32_MC[I]_STATUS = uncorrectable error
    THEN TXT-SHUTDOWN(#UnrecovMCError);
  FI;
OD;
IF (IA32_MCG_STATUS.MCIP=1) or (IERR pin is asserted)
  THEN TXT-SHUTDOWN(#UnrecovMCError);
IF (Voltage or bus ratio status are NOT at a known good state)
  THEN IF (Voltage select and bus ratio are internally adjustable)

```

```

THEN
    Make product-specific adjustment on operating parameters;
ELSE
    TXT-SHUTDOWN(#IllegalVIDBRatio);
FI;

IA32_MISC_ENABLE← (IA32_MISC_ENABLE & MASK_CONST*)
(* The hexadecimal value of MASK_CONST may vary due to processor implementations *)
A20M← 0;
IA32_DEBUGCTL← 0;
Invalidate processor TLB(s);
Drain outgoing transactions;
Clear performance monitor counters and control;
SENDERFLAG← 1;
SignalTXTMsg(SENDERAck);
IF (logical processor is not ILP)
    THEN GOTO RLP_SENTER_ROUTINE;
(* ILP waits for all logical processors to ACK *)
DO
    DONE← TXT.READ(LT.STS);
WHILE (not DONE);
SignalTXTMsg(SENTERContinue);
SignalTXTMsg(ProcessorHold);
FOR I=ACBASE to ACBASE+ACSIZE-1 DO
    ACRAM[I-ACBASE].ADDR← I;
    ACRAM[I-ACBASE].DATA← LOAD(I);
OD;
IF (ACRAM memory type != WB)
    THEN TXT-SHUTDOWN(#BadACMMType);
IF (AC module header version is not supported) OR (ACRAM[ModuleType] <> 2)
    THEN TXT-SHUTDOWN(#UnsupportedACM);
KEY← GETKEY(ACRAM, ACBASE);
KEYHASH← HASH(KEY);
CSKEYHASH← LT.READ(LT.PUBLIC.KEY);
IF (KEYHASH <> CSKEYHASH)
    THEN TXT-SHUTDOWN(#AuthenticateFail);
SIGNATURE← DECRYPT(ACRAM, ACBASE, KEY);
(* The value of SIGNATURE_LEN_CONST is implementation-specific*)
FOR I=0 to SIGNATURE_LEN_CONST - 1 DO
    ACRAM[SCRATCH.I]← SIGNATURE[I];
COMPUTEDSIGNATURE← HASH(ACRAM, ACBASE, ACSIZE);
FOR I=0 to SIGNATURE_LEN_CONST - 1 DO
    ACRAM[SCRATCH.SIGNATURE_LEN_CONST+I]← COMPUTEDSIGNATURE[I];

```

## SAFER MODE EXTENSIONS REFERENCE

```
IF (SIGNATURE != COMPUTEDSIGNATURE)
    THEN TXT-SHUTDOWN(#AuthenticateFail);
ACMCONTROL← ACMRAM[CodeControl];
IF ((ACMCONTROL.0 = 0) and (ACMCONTROL.1 = 1) and (snoop hit to modified line detected on
ACRAM load))
    THEN TXT-SHUTDOWN(#UnexpectedHITM);
IF (ACMCONTROL reserved bits are set)
    THEN TXT-SHUTDOWN(#BadACMFormat);
IF ((ACRAM[GDTBasePtr] < (ACRAM[HeaderLen] * 4 + Scratch_size)) OR
((ACRAM[GDTBasePtr] + ACMRAM[GDTLimit]) >= ACSIZE))
    THEN TXT-SHUTDOWN(#BadACMFormat);
IF ((ACMCONTROL.0 = 1) and (ACMCONTROL.1 = 1) and (snoop hit to modified
line detected on ACRAM load))
    THEN ACEntryPoint← ACBASE+ACRAM[ErrorEntryPoint];
ELSE
    ACEntryPoint← ACBASE+ACRAM[EntryPoint];
IF ((ACEntryPoint >= ACSIZE) or (ACEntryPoint < (ACRAM[HeaderLen] * 4 + Scratch_size)))
    THEN TXT-SHUTDOWN(#BadACMFormat);
IF ((ACRAM[SegSel] > (ACRAM[GDTLimit] - 15)) or (ACRAM[SegSel] < 8))
    THEN TXT-SHUTDOWN(#BadACMFormat);
IF ((ACRAM[SegSel].TI=1) or (ACRAM[SegSel].RPL=0))
    THEN TXT-SHUTDOWN(#BadACMFormat);
ACRAM[SCRATCH.SIGNATURE_LEN_CONST]← EDX;
WRITE(TPM.HASH.START)← 0;
FOR I=0 to SIGNATURE_LEN_CONST + 3 DO
    WRITE(TPM.HASH.DATA)← ACMRAM[SCRATCH.I];
WRITE(TPM.HASH.END)← 0;
ACMODEFLAG← 1;
CRO.[PG.AM.WP]← 0;
CR4← 00004000h;
EFLAGS← 00000002h;
IA32_EFER← 0;
EBP← ACBASE;
GDTR.BASE← ACBASE+ACRAM[GDTBasePtr];
GDTR.LIMIT← ACMRAM[GDTLimit];
CS.SEL← ACMRAM[SegSel];
CS.BASE← 0;
CS.LIMIT← FFFFFFFh;
CS.G← 1;
CS.D← 1;
CS.AR← 9Bh;
DS.SEL← ACMRAM[SegSel]+8;
DS.BASE← 0;
DS.LIMIT← FFFFFFFh;
```

```

DS.G← 1;
DS.D← 1;
DS.AR← 93h;
SS← DS;
ES← DS;
DR7← 00000400h;
IA32_DEBUGCTL← 0;
SignalTXTMsg(UnlockSMRAM);
SignalTXTMsg(OpenPrivate);
SignalTXTMsg(OpenLocality3);
EIP← ACEntryPoint;
END;

```

### **RLP\_SENTER\_ROUTINE: (RPL only)**

```

Mask SMI, INIT, A20M, and NMI external pin events
Unmask SignalWAKEUP event;
Wait for SignalSENTERContinue message;
IA32_APIC_BASE.BSP← 0;
GOTO SENTER sleep state;
END;

```

### **Flags Affected**

All flags are cleared.

### **Use of Prefixes**

LOCK	Causes #UD
REP*	Cause #UD (includes REPNE/REPNZ and REP/REPE/REPZ)
Operand size	Causes #UD
Segment overrides	Ignored
Address size	Ignored
REX	Ignored

### **Protected Mode Exceptions**

#UD	<p>If CR4.SMXE = 0.</p> <p>If GETSEC[SENTER] is not reported as supported by GETSEC[CAPABILITIES].</p>
#GP(0)	<p>If CR0.CD = 1 or CR0.NW = 1 or CR0.NE = 0 or CR0.PE = 0 or CPL &gt; 0 or EFLAGS.VM = 1.</p> <p>If in VMX root operation.</p> <p>If the initiating processor is not designated as the bootstrap processor via the MSR bit IA32_APIC_BASE.BSP.</p>

If an Intel® TXT-capable chipset is not present.

If an Intel® TXT-capable chipset interface to TPM is not detected as present.

If a protected partition is already active or the processor is already in authenticated code mode.

If the processor is in SMM.

If a valid uncorrectable machine check error is logged in IA32\_MC[1]\_STATUS.

If the authenticated code base is not on a 4096 byte boundary.

If the authenticated code size > processor's authenticated code execution area storage capacity.

If the authenticated code size is not modulo 64.

### Real-Address Mode Exceptions

- #UD If CR4.SMXE = 0.  
If GETSEC[SENTER] is not reported as supported by GETSEC[CAPABILITIES].
- #GP(0) GETSEC[SENTER] is not recognized in real-address mode.

### Virtual-8086 Mode Exceptions

- #UD If CR4.SMXE = 0.  
If GETSEC[SENTER] is not reported as supported by GETSEC[CAPABILITIES].
- #GP(0) GETSEC[SENTER] is not recognized in virtual-8086 mode.

### Compatibility Mode Exceptions

All protected mode exceptions apply.

- #GP IF AC code module does not reside in physical address below  $2^{32} - 1$ .

### 64-Bit Mode Exceptions

All protected mode exceptions apply.

- #GP IF AC code module does not reside in physical address below  $2^{32} - 1$ .

### VM-Exit Condition

- Reason (GETSEC) IF in VMX non-root operation.

## GETSEC[SEXIT]—Exit Measured Environment

Opcode	Instruction	Description
0F 37 (EAX=5)	GETSEC[SEXIT]	Exit measured environment

### Description

The GETSEC[SEXIT] instruction initiates an exit of a measured environment established by GETSEC[SENDER]. The SEXIT leaf of GETSEC is selected with EAX set to 5 at execution. This instruction leaf sends a message to all logical processors in the platform to signal the measured environment exit.

There are restrictions enforced by the processor for the execution of the GETSEC[SEXIT] instruction:

- Execution is not allowed unless the processor is in protected mode (CR0.PE = 1) with CPL = 0 and EFLAGS.VM = 0.
- The processor must be in a measured environment as launched by a previous GETSEC[SENDER] instruction, but not still in authenticated code execution mode.
- To avoid potential inter-operability conflicts between modes, the processor is not allowed to execute this instruction if it currently is in SMM or in VMX operation.
- To insure consistent handling of SIPI messages, the processor executing the GETSEC[SEXIT] instruction must also be designated the BSP (bootstrap processor) as defined by the register bit IA32\_APIC\_BASE.BSP (bit 8).

Failure to abide by the above conditions results in the processor signaling a general protection violation.

This instruction initiates a sequence to rendezvous the RLPs with the ILP. It then clears the internal processor flag indicating the processor is operating in a measured environment.

In response to a message signaling the completion of rendezvous, all RLPs restart execution with the instruction that was to be executed at the time GETSEC[SEXIT] was recognized. This applies to all processor conditions, with the following exceptions:

- If an RLP executed HLT and was in this halt state at the time of the message initiated by GETSEC[SEXIT], then execution resumes in the halt state.
- If an RLP was executing MWAIT, then a message initiated by GETSEC[SEXIT] causes an exit of the MWAIT state, falling through to the next instruction.
- If an RLP was executing an intermediate iteration of a string instruction, then the processor resumes execution of the string instruction at the point which the message initiated by GETSEC[SEXIT] was recognized.
- If an RLP is still in the SENTER sleep state (never awakened with GETSEC[WAKEUP]), it will be sent to the wait-for-SIPI state after first clearing

the bootstrap processor indicator flag (IA32\_APIC\_BASE.BSP) and any pending SIPI state. In this case, such RLPs are initialized to an architectural state consistent with having taken a soft reset using the INIT# pin.

Prior to completion of the GETSEC[SEXIT] operation, both the ILP and any active RLPs unmask the response of the external event signals INIT#, A20M, NMI#, and SMI#. This unmasking is performed unconditionally to recognize pin events which are masked after a GETSEC[SENDER]. The state of A20M is unmasked, as the A20M pin is not recognized while the measured environment is active.

On a successful exit of the measured environment, the ILP re-locks the Intel® TXT-capable chipset private configuration space. GETSEC[SEXIT] does not affect the content of any PCR.

At completion of GETSEC[SEXIT] by the ILP, execution proceeds to the next instruction. Since EFLAGS and the debug register state are not modified by this instruction, a pending trap condition is free to be signaled if previously enabled.

### Operation in a Uni-Processor Platform

(\* The state of the internal flag ACMODEFLAG and SENTERFLAG persist across instruction boundary \*)

#### GETSEC[SEXIT] (ILP only):

```

IF (CR4.SMXE=0)
    THEN #UD;
ELSE IF (in VMX non-root operation)
    THEN VM Exit (reason="GETSEC instruction");
ELSE IF (GETSEC leaf unsupported)
    THEN #UD;
ELSE IF ((in VMX root operation) or
    (CR0.PE=0) or (CPL>0) or (EFLAGS.VM=1) or
    (IA32_APIC_BASE.BSP=0) or
    (TXT chipset not present) or
    (SENDERFLAG=0) or (ACMODEFLAG=1) or (IN_SMM=1))
    THEN #GP(0);
SignalTXTMsg(SEXIT);
DO
WHILE (no SignalSEXIT message);
    
```

#### TXT\_SEXIT\_MSG\_EVENT (ILP & RLP):

```

Mask and clear SignalSEXIT event;
Clear MONITOR FSM;
Unmask SignalSENDER event;
IF (in VMX operation)
    THEN TXT-SHUTDOWN(#IllegalEvent);
SignalTXTMsg(SEXITAck);
IF (logical processor is not ILP)
    
```



```

    THEN GOTO RLP_SEXIT_ROUTINE;
(* ILP waits for all logical processors to ACK *)
DO
    DONE← READ(LT.STS);
WHILE (NOT DONE);
SignalTXTMsg(SEXITContinue);
SignalTXTMsg(ClosePrivate);
SENDERFLAG← 0;
Unmask SMI, INIT, A20M, and NMI external pin events;
END;

```

**RLP\_SEXIT\_ROUTINE (RLPs only):**

```

Wait for SignalSEXITContinue message;
Unmask SMI, INIT, A20M, and NMI external pin events;
IF (prior execution state = HLT)
    THEN reenter HLT state;
IF (prior execution state = SENTER sleep)
    THEN
        IA32_APIC_BASE.BSP← 0;
        Clear pending SIPI state;
        Call INIT_PROCESSOR_STATE;
        Unmask SIPI event;
        GOTO WAIT-FOR-SIPI;
FI;
END;

```

**Flags Affected**

ILP: None.

RLPs: all flags are modified for an RLP. returning to wait-for-SIPI state, none otherwise

**Use of Prefixes**

LOCK	Causes #UD
REP*	Cause #UD (includes REPNE/REPNZ and REP/REPE/REPZ)
Operand size	Causes #UD
Segment overrides	Ignored
Address size	Ignored
REX	Ignored

**Protected Mode Exceptions**

#UD                      If CR4.SMXE = 0.

If GETSEC[SEXIT] is not reported as supported by GETSEC[CAPABILITIES].

#GP(0) If CR0.PE = 0 or CPL > 0 or EFLAGS.VM = 1.  
If in VMX root operation.  
If the initiating processor is not designated as the `via` the MSR bit IA32\_APIC\_BASE.BSP.  
If an Intel® TXT-capable chipset is not present.  
If a protected partition is not already active or the processor is already in authenticated code mode.  
If the processor is in SMM.

### Real-Address Mode Exceptions

#UD If CR4.SMXE = 0.  
If GETSEC[SEXIT] is not reported as supported by GETSEC[CAPABILITIES].

#GP(0) GETSEC[SEXIT] is not recognized in real-address mode.

### Virtual-8086 Mode Exceptions

#UD If CR4.SMXE = 0.  
If GETSEC[SEXIT] is not reported as supported by GETSEC[CAPABILITIES].

#GP(0) GETSEC[SEXIT] is not recognized in virtual-8086 mode.

### Compatibility Mode Exceptions

All protected mode exceptions apply.

### 64-Bit Mode Exceptions

All protected mode exceptions apply.

### VM-Exit Condition

Reason (GETSEC) IF in VMX non-root operation.

## GETSEC[PARAMETERS]—Report the SMX Parameters

Opcode	Instruction	Description
0F 37 (EAX=6)	GETSEC[PARAMETERS]	<i>Report the SMX Parameters</i>  <i>The parameters index is input in EBX with the result returned in EAX, EBX, and ECX.</i>

### Description

The GETSEC[PARAMETERS] instruction returns specific parameter information for SMX features supported by the processor. Parameter information is returned in EAX, EBX, and ECX, with the input parameter selected using EBX.

Software retrieves parameter information by searching with an input index for EBX starting at 0, and then reading the returned results in EAX, EBX, and ECX. EAX[4:0] is designated to return a parameter type field indicating if a parameter is available and what type it is. If EAX[4:0] is returned with 0, this designates a null parameter and indicates no more parameters are available.

Table 6-7 defines the parameter types supported in current and future implementations.

**Table 6-7. SMX Reporting Parameters Format**

Parameter Type EAX[4:0]	Parameter Description	EAX[31:5]	EBX[31:0]	ECX[31:0]
0	NULL	Reserved (0 returned)	Reserved (unmodified)	Reserved (unmodified)
1	Supported AC module versions	Reserved (0 returned)	version comparison mask	version numbers supported
2	Max size of authenticated code execution area	Multiply by 32 for size in bytes	Reserved (unmodified)	Reserved (unmodified)
3	External memory types supported during AC mode	Memory type bit mask	Reserved (unmodified)	Reserved (unmodified)

**Table 6-7. SMX Reporting Parameters Format (Contd.)**

Parameter Type EAX[4:0]	Parameter Description	EAX[31:5]	EBX[31:0]	ECX[31:0]
4	Selective SENTER functionality control	EAX[14:8] correspond to available SENTER function disable controls	Reserved (unmodified)	Reserved (unmodified)
5	TXT extensions support	TXT Feature Extensions Flags (see Table 6-8)	Reserved	Reserved
6-31	Undefined	Reserved (unmodified)	Reserved (unmodified)	Reserved (unmodified)

**Table 6-8. TXT Feature Extensions Flags**

Bit	Definition	Description
5	Processor based S-CRTM support	Returns 1 if this processor implements a processor-rooted S-CRTM capability and 0 if not (S-CRTM is rooted in BIOS). This flag cannot be used to infer whether the chipset supports TXT or whether the processor support SMX.
6	Machine Check Handling	Returns 1 if it machine check status registers can be preserved through ENTERACCS and SENTER. If this bit is 1, the caller of ENTERACCS and SENTER is not required to clear machine check error status bits before invoking these GETSEC leaves. If this bit returns 0, the caller of ENTERACCS and SENTER must clear all machine check error status bits before invoking these GETSEC leaves.
31:7	Reserved	Reserved for future use. Will return 0.

Supported AC module versions (as defined by the AC module HeaderVersion field) can be determined for a particular SMX capable processor by the type 1 parameter. Using EBX to index through the available parameters reported by GETSEC[PARAMETERS] for each unique parameter set returned for type 1, software can determine the complete list of AC module version(s) supported.

For each parameter set, EBX returns the comparison mask and ECX returns the available HeaderVersion field values supported, after AND'ing the target HeaderVersion with the comparison mask. Software can then determine if a particular AC module version is supported by following the pseudo-code search routine given below:

```
parameter_search_index= 0
do {
    EBX= parameter_search_index++
    EAX= 6
    GETSEC
    if (EAX[4:0] = 1) {
        if ((version_query & EBX) = ECX) {
            version_is_supported= 1
            break
        }
    }
} while (EAX[4:0]!= 0)
```

If only AC modules with a HeaderVersion of 0 are supported by the processor, then only one parameter set of type 1 will be returned, as follows: EAX = 00000001H, EBX = FFFFFFFFH and ECX = 00000000H.

The maximum capacity for an authenticated code execution area supported by the processor is reported with the parameter type of 2. The maximum supported size in bytes is determined by multiplying the returned size in EAX[31:5] by 32. Thus, for a maximum supported authenticated RAM size of 32KBytes, EAX returns with 00008002H.

Supportable memory types for memory mapped outside of the authenticated code execution area are reported with the parameter type of 3. While is active, as initiated by the GETSEC functions SENTER and ENTERACCS and terminated by EXITAC, there are restrictions on what memory types are allowed for the rest of system memory. It is the responsibility of the system software to initialize the memory type range register (MTRR) MSRs and/or the page attribute table (PAT) to only map memory types consistent with the reporting of this parameter. The reporting of supportable memory types of external memory is indicated using a bit map returned in EAX[31:8]. These bit positions correspond to the memory type encodings defined for the MTRR MSR and PAT programming. See Table 6-9.

The parameter type of 4 is used for enumerating the availability of selective GETSEC[SENDER] function disable controls. If a 1 is reported in bits 14:8 of the returned parameter EAX, then this indicates a disable control capability exists with SENTER for a particular function. The enumerated field in bits 14:8 corresponds to use of the EDX input parameter bits 6:0 for SENTER. If an enumerated field bit is set to 1, then the corresponding EDX input parameter bit of EDX may be set to 1 to disable that designated function. If the enumerated field bit is 0 or this parameter is not reported, then no disable capability exists with the corresponding EDX input parameter for SENTER, and EDX bit(s) must be cleared to 0 to enable execution of

SENDER. If no selective disable capability for SENDER exists as enumerated, then the corresponding bits in the IA32\_FEATURE\_CONTROL MSR bits 14:8 must also be programmed to 1 if the SENDER global enable bit 15 of the MSR is set. This is required to enable future extensibility of SENDER selective disable capability with respect to potentially separate software initialization of the MSR.

**Table 6-9. External Memory Types Using Parameter 3**

EAX Bit Position	Parameter Description
8	Uncacheable (UC)
9	Write Combining (WC)
11:10	Reserved
12	Write-through (WT)
13	Write-protected (WP)
14	Write-back (WB)
31:15	Reserved

If the GETSEC[PARAMETERS] leaf or specific parameter is not present for a given SMX capable processor, then default parameter values should be assumed. These are defined in Table 6-10.

**Table 6-10. Default Parameter Values**

Parameter Type EAX[4:0]	Default Setting	Parameter Description
1	0.0 only	Supported AC module versions
2	32 KBytes	Authenticated code execution area size
3	UC only	External memory types supported during AC execution mode
4	None	Available SENDER selective disable controls

**Operation**

(\* example of a processor supporting only a 0.0 HeaderVersion, 32K ACRAM size, memory types UC and WC \*)

IF (CR4.SMXE=0)

THEN #UD;

ELSE IF (in VMX non-root operation)

```

    THEN VM Exit (reason="GETSEC instruction");
ELSE IF (GETSEC leaf unsupported)
    THEN #UD;
    (* example of a processor supporting a 0.0 HeaderVersion *)
IF (EBX=0) THEN
    EAX← 00000001h;
    EBX← FFFFFFFFh;
    ECX← 00000000h;
ELSE IF (EBX=1)
    (* example of a processor supporting a 32K ACRAM size *)
    THEN EAX← 00008002h;
ELSE IF (EBX= 2)
    (* example of a processor supporting external memory types of UC and WC *)
    THEN EAX← 00000303h;
ELSE IF (EBX= other value(s) less than unsupported index value)
    (* EAX value varies. Consult Table 6-7 and Table 6-8*)
ELSE (* unsupported index*)
    EAX" 00000000h;
END;
```

### Flags Affected

None.

### Use of Prefixes

LOCK	Causes #UD
REP*	Cause #UD (includes REPNE/REPNZ and REP/REPE/REPZ)
Operand size	Causes #UD
Segment overrides	Ignored
Address size	Ignored
REX	Ignored

### Protected Mode Exceptions

#UD	If CR4.SMXE = 0. If GETSEC[PARAMETERS] is not reported as supported by GETSEC[CAPABILITIES].
-----	---

### Real-Address Mode Exceptions

#UD	If CR4.SMXE = 0. If GETSEC[PARAMETERS] is not reported as supported by GETSEC[CAPABILITIES].
-----	---

### Virtual-8086 Mode Exceptions

#UD                      If CR4.SMXE = 0.  
                            If GETSEC[PARAMETERS] is not reported as supported by  
                            GETSEC[CAPABILITIES].

### Compatibility Mode Exceptions

All protected mode exceptions apply.

### 64-Bit Mode Exceptions

All protected mode exceptions apply.

### VM-Exit Condition

Reason (GETSEC)    IF in VMX non-root operation.



## GETSEC[SMCTRL]—SMX Mode Control

Opcode	Instruction	Description
OF 37 (EAX = 7)	GETSEC[SMCTRL]	<i>Perform specified SMX mode control as selected with the input EBX.</i>

### Description

The GETSEC[SMCTRL] instruction is available for performing certain SMX specific mode control operations. The operation to be performed is selected through the input register EBX. Currently only an input value in EBX of 0 is supported. All other EBX settings will result in the signaling of a general protection violation.

If EBX is set to 0, then the SMCTRL leaf is used to re-enable SMI events. SMI is masked by the ILP executing the GETSEC[SENDER] instruction (SMI is also masked in the responding logical processors in response to SENTER rendezvous messages.). The determination of when this instruction is allowed and the events that are unmasked is dependent on the processor context (See Table 6-11). For brevity, the usage of SMCTRL where EBX=0 will be referred to as GETSEC[SMCTRL(0)].

As part of support for launching a measured environment, the SMI, NMI and INIT events are masked after GETSEC[SENDER], and remain masked after exiting authenticated execution mode. Unmasking these events should be accompanied by securely enabling these event handlers. These security concerns can be addressed in VMX operation by a MVMM.

The VM monitor can choose two approaches:

- In a dual monitor approach, the executive software will set up an SMM monitor in parallel to the executive VMM (i.e. the MVMM), see Chapter 29, “System Management Mode” of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3C*. The SMM monitor is dedicated to handling SMI events without compromising the security of the MVMM. This usage model of handling SMI while a measured environment is active does not require the use of GETSEC[SMCTRL(0)] as event re-enabling after the VMX environment launch is handled implicitly and through separate VMX based controls.
- If a dedicated SMM monitor will not be established and SMIs are to be handled within the measured environment, then GETSEC[SMCTRL(0)] can be used by the executive software to re-enable SMI that has been masked as a result of SENTER.

Table 6-11 defines the processor context in which GETSEC[SMCTRL(0)] can be used and which events will be unmasked. Note that the events that are unmasked are dependent upon the currently operating processor context.

**Table 6-11. Supported Actions for GETSEC[SMCTRL(0)]**

ILP Mode of Operation	SMCTRL execution action
In VMX non-root operation	VM exit
SENTERFLAG = 0	#GP(0), illegal context
In authenticated code execution mode (ACMODEFLAG = 1)	#GP(0), illegal context
SENTERFLAG = 1, not in VMX operation, not in SMM	Unmask SMI
SENTERFLAG = 1, in VMX root operation, not in SMM	Unmask SMI if SMM monitor is not configured, otherwise #GP(0)
SENTERFLAG = 1, In VMX root operation, in SMM	#GP(0), illegal context

### Operation

(\* The state of the internal flag ACMODEFLAG and SENTERFLAG persist across instruction boundary \*)

```

IF (CR4.SMXE=0)
  THEN #UD;
ELSE IF (in VMX non-root operation)
  THEN VM Exit (reason="GETSEC instruction");
ELSE IF (GETSEC leaf unsupported)
  THEN #UD;
ELSE IF ((CR0.PE=0) or (CPL>0) OR (EFLAGS.VM=1))
  THEN #GP(0);
ELSE IF ((EBX=0) and (SENTERFLAG=1) and (ACMODEFLAG=0) and (IN_SMM=0) and
  (((in VMX root operation) and (SMM monitor not configured)) or (not in VMX operation)))
  THEN unmask SMI;
ELSE
  #GP(0);
END

```

### Flags Affected

None.

### Use of Prefixes

LOCK Causes #UD

REP*	Cause #UD (includes REPNE/REPZ and REP/REPE/REPZ)
Operand size	Causes #UD
Segment overrides	Ignored
Address size	Ignored
REX	Ignored

### Protected Mode Exceptions

#UD	<p>If CR4.SMXE = 0.</p> <p>If GETSEC[SMCTRL] is not reported as supported by GETSEC[CAPABILITIES].</p>
#GP(0)	<p>If CR0.PE = 0 or CPL &gt; 0 or EFLAGS.VM = 1.</p> <p>If in VMX root operation.</p> <p>If a protected partition is not already active or the processor is currently in authenticated code mode.</p> <p>If the processor is in SMM.</p> <p>If the SMM monitor is not configured</p>

### Real-Address Mode Exceptions

#UD	<p>If CR4.SMXE = 0.</p> <p>If GETSEC[SMCTRL] is not reported as supported by GETSEC[CAPABILITIES].</p>
#GP(0)	GETSEC[SMCTRL] is not recognized in real-address mode.

### Virtual-8086 Mode Exceptions

#UD	<p>If CR4.SMXE = 0.</p> <p>If GETSEC[SMCTRL] is not reported as supported by GETSEC[CAPABILITIES].</p>
#GP(0)	GETSEC[SMCTRL] is not recognized in virtual-8086 mode.

### Compatibility Mode Exceptions

All protected mode exceptions apply.

### 64-Bit Mode Exceptions

All protected mode exceptions apply.

### VM-exit Condition

Reason (GETSEC) IF in VMX non-root operation.

## GETSEC[WAKEUP]—Wake up sleeping processors in measured environment

Opcode	Instruction	Description
0F 37 (EAX=8)	GETSEC[WAKEUP]	Wake up the responding logical processors from the SENTER sleep state.

### Description

The GETSEC[WAKEUP] leaf function broadcasts a wake-up message to all logical processors currently in the SENTER sleep state. This GETSEC leaf must be executed only by the ILP, in order to wake-up the RLPs. Responding logical processors (RLPs) enter the SENTER sleep state after completion of the SENTER rendezvous sequence.

The GETSEC[WAKEUP] instruction may only be executed:

- In a measured environment as initiated by execution of GETSEC[SENTER].
- Outside of authenticated code execution mode.
- Execution is not allowed unless the processor is in protected mode with CPL = 0 and EFLAGS.VM = 0.
- In addition, the logical processor must be designated as the boot-strap processor as configured by setting IA32\_APIC\_BASE.BSP = 1.

If these conditions are not met, attempts to execute GETSEC[WAKEUP] result in a general protection violation.

An RLP exits the SENTER sleep state and start execution in response to a WAKEUP signal initiated by ILP’s execution of GETSEC[WAKEUP]. The RLP retrieves a pointer to a data structure that contains information to enable execution from a defined entry point. This data structure is located using a physical address held in the Intel® TXT-capable chipset configuration register LT.MLE.JOIN. The register is publicly writable in the chipset by all processors and is not restricted by the Intel® TXT-capable chipset configuration register lock status. The format of this data structure is defined in Table 6-12.

**Table 6-12. RLP MVMM JOIN Data Structure**

Offset	Field
0	GDT limit
4	GDT base pointer
8	Segment selector initializer
12	EIP

The MLE JOIN data structure contains the information necessary to initialize RLP processor state and permit the processor to join the measured environment. The GDTR, LIP, and CS, DS, SS, and ES selector values are initialized using this data structure. The CS selector index is derived directly from the segment selector initializer field; DS, SS, and ES selectors are initialized to CS+8. The segment descriptor fields are initialized implicitly with BASE = 0, LIMIT = FFFFFH, G = 1, D = 1, P = 1, S = 1; read/write/access for DS, SS, and ES; and execute/read/access for CS. It is the responsibility of external software to establish a GDT pointed to by the MLE JOIN data structure that contains descriptor entries consistent with the implicit settings initialized by the processor (see Table 6-6). Certain states from the content of Table 6-12 are checked for consistency by the processor prior to execution. A failure of any consistency check results in the RLP aborting entry into the protected environment and signaling an Intel® TXT shutdown condition. The specific checks performed are documented later in this section. After successful completion of processor consistency checks and subsequent initialization, RLP execution in the measured environment begins from the entry point at offset 12 (as indicated in Table 6-12).

## Operation

(\* The state of the internal flag ACMODEFLAG and SENTERFLAG persist across instruction boundary \*)

```
IF (CR4.SMXE=0)
    THEN #UD;
ELSE IF (in VMX non-root operation)
    THEN VM Exit (reason="GETSEC instruction");
ELSE IF (GETSEC leaf unsupported)
    THEN #UD;
ELSE IF ((CR0.PE=0) or (CPL>0) or (EFLAGS.VM=1) or (SENTERFLAG=0) or (ACMODEFLAG=1) or
(IN_SMM=0) or (in VMX operation) or (IA32_APIC_BASE.BSP=0) or (TXT chipset not present))
    THEN #GP(0);
ELSE
    SignalTXTMsg(WAKEUP);
END;
```

### RLP\_SIPWAKEUP\_FROM\_SENTER\_ROUTINE: (RLP only)

```
WHILE (no SignalWAKEUP event);
IF (IA32_SMM_MONITOR_CTL[0] != ILP.IA32_SMM_MONITOR_CTL[0])
    THEN TXT-SHUTDOWN(#IllegalEvent)
IF (IA32_SMM_MONITOR_CTL[0] = 0)
    THEN Unmask SMI pin event;
ELSE
    Mask SMI pin event;
Mask A20M, and NMI external pin events (unmask INIT);
Mask SignalWAKEUP event;
```

## SAFER MODE EXTENSIONS REFERENCE

```
Invalidate processor TLB(s);
Drain outgoing transactions;
TempGDTRLIMIT← LOAD(LT.MLE.JOIN);
TempGDTRBASE← LOAD(LT.MLE.JOIN+4);
TempSegSel← LOAD(LT.MLE.JOIN+8);
TempEIP← LOAD(LT.MLE.JOIN+12);
IF (TempGDTLimit & FFFF0000h)
    THEN TXT-SHUTDOWN(#BadJOINFormat);
IF ((TempSegSel > TempGDTRLIMIT-15) or (TempSegSel < 8))
    THEN TXT-SHUTDOWN(#BadJOINFormat);
IF ((TempSegSel.TI=1) or (TempSegSel.RPL=0))
    THEN TXT-SHUTDOWN(#BadJOINFormat);
CRO.[PG,CD,NW,AM,WP]← 0;
CRO.[NE,PE]← 1;
CR4← 00004000h;
EFLAGS← 00000002h;
IA32_EFER← 0;
GDTR.BASE← TempGDTRBASE;
GDTR.LIMIT← TempGDTRLIMIT;
CS.SEL← TempSegSel;
CS.BASE← 0;
CS.LIMIT← FFFFFh;
CS.G← 1;
CS.D← 1;
CS.AR← 9Bh;
DS.SEL← TempSegSel+8;
DS.BASE← 0;
DS.LIMIT← FFFFFh;
DS.G← 1;
DS.D← 1;
DS.AR← 93h;
SS← DS;
ES← DS;
DR7← 00000400h;
IA32_DEBUGCTL← 0;
EIP← TempEIP;
END;
```

### Flags Affected

None.

### Use of Prefixes

LOCK Causes #UD

REP*	Cause #UD (includes REPNE/REPNZ and REP/REPE/REPZ)
Operand size	Causes #UD
Segment overrides	Ignored
Address size	Ignored
REX	Ignored

### Protected Mode Exceptions

#UD	If CR4.SMXE = 0. If GETSEC[WAKEUP] is not reported as supported by GETSEC[CAPABILITIES].
#GP(0)	If CR0.PE = 0 or CPL > 0 or EFLAGS.VM = 1. If in VMX operation. If a protected partition is not already active or the processor is currently in authenticated code mode. If the processor is in SMM.
#UD	If CR4.SMXE = 0. If GETSEC[WAKEUP] is not reported as supported by GETSEC[CAPABILITIES].
#GP(0)	GETSEC[WAKEUP] is not recognized in real-address mode.

### Virtual-8086 Mode Exceptions

#UD	If CR4.SMXE = 0. If GETSEC[WAKEUP] is not reported as supported by GETSEC[CAPABILITIES].
#GP(0)	GETSEC[WAKEUP] is not recognized in virtual-8086 mode.

### Compatibility Mode Exceptions

All protected mode exceptions apply.

### 64-Bit Mode Exceptions

All protected mode exceptions apply.

### VM-exit Condition

Reason (GETSEC) IF in VMX non-root operation.

Use the opcode tables in this chapter to interpret IA-32 and Intel 64 architecture object code. Instructions are divided into encoding groups:

- 1-byte, 2-byte and 3-byte opcode encodings are used to encode integer, system, MMX technology, SSE/SSE2/SSE3/SSSE3/SSE4, and VMX instructions. Maps for these instructions are given in Table A-2 through Table A-6.
- Escape opcodes (in the format: ESC character, opcode, ModR/M byte) are used for floating-point instructions. The maps for these instructions are provided in Table A-7 through Table A-22.

## NOTE

All blanks in opcode maps are reserved and must not be used. Do not depend on the operation of undefined or blank opcodes.

## A.1 USING OPCODE TABLES

Tables in this appendix list opcodes of instructions (including required instruction prefixes, opcode extensions in associated ModR/M byte). Blank cells in the tables indicate opcodes that are reserved or undefined.

The opcode map tables are organized by hex values of the upper and lower 4 bits of an opcode byte. For 1-byte encodings (Table A-2), use the four high-order bits of an opcode to index a row of the opcode table; use the four low-order bits to index a column of the table. For 2-byte opcodes beginning with 0FH (Table A-3), skip any instruction prefixes, the 0FH byte (0FH may be preceded by 66H, F2H, or F3H) and use the upper and lower 4-bit values of the next opcode byte to index table rows and columns. Similarly, for 3-byte opcodes beginning with 0F38H or 0F3AH (Table A-4), skip any instruction prefixes, 0F38H or 0F3AH and use the upper and lower 4-bit values of the third opcode byte to index table rows and columns. See Section A.2.4, "Opcode Look-up Examples for One, Two, and Three-Byte Opcodes."

When a ModR/M byte provides opcode extensions, this information qualifies opcode execution. For information on how an opcode extension in the ModR/M byte modifies the opcode map in Table A-2 and Table A-3, see Section A.4.

The escape (ESC) opcode tables for floating point instructions identify the eight high order bits of opcodes at the top of each page. See Section A.5. If the accompanying ModR/M byte is in the range of 00H-BFH, bits 3-5 (the top row of the third table on each page) along with the reg bits of ModR/M determine the opcode. ModR/M bytes



outside the range of 00H-BFH are mapped by the bottom two tables on each page of the section.

## A.2 KEY TO ABBREVIATIONS

Operands are identified by a two-character code of the form Zz. The first character, an uppercase letter, specifies the addressing method; the second character, a lower-case letter, specifies the type of operand.

### A.2.1 Codes for Addressing Method

The following abbreviations are used to document addressing methods:

- A Direct address: the instruction has no ModR/M byte; the address of the operand is encoded in the instruction. No base register, index register, or scaling factor can be applied (for example, far JMP (EA)).
- B The VEX.vvvv field of the VEX prefix selects a general purpose register.
- C The reg field of the ModR/M byte selects a control register (for example, MOV (0F20, 0F22)).
- D The reg field of the ModR/M byte selects a debug register (for example, MOV (0F21,0F23)).
- E A ModR/M byte follows the opcode and specifies the operand. The operand is either a general-purpose register or a memory address. If it is a memory address, the address is computed from a segment register and any of the following values: a base register, an index register, a scaling factor, a displacement.
- F EFLAGS/RFLAGS Register.
- G The reg field of the ModR/M byte selects a general register (for example, AX (000)).
- H The VEX.vvvv field of the VEX prefix selects a 128-bit XMM register or a 256-bit YMM register, determined by operand type. For legacy SSE encodings this operand does not exist, changing the instruction to destructive form.
- I Immediate data: the operand value is encoded in subsequent bytes of the instruction.
- J The instruction contains a relative offset to be added to the instruction pointer register (for example, JMP (0E9), LOOP).
- L The upper 4 bits of the 8-bit immediate selects a 128-bit XMM register or a 256-bit YMM register, determined by operand type. (the MSB is ignored in 32-bit mode)

- M The ModR/M byte may refer only to memory (for example, BOUND, LES, LDS, LSS, LFS, LGS, CMPXCHG8B).
- N The R/M field of the ModR/M byte selects a packed-quadword, MMX technology register.
- O The instruction has no ModR/M byte. The offset of the operand is coded as a word or double word (depending on address size attribute) in the instruction. No base register, index register, or scaling factor can be applied (for example, MOV (A0–A3)).
- P The reg field of the ModR/M byte selects a packed quadword MMX technology register.
- Q A ModR/M byte follows the opcode and specifies the operand. The operand is either an MMX technology register or a memory address. If it is a memory address, the address is computed from a segment register and any of the following values: a base register, an index register, a scaling factor, and a displacement.
- R The R/M field of the ModR/M byte may refer only to a general register (for example, MOV (0F20–0F23)).
- S The reg field of the ModR/M byte selects a segment register (for example, MOV (8C,8E)).
- U The R/M field of the ModR/M byte selects a 128-bit XMM register or a 256-bit YMM register, determined by operand type.
- V The reg field of the ModR/M byte selects a 128-bit XMM register or a 256-bit YMM register, determined by operand type.
- W A ModR/M byte follows the opcode and specifies the operand. The operand is either a 128-bit XMM register, a 256-bit YMM register (determined by operand type), or a memory address. If it is a memory address, the address is computed from a segment register and any of the following values: a base register, an index register, a scaling factor, and a displacement.
- X Memory addressed by the DS:rSI register pair (for example, MOVS, CMPS, OUTS, or LODS).
- Y Memory addressed by the ES:rDI register pair (for example, MOVS, CMPS, INS, STOS, or SCAS).

## A.2.2 Codes for Operand Type

The following abbreviations are used to document operand types:

- a Two one-word operands in memory or two double-word operands in memory, depending on operand-size attribute (used only by the BOUND instruction).
- b Byte, regardless of operand-size attribute.
- c Byte or word, depending on operand-size attribute.
- d Doubleword, regardless of operand-size attribute.

## OPCODE MAP

dq	Double-quadword, regardless of operand-size attribute.
p	32-bit, 48-bit, or 80-bit pointer, depending on operand-size attribute.
pd	128-bit or 256-bit packed double-precision floating-point data.
pi	Quadword MMX technology register (for example: mm0).
ps	128-bit or 256-bit packed single-precision floating-point data.
q	Quadword, regardless of operand-size attribute.
qq	Quad-Quadword (256-bits), regardless of operand-size attribute.
s	6-byte or 10-byte pseudo-descriptor.
sd	Scalar element of a 128-bit double-precision floating data.
ss	Scalar element of a 128-bit single-precision floating data.
si	Doubleword integer register (for example: eax).
v	Word, doubleword or quadword (in 64-bit mode), depending on operand-size attribute.
w	Word, regardless of operand-size attribute.
x	dq or qq based on the operand-size attribute.
y	Doubleword or quadword (in 64-bit mode), depending on operand-size attribute.
z	Word for 16-bit operand-size or doubleword for 32 or 64-bit operand-size.

### A.2.3 Register Codes

When an opcode requires a specific register as an operand, the register is identified by name (for example, AX, CL, or ESI). The name indicates whether the register is 64, 32, 16, or 8 bits wide.

A register identifier of the form eXX or rXX is used when register width depends on the operand-size attribute. eXX is used when 16 or 32-bit sizes are possible; rXX is used when 16, 32, or 64-bit sizes are possible. For example: eAX indicates that the AX register is used when the operand-size attribute is 16 and the EAX register is used when the operand-size attribute is 32. rAX can indicate AX, EAX or RAX.

When the REX.B bit is used to modify the register specified in the reg field of the opcode, this fact is indicated by adding "/x" to the register name to indicate the additional possibility. For example, rCX/r9 is used to indicate that the register could either be rCX or r9. Note that the size of r9 in this case is determined by the operand size attribute (just as for rCX).

## A.2.4 Opcode Look-up Examples for One, Two, and Three-Byte Opcodes

This section provides examples that demonstrate how opcode maps are used.

### A.2.4.1 One-Byte Opcode Instructions

The opcode map for 1-byte opcodes is shown in Table A-2. The opcode map for 1-byte opcodes is arranged by row (the least-significant 4 bits of the hexadecimal value) and column (the most-significant 4 bits of the hexadecimal value). Each entry in the table lists one of the following types of opcodes:

- Instruction mnemonics and operand types using the notations listed in Section A.2
- Opcodes used as an instruction prefix

For each entry in the opcode map that corresponds to an instruction, the rules for interpreting the byte following the primary opcode fall into one of the following cases:

- A ModR/M byte is required and is interpreted according to the abbreviations listed in Section A.1 and Chapter 2, “Instruction Format,” of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A*. Operand types are listed according to notations listed in Section A.2.
- A ModR/M byte is required and includes an opcode extension in the reg field in the ModR/M byte. Use Table A-6 when interpreting the ModR/M byte.
- Use of the ModR/M byte is reserved or undefined. This applies to entries that represent an instruction prefix or entries for instructions without operands that use ModR/M (for example: 60H, PUSHA; 06H, PUSH ES).

#### Example A-1. Look-up Example for 1-Byte Opcodes

Opcode 030500000000H for an ADD instruction is interpreted using the 1-byte opcode map (Table A-2) as follows:

- The first digit (0) of the opcode indicates the table row and the second digit (3) indicates the table column. This locates an opcode for ADD with two operands.
- The first operand (type Gv) indicates a general register that is a word or doubleword depending on the operand-size attribute. The second operand (type Ev) indicates a ModR/M byte follows that specifies whether the operand is a word or doubleword general-purpose register or a memory address.
- The ModR/M byte for this instruction is 05H, indicating that a 32-bit displacement follows (00000000H). The reg/opcode portion of the ModR/M byte (bits 3-5) is 000, indicating the EAX register.

The instruction for this opcode is ADD EAX, mem\_op, and the offset of mem\_op is 00000000H.

Some 1- and 2-byte opcodes point to group numbers (shaded entries in the opcode map table). Group numbers indicate that the instruction uses the reg/opcode bits in the ModR/M byte as an opcode extension (refer to Section A.4).

### A.2.4.2 Two-Byte Opcode Instructions

The two-byte opcode map shown in Table A-3 includes primary opcodes that are either two bytes or three bytes in length. Primary opcodes that are 2 bytes in length begin with an escape opcode 0FH. The upper and lower four bits of the second opcode byte are used to index a particular row and column in Table A-3.

Two-byte opcodes that are 3 bytes in length begin with a mandatory prefix (66H, F2H, or F3H) and the escape opcode (0FH). The upper and lower four bits of the third byte are used to index a particular row and column in Table A-3 (except when the second opcode byte is the 3-byte escape opcodes 38H or 3AH; in this situation refer to Section A.2.4.3).

For each entry in the opcode map, the rules for interpreting the byte following the primary opcode fall into one of the following cases:

- A ModR/M byte is required and is interpreted according to the abbreviations listed in Section A.1 and Chapter 2, "Instruction Format," of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*. The operand types are listed according to notations listed in Section A.2.
- A ModR/M byte is required and includes an opcode extension in the reg field in the ModR/M byte. Use Table A-6 when interpreting the ModR/M byte.
- Use of the ModR/M byte is reserved or undefined. This applies to entries that represent an instruction without operands that are encoded using ModR/M (for example: 0F77H, EMMS).

#### Example A-2. Look-up Example for 2-Byte Opcodes

Look-up opcode 0FA405000000003H for a SHLD instruction using Table A-3.

- The opcode is located in row A, column 4. The location indicates a SHLD instruction with operands Ev, Gv, and Ib. Interpret the operands as follows:
  - Ev: The ModR/M byte follows the opcode to specify a word or doubleword operand.
  - Gv: The reg field of the ModR/M byte selects a general-purpose register.
  - Ib: Immediate data is encoded in the subsequent byte of the instruction.
- The third byte is the ModR/M byte (05H). The mod and opcode/reg fields of ModR/M indicate that a 32-bit displacement is used to locate the first operand in memory and eAX as the second operand.
- The next part of the opcode is the 32-bit displacement for the destination memory operand (00000000H). The last byte stores immediate byte that provides the count of the shift (03H).

- By this breakdown, it has been shown that this opcode represents the instruction: SHLD DS:00000000H, EAX, 3.

### A.2.4.3 Three-Byte Opcode Instructions

The three-byte opcode maps shown in Table A-4 and Table A-5 includes primary opcodes that are either 3 or 4 bytes in length. Primary opcodes that are 3 bytes in length begin with two escape bytes 0F38H or 0F3A. The upper and lower four bits of the third opcode byte are used to index a particular row and column in Table A-4 or Table A-5.

Three-byte opcodes that are 4 bytes in length begin with a mandatory prefix (66H, F2H, or F3H) and two escape bytes (0F38H or 0F3AH). The upper and lower four bits of the fourth byte are used to index a particular row and column in Table A-4 or Table A-5.

For each entry in the opcode map, the rules for interpreting the byte following the primary opcode fall into the following case:

- A ModR/M byte is required and is interpreted according to the abbreviations listed in A.1 and Chapter 2, "Instruction Format," of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*. The operand types are listed according to notations listed in Section A.2.

#### Example A-3. Look-up Example for 3-Byte Opcodes

Look-up opcode 660F3A0FC108H for a PALIGNR instruction using Table A-5.

- 66H is a prefix and 0F3AH indicate to use Table A-5. The opcode is located in row 0, column F indicating a PALIGNR instruction with operands Vdq, Wdq, and Ib. Interpret the operands as follows:
  - Vdq: The reg field of the ModR/M byte selects a 128-bit XMM register.
  - Wdq: The R/M field of the ModR/M byte selects either a 128-bit XMM register or memory location.
  - Ib: Immediate data is encoded in the subsequent byte of the instruction.
- The next byte is the ModR/M byte (C1H). The reg field indicates that the first operand is XMM0. The mod shows that the R/M field specifies a register and the R/M indicates that the second operand is XMM1.
- The last byte is the immediate byte (08H).
- By this breakdown, it has been shown that this opcode represents the instruction: PALIGNR XMM0, XMM1, 8.

### A.2.4.4 VEX Prefix Instructions

Instructions that include a VEX prefix are organized relative to the 2-byte and 3-byte opcode maps, based on the VEX.mmmmm field encoding of implied 0F, 0F38H, 0F3AH, respectively. Each entry in the opcode map of a VEX-encoded instruction is based on the value of the opcode byte, similar to non-VEX-encoded instructions.

A VEX prefix includes several bit fields that encode implied 66H, F2H, F3H prefix functionality (VEX.pp) and operand size/opcode information (VEX.L). See chapter 4 for details.

Opcode tables A2-A6 include both instructions with a VEX prefix and instructions without a VEX prefix. Many entries are only made once, but represent both the VEX and non-VEX forms of the instruction. If the VEX prefix is present all the operands are valid and the mnemonic is usually prefixed with a "v". If the VEX prefix is not present the VEX.vvvv operand is not available and the prefix "v" is dropped from the mnemonic.

A few instructions exist only in VEX form and these are marked with a superscript "v".

Operand size of VEX prefix instructions can be determined by the operand type code. 128-bit vectors are indicated by 'dq', 256-bit vectors are indicated by 'qq', and instructions with operands supporting either 128 or 256-bit, determined by VEX.L, are indicated by 'x'. For example, the entry "VMOVUPD Vx,Wx" indicates both VEX.L=0 and VEX.L=1 are supported.

### A.2.5 Superscripts Utilized in Opcode Tables

Table A-1 contains notes on particular encodings. These notes are indicated in the following opcode maps by superscripts. Gray cells indicate instruction groupings.

**Table A-1. Superscripts Utilized in Opcode Tables**

Superscript Symbol	Meaning of Symbol
1A	Bits 5, 4, and 3 of ModR/M byte used as an opcode extension (refer to Section A.4, "Opcode Extensions For One-Byte And Two-byte Opcodes").
1B	Use the 0F0B opcode (UD2 instruction) or the 0FB9H opcode when deliberately trying to generate an invalid opcode exception (#UD).
1C	Some instructions use the same two-byte opcode. If the instruction has variations, or the opcode represents different instructions, the ModR/M byte will be used to differentiate the instruction. For the value of the ModR/M byte needed to decode the instruction, see Table A-6.
i64	The instruction is invalid or not encodable in 64-bit mode. 40 through 4F (single-byte INC and DEC) are REX prefix combinations when in 64-bit mode (use FE/FF Grp 4 and 5 for INC and DEC).
o64	Instruction is only available when in 64-bit mode.
d64	When in 64-bit mode, instruction defaults to 64-bit operand size and cannot encode 32-bit operand size.
f64	The operand size is forced to a 64-bit operand size when in 64-bit mode (prefixes that change operand size are ignored for this instruction in 64-bit mode).

**Table A-1. Superscripts Utilized in Opcode Tables**

Superscript Symbol	Meaning of Symbol
v	VEX form only exists. There is no legacy SSE form of the instruction. For Integer GPR instructions it means VEX prefix required.
v1	VEX128 & SSE forms only exist (no VEX256), when can't be inferred from the data size.

### A.3 ONE, TWO, AND THREE-BYTE OPCODE MAPS

See Table A-2 through Table A-5 below. The tables are multiple page presentations. Rows and columns with sequential relationships are placed on facing pages to make look-up tasks easier. Note that table footnotes are not presented on each page. Table footnotes for each table are presented on the last page of the table.



Table A-2. One-byte Opcode Map: (00H – F7H) \*

	0	1	2	3	4	5	6	7
0	ADD						PUSH ES <sup>64</sup>	POP ES <sup>64</sup>
	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, lb	rAX, lz		
1	ADC						PUSH SS <sup>64</sup>	POP SS <sup>64</sup>
	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, lb	rAX, lz		
2	AND						SEG=ES (Prefix)	DAA <sup>64</sup>
	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, lb	rAX, lz		
3	XOR						SEG=SS (Prefix)	AAA <sup>64</sup>
	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, lb	rAX, lz		
4	INC <sup>64</sup> general register / REX <sup>064</sup> Prefixes							
	eAX REX	eCX REX.B	eDX REX.X	eBX REX.XB	eSP REX.R	eBP REX.RB	eSI REX.RX	eDI REX.RXB
5	PUSH <sup>064</sup> general register							
	rAX/r8	rCX/r9	rDX/r10	rBX/r11	rSP/r12	rBP/r13	rSI/r14	rDI/r15
6	PUSHA <sup>64</sup> / PUSHAD <sup>64</sup>	POPA <sup>64</sup> / POPAD <sup>64</sup>	BOUND <sup>64</sup> Gv, Ma	ARPL <sup>64</sup> Ew, Gw MOVXD <sup>064</sup> Gv, Ev	SEG=FS (Prefix)	SEG=GS (Prefix)	Operand Size (Prefix)	Address Size (Prefix)
7	Jcc <sup>64</sup> , Jb - Short-displacement jump on condition							
	O	NO	B/NAE/C	NB/AE/NC	Z/E	NZ/NE	BE/NA	NBE/A
8	Immediate Grp 1 <sup>1A</sup>				TEST		XCHG	
	Eb, lb	Ev, lz	Eb, lb <sup>64</sup>	Ev, lb	Eb, Gb	Ev, Gv	Eb, Gb	Ev, Gv
9	NOP PAUSE(F3) XCHG r8, rAX	XCHG word, double-word or quad-word register with rAX						
		rCX/r9	rDX/r10	rBX/r11	rSP/r12	rBP/r13	rSI/r14	rDI/r15
A	MOV				MOVS/B Yb, Xb	MOVS/W/D/Q Yv, Xv	CMPS/B Xb, Yb	CMPS/W/D Xv, Yv
	AL, Ob	rAX, Ov	Ob, AL	Ov, rAX				
B	MOV immediate byte into byte register							
	AL/R8L, lb	CL/R9L, lb	DL/R10L, lb	BL/R11L, lb	AH/R12L, lb	CH/R13L, lb	DH/R14L, lb	BH/R15L, lb
C	Shift Grp 2 <sup>1A</sup>		RETN <sup>64</sup> lw	RETN <sup>64</sup>	LES <sup>64</sup> Gz, Mp VEX+2byte	LDS <sup>64</sup> Gz, Mp VEX+1byte	Grp 11 <sup>1A</sup> - MOV	
	Eb, lb	Ev, lb					Eb, lb	Ev, lz
D	Shift Grp 2 <sup>1A</sup>				AAM <sup>64</sup> lb	AAD <sup>64</sup> lb		XLAT/ XLATB
	Eb, 1	Ev, 1	Eb, CL	Ev, CL				
E	LOOPNE <sup>64</sup> / LOOPNZ <sup>64</sup> Jb	LOOPE <sup>64</sup> / LOOPZ <sup>64</sup> Jb	LOOP <sup>64</sup> Jb	JrCXZ <sup>64</sup> / Jb	IN		OUT	
					AL, lb	eAX, lb	lb, AL	lb, eAX
F	LOCK (Prefix)		REPNE (Prefix)	REP/REPE (Prefix)	HLT	CMC	Unary Grp 3 <sup>1A</sup>	
							Eb	Ev

Table A-2. One-byte Opcode Map: (08H – FFH) \*

	8	9	A	B	C	D	E	F
0	OR Eb, Gb   Ev, Gv   Gb, Eb   Gv, Ev   AL, lb   rAX, lz						PUSH CS <sup>64</sup>	2-byte escape (Table A-3)
1	SBB Eb, Gb   Ev, Gv   Gb, Eb   Gv, Ev   AL, lb   rAX, lz						PUSH DS <sup>64</sup>	POP DS <sup>64</sup>
2	SUB Eb, Gb   Ev, Gv   Gb, Eb   Gv, Ev   AL, lb   rAX, lz						SEG=CS (Prefix)	DAS <sup>64</sup>
3	CMP Eb, Gb   Ev, Gv   Gb, Eb   Gv, Ev   AL, lb   rAX, lz						SEG=DS (Prefix)	AAS <sup>64</sup>
4	DEC <sup>64</sup> general register / REX <sup>064</sup> Prefixes eAX REX.W   eCX REX.WB   eDX REX.WX   eBX REX.WXB   eSP REX.WR   eBP REX.WRB   eSI REX.WRX   eDI REX.WRXB							
5	POP <sup>d64</sup> into general register rAX/r8   rCX/r9   rDX/r10   rBX/r11   rSP/r12   rBP/r13   rSI/r14   rDI/r15							
6	PUSH <sup>d64</sup> lz	IMUL Gv, Ev, lz	PUSH <sup>d64</sup> lb	IMUL Gv, Ev, lb	INS/INSB Yb, DX	INS/INSD Yz, DX	OUTS/OUTSB DX, Xb	OUTS/OUTSD DX, Xz
7	Jcc <sup>64</sup> , Jb- Short displacement jump on condition S   NS   P/PE   NP/PO   L/NGE   NL/GE   LE/NG   NLE/G							
8	MOV Eb, Gb   Ev, Gv   Gb, Eb   Gv, Ev				MOV Ev, Sw	LEA Gv, M	MOV Sw, Ew	Grp 1A <sup>1A</sup> POP <sup>d64</sup> Ev
9	CBW/CWDE/CDQE	CWD/CDQ/CQO	CALLF <sup>64</sup> Ap	FWAIT/WAIT	PUSHF/D/Q <sup>d64</sup> Fv	POPF/D/Q <sup>d64</sup> Fv	SAHF	LAHF
A	TEST AL, lb   rAX, lz	STOS/B Yb, AL	STOS/W/D/Q Yv, rAX	LODS/B AL, Xb	LODS/W/D/Q rAX, Xv	SCAS/B AL, Yb	SCAS/W/D/Q rAX, Xv	
B	MOV immediate word or double into word, double, or quad register rAX/r8, lv   rCX/r9, lv   rDX/r10, lv   rBX/r11, lv   rSP/r12, lv   rBP/r13, lv   rSI/r14, lv   rDI/r15, lv							
C	ENTER lw, lb	LEAVE <sup>d64</sup>	RETF lw	RETF	INT 3	INT lb	INTO <sup>64</sup>	IRET/D/Q
D	ESC (Escape to coprocessor instruction set)							
E	CALL <sup>64</sup> Jz	near <sup>64</sup> Jz	JMP far <sup>64</sup> Ap	short <sup>64</sup> Jb	IN AL, DX   eAX, DX		OUT DX, AL   DX, eAX	
F	CLC	STC	CLI	STI	CLD	STD	INC/DEC Grp 4 <sup>1A</sup>	INC/DEC Grp 5 <sup>1A</sup>

NOTES:

\* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

**Table A-3. Two-byte Opcode Map: 00H – 77H (First Byte is 0FH) \***

	pxf	0	1	2	3	4	5	6	7
0		Grp 6 <sup>1A</sup>	Grp 7 <sup>1A</sup>	LAR Gv, Ew	LSL Gv, Ew		SYSCALL <sup>064</sup>	CLTS	SYSRET <sup>064</sup>
1		vmovups	vmovups	vmovlps Vq, Hq, Mq vmovhlps Vq, Hq, Uq	vmovlps Mq, Vq	vunpcklps Vx, Hx, Wx	vunpckhps Vx, Hx, Wx	vmovhps <sup>V1</sup> Vdq, Hq, Mq vmovlhps Vdq, Hq, Uq	vmovhps <sup>V1</sup> Mq, Vq
	66	vmovupd	vmovupd Vpd, Vpd	vmovlpd Vq, Hq, Mq	vmovlpd Mq, Vq	vunpcklpd Vx, Hx, Wx	vunpckhpd Vx, Hx, Wx	vmovhpd <sup>V1</sup> Vdq, Hq, Mq	vmovhpd <sup>V1</sup> Mq, Vq
	F3	vmovss Vx, Hx, Wss	vmovss Wss, Hx, Vss	vmovsldup Vx, Wx				vmovshdup Vx, Wx	
	F2	vmovsd Vx, Hx, Wsd	vmovsd Wsd, Hx, Vsd	vmovddup Vx, Wx					
2	2	MOV Rd, Cd	MOV Rd, Dd	MOV Cd, Rd	MOV Dd, Rd				
3	3	WRMSR	RDTSR	RDMSR	RDPMSR	SYSENTER	SYSEXIT		GETSEC
4	4	CMOVcc, (Gv, Ev) - Conditional Move							
		O	NO	B/C/NAE	AE/NB/NC	E/Z	NE/NZ	BE/NA	A/NBE
5		vmovmskps Gy, Ups	vsqrtps Vps, Wps	vrsqrtps Vps, Wps	vrcpps Vps, Wps	vandps Vps, Hps, Wps	vandnps Vps, Hps, Wps	vorps Vps, Hps, Wps	vxorps Vps, Hps, Wps
	66	vmovmskpd Gy, Upd	vsqrtpd Vpd, Wpd			vandpd Vpd, Hpd, Wpd	vandnpd Vpd, Hpd, Wpd	vorpd Vpd, Hpd, Wpd	vxorpd Vpd, Hpd, Wpd
	F3		vsqrtss Vss, Hss, Wss	vrsqrtss Vss, Hss, Wss	vrcpps Vss, Hss, Wss				
	F2		vsrtsd Vsd, Hsd, Wsd						
6		punpcklbw Pq, Qd	punpcklwd Pq, Qd	punpckldq Pq, Qd	packsswb Pq, Qq	pcmpgtb Pq, Qq	pcmpgtw Pq, Qq	pcmpgtd Pq, Qq	packuswb Pq, Qq
	66	vpunpcklbw Vx, Hx, Wx	vpunpcklwd Vx, Hx, Wx	vpunpckldq Vx, Hx, Wx	vpacksswb Vx, Hx, Wx	vpcmpgtb Vx, Hx, Wx	vpcmpgtw Vx, Hx, Wx	vpcmpgtd Vx, Hx, Wx	vpackuswb Vx, Hx, Wx
	F3								
7		pshufw Pq, Qq, lb	(Grp 12 <sup>1A</sup> )	(Grp 13 <sup>1A</sup> )	(Grp 14 <sup>1A</sup> )	pcmpeqb Pq, Qq	pcmpeqw Pq, Qq	pcmpeqd Pq, Qq	emms vzeroupper <sup>V</sup> vzeroall <sup>V</sup>
	66	vpshufd Vx, Wx, lb				vpcmpeqb Vx, Hx, Wx	vpcmpeqw Vx, Hx, Wx	vpcmpeqd Vx, Hx, Wx	
	F3	vpshufhw Vx, Wx, lb							
	F2	vpshufw Vx, Wx, lb							

**Table A-3. Two-byte Opcode Map: 08H – 7FH (First Byte is 0FH) \***

	pxf	8	9	A	B	C	D	E	F
0		INVD	WBINVD		2-byte Illegal Opcodes UD2 <sup>1B</sup>		NOP Ev		
1		Prefetch <sup>1C</sup> (Grp 16 <sup>1A</sup> )							NOP Ev
2		vmovaps Vps, Wps	vmovaps Wps, Vps	cvtpi2ps Vps, Qpi	vmovntps Mps, Vps	cvttps2pi Ppi, Wps	cvtps2pi Ppi, Wps	vucomiss Vss, Wss	vcomiss Vss, Wss
	66	vmovapd Vpd, Wpd	vmovapd Wpd, Vpd	cvtpi2pd Vpd, Qpi	vmovntpd Mpd, Vpd	cvttpd2pi Ppi, Wpd	cvtpd2pi Qpi, Wpd	vucomisd Vsd, Wsd	vcomisd Vsd, Wsd
	F3			vcvttsi2ss Vss, Hss, Ey		vcvttss2si Gy, Wss	vcvtss2si Gy, Wss		
	F2			vcvttsi2sd Vsd, Hsd, Ey		vcvttss2si Gy, Wsd	vcvtss2si Gy, Wsd		
3	3	3-byte escape (Table A-4)		3-byte escape (Table A-5)					
4	4	CMOVcc(Gv, Ev) - Conditional Move							
		S	NS	P/PE	NP/PO	L/NGE	NL/GE	LE/NG	NLE/G
5		vaddps Vps, Hps, Wps	vmulps Vps, Hps, Wps	vcvtps2pd Vpd, Wps	vcvtdq2ps Vps, Wdq	vsubps Vps, Hps, Wps	vminps Vps, Hps, Wps	vdivps Vps, Hps, Wps	vmaxps Vps, Hps, Wps
	66	vaddpd Vpd, Hpd, Wpd	vmulpd Vpd, Hpd, Wpd	vcvtpd2ps Vps, Wpd	vcvtps2dq Vdq, Wps	vsubpd Vpd, Hpd, Wpd	vminpd Vpd, Hpd, Wpd	vdivpd Vpd, Hpd, Wpd	vmaxpd Vpd, Hpd, Wpd
	F3	vaddss Vss, Hss, Wss	vmulss Vss, Hss, Wss	vcvtss2sd Vsd, Hx, Wss	vcvtpps2dq Vdq, Wps	vsubss Vss, Hss, Wss	vminss Vss, Hss, Wss	vdivss Vss, Hss, Wss	vmaxss Vss, Hss, Wss
	F2	vaddsd Vsd, Hsd, Wsd	vmulsd Vsd, Hsd, Wsd	vcvtss2sd Vss, Hx, Wsd		vsubsd Vsd, Hsd, Wsd	vminsd Vsd, Hsd, Wsd	vdivsd Vsd, Hsd, Wsd	vmaxsd Vsd, Hsd, Wsd
6		punpckhbw Pq, Qd	punpckhwd Pq, Qd	punpckhdq Pq, Qd	packssdw Pq, Qd			movd/q Pd, Ey	movq Pq, Qq
	66	vpunpckhbw Vx, Hx, Wx	vpunpckhwd Vx, Hx, Wx	vpunpckhdq Vx, Hx, Wx	vpackssdw Vx, Hx, Wx	vpunpckldq Vx, Hx, Wx	vpunpckhdq Vx, Hx, Wx	vmovd/q Vy, Ey	vmovdqa Vx, Wx
	F3								vmovdqu Vx, Wx
7		VMREAD Ey, Gy	VMWRITE Gy, Ey					movd/q Ey, Pd	movq Qq, Pq
	66					vhaddpd Vpd, Hpd, Wpd	vhsbpd Vpd, Hpd, Wpd	vmovd/q Ey, Vy	vmovdqa Wx, Vx
	F3							vmovq Vq, Wq	vmovdqu Wx, Vx
	F2					vhaddps Vps, Hps, Wps	vhsbpps Vps, Hps, Wps		

OPCODE MAP

Table A-3. Two-byte Opcode Map: 80H – F7H (First Byte is 0FH) \*

pfh	0	1	2	3	4	5	6	7	
8	Jcc <sup>64</sup> , Jz - Long-displacement jump on condition								
	O	NO	B/C/NAE	AE/NB/NC	E/Z	NE/NZ	BE/NA	A/NBE	
9	SETcc, Eb - Byte Set on condition								
	O	NO	B/C/NAE	AE/NB/NC	E/Z	NE/NZ	BE/NA	A/NBE	
A	PUSH <sup>d64</sup> FS	POP <sup>d64</sup> FS	CPUID	BT Ev, Gv	SHLD Ev, Gv, Ib	SHLD Ev, Gv, CL			
B	CMPXCHG Eb, Gb		LSS Gv, Mp	BTR Ev, Gv	LFS Gv, Mp	LGS Gv, Mp	MOVZX Gv, Eb, Gv, Ew		
C	XADD Eb, Gb	XADD Ev, Gv	vcmps Vps, Hps, Wps, Ib	movnti My, Gy	pinsrw Pq, Ry/Mw, Ib	pextrw Gd, Nq, Ib	vshufps Vps, Hps, Wps, Ib	Grp 9 <sup>1A</sup>	
			vcmpd Vpd, Hpd, Wpd, Ib		vpinsrw Vdq, Hdq, Ry/Mw, Ib	vpextrw Gd, Udq, Ib	vshufpd Vpd, Hpd, Wpd, Ib		
	F3		vcmpss Vss, Hss, Wss, Ib						
	F2		vcmps Vsd, Hsd, Wsd, Ib						
D		psrlw Pq, Qq	psrl Pq, Qq	psrlq Pq, Qq	paddq Pq, Qq	pmullw Pq, Qq		pmovmskb Gd, Nq	
	66	vaddsubpd Vpd, Hpd, Wpd	vpsrlw Vx, Hx, Wx	vpsrl Vx, Hx, Wx	vpsrlq Vx, Hx, Wx	vpaddq Vx, Hx, Wx	vpmullw Vx, Hx, Wx	vmovq Wq, Vq	vpmovmskb Gd, Ux
	F3							movq2dq Vdq, Nq	
	F2	vaddsubps Vps, Hps, Wps						movdq2q Pq, Uq	
E		pavgb Pq, Qq	psraw Pq, Qq	psrad Pq, Qq	pavgw Pq, Qq	pmulhw Pq, Qq	pmulhw Pq, Qq		movntq Mq, Pq
	66	vpavgb Vx, Hx, Wx	vpsraw Vx, Hx, Wx	vpsrad Vx, Hx, Wx	vpavgw Vx, Hx, Wx	vpmulhw Vx, Hx, Wx	vpmulhw Vx, Hx, Wx	vcvt2dq Vx, Wpd	vmovntdq Mx, Vx
	F3							vcvt2pd Vx, Wpd	
	F2							vcvt2dq Vx, Wpd	
F			psllw Pq, Qq	pslld Pq, Qq	psllq Pq, Qq	pmuludq Pq, Qq	pmaddwd Pq, Qq	psadbw Pq, Qq	maskmovq Pq, Nq
	66		vpsllw Vx, Hx, Wx	vpslld Vx, Hx, Wx	vpsllq Vx, Hx, Wx	vpmuludq Vx, Hx, Wx	vpmaddwd Vx, Hx, Wx	vpsadbw Vx, Hx, Wx	vmaskmovdq Vx, Ux
	F2	vlddqu Vx, Mx							

**Table A-3. Two-byte Opcode Map: 88H – FFH (First Byte is 0FH) \***

	px	8	9	A	B	C	D	E	F
8		Jcc <sup>64</sup> , Jz - Long-displacement jump on condition							
		S	NS	P/PE	NP/PO	L/NGE	NL/GE	LE/NG	NLE/G
9		SETcc, Eb - Byte Set on condition							
		S	NS	P/PE	NP/PO	L/NGE	NL/GE	LE/NG	NLE/G
A		PUSH <sup>d64</sup> GS	POP <sup>d64</sup> GS	RSM	BTS Ev, Gv	SHRD Ev, Gv, Ib	SHRD Ev, Gv, CL	(Grp 15 <sup>1A</sup> ) <sup>1C</sup>	IMUL Gv, Ev
B		<i>JMPE</i> (reserved for emulator on IPF)	Grp 10 <sup>1A</sup> Invalid Opcode <sup>1B</sup>	Grp 8 <sup>1A</sup> Ev, Ib	BTC Ev, Gv	BSF Gv, Ev	BSR Gv, Ev	MOVSBX Gv, Eb   Gv, Ew	
	F3	POPCNT Gv, Ev				TZCNT Gv, Ev	LZCNT Gv, Ev		
C		BSWAP							
		RAX/EAX/ R8/R8D	RCX/ECX/ R9/R9D	RDX/EDX/ R10/R10D	RBX/EBX/ R11/R11D	RSP/ESP/ R12/R12D	RBP/EBP/ R13/R13D	RSI/ESI/ R14/R14D	EDI/EDI/ R15/R15D
D		psubusb Pq, Qq	psubusw Pq, Qq	pminub Pq, Qq	pand Pq, Qq	paddusb Pq, Qq	paddusw Pq, Qq	pmaxub Pq, Qq	pandn Pq, Qq
	66	vpsubusb Vx, Hx, Wx	vpsubusw Vx, Hx, Wx	vpminub Vx, Hx, Wx	vpand Vx, Hx, Wx	vpaddusb Vx, Hx, Wx	vpaddusw Vx, Hx, Wx	vpmaxub Vx, Hx, Wx	vpandn Vx, Hx, Wx
	F3								
	F2								
E		psubsb Pq, Qq	psubsw Pq, Qq	pminsw Pq, Qq	por Pq, Qq	paddsb Pq, Qq	paddsw Pq, Qq	pmaxsw Pq, Qq	pxor Pq, Qq
	66	vpsubsb Vx, Hx, Wx	vpsubsw Vx, Hx, Wx	vpminsw Vx, Hx, Wx	vpor Vx, Hx, Wx	vpaddsb Vx, Hx, Wx	vpaddsw Vx, Hx, Wx	vpmaxsw Vx, Hx, Wx	vpxor Vx, Hx, Wx
	F3								
	F2								
F		psubb Pq, Qq	psubw Pq, Qq	psubd Pq, Qq	psubq Pq, Qq	paddb Pq, Qq	paddw Pq, Qq	paddd Pq, Qq	
	66	vpsubb Vx, Hx, Wx	vpsubw Vx, Hx, Wx	vpsubd Vx, Hx, Wx	vpsubq Vx, Hx, Wx	vpaddb Vx, Hx, Wx	vpaddw Vx, Hx, Wx	vpaddd Vx, Hx, Wx	
	F2								

**NOTES:**

\* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

OPCODE MAP

**Table A-4. Three-byte Opcode Map: 00H — F7H (First Two Bytes are 0F 38H) \***

	pxf	0	1	2	3	4	5	6	7
0		pshufb Pq, Qq	phaddw Pq, Qq	phadd Pq, Qq	phaddsw Pq, Qq	pmaddubsw Pq, Qq	phsubw Pq, Qq	phsubd Pq, Qq	phsubsw Pq, Qq
	66	vpshufb Vx, Hx, Wx	vphaddw Vx, Hx, Wx	vphadd Vx, Hx, Wx	vphaddsw Vx, Hx, Wx	vpmaddubsw Vx, Hx, Wx	vphsubw Vx, Hx, Wx	vphsubd Vx, Hx, Wx	vphsubsw Vx, Hx, Wx
1	66	pblendvb Vdq, Wdq			vcvtph2ps <sup>V</sup> Vx, Wx, lb	blendvps Vdq, Wdq	blendvpd Vdq, Wdq	vpermps <sup>V</sup> Vqq, Hqq, Wqq	vptest Vx, Wx
2	66	vpmovsxbw Vx, Ux/Mq	vpmovsxbd Vx, Ux/Md	vpmovsxbq Vx, Ux/Mw	vpmovsxdw Vx, Ux/Mq	vpmovsxwq Vx, Ux/Md	vpmovsxdq Vx, Ux/Mq		
3	66	vpmovzxbw Vx, Ux/Mq	vpmovzxbd Vx, Ux/Md	vpmovzxbq Vx, Ux/Mw	vpmovzxdw Vx, Ux/Mq	vpmovzxwq Vx, Ux/Md	vpmovzxdq Vx, Ux/Mq	vpermd <sup>V</sup> Vqq, Hqq, Wqq	vpcmpgtq Vx, Hx, Wx
4	66	vpmulld Vx, Hx, Wx	vphminposuw Vdq, Wdq				vpsrld/q <sup>V</sup> Vx, Hx, Wx	vpsravd <sup>V</sup> Vx, Hx, Wx	vpsllvd/q <sup>V</sup> Vx, Hx, Wx
5									
6									
7									
8	66	INVEPT Gy, Mdq	INVVPID Gy, Mdq	INVPCID Gy, Mdq					
9	66	vgatherdd/q <sup>V</sup> Vx,Hx,Wx	vgatherqd/q <sup>V</sup> Vx,Hx,Wx	vgatherdps/d <sup>V</sup> Vx,Hx,Wx	vgatherqps/d <sup>V</sup> Vx,Hx,Wx			vfmaddsub132ps/d <sup>V</sup> Vx,Hx,Wx	vfmsubadd132ps/d <sup>V</sup> Vx,Hx,Wx
A	66							vfmaddsub213ps/d <sup>V</sup> Vx,Hx,Wx	vfmsubadd213ps/d <sup>V</sup> Vx,Hx,Wx
B	66							vfmaddsub231ps/d <sup>V</sup> Vx,Hx,Wx	vfmsubadd231ps/d <sup>V</sup> Vx,Hx,Wx
C									
D									
E									
F		MOVBE Gy, My	MOVBE My, Gy	ANDN <sup>V</sup> Gy, By, Ey	Grp 17 <sup>1A</sup>		BZHI <sup>V</sup> Gy, Ey, By		BEXTR <sup>V</sup> Gy, Ey, By
	66	MOVBE Gw, Mw	MOVBE Mw, Gw						SHLX <sup>V</sup> Gy, Ey, By
	F3						PEXT <sup>V</sup> Gy, By, Ey		SARX <sup>V</sup> Gy, Ey, By
	F2	CRC32 Gd, Eb	CRC32 Gd, Ey				PDEP <sup>V</sup> Gy, By, Ey	MULX <sup>V</sup> By,Gy,rDX,Ey	SHRX <sup>V</sup> Gy, Ey, By
	66 & F2	CRC32 Gd, Eb	CRC32 Gd, Ew						

**Table A-4. Three-byte Opcode Map: 08H — FFH (First Two Bytes are 0F 38H) \***

	pxf	8	9	A	B	C	D	E	F
0		psignb Pq, Qq	psignw Pq, Qq	psignd Pq, Qq	pmulhrsw Pq, Qq				
	66	vpsignb Vx, Hx, Wx	vpsignw Vx, Hx, Wx	vpsignd Vx, Hx, Wx	vpmulhrsw Vx, Hx, Wx	vpermilps <sup>v</sup> Vx,Hx,Wx	vpermilpd <sup>v</sup> Vx,Hx,Wx	vtestps <sup>v</sup> Vx, Wx	vtestpd <sup>v</sup> Vx, Wx
1						pabsb Pq, Qq	pabsw Pq, Qq	pabsd Pq, Qq	
	66	vbroadcastss <sup>v</sup> Vx, Wd	vbroadcasts <sup>v</sup> Vq, Wq	vbroadcastf128 <sup>v</sup> Vq, Mdq		vpabsb Vx, Wx	vpabsw Vx, Wx	vpabsd Vx, Wx	
2	66	vpmuldq Vx, Hx, Wx	vpcmpeq Vx, Hx, Wx	vmovntdqa Vx, Mx	vpackusdw Vx, Hx, Wx	vmaskmovps <sup>v</sup> Vx,Hx,Mx	vmaskmovpd <sup>v</sup> Vx,Hx,Mx	vmaskmovps <sup>v</sup> Mx,Hx,Vx	vmaskmovpd <sup>v</sup> Mx,Hx,Vx
3	66	vpminsb Vx, Hx, Wx	vpmins d Vx, Hx, Wx	vpminuw Vx, Hx, Wx	vpminud Vx, Hx, Wx	vpmaxsb Vx, Hx, Wx	vpmaxsd Vx, Hx, Wx	vpmaxuw Vx, Hx, Wx	vpmaxud Vx, Hx, Wx
4									
5	66	vpbroadcast <sup>v</sup> Vx, Wx	vpbroadcast <sup>v</sup> Vx, Wx	vpbroadcasti128 <sup>v</sup> Vq, Mdq					
6									
7	66	vpbroadcast <sup>v</sup> Vx, Wx	vpbroadcast <sup>v</sup> Vx, Wx						
8	66					vpmaskmovd/q <sup>v</sup> Vx,Hx,Mx		vpmaskmovd/q <sup>v</sup> Mx,Vx,Hx	
9	66	vfmadd132ps/d <sup>v</sup> Vx, Hx, Wx	vfmadd132ss/d <sup>v</sup> Vx, Hx, Wx	vfmsub132ps/d <sup>v</sup> Vx, Hx, Wx	vfmsub132ss/d <sup>v</sup> Vx, Hx, Wx	vfnmadd132ps/d <sup>v</sup> Vx, Hx, Wx	vfnmadd132ss/d <sup>v</sup> Vx, Hx, Wx	vfnmsub132ps/d <sup>v</sup> Vx, Hx, Wx	vfnmsub132ss/d <sup>v</sup> Vx, Hx, Wx
A	66	vfmadd213ps/d <sup>v</sup> Vx, Hx, Wx	vfmadd213ss/d <sup>v</sup> Vx, Hx, Wx	vfmsub213ps/d <sup>v</sup> Vx, Hx, Wx	vfmsub213ss/d <sup>v</sup> Vx, Hx, Wx	vfnmadd213ps/d <sup>v</sup> Vx, Hx, Wx	vfnmadd213ss/d <sup>v</sup> Vx, Hx, Wx	vfnmsub213ps/d <sup>v</sup> Vx, Hx, Wx	vfnmsub213ss/d <sup>v</sup> Vx, Hx, Wx
B	66	vfmadd231ps/d <sup>v</sup> Vx, Hx, Wx	vfmadd231ss/d <sup>v</sup> Vx, Hx, Wx	vfmsub231ps/d <sup>v</sup> Vx, Hx, Wx	vfmsub231ss/d <sup>v</sup> Vx, Hx, Wx	vfnmadd231ps/d <sup>v</sup> Vx, Hx, Wx	vfnmadd231ss/d <sup>v</sup> Vx, Hx, Wx	vfnmsub231ps/d <sup>v</sup> Vx, Hx, Wx	vfnmsub231ss/d <sup>v</sup> Vx, Hx, Wx
C									
D	66				VAESIMC Vdq, Wdq	VAEENC Vdq,Hdq,Wdq	VAEENCLAST Vdq,Hdq,Wdq	VAESDEC Vdq,Hdq,Wdq	VAESDECLAST Vdq,Hdq,Wdq
E									
F	66								
	F3								
	F2								
	66 & F2								

**NOTES:**

\* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.



**Table A-5. Three-byte Opcode Map: 00H — F7H (First two bytes are 0F 3AH) \***

	pref	0	1	2	3	4	5	6	7
0	66	vpermq <sup>v</sup> Vqq, Wqq, lb	vpermpd <sup>v</sup> Vqq, Wqq, lb	vpblendd <sup>v</sup> Vx,Hx,Wx,lb		vpermilps <sup>v</sup> Vx, Wx, lb	vpermilpd <sup>v</sup> Vx, Wx, lb	vperm2f128 <sup>v</sup> Vqq,Hqq,Wqq,lb	
1	66					vpextrb Rd/Mb, Vdq, lb	vpextrw Rd/Mw, Vdq, lb	vpextrd/q Ey, Vdq, lb	vextractps Ed, Vdq, lb
2	66	vpinsrb Vdq,Hdq, Ry/Mb,lb	vinsertps Vdq,Hdq, Udq/Md,lb	vpinsrd/q Vdq,Hdq,Ey,lb					
3									
4	66	vdpps Vx,Hx,Wx,lb	vdppd Vdq,Hdq,Wdq,lb	vmpsadbw Vx,Hx,Wx,lb		vpclmulqdq Vdq,Hdq,Wdq,lb		vperm2i128 <sup>v</sup> Vqq,Hqq,Wqq,lb	
5									
6	66	vpcmpstrm Vdq, Wdq, lb	vpcmpstri Vdq, Wdq, lb	vpcmpstrm Vdq, Wdq, lb	vpcmpstri Vdq, Wdq, lb				
7									
8									
9									
A									
B									
C									
D									
E									
F	F2	RORX <sup>v</sup> Gy, Ey, lb							

**Table A-5. Three-byte Opcode Map: 08H – FFH (First Two Bytes are 0F 3AH) \***

	pxf	8	9	A	B	C	D	E	F
0									palignr Pq, Qq, Ib
	66	vroundps Vx, Wx, Ib	vroundpd Vx, Wx, Ib	vroundss Vss, Wss, Ib	vroundsd Vsd, Wsd, Ib	vblendps Vx, Hx, Wx, Ib	vblendpd Vx, Hx, Wx, Ib	vpblendw Vx, Hx, Wx, Ib	vpalignr Vx, Hx, Wx, Ib
1	66	vinserftf128 <sup>v</sup> Vqq, Hqq, Wqq, Ib	vextractf128 <sup>v</sup> Wdq, Vqq, Ib				vcvtps2ph <sup>v</sup> Wx, Vx, Ib		
2									
3	66	vinserfti128 <sup>v</sup> Vqq, Hqq, Wqq, Ib	vextracti128 <sup>v</sup> Wdq, Vqq, Ib						
4	66			vblendvps <sup>v</sup> Vx, Hx, Wx, Lx	vblendvpd <sup>v</sup> Vx, Hx, Wx, Lx	vpblendvb <sup>v</sup> Vx, Hx, Wx, Lx			
5									
6									
7									
8									
9									
A									
B									
C									
D	66								VAESKEYGEN Vdq, Wdq, Ib
E									
F									

**NOTES:**

\* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

## A.4 OPCODE EXTENSIONS FOR ONE-BYTE AND TWO-BYTE OPCODES

Some 1-byte and 2-byte opcodes use bits 3-5 of the ModR/M byte (the nnn field in Figure A-1) as an extension of the opcode.

<b>mod</b>	<b>nnn</b>	<b>R/M</b>
------------	------------	------------

**Figure A-1. ModR/M Byte nnn Field (Bits 5, 4, and 3)**

OpCodes that have opcode extensions are indicated in Table A-6 and organized by group number. Group numbers (from 1 to 16, second column) provide a table entry point. The encoding for the r/m field for each instruction can be established using the third column of the table.

### A.4.1 Opcode Look-up Examples Using Opcode Extensions

An Example is provided below.

#### Example A-4. Interpreting an ADD Instruction

An ADD instruction with a 1-byte opcode of 80H is a Group 1 instruction:

- Table A-6 indicates that the opcode extension field encoded in the ModR/M byte for this instruction is 000B.
- The r/m field can be encoded to access a register (11B) or a memory address using a specified addressing mode (for example: mem = 00B, 01B, 10B).

#### Example A-5. Looking Up 0F01C3H

Look up opcode 0F01C3 for a VMRESUME instruction by using Table A-2, Table A-3 and Table A-6:

- 0F tells us that this instruction is in the 2-byte opcode map.
- 01 (row 0, column 1 in Table A-3) reveals that this opcode is in Group 7 of Table A-6.
- C3 is the ModR/M byte. The first two bits of C3 are 11B. This tells us to look at the second of the Group 7 rows in Table A-6.
- The Op/Reg bits [5,4,3] are 000B. This tells us to look in the 000 column for Group 7.
- Finally, the R/M bits [2,1,0] are 011B. This identifies the opcode as the VMRESUME instruction.

## A.4.2 Opcode Extension Tables

See Table A-6 below.

**Table A-6. Opcode Extensions for One- and Two-byte Opcodes by Group Number \***

Opcode	Group	Mod 7,6	pfx	Encoding of Bits 5,4,3 of the ModR/M Byte (bits 2,1,0 in parenthesis)							
				000	001	010	011	100	101	110	111
80-83	1	mem, 11B		ADD	OR	ADC	SBB	AND	SUB	XOR	CMP
8F	1A	mem, 11B		POP							
C0,C1 reg, imm D0, D1 reg, 1 D2, D3 reg, CL	2	mem, 11B		ROL	ROR	RCL	RCR	SHL/SAL	SHR		SAR
F6, F7	3	mem, 11B		TEST lb/lz		NOT	NEG	MUL AL/rAX	IMUL AL/rAX	DIV AL/rAX	IDIV AL/rAX
FE	4	mem, 11B		INC Eb	DEC Eb						
FF	5	mem, 11B		INC Ev	DEC Ev	CALLN <sup>f64</sup> Ev	CALLF Ep	JMPN <sup>f64</sup> Ev	JMPF Mp	PUSH <sup>d64</sup> Ev	
0F 00	6	mem, 11B		SLDT Rv/Mw	STR Rv/Mw	LLDT Ew	LTR Ew	VERR Ew	VERW Ew		
0F 01	7	mem		SGDT Ms	SIDT Ms	LGDT Ms	LIDT Ms	SMSW Mw/Rv		LMSW Ew	INVLPG Mb
		11B	VMCALL (001) VMLAUNCH (010) VMRESUME (011) VMXOFF (100)	MONITOR (000) MWAIT (001)	XGETBV (000) XSETBV (001) VMFUNC (100)						
0F BA	8	mem, 11B						BT	BTS	BTR	BTC
0F C7	9	mem			CMPXCH8B Mq CMPXCHG16B Mdq					VMPTRLD Mq	VMPTRST Mq
			66							VMCLEAR Mq	
		F3								VMXON Mq	VMPTRST Mq
		11B								RDRAND Rv	
0F B9	10	mem 11B									
C6	11	mem, 11B		MOV Eb, lb							
C7		mem 11B		MOV Ev, lz							

**Table A-6. Opcode Extensions for One- and Two-byte Opcodes by Group Number \***

Opcode	Group	Mod 7,6	pfx	Encoding of Bits 5,4,3 of the ModR/M Byte (bits 2,1,0 in parenthesis)								
				000	001	010	011	100	101	110	111	
0F 71	12	mem										
		11B				psrlw Nq, lb		psraw Nq, lb		psllw Nq, lb		
			66			vpsrlw Hx,Ux,lb		vpsraw Hx,Ux,lb		vpsllw Hx,Ux,lb		
0F 72	13	mem										
		11B				psrld Nq, lb		psrad Nq, lb		pslld Nq, lb		
			66			vpsrld Hx,Ux,lb		vpsrad Hx,Ux,lb		vpslld Hx,Ux,lb		
0F 73	14	mem										
		11B				psrlq Nq, lb				psllq Nq, lb		
			66			vpsrlq Hx,Ux,lb	vpsrlq Hx,Ux,lb			vpsllq Hx,Ux,lb	vpsllq Hx,Ux,lb	
0F AE	15	mem		fxsave	fxrstor	ldmxcsr	stmxcsr	XSAVE	XRSTOR	XSAVEOPT	clflush	
		11B								lfence	mfence	sfence
			F3	RDFSBASE Ry	RDGSBASE Ry	WRFSBASE Ry	WRGSBASE Ry					
0F 18	16	mem		prefetch NTA	prefetch T0	prefetch T1	prefetch T2					
		11B										
VEX.0F38 F3	17	mem			BLSR <sup>V</sup> By, Ey	BLSMSK <sup>V</sup> By, Ey	BLSI <sup>V</sup> By, Ey					
		11B										

**NOTES:**

\* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

## A.5 ESCAPE OPCODE INSTRUCTIONS

Opcode maps for coprocessor escape instruction opcodes (x87 floating-point instruction opcodes) are in Table A-7 through Table A-22. These maps are grouped by the first byte of the opcode, from D8-DF. Each of these opcodes has a ModR/M byte. If the ModR/M byte is within the range of 00H-BFH, bits 3-5 of the ModR/M byte are used as an opcode extension, similar to the technique used for 1- and 2-byte opcodes (see A.4). If the ModR/M byte is outside the range of 00H through BFH, the entire ModR/M byte is used as an opcode extension.

### A.5.1 Opcode Look-up Examples for Escape Instruction Opcodes

Examples are provided below.

#### Example A-6. Opcode with ModR/M Byte in the 00H through BFH Range

DD0504000000H can be interpreted as follows:

- The instruction encoded with this opcode can be located in Section . Since the ModR/M byte (05H) is within the 00H through BFH range, bits 3 through 5 (000) of this byte indicate the opcode for an FLD double-real instruction (see Table A-9).
- The double-real value to be loaded is at 00000004H (the 32-bit displacement that follows and belongs to this opcode).

#### Example A-7. Opcode with ModR/M Byte outside the 00H through BFH Range

D8C1H can be interpreted as follows:

- This example illustrates an opcode with a ModR/M byte outside the range of 00H through BFH. The instruction can be located in Section A.4.
- In Table A-8, the ModR/M byte C1H indicates row C, column 1 (the FADD instruction using ST(0), ST(1) as operands).

### A.5.2 Escape Opcode Instruction Tables

Tables are listed below.

### A.5.2.1 Escape Opcodes with D8 as First Byte

Table A-7 and A-8 contain maps for the escape instruction opcodes that begin with D8H. Table A-7 shows the map if the ModR/M byte is in the range of 00H-BFH. Here, the value of bits 3-5 (the nnn field in Figure A-1) selects the instruction.

**Table A-7. D8 Opcode Map When ModR/M Byte is Within 00H to BFH \***

nnn Field of ModR/M Byte (refer to Figure A.4)							
000B	001B	010B	011B	100B	101B	110B	111B
FADD single-real	FMUL single-real	FCOM single-real	FCOMP single-real	FSUB single-real	FSUBR single-real	FDIV single-real	FDIVR single-real

**NOTES:**

- \* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

Table A-8 shows the map if the ModR/M byte is outside the range of 00H-BFH. Here, the first digit of the ModR/M byte selects the table row and the second digit selects the column.

**Table A-8. D8 Opcode Map When ModR/M Byte is Outside 00H to BFH \***

	0	1	2	3	4	5	6	7
C	FADD							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
D	FCOM							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),T(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
E	FSUB							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
F	FDIV							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)

	8	9	A	B	C	D	E	F
C	FMUL							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
D	FCOMP							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),T(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
E	FSUBR							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
F	FDIVR							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)

**NOTES:**

- \* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

### A.5.2.2 Escape Opcodes with D9 as First Byte

Table A-9 and A-10 contain maps for escape instruction opcodes that begin with D9H. Table A-9 shows the map if the ModR/M byte is in the range of 00H-BFH. Here, the value of bits 3-5 (the nnn field in Figure A-1) selects the instruction.

**Table A-9. D9 Opcode Map When ModR/M Byte is Within 00H to BFH \***

nnn Field of ModR/M Byte							
000B	001B	010B	011B	100B	101B	110B	111B
FLD single-real		FST single-real	FSTP single-real	FLDENV 14/28 bytes	FLDCW 2 bytes	FSTENV 14/28 bytes	FSTCW 2 bytes

**NOTES:**

- \* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

Table A-10 shows the map if the ModR/M byte is outside the range of 00H-BFH. Here, the first digit of the ModR/M byte selects the table row and the second digit selects the column.

**Table A-10. D9 Opcode Map When ModR/M Byte is Outside 00H to BFH \***

	0	1	2	3	4	5	6	7
C	FLD							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
D	FNOP							
E	FCHS	FABS			FTST	FXAM		
F	F2XM1	FYL2X	FPTAN	FPATAN	FXTRACT	FPREM1	FDECSTP	FINCSTP

	8	9	A	B	C	D	E	F
C	FXCH							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
D								
E	FLD1	FLDL2T	FLDL2E	FLDPI	FLDLG2	FLDLN2	FLDZ	
F	FPREM	FYL2XP1	FSQRT	FSINCOS	FRNDINT	FSCALE	FSIN	FCOS

**NOTES:**

- \* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.



### A.5.2.3 Escape Opcodes with DA as First Byte

Table A-11 and A-12 contain maps for escape instruction opcodes that begin with DAH. Table A-11 shows the map if the ModR/M byte is in the range of 00H-BFH. Here, the value of bits 3-5 (the nnn field in Figure A-1) selects the instruction.

**Table A-11. DA Opcode Map When ModR/M Byte is Within 00H to BFH \***

nnn Field of ModR/M Byte							
000B	001B	010B	011B	100B	101B	110B	111B
FIADD dword-integer	FIMUL dword-integer	FICOM dword-integer	FICOMP dword-integer	FISUB dword-integer	FISUBR dword-integer	FIDIV dword-integer	FIDIVR dword-integer

**NOTES:**

\* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

Table A-11 shows the map if the ModR/M byte is outside the range of 00H-BFH. Here, the first digit of the ModR/M byte selects the table row and the second digit selects the column.

**Table A-12. DA Opcode Map When ModR/M Byte is Outside 00H to BFH \***

	0	1	2	3	4	5	6	7
C	FCMOVB							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
D	FCMOVBE							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
E								
F								

	8	9	A	B	C	D	E	F
C	FCMOVE							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
D	FCMOVU							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
E		FUCOMPP						
F								

**NOTES:**

\* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

### A.5.2.4 Escape Opcodes with DB as First Byte

Table A-13 and A-14 contain maps for escape instruction opcodes that begin with DBH. Table A-13 shows the map if the ModR/M byte is in the range of 00H-BFH. Here, the value of bits 3-5 (the nnn field in Figure A-1) selects the instruction.

**Table A-13. DB Opcode Map When ModR/M Byte is Within 00H to BFH \***

nnn Field of ModR/M Byte							
000B	001B	010B	011B	100B	101B	110B	111B
FILD dword-integer	FISTTP dword-integer	FIST dword-integer	FISTP dword-integer		FLD extended-real		FSTP extended-real

**NOTES:**

\* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

Table A-14 shows the map if the ModR/M byte is outside the range of 00H-BFH. Here, the first digit of the ModR/M byte selects the table row and the second digit selects the column.

**Table A-14. DB Opcode Map When ModR/M Byte is Outside 00H to BFH \***

	0	1	2	3	4	5	6	7
C	FCMOVNB							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
D	FCMOVNBE							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
E			FCLEX	FINIT				
F	FCOMI							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)

	8	9	A	B	C	D	E	F
C	FCMOVNE							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
D	FCMOVNU							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
E	FUCOMI							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
F								

**NOTES:**

\* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

### A.5.2.5 Escape Opcodes with DC as First Byte

Table A-15 and A-16 contain maps for escape instruction opcodes that begin with DCH. Table A-15 shows the map if the ModR/M byte is in the range of 00H-BFH. Here, the value of bits 3-5 (the nnn field in Figure A-1) selects the instruction.

**Table A-15. DC Opcode Map When ModR/M Byte is Within 00H to BFH \***

nnn Field of ModR/M Byte (refer to Figure A-1)							
000B	001B	010B	011B	100B	101B	110B	111B
FADD double-real	FMUL double-real	FCOM double-real	FCOMP double-real	FSUB double-real	FSUBR double-real	FDIV double-real	FDIVR double-real

**NOTES:**

\* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

Table A-16 shows the map if the ModR/M byte is outside the range of 00H-BFH. In this case the first digit of the ModR/M byte selects the table row and the second digit selects the column.

**Table A-16. DC Opcode Map When ModR/M Byte is Outside 00H to BFH \***

	0	1	2	3	4	5	6	7
C	FADD							
	ST(0),ST(0)	ST(1),ST(0)	ST(2),ST(0)	ST(3),ST(0)	ST(4),ST(0)	ST(5),ST(0)	ST(6),ST(0)	ST(7),ST(0)
D								
E	FSUBR							
	ST(0),ST(0)	ST(1),ST(0)	ST(2),ST(0)	ST(3),ST(0)	ST(4),ST(0)	ST(5),ST(0)	ST(6),ST(0)	ST(7),ST(0)
F	FDIVR							
	ST(0),ST(0)	ST(1),ST(0)	ST(2),ST(0)	ST(3),ST(0)	ST(4),ST(0)	ST(5),ST(0)	ST(6),ST(0)	ST(7),ST(0)

	8	9	A	B	C	D	E	F
C	FMUL							
	ST(0),ST(0)	ST(1),ST(0)	ST(2),ST(0)	ST(3),ST(0)	ST(4),ST(0)	ST(5),ST(0)	ST(6),ST(0)	ST(7),ST(0)
D								
E	FSUB							
	ST(0),ST(0)	ST(1),ST(0)	ST(2),ST(0)	ST(3),ST(0)	ST(4),ST(0)	ST(5),ST(0)	ST(6),ST(0)	ST(7),ST(0)
F	FDIV							
	ST(0),ST(0)	ST(1),ST(0)	ST(2),ST(0)	ST(3),ST(0)	ST(4),ST(0)	ST(5),ST(0)	ST(6),ST(0)	ST(7),ST(0)

**NOTES:**

\* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

### A.5.2.6 Escape Opcodes with DD as First Byte

Table A-17 and A-18 contain maps for escape instruction opcodes that begin with DDH. Table A-17 shows the map if the ModR/M byte is in the range of 00H-BFH. Here, the value of bits 3-5 (the nnn field in Figure A-1) selects the instruction.

**Table A-17. DD Opcode Map When ModR/M Byte is Within 00H to BFH \***

nnn Field of ModR/M Byte							
000B	001B	010B	011B	100B	101B	110B	111B
FLD double-real	FISTTP integer64	FST double-real	FSTP double-real	FRSTOR 98/108bytes		FSAVE 98/108bytes	FSTSW 2 bytes

**NOTES:**

\* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

Table A-18 shows the map if the ModR/M byte is outside the range of 00H-BFH. The first digit of the ModR/M byte selects the table row and the second digit selects the column.

**Table A-18. DD Opcode Map When ModR/M Byte is Outside 00H to BFH \***

	0	1	2	3	4	5	6	7
C	FFREE							
	ST(0)	ST(1)	ST(2)	ST(3)	ST(4)	ST(5)	ST(6)	ST(7)
D	FST							
	ST(0)	ST(1)	ST(2)	ST(3)	ST(4)	ST(5)	ST(6)	ST(7)
E	FUCOM							
	ST(0),ST(0)	ST(1),ST(0)	ST(2),ST(0)	ST(3),ST(0)	ST(4),ST(0)	ST(5),ST(0)	ST(6),ST(0)	ST(7),ST(0)
F								

	8	9	A	B	C	D	E	F
C								
D	FSTP							
	ST(0)	ST(1)	ST(2)	ST(3)	ST(4)	ST(5)	ST(6)	ST(7)
E	FUCOMP							
	ST(0)	ST(1)	ST(2)	ST(3)	ST(4)	ST(5)	ST(6)	ST(7)
F								

**NOTES:**

\* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

### A.5.2.7 Escape Opcodes with DE as First Byte

Table A-19 and A-20 contain opcode maps for escape instruction opcodes that begin with DEH. Table A-19 shows the opcode map if the ModR/M byte is in the range of 00H-BFH. In this case, the value of bits 3-5 (the nnn field in Figure A-1) selects the instruction.

**Table A-19. DE Opcode Map When ModR/M Byte is Within 00H to BFH \***

nnn Field of ModR/M Byte							
000B	001B	010B	011B	100B	101B	110B	111B
FIADD word-integer	FIMUL word-integer	FICOM word-integer	FICOMP word-integer	FISUB word-integer	FISUBR word-integer	FIDIV word-integer	FIDIVR word-integer

**NOTES:**

\* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

Table A-20 shows the opcode map if the ModR/M byte is outside the range of 00H-BFH. The first digit of the ModR/M byte selects the table row and the second digit selects the column.

**Table A-20. DE Opcode Map When ModR/M Byte is Outside 00H to BFH \***

	0	1	2	3	4	5	6	7
C	FADDP							
	ST(0),ST(0)	ST(1),ST(0)	ST(2),ST(0)	ST(3),ST(0)	ST(4),ST(0)	ST(5),ST(0)	ST(6),ST(0)	ST(7),ST(0)
D								
E	FSUBRP							
	ST(0),ST(0)	ST(1),ST(0)	ST(2),ST(0)	ST(3),ST(0)	ST(4),ST(0)	ST(5),ST(0)	ST(6),ST(0)	ST(7),ST(0)
F	FDIVRP							
	ST(0),ST(0)	ST(1),ST(0)	ST(2),ST(0)	ST(3),ST(0)	ST(4),ST(0)	ST(5),ST(0)	ST(6),ST(0)	ST(7),ST(0)

	8	9	A	B	C	D	E	F
C	FMULP							
	ST(0),ST(0)	ST(1),ST(0)	ST(2),ST(0)	ST(3),ST(0)	ST(4),ST(0)	ST(5),ST(0)	ST(6),ST(0)	ST(7),ST(0)
D		FCOMP						
E	FSUBP							
	ST(0),ST(0)	ST(1),ST(0)	ST(2),ST(0)	ST(3),ST(0)	ST(4),ST(0)	ST(5),ST(0)	ST(6),ST(0)	ST(7),ST(0)
F	FDIVP							
	ST(0),ST(0)	ST(1),ST(0)	ST(2),ST(0)	ST(3),ST(0)	ST(4),ST(0)	ST(5),ST(0)	ST(6),ST(0)	ST(7),ST(0)

**NOTES:**

\* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

### A.5.2.8 Escape Opcodes with DF As First Byte

Table A-21 and A-22 contain the opcode maps for escape instruction opcodes that begin with DFH. Table A-21 shows the opcode map if the ModR/M byte is in the range of 00H-BFH. Here, the value of bits 3-5 (the nnn field in Figure A-1) selects the instruction.

**Table A-21. DF Opcode Map When ModR/M Byte is Within 00H to BFH \***

nnn Field of ModR/M Byte							
000B	001B	010B	011B	100B	101B	110B	111B
FILD word-integer	FISTTP word-integer	FIST word-integer	FISTP word-integer	FBLD packed-BCD	FILD qword-integer	FBSTP packed-BCD	FISTP qword-integer

**NOTES:**

\* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

Table A-22 shows the opcode map if the ModR/M byte is outside the range of 00H-BFH. The first digit of the ModR/M byte selects the table row and the second digit selects the column.

**Table A-22. DF Opcode Map When ModR/M Byte is Outside 00H to BFH \***

	0	1	2	3	4	5	6	7
C								
D								
E	FSTSW AX							
F	FCOMIP							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)

	8	9	A	B	C	D	E	F
C								
D								
E	FUCOMP							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
F								

**NOTES:**

\* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

OPCODE MAP

# APPENDIX B

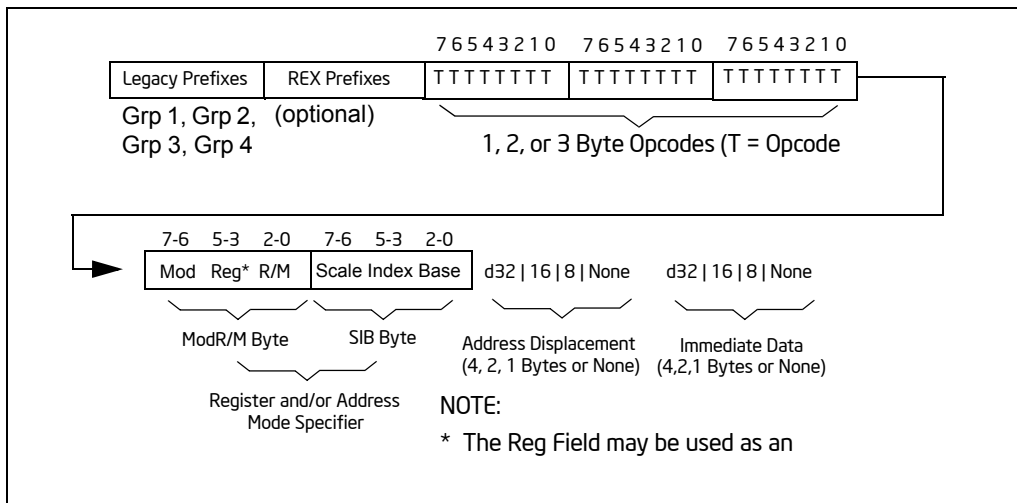
## INSTRUCTION FORMATS AND ENCODINGS

This appendix provides machine instruction formats and encodings of IA-32 instructions. The first section describes the IA-32 architecture's machine instruction format. The remaining sections show the formats and encoding of general-purpose, MMX, P6 family, SSE/SSE2/SSE3, x87 FPU instructions, and VMX instructions. Those instruction formats also apply to Intel 64 architecture. Instruction formats used in 64-bit mode are provided as supersets of the above.

### B.1 MACHINE INSTRUCTION FORMAT

All Intel Architecture instructions are encoded using subsets of the general machine instruction format shown in Figure B-1. Each instruction consists of:

- an opcode
- a register and/or address mode specifier consisting of the ModR/M byte and sometimes the scale-index-base (SIB) byte (if required)
- a displacement and an immediate data field (if required)



**Figure B-1. General Machine Instruction Format**

The following sections discuss this format.



### B.1.1 Legacy Prefixes

The legacy prefixes noted in Figure B-1 include 66H, 67H, F2H and F3H. They are optional, except when F2H, F3H and 66H are used in new instruction extensions. Legacy prefixes must be placed before REX prefixes.

Refer to Chapter 2, “Instruction Format,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A*, for more information on legacy prefixes.

### B.1.2 REX Prefixes

REX prefixes are a set of 16 opcodes that span one row of the opcode map and occupy entries 40H to 4FH. These opcodes represent valid instructions (INC or DEC) in IA-32 operating modes and in compatibility mode. In 64-bit mode, the same opcodes represent the instruction prefix REX and are not treated as individual instructions.

Refer to Chapter 2, “Instruction Format,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A*, for more information on REX prefixes.

### B.1.3 Opcode Fields

The primary opcode for an instruction is encoded in one to three bytes of the instruction. Within the primary opcode, smaller encoding fields may be defined. These fields vary according to the class of operation being performed.

Almost all instructions that refer to a register and/or memory operand have a register and/or address mode byte following the opcode. This byte, the ModR/M byte, consists of the mod field (2 bits), the reg field (3 bits; this field is sometimes an opcode extension), and the R/M field (3 bits). Certain encodings of the ModR/M byte indicate that a second address mode byte, the SIB byte, must be used.

If the addressing mode specifies a displacement, the displacement value is placed immediately following the ModR/M byte or SIB byte. Possible sizes are 8, 16, or 32 bits. If the instruction specifies an immediate value, the immediate value follows any displacement bytes. The immediate, if specified, is always the last field of the instruction.

Refer to Chapter 2, “Instruction Format,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A*, for more information on opcodes.

### B.1.4 Special Fields

Table B-1 lists bit fields that appear in certain instructions, sometimes within the opcode bytes. All of these fields (except the d bit) occur in the general-purpose instruction formats in Table B-13.

**Table B-1. Special Fields Within Instruction Encodings**

Field Name	Description	Number of Bits
reg	General-register specifier (see Table B-4 or B-5)	3
w	Specifies if data is byte or full-sized, where full-sized is 16 or 32 bits (see Table B-6)	1
s	Specifies sign extension of an immediate field (see Table B-7)	1
sreg2	Segment register specifier for CS, SS, DS, ES (see Table B-8)	2
sreg3	Segment register specifier for CS, SS, DS, ES, FS, GS (see Table B-8)	3
eee	Specifies a special-purpose (control or debug) register (see Table B-9)	3
ttn	For conditional instructions, specifies a condition asserted or negated (see Table B-12)	4
d	Specifies direction of data operation (see Table B-11)	1

#### B.1.4.1 Reg Field (reg) for Non-64-Bit Modes

The reg field in the ModR/M byte specifies a general-purpose register operand. The group of registers specified is modified by the presence and state of the w bit in an encoding (refer to Section B.1.4.3). Table B-2 shows the encoding of the reg field when the w bit is not present in an encoding; Table B-3 shows the encoding of the reg field when the w bit is present.

**Table B-2. Encoding of reg Field When w Field is Not Present in Instruction**

reg Field	Register Selected during 16-Bit Data Operations	Register Selected during 32-Bit Data Operations
000	AX	EAX
001	CX	ECX
010	DX	EDX
011	BX	EBX
100	SP	ESP
101	BP	EBP
110	SI	ESI
111	DI	EDI

**Table B-3. Encoding of reg Field When w Field is Present in Instruction**

Register Specified by reg Field During 16-Bit Data Operations			Register Specified by reg Field During 32-Bit Data Operations		
reg	Function of w Field		reg	Function of w Field	
	When w = 0	When w = 1		When w = 0	When w = 1
000	AL	AX	000	AL	EAX
001	CL	CX	001	CL	ECX
010	DL	DX	010	DL	EDX
011	BL	BX	011	BL	EBX
100	AH	SP	100	AH	ESP
101	CH	BP	101	CH	EBP
110	DH	SI	110	DH	ESI
111	BH	DI	111	BH	EDI

### B.1.4.2 Reg Field (reg) for 64-Bit Mode

Just like in non-64-bit modes, the reg field in the ModR/M byte specifies a general-purpose register operand. The group of registers specified is modified by the presence of and state of the w bit in an encoding (refer to Section B.1.4.3). Table B-4 shows the encoding of the reg field when the w bit is not present in an encoding; Table B-5 shows the encoding of the reg field when the w bit is present.

**Table B-4. Encoding of reg Field When w Field is Not Present in Instruction**

reg Field	Register Selected during 16-Bit Data Operations	Register Selected during 32-Bit Data Operations	Register Selected during 64-Bit Data Operations
000	AX	EAX	RAX
001	CX	ECX	RCX
010	DX	EDX	RDX
011	BX	EBX	RBX
100	SP	ESP	RSP
101	BP	EBP	RBP
110	SI	ESI	RSI
111	DI	EDI	RDI

**Table B-5. Encoding of reg Field When w Field is Present in Instruction**

Register Specified by reg Field During 16-Bit Data Operations			Register Specified by reg Field During 32-Bit Data Operations		
reg	Function of w Field		reg	Function of w Field	
	When w = 0	When w = 1		When w = 0	When w = 1
000	AL	AX	000	AL	EAX
001	CL	CX	001	CL	ECX
010	DL	DX	010	DL	EDX
011	BL	BX	011	BL	EBX
100	AH <sup>1</sup>	SP	100	AH*	ESP
101	CH <sup>1</sup>	BP	101	CH*	EBP
110	DH <sup>1</sup>	SI	110	DH*	ESI
111	BH <sup>1</sup>	DI	111	BH*	EDI

**NOTES:**

1. AH, CH, DH, BH can not be encoded when REX prefix is used. Such an expression defaults to the low byte.

**B.1.4.3 Encoding of Operand Size (w) Bit**

The current operand-size attribute determines whether the processor is performing 16-bit, 32-bit or 64-bit operations. Within the constraints of the current operand-size attribute, the operand-size bit (w) can be used to indicate operations on 8-bit operands or the full operand size specified with the operand-size attribute. Table B-6 shows the encoding of the w bit depending on the current operand-size attribute.

**Table B-6. Encoding of Operand Size (w) Bit**

w Bit	Operand Size When Operand-Size Attribute is 16 Bits	Operand Size When Operand-Size Attribute is 32 Bits
0	8 Bits	8 Bits
1	16 Bits	32 Bits

**B.1.4.4 Sign-Extend (s) Bit**

The sign-extend (s) bit occurs in instructions with immediate data fields that are being extended from 8 bits to 16 or 32 bits. See Table B-7.

**Table B-7. Encoding of Sign-Extend (s) Bit**

<b>s</b>	<b>Effect on 8-Bit Immediate Data</b>	<b>Effect on 16- or 32-Bit Immediate Data</b>
0	None	None
1	Sign-extend to fill 16-bit or 32-bit destination	None

### B.1.4.5 Segment Register (sreg) Field

When an instruction operates on a segment register, the reg field in the ModR/M byte is called the sreg field and is used to specify the segment register. Table B-8 shows the encoding of the sreg field. This field is sometimes a 2-bit field (sreg2) and other times a 3-bit field (sreg3).

**Table B-8. Encoding of the Segment Register (sreg) Field**

<b>2-Bit sreg2 Field</b>	<b>Segment Register Selected</b>	<b>3-Bit sreg3 Field</b>	<b>Segment Register Selected</b>
00	ES	000	ES
01	CS	001	CS
10	SS	010	SS
11	DS	011	DS
		100	FS
		101	GS
		110	Reserved <sup>1</sup>
		111	Reserved

#### NOTES:

1. Do not use reserved encodings.

### B.1.4.6 Special-Purpose Register (eee) Field

When control or debug registers are referenced in an instruction they are encoded in the eee field, located in bits 5 through 3 of the ModR/M byte (an alternate encoding of the sreg field). See Table B-9.

**Table B-9. Encoding of Special-Purpose Register (eee) Field**

eee	Control Register	Debug Register
000	CR0	DR0
001	Reserved <sup>1</sup>	DR1
010	CR2	DR2
011	CR3	DR3
100	CR4	Reserved
101	Reserved	Reserved
110	Reserved	DR6
111	Reserved	DR7

**NOTES:**

1. Do not use reserved encodings.

**B.1.4.7 Condition Test (ttn) Field**

For conditional instructions (such as conditional jumps and set on condition), the condition test field (ttn) is encoded for the condition being tested. The ttt part of the field gives the condition to test and the n part indicates whether to use the condition ( $n = 0$ ) or its negation ( $n = 1$ ).

- For 1-byte primary opcodes, the ttn field is located in bits 3, 2, 1, and 0 of the opcode byte.
- For 2-byte primary opcodes, the ttn field is located in bits 3, 2, 1, and 0 of the second opcode byte.

Table B-10 shows the encoding of the ttn field.

**Table B-10. Encoding of Conditional Test (ttn) Field**

<b>t t t n</b>	<b>Mnemonic</b>	<b>Condition</b>
0000	O	Overflow
0001	NO	No overflow
0010	B, NAE	Below, Not above or equal
0011	NB, AE	Not below, Above or equal
0100	E, Z	Equal, Zero
0101	NE, NZ	Not equal, Not zero
0110	BE, NA	Below or equal, Not above
0111	NBE, A	Not below or equal, Above
1000	S	Sign
1001	NS	Not sign
1010	P, PE	Parity, Parity Even
1011	NP, PO	Not parity, Parity Odd
1100	L, NGE	Less than, Not greater than or equal to
1101	NL, GE	Not less than, Greater than or equal to
1110	LE, NG	Less than or equal to, Not greater than
1111	NLE, G	Not less than or equal to, Greater than

#### B.1.4.8 Direction (d) Bit

In many two-operand instructions, a direction bit (d) indicates which operand is considered the source and which is the destination. See Table B-11.

- When used for integer instructions, the d bit is located at bit 1 of a 1-byte primary opcode. Note that this bit does not appear as the symbol “d” in Table B-13; the actual encoding of the bit as 1 or 0 is given.
- When used for floating-point instructions (in Table B-16), the d bit is shown as bit 2 of the first byte of the primary opcode.

**Table B-11. Encoding of Operation Direction (d) Bit**

<b>d</b>	<b>Source</b>	<b>Destination</b>
0	reg Field	ModR/M or SIB Byte
1	ModR/M or SIB Byte	reg Field

## B.1.5 Other Notes

Table B-12 contains notes on particular encodings. These notes are indicated in the tables shown in the following sections by superscripts.

**Table B-12. Notes on Instruction Encoding**

Symbol	Note
A	A value of 11B in bits 7 and 6 of the ModR/M byte is reserved.
B	A value of 01B (or 10B) in bits 7 and 6 of the ModR/M byte is reserved.

## B.2 GENERAL-PURPOSE INSTRUCTION FORMATS AND ENCODINGS FOR NON-64-BIT MODES

Table B-13 shows machine instruction formats and encodings for general purpose instructions in non-64-bit modes.

**Table B-13. General Purpose Instruction Formats and Encodings for Non-64-Bit Modes**

Instruction and Format	Encoding
<b>AAA – ASCII Adjust after Addition</b>	0011 0111
<b>AAD – ASCII Adjust AX before Division</b>	1101 0101 : 0000 1010
<b>AAM – ASCII Adjust AX after Multiply</b>	1101 0100 : 0000 1010
<b>AAS – ASCII Adjust AL after Subtraction</b>	0011 1111
<b>ADC – ADD with Carry</b>	
register1 to register2	0001 000w : 11 reg1 reg2
register2 to register1	0001 001w : 11 reg1 reg2
memory to register	0001 001w : mod reg r/m
register to memory	0001 000w : mod reg r/m
immediate to register	1000 00sw : 11 010 reg : immediate data
immediate to AL, AX, or EAX	0001 010w : immediate data
immediate to memory	1000 00sw : mod 010 r/m : immediate data
<b>ADD – Add</b>	
register1 to register2	0000 000w : 11 reg1 reg2
register2 to register1	0000 001w : 11 reg1 reg2
memory to register	0000 001w : mod reg r/m
register to memory	0000 000w : mod reg r/m



**Table B-13. General Purpose Instruction Formats and Encodings  
for Non-64-Bit Modes (Contd.)**

<b>Instruction and Format</b>	<b>Encoding</b>
immediate to register	1000 00sw : 11 000 reg : immediate data
immediate to AL, AX, or EAX	0000 010w : immediate data
immediate to memory	1000 00sw : mod 000 r/m : immediate data
<b>AND - Logical AND</b>	
register1 to register2	0010 000w : 11 reg1 reg2
register2 to register1	0010 001w : 11 reg1 reg2
memory to register	0010 001w : mod reg r/m
register to memory	0010 000w : mod reg r/m
immediate to register	1000 00sw : 11 100 reg : immediate data
immediate to AL, AX, or EAX	0010 010w : immediate data
immediate to memory	1000 00sw : mod 100 r/m : immediate data
<b>ARPL - Adjust RPL Field of Selector</b>	
from register	0110 0011 : 11 reg1 reg2
from memory	0110 0011 : mod reg r/m
<b>BOUND - Check Array Against Bounds</b>	0110 0010 : mod <sup>A</sup> reg r/m
<b>BSF - Bit Scan Forward</b>	
register1, register2	0000 1111 : 1011 1100 : 11 reg1 reg2
memory, register	0000 1111 : 1011 1100 : mod reg r/m
<b>BSR - Bit Scan Reverse</b>	
register1, register2	0000 1111 : 1011 1101 : 11 reg1 reg2
memory, register	0000 1111 : 1011 1101 : mod reg r/m
<b>BSWAP - Byte Swap</b>	0000 1111 : 1100 1 reg
<b>BT - Bit Test</b>	
register, immediate	0000 1111 : 1011 1010 : 11 100 reg: imm8 data
memory, immediate	0000 1111 : 1011 1010 : mod 100 r/m : imm8 data
register1, register2	0000 1111 : 1010 0011 : 11 reg2 reg1
memory, reg	0000 1111 : 1010 0011 : mod reg r/m
<b>BTC - Bit Test and Complement</b>	

**Table B-13. General Purpose Instruction Formats and Encodings  
for Non-64-Bit Modes (Contd.)**

Instruction and Format	Encoding
register, immediate	0000 1111 : 1011 1010 : 11 111 reg: imm8 data
memory, immediate	0000 1111 : 1011 1010 : mod 111 r/m : imm8 data
register1, register2	0000 1111 : 1011 1011 : 11 reg2 reg1
memory, reg	0000 1111 : 1011 1011 : mod reg r/m
<b>BTR - Bit Test and Reset</b>	
register, immediate	0000 1111 : 1011 1010 : 11 110 reg: imm8 data
memory, immediate	0000 1111 : 1011 1010 : mod 110 r/m : imm8 data
register1, register2	0000 1111 : 1011 0011 : 11 reg2 reg1
memory, reg	0000 1111 : 1011 0011 : mod reg r/m
<b>BTS - Bit Test and Set</b>	
register, immediate	0000 1111 : 1011 1010 : 11 101 reg: imm8 data
memory, immediate	0000 1111 : 1011 1010 : mod 101 r/m : imm8 data
register1, register2	0000 1111 : 1010 1011 : 11 reg2 reg1
memory, reg	0000 1111 : 1010 1011 : mod reg r/m
<b>CALL - Call Procedure (in same segment)</b>	
direct	1110 1000 : full displacement
register indirect	1111 1111 : 11 010 reg
memory indirect	1111 1111 : mod 010 r/m
<b>CALL - Call Procedure (in other segment)</b>	
direct	1001 1010 : unsigned full offset, selector
indirect	1111 1111 : mod 011 r/m
<b>CBW - Convert Byte to Word</b>	1001 1000
<b>CDQ - Convert Doubleword to Qword</b>	1001 1001
<b>CLC - Clear Carry Flag</b>	1111 1000
<b>CLD - Clear Direction Flag</b>	1111 1100

**Table B-13. General Purpose Instruction Formats and Encodings  
for Non-64-Bit Modes (Contd.)**

Instruction and Format	Encoding
<b>CLI - Clear Interrupt Flag</b>	1111 1010
<b>CLTS - Clear Task-Switched Flag in CRO</b>	0000 1111 : 0000 0110
<b>CMC - Complement Carry Flag</b>	1111 0101
<b>CMP - Compare Two Operands</b>	
register1 with register2	0011 100w : 11 reg1 reg2
register2 with register1	0011 101w : 11 reg1 reg2
memory with register	0011 100w : mod reg r/m
register with memory	0011 101w : mod reg r/m
immediate with register	1000 00sw : 11 111 reg : immediate data
immediate with AL, AX, or EAX	0011 110w : immediate data
immediate with memory	1000 00sw : mod 111 r/m : immediate data
<b>CMPS/CMPSB/CMPSW/CMPSD - Compare String Operands</b>	1010 011w
<b>CMPXCHG - Compare and Exchange</b>	
register1, register2	0000 1111 : 1011 000w : 11 reg2 reg1
memory, register	0000 1111 : 1011 000w : mod reg r/m
<b>CPUID - CPU Identification</b>	0000 1111 : 1010 0010
<b>CWD - Convert Word to Doubleword</b>	1001 1001
<b>CWDE - Convert Word to Doubleword</b>	1001 1000
<b>DAA - Decimal Adjust AL after Addition</b>	0010 0111
<b>DAS - Decimal Adjust AL after Subtraction</b>	0010 1111
<b>DEC - Decrement by 1</b>	
register	1111 111w : 11 001 reg
register (alternate encoding)	0100 1 reg
memory	1111 111w : mod 001 r/m
<b>DIV - Unsigned Divide</b>	
AL, AX, or EAX by register	1111 011w : 11 110 reg
AL, AX, or EAX by memory	1111 011w : mod 110 r/m
<b>HLT - Halt</b>	1111 0100

**Table B-13. General Purpose Instruction Formats and Encodings  
for Non-64-Bit Modes (Contd.)**

Instruction and Format	Encoding
<b>IDIV - Signed Divide</b>	
AL, AX, or EAX by register	1111 011w : 11 111 reg
AL, AX, or EAX by memory	1111 011w : mod 111 r/m
<b>IMUL - Signed Multiply</b>	
AL, AX, or EAX with register	1111 011w : 11 101 reg
AL, AX, or EAX with memory	1111 011w : mod 101 reg
register1 with register2	0000 1111 : 1010 1111 : 11 : reg1 reg2
register with memory	0000 1111 : 1010 1111 : mod reg r/m
register1 with immediate to register2	0110 10s1 : 11 reg1 reg2 : immediate data
memory with immediate to register	0110 10s1 : mod reg r/m : immediate data
<b>IN - Input From Port</b>	
fixed port	1110 010w : port number
variable port	1110 110w
<b>INC - Increment by 1</b>	
reg	1111 111w : 11 000 reg
reg (alternate encoding)	0100 0 reg
memory	1111 111w : mod 000 r/m
<b>INS - Input from DX Port</b>	0110 110w
<b>INT n - Interrupt Type n</b>	1100 1101 : type
<b>INT - Single-Step Interrupt 3</b>	1100 1100
<b>INTO - Interrupt 4 on Overflow</b>	1100 1110
<b>INVD - Invalidate Cache</b>	0000 1111 : 0000 1000
<b>INVLPG - Invalidate TLB Entry</b>	0000 1111 : 0000 0001 : mod 111 r/m
<b>INVPCID - Invalidate Process-Context Identifier</b>	0110 0110:0000 1111:0011 1000:1000 0010: mod reg r/m
<b>IRET/IRETD - Interrupt Return</b>	1100 1111
<b>Jcc - Jump if Condition is Met</b>	
8-bit displacement	0111 tttt : 8-bit displacement
full displacement	0000 1111 : 1000 tttt : full displacement

**Table B-13. General Purpose Instruction Formats and Encodings  
for Non-64-Bit Modes (Contd.)**

Instruction and Format	Encoding
<b>JCXZ/JECXZ - Jump on CX/ECX Zero</b> Address-size prefix differentiates JXCZ and JECXZ	1110 0011 : 8-bit displacement
<b>JMP - Unconditional Jump (to same segment)</b>	
short	1110 1011 : 8-bit displacement
direct	1110 1001 : full displacement
register indirect	1111 1111 : 11 100 reg
memory indirect	1111 1111 : mod 100 r/m
<b>JMP - Unconditional Jump (to other segment)</b>	
direct intersegment	1110 1010 : unsigned full offset, selector
indirect intersegment	1111 1111 : mod 101 r/m
<b>LAHF - Load Flags into AH Register</b>	1001 1111
<b>LAR - Load Access Rights Byte</b>	
from register	0000 1111 : 0000 0010 : 11 reg1 reg2
from memory	0000 1111 : 0000 0010 : mod reg r/m
<b>LDS - Load Pointer to DS</b>	1100 0101 : mod <sup>A,B</sup> reg r/m
<b>LEA - Load Effective Address</b>	1000 1101 : mod <sup>A</sup> reg r/m
<b>LEAVE - High Level Procedure Exit</b>	1100 1001
<b>LES - Load Pointer to ES</b>	1100 0100 : mod <sup>A,B</sup> reg r/m
<b>LFS - Load Pointer to FS</b>	0000 1111 : 1011 0100 : mod <sup>A</sup> reg r/m
<b>LGDT - Load Global Descriptor Table Register</b>	0000 1111 : 0000 0001 : mod <sup>A</sup> 010 r/m
<b>LGS - Load Pointer to GS</b>	0000 1111 : 1011 0101 : mod <sup>A</sup> reg r/m
<b>LIDT - Load Interrupt Descriptor Table Register</b>	0000 1111 : 0000 0001 : mod <sup>A</sup> 011 r/m
<b>LLDT - Load Local Descriptor Table Register</b>	
LDTR from register	0000 1111 : 0000 0000 : 11 010 reg
LDTR from memory	0000 1111 : 0000 0000 : mod 010 r/m
<b>LMSW - Load Machine Status Word</b>	
from register	0000 1111 : 0000 0001 : 11 110 reg
from memory	0000 1111 : 0000 0001 : mod 110 r/m
<b>LOCK - Assert LOCK# Signal Prefix</b>	1111 0000

**Table B-13. General Purpose Instruction Formats and Encodings  
for Non-64-Bit Modes (Contd.)**

Instruction and Format	Encoding
<b>LODS/LODSB/LODSw/LODSD - Load String Operand</b>	1010 110w
<b>LOOP - Loop Count</b>	1110 0010 : 8-bit displacement
<b>LOOPZ/LOOPE - Loop Count while Zero/Equal</b>	1110 0001 : 8-bit displacement
<b>LOOPNZ/LOOPNE - Loop Count while not Zero/Equal</b>	1110 0000 : 8-bit displacement
<b>LSL - Load Segment Limit</b>	
from register	0000 1111 : 0000 0011 : 11 reg1 reg2
from memory	0000 1111 : 0000 0011 : mod reg r/m
<b>LSS - Load Pointer to SS</b>	0000 1111 : 1011 0010 : mod <sup>A</sup> reg r/m
<b>LTR - Load Task Register</b>	
from register	0000 1111 : 0000 0000 : 11 011 reg
from memory	0000 1111 : 0000 0000 : mod 011 r/m
<b>MOV - Move Data</b>	
register1 to register2	1000 100w : 11 reg1 reg2
register2 to register1	1000 101w : 11 reg1 reg2
memory to reg	1000 101w : mod reg r/m
reg to memory	1000 100w : mod reg r/m
immediate to register	1100 011w : 11 000 reg : immediate data
immediate to register (alternate encoding)	1011 w reg : immediate data
immediate to memory	1100 011w : mod 000 r/m : immediate data
memory to AL, AX, or EAX	1010 000w : full displacement
AL, AX, or EAX to memory	1010 001w : full displacement
<b>MOV - Move to/from Control Registers</b>	
CR0 from register	0000 1111 : 0010 0010 : 11 000 reg
CR2 from register	0000 1111 : 0010 0010 : 11 010reg
CR3 from register	0000 1111 : 0010 0010 : 11 011 reg
CR4 from register	0000 1111 : 0010 0010 : 11 100 reg
register from CR0-CR4	0000 1111 : 0010 0000 : 11 eee reg
<b>MOV - Move to/from Debug Registers</b>	

**Table B-13. General Purpose Instruction Formats and Encodings  
for Non-64-Bit Modes (Contd.)**

Instruction and Format	Encoding
DR0-DR3 from register	0000 1111 : 0010 0011 : 11 eee reg
DR4-DR5 from register	0000 1111 : 0010 0011 : 11 eee reg
DR6-DR7 from register	0000 1111 : 0010 0011 : 11 eee reg
register from DR6-DR7	0000 1111 : 0010 0001 : 11 eee reg
register from DR4-DR5	0000 1111 : 0010 0001 : 11 eee reg
register from DR0-DR3	0000 1111 : 0010 0001 : 11 eee reg
<b>MOV - Move to/from Segment Registers</b>	
register to segment register	1000 1110 : 11 sreg3 reg
register to SS	1000 1110 : 11 sreg3 reg
memory to segment reg	1000 1110 : mod sreg3 r/m
memory to SS	1000 1110 : mod sreg3 r/m
segment register to register	1000 1100 : 11 sreg3 reg
segment register to memory	1000 1100 : mod sreg3 r/m
<b>MOVBE - Move data after swapping bytes</b>	
memory to register	0000 1111 : 0011 1000:1111 0000 : mod reg r/m
register to memory	0000 1111 : 0011 1000:1111 0001 : mod reg r/m
<b>MOVS/MOVSb/MOVSW/MOVSd - Move Data from String to String</b>	1010 010w
<b>MOVsx - Move with Sign-Extend</b>	
memory to reg	0000 1111 : 1011 111w : mod reg r/m
<b>MOVzx - Move with Zero-Extend</b>	
register2 to register1	0000 1111 : 1011 011w : 11 reg1 reg2
memory to register	0000 1111 : 1011 011w : mod reg r/m
<b>MUL - Unsigned Multiply</b>	
AL, AX, or EAX with register	1111 011w : 11 100 reg
AL, AX, or EAX with memory	1111 011w : mod 100 r/m
<b>NEG - Two's Complement Negation</b>	
register	1111 011w : 11 011 reg
memory	1111 011w : mod 011 r/m

**Table B-13. General Purpose Instruction Formats and Encodings  
for Non-64-Bit Modes (Contd.)**

Instruction and Format	Encoding
<b>NOP - No Operation</b>	1001 0000
<b>NOP - Multi-byte No Operation<sup>1</sup></b>	
register	0000 1111 0001 1111 : 11 000 reg
memory	0000 1111 0001 1111 : mod 000 r/m
<b>NOT - One's Complement Negation</b>	
register	1111 011w : 11 010 reg
memory	1111 011w : mod 010 r/m
<b>OR - Logical Inclusive OR</b>	
register1 to register2	0000 100w : 11 reg1 reg2
register2 to register1	0000 101w : 11 reg1 reg2
memory to register	0000 101w : mod reg r/m
register to memory	0000 100w : mod reg r/m
immediate to register	1000 00sw : 11 001 reg : immediate data
immediate to AL, AX, or EAX	0000 110w : immediate data
immediate to memory	1000 00sw : mod 001 r/m : immediate data
<b>OUT - Output to Port</b>	
fixed port	1110 011w : port number
variable port	1110 111w
<b>OUTS - Output to DX Port</b>	0110 111w
<b>POP - Pop a Word from the Stack</b>	
register	1000 1111 : 11 000 reg
register (alternate encoding)	0101 1 reg
memory	1000 1111 : mod 000 r/m
<b>POP - Pop a Segment Register from the Stack</b> (Note: CS cannot be sreg2 in this usage.)	
segment register DS, ES	000 sreg2 111
segment register SS	000 sreg2 111
segment register FS, GS	0000 1111: 10 sreg3 001
<b>POPA/POPAD - Pop All General Registers</b>	0110 0001



**Table B-13. General Purpose Instruction Formats and Encodings  
for Non-64-Bit Modes (Contd.)**

Instruction and Format	Encoding
<b>POPF/POPFD – Pop Stack into FLAGS or EFLAGS Register</b>	1001 1101
<b>PUSH – Push Operand onto the Stack</b>	
register	1111 1111 : 11 110 reg
register (alternate encoding)	0101 0 reg
memory	1111 1111 : mod 110 r/m
immediate	0110 10s0 : immediate data
<b>PUSH – Push Segment Register onto the Stack</b>	
segment register CS,DS,ES,SS	000 sreg2 110
segment register FS,GS	0000 1111: 10 sreg3 000
<b>PUSHA/PUSHAD – Push All General Registers</b>	0110 0000
<b>PUSHF/PUSHFD – Push Flags Register onto the Stack</b>	1001 1100
<b>RCL – Rotate thru Carry Left</b>	
register by 1	1101 000w : 11 010 reg
memory by 1	1101 000w : mod 010 r/m
register by CL	1101 001w : 11 010 reg
memory by CL	1101 001w : mod 010 r/m
register by immediate count	1100 000w : 11 010 reg : imm8 data
memory by immediate count	1100 000w : mod 010 r/m : imm8 data
<b>RCR – Rotate thru Carry Right</b>	
register by 1	1101 000w : 11 011 reg
memory by 1	1101 000w : mod 011 r/m
register by CL	1101 001w : 11 011 reg
memory by CL	1101 001w : mod 011 r/m
register by immediate count	1100 000w : 11 011 reg : imm8 data
memory by immediate count	1100 000w : mod 011 r/m : imm8 data
<b>RDMSR – Read from Model-Specific Register</b>	0000 1111 : 0011 0010

**Table B-13. General Purpose Instruction Formats and Encodings  
for Non-64-Bit Modes (Contd.)**

Instruction and Format	Encoding
<b>RDPMC - Read Performance Monitoring Counters</b>	0000 1111 : 0011 0011
<b>RDTSC - Read Time-Stamp Counter</b>	0000 1111 : 0011 0001
<b>RDTSCP - Read Time-Stamp Counter and Processor ID</b>	0000 1111 : 0000 0001: 1111 1001
<b>REP INS - Input String</b>	1111 0011 : 0110 110w
<b>REP LODS - Load String</b>	1111 0011 : 1010 110w
<b>REP MOVS - Move String</b>	1111 0011 : 1010 010w
<b>REP OUTS - Output String</b>	1111 0011 : 0110 111w
<b>REP STOS - Store String</b>	1111 0011 : 1010 101w
<b>REPE CMPS - Compare String</b>	1111 0011 : 1010 011w
<b>REPE SCAS - Scan String</b>	1111 0011 : 1010 111w
<b>REPNE CMPS - Compare String</b>	1111 0010 : 1010 011w
<b>REPNE SCAS - Scan String</b>	1111 0010 : 1010 111w
<b>RET - Return from Procedure (to same segment)</b>	
no argument	1100 0011
adding immediate to SP	1100 0010 : 16-bit displacement
<b>RET - Return from Procedure (to other segment)</b>	
intersegment	1100 1011
adding immediate to SP	1100 1010 : 16-bit displacement
<b>ROL - Rotate Left</b>	
register by 1	1101 000w : 11 000 reg
memory by 1	1101 000w : mod 000 r/m
register by CL	1101 001w : 11 000 reg
memory by CL	1101 001w : mod 000 r/m
register by immediate count	1100 000w : 11 000 reg : imm8 data
memory by immediate count	1100 000w : mod 000 r/m : imm8 data
<b>ROR - Rotate Right</b>	
register by 1	1101 000w : 11 001 reg

**Table B-13. General Purpose Instruction Formats and Encodings  
for Non-64-Bit Modes (Contd.)**

<b>Instruction and Format</b>	<b>Encoding</b>
memory by 1	1101 000w : mod 001 r/m
register by CL	1101 001w : 11 001 reg
memory by CL	1101 001w : mod 001 r/m
register by immediate count	1100 000w : 11 001 reg : imm8 data
memory by immediate count	1100 000w : mod 001 r/m : imm8 data
<b>RSM - Resume from System Management Mode</b>	0000 1111 : 1010 1010
<b>SAHF - Store AH into Flags</b>	1001 1110
<b>SAL - Shift Arithmetic Left</b>	same instruction as SHL
<b>SAR - Shift Arithmetic Right</b>	
register by 1	1101 000w : 11 111 reg
memory by 1	1101 000w : mod 111 r/m
register by CL	1101 001w : 11 111 reg
memory by CL	1101 001w : mod 111 r/m
register by immediate count	1100 000w : 11 111 reg : imm8 data
memory by immediate count	1100 000w : mod 111 r/m : imm8 data
<b>SBB - Integer Subtraction with Borrow</b>	
register1 to register2	0001 100w : 11 reg1 reg2
register2 to register1	0001 101w : 11 reg1 reg2
memory to register	0001 101w : mod reg r/m
register to memory	0001 100w : mod reg r/m
immediate to register	1000 00sw : 11 011 reg : immediate data
immediate to AL, AX, or EAX	0001 110w : immediate data
immediate to memory	1000 00sw : mod 011 r/m : immediate data
<b>SCAS/SCASB/SCASW/SCASD - Scan String</b>	1010 111w
<b>SETcc - Byte Set on Condition</b>	
register	0000 1111 : 1001 ttn : 11 000 reg
memory	0000 1111 : 1001 ttn : mod 000 r/m
<b>SGDT - Store Global Descriptor Table Register</b>	0000 1111 : 0000 0001 : mod <sup>A</sup> 000 r/m

**Table B-13. General Purpose Instruction Formats and Encodings  
for Non-64-Bit Modes (Contd.)**

Instruction and Format	Encoding
<b>SHL - Shift Left</b>	
register by 1	1101 000w : 11 100 reg
memory by 1	1101 000w : mod 100 r/m
register by CL	1101 001w : 11 100 reg
memory by CL	1101 001w : mod 100 r/m
register by immediate count	1100 000w : 11 100 reg : imm8 data
memory by immediate count	1100 000w : mod 100 r/m : imm8 data
<b>SHLD - Double Precision Shift Left</b>	
register by immediate count	0000 1111 : 1010 0100 : 11 reg2 reg1 : imm8
memory by immediate count	0000 1111 : 1010 0100 : mod reg r/m : imm8
register by CL	0000 1111 : 1010 0101 : 11 reg2 reg1
memory by CL	0000 1111 : 1010 0101 : mod reg r/m
<b>SHR - Shift Right</b>	
register by 1	1101 000w : 11 101 reg
memory by 1	1101 000w : mod 101 r/m
register by CL	1101 001w : 11 101 reg
memory by CL	1101 001w : mod 101 r/m
register by immediate count	1100 000w : 11 101 reg : imm8 data
memory by immediate count	1100 000w : mod 101 r/m : imm8 data
<b>SHRD - Double Precision Shift Right</b>	
register by immediate count	0000 1111 : 1010 1100 : 11 reg2 reg1 : imm8
memory by immediate count	0000 1111 : 1010 1100 : mod reg r/m : imm8
register by CL	0000 1111 : 1010 1101 : 11 reg2 reg1
memory by CL	0000 1111 : 1010 1101 : mod reg r/m
<b>SIDT - Store Interrupt Descriptor Table Register</b>	0000 1111 : 0000 0001 : mod <sup>A</sup> 001 r/m
<b>SLDT - Store Local Descriptor Table Register</b>	
to register	0000 1111 : 0000 0000 : 11 000 reg
to memory	0000 1111 : 0000 0000 : mod 000 r/m
<b>SMSW - Store Machine Status Word</b>	

**Table B-13. General Purpose Instruction Formats and Encodings  
for Non-64-Bit Modes (Contd.)**

Instruction and Format	Encoding
to register	0000 1111 : 0000 0001 : 11 100 reg
to memory	0000 1111 : 0000 0001 : mod 100 r/m
<b>STC - Set Carry Flag</b>	1111 1001
<b>STD - Set Direction Flag</b>	1111 1101
<b>STI - Set Interrupt Flag</b>	1111 1011
<b>STOS/STOSB/STOSW/STOSD - Store String Data</b>	1010 101w
<b>STR - Store Task Register</b>	
to register	0000 1111 : 0000 0000 : 11 001 reg
to memory	0000 1111 : 0000 0000 : mod 001 r/m
<b>SUB - Integer Subtraction</b>	
register1 to register2	0010 100w : 11 reg1 reg2
register2 to register1	0010 101w : 11 reg1 reg2
memory to register	0010 101w : mod reg r/m
register to memory	0010 100w : mod reg r/m
immediate to register	1000 00sw : 11 101 reg : immediate data
immediate to AL, AX, or EAX	0010 110w : immediate data
immediate to memory	1000 00sw : mod 101 r/m : immediate data
<b>TEST - Logical Compare</b>	
register1 and register2	1000 010w : 11 reg1 reg2
memory and register	1000 010w : mod reg r/m
immediate and register	1111 011w : 11 000 reg : immediate data
immediate and AL, AX, or EAX	1010 100w : immediate data
immediate and memory	1111 011w : mod 000 r/m : immediate data
<b>UD2 - Undefined instruction</b>	0000 FFFF : 0000 1011
<b>VERR - Verify a Segment for Reading</b>	
register	0000 1111 : 0000 0000 : 11 100 reg
memory	0000 1111 : 0000 0000 : mod 100 r/m
<b>VERW - Verify a Segment for Writing</b>	
register	0000 1111 : 0000 0000 : 11 101 reg

**Table B-13. General Purpose Instruction Formats and Encodings  
for Non-64-Bit Modes (Contd.)**

Instruction and Format	Encoding
memory	0000 1111 : 0000 0000 : mod 101 r/m
<b>WAIT - Wait</b>	1001 1011
<b>WBINVD - Writeback and Invalidate Data Cache</b>	0000 1111 : 0000 1001
<b>WRMSR - Write to Model-Specific Register</b>	0000 1111 : 0011 0000
<b>XADD - Exchange and Add</b>	
register1, register2	0000 1111 : 1100 000w : 11 reg2 reg1
memory, reg	0000 1111 : 1100 000w : mod reg r/m
<b>XCHG - Exchange Register/Memory with Register</b>	
register1 with register2	1000 011w : 11 reg1 reg2
AX or EAX with reg	1001 0 reg
memory with reg	1000 011w : mod reg r/m
<b>XLAT/XLATB - Table Look-up Translation</b>	1101 0111
<b>XOR - Logical Exclusive OR</b>	
register1 to register2	0011 000w : 11 reg1 reg2
register2 to register1	0011 001w : 11 reg1 reg2
memory to register	0011 001w : mod reg r/m
register to memory	0011 000w : mod reg r/m
immediate to register	1000 00sw : 11 110 reg : immediate data
immediate to AL, AX, or EAX	0011 010w : immediate data
immediate to memory	1000 00sw : mod 110 r/m : immediate data
<b>Prefix Bytes</b>	
address size	0110 0111
LOCK	1111 0000
operand size	0110 0110
CS segment override	0010 1110
DS segment override	0011 1110
ES segment override	0010 0110
FS segment override	0110 0100

**Table B-13. General Purpose Instruction Formats and Encodings for Non-64-Bit Modes (Contd.)**

Instruction and Format	Encoding
GS segment override	0110 0101
SS segment override	0011 0110

**NOTES:**

1. The multi-byte NOP instruction does not alter the content of the register and will not issue a memory operation.

**B.2.1 General Purpose Instruction Formats and Encodings for 64-Bit Mode**

Table B-15 shows machine instruction formats and encodings for general purpose instructions in 64-bit mode.

**Table B-14. Special Symbols**

Symbol	Application
S	If the value of REX.W. is 1, it overrides the presence of 66H.
w	The value of bit W. in REX is has no effect.

**Table B-15. General Purpose Instruction Formats and Encodings for 64-Bit Mode**

Instruction and Format	Encoding
<b>ADC - ADD with Carry</b>	
register1 to register2	0100 0ROB : 0001 000w : 11 reg1 reg2
qwordregister1 to qwordregister2	0100 1ROB : 0001 0001 : 11 qwordreg1 qwordreg2
register2 to register1	0100 0ROB : 0001 001w : 11 reg1 reg2
qwordregister1 to qwordregister2	0100 1ROB : 0001 0011 : 11 qwordreg1 qwordreg2
memory to register	0100 0RXB : 0001 001w : mod reg r/m
memory to qwordregister	0100 1RXB : 0001 0011 : mod qwordreg r/m
register to memory	0100 0RXB : 0001 000w : mod reg r/m
qwordregister to memory	0100 1RXB : 0001 0001 : mod qwordreg r/m

**Table B-15. General Purpose Instruction Formats and Encodings  
for 64-Bit Mode (Contd.)**

Instruction and Format	Encoding
immediate to register	0100 000B : 1000 00sw : 11 010 reg : immediate
immediate to qwordregister	0100 100B : 1000 0001 : 11 010 qwordreg : imm32
immediate to qwordregister	0100 1R0B : 1000 0011 : 11 010 qwordreg : imm8
immediate to AL, AX, or EAX	0001 010w : immediate data
immediate to RAX	0100 1000 : 0000 0101 : imm32
immediate to memory	0100 00XB : 1000 00sw : mod 010 r/m : immediate
immediate32 to memory64	0100 10XB : 1000 0001 : mod 010 r/m : imm32
immediate8 to memory64	0100 10XB : 1000 0031 : mod 010 r/m : imm8
<b>ADD – Add</b>	
register1 to register2	0100 0R0B : 0000 000w : 11 reg1 reg2
qwordregister1 to qwordregister2	0100 1R0B 0000 0000 : 11 qwordreg1 qwordreg2
register2 to register1	0100 0R0B : 0000 001w : 11 reg1 reg2
qwordregister1 to qwordregister2	0100 1R0B 0000 0010 : 11 qwordreg1 qwordreg2
memory to register	0100 0RXB : 0000 001w : mod reg r/m
memory64 to qwordregister	0100 1RXB : 0000 0000 : mod qwordreg r/m
register to memory	0100 0RXB : 0000 000w : mod reg r/m
qwordregister to memory64	0100 1RXB : 0000 0011 : mod qwordreg r/m
immediate to register	0100 0000B : 1000 00sw : 11 000 reg : immediate data
immediate32 to qwordregister	0100 100B : 1000 0001 : 11 010 qwordreg : imm
immediate to AL, AX, or EAX	0000 010w : immediate8
immediate to RAX	0100 1000 : 0000 0101 : imm32
immediate to memory	0100 00XB : 1000 00sw : mod 000 r/m : immediate



**Table B-15. General Purpose Instruction Formats and Encodings  
for 64-Bit Mode (Contd.)**

Instruction and Format	Encoding
immediate32 to memory64	0100 10XB : 1000 0001 : mod 010 r/m : imm32
immediate8 to memory64	0100 10XB : 1000 0011 : mod 010 r/m : imm8
<b>AND - Logical AND</b>	
register1 to register2	0100 0ROB 0010 000w : 11 reg1 reg2
qwordregister1 to qwordregister2	0100 1ROB 0010 0001 : 11 qwordreg1 qwordreg2
register2 to register1	0100 0ROB 0010 001w : 11 reg1 reg2
register1 to register2	0100 1ROB 0010 0011 : 11 qwordreg1 qwordreg2
memory to register	0100 0RXB 0010 001w : mod reg r/m
memory64 to qwordregister	0100 1RXB : 0010 0011 : mod qwordreg r/m
register to memory	0100 0RXB : 0010 000w : mod reg r/m
qwordregister to memory64	0100 1RXB : 0010 0001 : mod qwordreg r/m
immediate to register	0100 000B : 1000 00sw : 11 100 reg : immediate
immediate32 to qwordregister	0100 100B 1000 0001 : 11 100 qwordreg : imm32
immediate to AL, AX, or EAX	0010 010w : immediate
immediate32 to RAX	0100 1000 0010 1001 : imm32
immediate to memory	0100 00XB : 1000 00sw : mod 100 r/m : immediate
immediate32 to memory64	0100 10XB : 1000 0001 : mod 100 r/m : immediate32
immediate8 to memory64	0100 10XB : 1000 0011 : mod 100 r/m : imm8
<b>BSF - Bit Scan Forward</b>	
register1, register2	0100 0ROB 0000 1111 : 1011 1100 : 11 reg1 reg2
qwordregister1, qwordregister2	0100 1ROB 0000 1111 : 1011 1100 : 11 qwordreg1 qwordreg2
memory, register	0100 0RXB 0000 1111 : 1011 1100 : mod reg r/m

**Table B-15. General Purpose Instruction Formats and Encodings  
for 64-Bit Mode (Contd.)**

Instruction and Format	Encoding
memory64, qwordregister	0100 1RXB 0000 1111 : 1011 1100 : mod qwordreg r/m
<b>BSR – Bit Scan Reverse</b>	
register1, register2	0100 0ROB 0000 1111 : 1011 1101 : 11 reg1 reg2
qwordregister1, qwordregister2	0100 1ROB 0000 1111 : 1011 1101 : 11 qwordreg1 qwordreg2
memory, register	0100 0RXB 0000 1111 : 1011 1101 : mod reg r/m
memory64, qwordregister	0100 1RXB 0000 1111 : 1011 1101 : mod qwordreg r/m
<b>BSWAP – Byte Swap</b>	0000 1111 : 1100 1 reg
BSWAP – Byte Swap	0100 100B 0000 1111 : 1100 1 qwordreg
<b>BT – Bit Test</b>	
register, immediate	0100 000B 0000 1111 : 1011 1010 : 11 100 reg: imm8
qwordregister, immediate8	0100 100B 1111 : 1011 1010 : 11 100 qwordreg: imm8 data
memory, immediate	0100 00XB 0000 1111 : 1011 1010 : mod 100 r/m : imm8
memory64, immediate8	0100 10XB 0000 1111 : 1011 1010 : mod 100 r/m : imm8 data
register1, register2	0100 0ROB 0000 1111 : 1010 0011 : 11 reg2 reg1
qwordregister1, qwordregister2	0100 1ROB 0000 1111 : 1010 0011 : 11 qwordreg2 qwordreg1
memory, reg	0100 0RXB 0000 1111 : 1010 0011 : mod reg r/m
memory, qwordreg	0100 1RXB 0000 1111 : 1010 0011 : mod qwordreg r/m
<b>BTC – Bit Test and Complement</b>	

**Table B-15. General Purpose Instruction Formats and Encodings  
for 64-Bit Mode (Contd.)**

<b>Instruction and Format</b>	<b>Encoding</b>
register, immediate	0100 000B 0000 1111 : 1011 1010 : 11 111 reg: imm8
qwordregister, immediate8	0100 100B 0000 1111 : 1011 1010 : 11 111 qwordreg: imm8
memory, immediate	0100 00XB 0000 1111 : 1011 1010 : mod 111 r/m : imm8
memory64, immediate8	0100 10XB 0000 1111 : 1011 1010 : mod 111 r/m : imm8
register1, register2	0100 0ROB 0000 1111 : 1011 1011 : 11 reg2 reg1
qwordregister1, qwordregister2	0100 1ROB 0000 1111 : 1011 1011 : 11 qwordreg2 qwordreg1
memory, register	0100 0RXB 0000 1111 : 1011 1011 : mod reg r/m
memory, qwordreg	0100 1RXB 0000 1111 : 1011 1011 : mod qwordreg r/m
<b>BTR - Bit Test and Reset</b>	
register, immediate	0100 000B 0000 1111 : 1011 1010 : 11 110 reg: imm8
qwordregister, immediate8	0100 100B 0000 1111 : 1011 1010 : 11 110 qwordreg: imm8
memory, immediate	0100 00XB 0000 1111 : 1011 1010 : mod 110 r/m : imm8
memory64, immediate8	0100 10XB 0000 1111 : 1011 1010 : mod 110 r/m : imm8
register1, register2	0100 0ROB 0000 1111 : 1011 0011 : 11 reg2 reg1
qwordregister1, qwordregister2	0100 1ROB 0000 1111 : 1011 0011 : 11 qwordreg2 qwordreg1
memory, register	0100 0RXB 0000 1111 : 1011 0011 : mod reg r/m
memory64, qwordreg	0100 1RXB 0000 1111 : 1011 0011 : mod qwordreg r/m

**Table B-15. General Purpose Instruction Formats and Encodings  
for 64-Bit Mode (Contd.)**

Instruction and Format	Encoding
<b>BTS - Bit Test and Set</b>	
register, immediate	0100 000B 0000 1111 : 1011 1010 : 11 101 reg: imm8
qwordregister, immediate8	0100 100B 0000 1111 : 1011 1010 : 11 101 qwordreg: imm8
memory, immediate	0100 00XB 0000 1111 : 1011 1010 : mod 101 r/m : imm8
memory64, immediate8	0100 10XB 0000 1111 : 1011 1010 : mod 101 r/m : imm8
register1, register2	0100 0R0B 0000 1111 : 1010 1011 : 11 reg2 reg1
qwordregister1, qwordregister2	0100 1R0B 0000 1111 : 1010 1011 : 11 qwordreg2 qwordreg1
memory, register	0100 0RXB 0000 1111 : 1010 1011 : mod reg r/m
memory64, qwordreg	0100 1RXB 0000 1111 : 1010 1011 : mod qwordreg r/m
<b>CALL - Call Procedure (in same segment)</b>	
direct	1110 1000 : displacement32
register indirect	0100 WR00 <sup>w</sup> 1111 1111 : 11 010 reg
memory indirect	0100 W0XB <sup>w</sup> 1111 1111 : mod 010 r/m
<b>CALL - Call Procedure (in other segment)</b>	
indirect	1111 1111 : mod 011 r/m
indirect	0100 10XB 0100 1000 1111 1111 : mod 011 r/m
<b>CBW - Convert Byte to Word</b>	1001 1000
<b>CDQ - Convert Doubleword to Qword+</b>	1001 1001
CDQE - RAX, Sign-Extend of EAX	0100 1000 1001 1001
<b>CLC - Clear Carry Flag</b>	1111 1000
<b>CLD - Clear Direction Flag</b>	1111 1100
<b>CLI - Clear Interrupt Flag</b>	1111 1010
<b>CLTS - Clear Task-Switched Flag in CRO</b>	0000 1111 : 0000 0110
<b>CMC - Complement Carry Flag</b>	1111 0101

**Table B-15. General Purpose Instruction Formats and Encodings  
for 64-Bit Mode (Contd.)**

Instruction and Format	Encoding
<b>CMP - Compare Two Operands</b>	
register1 with register2	0100 0ROB 0011 100w : 11 reg1 reg2
qwordregister1 with qwordregister2	0100 1ROB 0011 1001 : 11 qwordreg1 qwordreg2
register2 with register1	0100 0ROB 0011 101w : 11 reg1 reg2
qwordregister2 with qwordregister1	0100 1ROB 0011 101w : 11 qwordreg1 qwordreg2
memory with register	0100 0RXB 0011 100w : mod reg r/m
memory64 with qwordregister	0100 1RXB 0011 1001 : mod qwordreg r/m
register with memory	0100 0RXB 0011 101w : mod reg r/m
qwordregister with memory64	0100 1RXB 0011 101w1 : mod qwordreg r/m
immediate with register	0100 000B 1000 00sw : 11 111 reg : imm
immediate32 with qwordregister	0100 100B 1000 0001 : 11 111 qwordreg : imm64
immediate with AL, AX, or EAX	0011 110w : imm
immediate32 with RAX	0100 1000 0011 1101 : imm32
immediate with memory	0100 00XB 1000 00sw : mod 111 r/m : imm
immediate32 with memory64	0100 1RXB 1000 0001 : mod 111 r/m : imm64
immediate8 with memory64	0100 1RXB 1000 0011 : mod 111 r/m : imm8
<b>CMPS/CMPSB/CMPSW/CMPSD/CMPSQ - Compare String Operands</b>	
compare string operands [ X at DS:(E)SI with Y at ES:(E)DI ]	1010 011w
qword at address RSI with qword at address RDI	0100 1000 1010 0111
<b>CMPXCHG - Compare and Exchange</b>	
register1, register2	0000 1111 : 1011 000w : 11 reg2 reg1
byteregister1, byteregister2	0100 000B 0000 1111 : 1011 0000 : 11 bytereg2 reg1
qwordregister1, qwordregister2	0100 100B 0000 1111 : 1011 0001 : 11 qwordreg2 reg1
memory, register	0000 1111 : 1011 000w : mod reg r/m

**Table B-15. General Purpose Instruction Formats and Encodings  
for 64-Bit Mode (Contd.)**

Instruction and Format	Encoding
memory8, byteregister	0100 00XB 0000 1111 : 1011 0000 : mod bytereg r/m
memory64, qwordregister	0100 10XB 0000 1111 : 1011 0001 : mod qwordreg r/m
<b>CPUID - CPU Identification</b>	0000 1111 : 1010 0010
CQO - Sign-Extend RAX	0100 1000 1001 1001
<b>CWD - Convert Word to Doubleword</b>	1001 1001
<b>CWDE - Convert Word to Doubleword</b>	1001 1000
<b>DEC - Decrement by 1</b>	
register	0100 000B 1111 111w : 11 001 reg
qwordregister	0100 100B 1111 1111 : 11 001 qwordreg
memory	0100 00XB 1111 111w : mod 001 r/m
memory64	0100 10XB 1111 1111 : mod 001 r/m
<b>DIV - Unsigned Divide</b>	
AL, AX, or EAX by register	0100 000B 1111 011w : 11 110 reg
Divide RDX:RAX by qwordregister	0100 100B 1111 0111 : 11 110 qwordreg
AL, AX, or EAX by memory	0100 00XB 1111 011w : mod 110 r/m
Divide RDX:RAX by memory64	0100 10XB 1111 0111 : mod 110 r/m
<b>ENTER - Make Stack Frame for High Level Procedure</b>	1100 1000 : 16-bit displacement : 8-bit level (L)
<b>HLT - Halt</b>	1111 0100
<b>IDIV - Signed Divide</b>	
AL, AX, or EAX by register	0100 000B 1111 011w : 11 111 reg
RDX:RAX by qwordregister	0100 100B 1111 0111 : 11 111 qwordreg
AL, AX, or EAX by memory	0100 00XB 1111 011w : mod 111 r/m
RDX:RAX by memory64	0100 10XB 1111 0111 : mod 111 r/m
<b>IMUL - Signed Multiply</b>	
AL, AX, or EAX with register	0100 000B 1111 011w : 11 101 reg
RDX:RAX <- RAX with qwordregister	0100 100B 1111 0111 : 11 101 qwordreg
AL, AX, or EAX with memory	0100 00XB 1111 011w : mod 101 r/m
RDX:RAX <- RAX with memory64	0100 10XB 1111 0111 : mod 101 r/m

**Table B-15. General Purpose Instruction Formats and Encodings  
for 64-Bit Mode (Contd.)**

Instruction and Format	Encoding
register1 with register2	0000 1111 : 1010 1111 : 11 : reg1 reg2
qwordregister1 <- qwordregister1 with qwordregister2	0100 1ROB 0000 1111 : 1010 1111 : 11 : qwordreg1 qwordreg2
register with memory	0100 ORXB 0000 1111 : 1010 1111 : mod reg r/m
qwordregister <- qwordregister withmemory64	0100 1RXB 0000 1111 : 1010 1111 : mod qwordreg r/m
register1 with immediate to register2	0100 0ROB 0110 10s1 : 11 reg1 reg2 : imm
qwordregister1 <- qwordregister2 with sign-extended immediate8	0100 1ROB 0110 1011 : 11 qwordreg1 qwordreg2 : imm8
qwordregister1 <- qwordregister2 with immediate32	0100 1ROB 0110 1001 : 11 qwordreg1 qwordreg2 : imm32
memory with immediate to register	0100 0RXB 0110 10s1 : mod reg r/m : imm
qwordregister <- memory64 with sign-extended immediate8	0100 1RXB 0110 1011 : mod qwordreg r/m : imm8
qwordregister <- memory64 with immediate32	0100 1RXB 0110 1001 : mod qwordreg r/m : imm32
<b>IN - Input From Port</b>	
fixed port	1110 010w : port number
variable port	1110 110w
<b>INC - Increment by 1</b>	
reg	0100 000B 1111 111w : 11 000 reg
qwordreg	0100 100B 1111 1111 : 11 000 qwordreg
memory	0100 00XB 1111 111w : mod 000 r/m
memory64	0100 10XB 1111 1111 : mod 000 r/m
<b>INS - Input from DX Port</b>	0110 110w
<b>INT n - Interrupt Type n</b>	1100 1101 : type
<b>INT - Single-Step Interrupt 3</b>	1100 1100
<b>INTO - Interrupt 4 on Overflow</b>	1100 1110
<b>INVD - Invalidate Cache</b>	0000 1111 : 0000 1000
<b>INVLPG - Invalidate TLB Entry</b>	0000 1111 : 0000 0001 : mod 111 r/m

**Table B-15. General Purpose Instruction Formats and Encodings  
for 64-Bit Mode (Contd.)**

Instruction and Format	Encoding
<b>INVPID - Invalidate Process-Context Identifier</b>	0110 0110:0000 1111:0011 1000:1000 0010: mod reg r/m
<b>IRETO - Interrupt Return</b>	1100 1111
<b>Jcc - Jump if Condition is Met</b>	
8-bit displacement	0111 tttt : 8-bit displacement
displacements (excluding 16-bit relative offsets)	0000 1111 : 1000 tttt : displacement32
<b>JCXZ/JECXZ - Jump on CX/ECX Zero</b>	
Address-size prefix differentiates JCXZ and JECXZ	1110 0011 : 8-bit displacement
<b>JMP - Unconditional Jump (to same segment)</b>	
short	1110 1011 : 8-bit displacement
direct	1110 1001 : displacement32
register indirect	0100 w00B <sup>w</sup> : 1111 1111 : 11 100 reg
memory indirect	0100 w0XB <sup>w</sup> : 1111 1111 : mod 100 r/m
<b>JMP - Unconditional Jump (to other segment)</b>	
indirect intersegment	0100 00XB : 1111 1111 : mod 101 r/m
64-bit indirect intersegment	0100 10XB : 1111 1111 : mod 101 r/m
<b>LAR - Load Access Rights Byte</b>	
from register	0100 0ROB : 0000 1111 : 0000 0010 : 11 reg1 reg2
from dwordregister to qwordregister, masked by 00FxFF00H	0100 wROB : 0000 1111 : 0000 0010 : 11 qwordreg1 dwordreg2
from memory	0100 0RXB : 0000 1111 : 0000 0010 : mod reg r/m
from memory32 to qwordregister, masked by 00FxFF00H	0100 wRXB 0000 1111 : 0000 0010 : mod r/m
<b>LEA - Load Effective Address</b>	
in wordregister/dwordregister	0100 0RXB : 1000 1101 : mod <sup>A</sup> reg r/m
in qwordregister	0100 1RXB : 1000 1101 : mod <sup>A</sup> qwordreg r/m
<b>LEAVE - High Level Procedure Exit</b>	1100 1001



**Table B-15. General Purpose Instruction Formats and Encodings  
for 64-Bit Mode (Contd.)**

Instruction and Format	Encoding
<b>LFS - Load Pointer to FS</b>	
FS:r16/r32 with far pointer from memory	0100 0RXB : 0000 1111 : 1011 0100 : mod <sup>A</sup> reg r/m
FS:r64 with far pointer from memory	0100 1RXB : 0000 1111 : 1011 0100 : mod <sup>A</sup> qwordreg r/m
<b>LGDT - Load Global Descriptor Table Register</b>	0100 10XB : 0000 1111 : 0000 0001 : mod <sup>A</sup> 010 r/m
<b>LGS - Load Pointer to GS</b>	
GS:r16/r32 with far pointer from memory	0100 0RXB : 0000 1111 : 1011 0101 : mod <sup>A</sup> reg r/m
GS:r64 with far pointer from memory	0100 1RXB : 0000 1111 : 1011 0101 : mod <sup>A</sup> qwordreg r/m
<b>LIDT - Load Interrupt Descriptor Table Register</b>	0100 10XB : 0000 1111 : 0000 0001 : mod <sup>A</sup> 011 r/m
<b>LLDT - Load Local Descriptor Table Register</b>	
LDTR from register	0100 000B : 0000 1111 : 0000 0000 : 11 010 reg
LDTR from memory	0100 00XB : 0000 1111 : 0000 0000 : mod 010 r/m
<b>LMSW - Load Machine Status Word</b>	
from register	0100 000B : 0000 1111 : 0000 0001 : 11 110 reg
from memory	0100 00XB : 0000 1111 : 0000 0001 : mod 110 r/m
<b>LOCK - Assert LOCK# Signal Prefix</b>	1111 0000
<b>LODS/LODSB/LODSW/LODSD/LODSQ - Load String Operand</b>	
at DS:(E)SI to AL/EAX/EAX	1010 110w
at (R)SI to RAX	0100 1000 1010 1101
<b>LOOP - Loop Count</b>	
if count != 0, 8-bit displacement	1110 0010
if count != 0, RIP + 8-bit displacement sign-extended to 64-bits	0100 1000 1110 0010
<b>LOOPE - Loop Count while Zero/Equal</b>	

**Table B-15. General Purpose Instruction Formats and Encodings  
for 64-Bit Mode (Contd.)**

Instruction and Format	Encoding
if count != 0 & ZF = 1, 8-bit displacement	1110 0001
if count != 0 & ZF = 1, RIP + 8-bit displacement sign-extended to 64-bits	0100 1000 1110 0001
<b>LOOPNE/LOOPNZ - Loop Count while not Zero/Equal</b>	
if count != 0 & ZF = 0, 8-bit displacement	1110 0000
if count != 0 & ZF = 0, RIP + 8-bit displacement sign-extended to 64-bits	0100 1000 1110 0000
<b>LSL - Load Segment Limit</b>	
from register	0000 1111 : 0000 0011 : 11 reg1 reg2
from qwordregister	0100 1R00 0000 1111 : 0000 0011 : 11 qwordreg1 reg2
from memory16	0000 1111 : 0000 0011 : mod reg r/m
from memory64	0100 1RXB 0000 1111 : 0000 0011 : mod qwordreg r/m
<b>LSS - Load Pointer to SS</b>	
SS:r16/r32 with far pointer from memory	0100 0RXB : 0000 1111 : 1011 0010 : mod <sup>A</sup> reg r/m
SS:r64 with far pointer from memory	0100 1WXB : 0000 1111 : 1011 0010 : mod <sup>A</sup> qwordreg r/m
<b>LTR - Load Task Register</b>	
from register	0100 0R00 : 0000 1111 : 0000 0000 : 11 011 reg
from memory	0100 00XB : 0000 1111 : 0000 0000 : mod 011 r/m
<b>MOV - Move Data</b>	
register1 to register2	0100 0ROB : 1000 100w : 11 reg1 reg2
qwordregister1 to qwordregister2	0100 1ROB 1000 1001 : 11 qwordreg1 qwordreg2
register2 to register1	0100 0ROB : 1000 101w : 11 reg1 reg2
qwordregister2 to qwordregister1	0100 1ROB 1000 1011 : 11 qwordreg1 qwordreg2
memory to reg	0100 0RXB : 1000 101w : mod reg r/m

**Table B-15. General Purpose Instruction Formats and Encodings  
for 64-Bit Mode (Contd.)**

Instruction and Format	Encoding
memory64 to qwordregister	0100 1RXB 1000 1011 : mod qwordreg r/m
reg to memory	0100 0RXB : 1000 100w : mod reg r/m
qwordregister to memory64	0100 1RXB 1000 1001 : mod qwordreg r/m
immediate to register	0100 000B : 1100 011w : 11 000 reg : imm
immediate32 to qwordregister (zero extend)	0100 100B 1100 0111 : 11 000 qwordreg : imm32
immediate to register (alternate encoding)	0100 000B : 1011 w reg : imm
immediate64 to qwordregister (alternate encoding)	0100 100B 1011 1000 reg : imm64
immediate to memory	0100 00XB : 1100 011w : mod 000 r/m : imm
immediate32 to memory64 (zero extend)	0100 10XB 1100 0111 : mod 000 r/m : imm32
memory to AL, AX, or EAX	0100 0000 : 1010 000w : displacement
memory64 to RAX	0100 1000 1010 0001 : displacement64
AL, AX, or EAX to memory	0100 0000 : 1010 001w : displacement
RAX to memory64	0100 1000 1010 0011 : displacement64
<b>MOV - Move to/from Control Registers</b>	
CR0-CR4 from register	0100 0ROB : 0000 1111 : 0010 0010 : 11 eee reg (eee = CR#)
CRx from qwordregister	0100 1ROB : 0000 1111 : 0010 0010 : 11 eee qwordreg (Reee = CR#)
register from CR0-CR4	0100 0ROB : 0000 1111 : 0010 0000 : 11 eee reg (eee = CR#)
qwordregister from CRx	0100 1ROB 0000 1111 : 0010 0000 : 11 eee qwordreg (Reee = CR#)
<b>MOV - Move to/from Debug Registers</b>	
DR0-DR7 from register	0000 1111 : 0010 0011 : 11 eee reg (eee = DR#)
DR0-DR7 from quadregister	0100 100B 0000 1111 : 0010 0011 : 11 eee reg (eee = DR#)
register from DR0-DR7	0000 1111 : 0010 0001 : 11 eee reg (eee = DR#)
quadregister from DR0-DR7	0100 100B 0000 1111 : 0010 0001 : 11 eee quadreg (eee = DR#)

**Table B-15. General Purpose Instruction Formats and Encodings  
for 64-Bit Mode (Contd.)**

Instruction and Format	Encoding
<b>MOV - Move to/from Segment Registers</b>	
register to segment register	0100 w00B <sup>w</sup> : 1000 1110 : 11 sreg reg
register to SS	0100 000B : 1000 1110 : 11 sreg reg
memory to segment register	0100 00XB : 1000 1110 : mod sreg r/m
memory64 to segment register (lower 16 bits)	0100 10XB 1000 1110 : mod sreg r/m
memory to SS	0100 00XB : 1000 1110 : mod sreg r/m
segment register to register	0100 000B : 1000 1100 : 11 sreg reg
segment register to qwordregister (zero extended)	0100 100B 1000 1100 : 11 sreg qwordreg
segment register to memory	0100 00XB : 1000 1100 : mod sreg r/m
segment register to memory64 (zero extended)	0100 10XB 1000 1100 : mod sreg3 r/m
<b>MOVBE - Move data after swapping bytes</b>	
memory to register	0100 0RXB : 0000 1111 : 0011 1000:1111 0000 : mod reg r/m
memory64 to qwordregister	0100 1RXB : 0000 1111 : 0011 1000:1111 0000 : mod reg r/m
register to memory	0100 0RXB : 0000 1111 : 0011 1000:1111 0001 : mod reg r/m
qwordregister to memory64	0100 1RXB : 0000 1111 : 0011 1000:1111 0001 : mod reg r/m
<b>MOVS/MOVSb/MOVSsw/MOVSD/MOVSQ - Move Data from String to String</b>	
Move data from string to string	1010 010w
Move data from string to string (qword)	0100 1000 1010 0101
<b>MOVSX/MOVSD - Move with Sign-Extend</b>	
register2 to register1	0100 0ROB : 0000 1111 : 1011 111w : 11 reg1 reg2
byteregister2 to qwordregister1 (sign-extend)	0100 1ROB 0000 1111 : 1011 1110 : 11 quadreg1 bytereg2
wordregister2 to qwordregister1	0100 1ROB 0000 1111 : 1011 1111 : 11 quadreg1 wordreg2

**Table B-15. General Purpose Instruction Formats and Encodings  
for 64-Bit Mode (Contd.)**

Instruction and Format	Encoding
dwordregister2 to qwordregister1	0100 1ROB 0110 0011 : 11 quadreg1 dwordreg2
memory to register	0100 0RXB : 0000 1111 : 1011 111w : mod reg r/m
memory8 to qwordregister (sign-extend)	0100 1RXB 0000 1111 : 1011 1110 : mod qwordreg r/m
memory16 to qwordregister	0100 1RXB 0000 1111 : 1011 1111 : mod qwordreg r/m
memory32 to qwordregister	0100 1RXB 0110 0011 : mod qwordreg r/m
<b>MOVZX - Move with Zero-Extend</b>	
register2 to register1	0100 0ROB : 0000 1111 : 1011 011w : 11 reg1 reg2
dwordregister2 to qwordregister1	0100 1ROB 0000 1111 : 1011 0111 : 11 qwordreg1 dwordreg2
memory to register	0100 0RXB : 0000 1111 : 1011 011w : mod reg r/m
memory32 to qwordregister	0100 1RXB 0000 1111 : 1011 0111 : mod qwordreg r/m
<b>MUL - Unsigned Multiply</b>	
AL, AX, or EAX with register	0100 000B : 1111 011w : 11 100 reg
RAX with qwordregister (to RDX:RAX)	0100 100B 1111 0111 : 11 100 qwordreg
AL, AX, or EAX with memory	0100 00XB 1111 011w : mod 100 r/m
RAX with memory64 (to RDX:RAX)	0100 10XB 1111 0111 : mod 100 r/m
<b>NEG - Two's Complement Negation</b>	
register	0100 000B : 1111 011w : 11 011 reg
qwordregister	0100 100B 1111 0111 : 11 011 qwordreg
memory	0100 00XB : 1111 011w : mod 011 r/m
memory64	0100 10XB 1111 0111 : mod 011 r/m
<b>NOP - No Operation</b>	
1001 0000	
<b>NOT - One's Complement Negation</b>	
register	0100 000B : 1111 011w : 11 010 reg
qwordregister	0100 000B 1111 0111 : 11 010 qwordreg

**Table B-15. General Purpose Instruction Formats and Encodings  
for 64-Bit Mode (Contd.)**

Instruction and Format	Encoding
memory	0100 00XB : 1111 011w : mod 010 r/m
memory64	0100 1RXB 1111 0111 : mod 010 r/m
<b>OR - Logical Inclusive OR</b>	
register1 to register2	0000 100w : 11 reg1 reg2
byteregister1 to byteregister2	0100 0ROB 0000 1000 : 11 bytereg1 bytereg2
qwordregister1 to qwordregister2	0100 1ROB 0000 1001 : 11 qwordreg1 qwordreg2
register2 to register1	0000 101w : 11 reg1 reg2
byteregister2 to byteregister1	0100 0ROB 0000 1010 : 11 bytereg1 bytereg2
qwordregister2 to qwordregister1	0100 0ROB 0000 1011 : 11 qwordreg1 qwordreg2
memory to register	0000 101w : mod reg r/m
memory8 to byteregister	0100 0RXB 0000 1010 : mod bytereg r/m
memory8 to qwordregister	0100 0RXB 0000 1011 : mod qwordreg r/m
register to memory	0000 100w : mod reg r/m
byteregister to memory8	0100 0RXB 0000 1000 : mod bytereg r/m
qwordregister to memory64	0100 1RXB 0000 1001 : mod qwordreg r/m
immediate to register	1000 00sw : 11 001 reg : imm
immediate8 to byteregister	0100 000B 1000 0000 : 11 001 bytereg : imm8
immediate32 to qwordregister	0100 000B 1000 0001 : 11 001 qwordreg : imm32
immediate8 to qwordregister	0100 000B 1000 0011 : 11 001 qwordreg : imm8
immediate to AL, AX, or EAX	0000 110w : imm
immediate64 to RAX	0100 1000 0000 1101 : imm64
immediate to memory	1000 00sw : mod 001 r/m : imm
immediate8 to memory8	0100 00XB 1000 0000 : mod 001 r/m : imm8
immediate32 to memory64	0100 00XB 1000 0001 : mod 001 r/m : imm32
immediate8 to memory64	0100 00XB 1000 0011 : mod 001 r/m : imm8

**Table B-15. General Purpose Instruction Formats and Encodings  
for 64-Bit Mode (Contd.)**

Instruction and Format	Encoding
<b>OUT - Output to Port</b>	
fixed port	1110 011w : port number
variable port	1110 111w
<b>OUTS - Output to DX Port</b>	
output to DX Port	0110 111w
<b>POP - Pop a Value from the Stack</b>	
wordregister	0101 0101 : 0100 000B : 1000 1111 : 11 000 reg16
qwordregister	0100 W00B <sup>S</sup> : 1000 1111 : 11 000 reg64
wordregister (alternate encoding)	0101 0101 : 0100 000B : 0101 1 reg16
qwordregister (alternate encoding)	0100 W00B : 0101 1 reg64
memory64	0100 W0XB <sup>S</sup> : 1000 1111 : mod 000 r/m
memory16	0101 0101 : 0100 00XB 1000 1111 : mod 000 r/m
<b>POP - Pop a Segment Register from the Stack</b> (Note: CS cannot be sreg2 in this usage.)	
segment register FS, GS	0000 1111 : 10 sreg3 001
<b>POPF/POPFQ - Pop Stack into FLAGS/RFLAGS Register</b>	
pop stack to FLAGS register	0101 0101 : 1001 1101
pop Stack to RFLAGS register	0100 1000 1001 1101
<b>PUSH - Push Operand onto the Stack</b>	
wordregister	0101 0101 : 0100 000B : 1111 1111 : 11 110 reg16
qwordregister	0100 W00B <sup>S</sup> : 1111 1111 : 11 110 reg64
wordregister (alternate encoding)	0101 0101 : 0100 000B : 0101 0 reg16
qwordregister (alternate encoding)	0100 W00B <sup>S</sup> : 0101 0 reg64
memory16	0101 0101 : 0100 000B : 1111 1111 : mod 110 r/m
memory64	0100 W00B <sup>S</sup> : 1111 1111 : mod 110 r/m
immediate8	0110 1010 : imm8
immediate16	0101 0101 : 0110 1000 : imm16

**Table B-15. General Purpose Instruction Formats and Encodings  
for 64-Bit Mode (Contd.)**

Instruction and Format	Encoding
immediate64	0110 1000 : imm64
<b>PUSH - Push Segment Register onto the Stack</b>	
segment register FS,GS	0000 1111: 10 sreg3 000
<b>PUSHF/PUSHFD - Push Flags Register onto the Stack</b>	1001 1100
<b>RCL - Rotate thru Carry Left</b>	
register by 1	0100 000B : 1101 000w : 11 010 reg
qwordregister by 1	0100 100B 1101 0001 : 11 010 qwordreg
memory by 1	0100 00XB : 1101 000w : mod 010 r/m
memory64 by 1	0100 10XB 1101 0001 : mod 010 r/m
register by CL	0100 000B : 1101 001w : 11 010 reg
qwordregister by CL	0100 100B 1101 0011 : 11 010 qwordreg
memory by CL	0100 00XB : 1101 001w : mod 010 r/m
memory64 by CL	0100 10XB 1101 0011 : mod 010 r/m
register by immediate count	0100 000B : 1100 000w : 11 010 reg : imm
qwordregister by immediate count	0100 100B 1100 0001 : 11 010 qwordreg : imm8
memory by immediate count	0100 00XB : 1100 000w : mod 010 r/m : imm
memory64 by immediate count	0100 10XB 1100 0001 : mod 010 r/m : imm8
<b>RCR - Rotate thru Carry Right</b>	
register by 1	0100 000B : 1101 000w : 11 011 reg
qwordregister by 1	0100 100B 1101 0001 : 11 011 qwordreg
memory by 1	0100 00XB : 1101 000w : mod 011 r/m
memory64 by 1	0100 10XB 1101 0001 : mod 011 r/m
register by CL	0100 000B : 1101 001w : 11 011 reg
qwordregister by CL	0100 000B 1101 0010 : 11 011 qwordreg
memory by CL	0100 00XB : 1101 001w : mod 011 r/m
memory64 by CL	0100 10XB 1101 0011 : mod 011 r/m
register by immediate count	0100 000B : 1100 000w : 11 011 reg : imm8



**Table B-15. General Purpose Instruction Formats and Encodings  
for 64-Bit Mode (Contd.)**

Instruction and Format	Encoding
qword register by immediate count	0100 100B 1100 0001 : 11 011 qwordreg : imm8
memory by immediate count	0100 00XB : 1100 000w : mod 011 r/m : imm8
memory64 by immediate count	0100 10XB 1100 0001 : mod 011 r/m : imm8
<b>RDMSR - Read from Model-Specific Register</b>	
load ECX-specified register into EDX:EAX	0000 1111 : 0011 0010
<b>RDPMC - Read Performance Monitoring Counters</b>	
load ECX-specified performance counter into EDX:EAX	0000 1111 : 0011 0011
<b>RDTSC - Read Time-Stamp Counter</b>	
read time-stamp counter into EDX:EAX	0000 1111 : 0011 0001
<b>RDTSCP - Read Time-Stamp Counter and Processor ID</b>	0000 1111 : 0000 0001 : 1111 1001
<b>REP INS - Input String</b>	
<b>REP LODS - Load String</b>	
<b>REP MOVS - Move String</b>	
<b>REP OUTS - Output String</b>	
<b>REP STOS - Store String</b>	
<b>REPE CMPS - Compare String</b>	
<b>REPE SCAS - Scan String</b>	
<b>REPNE CMPS - Compare String</b>	
<b>REPNE SCAS - Scan String</b>	
<b>RET - Return from Procedure (to same segment)</b>	
no argument	1100 0011
adding immediate to SP	1100 0010 : 16-bit displacement
<b>RET - Return from Procedure (to other segment)</b>	
intersegment	1100 1011
adding immediate to SP	1100 1010 : 16-bit displacement

**Table B-15. General Purpose Instruction Formats and Encodings  
for 64-Bit Mode (Contd.)**

Instruction and Format	Encoding
<b>ROL - Rotate Left</b>	
register by 1	0100 000B 1101 000w : 11 000 reg
byteregister by 1	0100 000B 1101 0000 : 11 000 bytereg
qwordregister by 1	0100 100B 1101 0001 : 11 000 qwordreg
memory by 1	0100 00XB 1101 000w : mod 000 r/m
memory8 by 1	0100 00XB 1101 0000 : mod 000 r/m
memory64 by 1	0100 10XB 1101 0001 : mod 000 r/m
register by CL	0100 000B 1101 001w : 11 000 reg
byteregister by CL	0100 000B 1101 0010 : 11 000 bytereg
qwordregister by CL	0100 100B 1101 0011 : 11 000 qwordreg
memory by CL	0100 00XB 1101 001w : mod 000 r/m
memory8 by CL	0100 00XB 1101 0010 : mod 000 r/m
memory64 by CL	0100 10XB 1101 0011 : mod 000 r/m
register by immediate count	1100 000w : 11 000 reg : imm8
byteregister by immediate count	0100 000B 1100 0000 : 11 000 bytereg : imm8
qwordregister by immediate count	0100 100B 1100 0001 : 11 000 bytereg : imm8
memory by immediate count	1100 000w : mod 000 r/m : imm8
memory8 by immediate count	0100 00XB 1100 0000 : mod 000 r/m : imm8
memory64 by immediate count	0100 10XB 1100 0001 : mod 000 r/m : imm8
<b>ROR - Rotate Right</b>	
register by 1	0100 000B 1101 000w : 11 001 reg
byteregister by 1	0100 000B 1101 0000 : 11 001 bytereg
qwordregister by 1	0100 100B 1101 0001 : 11 001 qwordreg
memory by 1	0100 00XB 1101 000w : mod 001 r/m
memory8 by 1	0100 00XB 1101 0000 : mod 001 r/m
memory64 by 1	0100 10XB 1101 0001 : mod 001 r/m
register by CL	0100 000B 1101 001w : 11 001 reg

**Table B-15. General Purpose Instruction Formats and Encodings  
for 64-Bit Mode (Contd.)**

<b>Instruction and Format</b>	<b>Encoding</b>
byte register by CL	0100 000B 1101 0010 : 11 001 bytereg
qword register by CL	0100 100B 1101 0011 : 11 001 qwordreg
memory by CL	0100 00XB 1101 001w : mod 001 r/m
memory8 by CL	0100 00XB 1101 0010 : mod 001 r/m
memory64 by CL	0100 10XB 1101 0011 : mod 001 r/m
register by immediate count	0100 000B 1100 000w : 11 001 reg : imm8
byte register by immediate count	0100 000B 1100 0000 : 11 001 reg : imm8
qword register by immediate count	0100 100B 1100 0001 : 11 001 qwordreg : imm8
memory by immediate count	0100 00XB 1100 000w : mod 001 r/m : imm8
memory8 by immediate count	0100 00XB 1100 0000 : mod 001 r/m : imm8
memory64 by immediate count	0100 10XB 1100 0001 : mod 001 r/m : imm8
<b>RSM - Resume from System Management Mode</b>	0000 1111 : 1010 1010
<b>SAL - Shift Arithmetic Left</b>	same instruction as SHL
<b>SAR - Shift Arithmetic Right</b>	
register by 1	0100 000B 1101 000w : 11 111 reg
byte register by 1	0100 000B 1101 0000 : 11 111 bytereg
qword register by 1	0100 100B 1101 0001 : 11 111 qwordreg
memory by 1	0100 00XB 1101 000w : mod 111 r/m
memory8 by 1	0100 00XB 1101 0000 : mod 111 r/m
memory64 by 1	0100 10XB 1101 0001 : mod 111 r/m
register by CL	0100 000B 1101 001w : 11 111 reg
byte register by CL	0100 000B 1101 0010 : 11 111 bytereg
qword register by CL	0100 100B 1101 0011 : 11 111 qwordreg
memory by CL	0100 00XB 1101 001w : mod 111 r/m
memory8 by CL	0100 00XB 1101 0010 : mod 111 r/m
memory64 by CL	0100 10XB 1101 0011 : mod 111 r/m
register by immediate count	0100 000B 1100 000w : 11 111 reg : imm8
byte register by immediate count	0100 000B 1100 0000 : 11 111 bytereg : imm8

**Table B-15. General Purpose Instruction Formats and Encodings  
for 64-Bit Mode (Contd.)**

Instruction and Format	Encoding
qwordregister by immediate count	0100 100B 1100 0001 : 11 111 qwordreg : imm8
memory by immediate count	0100 00XB 1100 000w : mod 111 r/m : imm8
memory8 by immediate count	0100 00XB 1100 0000 : mod 111 r/m : imm8
memory64 by immediate count	0100 10XB 1100 0001 : mod 111 r/m : imm8
<b>SBB - Integer Subtraction with Borrow</b>	
register1 to register2	0100 0R0B 0001 100w : 11 reg1 reg2
byteregister1 to byteregister2	0100 0R0B 0001 1000 : 11 bytereg1 bytereg2
quadregister1 to quadregister2	0100 1R0B 0001 1001 : 11 quadreg1 quadreg2
register2 to register1	0100 0R0B 0001 101w : 11 reg1 reg2
byteregister2 to byteregister1	0100 0R0B 0001 1010 : 11 reg1 bytereg2
byteregister2 to byteregister1	0100 1R0B 0001 1011 : 11 reg1 bytereg2
memory to register	0100 0RXB 0001 101w : mod reg r/m
memory8 to byteregister	0100 0RXB 0001 1010 : mod bytereg r/m
memory64 to byteregister	0100 1RXB 0001 1011 : mod quadreg r/m
register to memory	0100 0RXB 0001 100w : mod reg r/m
byteregister to memory8	0100 0RXB 0001 1000 : mod reg r/m
quadregister to memory64	0100 1RXB 0001 1001 : mod reg r/m
immediate to register	0100 000B 1000 00sw : 11 011 reg : imm
immediate8 to byteregister	0100 000B 1000 0000 : 11 011 bytereg : imm8
immediate32 to qwordregister	0100 100B 1000 0001 : 11 011 qwordreg : imm32
immediate8 to qwordregister	0100 100B 1000 0011 : 11 011 qwordreg : imm8
immediate to AL, AX, or EAX	0100 000B 0001 110w : imm
immediate32 to RAL	0100 1000 0001 1101 : imm32
immediate to memory	0100 00XB 1000 00sw : mod 011 r/m : imm
immediate8 to memory8	0100 00XB 1000 0000 : mod 011 r/m : imm8
immediate32 to memory64	0100 10XB 1000 0001 : mod 011 r/m : imm32

**Table B-15. General Purpose Instruction Formats and Encodings  
for 64-Bit Mode (Contd.)**

Instruction and Format	Encoding
immediate8 to memory64	0100 10XB 1000 0011 : mod 011 r/m : imm8
<b>SCAS/SCASB/SCASW/SCASD - Scan String</b>	
scan string	1010 111w
scan string (compare AL with byte at RDI)	0100 1000 1010 1110
scan string (compare RAX with qword at RDI)	0100 1000 1010 1111
<b>SETcc - Byte Set on Condition</b>	
register	0100 000B 0000 1111 : 1001 ttn : 11 000 reg
register	0100 0000 0000 1111 : 1001 ttn : 11 000 reg
memory	0100 00XB 0000 1111 : 1001 ttn : mod 000 r/m
memory	0100 0000 0000 1111 : 1001 ttn : mod 000 r/m
<b>SGDT - Store Global Descriptor Table Register</b>	0000 1111 : 0000 0001 : mod <sup>A</sup> 000 r/m
<b>SHL - Shift Left</b>	
register by 1	0100 000B 1101 000w : 11 100 reg
byteregister by 1	0100 000B 1101 0000 : 11 100 bytereg
qwordregister by 1	0100 100B 1101 0001 : 11 100 qwordreg
memory by 1	0100 00XB 1101 000w : mod 100 r/m
memory8 by 1	0100 00XB 1101 0000 : mod 100 r/m
memory64 by 1	0100 10XB 1101 0001 : mod 100 r/m
register by CL	0100 000B 1101 001w : 11 100 reg
byteregister by CL	0100 000B 1101 0010 : 11 100 bytereg
qwordregister by CL	0100 100B 1101 0011 : 11 100 qwordreg
memory by CL	0100 00XB 1101 001w : mod 100 r/m
memory8 by CL	0100 00XB 1101 0010 : mod 100 r/m
memory64 by CL	0100 10XB 1101 0011 : mod 100 r/m
register by immediate count	0100 000B 1100 000w : 11 100 reg : imm8
byteregister by immediate count	0100 000B 1100 0000 : 11 100 bytereg : imm8

**Table B-15. General Purpose Instruction Formats and Encodings  
for 64-Bit Mode (Contd.)**

Instruction and Format	Encoding
quadregister by immediate count	0100 100B 1100 0001 : 11 100 quadreg : imm8
memory by immediate count	0100 00XB 1100 000w : mod 100 r/m : imm8
memory8 by immediate count	0100 00XB 1100 0000 : mod 100 r/m : imm8
memory64 by immediate count	0100 10XB 1100 0001 : mod 100 r/m : imm8
<b>SHLD - Double Precision Shift Left</b>	
register by immediate count	0100 0R0B 0000 1111 : 1010 0100 : 11 reg2 reg1 : imm8
qwordregister by immediate8	0100 1R0B 0000 1111 : 1010 0100 : 11 qworddreg2 qwordreg1 : imm8
memory by immediate count	0100 0RXB 0000 1111 : 1010 0100 : mod reg r/m : imm8
memory64 by immediate8	0100 1RXB 0000 1111 : 1010 0100 : mod qwordreg r/m : imm8
register by CL	0100 0R0B 0000 1111 : 1010 0101 : 11 reg2 reg1
quadregister by CL	0100 1R0B 0000 1111 : 1010 0101 : 11 quadreg2 quadreg1
memory by CL	0100 00XB 0000 1111 : 1010 0101 : mod reg r/m
memory64 by CL	0100 1RXB 0000 1111 : 1010 0101 : mod quadreg r/m
<b>SHR - Shift Right</b>	
register by 1	0100 000B 1101 000w : 11 101 reg
byteregister by 1	0100 000B 1101 0000 : 11 101 bytereg
qwordregister by 1	0100 100B 1101 0001 : 11 101 qwordreg
memory by 1	0100 00XB 1101 000w : mod 101 r/m
memory8 by 1	0100 00XB 1101 0000 : mod 101 r/m
memory64 by 1	0100 10XB 1101 0001 : mod 101 r/m
register by CL	0100 000B 1101 001w : 11 101 reg
byteregister by CL	0100 000B 1101 0010 : 11 101 bytereg

**Table B-15. General Purpose Instruction Formats and Encodings  
for 64-Bit Mode (Contd.)**

Instruction and Format	Encoding
qwordregister by CL	0100 100B 1101 0011 : 11 101 qwordreg
memory by CL	0100 00XB 1101 001w : mod 101 r/m
memory8 by CL	0100 00XB 1101 0010 : mod 101 r/m
memory64 by CL	0100 10XB 1101 0011 : mod 101 r/m
register by immediate count	0100 000B 1100 000w : 11 101 reg : imm8
byteregister by immediate count	0100 000B 1100 0000 : 11 101 reg : imm8
qwordregister by immediate count	0100 100B 1100 0001 : 11 101 reg : imm8
memory by immediate count	0100 00XB 1100 000w : mod 101 r/m : imm8
memory8 by immediate count	0100 00XB 1100 0000 : mod 101 r/m : imm8
memory64 by immediate count	0100 10XB 1100 0001 : mod 101 r/m : imm8
<b>SHRD - Double Precision Shift Right</b>	
register by immediate count	0100 0ROB 0000 1111 : 1010 1100 : 11 reg2 reg1 : imm8
qwordregister by immediate8	0100 1ROB 0000 1111 : 1010 1100 : 11 qwordreg2 qwordreg1 : imm8
memory by immediate count	0100 00XB 0000 1111 : 1010 1100 : mod reg r/m : imm8
memory64 by immediate8	0100 1RXB 0000 1111 : 1010 1100 : mod qwordreg r/m : imm8
register by CL	0100 000B 0000 1111 : 1010 1101 : 11 reg2 reg1
qwordregister by CL	0100 1ROB 0000 1111 : 1010 1101 : 11 qwordreg2 qwordreg1
memory by CL	0000 1111 : 1010 1101 : mod reg r/m
memory64 by CL	0100 1RXB 0000 1111 : 1010 1101 : mod qwordreg r/m
<b>SIDT - Store Interrupt Descriptor Table Register</b>	0000 1111 : 0000 0001 : mod <sup>A</sup> 001 r/m
<b>SLDT - Store Local Descriptor Table Register</b>	
to register	0100 000B 0000 1111 : 0000 0000 : 11 000 reg

**Table B-15. General Purpose Instruction Formats and Encodings  
for 64-Bit Mode (Contd.)**

Instruction and Format	Encoding
to memory	0100 00XB 0000 1111 : 0000 0000 : mod 000 r/m
<b>SMSW - Store Machine Status Word</b>	
to register	0100 000B 0000 1111 : 0000 0001 : 11 100 reg
to memory	0100 00XB 0000 1111 : 0000 0001 : mod 100 r/m
<b>STC - Set Carry Flag</b>	1111 1001
<b>STD - Set Direction Flag</b>	1111 1101
<b>STI - Set Interrupt Flag</b>	1111 1011
<b>STOS/STOSB/STOSW/STOSD/STOSQ - Store String Data</b>	
store string data	1010 101w
store string data (RAX at address RDI)	0100 1000 1010 1011
<b>STR - Store Task Register</b>	
to register	0100 000B 0000 1111 : 0000 0000 : 11 001 reg
to memory	0100 00XB 0000 1111 : 0000 0000 : mod 001 r/m
<b>SUB - Integer Subtraction</b>	
register1 from register2	0100 0ROB 0010 100w : 11 reg1 reg2
byteregister1 from byteregister2	0100 0ROB 0010 1000 : 11 bytereg1 bytereg2
qwordregister1 from qwordregister2	0100 1ROB 0010 1000 : 11 qwordreg1 qwordreg2
register2 from register1	0100 0ROB 0010 101w : 11 reg1 reg2
byteregister2 from byteregister1	0100 0ROB 0010 1010 : 11 bytereg1 bytereg2
qwordregister2 from qwordregister1	0100 1ROB 0010 1011 : 11 qwordreg1 qwordreg2
memory from register	0100 00XB 0010 101w : mod reg r/m



**Table B-15. General Purpose Instruction Formats and Encodings  
for 64-Bit Mode (Contd.)**

<b>Instruction and Format</b>	<b>Encoding</b>
memory8 from byteregister	0100 0RXB 0010 1010 : mod bytereg r/m
memory64 from qwordregister	0100 1RXB 0010 1011 : mod qwordreg r/m
register from memory	0100 0RXB 0010 100w : mod reg r/m
byteregister from memory8	0100 0RXB 0010 1000 : mod bytereg r/m
qwordregister from memory8	0100 1RXB 0010 1000 : mod qwordreg r/m
immediate from register	0100 000B 1000 00sw : 11 101 reg : imm
immediate8 from byteregister	0100 000B 1000 0000 : 11 101 bytereg : imm8
immediate32 from qwordregister	0100 100B 1000 0001 : 11 101 qwordreg : imm32
immediate8 from qwordregister	0100 100B 1000 0011 : 11 101 qwordreg : imm8
immediate from AL, AX, or EAX	0100 000B 0010 110w : imm
immediate32 from RAX	0100 1000 0010 1101 : imm32
immediate from memory	0100 00XB 1000 00sw : mod 101 r/m : imm
immediate8 from memory8	0100 00XB 1000 0000 : mod 101 r/m : imm8
immediate32 from memory64	0100 10XB 1000 0001 : mod 101 r/m : imm32
immediate8 from memory64	0100 10XB 1000 0011 : mod 101 r/m : imm8
<b>SWAPGS - Swap GS Base Register</b>	
GS base register value for value in MSR C0000102H	0000 1111 0000 0001 [this one incomplete]
<b>SYSCALL - Fast System Call</b>	
fast call to privilege level 0 system procedures	0000 1111 0000 0101
<b>SYSRET - Return From Fast System Call</b>	
return from fast system call	0000 1111 0000 0111
<b>TEST - Logical Compare</b>	
register1 and register2	0100 0ROB 1000 010w : 11 reg1 reg2
byteregister1 and byteregister2	0100 0ROB 1000 0100 : 11 bytereg1 bytereg2

**Table B-15. General Purpose Instruction Formats and Encodings  
for 64-Bit Mode (Contd.)**

<b>Instruction and Format</b>	<b>Encoding</b>
qwordregister1 and qwordregister2	0100 1ROB 1000 0101 : 11 qwordreg1 qwordreg2
memory and register	0100 0ROB 1000 010w : mod reg r/m
memory8 and byteregister	0100 0RXB 1000 0100 : mod bytereg r/m
memory64 and qwordregister	0100 1RXB 1000 0101 : mod qwordreg r/m
immediate and register	0100 000B 1111 011w : 11 000 reg : imm
immediate8 and byteregister	0100 000B 1111 0110 : 11 000 bytereg : imm8
immediate32 and qwordregister	0100 100B 1111 0111 : 11 000 bytereg : imm8
immediate and AL, AX, or EAX	0100 000B 1010 100w : imm
immediate32 and RAX	0100 1000 1010 1001 : imm32
immediate and memory	0100 00XB 1111 011w : mod 000 r/m : imm
immediate8 and memory8	0100 1000 1111 0110 : mod 000 r/m : imm8
immediate32 and memory64	0100 1000 1111 0111 : mod 000 r/m : imm32
<b>UD2 - Undefined instruction</b>	0000 FFFF : 0000 1011
<b>VERR - Verify a Segment for Reading</b>	
register	0100 000B 0000 1111 : 0000 0000 : 11 100 reg
memory	0100 00XB 0000 1111 : 0000 0000 : mod 100 r/m
<b>VERW - Verify a Segment for Writing</b>	
register	0100 000B 0000 1111 : 0000 0000 : 11 101 reg
memory	0100 00XB 0000 1111 : 0000 0000 : mod 101 r/m
<b>WAIT - Wait</b>	1001 1011
<b>WBINVD - Writeback and Invalidate Data Cache</b>	0000 1111 : 0000 1001
<b>WRMSR - Write to Model-Specific Register</b>	
write EDX:EAX to ECX specified MSR	0000 1111 : 0011 0000

**Table B-15. General Purpose Instruction Formats and Encodings  
for 64-Bit Mode (Contd.)**

Instruction and Format	Encoding
write RDX[31:0]:RAX[31:0] to RCX specified MSR	0100 1000 0000 1111 : 0011 0000
<b>XADD - Exchange and Add</b>	
register1, register2	0100 0ROB 0000 1111 : 1100 000w : 11 reg2 reg1
byteregister1, byteregister2	0100 0ROB 0000 1111 : 1100 0000 : 11 bytereg2 bytereg1
qwordregister1, qwordregister2	0100 0ROB 0000 1111 : 1100 0001 : 11 qwordreg2 qwordreg1
memory, register	0100 0RXB 0000 1111 : 1100 000w : mod reg r/m
memory8, bytereg	0100 1RXB 0000 1111 : 1100 0000 : mod bytereg r/m
memory64, qwordreg	0100 1RXB 0000 1111 : 1100 0001 : mod qwordreg r/m
<b>XCHG - Exchange Register/Memory with Register</b>	
register1 with register2	1000 011w : 11 reg1 reg2
AX or EAX with register	1001 0 reg
memory with register	1000 011w : mod reg r/m
<b>XLAT/XLATB - Table Look-up Translation</b>	
AL to byte DS:[(E)BX + unsigned AL]	1101 0111
AL to byte DS:[RBX + unsigned AL]	0100 1000 1101 0111
<b>XOR - Logical Exclusive OR</b>	
register1 to register2	0100 0RXB 0011 000w : 11 reg1 reg2
byteregister1 to byteregister2	0100 0ROB 0011 0000 : 11 bytereg1 bytereg2
qwordregister1 to qwordregister2	0100 1ROB 0011 0001 : 11 qwordreg1 qwordreg2
register2 to register1	0100 0ROB 0011 001w : 11 reg1 reg2
byteregister2 to byteregister1	0100 0ROB 0011 0010 : 11 bytereg1 bytereg2
qwordregister2 to qwordregister1	0100 1ROB 0011 0011 : 11 qwordreg1 qwordreg2

**Table B-15. General Purpose Instruction Formats and Encodings  
for 64-Bit Mode (Contd.)**

Instruction and Format	Encoding
memory to register	0100 0RXB 0011 001w : mod reg r/m
memory8 to byteregister	0100 0RXB 0011 0010 : mod bytereg r/m
memory64 to qwordregister	0100 1RXB 0011 0011 : mod qwordreg r/m
register to memory	0100 0RXB 0011 000w : mod reg r/m
byteregister to memory8	0100 0RXB 0011 0000 : mod bytereg r/m
qwordregister to memory8	0100 1RXB 0011 0001 : mod qwordreg r/m
immediate to register	0100 000B 1000 00sw : 11 110 reg : imm
immediate8 to byteregister	0100 000B 1000 0000 : 11 110 bytereg : imm8
immediate32 to qwordregister	0100 100B 1000 0001 : 11 110 qwordreg : imm32
immediate8 to qwordregister	0100 100B 1000 0011 : 11 110 qwordreg : imm8
immediate to AL, AX, or EAX	0100 000B 0011 010w : imm
immediate to RAX	0100 1000 0011 0101 : immediate data
immediate to memory	0100 00XB 1000 00sw : mod 110 r/m : imm
immediate8 to memory8	0100 00XB 1000 0000 : mod 110 r/m : imm8
immediate32 to memory64	0100 10XB 1000 0001 : mod 110 r/m : imm32
immediate8 to memory64	0100 10XB 1000 0011 : mod 110 r/m : imm8
<b>Prefix Bytes</b>	
address size	0110 0111
LOCK	1111 0000
operand size	0110 0110
CS segment override	0010 1110
DS segment override	0011 1110
ES segment override	0010 0110
FS segment override	0110 0100
GS segment override	0110 0101
SS segment override	0011 0110

## B.3 PENTIUM® PROCESSOR FAMILY INSTRUCTION FORMATS AND ENCODINGS

The following table shows formats and encodings introduced by the Pentium processor family.

**Table B-16. Pentium Processor Family Instruction Formats and Encodings, Non-64-Bit Modes**

Instruction and Format	Encoding
<b>CMPXCHG8B - Compare and Exchange 8 Bytes</b>	
EDX:EAX with memory64	0000 1111 : 1100 0111 : mod 001 r/m

**Table B-17. Pentium Processor Family Instruction Formats and Encodings, 64-Bit Mode**

Instruction and Format	Encoding
<b>CMPXCHG8B/CMPXCHG16B - Compare and Exchange Bytes</b>	
EDX:EAX with memory64	0000 1111 : 1100 0111 : mod 001 r/m
RDX:RAX with memory128	0100 10XB 0000 1111 : 1100 0111 : mod 001 r/m

## B.4 64-BIT MODE INSTRUCTION ENCODINGS FOR SIMD INSTRUCTION EXTENSIONS

Non-64-bit mode instruction encodings for MMX Technology, SSE, SSE2, and SSE3 are covered by applying these rules to Table B-19 through Table B-31. Table B-34 lists special encodings (instructions that do not follow the rules below).

- The REX instruction has no effect:
  - On immediates
  - If both operands are MMX registers
  - On MMX registers and XMM registers
  - If an MMX register is encoded in the reg field of the ModR/M byte
- If a memory operand is encoded in the r/m field of the ModR/M byte, REX.X and REX.B may be used for encoding the memory operand.

3. If a general-purpose register is encoded in the r/m field of the ModR/M byte, REX.B may be used for register encoding and REX.W may be used to encode the 64-bit operand size.
4. If an XMM register operand is encoded in the reg field of the ModR/M byte, REX.R may be used for register encoding. If an XMM register operand is encoded in the r/m field of the ModR/M byte, REX.B may be used for register encoding.

## B.5 MMX INSTRUCTION FORMATS AND ENCODINGS

MMX instructions, except the EMMS instruction, use a format similar to the 2-byte Intel Architecture integer format. Details of subfield encodings within these formats are presented below.

### B.5.1 Granularity Field (gg)

The granularity field (gg) indicates the size of the packed operands that the instruction is operating on. When this field is used, it is located in bits 1 and 0 of the second opcode byte. Table B-18 shows the encoding of the gg field.

**Table B-18. Encoding of Granularity of Data Field (gg)**

gg	Granularity of Data
00	Packed Bytes
01	Packed Words
10	Packed Doublewords
11	Quadword

### B.5.2 MMX Technology and General-Purpose Register Fields (mmxreg and reg)

When MMX technology registers (mmxreg) are used as operands, they are encoded in the ModR/M byte in the reg field (bits 5, 4, and 3) and/or the R/M field (bits 2, 1, and 0).

If an MMX instruction operates on a general-purpose register (reg), the register is encoded in the R/M field of the ModR/M byte.

### B.5.3 MMX Instruction Formats and Encodings Table

Table B-19 shows the formats and encodings of the integer instructions.

**Table B-19. MMX Instruction Formats and Encodings**

<b>Instruction and Format</b>	<b>Encoding</b>
<b>EMMS - Empty MMX technology state</b>	0000 1111:01110111
<b>MOVD - Move doubleword</b>	
reg to mmxreg	0000 1111:0110 1110: 11 mmxreg reg
reg from mmxreg	0000 1111:0111 1110: 11 mmxreg reg
mem to mmxreg	0000 1111:0110 1110: mod mmxreg r/m
mem from mmxreg	0000 1111:0111 1110: mod mmxreg r/m
<b>MOVQ - Move quadword</b>	
mmxreg2 to mmxreg1	0000 1111:0110 1111: 11 mmxreg1 mmxreg2
mmxreg2 from mmxreg1	0000 1111:0111 1111: 11 mmxreg1 mmxreg2
mem to mmxreg	0000 1111:0110 1111: mod mmxreg r/m
mem from mmxreg	0000 1111:0111 1111: mod mmxreg r/m
<b>PACKSSDW<sup>1</sup> - Pack dword to word data (signed with saturation)</b>	
mmxreg2 to mmxreg1	0000 1111:0110 1011: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111:0110 1011: mod mmxreg r/m
<b>PACKSSWB<sup>1</sup> - Pack word to byte data (signed with saturation)</b>	
mmxreg2 to mmxreg1	0000 1111:0110 0011: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111:0110 0011: mod mmxreg r/m
<b>PACKUSWB<sup>1</sup> - Pack word to byte data (unsigned with saturation)</b>	
mmxreg2 to mmxreg1	0000 1111:0110 0111: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111:0110 0111: mod mmxreg r/m
<b>PADD - Add with wrap-around</b>	
mmxreg2 to mmxreg1	0000 1111: 1111 11gg: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111: 1111 11gg: mod mmxreg r/m
<b>PADDs - Add signed with saturation</b>	
mmxreg2 to mmxreg1	0000 1111: 1110 11gg: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111: 1110 11gg: mod mmxreg r/m
<b>PADDUS - Add unsigned with saturation</b>	
mmxreg2 to mmxreg1	0000 1111: 1101 11gg: 11 mmxreg1 mmxreg2

**Table B-19. MMX Instruction Formats and Encodings (Contd.)**

<b>Instruction and Format</b>	<b>Encoding</b>
memory to mmxreg	0000 1111:1101 11gg: mod mmxreg r/m
<b>PAND - Bitwise And</b>	
mmxreg2 to mmxreg1	0000 1111:1101 1011: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111:1101 1011: mod mmxreg r/m
<b>PANDN - Bitwise AndNot</b>	
mmxreg2 to mmxreg1	0000 1111:1101 1111: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111:1101 1111: mod mmxreg r/m
<b>PCMPEQ - Packed compare for equality</b>	
mmxreg1 with mmxreg2	0000 1111:0111 01gg: 11 mmxreg1 mmxreg2
mmxreg with memory	0000 1111:0111 01gg: mod mmxreg r/m
<b>PCMPGT - Packed compare greater (signed)</b>	
mmxreg1 with mmxreg2	0000 1111:0110 01gg: 11 mmxreg1 mmxreg2
mmxreg with memory	0000 1111:0110 01gg: mod mmxreg r/m
<b>PMADDWD - Packed multiply add</b>	
mmxreg2 to mmxreg1	0000 1111:1111 0101: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111:1111 0101: mod mmxreg r/m
<b>PMULHUW - Packed multiplication, store high word (unsigned)</b>	
mmxreg2 to mmxreg1	0000 1111: 1110 0100: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111: 1110 0100: mod mmxreg r/m
<b>PMULHW - Packed multiplication, store high word</b>	
mmxreg2 to mmxreg1	0000 1111:1110 0101: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111:1110 0101: mod mmxreg r/m
<b>PMULLW - Packed multiplication, store low word</b>	
mmxreg2 to mmxreg1	0000 1111:1101 0101: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111:1101 0101: mod mmxreg r/m
<b>POR - Bitwise Or</b>	
mmxreg2 to mmxreg1	0000 1111:1110 1011: 11 mmxreg1 mmxreg2



**Table B-19. MMX Instruction Formats and Encodings (Contd.)**

<b>Instruction and Format</b>	<b>Encoding</b>
memory to mmxreg	0000 1111:1110 1011: mod mmxreg r/m
<b>PSLL<sup>2</sup> - Packed shift left logical</b>	
mmxreg1 by mmxreg2	0000 1111:1111 00gg: 11 mmxreg1 mmxreg2
mmxreg by memory	0000 1111:1111 00gg: mod mmxreg r/m
mmxreg by immediate	0000 1111:0111 00gg: 11 110 mmxreg: imm8 data
<b>PSRA<sup>2</sup> - Packed shift right arithmetic</b>	
mmxreg1 by mmxreg2	0000 1111:1110 00gg: 11 mmxreg1 mmxreg2
mmxreg by memory	0000 1111:1110 00gg: mod mmxreg r/m
mmxreg by immediate	0000 1111:0111 00gg: 11 100 mmxreg: imm8 data
<b>PSRL<sup>2</sup> - Packed shift right logical</b>	
mmxreg1 by mmxreg2	0000 1111:1101 00gg: 11 mmxreg1 mmxreg2
mmxreg by memory	0000 1111:1101 00gg: mod mmxreg r/m
mmxreg by immediate	0000 1111:0111 00gg: 11 010 mmxreg: imm8 data
<b>PSUB - Subtract with wrap-around</b>	
mmxreg2 from mmxreg1	0000 1111:1111 10gg: 11 mmxreg1 mmxreg2
memory from mmxreg	0000 1111:1111 10gg: mod mmxreg r/m
<b>PSUBS - Subtract signed with saturation</b>	
mmxreg2 from mmxreg1	0000 1111:1110 10gg: 11 mmxreg1 mmxreg2
memory from mmxreg	0000 1111:1110 10gg: mod mmxreg r/m
<b>PSUBUS - Subtract unsigned with saturation</b>	
mmxreg2 from mmxreg1	0000 1111:1101 10gg: 11 mmxreg1 mmxreg2
memory from mmxreg	0000 1111:1101 10gg: mod mmxreg r/m
<b>PUNPCKH - Unpack high data to next larger type</b>	
mmxreg2 to mmxreg1	0000 1111:0110 10gg: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111:0110 10gg: mod mmxreg r/m
<b>PUNPCKL - Unpack low data to next larger type</b>	
mmxreg2 to mmxreg1	0000 1111:0110 00gg: 11 mmxreg1 mmxreg2

**Table B-19. MMX Instruction Formats and Encodings (Contd.)**

Instruction and Format	Encoding
memory to mmxreg	0000 1111:0110 00gg: mod mmxreg r/m
<b>PXOR - Bitwise Xor</b>	
mmxreg2 to mmxreg1	0000 1111:1110 1111: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111:1110 1111: mod mmxreg r/m

**NOTES:**

1. The pack instructions perform saturation from signed packed data of one type to signed or unsigned data of the next smaller type.
2. The format of the shift instructions has one additional format to support shifting by immediate shift-counts. The shift operations are not supported equally for all data types.

## B.6 PROCESSOR EXTENDED STATE INSTRUCTION FORMATS AND ENCODINGS

Table B-20 shows the formats and encodings for several instructions that relate to processor extended state management.

**Table B-20. Formats and Encodings of XSAVE/XRSTOR/XGETBV/XSETBV Instructions**

Instruction and Format	Encoding
<b>XGETBV - Get Value of Extended Control Register</b>	0000 1111:0000 0001: 1101 0000
<b>XRSTOR - Restore Processor Extended States<sup>1</sup></b>	0000 1111:1010 1110: mod <sup>A</sup> 101 r/m
<b>XSAVE - Save Processor Extended States<sup>1</sup></b>	0000 1111:1010 1110: mod <sup>A</sup> 100 r/m
<b>XSETBV - Set Extended Control Register</b>	0000 1111:0000 0001: 1101 0001

**NOTES:**

1. For XSAVE and XRSTOR, "mod = 11" is reserved.

## B.7 P6 FAMILY INSTRUCTION FORMATS AND ENCODINGS

Table B-20 shows the formats and encodings for several instructions that were introduced into the IA-32 architecture in the P6 family processors.

**Table B-21. Formats and Encodings of P6 Family Instructions**

Instruction and Format	Encoding
<b>CMOVcc - Conditional Move</b>	
register2 to register1	0000 1111: 0100 tttt: 11 reg1 reg2
memory to register	0000 1111 : 0100 tttt : mod reg r/m
<b>FCMOVcc - Conditional Move on EFLAG Register Condition Codes</b>	
move if below (B)	11011 010: 11 000 ST(i)
move if equal (E)	11011 010: 11 001 ST(i)
move if below or equal (BE)	11011 010: 11 010 ST(i)
move if unordered (U)	11011 010: 11 011 ST(i)
move if not below (NB)	11011 011: 11 000 ST(i)
move if not equal (NE)	11011 011: 11 001 ST(i)
move if not below or equal (NBE)	11011 011: 11 010 ST(i)
move if not unordered (NU)	11011 011: 11 011 ST(i)
FCOMI - Compare Real and Set EFLAGS	11011 011: 11 110 ST(i)
<b>FXRSTOR - Restore x87 FPU, MMX, SSE, and SSE2 State<sup>1</sup></b>	0000 1111:1010 1110: mod <sup>A</sup> 001 r/m
<b>FXSAVE - Save x87 FPU, MMX, SSE, and SSE2 State<sup>1</sup></b>	0000 1111:1010 1110: mod <sup>A</sup> 000 r/m
<b>SYSENTER - Fast System Call</b>	0000 1111:0011 0100
<b>SYSEXIT - Fast Return from Fast System Call</b>	0000 1111:0011 0101

**NOTES:**

1. For FXSAVE and FXRSTOR, "mod = 11" is reserved.

## B.8 SSE INSTRUCTION FORMATS AND ENCODINGS

The SSE instructions use the ModR/M format and are preceded by the 0FH prefix byte. In general, operations are not duplicated to provide two directions (that is, separate load and store variants).

The following three tables (Tables B-22, B-23, and B-24) show the formats and encodings for the SSE SIMD floating-point, SIMD integer, and cacheability and memory ordering instructions, respectively. Some SSE instructions require a mandatory prefix (66H, F2H, F3H) as part of the two-byte opcode. Mandatory prefixes are included in the tables.

**Table B-22. Formats and Encodings of SSE Floating-Point Instructions**

Instruction and Format	Encoding
<b>ADDPS—Add Packed Single-Precision Floating-Point Values</b>	
xmmreg2 to xmmreg1	0000 1111:0101 1000:11 xmmreg1 xmmreg2
mem to xmmreg	0000 1111:0101 1000: mod xmmreg r/m
<b>ADDSS—Add Scalar Single-Precision Floating-Point Values</b>	
xmmreg2 to xmmreg1	1111 0011:0000 1111:01011000:11 xmmreg1 xmmreg2
mem to xmmreg	1111 0011:0000 1111:01011000: mod xmmreg r/m
<b>ANDNPS—Bitwise Logical AND NOT of Packed Single-Precision Floating-Point Values</b>	
xmmreg2 to xmmreg1	0000 1111:0101 0101:11 xmmreg1 xmmreg2
mem to xmmreg	0000 1111:0101 0101: mod xmmreg r/m
<b>ANDPS—Bitwise Logical AND of Packed Single-Precision Floating-Point Values</b>	
xmmreg2 to xmmreg1	0000 1111:0101 0100:11 xmmreg1 xmmreg2
mem to xmmreg	0000 1111:0101 0100: mod xmmreg r/m
<b>CMPPS—Compare Packed Single-Precision Floating-Point Values</b>	
xmmreg2 to xmmreg1, imm8	0000 1111:1100 0010:11 xmmreg1 xmmreg2: imm8
mem to xmmreg, imm8	0000 1111:1100 0010: mod xmmreg r/m: imm8
<b>CMPSD—Compare Scalar Single-Precision Floating-Point Values</b>	
xmmreg2 to xmmreg1, imm8	1111 0011:0000 1111:1100 0010:11 xmmreg1 xmmreg2: imm8
mem to xmmreg, imm8	1111 0011:0000 1111:1100 0010: mod xmmreg r/m: imm8
<b>COMISS—Compare Scalar Ordered Single-Precision Floating-Point Values and Set EFLAGS</b>	
xmmreg2 to xmmreg1	0000 1111:0010 1111:11 xmmreg1 xmmreg2
mem to xmmreg	0000 1111:0010 1111: mod xmmreg r/m

**Table B-22. Formats and Encodings of SSE Floating-Point Instructions (Contd.)**

Instruction and Format	Encoding
<b>CVTPI2PS—Convert Packed Doubleword Integers to Packed Single-Precision Floating-Point Values</b>	
mmreg to xmmreg	0000 1111:0010 1010:11 xmmreg1 mmreg1
mem to xmmreg	0000 1111:0010 1010: mod xmmreg r/m
<b>CVTPS2PI—Convert Packed Single-Precision Floating-Point Values to Packed Doubleword Integers</b>	
xmmreg to mmreg	0000 1111:0010 1101:11 mmreg1 xmmreg1
mem to mmreg	0000 1111:0010 1101: mod mmreg r/m
<b>CVTSI2SS—Convert Doubleword Integer to Scalar Single-Precision Floating-Point Value</b>	
r32 to xmmreg1	1111 0011:0000 1111:00101010:11 xmmreg1 r32
mem to xmmreg	1111 0011:0000 1111:00101010: mod xmmreg r/m
<b>CVTSS2SI—Convert Scalar Single-Precision Floating-Point Value to Doubleword Integer</b>	
xmmreg to r32	1111 0011:0000 1111:0010 1101:11 r32 xmmreg
mem to r32	1111 0011:0000 1111:0010 1101: mod r32 r/m
<b>CVTTPS2PI—Convert with Truncation Packed Single-Precision Floating-Point Values to Packed Doubleword Integers</b>	
xmmreg to mmreg	0000 1111:0010 1100:11 mmreg1 xmmreg1
mem to mmreg	0000 1111:0010 1100: mod mmreg r/m
<b>CVTTSS2SI—Convert with Truncation Scalar Single-Precision Floating-Point Value to Doubleword Integer</b>	
xmmreg to r32	1111 0011:0000 1111:0010 1100:11 r32 xmmreg1
mem to r32	1111 0011:0000 1111:0010 1100: mod r32 r/m
<b>DIVPS—Divide Packed Single-Precision Floating-Point Values</b>	
xmmreg2 to xmmreg1	0000 1111:0101 1110:11 xmmreg1 xmmreg2
mem to xmmreg	0000 1111:0101 1110: mod xmmreg r/m
<b>DIVSS—Divide Scalar Single-Precision Floating-Point Values</b>	

**Table B-22. Formats and Encodings of SSE Floating-Point Instructions (Contd.)**

Instruction and Format	Encoding
xmmreg2 to xmmreg1	1111 0011:0000 1111:0101 1110:11 xmmreg1 xmmreg2
mem to xmmreg	1111 0011:0000 1111:0101 1110: mod xmmreg r/m
<b>LDMXCSR—Load MXCSR Register State</b>	
m32 to MXCSR	0000 1111:1010 1110:mod <sup>A</sup> 010 mem
<b>MAXPS—Return Maximum Packed Single-Precision Floating-Point Values</b>	
xmmreg2 to xmmreg1	0000 1111:0101 1111:11 xmmreg1 xmmreg2
mem to xmmreg	0000 1111:0101 1111: mod xmmreg r/m
<b>MAXSS—Return Maximum Scalar Double-Precision Floating-Point Value</b>	
xmmreg2 to xmmreg1	1111 0011:0000 1111:0101 1111:11 xmmreg1 xmmreg2
mem to xmmreg	1111 0011:0000 1111:0101 1111: mod xmmreg r/m
<b>MINPS—Return Minimum Packed Double-Precision Floating-Point Values</b>	
xmmreg2 to xmmreg1	0000 1111:0101 1101:11 xmmreg1 xmmreg2
mem to xmmreg	0000 1111:0101 1101: mod xmmreg r/m
<b>MINSS—Return Minimum Scalar Double-Precision Floating-Point Value</b>	
xmmreg2 to xmmreg1	1111 0011:0000 1111:0101 1101:11 xmmreg1 xmmreg2
mem to xmmreg	1111 0011:0000 1111:0101 1101: mod xmmreg r/m
<b>MOVAPS—Move Aligned Packed Single-Precision Floating-Point Values</b>	
xmmreg2 to xmmreg1	0000 1111:0010 1000:11 xmmreg2 xmmreg1
mem to xmmreg1	0000 1111:0010 1000: mod xmmreg r/m
xmmreg1 to xmmreg2	0000 1111:0010 1001:11 xmmreg1 xmmreg2
xmmreg1 to mem	0000 1111:0010 1001: mod xmmreg r/m

**Table B-22. Formats and Encodings of SSE Floating-Point Instructions (Contd.)**

Instruction and Format	Encoding
<b>MOVHPS—Move Packed Single-Precision Floating-Point Values High to Low</b>	
xmmreg2 to xmmreg1	0000 1111:0001 0010:11 xmmreg1 xmmreg2
<b>MOVHPS—Move High Packed Single-Precision Floating-Point Values</b>	
mem to xmmreg	0000 1111:0001 0110: mod xmmreg r/m
xmmreg to mem	0000 1111:0001 0111: mod xmmreg r/m
<b>MOVLHPS—Move Packed Single-Precision Floating-Point Values Low to High</b>	
xmmreg2 to xmmreg1	0000 1111:00010110:11 xmmreg1 xmmreg2
<b>MOVLPS—Move Low Packed Single-Precision Floating-Point Values</b>	
mem to xmmreg	0000 1111:0001 0010: mod xmmreg r/m
xmmreg to mem	0000 1111:0001 0011: mod xmmreg r/m
<b>MOVMSKPS—Extract Packed Single-Precision Floating-Point Sign Mask</b>	
xmmreg to r32	0000 1111:0101 0000:11 r32 xmmreg
<b>MOVSS—Move Scalar Single-Precision Floating-Point Values</b>	
xmmreg2 to xmmreg1	1111 0011:0000 1111:0001 0000:11 xmmreg2 xmmreg1
mem to xmmreg1	1111 0011:0000 1111:0001 0000: mod xmmreg r/m
xmmreg1 to xmmreg2	1111 0011:0000 1111:0001 0001:11 xmmreg1 xmmreg2
xmmreg1 to mem	1111 0011:0000 1111:0001 0001: mod xmmreg r/m
<b>MOVUPS—Move Unaligned Packed Single-Precision Floating-Point Values</b>	
xmmreg2 to xmmreg1	0000 1111:0001 0000:11 xmmreg2 xmmreg1
mem to xmmreg1	0000 1111:0001 0000: mod xmmreg r/m
xmmreg1 to xmmreg2	0000 1111:0001 0001:11 xmmreg1 xmmreg2
xmmreg1 to mem	0000 1111:0001 0001: mod xmmreg r/m

**Table B-22. Formats and Encodings of SSE Floating-Point Instructions (Contd.)**

<b>Instruction and Format</b>	<b>Encoding</b>
<b>MULPS—Multiply Packed Single-Precision Floating-Point Values</b>	
xmmreg2 to xmmreg1	0000 1111:0101 1001:11 xmmreg1 xmmreg2
mem to xmmreg	0000 1111:0101 1001: mod xmmreg r/m
<b>MULSS—Multiply Scalar Single-Precision Floating-Point Values</b>	
xmmreg2 to xmmreg1	1111 0011:0000 1111:0101 1001:11 xmmreg1 xmmreg2
mem to xmmreg	1111 0011:0000 1111:0101 1001: mod xmmreg r/m
<b>ORPS—Bitwise Logical OR of Single-Precision Floating-Point Values</b>	
xmmreg2 to xmmreg1	0000 1111:0101 0110:11 xmmreg1 xmmreg2
mem to xmmreg	0000 1111:0101 0110: mod xmmreg r/m
<b>RCPPS—Compute Reciprocals of Packed Single-Precision Floating-Point Values</b>	
xmmreg2 to xmmreg1	0000 1111:0101 0011:11 xmmreg1 xmmreg2
mem to xmmreg	0000 1111:0101 0011: mod xmmreg r/m
<b>RCPSS—Compute Reciprocals of Scalar Single-Precision Floating-Point Value</b>	
xmmreg2 to xmmreg1	1111 0011:0000 1111:01010011:11 xmmreg1 xmmreg2
mem to xmmreg	1111 0011:0000 1111:01010011: mod xmmreg r/m
<b>RSQRTPS—Compute Reciprocals of Square Roots of Packed Single-Precision Floating-Point Values</b>	
xmmreg2 to xmmreg1	0000 1111:0101 0010:11 xmmreg1 xmmreg2
mem to xmmreg	0000 1111:0101 0010: mode xmmreg r/m
<b>RSQRTSS—Compute Reciprocals of Square Roots of Scalar Single-Precision Floating-Point Value</b>	
xmmreg2 to xmmreg1	1111 0011:0000 1111:0101 0010:11 xmmreg1 xmmreg2
mem to xmmreg	1111 0011:0000 1111:0101 0010: mod xmmreg r/m



**Table B-22. Formats and Encodings of SSE Floating-Point Instructions (Contd.)**

Instruction and Format	Encoding
<b>SHUFPS—Shuffle Packed Single-Precision Floating-Point Values</b>	
xmmreg2 to xmmreg1, imm8	0000 1111:1100 0110:11 xmmreg1 xmmreg2: imm8
mem to xmmreg, imm8	0000 1111:1100 0110: mod xmmreg r/m: imm8
<b>SQRTPS—Compute Square Roots of Packed Single-Precision Floating-Point Values</b>	
xmmreg2 to xmmreg1	0000 1111:0101 0001:11 xmmreg1 xmmreg2
mem to xmmreg	0000 1111:0101 0001: mod xmmreg r/m
<b>SQRTSS—Compute Square Root of Scalar Single-Precision Floating-Point Value</b>	
xmmreg2 to xmmreg1	1111 0011:0000 1111:0101 0001:11 xmmreg1 xmmreg2
mem to xmmreg	1111 0011:0000 1111:0101 0001:mod xmmreg r/m
<b>STMXCSR—Store MXCSR Register State</b>	
MXCSR to mem	0000 1111:1010 1110:mod <sup>A</sup> 011 mem
<b>SUBPS—Subtract Packed Single-Precision Floating-Point Values</b>	
xmmreg2 to xmmreg1	0000 1111:0101 1100:11 xmmreg1 xmmreg2
mem to xmmreg	0000 1111:0101 1100:mod xmmreg r/m
<b>SUBSS—Subtract Scalar Single-Precision Floating-Point Values</b>	
xmmreg2 to xmmreg1	1111 0011:0000 1111:0101 1100:11 xmmreg1 xmmreg2
mem to xmmreg	1111 0011:0000 1111:0101 1100:mod xmmreg r/m
<b>UCOMISS—Unordered Compare Scalar Ordered Single-Precision Floating-Point Values and Set EFLAGS</b>	
xmmreg2 to xmmreg1	0000 1111:0010 1110:11 xmmreg1 xmmreg2
mem to xmmreg	0000 1111:0010 1110: mod xmmreg r/m
<b>UNPCKHPS—Unpack and Interleave High Packed Single-Precision Floating-Point Values</b>	

**Table B-22. Formats and Encodings of SSE Floating-Point Instructions (Contd.)**

Instruction and Format	Encoding
xmmreg2 to xmmreg1	0000 1111:0001 0101:11 xmmreg1 xmmreg2
mem to xmmreg	0000 1111:0001 0101: mod xmmreg r/m
<b>UNPCKLPS—Unpack and Interleave Low Packed Single-Precision Floating-Point Values</b>	
xmmreg2 to xmmreg1	0000 1111:0001 0100:11 xmmreg1 xmmreg2
mem to xmmreg	0000 1111:0001 0100: mod xmmreg r/m
<b>XORPS—Bitwise Logical XOR of Single-Precision Floating-Point Values</b>	
xmmreg2 to xmmreg1	0000 1111:0101 0111:11 xmmreg1 xmmreg2
mem to xmmreg	0000 1111:0101 0111: mod xmmreg r/m

**Table B-23. Formats and Encodings of SSE Integer Instructions**

Instruction and Format	Encoding
<b>PAVGB/PAVGW—Average Packed Integers</b>	
mmreg2 to mmreg1	0000 1111:1110 0000:11 mmreg1 mmreg2
	0000 1111:1110 0011:11 mmreg1 mmreg2
mem to mmreg	0000 1111:1110 0000: mod mmreg r/m
	0000 1111:1110 0011: mod mmreg r/m
<b>PEXTRW—Extract Word</b>	
mmreg to reg32, imm8	0000 1111:1100 0101:11 r32 mmreg: imm8
<b>PINSRW—Insert Word</b>	
reg32 to mmreg, imm8	0000 1111:1100 0100:11 mmreg r32: imm8
m16 to mmreg, imm8	0000 1111:1100 0100: mod mmreg r/m: imm8
<b>PMAXSW—Maximum of Packed Signed Word Integers</b>	
mmreg2 to mmreg1	0000 1111:1110 1110:11 mmreg1 mmreg2
mem to mmreg	0000 1111:1110 1110: mod mmreg r/m
<b>PMAXB—Maximum of Packed Unsigned Byte Integers</b>	
mmreg2 to mmreg1	0000 1111:1101 1110:11 mmreg1 mmreg2
mem to mmreg	0000 1111:1101 1110: mod mmreg r/m

**Table B-23. Formats and Encodings of SSE Integer Instructions (Contd.)**

Instruction and Format	Encoding
<b>PMINSW—Minimum of Packed Signed Word Integers</b>	
mmreg2 to mmreg1	0000 1111:1110 1010:11 mmreg1 mmreg2
mem to mmreg	0000 1111:1110 1010: mod mmreg r/m
<b>PMINUB—Minimum of Packed Unsigned Byte Integers</b>	
mmreg2 to mmreg1	0000 1111:1101 1010:11 mmreg1 mmreg2
mem to mmreg	0000 1111:1101 1010: mod mmreg r/m
<b>PMOVBK—Move Byte Mask To Integer</b>	
mmreg to reg32	0000 1111:1101 0111:11 r32 mmreg
<b>PMULHUW—Multiply Packed Unsigned Integers and Store High Result</b>	
mmreg2 to mmreg1	0000 1111:1110 0100:11 mmreg1 mmreg2
mem to mmreg	0000 1111:1110 0100: mod mmreg r/m
<b>PSADB—Compute Sum of Absolute Differences</b>	
mmreg2 to mmreg1	0000 1111:1111 0110:11 mmreg1 mmreg2
mem to mmreg	0000 1111:1111 0110: mod mmreg r/m
<b>PSHUFW—Shuffle Packed Words</b>	
mmreg2 to mmreg1, imm8	0000 1111:0111 0000:11 mmreg1 mmreg2: imm8
mem to mmreg, imm8	0000 1111:0111 0000: mod mmreg r/m: imm8

**Table B-24. Format and Encoding of SSE Cacheability & Memory Ordering Instructions**

Instruction and Format	Encoding
<b>MASKMOVQ—Store Selected Bytes of Quadword</b>	
mmreg2 to mmreg1	0000 1111:1111 0111:11 mmreg1 mmreg2
<b>MOVNTPS—Store Packed Single-Precision Floating-Point Values Using Non-Temporal Hint</b>	
xmmreg to mem	0000 1111:0010 1011: mod xmmreg r/m

**Table B-24. Format and Encoding of SSE Cacheability & Memory Ordering Instructions (Contd.)**

Instruction and Format	Encoding
<b>MOVNTQ—Store Quadword Using Non-Temporal Hint</b>	
mmreg to mem	0000 1111:1110 0111: mod mmreg r/m
<b>PREFETCHT0—Prefetch Temporal to All Cache Levels</b>	0000 1111:0001 1000:mod <sup>A</sup> 001 mem
<b>PREFETCHT1—Prefetch Temporal to First Level Cache</b>	0000 1111:0001 1000:mod <sup>A</sup> 010 mem
<b>PREFETCHT2—Prefetch Temporal to Second Level Cache</b>	0000 1111:0001 1000:mod <sup>A</sup> 011 mem
<b>PREFETCHNTA—Prefetch Non-Temporal to All Cache Levels</b>	0000 1111:0001 1000:mod <sup>A</sup> 000 mem
<b>SFENCE—Store Fence</b>	0000 1111:1010 1110:11 111 000

## B.9 SSE2 INSTRUCTION FORMATS AND ENCODINGS

The SSE2 instructions use the ModR/M format and are preceded by the 0FH prefix byte. In general, operations are not duplicated to provide two directions (that is, separate load and store variants).

The following three tables show the formats and encodings for the SSE2 SIMD floating-point, SIMD integer, and cacheability instructions, respectively. Some SSE2 instructions require a mandatory prefix (66H, F2H, F3H) as part of the two-byte opcode. These prefixes are included in the tables.

### B.9.1 Granularity Field (gg)

The granularity field (gg) indicates the size of the packed operands that the instruction is operating on. When this field is used, it is located in bits 1 and 0 of the second opcode byte. Table B-25 shows the encoding of this gg field.

**Table B-25. Encoding of Granularity of Data Field (gg)**

gg	Granularity of Data
00	Packed Bytes
01	Packed Words
10	Packed Doublewords
11	Quadword

**Table B-26. Formats and Encodings of SSE2 Floating-Point Instructions**

<b>Instruction and Format</b>	<b>Encoding</b>
<b>ADDPD—Add Packed Double-Precision Floating-Point Values</b>	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0101 1000:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0101 1000: mod xmmreg r/m
<b>ADDSD—Add Scalar Double-Precision Floating-Point Values</b>	
xmmreg2 to xmmreg1	1111 0010:0000 1111:0101 1000:11 xmmreg1 xmmreg2
mem to xmmreg	1111 0010:0000 1111:0101 1000: mod xmmreg r/m
<b>ANDNPD—Bitwise Logical AND NOT of Packed Double-Precision Floating-Point Values</b>	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0101 0101:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0101 0101: mod xmmreg r/m
<b>ANDPD—Bitwise Logical AND of Packed Double-Precision Floating-Point Values</b>	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0101 0100:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0101 0100: mod xmmreg r/m
<b>CMPPD—Compare Packed Double-Precision Floating-Point Values</b>	
xmmreg2 to xmmreg1, imm8	0110 0110:0000 1111:1100 0010:11 xmmreg1 xmmreg2: imm8
mem to xmmreg, imm8	0110 0110:0000 1111:1100 0010: mod xmmreg r/m: imm8
<b>CMPD—Compare Scalar Double-Precision Floating-Point Values</b>	
xmmreg2 to xmmreg1, imm8	1111 0010:0000 1111:1100 0010:11 xmmreg1 xmmreg2: imm8
mem to xmmreg, imm8	11110 010:0000 1111:1100 0010: mod xmmreg r/m: imm8

**Table B-26. Formats and Encodings of SSE2 Floating-Point Instructions (Contd.)**

Instruction and Format	Encoding
<b>COMISD—Compare Scalar Ordered Double-Precision Floating-Point Values and Set EFLAGS</b>	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0010 1111:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0010 1111: mod xmmreg r/m
<b>CVTPI2PD—Convert Packed Doubleword Integers to Packed Double-Precision Floating-Point Values</b>	
mmreg to xmmreg	0110 0110:0000 1111:0010 1010:11 xmmreg1 mmreg1
mem to xmmreg	0110 0110:0000 1111:0010 1010: mod xmmreg r/m
<b>CVTPD2PI—Convert Packed Double-Precision Floating-Point Values to Packed Doubleword Integers</b>	
xmmreg to mmreg	0110 0110:0000 1111:0010 1101:11 mmreg1 xmmreg1
mem to mmreg	0110 0110:0000 1111:0010 1101: mod mmreg r/m
<b>CVTSI2SD—Convert Doubleword Integer to Scalar Double-Precision Floating-Point Value</b>	
r32 to xmmreg1	1111 0010:0000 1111:0010 1010:11 xmmreg r32
mem to xmmreg	1111 0010:0000 1111:0010 1010: mod xmmreg r/m
<b>CVTSD2SI—Convert Scalar Double-Precision Floating-Point Value to Doubleword Integer</b>	
xmmreg to r32	1111 0010:0000 1111:0010 1101:11 r32 xmmreg
mem to r32	1111 0010:0000 1111:0010 1101: mod r32 r/m
<b>CVTTPD2PI—Convert with Truncation Packed Double-Precision Floating-Point Values to Packed Doubleword Integers</b>	
xmmreg to mmreg	0110 0110:0000 1111:0010 1100:11 mmreg xmmreg
mem to mmreg	0110 0110:0000 1111:0010 1100: mod mmreg r/m

**Table B-26. Formats and Encodings of SSE2 Floating-Point Instructions (Contd.)**

Instruction and Format	Encoding
<b>CVTTSD2SI—Convert with Truncation Scalar Double-Precision Floating-Point Value to Doubleword Integer</b>	
xmmreg to r32	1111 0010:0000 1111:0010 1100:11 r32 xmmreg
mem to r32	1111 0010:0000 1111:0010 1100: mod r32 r/m
<b>CVTPD2PS—Covert Packed Double-Precision Floating-Point Values to Packed Single-Precision Floating-Point Values</b>	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0101 1010:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0101 1010: mod xmmreg r/m
<b>CVTPS2PD—Covert Packed Single-Precision Floating-Point Values to Packed Double-Precision Floating-Point Values</b>	
xmmreg2 to xmmreg1	0000 1111:0101 1010:11 xmmreg1 xmmreg2
mem to xmmreg	0000 1111:0101 1010: mod xmmreg r/m
<b>CVTSD2SS—Covert Scalar Double-Precision Floating-Point Value to Scalar Single-Precision Floating-Point Value</b>	
xmmreg2 to xmmreg1	1111 0010:0000 1111:0101 1010:11 xmmreg1 xmmreg2
mem to xmmreg	1111 0010:0000 1111:0101 1010: mod xmmreg r/m
<b>CVTSS2SD—Covert Scalar Single-Precision Floating-Point Value to Scalar Double-Precision Floating-Point Value</b>	
xmmreg2 to xmmreg1	1111 0011:0000 1111:0101 1010:11 xmmreg1 xmmreg2
mem to xmmreg	1111 0011:0000 1111:0101 1010: mod xmmreg r/m

**Table B-26. Formats and Encodings of SSE2 Floating-Point Instructions (Contd.)**

Instruction and Format	Encoding
<b>CVTPD2DQ—Convert Packed Double-Precision Floating-Point Values to Packed Doubleword Integers</b>	
xmmreg2 to xmmreg1	1111 0010:0000 1111:1110 0110:11 xmmreg1 xmmreg2
mem to xmmreg	1111 0010:0000 1111:1110 0110: mod xmmreg r/m
<b>CVTTPD2DQ—Convert With Truncation Packed Double-Precision Floating-Point Values to Packed Doubleword Integers</b>	
xmmreg2 to xmmreg1	0110 0110:0000 1111:1110 0110:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:1110 0110: mod xmmreg r/m
<b>CVTDQ2PD—Convert Packed Doubleword Integers to Packed Single-Precision Floating-Point Values</b>	
xmmreg2 to xmmreg1	1111 0011:0000 1111:1110 0110:11 xmmreg1 xmmreg2
mem to xmmreg	1111 0011:0000 1111:1110 0110: mod xmmreg r/m
<b>CVTPS2DQ—Convert Packed Single-Precision Floating-Point Values to Packed Doubleword Integers</b>	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0101 1011:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0101 1011: mod xmmreg r/m
<b>CVTTPS2DQ—Convert With Truncation Packed Single-Precision Floating-Point Values to Packed Doubleword Integers</b>	
xmmreg2 to xmmreg1	1111 0011:0000 1111:0101 1011:11 xmmreg1 xmmreg2
mem to xmmreg	1111 0011:0000 1111:0101 1011: mod xmmreg r/m
<b>CVTDQ2PS—Convert Packed Doubleword Integers to Packed Double-Precision Floating-Point Values</b>	
xmmreg2 to xmmreg1	0000 1111:0101 1011:11 xmmreg1 xmmreg2



**Table B-26. Formats and Encodings of SSE2 Floating-Point Instructions (Contd.)**

<b>Instruction and Format</b>	<b>Encoding</b>
mem to xmmreg	0000 1111:0101 1011: mod xmmreg r/m
<b>DIVPD—Divide Packed Double-Precision Floating-Point Values</b>	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0101 1110:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0101 1110: mod xmmreg r/m
<b>DIVSD—Divide Scalar Double-Precision Floating-Point Values</b>	
xmmreg2 to xmmreg1	1111 0010:0000 1111:0101 1110:11 xmmreg1 xmmreg2
mem to xmmreg	1111 0010:0000 1111:0101 1110: mod xmmreg r/m
<b>MAXPD—Return Maximum Packed Double-Precision Floating-Point Values</b>	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0101 1111:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0101 1111: mod xmmreg r/m
<b>MAXSD—Return Maximum Scalar Double-Precision Floating-Point Value</b>	
xmmreg2 to xmmreg1	1111 0010:0000 1111:0101 1111:11 xmmreg1 xmmreg2
mem to xmmreg	1111 0010:0000 1111:0101 1111: mod xmmreg r/m
<b>MINPD—Return Minimum Packed Double-Precision Floating-Point Values</b>	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0101 1101:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0101 1101: mod xmmreg r/m
<b>MINSD—Return Minimum Scalar Double-Precision Floating-Point Value</b>	
xmmreg2 to xmmreg1	1111 0010:0000 1111:0101 1101:11 xmmreg1 xmmreg2
mem to xmmreg	1111 0010:0000 1111:0101 1101: mod xmmreg r/m

**Table B-26. Formats and Encodings of SSE2 Floating-Point Instructions (Contd.)**

Instruction and Format	Encoding
<b>MOVAPD—Move Aligned Packed Double-Precision Floating-Point Values</b>	
xmmreg1 to xmmreg2	0110 0110:0000 1111:0010 1001:11 xmmreg2 xmmreg1
xmmreg1 to mem	0110 0110:0000 1111:0010 1001: mod xmmreg r/m
xmmreg2 to xmmreg1	0110 0110:0000 1111:0010 1000:11 xmmreg1 xmmreg2
mem to xmmreg1	0110 0110:0000 1111:0010 1000: mod xmmreg r/m
<b>MOVHPD—Move High Packed Double-Precision Floating-Point Values</b>	
xmmreg to mem	0110 0110:0000 1111:0001 0111: mod xmmreg r/m
mem to xmmreg	0110 0110:0000 1111:0001 0110: mod xmmreg r/m
<b>MOVLPD—Move Low Packed Double-Precision Floating-Point Values</b>	
xmmreg to mem	0110 0110:0000 1111:0001 0011: mod xmmreg r/m
mem to xmmreg	0110 0110:0000 1111:0001 0010: mod xmmreg r/m
<b>MOVMSKPD—Extract Packed Double-Precision Floating-Point Sign Mask</b>	
xmmreg to r32	0110 0110:0000 1111:0101 0000:11 r32 xmmreg
<b>MOVSD—Move Scalar Double-Precision Floating-Point Values</b>	
xmmreg1 to xmmreg2	1111 0010:0000 1111:0001 0001:11 xmmreg2 xmmreg1
xmmreg1 to mem	1111 0010:0000 1111:0001 0001: mod xmmreg r/m
xmmreg2 to xmmreg1	1111 0010:0000 1111:0001 0000:11 xmmreg1 xmmreg2
mem to xmmreg1	1111 0010:0000 1111:0001 0000: mod xmmreg r/m
<b>MOVUPD—Move Unaligned Packed Double-Precision Floating-Point Values</b>	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0001 0001:11 xmmreg2 xmmreg1
mem to xmmreg1	0110 0110:0000 1111:0001 0001: mod xmmreg r/m

**Table B-26. Formats and Encodings of SSE2 Floating-Point Instructions (Contd.)**

<b>Instruction and Format</b>	<b>Encoding</b>
xmmreg1 to xmmreg2	0110 0110:0000 1111:0001 0000:11 xmmreg1 xmmreg2
xmmreg1 to mem	0110 0110:0000 1111:0001 0000: mod xmmreg r/m
<b>MULPD—Multiply Packed Double-Precision Floating-Point Values</b>	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0101 1001:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0101 1001: mod xmmreg r/m
<b>MULSD—Multiply Scalar Double-Precision Floating-Point Values</b>	
xmmreg2 to xmmreg1	1111 0010:0000 1111:0101 1001:11 xmmreg1 xmmreg2
mem to xmmreg	1111 0010:0000 1111:0101 1001: mod xmmreg r/m
<b>ORPD—Bitwise Logical OR of Double-Precision Floating-Point Values</b>	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0101 0110:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0101 0110: mod xmmreg r/m
<b>SHUFPS—Shuffle Packed Single-Precision Floating-Point Values</b>	
xmmreg2 to xmmreg1, imm8	0110 0110:0000 1111:1100 0110:11 xmmreg1 xmmreg2: imm8
mem to xmmreg, imm8	0110 0110:0000 1111:1100 0110: mod xmmreg r/m: imm8
<b>SQRTPD—Compute Square Roots of Packed Double-Precision Floating-Point Values</b>	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0101 0001:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0101 0001: mod xmmreg r/m
<b>SQRTSD—Compute Square Root of Scalar Double-Precision Floating-Point Value</b>	
xmmreg2 to xmmreg1	1111 0010:0000 1111:0101 0001:11 xmmreg1 xmmreg2

**Table B-26. Formats and Encodings of SSE2 Floating-Point Instructions (Contd.)**

Instruction and Format	Encoding
mem to xmmreg	1111 0010:0000 1111:0101 0001: mod xmmreg r/m
<b>SUBPD—Subtract Packed Double-Precision Floating-Point Values</b>	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0101 1100:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0101 1100: mod xmmreg r/m
<b>SUBSD—Subtract Scalar Double-Precision Floating-Point Values</b>	
xmmreg2 to xmmreg1	1111 0010:0000 1111:0101 1100:11 xmmreg1 xmmreg2
mem to xmmreg	1111 0010:0000 1111:0101 1100: mod xmmreg r/m
<b>UCOMISD—Unordered Compare Scalar Ordered Double-Precision Floating-Point Values and Set EFLAGS</b>	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0010 1110:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0010 1110: mod xmmreg r/m
<b>UNPCKHPD—Unpack and Interleave High Packed Double-Precision Floating-Point Values</b>	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0001 0101:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0001 0101: mod xmmreg r/m
<b>UNPCKLPD—Unpack and Interleave Low Packed Double-Precision Floating-Point Values</b>	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0001 0100:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0001 0100: mod xmmreg r/m
<b>XORPD—Bitwise Logical OR of Double-Precision Floating-Point Values</b>	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0101 0111:11 xmmreg1 xmmreg2

**Table B-26. Formats and Encodings of SSE2 Floating-Point Instructions (Contd.)**

Instruction and Format	Encoding
mem to xmmreg	0110 0110:0000 1111:0101 0111: mod xmmreg r/m

**Table B-27. Formats and Encodings of SSE2 Integer Instructions**

<b>Instruction and Format</b>	<b>Encoding</b>
<b>MOVD—Move Doubleword</b>	
reg to xmmreg	0110 0110:0000 1111:0110 1110: 11 xmmreg reg
reg from xmmreg	0110 0110:0000 1111:0111 1110: 11 xmmreg reg
mem to xmmreg	0110 0110:0000 1111:0110 1110: mod xmmreg r/m
mem from xmmreg	0110 0110:0000 1111:0111 1110: mod xmmreg r/m
<b>MOVDQA—Move Aligned Double Quadword</b>	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0110 1111:11 xmmreg1 xmmreg2
xmmreg2 from xmmreg1	0110 0110:0000 1111:0111 1111:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0110 1111: mod xmmreg r/m
mem from xmmreg	0110 0110:0000 1111:0111 1111: mod xmmreg r/m
<b>MOVDQU—Move Unaligned Double Quadword</b>	
xmmreg2 to xmmreg1	1111 0011:0000 1111:0110 1111:11 xmmreg1 xmmreg2
xmmreg2 from xmmreg1	1111 0011:0000 1111:0111 1111:11 xmmreg1 xmmreg2
mem to xmmreg	1111 0011:0000 1111:0110 1111: mod xmmreg r/m
mem from xmmreg	1111 0011:0000 1111:0111 1111: mod xmmreg r/m
<b>MOVQ2DQ—Move Quadword from MMX to XMM Register</b>	
mmreg to xmmreg	1111 0011:0000 1111:1101 0110:11 mmreg1 mmreg2
<b>MOVDQ2Q—Move Quadword from XMM to MMX Register</b>	
xmmreg to mmreg	1111 0010:0000 1111:1101 0110:11 mmreg1 mmreg2
<b>MOVQ—Move Quadword</b>	
xmmreg2 to xmmreg1	1111 0011:0000 1111:0111 1110: 11 xmmreg1 xmmreg2
xmmreg2 from xmmreg1	0110 0110:0000 1111:1101 0110: 11 xmmreg1 xmmreg2
mem to xmmreg	1111 0011:0000 1111:0111 1110: mod xmmreg r/m

**Table B-27. Formats and Encodings of SSE2 Integer Instructions (Contd.)**

<b>Instruction and Format</b>	<b>Encoding</b>
mem from xmmreg	0110 0110:0000 1111:1101 0110: mod xmmreg r/m
<b>PACKSSDW<sup>1</sup>—Pack Dword To Word Data (signed with saturation)</b>	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0110 1011: 11 xmmreg1 xmmreg2
memory to xmmreg	0110 0110:0000 1111:0110 1011: mod xmmreg r/m
<b>PACKSSWB—Pack Word To Byte Data (signed with saturation)</b>	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0110 0011: 11 xmmreg1 xmmreg2
memory to xmmreg	0110 0110:0000 1111:0110 0011: mod xmmreg r/m
<b>PACKUSWB—Pack Word To Byte Data (unsigned with saturation)</b>	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0110 0111: 11 xmmreg1 xmmreg2
memory to xmmreg	0110 0110:0000 1111:0110 0111: mod xmmreg r/m
<b>PADDQ—Add Packed Quadword Integers</b>	
mmreg2 to mmreg1	0000 1111:1101 0100:11 mmreg1 mmreg2
mem to mmreg	0000 1111:1101 0100: mod mmreg r/m
xmmreg2 to xmmreg1	0110 0110:0000 1111:1101 0100:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:1101 0100: mod xmmreg r/m
<b>PADD—Add With Wrap-around</b>	
xmmreg2 to xmmreg1	0110 0110:0000 1111: 1111 11gg: 11 xmmreg1 xmmreg2
memory to xmmreg	0110 0110:0000 1111: 1111 11gg: mod xmmreg r/m
<b>PADDSD—Add Signed With Saturation</b>	
xmmreg2 to xmmreg1	0110 0110:0000 1111: 1110 11gg: 11 xmmreg1 xmmreg2
memory to xmmreg	0110 0110:0000 1111: 1110 11gg: mod xmmreg r/m
<b>PADDUS—Add Unsigned With Saturation</b>	
xmmreg2 to xmmreg1	0110 0110:0000 1111: 1101 11gg: 11 xmmreg1 xmmreg2

**Table B-27. Formats and Encodings of SSE2 Integer Instructions (Contd.)**

<b>Instruction and Format</b>	<b>Encoding</b>
memory to xmmreg	0110 0110:0000 1111: 1101 11gg: mod xmmreg r/m
<b>PAND—Bitwise And</b>	
xmmreg2 to xmmreg1	0110 0110:0000 1111:1101 1011: 11 xmmreg1 xmmreg2
memory to xmmreg	0110 0110:0000 1111:1101 1011: mod xmmreg r/m
<b>PANDN—Bitwise AndNot</b>	
xmmreg2 to xmmreg1	0110 0110:0000 1111:1101 1111: 11 xmmreg1 xmmreg2
memory to xmmreg	0110 0110:0000 1111:1101 1111: mod xmmreg r/m
<b>PAVGB—Average Packed Integers</b>	
xmmreg2 to xmmreg1	0110 0110:0000 1111:11100 000:11 xmmreg1 xmmreg2
mem to xmmreg	01100110:00001111:11100000 mod xmmreg r/m
<b>PAVGW—Average Packed Integers</b>	
xmmreg2 to xmmreg1	0110 0110:0000 1111:1110 0011:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:1110 0011 mod xmmreg r/m
<b>PCMPEQ—Packed Compare For Equality</b>	
xmmreg1 with xmmreg2	0110 0110:0000 1111:0111 01gg: 11 xmmreg1 xmmreg2
xmmreg with memory	0110 0110:0000 1111:0111 01gg: mod xmmreg r/m
<b>PCMPGT—Packed Compare Greater (signed)</b>	
xmmreg1 with xmmreg2	0110 0110:0000 1111:0110 01gg: 11 xmmreg1 xmmreg2
xmmreg with memory	0110 0110:0000 1111:0110 01gg: mod xmmreg r/m
<b>PEXTRW—Extract Word</b>	
xmmreg to reg32, imm8	0110 0110:0000 1111:1100 0101:11 r32 xmmreg: imm8
<b>PINSRW—Insert Word</b>	
reg32 to xmmreg, imm8	0110 0110:0000 1111:1100 0100:11 xmmreg r32: imm8



**Table B-27. Formats and Encodings of SSE2 Integer Instructions (Contd.)**

<b>Instruction and Format</b>	<b>Encoding</b>
m16 to xmmreg, imm8	0110 0110:0000 1111:1100 0100: mod xmmreg r/m: imm8
<b>PMADDWD—Packed Multiply Add</b>	
xmmreg2 to xmmreg1	0110 0110:0000 1111:1111 0101: 11 xmmreg1 xmmreg2
memory to xmmreg	0110 0110:0000 1111:1111 0101: mod xmmreg r/m
<b>PMAXSW—Maximum of Packed Signed Word Integers</b>	
xmmreg2 to xmmreg1	0110 0110:0000 1111:1110 1110:11 xmmreg1 xmmreg2
mem to xmmreg	01100110:00001111:11101110: mod xmmreg r/m
<b>PMAXBW—Maximum of Packed Unsigned Byte Integers</b>	
xmmreg2 to xmmreg1	0110 0110:0000 1111:1101 1110:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:1101 1110: mod xmmreg r/m
<b>PMINSW—Minimum of Packed Signed Word Integers</b>	
xmmreg2 to xmmreg1	0110 0110:0000 1111:1110 1010:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:1110 1010: mod xmmreg r/m
<b>PMINUB—Minimum of Packed Unsigned Byte Integers</b>	
xmmreg2 to xmmreg1	0110 0110:0000 1111:1101 1010:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:1101 1010 mod xmmreg r/m
<b>PMOVBKB—Move Byte Mask To Integer</b>	
xmmreg to reg32	0110 0110:0000 1111:1101 0111:11 r32 xmmreg
<b>PMULHUW—Packed multiplication, store high word (unsigned)</b>	
xmmreg2 to xmmreg1	0110 0110:0000 1111:1110 0100: 11 xmmreg1 xmmreg2
memory to xmmreg	0110 0110:0000 1111:1110 0100: mod xmmreg r/m

**Table B-27. Formats and Encodings of SSE2 Integer Instructions (Contd.)**

<b>Instruction and Format</b>	<b>Encoding</b>
<b>PMULHW—Packed Multiplication, store high word</b>	
xmmreg2 to xmmreg1	0110 0110:0000 1111:1110 0101: 11 xmmreg1 xmmreg2
memory to xmmreg	0110 0110:0000 1111:1110 0101: mod xmmreg r/m
<b>PMULLW—Packed Multiplication, store low word</b>	
xmmreg2 to xmmreg1	0110 0110:0000 1111:1101 0101: 11 xmmreg1 xmmreg2
memory to xmmreg	0110 0110:0000 1111:1101 0101: mod xmmreg r/m
<b>PMULUDQ—Multiply Packed Unsigned Doubleword Integers</b>	
mmreg2 to mmreg1	0000 1111:1111 0100:11 mmreg1 mmreg2
mem to mmreg	0000 1111:1111 0100: mod mmreg r/m
xmmreg2 to xmmreg1	0110 0110:00001111:1111 0100:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:00001111:1111 0100: mod xmmreg r/m
<b>POR—Bitwise Or</b>	
xmmreg2 to xmmreg1	0110 0110:0000 1111:1110 1011: 11 xmmreg1 xmmreg2
memory to xmmreg	0110 0110:0000 1111:1110 1011: mod xmmreg r/m
<b>PSADBW—Compute Sum of Absolute Differences</b>	
xmmreg2 to xmmreg1	0110 0110:0000 1111:1111 0110:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:1111 0110: mod xmmreg r/m
<b>PSHUFLW—Shuffle Packed Low Words</b>	
xmmreg2 to xmmreg1, imm8	1111 0010:0000 1111:0111 0000:11 xmmreg1 xmmreg2: imm8
mem to xmmreg, imm8	1111 0010:0000 1111:0111 0000:11 mod xmmreg r/m: imm8

**Table B-27. Formats and Encodings of SSE2 Integer Instructions (Contd.)**

<b>Instruction and Format</b>	<b>Encoding</b>
<b>PSHUFHW—Shuffle Packed High Words</b>	
xmmreg2 to xmmreg1, imm8	1111 0011:0000 1111:0111 0000:11 xmmreg1 xmmreg2: imm8
mem to xmmreg, imm8	1111 0011:0000 1111:0111 0000: mod xmmreg r/m: imm8
<b>PSHUFD—Shuffle Packed Doublewords</b>	
xmmreg2 to xmmreg1, imm8	0110 0110:0000 1111:0111 0000:11 xmmreg1 xmmreg2: imm8
mem to xmmreg, imm8	0110 0110:0000 1111:0111 0000: mod xmmreg r/m: imm8
<b>PSLLDQ—Shift Double Quadword Left Logical</b>	
xmmreg, imm8	0110 0110:0000 1111:0111 0011:11 111 xmmreg: imm8
<b>PSLL—Packed Shift Left Logical</b>	
xmmreg1 by xmmreg2	0110 0110:0000 1111:1111 00gg: 11 xmmreg1 xmmreg2
xmmreg by memory	0110 0110:0000 1111:1111 00gg: mod xmmreg r/m
xmmreg by immediate	0110 0110:0000 1111:0111 00gg: 11 110 xmmreg: imm8
<b>PSRA—Packed Shift Right Arithmetic</b>	
xmmreg1 by xmmreg2	0110 0110:0000 1111:1110 00gg: 11 xmmreg1 xmmreg2
xmmreg by memory	0110 0110:0000 1111:1110 00gg: mod xmmreg r/m
xmmreg by immediate	0110 0110:0000 1111:0111 00gg: 11 100 xmmreg: imm8
<b>PSRLDQ—Shift Double Quadword Right Logical</b>	
xmmreg, imm8	0110 0110:0000 1111:0111 0011:11 011 xmmreg: imm8
<b>PSRL—Packed Shift Right Logical</b>	
xmmreg1 by xmmreg2	0110 0110:0000 1111:1101 00gg: 11 xmmreg1 xmmreg2
xmmreg by memory	0110 0110:0000 1111:1101 00gg: mod xmmreg r/m

**Table B-27. Formats and Encodings of SSE2 Integer Instructions (Contd.)**

<b>Instruction and Format</b>	<b>Encoding</b>
xmmreg by immediate	0110 0110:0000 1111:0111 00gg: 11 010 xmmreg: imm8
<b>PSUBQ—Subtract Packed Quadword Integers</b>	
mmreg2 to mmreg1	0000 1111:11111 011:11 mmreg1 mmreg2
mem to mmreg	0000 1111:1111 1011: mod mmreg r/m
xmmreg2 to xmmreg1	0110 0110:0000 1111:1111 1011:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:1111 1011: mod xmmreg r/m
<b>PSUB—Subtract With Wrap-around</b>	
xmmreg2 from xmmreg1	0110 0110:0000 1111:1111 10gg: 11 xmmreg1 xmmreg2
memory from xmmreg	0110 0110:0000 1111:1111 10gg: mod xmmreg r/m
<b>PSUBS—Subtract Signed With Saturation</b>	
xmmreg2 from xmmreg1	0110 0110:0000 1111:1110 10gg: 11 xmmreg1 xmmreg2
memory from xmmreg	0110 0110:0000 1111:1110 10gg: mod xmmreg r/m
<b>PSUBUS—Subtract Unsigned With Saturation</b>	
xmmreg2 from xmmreg1	0000 1111:1101 10gg: 11 xmmreg1 xmmreg2
memory from xmmreg	0000 1111:1101 10gg: mod xmmreg r/m
<b>PUNPCKH—Unpack High Data To Next Larger Type</b>	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0110 10gg:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0110 10gg: mod xmmreg r/m
<b>PUNPCKHQDQ—Unpack High Data</b>	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0110 1101:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0110 1101: mod xmmreg r/m
<b>PUNPCKL—Unpack Low Data To Next Larger Type</b>	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0110 00gg:11 xmmreg1 xmmreg2

**Table B-27. Formats and Encodings of SSE2 Integer Instructions (Contd.)**

Instruction and Format	Encoding
mem to xmmreg	0110 0110:0000 1111:0110 00gg: mod xmmreg r/m
<b>PUNPCKLQDQ—Unpack Low Data</b>	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0110 1100:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0110 1100: mod xmmreg r/m
<b>PXOR—Bitwise Xor</b>	
xmmreg2 to xmmreg1	0110 0110:0000 1111:1110 1111: 11 xmmreg1 xmmreg2
memory to xmmreg	0110 0110:0000 1111:1110 1111: mod xmmreg r/m

**Table B-28. Format and Encoding of SSE2 Cacheability Instructions**

Instruction and Format	Encoding
<b>MASKMOVDQU—Store Selected Bytes of Double Quadword</b>	
xmmreg2 to xmmreg1	0110 0110:0000 1111:1111 0111:11 xmmreg1 xmmreg2
<b>CLFLUSH—Flush Cache Line</b>	
mem	0000 1111:1010 1110: mod 111 r/m
<b>MOVNTPD—Store Packed Double-Precision Floating-Point Values Using Non-Temporal Hint</b>	
xmmreg to mem	0110 0110:0000 1111:0010 1011: mod xmmreg r/m
<b>MOVNTDQ—Store Double Quadword Using Non-Temporal Hint</b>	
xmmreg to mem	0110 0110:0000 1111:1110 0111: mod xmmreg r/m
<b>MOVNTI—Store Doubleword Using Non-Temporal Hint</b>	
reg to mem	0000 1111:1100 0011: mod reg r/m
<b>PAUSE—Spin Loop Hint</b>	1111 0011:1001 0000
<b>LFENCE—Load Fence</b>	0000 1111:1010 1110: 11 101 000
<b>MFENCE—Memory Fence</b>	0000 1111:1010 1110: 11 110 000

## B.10 SSE3 FORMATS AND ENCODINGS TABLE

The tables in this section provide SSE3 formats and encodings. Some SSE3 instructions require a mandatory prefix (66H, F2H, F3H) as part of the two-byte opcode. These prefixes are included in the tables.

When in IA-32e mode, use of the REX.R prefix permits instructions that use general purpose and XMM registers to access additional registers. Some instructions require the REX.W prefix to promote the instruction to 64-bit operation. Instructions that require the REX.W prefix are listed (with their opcodes) in Section B.13.

**Table B-29. Formats and Encodings of SSE3 Floating-Point Instructions**

Instruction and Format	Encoding
<b>ADDSD—Add /Sub packed DP FP numbers from XMM2/Mem to XMM1</b>	
xmmreg2 to xmmreg1	01100110:00001111:11010000:11 xmmreg1 xmmreg2
mem to xmmreg	01100110:00001111:11010000: mod xmmreg r/m
<b>ADDSPS—Add /Sub packed SP FP numbers from XMM2/Mem to XMM1</b>	
xmmreg2 to xmmreg1	11110010:00001111:11010000:11 xmmreg1 xmmreg2
mem to xmmreg	11110010:00001111:11010000: mod xmmreg r/m
<b>HADDSD—Add horizontally packed DP FP numbers XMM2/Mem to XMM1</b>	
xmmreg2 to xmmreg1	01100110:00001111:01111100:11 xmmreg1 xmmreg2
mem to xmmreg	01100110:00001111:01111100: mod xmmreg r/m
<b>HADDSPS—Add horizontally packed SP FP numbers XMM2/Mem to XMM1</b>	
xmmreg2 to xmmreg1	11110010:00001111:01111100:11 xmmreg1 xmmreg2
mem to xmmreg	11110010:00001111:01111100: mod xmmreg r/m
<b>HSUBSD—Sub horizontally packed DP FP numbers XMM2/Mem to XMM1</b>	
xmmreg2 to xmmreg1	01100110:00001111:01111101:11 xmmreg1 xmmreg2

**Table B-29. Formats and Encodings of SSE3 Floating-Point Instructions (Contd.)**

Instruction and Format	Encoding
mem to xmmreg	01100110:00001111:01111101: mod xmmreg r/m
<b>HSUBPS—Sub horizontally packed SP FP numbers XMM2/Mem to XMM1</b>	
xmmreg2 to xmmreg1	11110010:00001111:01111101:11 xmmreg1 xmmreg2
mem to xmmreg	11110010:00001111:01111101: mod xmmreg r/m

**Table B-30. Formats and Encodings for SSE3 Event Management Instructions**

Instruction and Format	Encoding
<b>MONITOR—Set up a linear address range to be monitored by hardware</b>	
eax, ecx, edx	0000 1111 : 0000 0001:11 001 000
<b>MWAIT—Wait until write-back store performed within the range specified by the instruction MONITOR</b>	
eax, ecx	0000 1111 : 0000 0001:11 001 001

**Table B-31. Formats and Encodings for SSE3 Integer and Move Instructions**

Instruction and Format	Encoding
<b>FISTTP—Store ST in int16 (chop) and pop</b>	
m16int	11011 111 : mod <sup>A</sup> 001 r/m
<b>FISTTP—Store ST in int32 (chop) and pop</b>	
m32int	11011 011 : mod <sup>A</sup> 001 r/m
<b>FISTTP—Store ST in int64 (chop) and pop</b>	
m64int	11011 101 : mod <sup>A</sup> 001 r/m
<b>LDDQU—Load unaligned integer 128-bit</b>	
xmm, m128	11110010:00001111:11110000: mod <sup>A</sup> xmmreg r/m
<b>MOVDDUP—Move 64 bits representing one DP data from XMM2/Mem to XMM1 and duplicate</b>	
xmmreg2 to xmmreg1	11110010:00001111:00010010:11 xmmreg1 xmmreg2

**Table B-31. Formats and Encodings for SSE3 Integer and Move Instructions (Contd.)**

Instruction and Format	Encoding
mem to xmmreg	11110010:00001111:00010010: mod xmmreg r/m
<b>MOVSHDUP—Move 128 bits representing 4 SP data from XMM2/Mem to XMM1 and duplicate high</b>	
xmmreg2 to xmmreg1	11110011:00001111:00010110:11 xmmreg1 xmmreg2
mem to xmmreg	11110011:00001111:00010110: mod xmmreg r/m
<b>MOVSLDUP—Move 128 bits representing 4 SP data from XMM2/Mem to XMM1 and duplicate low</b>	
xmmreg2 to xmmreg1	11110011:00001111:00010010:11 xmmreg1 xmmreg2
mem to xmmreg	11110011:00001111:00010010: mod xmmreg r/m

## B.11 SSSE3 FORMATS AND ENCODING TABLE

The tables in this section provide SSSE3 formats and encodings. Some SSSE3 instructions require a mandatory prefix (66H) as part of the three-byte opcode. These prefixes are included in the table below.

**Table B-32. Formats and Encodings for SSSE3 Instructions**

Instruction and Format	Encoding
<b>PABSB—Packed Absolute Value Bytes</b>	
mmreg2 to mmreg1	0000 1111:0011 1000: 0001 1100:11 mmreg1 mmreg2
mem to mmreg	0000 1111:0011 1000: 0001 1100: mod mmreg r/m
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000: 0001 1100:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0001 1100: mod xmmreg r/m
<b>PABSD—Packed Absolute Value Double Words</b>	
mmreg2 to mmreg1	0000 1111:0011 1000: 0001 1110:11 mmreg1 mmreg2



**Table B-32. Formats and Encodings for SSSE3 Instructions (Contd.)**

<b>Instruction and Format</b>	<b>Encoding</b>
mem to mmreg	0000 1111:0011 1000: 0001 1110: mod mmreg r/m
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000: 0001 1110:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0001 1110: mod xmmreg r/m
<b>PABSW—Packed Absolute Value Words</b>	
mmreg2 to mmreg1	0000 1111:0011 1000: 0001 1101:11 mmreg1 mmreg2
mem to mmreg	0000 1111:0011 1000: 0001 1101: mod mmreg r/m
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000: 0001 1101:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0001 1101: mod xmmreg r/m
<b>PALIGNR—Packed Align Right</b>	
mmreg2 to mmreg1, imm8	0000 1111:0011 1010: 0000 1111:11 mmreg1 mmreg2: imm8
mem to mmreg, imm8	0000 1111:0011 1010: 0000 1111: mod mmreg r/m: imm8
xmmreg2 to xmmreg1, imm8	0110 0110:0000 1111:0011 1010: 0000 1111:11 xmmreg1 xmmreg2: imm8
mem to xmmreg, imm8	0110 0110:0000 1111:0011 1010: 0000 1111: mod xmmreg r/m: imm8
<b>PHADDD—Packed Horizontal Add Double Words</b>	
mmreg2 to mmreg1	0000 1111:0011 1000: 0000 0010:11 mmreg1 mmreg2
mem to mmreg	0000 1111:0011 1000: 0000 0010: mod mmreg r/m
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000: 0000 0010:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0000 0010: mod xmmreg r/m
<b>PHADDSW—Packed Horizontal Add and Saturate</b>	
mmreg2 to mmreg1	0000 1111:0011 1000: 0000 0011:11 mmreg1 mmreg2
mem to mmreg	0000 1111:0011 1000: 0000 0011: mod mmreg r/m
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000: 0000 0011:11 xmmreg1 xmmreg2

**Table B-32. Formats and Encodings for SSE3 Instructions (Contd.)**

<b>Instruction and Format</b>	<b>Encoding</b>
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0000 0011: mod xmmreg r/m
<b>PHADDW—Packed Horizontal Add Words</b>	
mmreg2 to mmreg1	0000 1111:0011 1000: 0000 0001:11 mmreg1 mmreg2
mem to mmreg	0000 1111:0011 1000: 0000 0001: mod mmreg r/m
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000: 0000 0001:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0000 0001: mod xmmreg r/m
<b>PHSUBD—Packed Horizontal Subtract Double Words</b>	
mmreg2 to mmreg1	0000 1111:0011 1000: 0000 0110:11 mmreg1 mmreg2
mem to mmreg	0000 1111:0011 1000: 0000 0110: mod mmreg r/m
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000: 0000 0110:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0000 0110: mod xmmreg r/m
<b>PHSUBSW—Packed Horizontal Subtract and Saturate</b>	
mmreg2 to mmreg1	0000 1111:0011 1000: 0000 0111:11 mmreg1 mmreg2
mem to mmreg	0000 1111:0011 1000: 0000 0111: mod mmreg r/m
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000: 0000 0111:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0000 0111: mod xmmreg r/m
<b>PHSUBW—Packed Horizontal Subtract Words</b>	
mmreg2 to mmreg1	0000 1111:0011 1000: 0000 0101:11 mmreg1 mmreg2
mem to mmreg	0000 1111:0011 1000: 0000 0101: mod mmreg r/m
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000: 0000 0101:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0000 0101: mod xmmreg r/m

**Table B-32. Formats and Encodings for SSSE3 Instructions (Contd.)**

<b>Instruction and Format</b>	<b>Encoding</b>
<b>PMADDUBSW—Multiply and Add Packed Signed and Unsigned Bytes</b>	
mmreg2 to mmreg1	0000 1111:0011 1000: 0000 0100:11 mmreg1 mmreg2
mem to mmreg	0000 1111:0011 1000: 0000 0100: mod mmreg r/m
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000: 0000 0100:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0000 0100: mod xmmreg r/m
<b>PMULHRSW—Packed Multiply Hlgn with Round and Scale</b>	
mmreg2 to mmreg1	0000 1111:0011 1000: 0000 1011:11 mmreg1 mmreg2
mem to mmreg	0000 1111:0011 1000: 0000 1011: mod mmreg r/m
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000: 0000 1011:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0000 1011: mod xmmreg r/m
<b>PSHUFB—Packed Shuffle Bytes</b>	
mmreg2 to mmreg1	0000 1111:0011 1000: 0000 0000:11 mmreg1 mmreg2
mem to mmreg	0000 1111:0011 1000: 0000 0000: mod mmreg r/m
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000: 0000 0000:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0000 0000: mod xmmreg r/m
<b>PSIGNB—Packed Sign Bytes</b>	
mmreg2 to mmreg1	0000 1111:0011 1000: 0000 1000:11 mmreg1 mmreg2
mem to mmreg	0000 1111:0011 1000: 0000 1000: mod mmreg r/m
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000: 0000 1000:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0000 1000: mod xmmreg r/m
<b>PSIGND—Packed Sign Double Words</b>	
mmreg2 to mmreg1	0000 1111:0011 1000: 0000 1010:11 mmreg1 mmreg2
mem to mmreg	0000 1111:0011 1000: 0000 1010: mod mmreg r/m

**Table B-32. Formats and Encodings for SSE3 Instructions (Contd.)**

Instruction and Format	Encoding
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000: 0000 1010:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0000 1010: mod xmmreg r/m
<b>PSIGNW—Packed Sign Words</b>	
mmreg2 to mmreg1	0000 1111:0011 1000: 0000 1001:11 mmreg1 mmreg2
mem to mmreg	0000 1111:0011 1000: 0000 1001: mod mmreg r/m
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000: 0000 1001:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0000 1001: mod xmmreg r/m

## B.12 AESNI AND PCLMULQDQ INSTRUCTION FORMATS AND ENCODINGS

Table B-33 shows the formats and encodings for AESNI and PCLMULQDQ instructions.

**Table B-33. Formats and Encodings of AESNI and PCLMULQDQ Instructions**

Instruction and Format	Encoding
<b>AESDEC—Perform One Round of an AES Decryption Flow</b>	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000:1101 1110:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000:1101 1110: mod xmmreg r/m
<b>AESDECLAST—Perform Last Round of an AES Decryption Flow</b>	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000:1101 1111:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000:1101 1111: mod xmmreg r/m
<b>AESENC—Perform One Round of an AES Encryption Flow</b>	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000:1101 1100:11 xmmreg1 xmmreg2

**Table B-33. Formats and Encodings of AESNI and PCLMULQDQ Instructions**

Instruction and Format	Encoding
mem to xmmreg	0110 0110:0000 1111:0011 1000:1101 1100: mod xmmreg r/m
<b>AESENCLAST—Perform Last Round of an AES Encryption Flow</b>	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000:1101 1101:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000:1101 1101: mod xmmreg r/m
<b>AESIMC—Perform the AES InvMixColumn Transformation</b>	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000:1101 1011:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000:1101 1011: mod xmmreg r/m
<b>AESKEYGENASSIST—AES Round Key Generation Assist</b>	
xmmreg2 to xmmreg1, imm8	0110 0110:0000 1111:0011 1010:1101 1111:11 xmmreg1 xmmreg2: imm8
mem to xmmreg, imm8	0110 0110:0000 1111:0011 1010:1101 1111: mod xmmreg r/m: imm8
<b>PCLMULQDQ—Carry-Less Multiplication Quadword</b>	
xmmreg2 to xmmreg1, imm8	0110 0110:0000 1111:0011 1010:0100 0100:11 xmmreg1 xmmreg2: imm8
mem to xmmreg, imm8	0110 0110:0000 1111:0011 1010:0100 0100: mod xmmreg r/m: imm8

## B.13 SPECIAL ENCODINGS FOR 64-BIT MODE

The following Pentium, P6, MMX, SSE, SSE2, SSE3 instructions are promoted to 64-bit operation in IA-32e mode by using REX.W. However, these entries are special cases that do not follow the general rules (specified in Section B.4).

**Table B-34. Special Case Instructions Promoted Using REX.W**

<b>Instruction and Format</b>	<b>Encoding</b>
<b>CMOVcc—Conditional Move</b>	
register2 to register1	0100 0ROB 0000 1111: 0100 ttn: 11 reg1 reg2
qwordregister2 to qwordregister1	0100 1ROB 0000 1111: 0100 ttn: 11 qwordreg1 qwordreg2
memory to register	0100 0RXB 0000 1111 : 0100 ttn : mod reg r/m
memory64 to qwordregister	0100 1RXB 0000 1111 : 0100 ttn : mod qwordreg r/m
<b>CVTSD2SI—Convert Scalar Double-Precision Floating-Point Value to Doubleword Integer</b>	
xmmreg to r32	0100 0ROB 1111 0010:0000 1111:0010 1101:11 r32 xmmreg
xmmreg to r64	0100 1ROB 1111 0010:0000 1111:0010 1101:11 r64 xmmreg
mem64 to r32	0100 0ROXB 1111 0010:0000 1111:0010 1101: mod r32 r/m
mem64 to r64	0100 1RXB 1111 0010:0000 1111:0010 1101: mod r64 r/m
<b>CVTSI2SS—Convert Doubleword Integer to Scalar Single-Precision Floating-Point Value</b>	
r32 to xmmreg1	0100 0ROB 1111 0011:0000 1111:0010 1010:11 xmmreg r32
r64 to xmmreg1	0100 1ROB 1111 0011:0000 1111:0010 1010:11 xmmreg r64
mem to xmmreg	0100 0RXB 1111 0011:0000 1111:0010 1010: mod xmmreg r/m
mem64 to xmmreg	0100 1RXB 1111 0011:0000 1111:0010 1010: mod xmmreg r/m
<b>CVTSI2SD—Convert Doubleword Integer to Scalar Double-Precision Floating-Point Value</b>	
r32 to xmmreg1	0100 0ROB 1111 0010:0000 1111:0010 1010:11 xmmreg r32
r64 to xmmreg1	0100 1ROB 1111 0010:0000 1111:0010 1010:11 xmmreg r64

**Table B-34. Special Case Instructions Promoted Using REX.W (Contd.)**

Instruction and Format	Encoding
mem to xmmreg	0100 0RXB 1111 0010:0000 1111:00101 010: mod xmmreg r/m
mem64 to xmmreg	0100 1RXB 1111 0010:0000 1111:0010 1010: mod xmmreg r/m
<b>CVTSS2SI—Convert Scalar Single-Precision Floating-Point Value to Doubleword Integer</b>	
xmmreg to r32	0100 0ROB 1111 0011:0000 1111:0010 1101:11 r32 xmmreg
xmmreg to r64	0100 1ROB 1111 0011:0000 1111:0010 1101:11 r64 xmmreg
mem to r32	0100 0RXB 11110011:00001111:00101101: mod r32 r/m
mem32 to r64	0100 1RXB 1111 0011:0000 1111:0010 1101: mod r64 r/m
<b>CVTTSD2SI—Convert with Truncation Scalar Double-Precision Floating-Point Value to Doubleword Integer</b>	
xmmreg to r32	0100 0ROB 11110010:00001111:00101100:11 r32 xmmreg
xmmreg to r64	0100 1ROB 1111 0010:0000 1111:0010 1100:11 r64 xmmreg
mem64 to r32	0100 0RXB 1111 0010:0000 1111:0010 1100: mod r32 r/m
mem64 to r64	0100 1RXB 1111 0010:0000 1111:0010 1100: mod r64 r/m
<b>CVTTSS2SI—Convert with Truncation Scalar Single-Precision Floating-Point Value to Doubleword Integer</b>	
xmmreg to r32	0100 0ROB 1111 0011:0000 1111:0010 1100:11 r32 xmmreg1
xmmreg to r64	0100 1ROB 1111 0011:0000 1111:0010 1100:11 r64 xmmreg1
mem to r32	0100 0RXB 1111 0011:0000 1111:0010 1100: mod r32 r/m

**Table B-34. Special Case Instructions Promoted Using REX.W (Contd.)**

Instruction and Format	Encoding
mem32 to r64	0100 1RXB 1111 0011:0000 1111:0010 1100: mod r64 r/m
<b>MOVD/MOVQ—Move doubleword</b>	
reg to mmxreg	0100 0ROB 0000 1111:0110 1110: 11 mmxreg reg
qwordreg to mmxreg	0100 1ROB 0000 1111:0110 1110: 11 mmxreg qwordreg
reg from mmxreg	0100 0ROB 0000 1111:0111 1110: 11 mmxreg reg
qwordreg from mmxreg	0100 1ROB 0000 1111:0111 1110: 11 mmxreg qwordreg
mem to mmxreg	0100 0RXB 0000 1111:0110 1110: mod mmxreg r/m
mem64 to mmxreg	0100 1RXB 0000 1111:0110 1110: mod mmxreg r/m
mem from mmxreg	0100 0RXB 0000 1111:0111 1110: mod mmxreg r/m
mem64 from mmxreg	0100 1RXB 0000 1111:0111 1110: mod mmxreg r/m
mmxreg with memory	0100 0RXB 0000 1111:0110 01gg: mod mmxreg r/m
<b>MOVMSKPS—Extract Packed Single-Precision Floating-Point Sign Mask</b>	
xmmreg to r32	0100 0ROB 0000 1111:0101 0000:11 r32 xmmreg
xmmreg to r64	0100 1ROB 00001111:01010000:11 r64 xmmreg
<b>PEXTRW—Extract Word</b>	
mmreg to reg32, imm8	0100 0ROB 0000 1111:1100 0101:11 r32 mmreg: imm8
mmreg to reg64, imm8	0100 1ROB 0000 1111:1100 0101:11 r64 mmreg: imm8
xmmreg to reg32, imm8	0100 0ROB 0110 0110 0000 1111:1100 0101:11 r32 xmmreg: imm8
xmmreg to reg64, imm8	0100 1ROB 0110 0110 0000 1111:1100 0101:11 r64 xmmreg: imm8



**Table B-34. Special Case Instructions Promoted Using REX.W (Contd.)**

Instruction and Format	Encoding
<b>PINSRW—Insert Word</b>	
reg32 to mmreg, imm8	0100 0R0B 0000 1111:1100 0100:11 mmreg r32: imm8
reg64 to mmreg, imm8	0100 1R0B 0000 1111:1100 0100:11 mmreg r64: imm8
m16 to mmreg, imm8	0100 0R0B 0000 1111:1100 0100 mod mmreg r/m: imm8
m16 to mmreg, imm8	0100 1RXB 0000 1111:11000100 mod mmreg r/m: imm8
reg32 to xmmreg, imm8	0100 0RXB 0110 0110 0000 1111:1100 0100:11 xmmreg r32: imm8
reg64 to xmmreg, imm8	0100 0RXB 0110 0110 0000 1111:1100 0100:11 xmmreg r64: imm8
m16 to xmmreg, imm8	0100 0RXB 0110 0110 0000 1111:1100 0100 mod xmmreg r/m: imm8
m16 to xmmreg, imm8	0100 1RXB 0110 0110 0000 1111:1100 0100 mod xmmreg r/m: imm8
<b>PMOVBMSKB—Move Byte Mask To Integer</b>	
mmreg to reg32	0100 0RXB 0000 1111:1101 0111:11 r32 mmreg
mmreg to reg64	0100 1R0B 0000 1111:1101 0111:11 r64 mmreg
xmmreg to reg32	0100 0RXB 0110 0110 0000 1111:1101 0111:11 r32 mmreg
xmmreg to reg64	0110 0110 0000 1111:1101 0111:11 r64 xmmreg

## B.14 SSE4.1 FORMATS AND ENCODING TABLE

The tables in this section provide SSE4.1 formats and encodings. Some SSE4.1 instructions require a mandatory prefix (66H, F2H, F3H) as part of the three-byte opcode. These prefixes are included in the tables.

In 64-bit mode, some instructions requires REX.W, the byte sequence of REX.W prefix in the opcode sequence is shown.

Table B-35. Encodings of SSE4.1 instructions

Instruction and Format	Encoding
<b>BLENDPD — Blend Packed Double-Precision Floats</b>	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1010: 0000 1101:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1010: 0000 1101: mod xmmreg r/m
<b>BLENDPS — Blend Packed Single-Precision Floats</b>	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1010: 0000 1100:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1010: 0000 1100: mod xmmreg r/m
<b>BLENDVPD — Variable Blend Packed Double-Precision Floats</b>	
xmmreg2 to xmmreg1 <xmm0>	0110 0110:0000 1111:0011 1000: 0001 0101:11 xmmreg1 xmmreg2
mem to xmmreg <xmm0>	0110 0110:0000 1111:0011 1000: 0001 0101: mod xmmreg r/m
<b>BLENDVPS — Variable Blend Packed Single-Precision Floats</b>	
xmmreg2 to xmmreg1 <xmm0>	0110 0110:0000 1111:0011 1000: 0001 0100:11 xmmreg1 xmmreg2
mem to xmmreg <xmm0>	0110 0110:0000 1111:0011 1000: 0001 0100: mod xmmreg r/m
<b>DPPD — Packed Double-Precision Dot Products</b>	
xmmreg2 to xmmreg1, imm8	0110 0110:0000 1111:0011 1010: 0100 0001:11 xmmreg1 xmmreg2: imm8
mem to xmmreg, imm8	0110 0110:0000 1111:0011 1010: 0100 0001: mod xmmreg r/m: imm8
<b>DPPS — Packed Single-Precision Dot Products</b>	
xmmreg2 to xmmreg1, imm8	0110 0110:0000 1111:0011 1010: 0100 0000:11 xmmreg1 xmmreg2: imm8
mem to xmmreg, imm8	0110 0110:0000 1111:0011 1010: 0100 0000: mod xmmreg r/m: imm8

Table B-35. Encodings of SSE4.1 instructions

Instruction and Format	Encoding
<b>EXTRACTPS — Extract From Packed Single-Precision Floats</b>	
reg from xmmreg , imm8	0110 0110:0000 1111:0011 1010: 0001 0111:11 xmmreg reg: imm8
mem from xmmreg , imm8	0110 0110:0000 1111:0011 1010: 0001 0111: mod xmmreg r/m: imm8
<b>INSERTPS — Insert Into Packed Single-Precision Floats</b>	
xmmreg2 to xmmreg1, imm8	0110 0110:0000 1111:0011 1010: 0010 0001:11 xmmreg1 xmmreg2: imm8
mem to xmmreg, imm8	0110 0110:0000 1111:0011 1010: 0010 0001: mod xmmreg r/m: imm8
<b>MOVNTDQA — Load Double Quadword Non-temporal Aligned</b>	
m128 to xmmreg	0110 0110:0000 1111:0011 1000: 0010 1010:11 r/m xmmreg2
<b>MPSADBW — Multiple Packed Sums of Absolute Difference</b>	
xmmreg2 to xmmreg1, imm8	0110 0110:0000 1111:0011 1010: 0100 0010:11 xmmreg1 xmmreg2: imm8
mem to xmmreg, imm8	0110 0110:0000 1111:0011 1010: 0100 0010: mod xmmreg r/m: imm8
<b>PACKUSDW — Pack with Unsigned Saturation</b>	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000: 0010 1011:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0010 1011: mod xmmreg r/m
<b>PBLENDVB — Variable Blend Packed Bytes</b>	
xmmreg2 to xmmreg1 <xmm0>	0110 0110:0000 1111:0011 1000: 0001 0000:11 xmmreg1 xmmreg2
mem to xmmreg <xmm0>	0110 0110:0000 1111:0011 1000: 0001 0000: mod xmmreg r/m
<b>PBLENDW — Blend Packed Words</b>	

Table B-35. Encodings of SSE4.1 instructions

Instruction and Format	Encoding
xmmreg2 to xmmreg1, imm8	0110 0110:0000 1111:0011 1010: 0001 1110:11 xmmreg1 xmmreg2: imm8
mem to xmmreg, imm8	0110 0110:0000 1111:0011 1010: 0000 1110: mod xmmreg r/m: imm8
<b>PCMPEQQ – Compare Packed Qword Data of Equal</b>	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000: 0010 1001:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0010 1001: mod xmmreg r/m
<b>PEXTRB – Extract Byte</b>	
reg from xmmreg , imm8	0110 0110:0000 1111:0011 1010: 0001 0100:11 xmmreg reg: imm8
xmmreg to mem, imm8	0110 0110:0000 1111:0011 1010: 0001 0100: mod xmmreg r/m: imm8
<b>PEXTRD – Extract DWord</b>	
reg from xmmreg, imm8	0110 0110:0000 1111:0011 1010: 0001 0110:11 xmmreg reg: imm8
xmmreg to mem, imm8	0110 0110:0000 1111:0011 1010: 0001 0110: mod xmmreg r/m: imm8
<b>PEXTRQ – Extract QWord</b>	
r64 from xmmreg, imm8	0110 0110:REX.W:0000 1111:0011 1010: 0001 0110:11 xmmreg reg: imm8
m64 from xmmreg, imm8	0110 0110:REX.W:0000 1111:0011 1010: 0001 0110: mod xmmreg r/m: imm8
<b>PEXTRW – Extract Word</b>	
reg from xmmreg, imm8	0110 0110:0000 1111:0011 1010: 0001 0101:11 reg xmmreg: imm8
mem from xmmreg, imm8	0110 0110:0000 1111:0011 1010: 0001 0101: mod xmmreg r/m: imm8
<b>PHMINPOSUW – Packed Horizontal Word Minimum</b>	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000: 0100 0001:11 xmmreg1 xmmreg2

**Table B-35. Encodings of SSE4.1 instructions**

Instruction and Format	Encoding
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0100 0001: mod xmmreg r/m
<b>PINSRB — Extract Byte</b>	
reg to xmmreg, imm8	0110 0110:0000 1111:0011 1010: 0010 0000:11 xmmreg reg: imm8
mem to xmmreg, imm8	0110 0110:0000 1111:0011 1010: 0010 0000: mod xmmreg r/m: imm8
<b>PINSRD — Extract DWord</b>	
reg to xmmreg, imm8	0110 0110:0000 1111:0011 1010: 0010 0010:11 xmmreg reg: imm8
mem to xmmreg, imm8	0110 0110:0000 1111:0011 1010: 0010 0010: mod xmmreg r/m: imm8
<b>PINSRQ — Extract QWord</b>	
r64 to xmmreg, imm8	0110 0110:REX.W:0000 1111:0011 1010: 0010 0010:11 xmmreg reg: imm8
m64 to xmmreg, imm8	0110 0110:REX.W:0000 1111:0011 1010: 0010 0010: mod xmmreg r/m: imm8
<b>PMASB — Maximum of Packed Signed Byte Integers</b>	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000: 0011 1100:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0011 1100: mod xmmreg r/m
<b>PMASD — Maximum of Packed Signed Dword Integers</b>	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000: 0011 1101:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0011 1101: mod xmmreg r/m
<b>PMASUD — Maximum of Packed Unsigned Dword Integers</b>	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000: 0011 1111:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0011 1111: mod xmmreg r/m

Table B-35. Encodings of SSE4.1 instructions

Instruction and Format	Encoding
<b>PMAUW – Maximum of Packed Unsigned Word Integers</b>	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000: 0011 1110:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0011 1110: mod xmmreg r/m
<b>PMINSB – Minimum of Packed Signed Byte Integers</b>	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000: 0011 1000:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0011 1000: mod xmmreg r/m
<b>PMINSD – Minimum of Packed Signed Dword Integers</b>	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000: 0011 1001:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0011 1001: mod xmmreg r/m
<b>PMINUD – Minimum of Packed Unsigned Dword Integers</b>	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000: 0011 1011:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0011 1011: mod xmmreg r/m
<b>PMINUW – Minimum of Packed Unsigned Word Integers</b>	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000: 0011 1010:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0011 1010: mod xmmreg r/m
<b>PMOVSXBD – Packed Move Sign Extend - Byte to Dword</b>	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000: 0010 0001:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0010 0001: mod xmmreg r/m

Table B-35. Encodings of SSE4.1 instructions

Instruction and Format	Encoding
<b>PMOVSXBQ — Packed Move Sign Extend - Byte to Qword</b>	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000: 0010 0010:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0010 0010: mod xmmreg r/m
<b>PMOVSXBW — Packed Move Sign Extend - Byte to Word</b>	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000: 0010 0000:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0010 0000: mod xmmreg r/m
<b>PMOVSXWD — Packed Move Sign Extend - Word to Dword</b>	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000: 0010 0011:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0010 0011: mod xmmreg r/m
<b>PMOVSXWQ — Packed Move Sign Extend - Word to Qword</b>	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000: 0010 0100:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0010 0100: mod xmmreg r/m
<b>PMOVSXDQ — Packed Move Sign Extend - Dword to Qword</b>	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000: 0010 0101:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0010 0101: mod xmmreg r/m
<b>PMOVZXBQ — Packed Move Zero Extend - Byte to Dword</b>	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000: 0011 0001:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0011 0001: mod xmmreg r/m

Table B-35. Encodings of SSE4.1 instructions

Instruction and Format	Encoding
<b>PMOVZXBQ — Packed Move Zero Extend - Byte to Qword</b>	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000: 0011 0010:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0011 0010: mod xmmreg r/m
<b>PMOVZXBW — Packed Move Zero Extend - Byte to Word</b>	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000: 0011 0000:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0011 0000: mod xmmreg r/m
<b>PMOVZXWD — Packed Move Zero Extend - Word to Dword</b>	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000: 0011 0011:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0011 0011: mod xmmreg r/m
<b>PMOVZXWQ — Packed Move Zero Extend - Word to Qword</b>	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000: 0011 0100:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0011 0100: mod xmmreg r/m
<b>PMOVZXDQ — Packed Move Zero Extend - Dword to Qword</b>	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000: 0011 0101:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0011 0101: mod xmmreg r/m
<b>PMULDQ — Multiply Packed Signed Dword Integers</b>	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000: 0010 1000:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0010 1000: mod xmmreg r/m



**Table B-35. Encodings of SSE4.1 instructions**

Instruction and Format	Encoding
<b>PMULLD — Multiply Packed Signed Dword Integers, Store low Result</b>	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000: 0100 0000:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0100 0000: mod xmmreg r/m
<b>PTEST — Logical Compare</b>	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000: 0001 0111:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0001 0111: mod xmmreg r/m
<b>ROUNDPD — Round Packed Double-Precision Values</b>	
xmmreg2 to xmmreg1, imm8	0110 0110:0000 1111:0011 1010: 0000 1001:11 xmmreg1 xmmreg2: imm8
mem to xmmreg, imm8	0110 0110:0000 1111:0011 1010: 0000 1001: mod xmmreg r/m: imm8
<b>ROUNDPS — Round Packed Single-Precision Values</b>	
xmmreg2 to xmmreg1, imm8	0110 0110:0000 1111:0011 1010: 0000 1000:11 xmmreg1 xmmreg2: imm8
mem to xmmreg, imm8	0110 0110:0000 1111:0011 1010: 0000 1000: mod xmmreg r/m: imm8
<b>ROUNDSD — Round Scalar Double-Precision Value</b>	
xmmreg2 to xmmreg1, imm8	0110 0110:0000 1111:0011 1010: 0000 1011:11 xmmreg1 xmmreg2: imm8
mem to xmmreg, imm8	0110 0110:0000 1111:0011 1010: 0000 1011: mod xmmreg r/m: imm8
<b>ROUNDSS — Round Scalar Single-Precision Value</b>	
xmmreg2 to xmmreg1, imm8	0110 0110:0000 1111:0011 1010: 0000 1010:11 xmmreg1 xmmreg2: imm8
mem to xmmreg, imm8	0110 0110:0000 1111:0011 1010: 0000 1010: mod xmmreg r/m: imm8

## B.15 SSE4.2 FORMATS AND ENCODING TABLE

The tables in this section provide SSE4.2 formats and encodings. Some SSE4.2 instructions require a mandatory prefix (66H, F2H, F3H) as part of the three-byte opcode. These prefixes are included in the tables. In 64-bit mode, some instructions requires REX.W, the byte sequence of REX.W prefix in the opcode sequence is shown.

**Table B-36. Encodings of SSE4.2 instructions**

Instruction and Format	Encoding
<b>CRC32 – Accumulate CRC32</b>	
reg2 to reg1	1111 0010:0000 1111:0011 1000: 1111 000w :11 reg1 reg2
mem to reg	1111 0010:0000 1111:0011 1000: 1111 000w : mod reg r/m
bytereg2 to reg1	1111 0010:0100 WROB:0000 1111:0011 1000: 1111 0000 :11 reg1 bytereg2
m8 to reg	1111 0010:0100 WROB:0000 1111:0011 1000: 1111 0000 : mod reg r/m
qwreg2 to qwreg1	1111 0010:0100 1ROB:0000 1111:0011 1000: 1111 0000 :11 qwreg1 qwreg2
mem64 to qwreg	1111 0010:0100 1ROB:0000 1111:0011 1000: 1111 0000 : mod qwreg r/m
<b>PCMPSTR— Packed Compare Explicit-Length Strings To Index</b>	
xmmreg2 to xmmreg1, imm8	0110 0110:0000 1111:0011 1010: 0110 0001:11 xmmreg1 xmmreg2: imm8
mem to xmmreg	0110 0110:0000 1111:0011 1010: 0110 0001: mod xmmreg r/m
<b>PCMPSTRM— Packed Compare Explicit-Length Strings To Mask</b>	
xmmreg2 to xmmreg1, imm8	0110 0110:0000 1111:0011 1010: 0110 0000:11 xmmreg1 xmmreg2: imm8
mem to xmmreg	0110 0110:0000 1111:0011 1010: 0110 0000: mod xmmreg r/m
<b>PCMPISTR— Packed Compare Implicit-Length String To Index</b>	
xmmreg2 to xmmreg1, imm8	0110 0110:0000 1111:0011 1010: 0110 0011:11 xmmreg1 xmmreg2: imm8

**Table B-36. Encodings of SSE4.2 instructions**

Instruction and Format	Encoding
mem to xmmreg	0110 0110:0000 1111:0011 1010: 0110 0011: mod xmmreg r/m
<b>PCMPISTRM— Packed Compare Implicit-Length Strings To Mask</b>	
xmmreg2 to xmmreg1, imm8	0110 0110:0000 1111:0011 1010: 0110 0010:11 xmmreg1 xmmreg2: imm8
mem to xmmreg	0110 0110:0000 1111:0011 1010: 0110 0010: mod xmmreg r/m
<b>PCMPGTQ— Packed Compare Greater Than</b>	
xmmreg to xmmreg	0110 0110:0000 1111:0011 1000: 0011 0111:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0011 0111: mod xmmreg r/m
<b>POPCNT— Return Number of Bits Set to 1</b>	
reg2 to reg1	1111 0011:0000 1111:1011 1000:11 reg1 reg2
mem to reg1	1111 0011:0000 1111:1011 1000:mod reg1 r/m
qwreg2 to qwreg1	1111 0011:0100 1R0B:0000 1111:1011 1000:11 reg1 reg2
mem64 to qwreg1	1111 0011:0100 1R0B:0000 1111:1011 1000:mod reg1 r/m

## B.16 AVX FORMATS AND ENCODING TABLE

The tables in this section provide AVX formats and encodings. A mixed form of bit/hex/symbolic forms are used to express the various bytes:

The C4/C5 and opcode bytes are expressed in hex notation; the first and second payload byte of VEX, the modR/M byte is expressed in combination of bit/symbolic form. The first payload byte of C4 is expressed as combination of bits and hex form, with the hex value preceded by an underscore. The VEX bit field to encode upper register 8-15 uses 1's complement form, each of those bit field is expressed as lower case notation rxb, instead of RXB.

The hybrid bit-nibble-byte form is depicted below:

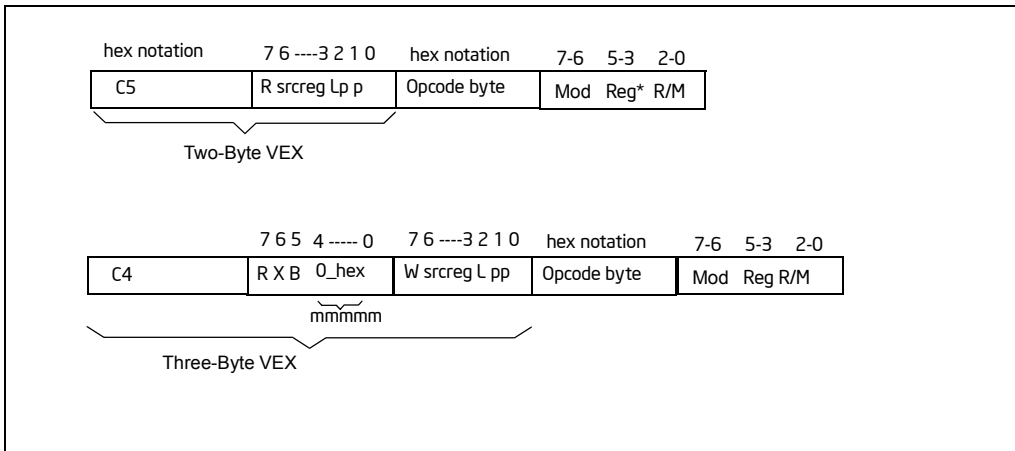


Figure B-2. Hybrid Notation of VEX-Encoded Key Instruction Bytes

Table B-37. Encodings of AVX instructions

Instruction and Format	Encoding
<b>VBLENDPD – Blend Packed Double-Precision Floats</b>	
xmmreg2 with xmmreg3 into xmmreg1	C4: rxb0_3: w xmmreg2 001:0D:11 xmmreg1 xmmreg3: imm
xmmreg2 with mem to xmmreg1	C4: rxb0_3: w xmmreg2 001:0D:mod xmmreg1 r/m: imm
ymmreg2 with ymmreg3 into ymmreg1	C4: rxb0_3: w ymmreg2 101:0D:11 ymmreg1 ymmreg3: imm
ymmreg2 with mem to ymmreg1	C4: rxb0_3: w ymmreg2 101:0D:mod ymmreg1 r/m: imm
<b>VBLENDPS – Blend Packed Single-Precision Floats</b>	
xmmreg2 with xmmreg3 into xmmreg1	C4: rxb0_3: w xmmreg2 001:0C:11 xmmreg1 xmmreg3: imm
xmmreg2 with mem to xmmreg1	C4: rxb0_3: w xmmreg2 001:0C:mod xmmreg1 r/m: imm
ymmreg2 with ymmreg3 into ymmreg1	C4: rxb0_3: w ymmreg2 101:0C:11 ymmreg1 ymmreg3: imm
ymmreg2 with mem to ymmreg1	C4: rxb0_3: w ymmreg2 101:0C:mod ymmreg1 r/m: imm
<b>VBLENDVPD – Variable Blend Packed Double-Precision Floats</b>	

INSTRUCTION FORMATS AND ENCODINGS

Instruction and Format	Encoding
xmmreg2 with xmmreg3 into xmmreg1 using xmmreg4 as mask	C4: rxb0_3: 0 xmmreg2 001:4B:11 xmmreg1 xmmreg3: xmmreg4
xmmreg2 with mem to xmmreg1 using xmmreg4 as mask	C4: rxb0_3: 0 xmmreg2 001:4B:mod xmmreg1 r/m: xmmreg4
yymmreg2 with yymmreg3 into yymmreg1 using yymmreg4 as mask	C4: rxb0_3: 0 yymmreg2 101:4B:11 yymmreg1 yymmreg3: yymmreg4
yymmreg2 with mem to yymmreg1 using yymmreg4 as mask	C4: rxb0_3: 0 yymmreg2 101:4B:mod yymmreg1 r/m: yymmreg4
<b>VBLENDVPS — Variable Blend Packed Single-Precision Floats</b>	
xmmreg2 with xmmreg3 into xmmreg1 using xmmreg4 as mask	C4: rxb0_3: 0 xmmreg2 001:4A:11 xmmreg1 xmmreg3: xmmreg4
xmmreg2 with mem to xmmreg1 using xmmreg4 as mask	C4: rxb0_3: 0 xmmreg2 001:4A:mod xmmreg1 r/m: xmmreg4
yymmreg2 with yymmreg3 into yymmreg1 using yymmreg4 as mask	C4: rxb0_3: 0 yymmreg2 101:4A:11 yymmreg1 yymmreg3: yymmreg4
yymmreg2 with mem to yymmreg1 using yymmreg4 as mask	C4: rxb0_3: 0 yymmreg2 101:4A:mod yymmreg1 r/m: yymmreg4
<b>VDPPD — Packed Double-Precision Dot Products</b>	
xmmreg2 with xmmreg3 into xmmreg1	C4: rxb0_3: w xmmreg2 001:41:11 xmmreg1 xmmreg3: imm
xmmreg2 with mem to xmmreg1	C4: rxb0_3: w xmmreg2 001:41:mod xmmreg1 r/m: imm
<b>VDPPS — Packed Single-Precision Dot Products</b>	
xmmreg2 with xmmreg3 into xmmreg1	C4: rxb0_3: w xmmreg2 001:40:11 xmmreg1 xmmreg3: imm
xmmreg2 with mem to xmmreg1	C4: rxb0_3: w xmmreg2 001:40:mod xmmreg1 r/m: imm
yymmreg2 with yymmreg3 into yymmreg1	C4: rxb0_3: w yymmreg2 101:40:11 yymmreg1 yymmreg3: imm
yymmreg2 with mem to yymmreg1	C4: rxb0_3: w yymmreg2 101:40:mod yymmreg1 r/m: imm
<b>VEEXTRACTPS — Extract From Packed Single-Precision Floats</b>	
reg from xmmreg1 using imm	C4: rxb0_3: w_F 001:17:11 xmmreg1 reg: imm

Instruction and Format	Encoding
mem from xmmreg1 using imm	C4: rxb0_3: w_F 001:17:mod xmmreg1 r/m: imm
<b>VINSERTPS – Insert Into Packed Single-Precision Floats</b>	
use imm to merge xmmreg3 with xmmreg2 into xmmreg1	C4: rxb0_3: w xmmreg2 001:21:11 xmmreg1 xmmreg3: imm
use imm to merge mem with xmmreg2 into xmmreg1	C4: rxb0_3: w xmmreg2 001:21:mod xmmreg1 r/m: imm
<b>VMOVNTDQA – Load Double Quadword Non-temporal Aligned</b>	
m128 to xmmreg1	C4: rxb0_2: w_F 001:2A:11 xmmreg1 r/m
<b>VMPSADBW – Multiple Packed Sums of Absolute Difference</b>	
xmmreg3 with xmmreg2 into xmmreg1	C4: rxb0_3: w xmmreg2 001:42:11 xmmreg1 xmmreg3: imm
m128 with xmmreg2 into xmmreg1	C4: rxb0_3: w xmmreg2 001:42:mod xmmreg1 r/m: imm
<b>VPACKUSDW – Pack with Unsigned Saturation</b>	
xmmreg3 and xmmreg2 to xmmreg1	C4: rxb0_2: w xmmreg2 001:2B:11 xmmreg1 xmmreg3: imm
m128 and xmmreg2 to xmmreg1	C4: rxb0_2: w xmmreg2 001:2B:mod xmmreg1 r/m: imm
<b>VPBLENDVB – Variable Blend Packed Bytes</b>	
xmmreg2 with xmmreg3 into xmmreg1 using xmmreg4 as mask	C4: rxb0_3: w xmmreg2 001:4C:11 xmmreg1 xmmreg3: xmmreg4
xmmreg2 with mem to xmmreg1 using xmmreg4 as mask	C4: rxb0_3: w xmmreg2 001:4C:mod xmmreg1 r/m: xmmreg4
<b>VPBLENDW – Blend Packed Words</b>	
xmmreg2 with xmmreg3 into xmmreg1	C4: rxb0_3: w xmmreg2 001:0E:11 xmmreg1 xmmreg3: imm
xmmreg2 with mem to xmmreg1	C4: rxb0_3: w xmmreg2 001:0E:mod xmmreg1 r/m: imm
<b>VPCMPEQQ – Compare Packed Qword Data of Equal</b>	

## INSTRUCTION FORMATS AND ENCODINGS

Instruction and Format	Encoding
xmmreg2 with xmmreg3 into xmmreg1	C4: rxb0_2: w xmmreg2 001:29:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_2: w xmmreg2 001:29:mod xmmreg1 r/m:
<b>VPEXTRB – Extract Byte</b>	
reg from xmmreg1 using imm	C4: rxb0_3: 0_F 001:14:11 xmmreg1 reg: imm
mem from xmmreg1 using imm	C4: rxb0_3: 0_F 001:14:mod xmmreg1 r/m: imm
<b>VPEXTRD – Extract DWord</b>	
reg from xmmreg1 using imm	C4: rxb0_3: 0_F 001:16:11 xmmreg1 reg: imm
mem from xmmreg1 using imm	C4: rxb0_3: 0_F 001:16:mod xmmreg1 r/m: imm
<b>VPEXTRQ – Extract QWord</b>	
reg from xmmreg1 using imm	C4: rxb0_3: 1_F 001:16:11 xmmreg1 reg: imm
mem from xmmreg1 using imm	C4: rxb0_3: 1_F 001:16:mod xmmreg1 r/m: imm
<b>VPEXTRW – Extract Word</b>	
reg from xmmreg1 using imm	C4: rxb0_3: 0_F 001:15:11 xmmreg1 reg: imm
mem from xmmreg1 using imm	C4: rxb0_3: 0_F 001:15:mod xmmreg1 r/m: imm
<b>VPHMINPOSUW – Packed Horizontal Word Minimum</b>	
xmmreg2 to xmmreg1	C4: rxb0_2: w_F 001:41:11 xmmreg1 xmmreg2
mem to xmmreg1	C4: rxb0_2: w_F 001:41:mod xmmreg1 r/m
<b>VPINSRB – Insert Byte</b>	
reg with xmmreg2 to xmmreg1, imm8	C4: rxb0_3: 0 xmmreg2 001:20:11 xmmreg1 reg: imm
mem with xmmreg2 to xmmreg1, imm8	C4: rxb0_3: 0 xmmreg2 001:20:mod xmmreg1 r/m: imm
<b>VPINSRD – Insert DWord</b>	
reg with xmmreg2 to xmmreg1, imm8	C4: rxb0_3: 0 xmmreg2 001:22:11 xmmreg1 reg: imm
mem with xmmreg2 to xmmreg1, imm8	C4: rxb0_3: 0 xmmreg2 001:22:mod xmmreg1 r/m: imm
<b>VPINSRQ – Insert QWord</b>	
r64 with xmmreg2 to xmmreg1, imm8	C4: rxb0_3: 1 xmmreg2 001:22:11 xmmreg1 reg: imm
m64 with xmmreg2 to xmmreg1, imm8	C4: rxb0_3: 1 xmmreg2 001:22:mod xmmreg1 r/m: imm
<b>VPMAXSB – Maximum of Packed Signed Byte Integers</b>	

Instruction and Format	Encoding
xmmreg2 with xmmreg3 into xmmreg1	C4: rxb0_2: w xmmreg2 001:3C:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_2: w xmmreg2 001:3C:mod xmmreg1 r/m
<b>VPMAXSD – Maximum of Packed Signed Dword Integers</b>	
xmmreg2 with xmmreg3 into xmmreg1	C4: rxb0_2: w xmmreg2 001:3D:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_2: w xmmreg2 001:3D:mod xmmreg1 r/m
<b>VPMAXUD – Maximum of Packed Unsigned Dword Integers</b>	
xmmreg2 with xmmreg3 into xmmreg1	C4: rxb0_2: w xmmreg2 001:3F:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_2: w xmmreg2 001:3F:mod xmmreg1 r/m
<b>VPMAXUW – Maximum of Packed Unsigned Word Integers</b>	
xmmreg2 with xmmreg3 into xmmreg1	C4: rxb0_2: w xmmreg2 001:3E:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_2: w xmmreg2 001:3E:mod xmmreg1 r/m
<b>VPMINSB – Minimum of Packed Signed Byte Integers</b>	
xmmreg2 with xmmreg3 into xmmreg1	C4: rxb0_2: w xmmreg2 001:38:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_2: w xmmreg2 001:38:mod xmmreg1 r/m
<b>VPMINSD – Minimum of Packed Signed Dword Integers</b>	
xmmreg2 with xmmreg3 into xmmreg1	C4: rxb0_2: w xmmreg2 001:39:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_2: w xmmreg2 001:39:mod xmmreg1 r/m
<b>VPMINUD – Minimum of Packed Unsigned Dword Integers</b>	
xmmreg2 with xmmreg3 into xmmreg1	C4: rxb0_2: w xmmreg2 001:3B:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_2: w xmmreg2 001:3B:mod xmmreg1 r/m
<b>VPMINUW – Minimum of Packed Unsigned Word Integers</b>	



## INSTRUCTION FORMATS AND ENCODINGS

Instruction and Format	Encoding
xmmreg2 with xmmreg3 into xmmreg1	C4: rxb0_2: w xmmreg2 001:3A:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_2: w xmmreg2 001:3A:mod xmmreg1 r/m
<b>VPMOVSXBD – Packed Move Sign Extend - Byte to Dword</b>	
xmmreg2 to xmmreg1	C4: rxb0_2: w_F 001:21:11 xmmreg1 xmmreg2
mem to xmmreg1	C4: rxb0_2: w_F 001:21:mod xmmreg1 r/m
<b>VPMOVSXBQ – Packed Move Sign Extend - Byte to Qword</b>	
xmmreg2 to xmmreg1	C4: rxb0_2: w_F 001:22:11 xmmreg1 xmmreg2
mem to xmmreg1	C4: rxb0_2: w_F 001:22:mod xmmreg1 r/m
<b>VPMOVSXBW – Packed Move Sign Extend - Byte to Word</b>	
xmmreg2 to xmmreg1	C4: rxb0_2: w_F 001:20:11 xmmreg1 xmmreg2
mem to xmmreg1	C4: rxb0_2: w_F 001:20:mod xmmreg1 r/m
<b>VPMOVSXWD – Packed Move Sign Extend - Word to Dword</b>	
xmmreg2 to xmmreg1	C4: rxb0_2: w_F 001:23:11 xmmreg1 xmmreg2
mem to xmmreg1	C4: rxb0_2: w_F 001:23:mod xmmreg1 r/m
<b>VPMOVSXWQ – Packed Move Sign Extend - Word to Qword</b>	
xmmreg2 to xmmreg1	C4: rxb0_2: w_F 001:24:11 xmmreg1 xmmreg2
mem to xmmreg1	C4: rxb0_2: w_F 001:24:mod xmmreg1 r/m
<b>VPMOVSXDQ – Packed Move Sign Extend - Dword to Qword</b>	
xmmreg2 to xmmreg1	C4: rxb0_2: w_F 001:25:11 xmmreg1 xmmreg2
mem to xmmreg1	C4: rxb0_2: w_F 001:25:mod xmmreg1 r/m
<b>VPMOVZXBBD – Packed Move Zero Extend - Byte to Dword</b>	
xmmreg2 to xmmreg1	C4: rxb0_2: w_F 001:31:11 xmmreg1 xmmreg2
mem to xmmreg1	C4: rxb0_2: w_F 001:31:mod xmmreg1 r/m
<b>VPMOVZXBQ – Packed Move Zero Extend - Byte to Qword</b>	
xmmreg2 to xmmreg1	C4: rxb0_2: w_F 001:32:11 xmmreg1 xmmreg2
mem to xmmreg1	C4: rxb0_2: w_F 001:32:mod xmmreg1 r/m

Instruction and Format	Encoding
<b>VPMOVZXBW — Packed Move Zero Extend - Byte to Word</b>	
xmmreg2 to xmmreg1	C4: rxb0_2: w_F 001:30:11 xmmreg1 xmmreg2
mem to xmmreg1	C4: rxb0_2: w_F 001:30:mod xmmreg1 r/m
<b>VPMOVZXWD — Packed Move Zero Extend - Word to Dword</b>	
xmmreg2 to xmmreg1	C4: rxb0_2: w_F 001:33:11 xmmreg1 xmmreg2
mem to xmmreg1	C4: rxb0_2: w_F 001:33:mod xmmreg1 r/m
<b>VPMOVZXWQ — Packed Move Zero Extend - Word to Qword</b>	
xmmreg2 to xmmreg1	C4: rxb0_2: w_F 001:34:11 xmmreg1 xmmreg2
mem to xmmreg1	C4: rxb0_2: w_F 001:34:mod xmmreg1 r/m
<b>VPMOVZXDQ — Packed Move Zero Extend - Dword to Qword</b>	
xmmreg2 to xmmreg1	C4: rxb0_2: w_F 001:35:11 xmmreg1 xmmreg2
mem to xmmreg1	C4: rxb0_2: w_F 001:35:mod xmmreg1 r/m
<b>VPMULDQ — Multiply Packed Signed Dword Integers</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_2: w xmmreg2 001:28:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_2: w xmmreg2 001:28:mod xmmreg1 r/m
<b>VPMULLD — Multiply Packed Signed Dword Integers, Store low Result</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_2: w xmmreg2 001:40:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_2: w xmmreg2 001:40:mod xmmreg1 r/m
<b>VPTTEST — Logical Compare</b>	
xmmreg2 to xmmreg1	C4: rxb0_2: w_F 001:17:11 xmmreg1 xmmreg2
mem to xmmreg	C4: rxb0_2: w_F 001:17:mod xmmreg1 r/m
yymmreg2 to yymmreg1	C4: rxb0_2: w_F 101:17:11 yymmreg1 yymmreg2
mem to yymmreg	C4: rxb0_2: w_F 101:17:mod yymmreg1 r/m
<b>VROUNDPD — Round Packed Double-Precision Values</b>	
xmmreg2 to xmmreg1, imm8	C4: rxb0_3: w_F 001:09:11 xmmreg1 xmmreg2: imm
mem to xmmreg1, imm8	C4: rxb0_3: w_F 001:09:mod xmmreg1 r/m: imm
yymmreg2 to yymmreg1, imm8	C4: rxb0_3: w_F 101:09:11 yymmreg1 yymmreg2: imm

## INSTRUCTION FORMATS AND ENCODINGS

Instruction and Format	Encoding
mem to ymmreg1, imm8	C4: rxb0_3: w_F 101:09:mod ymmreg1 r/m: imm
<b>VROUNDPS — Round Packed Single-Precision Values</b>	
xmmreg2 to xmmreg1, imm8	C4: rxb0_3: w_F 001:08:11 xmmreg1 xmmreg2: imm
mem to xmmreg1, imm8	C4: rxb0_3: w_F 001:08:mod xmmreg1 r/m: imm
ymmreg2 to ymmreg1, imm8	C4: rxb0_3: w_F 101:08:11 ymmreg1 ymmreg2: imm
mem to ymmreg1, imm8	C4: rxb0_3: w_F 101:08:mod ymmreg1 r/m: imm
<b>VROUNDSD — Round Scalar Double-Precision Value</b>	
xmmreg2 and xmmreg3 to xmmreg1, imm8	C4: rxb0_3: w xmmreg2 001:0B:11 xmmreg1 xmmreg3: imm
xmmreg2 and mem to xmmreg1, imm8	C4: rxb0_3: w xmmreg2 001:0B:mod xmmreg1 r/m: imm
<b>VROUNDSS — Round Scalar Single-Precision Value</b>	
xmmreg2 and xmmreg3 to xmmreg1, imm8	C4: rxb0_3: w xmmreg2 001:0A:11 xmmreg1 xmmreg3: imm
xmmreg2 and mem to xmmreg1, imm8	C4: rxb0_3: w xmmreg2 001:0A:mod xmmreg1 r/m: imm
<b>VPCMPESTRI — Packed Compare Explicit Length Strings, Return Index</b>	
xmmreg2 with xmmreg1, imm8	C4: rxb0_3: w_F 001:61:11 xmmreg1 xmmreg2: imm
mem with xmmreg1, imm8	C4: rxb0_3: w_F 001:61:mod xmmreg1 r/m: imm
<b>VPCMPESTRM — Packed Compare Explicit Length Strings, Return Mask</b>	
xmmreg2 with xmmreg1, imm8	C4: rxb0_3: w_F 001:60:11 xmmreg1 xmmreg2: imm
mem with xmmreg1, imm8	C4: rxb0_3: w_F 001:60:mod xmmreg1 r/m: imm
<b>VPCMPGTQ — Compare Packed Data for Greater Than</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_2: w xmmreg2 001:28:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_2: w xmmreg2 001:28:mod xmmreg1 r/m
<b>VPCMPISTRI — Packed Compare Implicit Length Strings, Return Index</b>	
xmmreg2 with xmmreg1, imm8	C4: rxb0_3: w_F 001:63:11 xmmreg1 xmmreg2: imm
mem with xmmreg1, imm8	C4: rxb0_3: w_F 001:63:mod xmmreg1 r/m: imm

Instruction and Format	Encoding
<b>VPCMPISTRM — Packed Compare Implicit Length Strings, Return Mask</b>	
xmmreg2 with xmmreg1, imm8	C4: rxb0_3: w_F 001:62:11 xmmreg1 xmmreg2: imm
mem with xmmreg, imm8	C4: rxb0_3: w_F 001:62:mod xmmreg1 r/m: imm
<b>VAESDEC — Perform One Round of an AES Decryption Flow</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_2: w xmmreg2 001:DE:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_2: w xmmreg2 001:DE:mod xmmreg1 r/m
<b>VAESDECLAST — Perform Last Round of an AES Decryption Flow</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_2: w xmmreg2 001:DF:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_2: w xmmreg2 001:DF:mod xmmreg1 r/m
<b>VAESEC — Perform One Round of an AES Encryption Flow</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_2: w xmmreg2 001:DC:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_2: w xmmreg2 001:DC:mod xmmreg1 r/m
<b>VAESENCLAST — Perform Last Round of an AES Encryption Flow</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_2: w xmmreg2 001:DD:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_2: w xmmreg2 001:DD:mod xmmreg1 r/m
<b>VAESIMC — Perform the AES InvMixColumn Transformation</b>	
xmmreg2 to xmmreg1	C4: rxb0_2: w_F 001:DB:11 xmmreg1 xmmreg2
mem to xmmreg1	C4: rxb0_2: w_F 001:DB:mod xmmreg1 r/m
<b>VAESKEYGENASSIST — AES Round Key Generation Assist</b>	
xmmreg2 to xmmreg1, imm8	C4: rxb0_3: w_F 001:DF:11 xmmreg1 xmmreg2: imm
mem to xmmreg, imm8	C4: rxb0_3: w_F 001:DF:mod xmmreg1 r/m: imm
<b>VPABSB — Packed Absolute Value</b>	
xmmreg2 to xmmreg1	C4: rxb0_2: w_F 001:1C:11 xmmreg1 xmmreg2
mem to xmmreg1	C4: rxb0_2: w_F 001:1C:mod xmmreg1 r/m
<b>VPABSD — Packed Absolute Value</b>	
xmmreg2 to xmmreg1	C4: rxb0_2: w_F 001:1E:11 xmmreg1 xmmreg2
mem to xmmreg1	C4: rxb0_2: w_F 001:1E:mod xmmreg1 r/m

INSTRUCTION FORMATS AND ENCODINGS

Instruction and Format	Encoding
<b>VPABSW — Packed Absolute Value</b>	
xmmreg2 to xmmreg1	C4: rxb0_2: w_F 001:1D:11 xmmreg1 xmmreg2
mem to xmmreg1	C4: rxb0_2: w_F 001:1D:mod xmmreg1 r/m
<b>VPALIGNR — Packed Align Right</b>	
xmmreg2 with xmmreg3 to xmmreg1, imm8	C4: rxb0_3: w xmmreg2 001:DD:11 xmmreg1 xmmreg3: imm
xmmreg2 with mem to xmmreg1, imm8	C4: rxb0_3: w xmmreg2 001:DD:mod xmmreg1 r/m: imm
<b>VPHADD — Packed Horizontal Add</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_2: w xmmreg2 001:02:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_2: w xmmreg2 001:02:mod xmmreg1 r/m
<b>VPHADDW — Packed Horizontal Add</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_2: w xmmreg2 001:01:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_2: w xmmreg2 001:01:mod xmmreg1 r/m
<b>VPHADDSW — Packed Horizontal Add and Saturate</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_2: w xmmreg2 001:03:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_2: w xmmreg2 001:03:mod xmmreg1 r/m
<b>VPHSUBD — Packed Horizontal Subtract</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_2: w xmmreg2 001:06:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_2: w xmmreg2 001:06:mod xmmreg1 r/m
<b>VPHSUBW — Packed Horizontal Subtract</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_2: w xmmreg2 001:05:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_2: w xmmreg2 001:05:mod xmmreg1 r/m
<b>VPHSUBSW — Packed Horizontal Subtract and Saturate</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_2: w xmmreg2 001:07:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_2: w xmmreg2 001:07:mod xmmreg1 r/m
<b>VPMADDUBSW — Multiply and Add Packed Signed and Unsigned Bytes</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_2: w xmmreg2 001:04:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_2: w xmmreg2 001:04:mod xmmreg1 r/m

Instruction and Format	Encoding
<b>VPMULHRW — Packed Multiply High with Round and Scale</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_2: w xmmreg2 001:0B:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_2: w xmmreg2 001:0B:mod xmmreg1 r/m
<b>VPSHUFB — Packed Shuffle Bytes</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_2: w xmmreg2 001:00:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_2: w xmmreg2 001:00:mod xmmreg1 r/m
<b>VPSIGNB — Packed SIGN</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_2: w xmmreg2 001:08:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_2: w xmmreg2 001:08:mod xmmreg1 r/m
<b>VPSIGND — Packed SIGN</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_2: w xmmreg2 001:0A:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_2: w xmmreg2 001:0A:mod xmmreg1 r/m
<b>VPSIGNW — Packed SIGN</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_2: w xmmreg2 001:09:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_2: w xmmreg2 001:09:mod xmmreg1 r/m
<b>VADDSUBPD — Packed Double-FP Add/Subtract</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:D0:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:D0:mod xmmreg1 r/m
xmmreglo2 <sup>1</sup> with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:D0:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:D0:mod xmmreg1 r/m
yymmreg2 with yymmreg3 to yymmreg1	C4: rxb0_1: w yymmreg2 101:D0:11 yymmreg1 yymmreg3
yymmreg2 with mem to yymmreg1	C4: rxb0_1: w yymmreg2 101:D0:mod yymmreg1 r/m
yymmreglo2 with yymmreglo3 to yymmreg1	C5: r_yymmreglo2 101:D0:11 yymmreg1 yymmreglo3
yymmreglo2 with mem to yymmreg1	C5: r_yymmreglo2 101:D0:mod yymmreg1 r/m
<b>VADDSUBPS — Packed Single-FP Add/Subtract</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 011:D0:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 011:D0:mod xmmreg1 r/m

INSTRUCTION FORMATS AND ENCODINGS

Instruction and Format	Encoding
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 011:D0:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 011:D0:mod xmmreg1 r/m
ymmreg2 with ymmreg3 to ymmreg1	C4: rxb0_1: w ymmreg2 111:D0:11 ymmreg1 ymmreg3
ymmreg2 with mem to ymmreg1	C4: rxb0_1: w ymmreg2 111:D0:mod ymmreg1 r/m
ymmreglo2 with ymmreglo3 to ymmreg1	C5: r_ymmreglo2 111:D0:11 ymmreg1 ymmreglo3
ymmreglo2 with mem to ymmreg1	C5: r_ymmreglo2 111:D0:mod ymmreg1 r/m
<b>VHADDPD – Packed Double-FP Horizontal Add</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:7C:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:7C:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:7C:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:7C:mod xmmreg1 r/m
ymmreg2 with ymmreg3 to ymmreg1	C4: rxb0_1: w ymmreg2 101:7C:11 ymmreg1 ymmreg3
ymmreg2 with mem to ymmreg1	C4: rxb0_1: w ymmreg2 101:7C:mod ymmreg1 r/m
ymmreglo2 with ymmreglo3 to ymmreg1	C5: r_ymmreglo2 101:7C:11 ymmreg1 ymmreglo3
ymmreglo2 with mem to ymmreg1	C5: r_ymmreglo2 101:7C:mod ymmreg1 r/m
<b>VHADDPS – Packed Single-FP Horizontal Add</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 011:7C:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 011:7C:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 011:7C:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 011:7C:mod xmmreg1 r/m
ymmreg2 with ymmreg3 to ymmreg1	C4: rxb0_1: w ymmreg2 111:7C:11 ymmreg1 ymmreg3
ymmreg2 with mem to ymmreg1	C4: rxb0_1: w ymmreg2 111:7C:mod ymmreg1 r/m
ymmreglo2 with ymmreglo3 to ymmreg1	C5: r_ymmreglo2 111:7C:11 ymmreg1 ymmreglo3
ymmreglo2 with mem to ymmreg1	C5: r_ymmreglo2 111:7C:mod ymmreg1 r/m
<b>VHSUBPD – Packed Double-FP Horizontal Subtract</b>	

Instruction and Format	Encoding
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:7D:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:7D:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:7D:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:7D:mod xmmreg1 r/m
yymmreg2 with yymmreg3 to yymmreg1	C4: rxb0_1: w yymmreg2 101:7D:11 yymmreg1 yymmreg3
yymmreg2 with mem to yymmreg1	C4: rxb0_1: w yymmreg2 101:7D:mod yymmreg1 r/m
yymmreglo2 with yymmreglo3 to yymmreg1	C5: r_yymmreglo2 101:7D:11 yymmreg1 yymmreglo3
yymmreglo2 with mem to yymmreg1	C5: r_yymmreglo2 101:7D:mod yymmreg1 r/m
<b>VHSUBPS – Packed Single-FP Horizontal Subtract</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 011:7D:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 011:7D:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 011:7D:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 011:7D:mod xmmreg1 r/m
yymmreg2 with yymmreg3 to yymmreg1	C4: rxb0_1: w yymmreg2 111:7D:11 yymmreg1 yymmreg3
yymmreg2 with mem to yymmreg1	C4: rxb0_1: w yymmreg2 111:7D:mod yymmreg1 r/m
yymmreglo2 with yymmreglo3 to yymmreg1	C5: r_yymmreglo2 111:7D:11 yymmreg1 yymmreglo3
yymmreglo2 with mem to yymmreg1	C5: r_yymmreglo2 111:7D:mod yymmreg1 r/m
<b>VLDDQU – Load Unaligned Integer 128 Bits</b>	
mem to xmmreg1	C4: rxb0_1: w_F 011:F0:mod xmmreg1 r/m
mem to xmmreg1	C5: r_F 011:F0:mod xmmreg1 r/m
mem to yymmreg1	C4: rxb0_1: w_F 111:F0:mod yymmreg1 r/m
mem to yymmreg1	C5: r_F 111:F0:mod yymmreg1 r/m
<b>VMOVDDUP – Move One Double-FP and Duplicate</b>	
xmmreg2 to xmmreg1	C4: rxb0_1: w_F 011:12:11 xmmreg1 xmmreg2
mem to xmmreg1	C4: rxb0_1: w_F 011:12:mod xmmreg1 r/m
xmmreglo to xmmreg1	C5: r_F 011:12:11 xmmreg1 xmmreglo



## INSTRUCTION FORMATS AND ENCODINGS

Instruction and Format	Encoding
mem to xmmreg1	C5: r_F 011:12:mod xmmreg1 r/m
yymmreg2 to yymmreg1	C4: rxb0_1: w_F 111:12:11 yymmreg1 yymmreg2
mem to yymmreg1	C4: rxb0_1: w_F 111:12:mod yymmreg1 r/m
yymmreglo to yymmreg1	C5: r_F 111:12:11 yymmreg1 yymmreglo
mem to yymmreg1	C5: r_F 111:12:mod yymmreg1 r/m
<b>VMOVHLPS – Move Packed Single-Precision Floating-Point Values High to Low</b>	
xmmreg2 and xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 000:12:11 xmmreg1 xmmreg3
xmmreglo2 and xmmreglo3 to xmmreg1	C5: r_xmmreglo2 000:12:11 xmmreg1 xmmreglo3
<b>VMOVSHDUP – Move Packed Single-FP High and Duplicate</b>	
xmmreg2 to xmmreg1	C4: rxb0_1: w_F 010:16:11 xmmreg1 xmmreg2
mem to xmmreg1	C4: rxb0_1: w_F 010:16:mod xmmreg1 r/m
xmmreglo to xmmreg1	C5: r_F 010:16:11 xmmreg1 xmmreglo
mem to xmmreg1	C5: r_F 010:16:mod xmmreg1 r/m
yymmreg2 to yymmreg1	C4: rxb0_1: w_F 110:16:11 yymmreg1 yymmreg2
mem to yymmreg1	C4: rxb0_1: w_F 110:16:mod yymmreg1 r/m
yymmreglo to yymmreg1	C5: r_F 110:16:11 yymmreg1 yymmreglo
mem to yymmreg1	C5: r_F 110:16:mod yymmreg1 r/m
<b>VMOVSLDUP – Move Packed Single-FP Low and Duplicate</b>	
xmmreg2 to xmmreg1	C4: rxb0_1: w_F 010:12:11 xmmreg1 xmmreg2
mem to xmmreg1	C4: rxb0_1: w_F 010:12:mod xmmreg1 r/m
xmmreglo to xmmreg1	C5: r_F 010:12:11 xmmreg1 xmmreglo
mem to xmmreg1	C5: r_F 010:12:mod xmmreg1 r/m
yymmreg2 to yymmreg1	C4: rxb0_1: w_F 110:12:11 yymmreg1 yymmreg2
mem to yymmreg1	C4: rxb0_1: w_F 110:12:mod yymmreg1 r/m
yymmreglo to yymmreg1	C5: r_F 110:12:11 yymmreg1 yymmreglo
mem to yymmreg1	C5: r_F 110:12:mod yymmreg1 r/m
<b>VADDPD – Add Packed Double-Precision Floating-Point Values</b>	

Instruction and Format	Encoding
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:58:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:58:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:58:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:58:mod xmmreg1 r/m
ymmreg2 with ymmreg3 to ymmreg1	C4: rxb0_1: w ymmreg2 101:58:11 ymmreg1 ymmreg3
ymmreg2 with mem to ymmreg1	C4: rxb0_1: w ymmreg2 101:58:mod ymmreg1 r/m
ymmreglo2 with ymmreglo3 to ymmreg1	C5: r_ymmreglo2 101:58:11 ymmreg1 ymmreglo3
ymmreglo2 with mem to ymmreg1	C5: r_ymmreglo2 101:58:mod ymmreg1 r/m
<b>VADDSD – Add Scalar Double-Precision Floating-Point Values</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 011:58:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 011:58:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 011:58:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5 r_xmmreglo2 011:58:mod xmmreg1 r/m
<b>VANDPD – Bitwise Logical AND of Packed Double-Precision Floating-Point Values</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:54:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:54:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:54:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:54:mod xmmreg1 r/m
ymmreg2 with ymmreg3 to ymmreg1	C4: rxb0_1: w ymmreg2 101:54:11 ymmreg1 ymmreg3
ymmreg2 with mem to ymmreg1	C4: rxb0_1: w ymmreg2 101:54:mod ymmreg1 r/m
ymmreglo2 with ymmreglo3 to ymmreg1	C5: r_ymmreglo2 101:54:11 ymmreg1 ymmreglo3
ymmreglo2 with mem to ymmreg1	C5: r_ymmreglo2 101:54:mod ymmreg1 r/m
<b>VANDNPD – Bitwise Logical AND NOT of Packed Double-Precision Floating-Point Values</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:55:11 xmmreg1 xmmreg3

## INSTRUCTION FORMATS AND ENCODINGS

Instruction and Format	Encoding
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:55:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:55:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:55:mod xmmreg1 r/m
yymmreg2 with yymmreg3 to yymmreg1	C4: rxb0_1: w yymmreg2 101:55:11 yymmreg1 yymmreg3
yymmreg2 with mem to yymmreg1	C4: rxb0_1: w yymmreg2 101:55:mod yymmreg1 r/m
yymmreglo2 with yymmreglo3 to yymmreg1	C5: r_yymmreglo2 101:55:11 yymmreg1 yymmreglo3
yymmreglo2 with mem to yymmreg1	C5: r_yymmreglo2 101:55:mod yymmreg1 r/m
<b>VCMPD — Compare Packed Double-Precision Floating-Point Values</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:C2:11 xmmreg1 xmmreg3: imm
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:C2:mod xmmreg1 r/m: imm
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:C2:11 xmmreg1 xmmreglo3: imm
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:C2:mod xmmreg1 r/m: imm
yymmreg2 with yymmreg3 to yymmreg1	C4: rxb0_1: w yymmreg2 101:C2:11 yymmreg1 yymmreg3: imm
yymmreg2 with mem to yymmreg1	C4: rxb0_1: w yymmreg2 101:C2:mod yymmreg1 r/m: imm
yymmreglo2 with yymmreglo3 to yymmreg1	C5: r_yymmreglo2 101:C2:11 yymmreg1 yymmreglo3: imm
yymmreglo2 with mem to yymmreg1	C5: r_yymmreglo2 101:C2:mod yymmreg1 r/m: imm
<b>VCMPD — Compare Scalar Double-Precision Floating-Point Values</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 011:C2:11 xmmreg1 xmmreg3: imm
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 011:C2:mod xmmreg1 r/m: imm
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 011:C2:11 xmmreg1 xmmreglo3: imm
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 011:C2:mod xmmreg1 r/m: imm
<b>VCMPD — Compare Scalar Ordered Double-Precision Floating-Point Values and Set EFLAGS</b>	
xmmreg2 to xmmreg1	C4: rxb0_1: w_F 001:2F:11 xmmreg1 xmmreg2

Instruction and Format	Encoding
mem to xmmreg1	C4: rxb0_1: w_F 001:2F:mod xmmreg1 r/m
xmmreglo to xmmreg1	C5: r_F 001:2F:11 xmmreg1 xmmreglo
mem to xmmreg1	C5: r_F 001:2F:mod xmmreg1 r/m
<b>VCVTDQ2PD— Convert Packed Dword Integers to Packed Double-Precision FP Values</b>	
xmmreg2 to xmmreg1	C4: rxb0_1: w_F 010:E6:11 xmmreg1 xmmreg2
mem to xmmreg1	C4: rxb0_1: w_F 010:E6:mod xmmreg1 r/m
xmmreglo to xmmreg1	C5: r_F 010:E6:11 xmmreg1 xmmreglo
mem to xmmreg1	C5: r_F 010:E6:mod xmmreg1 r/m
yymmreg2 to yymmreg1	C4: rxb0_1: w_F 110:E6:11 yymmreg1 yymmreg2
mem to yymmreg1	C4: rxb0_1: w_F 110:E6:mod yymmreg1 r/m
yymmreglo to yymmreg1	C5: r_F 110:E6:11 yymmreg1 yymmreglo
mem to yymmreg1	C5: r_F 110:E6:mod yymmreg1 r/m
<b>VCVTDQ2PS— Convert Packed Dword Integers to Packed Single-Precision FP Values</b>	
xmmreg2 to xmmreg1	C4: rxb0_1: w_F 000:5B:11 xmmreg1 xmmreg2
mem to xmmreg1	C4: rxb0_1: w_F 000:5B:mod xmmreg1 r/m
xmmreglo to xmmreg1	C5: r_F 000:5B:11 xmmreg1 xmmreglo
mem to xmmreg1	C5: r_F 000:5B:mod xmmreg1 r/m
yymmreg2 to yymmreg1	C4: rxb0_1: w_F 100:5B:11 yymmreg1 yymmreg2
mem to yymmreg1	C4: rxb0_1: w_F 100:5B:mod yymmreg1 r/m
yymmreglo to yymmreg1	C5: r_F 100:5B:11 yymmreg1 yymmreglo
mem to yymmreg1	C5: r_F 100:5B:mod yymmreg1 r/m
<b>VCVTPD2DQ— Convert Packed Double-Precision FP Values to Packed Dword Integers</b>	
xmmreg2 to xmmreg1	C4: rxb0_1: w_F 011:E6:11 xmmreg1 xmmreg2
mem to xmmreg1	C4: rxb0_1: w_F 011:E6:mod xmmreg1 r/m
xmmreglo to xmmreg1	C5: r_F 011:E6:11 xmmreg1 xmmreglo
mem to xmmreg1	C5: r_F 011:E6:mod xmmreg1 r/m
yymmreg2 to yymmreg1	C4: rxb0_1: w_F 111:E6:11 yymmreg1 yymmreg2

## INSTRUCTION FORMATS AND ENCODINGS

Instruction and Format	Encoding
mem to ymmreg1	C4: rxb0_1: w_F 111:E6:mod ymmreg1 r/m
ymmreglo to ymmreg1	C5: r_F 111:E6:11 ymmreg1 ymmreglo
mem to ymmreg1	C5: r_F 111:E6:mod ymmreg1 r/m
<b>VCVTPD2PS— Convert Packed Double-Precision FP Values to Packed Single-Precision FP Values</b>	
xmmreg2 to xmmreg1	C4: rxb0_1: w_F 001:5A:11 xmmreg1 xmmreg2
mem to xmmreg1	C4: rxb0_1: w_F 001:5A:mod xmmreg1 r/m
xmmreglo to xmmreg1	C5: r_F 001:5A:11 xmmreg1 xmmreglo
mem to xmmreg1	C5: r_F 001:5A:mod xmmreg1 r/m
ymmreg2 to ymmreg1	C4: rxb0_1: w_F 101:5A:11 ymmreg1 ymmreg2
mem to ymmreg1	C4: rxb0_1: w_F 101:5A:mod ymmreg1 r/m
ymmreglo to ymmreg1	C5: r_F 101:5A:11 ymmreg1 ymmreglo
mem to ymmreg1	C5: r_F 101:5A:mod ymmreg1 r/m
<b>VCVTPS2DQ— Convert Packed Single-Precision FP Values to Packed Dword Integers</b>	
xmmreg2 to xmmreg1	C4: rxb0_1: w_F 001:5B:11 xmmreg1 xmmreg2
mem to xmmreg1	C4: rxb0_1: w_F 001:5B:mod xmmreg1 r/m
xmmreglo to xmmreg1	C5: r_F 001:5B:11 xmmreg1 xmmreglo
mem to xmmreg1	C5: r_F 001:5B:mod xmmreg1 r/m
ymmreg2 to ymmreg1	C4: rxb0_1: w_F 101:5B:11 ymmreg1 ymmreg2
mem to ymmreg1	C4: rxb0_1: w_F 101:5B:mod ymmreg1 r/m
ymmreglo to ymmreg1	C5: r_F 101:5B:11 ymmreg1 ymmreglo
mem to ymmreg1	C5: r_F 101:5B:mod ymmreg1 r/m
<b>VCVTPS2PD— Convert Packed Single-Precision FP Values to Packed Double-Precision FP Values</b>	
xmmreg2 to xmmreg1	C4: rxb0_1: w_F 000:5A:11 xmmreg1 xmmreg2
mem to xmmreg1	C4: rxb0_1: w_F 000:5A:mod xmmreg1 r/m
xmmreglo to xmmreg1	C5: r_F 000:5A:11 xmmreg1 xmmreglo
mem to xmmreg1	C5: r_F 000:5A:mod xmmreg1 r/m
ymmreg2 to ymmreg1	C4: rxb0_1: w_F 100:5A:11 ymmreg1 ymmreg2

Instruction and Format	Encoding
mem to ymmreg1	C4: rxb0_1: w_F 100:5A:mod ymmreg1 r/m
ymmreglo to ymmreg1	C5: r_F 100:5A:11 ymmreg1 ymmreglo
mem to ymmreg1	C5: r_F 100:5A:mod ymmreg1 r/m
<b>VCVTSD2SI— Convert Scalar Double-Precision FP Value to Integer</b>	
xmmreg1 to reg32	C4: rxb0_1: 0_F 011:2D:11 reg xmmreg1
mem to reg32	C4: rxb0_1: 0_F 011:2D:mod reg r/m
xmmreglo to reg32	C5: r_F 011:2D:11 reg xmmreglo
mem to reg32	C5: r_F 011:2D:mod reg r/m
ymmreg1 to reg64	C4: rxb0_1: 1_F 111:2D:11 reg ymmreg1
mem to reg64	C4: rxb0_1: 1_F 111:2D:mod reg r/m
<b>VCVTSD2SS — Convert Scalar Double-Precision FP Value to Scalar Single-Precision FP Value</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 011:5A:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 011:5A:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 011:5A:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 011:5A:mod xmmreg1 r/m
<b>VCVTSI2SD— Convert Dword Integer to Scalar Double-Precision FP Value</b>	
xmmreg2 with reg to xmmreg1	C4: rxb0_1: 0 xmmreg2 011:2A:11 xmmreg1 reg
xmmreg2 with mem to xmmreg1	C4: rxb0_1: 0 xmmreg2 011:2A:mod xmmreg1 r/m
xmmreglo2 with reglo to xmmreg1	C5: r_xmmreglo2 011:2A:11 xmmreg1 reglo
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 011:2A:mod xmmreg1 r/m
ymmreg2 with reg to ymmreg1	C4: rxb0_1: 1 ymmreg2 111:2A:11 ymmreg1 reg
ymmreg2 with mem to ymmreg1	C4: rxb0_1: 1 ymmreg2 111:2A:mod ymmreg1 r/m
<b>VCVTSS2SD — Convert Scalar Single-Precision FP Value to Scalar Double-Precision FP Value</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 010:5A:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 010:5A:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 010:5A:11 xmmreg1 xmmreglo3

INSTRUCTION FORMATS AND ENCODINGS

Instruction and Format	Encoding
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 010:5A:mod xmmreg1 r/m
<b>VCVTPD2DQ— Convert with Truncation Packed Double-Precision FP Values to Packed Dword Integers</b>	
xmmreg2 to xmmreg1	C4: rxb0_1: w_F 001:E6:11 xmmreg1 xmmreg2
mem to xmmreg1	C4: rxb0_1: w_F 001:E6:mod xmmreg1 r/m
xmmreglo to xmmreg1	C5: r_F 001:E6:11 xmmreg1 xmmreglo
mem to xmmreg1	C5: r_F 001:E6:mod xmmreg1 r/m
yymmreg2 to yymmreg1	C4: rxb0_1: w_F 101:E6:11 yymmreg1 yymmreg2
mem to yymmreg1	C4: rxb0_1: w_F 101:E6:mod yymmreg1 r/m
yymmreglo to yymmreg1	C5: r_F 101:E6:11 yymmreg1 yymmreglo
mem to yymmreg1	C5: r_F 101:E6:mod yymmreg1 r/m
<b>VCVTTPS2DQ— Convert with Truncation Packed Single-Precision FP Values to Packed Dword Integers</b>	
xmmreg2 to xmmreg1	C4: rxb0_1: w_F 010:5B:11 xmmreg1 xmmreg2
mem to xmmreg1	C4: rxb0_1: w_F 010:5B:mod xmmreg1 r/m
xmmreglo to xmmreg1	C5: r_F 010:5B:11 xmmreg1 xmmreglo
mem to xmmreg1	C5: r_F 010:5B:mod xmmreg1 r/m
yymmreg2 to yymmreg1	C4: rxb0_1: w_F 110:5B:11 yymmreg1 yymmreg2
mem to yymmreg1	C4: rxb0_1: w_F 110:5B:mod yymmreg1 r/m
yymmreglo to yymmreg1	C5: r_F 110:5B:11 yymmreg1 yymmreglo
mem to yymmreg1	C5: r_F 110:5B:mod yymmreg1 r/m
<b>VCVTSD2SI— Convert with Truncation Scalar Double-Precision FP Value to Signed Integer</b>	
xmmreg1 to reg32	C4: rxb0_1: 0_F 011:2C:11 reg xmmreg1
mem to reg32	C4: rxb0_1: 0_F 011:2C:mod reg r/m
xmmreglo to reg32	C5: r_F 011:2C:11 reg xmmreglo
mem to reg32	C5: r_F 011:2C:mod reg r/m
xmmreg1 to reg64	C4: rxb0_1: 1_F 011:2C:11 reg xmmreg1
mem to reg64	C4: rxb0_1: 1_F 011:2C:mod reg r/m

Instruction and Format	Encoding
<b>VDIVPD — Divide Packed Double-Precision Floating-Point Values</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:5E:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:5E:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:5E:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:5E:mod xmmreg1 r/m
yymmreg2 with yymmreg3 to yymmreg1	C4: rxb0_1: w yymmreg2 101:5E:11 yymmreg1 yymmreg3
yymmreg2 with mem to yymmreg1	C4: rxb0_1: w yymmreg2 101:5E:mod yymmreg1 r/m
yymmreglo2 with yymmreglo3 to yymmreg1	C5: r_yymmreglo2 101:5E:11 yymmreg1 yymmreglo3
yymmreglo2 with mem to yymmreg1	C5: r_yymmreglo2 101:5E:mod yymmreg1 r/m
<b>VDIVSD — Divide Scalar Double-Precision Floating-Point Values</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 011:5E:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 011:5E:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 011:5E:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 011:5E:mod xmmreg1 r/m
<b>VMASKMOVDQU— Store Selected Bytes of Double Quadword</b>	
xmmreg1 to mem; xmmreg2 as mask	C4: rxb0_1: w_F 001:F7:11 r/m xmmreg1: xmmreg2
xmmreg1 to mem; xmmreg2 as mask	C5: r_F 001:F7:11 r/m xmmreg1: xmmreg2
<b>VMAXPD — Return Maximum Packed Double-Precision Floating-Point Values</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:5F:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:5F:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:5F:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:5F:mod xmmreg1 r/m
yymmreg2 with yymmreg3 to yymmreg1	C4: rxb0_1: w yymmreg2 101:5F:11 yymmreg1 yymmreg3
yymmreg2 with mem to yymmreg1	C4: rxb0_1: w yymmreg2 101:5F:mod yymmreg1 r/m



## INSTRUCTION FORMATS AND ENCODINGS

Instruction and Format	Encoding
yymmreglo2 with yymmreglo3 to yymmreg1	C5: r_yymmreglo2 101:5F:11 yymmreg1 yymmreglo3
yymmreglo2 with mem to yymmreg1	C5: r_yymmreglo2 101:5F:mod yymmreg1 r/m
<b>VMAXSD – Return Maximum Scalar Double-Precision Floating-Point Value</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 011:5F:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 011:5F:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 011:5F:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 011:5F:mod xmmreg1 r/m
<b>VMINPD – Return Minimum Packed Double-Precision Floating-Point Values</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:5D:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:5D:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:5D:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:5D:mod xmmreg1 r/m
yymmreg2 with yymmreg3 to yymmreg1	C4: rxb0_1: w yymmreg2 101:5D:11 yymmreg1 yymmreg3
yymmreg2 with mem to yymmreg1	C4: rxb0_1: w yymmreg2 101:5D:mod yymmreg1 r/m
yymmreglo2 with yymmreglo3 to yymmreg1	C5: r_yymmreglo2 101:5D:11 yymmreg1 yymmreglo3
yymmreglo2 with mem to yymmreg1	C5: r_yymmreglo2 101:5D:mod yymmreg1 r/m
<b>VMINSD – Return Minimum Scalar Double-Precision Floating-Point Value</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 011:5D:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 011:5D:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 011:5D:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 011:5D:mod xmmreg1 r/m
<b>VMOVAPD – Move Aligned Packed Double-Precision Floating-Point Values</b>	
xmmreg2 to xmmreg1	C4: rxb0_1: w_F 001:28:11 xmmreg1 xmmreg2
mem to xmmreg1	C4: rxb0_1: w_F 001:28:mod xmmreg1 r/m

Instruction and Format	Encoding
xmmreglo to xmmreg1	C5: r_F 001:28:11 xmmreg1 xmmreglo
mem to xmmreg1	C5: r_F 001:28:mod xmmreg1 r/m
xmmreg1 to xmmreg2	C4: rxb0_1: w_F 001:29:11 xmmreg2 xmmreg1
xmmreg1 to mem	C4: rxb0_1: w_F 001:29:mod r/m xmmreg1
xmmreg1 to xmmreglo	C5: r_F 001:29:11 xmmreglo xmmreg1
xmmreg1 to mem	C5: r_F 001:29:mod r/m xmmreg1
yymmreg2 to yymmreg1	C4: rxb0_1: w_F 101:28:11 yymmreg1 yymmreg2
mem to yymmreg1	C4: rxb0_1: w_F 101:28:mod yymmreg1 r/m
yymmreglo to yymmreg1	C5: r_F 101:28:11 yymmreg1 yymmreglo
mem to yymmreg1	C5: r_F 101:28:mod yymmreg1 r/m
yymmreg1 to yymmreg2	C4: rxb0_1: w_F 101:29:11 yymmreg2 yymmreg1
yymmreg1 to mem	C4: rxb0_1: w_F 101:29:mod r/m yymmreg1
yymmreg1 to yymmreglo	C5: r_F 101:29:11 yymmreglo yymmreg1
yymmreg1 to mem	C5: r_F 101:29:mod r/m yymmreg1
<b>VMOVD — Move Doubleword</b>	
reg32 to xmmreg1	C4: rxb0_1: 0_F 001:6E:11 xmmreg1 reg32
mem32 to xmmreg1	C4: rxb0_1: 0_F 001:6E:mod xmmreg1 r/m
reg32 to xmmreg1	C5: r_F 001:6E:11 xmmreg1 reg32
mem32 to xmmreg1	C5: r_F 001:6E:mod xmmreg1 r/m
xmmreg1 to reg32	C4: rxb0_1: 0_F 001:7E:11 reg32 xmmreg1
xmmreg1 to mem32	C4: rxb0_1: 0_F 001:7E:mod mem32 xmmreg1
xmmreglo to reg32	C5: r_F 001:7E:11 reg32 xmmreglo
xmmreglo to mem32	C5: r_F 001:7E:mod mem32 xmmreglo
<b>MOVQ — Move Quadword</b>	
reg64 to xmmreg1	C4: rxb0_1: 1_F 001:6E:11 xmmreg1 reg64
mem64 to xmmreg1	C4: rxb0_1: 1_F 001:6E:mod xmmreg1 r/m
xmmreg1 to reg64	C4: rxb0_1: 1_F 001:7E:11 reg64 xmmreg1
xmmreg1 to mem64	C4: rxb0_1: 1_F 001:7E:mod r/m xmmreg1
<b>VMOVDQA — Move Aligned Double Quadword</b>	
xmmreg2 to xmmreg1	C4: rxb0_1: w_F 001:6F:11 xmmreg1 xmmreg2
mem to xmmreg1	C4: rxb0_1: w_F 001:6F:mod xmmreg1 r/m

## INSTRUCTION FORMATS AND ENCODINGS

Instruction and Format	Encoding
xmmreg0 to xmmreg1	C5: r_F 001:6F:11 xmmreg1 xmmreg0
mem to xmmreg1	C5: r_F 001:6F:mod xmmreg1 r/m
xmmreg1 to xmmreg2	C4: rxb0_1: w_F 001:7F:11 xmmreg2 xmmreg1
xmmreg1 to mem	C4: rxb0_1: w_F 001:7F:mod r/m xmmreg1
xmmreg1 to xmmreg0	C5: r_F 001:7F:11 xmmreg0 xmmreg1
xmmreg1 to mem	C5: r_F 001:7F:mod r/m xmmreg1
yymmreg2 to yymmreg1	C4: rxb0_1: w_F 101:6F:11 yymmreg1 yymmreg2
mem to yymmreg1	C4: rxb0_1: w_F 101:6F:mod yymmreg1 r/m
yymmreg0 to yymmreg1	C5: r_F 101:6F:11 yymmreg1 yymmreg0
mem to yymmreg1	C5: r_F 101:6F:mod yymmreg1 r/m
yymmreg1 to yymmreg2	C4: rxb0_1: w_F 101:7F:11 yymmreg2 yymmreg1
yymmreg1 to mem	C4: rxb0_1: w_F 101:7F:mod r/m yymmreg1
yymmreg1 to yymmreg0	C5: r_F 101:7F:11 yymmreg0 yymmreg1
yymmreg1 to mem	C5: r_F 101:7F:mod r/m yymmreg1
<b>VMOVDQU — Move Unaligned Double Quadword</b>	
xmmreg2 to xmmreg1	C4: rxb0_1: w_F 010:6F:11 xmmreg1 xmmreg2
mem to xmmreg1	C4: rxb0_1: w_F 010:6F:mod xmmreg1 r/m
xmmreg0 to xmmreg1	C5: r_F 010:6F:11 xmmreg1 xmmreg0
mem to xmmreg1	C5: r_F 010:6F:mod xmmreg1 r/m
xmmreg1 to xmmreg2	C4: rxb0_1: w_F 010:7F:11 xmmreg2 xmmreg1
xmmreg1 to mem	C4: rxb0_1: w_F 010:7F:mod r/m xmmreg1
xmmreg1 to xmmreg0	C5: r_F 010:7F:11 xmmreg0 xmmreg1
xmmreg1 to mem	C5: r_F 010:7F:mod r/m xmmreg1
yymmreg2 to yymmreg1	C4: rxb0_1: w_F 110:6F:11 yymmreg1 yymmreg2
mem to yymmreg1	C4: rxb0_1: w_F 110:6F:mod yymmreg1 r/m
yymmreg0 to yymmreg1	C5: r_F 110:6F:11 yymmreg1 yymmreg0
mem to yymmreg1	C5: r_F 110:6F:mod yymmreg1 r/m
yymmreg1 to yymmreg2	C4: rxb0_1: w_F 110:7F:11 yymmreg2 yymmreg1
yymmreg1 to mem	C4: rxb0_1: w_F 110:7F:mod r/m yymmreg1
yymmreg1 to yymmreg0	C5: r_F 110:7F:11 yymmreg0 yymmreg1
yymmreg1 to mem	C5: r_F 110:7F:mod r/m yymmreg1

Instruction and Format	Encoding
<b>VMOVHPD — Move High Packed Double-Precision Floating-Point Value</b>	
xmmreg1 and mem to xmmreg2	C4: rxb0_1: w xmmreg1 001:16:11 xmmreg2 r/m
xmmreg1 and mem to xmmreglo2	C5: r_xmmreg1 001:16:11 xmmreglo2 r/m
xmmreg1 to mem	C4: rxb0_1: w_F 001:17:mod r/m xmmreg1
xmmreglo to mem	C5: r_F 001:17:mod r/m xmmreglo
<b>VMOVLPD — Move Low Packed Double-Precision Floating-Point Value</b>	
xmmreg1 and mem to xmmreg2	C4: rxb0_1: w xmmreg1 001:12:11 xmmreg2 r/m
xmmreg1 and mem to xmmreglo2	C5: r_xmmreg1 001:12:11 xmmreglo2 r/m
xmmreg1 to mem	C4: rxb0_1: w_F 001:13:mod r/m xmmreg1
xmmreglo to mem	C5: r_F 001:13:mod r/m xmmreglo
<b>VMOVMSKPD — Extract Packed Double-Precision Floating-Point Sign Mask</b>	
xmmreg2 to reg	C4: rxb0_1: w_F 001:50:11 reg xmmreg1
xmmreglo to reg	C5: r_F 001:50:11 reg xmmreglo
yymmreg2 to reg	C4: rxb0_1: w_F 101:50:11 reg yymmreg1
yymmreglo to reg	C5: r_F 101:50:11 reg yymmreglo
<b>VMOVNTDQ — Store Double Quadword Using Non-Temporal Hint</b>	
xmmreg1 to mem	C4: rxb0_1: w_F 001:E7:11 r/m xmmreg1
xmmreglo to mem	C5: r_F 001:E7:11 r/m xmmreglo
yymmreg1 to mem	C4: rxb0_1: w_F 101:E7:11 r/m yymmreg1
yymmreglo to mem	C5: r_F 101:E7:11 r/m yymmreglo
<b>VMOVNTPD — Store Packed Double-Precision Floating-Point Values Using Non-Temporal Hint</b>	
xmmreg1 to mem	C4: rxb0_1: w_F 001:2B:11 r/m xmmreg1
xmmreglo to mem	C5: r_F 001:2B:11 r/m xmmreglo
yymmreg1 to mem	C4: rxb0_1: w_F 101:2B:11r/m yymmreg1
yymmreglo to mem	C5: r_F 101:2B:11r/m yymmreglo
<b>VMOVSD — Move Scalar Double-Precision Floating-Point Value</b>	

## INSTRUCTION FORMATS AND ENCODINGS

Instruction and Format	Encoding
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 011:10:11 xmmreg1 xmmreg3
mem to xmmreg1	C4: rxb0_1: w_F 011:10:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 011:10:11 xmmreg1 xmmreglo3
mem to xmmreg1	C5: r_F 011:10:mod xmmreg1 r/m
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 011:11:11 xmmreg1 xmmreg3
xmmreg1 to mem	C4: rxb0_1: w_F 011:11:mod r/m xmmreg1
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 011:11:11 xmmreg1 xmmreglo3
xmmreglo to mem	C5: r_F 011:11:mod r/m xmmreglo
<b>VMOVUPD – Move Unaligned Packed Double-Precision Floating-Point Values</b>	
xmmreg2 to xmmreg1	C4: rxb0_1: w_F 001:10:11 xmmreg1 xmmreg2
mem to xmmreg1	C4: rxb0_1: w_F 001:10:mod xmmreg1 r/m
xmmreglo to xmmreg1	C5: r_F 001:10:11 xmmreg1 xmmreglo
mem to xmmreg1	C5: r_F 001:10:mod xmmreg1 r/m
yymmreg2 to yymmreg1	C4: rxb0_1: w_F 101:10:11 yymmreg1 yymmreg2
mem to yymmreg1	C4: rxb0_1: w_F 101:10:mod yymmreg1 r/m
yymmreglo to yymmreg1	C5: r_F 101:10:11 yymmreg1 yymmreglo
mem to yymmreg1	C5: r_F 101:10:mod yymmreg1 r/m
xmmreg1 to xmmreg2	C4: rxb0_1: w_F 001:11:11 xmmreg2 xmmreg1
xmmreg1 to mem	C4: rxb0_1: w_F 001:11:mod r/m xmmreg1
xmmreg1 to xmmreglo	C5: r_F 001:11:11 xmmreglo xmmreg1
xmmreg1 to mem	C5: r_F 001:11:mod r/m xmmreg1
yymmreg1 to yymmreg2	C4: rxb0_1: w_F 101:11:11 yymmreg2 yymmreg1
yymmreg1 to mem	C4: rxb0_1: w_F 101:11:mod r/m yymmreg1
yymmreg1 to yymmreglo	C5: r_F 101:11:11 yymmreglo yymmreg1
yymmreg1 to mem	C5: r_F 101:11:mod r/m yymmreg1
<b>VMULPD – Multiply Packed Double-Precision Floating-Point Values</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:59:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:59:mod xmmreg1 r/m

Instruction and Format	Encoding
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:59:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:59:mod xmmreg1 r/m
yymmreg2 with yymmreg3 to yymmreg1	C4: rxb0_1: w yymmreg2 101:59:11 yymmreg1 yymmreg3
yymmreg2 with mem to yymmreg1	C4: rxb0_1: w yymmreg2 101:59:mod yymmreg1 r/m
yymmreglo2 with yymmreglo3 to yymmreg1	C5: r_yymmreglo2 101:59:11 yymmreg1 yymmreglo3
yymmreglo2 with mem to yymmreg1	C5: r_yymmreglo2 101:59:mod yymmreg1 r/m
<b>VMULSD – Multiply Scalar Double-Precision Floating-Point Values</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 011:59:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 011:59:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 011:59:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 011:59:mod xmmreg1 r/m
<b>VORPD – Bitwise Logical OR of Double-Precision Floating-Point Values</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:56:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:56:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:56:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:56:mod xmmreg1 r/m
yymmreg2 with yymmreg3 to yymmreg1	C4: rxb0_1: w yymmreg2 101:56:11 yymmreg1 yymmreg3
yymmreg2 with mem to yymmreg1	C4: rxb0_1: w yymmreg2 101:56:mod yymmreg1 r/m
yymmreglo2 with yymmreglo3 to yymmreg1	C5: r_yymmreglo2 101:56:11 yymmreg1 yymmreglo3
yymmreglo2 with mem to yymmreg1	C5: r_yymmreglo2 101:56:mod yymmreg1 r/m
<b>VPACKSSWB— Pack with Signed Saturation</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:63:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:63:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:63:11 xmmreg1 xmmreglo3

## INSTRUCTION FORMATS AND ENCODINGS

Instruction and Format	Encoding
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:63:mod xmmreg1 r/m
<b>VPACKSSDW— Pack with Signed Saturation</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:6B:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:6B:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:6B:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:6B:mod xmmreg1 r/m
<b>VPACKUSWB— Pack with Unsigned Saturation</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:67:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:67:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:67:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:67:mod xmmreg1 r/m
<b>VPADDB — Add Packed Integers</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:FC:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:FC:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:FC:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:FC:mod xmmreg1 r/m
<b>VPADDW — Add Packed Integers</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:FD:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:FD:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:FD:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:FD:mod xmmreg1 r/m
<b>VPADDD — Add Packed Integers</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:FE:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:FE:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:FE:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:FE:mod xmmreg1 r/m

Instruction and Format	Encoding
<b>VPADDQ — Add Packed Quadword Integers</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:D4:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:D4:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:D4:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:D4:mod xmmreg1 r/m
<b>VPADDSB — Add Packed Signed Integers with Signed Saturation</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:EC:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:EC:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:EC:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:EC:mod xmmreg1 r/m
<b>VPADDSW — Add Packed Signed Integers with Signed Saturation</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:ED:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:ED:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:ED:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:ED:mod xmmreg1 r/m
<b>VPADDUSB — Add Packed Unsigned Integers with Unsigned Saturation</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:DC:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:DC:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:DC:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:DC:mod xmmreg1 r/m
<b>VPADDUSW — Add Packed Unsigned Integers with Unsigned Saturation</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:DD:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:DD:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:DD:11 xmmreg1 xmmreglo3



## INSTRUCTION FORMATS AND ENCODINGS

Instruction and Format	Encoding
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:DD:mod xmmreg1 r/m
<b>VPAND – Logical AND</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:DB:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:DB:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:DB:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:DB:mod xmmreg1 r/m
<b>VPANDN – Logical AND NOT</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:DF:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:DF:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:DF:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:DF:mod xmmreg1 r/m
<b>VPAVGB – Average Packed Integers</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:E0:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:E0:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:E0:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:E0:mod xmmreg1 r/m
<b>VPAVGW – Average Packed Integers</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:E3:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:E3:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:E3:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:E3:mod xmmreg1 r/m
<b>VPCMPEQB – Compare Packed Data for Equal</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:74:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:74:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:74:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:74:mod xmmreg1 r/m

Instruction and Format	Encoding
<b>VPCMPEQW — Compare Packed Data for Equal</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:75:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:75:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:75:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:75:mod xmmreg1 r/m
<b>VPCMPEQD — Compare Packed Data for Equal</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:76:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:76:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:76:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:76:mod xmmreg1 r/m
<b>VPCMPGTB — Compare Packed Signed Integers for Greater Than</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:64:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:64:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:64:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:64:mod xmmreg1 r/m
<b>VPCMPGTW — Compare Packed Signed Integers for Greater Than</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:65:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:65:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:65:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:65:mod xmmreg1 r/m
<b>VPCMPGTD — Compare Packed Signed Integers for Greater Than</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:66:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:66:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:66:11 xmmreg1 xmmreglo3

## INSTRUCTION FORMATS AND ENCODINGS

Instruction and Format	Encoding
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:66:mod xmmreg1 r/m
<b>VPEXTRW – Extract Word</b>	
xmmreg1 to reg using imm	C4: rxb0_1: 0_F 001:C5:11 reg xmmreg1: imm
xmmreg1 to reg using imm	C5: r_F 001:C5:11 reg xmmreg1: imm
<b>VPINSRW – Insert Word</b>	
xmmreg2 with reg to xmmreg1	C4: rxb0_1: 0 xmmreg2 001:C4:11 xmmreg1 reg: imm
xmmreg2 with mem to xmmreg1	C4: rxb0_1: 0 xmmreg2 001:C4:mod xmmreg1 r/m: imm
xmmreglo2 with reglo to xmmreg1	C5: r_xmmreglo2 001:C4:11 xmmreg1 reglo: imm
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:C4:mod xmmreg1 r/m: imm
<b>VPMADDWD – Multiply and Add Packed Integers</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:F5:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:F5:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:F5:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:F5:mod xmmreg1 r/m
<b>VPMAXSW – Maximum of Packed Signed Word Integers</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:EE:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:EE:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:EE:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:EE:mod xmmreg1 r/m
<b>VPMAXUB – Maximum of Packed Unsigned Byte Integers</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:DE:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:DE:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:DE:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:DE:mod xmmreg1 r/m
<b>VPMINSW – Minimum of Packed Signed Word Integers</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:EA:11 xmmreg1 xmmreg3

Instruction and Format	Encoding
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:EA:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:EA:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:EA:mod xmmreg1 r/m
<b>VPMINUB – Minimum of Packed Unsigned Byte Integers</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:DA:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:DA:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:DA:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:DA:mod xmmreg1 r/m
<b>VPMOVMSKB – Move Byte Mask</b>	
xmmreg1 to reg	C4: rxb0_1: w_F 001:D7:11 reg xmmreg1
xmmreg1 to reg	C5: r_F 001:D7:11 reg xmmreg1
<b>VPMULHUW – Multiply Packed Unsigned Integers and Store High Result</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:E4:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:E4:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:E4:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:E4:mod xmmreg1 r/m
<b>VPMULHW – Multiply Packed Signed Integers and Store High Result</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:E5:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:E5:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:E5:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:E5:mod xmmreg1 r/m
<b>VPMULLW – Multiply Packed Signed Integers and Store Low Result</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:D5:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:D5:mod xmmreg1 r/m

## INSTRUCTION FORMATS AND ENCODINGS

Instruction and Format	Encoding
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:D5:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:D5:mod xmmreg1 r/m
<b>VPMULUDQ – Multiply Packed Unsigned Doubleword Integers</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:F4:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:F4:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:F4:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:F4:mod xmmreg1 r/m
<b>VPOR – Bitwise Logical OR</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:EB:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:EB:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:EB:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:EB:mod xmmreg1 r/m
<b>VPSADBW – Compute Sum of Absolute Differences</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:F6:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:F6:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:F6:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:F6:mod xmmreg1 r/m
<b>VPSHUFD – Shuffle Packed Doublewords</b>	
xmmreg2 to xmmreg1 using imm	C4: rxb0_1: w_F 001:70:11 xmmreg1 xmmreg2: imm
mem to xmmreg1 using imm	C4: rxb0_1: w_F 001:70:mod xmmreg1 r/m: imm
xmmreglo to xmmreg1 using imm	C5: r_F 001:70:11 xmmreg1 xmmreglo: imm
mem to xmmreg1 using imm	C5: r_F 001:70:mod xmmreg1 r/m: imm
<b>VPSHUFW – Shuffle Packed High Words</b>	
xmmreg2 to xmmreg1 using imm	C4: rxb0_1: w_F 010:70:11 xmmreg1 xmmreg2: imm
mem to xmmreg1 using imm	C4: rxb0_1: w_F 010:70:mod xmmreg1 r/m: imm
xmmreglo to xmmreg1 using imm	C5: r_F 010:70:11 xmmreg1 xmmreglo: imm

Instruction and Format	Encoding
mem to xmmreg1 using imm	C5: r_F 010:70:mod xmmreg1 r/m: imm
<b>VPSHUFLW – Shuffle Packed Low Words</b>	
xmmreg2 to xmmreg1 using imm	C4: rxb0_1: w_F 011:70:11 xmmreg1 xmmreg2: imm
mem to xmmreg1 using imm	C4: rxb0_1: w_F 011:70:mod xmmreg1 r/m: imm
xmmreglo to xmmreg1 using imm	C5: r_F 011:70:11 xmmreg1 xmmreglo: imm
mem to xmmreg1 using imm	C5: r_F 011:70:mod xmmreg1 r/m: imm
<b>VPSLLDQ – Shift Double Quadword Left Logical</b>	
xmmreg2 to xmmreg1 using imm	C4: rxb0_1: w_F 001:73:11 xmmreg1 xmmreg2: imm
xmmreglo to xmmreg1 using imm	C5: r_F 001:73:11 xmmreg1 xmmreglo: imm
<b>VPSLLW – Shift Packed Data Left Logical</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:F1:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:F1:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:F1:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:F1:mod xmmreg1 r/m
xmmreg2 to xmmreg1 using imm8	C4: rxb0_1: w_F 001:71:11 xmmreg1 xmmreg2: imm
xmmreglo to xmmreg1 using imm8	C5: r_F 001:71:11 xmmreg1 xmmreglo: imm
<b>VPSLLD – Shift Packed Data Left Logical</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:F2:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:F2:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:F2:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:F2:mod xmmreg1 r/m
xmmreg2 to xmmreg1 using imm8	C4: rxb0_1: w_F 001:72:11 xmmreg1 xmmreg2: imm
xmmreglo to xmmreg1 using imm8	C5: r_F 001:72:11 xmmreg1 xmmreglo: imm
<b>VPSLLQ – Shift Packed Data Left Logical</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:F3:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:F3:mod xmmreg1 r/m

## INSTRUCTION FORMATS AND ENCODINGS

Instruction and Format	Encoding
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:F3:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:F3:mod xmmreg1 r/m
xmmreg2 to xmmreg1 using imm8	C4: rxb0_1: w_F 001:73:11 xmmreg1 xmmreg2: imm
xmmreglo to xmmreg1 using imm8	C5: r_F 001:73:11 xmmreg1 xmmreglo: imm
<b>VPSRAW – Shift Packed Data Right Arithmetic</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:E1:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:E1:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:E1:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:E1:mod xmmreg1 r/m
xmmreg2 to xmmreg1 using imm8	C4: rxb0_1: w_F 001:71:11 xmmreg1 xmmreg2: imm
xmmreglo to xmmreg1 using imm8	C5: r_F 001:71:11 xmmreg1 xmmreglo: imm
<b>VPSRAD – Shift Packed Data Right Arithmetic</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:E2:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:E2:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:E2:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:E2:mod xmmreg1 r/m
xmmreg2 to xmmreg1 using imm8	C4: rxb0_1: w_F 001:72:11 xmmreg1 xmmreg2: imm
xmmreglo to xmmreg1 using imm8	C5: r_F 001:72:11 xmmreg1 xmmreglo: imm
<b>VPSRLDQ – Shift Double Quadword Right Logical</b>	
xmmreg2 to xmmreg1 using imm8	C4: rxb0_1: w_F 001:73:11 xmmreg1 xmmreg2: imm
xmmreglo to xmmreg1 using imm8	C5: r_F 001:73:11 xmmreg1 xmmreglo: imm
<b>VPSRLW – Shift Packed Data Right Logical</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:D1:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:D1:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:D1:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:D1:mod xmmreg1 r/m

Instruction and Format	Encoding
xmmreg2 to xmmreg1 using imm8	C4: rxb0_1: w_F 001:71:11 xmmreg1 xmmreg2: imm
xmmreglo to xmmreg1 using imm8	C5: r_F 001:71:11 xmmreg1 xmmreglo: imm
<b>VPSRLD – Shift Packed Data Right Logical</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:D2:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:D2:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:D2:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:D2:mod xmmreg1 r/m
xmmreg2 to xmmreg1 using imm8	C4: rxb0_1: w_F 001:72:11 xmmreg1 xmmreg2: imm
xmmreglo to xmmreg1 using imm8	C5: r_F 001:72:11 xmmreg1 xmmreglo: imm
<b>VPSRLQ – Shift Packed Data Right Logical</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:D3:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:D3:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:D3:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:D3:mod xmmreg1 r/m
xmmreg2 to xmmreg1 using imm8	C4: rxb0_1: w_F 001:73:11 xmmreg1 xmmreg2: imm
xmmreglo to xmmreg1 using imm8	C5: r_F 001:73:11 xmmreg1 xmmreglo: imm
<b>VPSUBB – Subtract Packed Integers</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:F8:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:F8:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:F8:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:F8:mod xmmreg1 r/m
<b>VPSUBW – Subtract Packed Integers</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:F9:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:F9:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:F9:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:F9:mod xmmreg1 r/m
<b>VPSUBD – Subtract Packed Integers</b>	



## INSTRUCTION FORMATS AND ENCODINGS

Instruction and Format	Encoding
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:FA:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:FA:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:FA:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:FA:mod xmmreg1 r/m
<b>VPSUBQ – Subtract Packed Quadword Integers</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:FB:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:FB:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:FB:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:FB:mod xmmreg1 r/m
<b>VPSUBSB – Subtract Packed Signed Integers with Signed Saturation</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:E8:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:E8:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:E8:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:E8:mod xmmreg1 r/m
<b>VPSUBSW – Subtract Packed Signed Integers with Signed Saturation</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:E9:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:E9:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:E9:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:E9:mod xmmreg1 r/m
<b>VPSUBUSB – Subtract Packed Unsigned Integers with Unsigned Saturation</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:D8:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:D8:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:D8:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:D8:mod xmmreg1 r/m

Instruction and Format	Encoding
<b>VPSUBUSW – Subtract Packed Unsigned Integers with Unsigned Saturation</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:D9:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:D9:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:D9:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:D9:mod xmmreg1 r/m
<b>VPUNPCKHBW – Unpack High Data</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:68:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:68:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:68:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:68:mod xmmreg1 r/m
<b>VPUNPCKHWD – Unpack High Data</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:69:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:69:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:69:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:69:mod xmmreg1 r/m
<b>VPUNPCKHDQ – Unpack High Data</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:6A:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:6A:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:6A:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:6A:mod xmmreg1 r/m
<b>VPUNPCKHQDQ – Unpack High Data</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:6D:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:6D:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:6D:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:6D:mod xmmreg1 r/m
<b>VPUNPCKLBW – Unpack Low Data</b>	

## INSTRUCTION FORMATS AND ENCODINGS

Instruction and Format	Encoding
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:60:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:60:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:60:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:60:mod xmmreg1 r/m
<b>VPUNPCKLWD – Unpack Low Data</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:61:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:61:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:61:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:61:mod xmmreg1 r/m
<b>VPUNPCKLDQ – Unpack Low Data</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:62:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:62:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:62:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:62:mod xmmreg1 r/m
<b>VPUNPCKLQDQ – Unpack Low Data</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:6C:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:6C:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:6C:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:6C:mod xmmreg1 r/m
<b>VPXOR – Logical Exclusive OR</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:EF:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:EF:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:EF:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:EF:mod xmmreg1 r/m
<b>VSHUFPD – Shuffle Packed Double-Precision Floating-Point Values</b>	
xmmreg2 with xmmreg3 to xmmreg1 using imm8	C4: rxb0_1: w xmmreg2 001:C6:11 xmmreg1 xmmreg3: imm

Instruction and Format	Encoding
xmmreg2 with mem to xmmreg1 using imm8	C4: rxb0_1: w xmmreg2 001:C6:mod xmmreg1 r/m: imm
xmmreglo2 with xmmreglo3 to xmmreg1 using imm8	C5: r_xmmreglo2 001:C6:11 xmmreg1 xmmreglo3: imm
xmmreglo2 with mem to xmmreg1 using imm8	C5: r_xmmreglo2 001:C6:mod xmmreg1 r/m: imm
yymmreg2 with yymmreg3 to yymmreg1 using imm8	C4: rxb0_1: w yymmreg2 101:C6:11 yymmreg1 yymmreg3: imm
yymmreg2 with mem to yymmreg1 using imm8	C4: rxb0_1: w yymmreg2 101:C6:mod yymmreg1 r/m: imm
yymmreglo2 with yymmreglo3 to yymmreg1 using imm8	C5: r_yymmreglo2 101:C6:11 yymmreg1 yymmreglo3: imm
yymmreglo2 with mem to yymmreg1 using imm8	C5: r_yymmreglo2 101:C6:mod yymmreg1 r/m: imm
<b>VSQRTPD – Compute Square Roots of Packed Double-Precision Floating-Point Values</b>	
xmmreg2 to xmmreg1	C4: rxb0_1: w_F 001:51:11 xmmreg1 xmmreg2
mem to xmmreg1	C4: rxb0_1: w_F 001:51:mod xmmreg1 r/m
xmmreglo to xmmreg1	C5: r_F 001:51:11 xmmreg1 xmmreglo
mem to xmmreg1	C5: r_F 001:51:mod xmmreg1 r/m
yymmreg2 to yymmreg1	C4: rxb0_1: w_F 101:51:11 yymmreg1 yymmreg2
mem to yymmreg1	C4: rxb0_1: w_F 101:51:mod yymmreg1 r/m
yymmreglo to yymmreg1	C5: r_F 101:51:11 yymmreg1 yymmreglo
mem to yymmreg1	C5: r_F 101:51:mod yymmreg1 r/m
<b>VSQRTSD – Compute Square Root of Scalar Double-Precision Floating-Point Value</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 011:51:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 011:51:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 011:51:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 011:51:mod xmmreg1 r/m
<b>VSUBPD – Subtract Packed Double-Precision Floating-Point Values</b>	

## INSTRUCTION FORMATS AND ENCODINGS

Instruction and Format	Encoding
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:5C:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:5C:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:5C:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:5C:mod xmmreg1 r/m
yymmreg2 with yymmreg3 to yymmreg1	C4: rxb0_1: w yymmreg2 101:5C:11 yymmreg1 yymmreg3
yymmreg2 with mem to yymmreg1	C4: rxb0_1: w yymmreg2 101:5C:mod yymmreg1 r/m
yymmreglo2 with yymmreglo3 to yymmreg1	C5: r_yymmreglo2 101:5C:11 yymmreg1 yymmreglo3
yymmreglo2 with mem to yymmreg1	C5: r_yymmreglo2 101:5C:mod yymmreg1 r/m
<b>VSUBSD – Subtract Scalar Double-Precision Floating-Point Values</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 011:5C:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 011:5C:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 011:5C:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 011:5C:mod xmmreg1 r/m
<b>VUCOMISD – Unordered Compare Scalar Double-Precision Floating-Point Values and Set EFLAGS</b>	
xmmreg2 with xmmreg1, set EFLAGS	C4: rxb0_1: w_F xmmreg1 001:2E:11 xmmreg2
mem with xmmreg1, set EFLAGS	C4: rxb0_1: w_F xmmreg1 001:2E:mod r/m
xmmreglo with xmmreg1, set EFLAGS	C5: r_F xmmreg1 001:2E:11 xmmreglo
mem with xmmreg1, set EFLAGS	C5: r_F xmmreg1 001:2E:mod r/m
<b>VUNPCKHPD – Unpack and Interleave High Packed Double-Precision Floating-Point Values</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:15:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:15:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:15:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:15:mod xmmreg1 r/m
yymmreg2 with yymmreg3 to yymmreg1	C4: rxb0_1: w yymmreg2 101:15:11 yymmreg1 yymmreg3
yymmreg2 with mem to yymmreg1	C4: rxb0_1: w yymmreg2 101:15:mod yymmreg1 r/m

Instruction and Format	Encoding
yymmreglo2 with ymmreglo3 to ymmreg1	C5: r_yymmreglo2 101:15:11 ymmreg1 ymmreglo3
yymmreglo2 with mem to ymmreg1	C5: r_yymmreglo2 101:15:mod ymmreg1 r/m
<b>VUNPCKHPS – Unpack and Interleave High Packed Single-Precision Floating-Point Values</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 000:15:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 000:15:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 000:15:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 000:15:mod xmmreg1 r/m
yymmreg2 with ymmreg3 to ymmreg1	C4: rxb0_1: w ymmreg2 100:15:11 ymmreg1 ymmreg3
yymmreg2 with mem to ymmreg1	C4: rxb0_1: w ymmreg2 100:15:mod ymmreg1 r/m
yymmreglo2 with ymmreglo3 to ymmreg1	C5: r_yymmreglo2 100:15:11 ymmreg1 ymmreglo3
yymmreglo2 with mem to ymmreg1	C5: r_yymmreglo2 100:15:mod ymmreg1 r/m
<b>VUNPCKLPD – Unpack and Interleave Low Packed Double-Precision Floating-Point Values</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:14:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:14:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:14:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:14:mod xmmreg1 r/m
yymmreg2 with ymmreg3 to ymmreg1	C4: rxb0_1: w ymmreg2 101:14:11 ymmreg1 ymmreg3
yymmreg2 with mem to ymmreg1	C4: rxb0_1: w ymmreg2 101:14:mod ymmreg1 r/m
yymmreglo2 with ymmreglo3 to ymmreg1	C5: r_yymmreglo2 101:14:11 ymmreg1 ymmreglo3
yymmreglo2 with mem to ymmreg1	C5: r_yymmreglo2 101:14:mod ymmreg1 r/m
<b>VUNPCKLPS – Unpack and Interleave Low Packed Single-Precision Floating-Point Values</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 000:14:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 000:14:mod xmmreg1 r/m

INSTRUCTION FORMATS AND ENCODINGS

Instruction and Format	Encoding
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 000:14:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 000:14:mod xmmreg1 r/m
ymmreg2 with ymmreg3 to ymmreg1	C4: rxb0_1: w ymmreg2 100:14:11 ymmreg1 ymmreg3
ymmreg2 with mem to ymmreg1	C4: rxb0_1: w ymmreg2 100:14:mod ymmreg1 r/m
ymmreglo2 with ymmreglo3 to ymmreg1	C5: r_ymmreglo2 100:14:11 ymmreg1 ymmreglo3
ymmreglo2 with mem to ymmreg1	C5: r_ymmreglo2 100:14:mod ymmreg1 r/m
<b>VXORPD – Bitwise Logical XOR for Double-Precision Floating-Point Values</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:57:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 001:57:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 001:57:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:57:mod xmmreg1 r/m
ymmreg2 with ymmreg3 to ymmreg1	C4: rxb0_1: w ymmreg2 101:57:11 ymmreg1 ymmreg3
ymmreg2 with mem to ymmreg1	C4: rxb0_1: w ymmreg2 101:57:mod ymmreg1 r/m
ymmreglo2 with ymmreglo3 to ymmreg1	C5: r_ymmreglo2 101:57:11 ymmreg1 ymmreglo3
ymmreglo2 with mem to ymmreg1	C5: r_ymmreglo2 101:57:mod ymmreg1 r/m
<b>VADDPS – Add Packed Single-Precision Floating-Point Values</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 000:58:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 000:58:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 000:58:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 000:58:mod xmmreg1 r/m
ymmreg2 with ymmreg3 to ymmreg1	C4: rxb0_1: w ymmreg2 100:58:11 ymmreg1 ymmreg3
ymmreg2 with mem to ymmreg1	C4: rxb0_1: w ymmreg2 100:58:mod ymmreg1 r/m
ymmreglo2 with ymmreglo3 to ymmreg1	C5: r_ymmreglo2 100:58:11 ymmreg1 ymmreglo3
ymmreglo2 with mem to ymmreg1	C5: r_ymmreglo2 100:58:mod ymmreg1 r/m

Instruction and Format	Encoding
<b>VADDSS – Add Scalar Single-Precision Floating-Point Values</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 010:58:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 010:58:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 010:58:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 010:58:mod xmmreg1 r/m
<b>VANDPS – Bitwise Logical AND of Packed Single-Precision Floating-Point Values</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 000:54:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 000:54:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 000:54:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 000:54:mod xmmreg1 r/m
ymmreg2 with ymmreg3 to ymmreg1	C4: rxb0_1: w ymmreg2 100:54:11 ymmreg1 ymmreg3
ymmreg2 with mem to ymmreg1	C4: rxb0_1: w ymmreg2 100:54:mod ymmreg1 r/m
ymmreglo2 with ymmreglo3 to ymmreg1	C5: r_ymmreglo2 100:54:11 ymmreg1 ymmreglo3
ymmreglo2 with mem to ymmreg1	C5: r_ymmreglo2 100:54:mod ymmreg1 r/m
<b>VANDNPS – Bitwise Logical AND NOT of Packed Single-Precision Floating-Point Values</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 000:55:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 000:55:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 000:55:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 000:55:mod xmmreg1 r/m
ymmreg2 with ymmreg3 to ymmreg1	C4: rxb0_1: w ymmreg2 100:55:11 ymmreg1 ymmreg3
ymmreg2 with mem to ymmreg1	C4: rxb0_1: w ymmreg2 100:55:mod ymmreg1 r/m
ymmreglo2 with ymmreglo3 to ymmreg1	C5: r_ymmreglo2 100:55:11 ymmreg1 ymmreglo3
ymmreglo2 with mem to ymmreg1	C5: r_ymmreglo2 100:55:mod ymmreg1 r/m
<b>VCMPSS – Compare Packed Single-Precision Floating-Point Values</b>	



## INSTRUCTION FORMATS AND ENCODINGS

Instruction and Format	Encoding
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 000:C2:11 xmmreg1 xmmreg3: imm
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 000:C2:mod xmmreg1 r/m: imm
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 000:C2:11 xmmreg1 xmmreglo3: imm
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 000:C2:mod xmmreg1 r/m: imm
yymmreg2 with yymmreg3 to yymmreg1	C4: rxb0_1: w yymmreg2 100:C2:11 yymmreg1 yymmreg3: imm
yymmreg2 with mem to yymmreg1	C4: rxb0_1: w yymmreg2 100:C2:mod yymmreg1 r/m: imm
yymmreglo2 with yymmreglo3 to yymmreg1	C5: r_yymmreglo2 100:C2:11 yymmreg1 yymmreglo3: imm
yymmreglo2 with mem to yymmreg1	C5: r_yymmreglo2 100:C2:mod yymmreg1 r/m: imm
<b>VCMPSS – Compare Scalar Single-Precision Floating-Point Values</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 010:C2:11 xmmreg1 xmmreg3: imm
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 010:C2:mod xmmreg1 r/m: imm
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 010:C2:11 xmmreg1 xmmreglo3: imm
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 010:C2:mod xmmreg1 r/m: imm
<b>VCOMISS – Compare Scalar Ordered Single-Precision Floating-Point Values and Set EFLAGS</b>	
xmmreg2 with xmmreg1	C4: rxb0_1: w_F 000:2F:11 xmmreg1 xmmreg2
mem with xmmreg1	C4: rxb0_1: w_F 000:2F:mod xmmreg1 r/m
xmmreglo with xmmreg1	C5: r_F 000:2F:11 xmmreg1 xmmreglo
mem with xmmreg1	C5: r_F 000:2F:mod xmmreg1 r/m
<b>VCVTSI2SS – Convert Dword Integer to Scalar Single-Precision FP Value</b>	
xmmreg2 with reg to xmmreg1	C4: rxb0_1: 0 xmmreg2 010:2A:11 xmmreg1 reg
xmmreg2 with mem to xmmreg1	C4: rxb0_1: 0 xmmreg2 010:2A:mod xmmreg1 r/m
xmmreglo2 with reglo to xmmreg1	C5: r_xmmreglo2 010:2A:11 xmmreg1 reglo
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 010:2A:mod xmmreg1 r/m
xmmreg2 with reg to xmmreg1	C4: rxb0_1: 1 xmmreg2 010:2A:11 xmmreg1 reg

Instruction and Format	Encoding
xmmreg2 with mem to xmmreg1	C4: rxb0_1: 1 xmmreg2 010:2A:mod xmmreg1 r/m
<b>VCVTSS2SI – Convert Scalar Single-Precision FP Value to Dword Integer</b>	
xmmreg1 to reg	C4: rxb0_1: 0_F 010:2D:11 reg xmmreg1
mem to reg	C4: rxb0_1: 0_F 010:2D:mod reg r/m
xmmreglo to reg	C5: r_F 010:2D:11 reg xmmreglo
mem to reg	C5: r_F 010:2D:mod reg r/m
xmmreg1 to reg	C4: rxb0_1: 1_F 010:2D:11 reg xmmreg1
mem to reg	C4: rxb0_1: 1_F 010:2D:mod reg r/m
<b>VCVTSS2SI – Convert with Truncation Scalar Single-Precision FP Value to Dword Integer</b>	
xmmreg1 to reg	C4: rxb0_1: 0_F 010:2C:11 reg xmmreg1
mem to reg	C4: rxb0_1: 0_F 010:2C:mod reg r/m
xmmreglo to reg	C5: r_F 010:2C:11 reg xmmreglo
mem to reg	C5: r_F 010:2C:mod reg r/m
xmmreg1 to reg	C4: rxb0_1: 1_F 010:2C:11 reg xmmreg1
mem to reg	C4: rxb0_1: 1_F 010:2C:mod reg r/m
<b>VDIVPS – Divide Packed Single-Precision Floating-Point Values</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 000:5E:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 000:5E:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 000:5E:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 000:5E:mod xmmreg1 r/m
yymmreg2 with yymmreg3 to yymmreg1	C4: rxb0_1: w yymmreg2 100:5E:11 yymmreg1 yymmreg3
yymmreg2 with mem to yymmreg1	C4: rxb0_1: w yymmreg2 100:5E:mod yymmreg1 r/m
yymmreglo2 with yymmreglo3 to yymmreg1	C5: r_yymmreglo2 100:5E:11 yymmreg1 yymmreglo3
yymmreglo2 with mem to yymmreg1	C5: r_yymmreglo2 100:5E:mod yymmreg1 r/m
<b>VDIVSS – Divide Scalar Single-Precision Floating-Point Values</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 010:5E:11 xmmreg1 xmmreg3

## INSTRUCTION FORMATS AND ENCODINGS

Instruction and Format	Encoding
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 010:5E:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 010:5E:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 010:5E:mod xmmreg1 r/m
<b>VLDMXCSR – Load MXCSR Register</b>	
mem to MXCSR reg	C4: rxb0_1: w_F 000:AE:mod 011 r/m
mem to MXCSR reg	C5: r_F 000:AE:mod 011 r/m
<b>VMAXPS – Return Maximum Packed Single-Precision Floating-Point Values</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 000:5F:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 000:5F:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 000:5F:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 000:5F:mod xmmreg1 r/m
yymmreg2 with yymmreg3 to yymmreg1	C4: rxb0_1: w yymmreg2 100:5F:11 yymmreg1 yymmreg3
yymmreg2 with mem to yymmreg1	C4: rxb0_1: w yymmreg2 100:5F:mod yymmreg1 r/m
yymmreglo2 with yymmreglo3 to yymmreg1	C5: r_yymmreglo2 100:5F:11 yymmreg1 yymmreglo3
yymmreglo2 with mem to yymmreg1	C5: r_yymmreglo2 100:5F:mod yymmreg1 r/m
<b>VMAXSS – Return Maximum Scalar Single-Precision Floating-Point Value</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 010:5F:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 010:5F:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 010:5F:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 010:5F:mod xmmreg1 r/m
<b>VMINPS – Return Minimum Packed Single-Precision Floating-Point Values</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 000:5D:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 000:5D:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 000:5D:11 xmmreg1 xmmreglo3

Instruction and Format	Encoding
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 000:5D:mod xmmreg1 r/m
yymmreg2 with yymmreg3 to yymmreg1	C4: rxb0_1: w yymmreg2 100:5D:11 yymmreg1 yymmreg3
yymmreg2 with mem to yymmreg1	C4: rxb0_1: w yymmreg2 100:5D:mod yymmreg1 r/m
yymmreglo2 with yymmreglo3 to yymmreg1	C5: r_yymmreglo2 100:5D:11 yymmreg1 yymmreglo3
yymmreglo2 with mem to yymmreg1	C5: r_yymmreglo2 100:5D:mod yymmreg1 r/m
<b>VMINSS – Return Minimum Scalar Single-Precision Floating-Point Value</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 010:5D:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 010:5D:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 010:5D:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 010:5D:mod xmmreg1 r/m
<b>VMOVAPS— Move Aligned Packed Single-Precision Floating-Point Values</b>	
xmmreg2 to xmmreg1	C4: rxb0_1: w_F 000:28:11 xmmreg1 xmmreg2
mem to xmmreg1	C4: rxb0_1: w_F 000:28:mod xmmreg1 r/m
xmmreglo to xmmreg1	C5: r_F 000:28:11 xmmreg1 xmmreglo
mem to xmmreg1	C5: r_F 000:28:mod xmmreg1 r/m
xmmreg1 to xmmreg2	C4: rxb0_1: w_F 000:29:11 xmmreg2 xmmreg1
xmmreg1 to mem	C4: rxb0_1: w_F 000:29:mod r/m xmmreg1
xmmreg1 to xmmreglo	C5: r_F 000:29:11 xmmreglo xmmreg1
xmmreg1 to mem	C5: r_F 000:29:mod r/m xmmreg1
yymmreg2 to yymmreg1	C4: rxb0_1: w_F 100:28:11 yymmreg1 yymmreg2
mem to yymmreg1	C4: rxb0_1: w_F 100:28:mod yymmreg1 r/m
yymmreglo to yymmreg1	C5: r_F 100:28:11 yymmreg1 yymmreglo
mem to yymmreg1	C5: r_F 100:28:mod yymmreg1 r/m
yymmreg1 to yymmreg2	C4: rxb0_1: w_F 100:29:11 yymmreg2 yymmreg1
yymmreg1 to mem	C4: rxb0_1: w_F 100:29:mod r/m yymmreg1
yymmreg1 to yymmreglo	C5: r_F 100:29:11 yymmreglo yymmreg1
yymmreg1 to mem	C5: r_F 100:29:mod r/m yymmreg1

## INSTRUCTION FORMATS AND ENCODINGS

Instruction and Format	Encoding
<b>VMOVHPS — Move High Packed Single-Precision Floating-Point Values</b>	
xmmreg1 with mem to xmmreg2	C4: rxb0_1: w xmmreg1 000:16:mod xmmreg2 r/m
xmmreg1 with mem to xmmreglo2	C5: r_xmmreg1 000:16:mod xmmreglo2 r/m
xmmreg1 to mem	C4: rxb0_1: w_F 000:17:mod r/m xmmreg1
xmmreglo to mem	C5: r_F 000:17:mod r/m xmmreglo
<b>VMOVLHPS — Move Packed Single-Precision Floating-Point Values Low to High</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 000:16:11 xmmreg1 xmmreg3
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 000:16:11 xmmreg1 xmmreglo3
<b>VMOVLPS — Move Low Packed Single-Precision Floating-Point Values</b>	
xmmreg1 with mem to xmmreg2	C4: rxb0_1: w xmmreg1 000:12:mod xmmreg2 r/m
xmmreg1 with mem to xmmreglo2	C5: r_xmmreg1 000:12:mod xmmreglo2 r/m
xmmreg1 to mem	C4: rxb0_1: w_F 000:13:mod r/m xmmreg1
xmmreglo to mem	C5: r_F 000:13:mod r/m xmmreglo
<b>VMOVMSKPS — Extract Packed Single-Precision Floating-Point Sign Mask</b>	
xmmreg2 to reg	C4: rxb0_1: w_F 000:50:11 reg xmmreg2
xmmreglo to reg	C5: r_F 000:50:11 reg xmmreglo
yymmreg2 to reg	C4: rxb0_1: w_F 100:50:11 reg yymmreg2
yymmreglo to reg	C5: r_F 100:50:11 reg yymmreglo
<b>VMOVNTPS — Store Packed Single-Precision Floating-Point Values Using Non-Temporal Hint</b>	
xmmreg1 to mem	C4: rxb0_1: w_F 000:2B:mod r/m xmmreg1
xmmreglo to mem	C5: r_F 000:2B:mod r/m xmmreglo
yymmreg1 to mem	C4: rxb0_1: w_F 100:2B:mod r/m yymmreg1
yymmreglo to mem	C5: r_F 100:2B:mod r/m yymmreglo
<b>VMOVSS — Move Scalar Single-Precision Floating-Point Values</b>	

Instruction and Format	Encoding
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 010:10:11 xmmreg1 xmmreg3
mem to xmmreg1	C4: rxb0_1: w_F 010:10:mod xmmreg1 r/m
xmmreg2 with xmmreg3 to xmmreg1	C5: r_xmmreg2 010:10:11 xmmreg1 xmmreg3
mem to xmmreg1	C5: r_F 010:10:mod xmmreg1 r/m
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 010:11:11 xmmreg1 xmmreg3
xmmreg1 to mem	C4: rxb0_1: w_F 010:11:mod r/m xmmreg1
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 010:11:11 xmmreg1 xmmreglo3
xmmreglo to mem	C5: r_F 010:11:mod r/m xmmreglo
<b>VMOVUPS— Move Unaligned Packed Single-Precision Floating-Point Values</b>	
xmmreg2 to xmmreg1	C4: rxb0_1: w_F 000:10:11 xmmreg1 xmmreg2
mem to xmmreg1	C4: rxb0_1: w_F 000:10:mod xmmreg1 r/m
xmmreglo to xmmreg1	C5: r_F 000:10:11 xmmreg1 xmmreglo
mem to xmmreg1	C5: r_F 000:10:mod xmmreg1 r/m
yymmreg2 to yymmreg1	C4: rxb0_1: w_F 100:10:11 yymmreg1 yymmreg2
mem to yymmreg1	C4: rxb0_1: w_F 100:10:mod yymmreg1 r/m
yymmreglo to yymmreg1	C5: r_F 100:10:11 yymmreg1 yymmreglo
mem to yymmreg1	C5: r_F 100:10:mod yymmreg1 r/m
xmmreg1 to xmmreg2	C4: rxb0_1: w_F 000:11:11 xmmreg2 xmmreg1
xmmreg1 to mem	C4: rxb0_1: w_F 000:11:mod r/m xmmreg1
xmmreg1 to xmmreglo	C5: r_F 000:11:11 xmmreglo xmmreg1
xmmreg1 to mem	C5: r_F 000:11:mod r/m xmmreg1
yymmreg1 to yymmreg2	C4: rxb0_1: w_F 100:11:11 yymmreg2 yymmreg1
yymmreg1 to mem	C4: rxb0_1: w_F 100:11:mod r/m yymmreg1
yymmreg1 to yymmreglo	C5: r_F 100:11:11 yymmreglo yymmreg1
yymmreg1 to mem	C5: r_F 100:11:mod r/m yymmreg1
<b>VMULPS — Multiply Packed Single-Precision Floating-Point Values</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 000:59:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 000:59:mod xmmreg1 r/m

## INSTRUCTION FORMATS AND ENCODINGS

Instruction and Format	Encoding
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 000:59:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 000:59:mod xmmreg1 r/m
ymmreg2 with ymmreg3 to ymmreg1	C4: rxb0_1: w ymmreg2 100:59:11 ymmreg1 ymmreg3
ymmreg2 with mem to ymmreg1	C4: rxb0_1: w ymmreg2 100:59:mod ymmreg1 r/m
ymmreglo2 with ymmreglo3 to ymmreg1	C5: r_ymmreglo2 100:59:11 ymmreg1 ymmreglo3
ymmreglo2 with mem to ymmreg1	C5: r_ymmreglo2 100:59:mod ymmreg1 r/m
<b>VMULSS – Multiply Scalar Single-Precision Floating-Point Values</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 010:59:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 010:59:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 010:59:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 010:59:mod xmmreg1 r/m
<b>VORPS – Bitwise Logical OR of Single-Precision Floating-Point Values</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 000:56:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 000:56:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 000:56:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 000:56:mod xmmreg1 r/m
ymmreg2 with ymmreg3 to ymmreg1	C4: rxb0_1: w ymmreg2 100:56:11 ymmreg1 ymmreg3
ymmreg2 with mem to ymmreg1	C4: rxb0_1: w ymmreg2 100:56:mod ymmreg1 r/m
ymmreglo2 with ymmreglo3 to ymmreg1	C5: r_ymmreglo2 100:56:11 ymmreg1 ymmreglo3
ymmreglo2 with mem to ymmreg1	C5: r_ymmreglo2 100:56:mod ymmreg1 r/m
<b>VRCPPS – Compute Reciprocals of Packed Single-Precision Floating-Point Values</b>	
xmmreg2 to xmmreg1	C4: rxb0_1: w_F 000:53:11 xmmreg1 xmmreg2
mem to xmmreg1	C4: rxb0_1: w_F 000:53:mod xmmreg1 r/m
xmmreglo to xmmreg1	C5: r_F 000:53:11 xmmreg1 xmmreglo

Instruction and Format	Encoding
mem to xmmreg1	C5: r_F 000:53:mod xmmreg1 r/m
yymmreg2 to yymmreg1	C4: rxb0_1: w_F 100:53:11 yymmreg1 yymmreg2
mem to yymmreg1	C4: rxb0_1: w_F 100:53:mod yymmreg1 r/m
yymmreglo to yymmreg1	C5: r_F 100:53:11 yymmreg1 yymmreglo
mem to yymmreg1	C5: r_F 100:53:mod yymmreg1 r/m
<b>VRCPPSS – Compute Reciprocal of Scalar Single-Precision Floating-Point Values</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 010:53:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 010:53:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 010:53:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 010:53:mod xmmreg1 r/m
<b>VRSQRTPS – Compute Reciprocals of Square Roots of Packed Single-Precision Floating-Point Values</b>	
xmmreg2 to xmmreg1	C4: rxb0_1: w_F 000:52:11 xmmreg1 xmmreg2
mem to xmmreg1	C4: rxb0_1: w_F 000:52:mod xmmreg1 r/m
xmmreglo to xmmreg1	C5: r_F 000:52:11 xmmreg1 xmmreglo
mem to xmmreg1	C5: r_F 000:52:mod xmmreg1 r/m
yymmreg2 to yymmreg1	C4: rxb0_1: w_F 100:52:11 yymmreg1 yymmreg2
mem to yymmreg1	C4: rxb0_1: w_F 100:52:mod yymmreg1 r/m
yymmreglo to yymmreg1	C5: r_F 100:52:11 yymmreg1 yymmreglo
mem to yymmreg1	C5: r_F 100:52:mod yymmreg1 r/m
<b>VRSQRTSS – Compute Reciprocal of Square Root of Scalar Single-Precision Floating-Point Value</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 010:52:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 010:52:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 010:52:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 010:52:mod xmmreg1 r/m
<b>VSHUFPS – Shuffle Packed Single-Precision Floating-Point Values</b>	



## INSTRUCTION FORMATS AND ENCODINGS

Instruction and Format	Encoding
xmmreg2 with xmmreg3 to xmmreg1, imm8	C4: rxb0_1: w xmmreg2 000:C6:11 xmmreg1 xmmreg3: imm
xmmreg2 with mem to xmmreg1, imm8	C4: rxb0_1: w xmmreg2 000:C6:mod xmmreg1 r/m: imm
xmmreglo2 with xmmreglo3 to xmmreg1, imm8	C5: r_xmmreglo2 000:C6:11 xmmreg1 xmmreglo3: imm
xmmreglo2 with mem to xmmreg1, imm8	C5: r_xmmreglo2 000:C6:mod xmmreg1 r/m: imm
ymmreg2 with ymmreg3 to ymmreg1, imm8	C4: rxb0_1: w ymmreg2 100:C6:11 ymmreg1 ymmreg3: imm
ymmreg2 with mem to ymmreg1, imm8	C4: rxb0_1: w ymmreg2 100:C6:mod ymmreg1 r/m: imm
ymmreglo2 with ymmreglo3 to ymmreg1, imm8	C5: r_ymmreglo2 100:C6:11 ymmreg1 ymmreglo3: imm
ymmreglo2 with mem to ymmreg1, imm8	C5: r_ymmreglo2 100:C6:mod ymmreg1 r/m: imm
<b>VSQRTPS — Compute Square Roots of Packed Single-Precision Floating-Point Values</b>	
xmmreg2 to xmmreg1	C4: rxb0_1: w_F 000:51:11 xmmreg1 xmmreg2
mem to xmmreg1	C4: rxb0_1: w_F 000:51:mod xmmreg1 r/m
xmmreglo to xmmreg1	C5: r_F 000:51:11 xmmreg1 xmmreglo
mem to xmmreg1	C5: r_F 000:51:mod xmmreg1 r/m
ymmreg2 to ymmreg1	C4: rxb0_1: w_F 100:51:11 ymmreg1 ymmreg2
mem to ymmreg1	C4: rxb0_1: w_F 100:51:mod ymmreg1 r/m
ymmreglo to ymmreg1	C5: r_F 100:51:11 ymmreg1 ymmreglo
mem to ymmreg1	C5: r_F 100:51:mod ymmreg1 r/m
<b>VSQRTSS — Compute Square Root of Scalar Single-Precision Floating-Point Value</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 010:51:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 010:51:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 010:51:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 010:51:mod xmmreg1 r/m

Instruction and Format	Encoding
<b>VSTMXCSR – Store MXCSR Register State</b>	
MXCSR to mem	C4: rxb0_1: w_F 000:AE:mod 011 r/m
MXCSR to mem	C5: r_F 000:AE:mod 011 r/m
<b>VSUBPS – Subtract Packed Single-Precision Floating-Point Values</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 000:5C:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 000:5C:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 000:5C:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 000:5C:mod xmmreg1 r/m
yymmreg2 with yymmreg3 to yymmreg1	C4: rxb0_1: w yymmreg2 100:5C:11 yymmreg1 yymmreg3
yymmreg2 with mem to yymmreg1	C4: rxb0_1: w yymmreg2 100:5C:mod yymmreg1 r/m
yymmreglo2 with yymmreglo3 to yymmreg1	C5: r_yymmreglo2 100:5C:11 yymmreg1 yymmreglo3
yymmreglo2 with mem to yymmreg1	C5: r_yymmreglo2 100:5C:mod yymmreg1 r/m
<b>VSUBSS – Subtract Scalar Single-Precision Floating-Point Values</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 010:5C:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 010:5C:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 010:5C:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 010:5C:mod xmmreg1 r/m
<b>VUCOMISS – Unordered Compare Scalar Single-Precision Floating-Point Values and Set EFLAGS</b>	
xmmreg2 with xmmreg1	C4: rxb0_1: w_F 000:2E:11 xmmreg1 xmmreg2
mem with xmmreg1	C4: rxb0_1: w_F 000:2E:mod xmmreg1 r/m
xmmreglo with xmmreg1	C5: r_F 000:2E:11 xmmreg1 xmmreglo
mem with xmmreg1	C5: r_F 000:2E:mod xmmreg1 r/m
<b>UNPCKHPS – Unpack and Interleave High Packed Single-Precision Floating-Point Values</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 000:15:11 xmmreg1 xmmreg3

INSTRUCTION FORMATS AND ENCODINGS

Instruction and Format	Encoding
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 000:15mod xmmreg1 r/m
yymmreg2 with yymmreg3 to yymmreg1	C4: rxb0_1: w yymmreg2 100:15:11 yymmreg1 yymmreg3
yymmreg2 with mem to yymmreg1	C4: rxb0_1: w yymmreg2 100:15mod yymmreg1 r/m
<b>UNPCKLPS – Unpack and Interleave Low Packed Single-Precision Floating-Point Value</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 000:14:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 000:14mod xmmreg1 r/m
yymmreg2 with yymmreg3 to yymmreg1	C4: rxb0_1: w yymmreg2 100:14:11 yymmreg1 yymmreg3
yymmreg2 with mem to yymmreg1	C4: rxb0_1: w yymmreg2 100:14mod yymmreg1 r/m
<b>VXORPS – Bitwise Logical XOR for Single-Precision Floating-Point Values</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 000:57:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_1: w xmmreg2 000:57:mod xmmreg1 r/m
xmmreglo2 with xmmreglo3 to xmmreg1	C5: r_xmmreglo2 000:57:11 xmmreg1 xmmreglo3
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 000:57:mod xmmreg1 r/m
yymmreg2 with yymmreg3 to yymmreg1	C4: rxb0_1: w yymmreg2 100:57:11 yymmreg1 yymmreg3
yymmreg2 with mem to yymmreg1	C4: rxb0_1: w yymmreg2 100:57:mod yymmreg1 r/m
yymmreglo2 with yymmreglo3 to yymmreg1	C5: r_yymmreglo2 100:57:11 yymmreg1 yymmreglo3
yymmreglo2 with mem to yymmreg1	C5: r_yymmreglo2 100:57:mod yymmreg1 r/m
<b>VBROADCAST –Load with Broadcast</b>	
mem to xmmreg1	C4: rxb0_2: 0_F 001:18:mod xmmreg1 r/m
mem to yymmreg1	C4: rxb0_2: 0_F 101:18:mod yymmreg1 r/m
mem to yymmreg1	C4: rxb0_2: 0_F 101:19:mod yymmreg1 r/m
mem to yymmreg1	C4: rxb0_2: 0_F 101:1A:mod yymmreg1 r/m
<b>VEEXTRACTF128 – Extract Packed Floating-Point Values</b>	
yymmreg2 to xmmreg1, imm8	C4: rxb0_3: 0_F 001:19:11 xmmreg1 yymmreg2: imm
yymmreg2 to mem, imm8	C4: rxb0_3: 0_F 001:19:mod r/m yymmreg2: imm
<b>VINSERTF128 – Insert Packed Floating-Point Values</b>	

Instruction and Format	Encoding
xmmreg3 and merge with ymmreg2 to ymmreg1, imm8	C4: rxb0_3: 0 ymmreg2 101:18:11 ymmreg1 xmmreg3: imm
mem and merge with ymmreg2 to ymmreg1, imm8	C4: rxb0_3: 0 ymmreg2 101:18:mod ymmreg1 r/m: imm
<b>VPERMILPD – Permute Double-Precision Floating-Point Values</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_2: 0 xmmreg2 001:0D:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_2: 0 xmmreg2 001:0D:mod xmmreg1 r/m
ymmreg2 with ymmreg3 to ymmreg1	C4: rxb0_2: 0 ymmreg2 101:0D:11 ymmreg1 ymmreg3
ymmreg2 with mem to ymmreg1	C4: rxb0_2: 0 ymmreg2 101:0D:mod ymmreg1 r/m
xmmreg2 to xmmreg1, imm	C4: rxb0_3: 0_F 001:05:11 xmmreg1 xmmreg2: imm
mem to xmmreg1, imm	C4: rxb0_3: 0_F 001:05:mod xmmreg1 r/m: imm
ymmreg2 to ymmreg1, imm	C4: rxb0_3: 0_F 101:05:11 ymmreg1 ymmreg2: imm
mem to ymmreg1, imm	C4: rxb0_3: 0_F 101:05:mod ymmreg1 r/m: imm
<b>VPERMILPS – Permute Single-Precision Floating-Point Values</b>	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_2: 0 xmmreg2 001:0C:11 xmmreg1 xmmreg3
xmmreg2 with mem to xmmreg1	C4: rxb0_2: 0 xmmreg2 001:0C:mod xmmreg1 r/m
xmmreg2 to xmmreg1, imm	C4: rxb0_3: 0_F 001:04:11 xmmreg1 xmmreg2: imm
mem to xmmreg1, imm	C4: rxb0_3: 0_F 001:04:mod xmmreg1 r/m: imm
ymmreg2 with ymmreg3 to ymmreg1	C4: rxb0_2: 0 ymmreg2 101:0C:11 ymmreg1 ymmreg3
ymmreg2 with mem to ymmreg1	C4: rxb0_2: 0 ymmreg2 101:0C:mod ymmreg1 r/m
ymmreg2 to ymmreg1, imm	C4: rxb0_3: 0_F 101:04:11 ymmreg1 ymmreg2: imm
mem to ymmreg1, imm	C4: rxb0_3: 0_F 101:04:mod ymmreg1 r/m: imm
<b>VPERM2F128 – Permute Floating-Point Values</b>	
ymmreg2 with ymmreg3 to ymmreg1	C4: rxb0_3: 0 ymmreg2 101:06:11 ymmreg1 ymmreg3: imm
ymmreg2 with mem to ymmreg1	C4: rxb0_3: 0 ymmreg2 101:06:mod ymmreg1 r/m: imm
<b>VTESTPD/VTESTPS – Packed Bit Test</b>	
xmmreg2 to xmmreg1	C4: rxb0_2: 0_F 001:0E:11 xmmreg2 xmmreg1
mem to xmmreg1	C4: rxb0_2: 0_F 001:0E:mod xmmreg2 r/m
ymmreg2 to ymmreg1	C4: rxb0_2: 0_F 101:0E:11 ymmreg2 ymmreg1

## INSTRUCTION FORMATS AND ENCODINGS

Instruction and Format	Encoding
mem to ymmreg1	C4: rxb0_2: 0_F 101:0E:mod ymmreg2 r/m
xmmreg2 to xmmreg1	C4: rxb0_2: 0_F 001:0F:11 xmmreg1 xmmreg2: imm
mem to xmmreg1	C4: rxb0_2: 0_F 001:0F:mod xmmreg1 r/m: imm
ymmreg2 to ymmreg1	C4: rxb0_2: 0_F 101:0F:11 ymmreg1 ymmreg2: imm
mem to ymmreg1	C4: rxb0_2: 0_F 101:0F:mod ymmreg1 r/m: imm

### NOTES:

1. The term “lo” refers to the lower eight registers, 0-7

## B.17 FLOATING-POINT INSTRUCTION FORMATS AND ENCODINGS

Table B-35 shows the five different formats used for floating-point instructions. In all cases, instructions are at least two bytes long and begin with the bit pattern 11011.

**Table B-38. General Floating-Point Instruction Formats**

Instruction										Optional Fields		
First Byte				Second Byte								
1	11011	OPA		1	mod		1	OPB	r/m		s-i-b	disp
2	11011	MF		OPA	mod		OPB		r/m		s-i-b	disp
3	11011	d	P	OPA	1	1	OPB	R	ST(i)			
4	11011	0	0	1	1	1	1	OP				
5	11011	0	1	1	1	1	1	OP				
	15–11	10	9	8	7	6	5	4	3	2	1	0

MF = Memory Format

00 – 32-bit real

01 – 32-bit integer

10 – 64-bit real

11 – 16-bit integer

P = Pop

0 – Do not pop stack

1 – Pop stack after operation

d = Destination

0 – Destination is ST(0)

1 – Destination is ST(i)

R XOR d = 0 – Destination OP Source

R XOR d = 1 – Source OP Destination

ST(i) = Register stack element *i*

000 = Stack Top

001 = Second stack element

.

.

.

111 = Eighth stack element

The Mod and R/M fields of the ModR/M byte have the same interpretation as the corresponding fields of the integer instructions. The SIB byte and disp (displacement) are optionally present in instructions that have Mod and R/M fields. Their presence depends on the values of Mod and R/M, as for integer instructions.

Table B-36 shows the formats and encodings of the floating-point instructions.

**Table B-39. Floating-Point Instruction Formats and Encodings**

Instruction and Format	Encoding
F2XM1 – Compute $2^{\text{ST}(0)} - 1$	11011 001 : 1111 0000
FABS – Absolute Value	11011 001 : 1110 0001
FADD – Add	

**Table B-39. Floating-Point Instruction Formats and Encodings (Contd.)**

Instruction and Format	Encoding
$ST(0) \leftarrow ST(0) + 32\text{-bit memory}$	11011 000 : mod 000 r/m
$ST(0) \leftarrow ST(0) + 64\text{-bit memory}$	11011 100 : mod 000 r/m
$ST(d) \leftarrow ST(0) + ST(i)$	11011 d00 : 11 000 ST(i)
<b>FADDP - Add and Pop</b>	
$ST(0) \leftarrow ST(0) + ST(i)$	11011 110 : 11 000 ST(i)
<b>FBLD - Load Binary Coded Decimal</b>	11011 111 : mod 100 r/m
<b>FBSTP - Store Binary Coded Decimal and Pop</b>	11011 111 : mod 110 r/m
<b>FCHS - Change Sign</b>	11011 001 : 1110 0000
<b>FCLEX - Clear Exceptions</b>	11011 011 : 1110 0010
<b>FCOM - Compare Real</b>	
32-bit memory	11011 000 : mod 010 r/m
64-bit memory	11011 100 : mod 010 r/m
ST(i)	11011 000 : 11 010 ST(i)
<b>FCOMP - Compare Real and Pop</b>	
32-bit memory	11011 000 : mod 011 r/m
64-bit memory	11011 100 : mod 011 r/m
ST(i)	11011 000 : 11 011 ST(i)
<b>FCOMPP - Compare Real and Pop Twice</b>	11011 110 : 11 011 001
<b>FCOMIP - Compare Real, Set EFLAGS, and Pop</b>	11011 111 : 11 110 ST(i)
<b>FCOS - Cosine of ST(0)</b>	11011 001 : 1111 1111
<b>FDECSTP - Decrement Stack-Top Pointer</b>	11011 001 : 1111 0110
<b>FDIV - Divide</b>	
$ST(0) \leftarrow ST(0) \div 32\text{-bit memory}$	11011 000 : mod 110 r/m
$ST(0) \leftarrow ST(0) \div 64\text{-bit memory}$	11011 100 : mod 110 r/m
$ST(d) \leftarrow ST(0) \div ST(i)$	11011 d00 : 1111 R ST(i)
<b>FDIVP - Divide and Pop</b>	
$ST(0) \leftarrow ST(0) \div ST(i)$	11011 110 : 1111 1 ST(i)
<b>FDIVR - Reverse Divide</b>	
$ST(0) \leftarrow 32\text{-bit memory} \div ST(0)$	11011 000 : mod 111 r/m
$ST(0) \leftarrow 64\text{-bit memory} \div ST(0)$	11011 100 : mod 111 r/m

**Table B-39. Floating-Point Instruction Formats and Encodings (Contd.)**

Instruction and Format	Encoding
$ST(d) \leftarrow ST(i) \div ST(0)$	11011 d00 : 1111 R ST(i)
<b>FDIVRP - Reverse Divide and Pop</b>	
$ST(0) \leftarrow ST(i) \div ST(0)$	11011 110 : 1111 0 ST(i)
<b>FFREE - Free ST(i) Register</b>	11011 101 : 1100 0 ST(i)
<b>FIADD - Add Integer</b>	
$ST(0) \leftarrow ST(0) + 16\text{-bit memory}$	11011 110 : mod 000 r/m
$ST(0) \leftarrow ST(0) + 32\text{-bit memory}$	11011 010 : mod 000 r/m
<b>FICom - Compare Integer</b>	
16-bit memory	11011 110 : mod 010 r/m
32-bit memory	11011 010 : mod 010 r/m
<b>FIComp - Compare Integer and Pop</b>	
16-bit memory	11011 110 : mod 011 r/m
32-bit memory	11011 010 : mod 011 r/m
<b>FIDIV</b>	
$ST(0) \leftarrow ST(0) \div 16\text{-bit memory}$	11011 110 : mod 110 r/m
$ST(0) \leftarrow ST(0) \div 32\text{-bit memory}$	11011 010 : mod 110 r/m
<b>FIDIVR</b>	
$ST(0) \leftarrow 16\text{-bit memory} \div ST(0)$	11011 110 : mod 111 r/m
$ST(0) \leftarrow 32\text{-bit memory} \div ST(0)$	11011 010 : mod 111 r/m
<b>FILD - Load Integer</b>	
16-bit memory	11011 111 : mod 000 r/m
32-bit memory	11011 011 : mod 000 r/m
64-bit memory	11011 111 : mod 101 r/m
<b>FIMUL</b>	
$ST(0) \leftarrow ST(0) \times 16\text{-bit memory}$	11011 110 : mod 001 r/m
$ST(0) \leftarrow ST(0) \times 32\text{-bit memory}$	11011 010 : mod 001 r/m
<b>FINCSTP - Increment Stack Pointer</b>	11011 001 : 1111 0111
<b>FINIT - Initialize Floating-Point Unit</b>	
<b>FIST - Store Integer</b>	
16-bit memory	11011 111 : mod 010 r/m



**Table B-39. Floating-Point Instruction Formats and Encodings (Contd.)**

Instruction and Format	Encoding
32-bit memory	11011 011 : mod 010 r/m
<b>FISTP - Store Integer and Pop</b>	
16-bit memory	11011 111 : mod 011 r/m
32-bit memory	11011 011 : mod 011 r/m
64-bit memory	11011 111 : mod 111 r/m
<b>FISUB</b>	
$ST(0) \leftarrow ST(0) - 16\text{-bit memory}$	11011 110 : mod 100 r/m
$ST(0) \leftarrow ST(0) - 32\text{-bit memory}$	11011 010 : mod 100 r/m
<b>FISUBR</b>	
$ST(0) \leftarrow 16\text{-bit memory} - ST(0)$	11011 110 : mod 101 r/m
$ST(0) \leftarrow 32\text{-bit memory} - ST(0)$	11011 010 : mod 101 r/m
<b>FLD - Load Real</b>	
32-bit memory	11011 001 : mod 000 r/m
64-bit memory	11011 101 : mod 000 r/m
80-bit memory	11011 011 : mod 101 r/m
$ST(i)$	11011 001 : 11 000 $ST(i)$
<b>FLD1 - Load +1.0 into <math>ST(0)</math></b>	11011 001 : 1110 1000
<b>FLDCW - Load Control Word</b>	11011 001 : mod 101 r/m
<b>FLDENV - Load FPU Environment</b>	11011 001 : mod 100 r/m
<b>FLDL2E - Load <math>\log_2(\epsilon)</math> into <math>ST(0)</math></b>	11011 001 : 1110 1010
<b>FLDL2T - Load <math>\log_2(10)</math> into <math>ST(0)</math></b>	11011 001 : 1110 1001
<b>FLDLG2 - Load <math>\log_{10}(2)</math> into <math>ST(0)</math></b>	11011 001 : 1110 1100
<b>FLDLN2 - Load <math>\log_e(2)</math> into <math>ST(0)</math></b>	11011 001 : 1110 1101
<b>FLDPI - Load <math>\pi</math> into <math>ST(0)</math></b>	11011 001 : 1110 1011
<b>FLDZ - Load +0.0 into <math>ST(0)</math></b>	11011 001 : 1110 1110
<b>FMUL - Multiply</b>	
$ST(0) \leftarrow ST(0) \times 32\text{-bit memory}$	11011 000 : mod 001 r/m
$ST(0) \leftarrow ST(0) \times 64\text{-bit memory}$	11011 100 : mod 001 r/m
$ST(d) \leftarrow ST(0) \times ST(i)$	11011 d00 : 1100 1 $ST(i)$
<b>FMULP - Multiply</b>	
$ST(i) \leftarrow ST(0) \times ST(i)$	11011 110 : 1100 1 $ST(i)$

Table B-39. Floating-Point Instruction Formats and Encodings (Contd.)

Instruction and Format	Encoding
<b>FNOP - No Operation</b>	11011 001 : 1101 0000
<b>FPATAN - Partial Arctangent</b>	11011 001 : 1111 0011
<b>FPREM - Partial Remainder</b>	11011 001 : 1111 1000
<b>FPREM1 - Partial Remainder (IEEE)</b>	11011 001 : 1111 0101
<b>FPTAN - Partial Tangent</b>	11011 001 : 1111 0010
<b>FRNDINT - Round to Integer</b>	11011 001 : 1111 1100
<b>FRSTOR - Restore FPU State</b>	11011 101 : mod 100 r/m
<b>FSAVE - Store FPU State</b>	11011 101 : mod 110 r/m
<b>FSCALE - Scale</b>	11011 001 : 1111 1101
<b>FSIN - Sine</b>	11011 001 : 1111 1110
<b>FSINCOS - Sine and Cosine</b>	11011 001 : 1111 1011
<b>FSQRT - Square Root</b>	11011 001 : 1111 1010
<b>FST - Store Real</b>	
32-bit memory	11011 001 : mod 010 r/m
64-bit memory	11011 101 : mod 010 r/m
ST(i)	11011 101 : 11 010 ST(i)
<b>FSTCW - Store Control Word</b>	11011 001 : mod 111 r/m
<b>FSTENV - Store FPU Environment</b>	11011 001 : mod 110 r/m
<b>FSTP - Store Real and Pop</b>	
32-bit memory	11011 001 : mod 011 r/m
64-bit memory	11011 101 : mod 011 r/m
80-bit memory	11011 011 : mod 111 r/m
ST(i)	11011 101 : 11 011 ST(i)
<b>FSTSW - Store Status Word into AX</b>	11011 111 : 1110 0000
<b>FSTSW - Store Status Word into Memory</b>	11011 101 : mod 111 r/m
<b>FSUB - Subtract</b>	
ST(0) ← ST(0) - 32-bit memory	11011 000 : mod 100 r/m
ST(0) ← ST(0) - 64-bit memory	11011 100 : mod 100 r/m
ST(d) ← ST(0) - ST(i)	11011 d00 : 1110 R ST(i)
<b>FSUBP - Subtract and Pop</b>	
ST(0) ← ST(0) - ST(i)	11011 110 : 1110 1 ST(i)

**Table B-39. Floating-Point Instruction Formats and Encodings (Contd.)**

<b>Instruction and Format</b>	<b>Encoding</b>
<b>FSUBR - Reverse Subtract</b>	
ST(0) ← 32-bit memory - ST(0)	11011 000 : mod 101 r/m
ST(0) ← 64-bit memory - ST(0)	11011 100 : mod 101 r/m
ST(d) ← ST(i) - ST(0)	11011 d00 : 1110 R ST(i)
<b>FSUBRP - Reverse Subtract and Pop</b>	
ST(i) ← ST(i) - ST(0)	11011 110 : 1110 0 ST(i)
<b>FTST - Test</b>	11011 001 : 1110 0100
<b>FUCOM - Unordered Compare Real</b>	11011 101 : 1110 0 ST(i)
<b>FUCOMP - Unordered Compare Real and Pop</b>	11011 101 : 1110 1 ST(i)
<b>FUCOMPP - Unordered Compare Real and Pop Twice</b>	11011 010 : 1110 1001
<b>FUCOMI - Unorderd Compare Real and Set EFLAGS</b>	11011 011 : 11 101 ST(i)
<b>FUCOMIP - Unorderd Compare Real, Set EFLAGS, and Pop</b>	11011 111 : 11 101 ST(i)
<b>FXAM - Examine</b>	11011 001 : 1110 0101
<b>FXCH - Exchange ST(0) and ST(i)</b>	11011 001 : 1100 1 ST(i)
<b>FXTRACT - Extract Exponent and Significand</b>	11011 001 : 1111 0100
<b>FYL2X - ST(1) × log<sub>2</sub>(ST(0))</b>	11011 001 : 1111 0001
<b>FYL2XP1 - ST(1) × log<sub>2</sub>(ST(0) + 1.0)</b>	11011 001 : 1111 1001
<b>FWAIT - Wait until FPU Ready</b>	1001 1011

## B.18 VMX INSTRUCTIONS

Table B-40 describes virtual-machine extensions (VMX).

**Table B-40. Encodings for VMX Instructions**

Instruction and Format	Encoding
<b>INVEPT—Invalidate Cached EPT Mappings</b>	
Descriptor m128 according to reg	01100110 00001111 00111000 10000000: mod reg r/m
<b>INVVPID—Invalidate Cached VPID Mappings</b>	
Descriptor m128 according to reg	01100110 00001111 00111000 10000001: mod reg r/m
<b>VMCALL—Call to VM Monitor</b>	
Call VMM: causes VM exit.	00001111 00000001 11000001
<b>VMCLEAR—Clear Virtual-Machine Control Structure</b>	
mem32:VMCS_data_ptr	01100110 00001111 11000111: mod 110 r/m
mem64:VMCS_data_ptr	01100110 00001111 11000111: mod 110 r/m
<b>VMFUNC—Invoke VM Function</b>	
Invoke VM function specified in EAX	00001111 00000001 11010100
<b>VMLAUNCH—Launch Virtual Machine</b>	
Launch VM managed by Current_VMCS	00001111 00000001 11000010
<b>VMRESUME—Resume Virtual Machine</b>	
Resume VM managed by Current_VMCS	00001111 00000001 11000011
<b>VMPTRLD—Load Pointer to Virtual-Machine Control Structure</b>	
mem32 to Current_VMCS_ptr	00001111 11000111: mod 110 r/m
mem64 to Current_VMCS_ptr	00001111 11000111: mod 110 r/m
<b>VMPTRST—Store Pointer to Virtual-Machine Control Structure</b>	
Current_VMCS_ptr to mem32	00001111 11000111: mod 111 r/m
Current_VMCS_ptr to mem64	00001111 11000111: mod 111 r/m
<b>VMREAD—Read Field from Virtual-Machine Control Structure</b>	

**Table B-40. Encodings for VMX Instructions**

Instruction and Format	Encoding
r32 ( <i>VMCS_fieldn</i> ) to r32 r32 ( <i>VMCS_fieldn</i> ) to mem32 r64 ( <i>VMCS_fieldn</i> ) to r64 r64 ( <i>VMCS_fieldn</i> ) to mem64	00001111 01111000: 11 reg2 reg1 00001111 01111000: mod r32 r/m 00001111 01111000: 11 reg2 reg1 00001111 01111000: mod r64 r/m
<b>VMWRITE—Write Field to Virtual-Machine Control Structure</b>	
r32 to r32 ( <i>VMCS_fieldn</i> ) mem32 to r32 ( <i>VMCS_fieldn</i> ) r64 to r64 ( <i>VMCS_fieldn</i> ) mem64 to r64 ( <i>VMCS_fieldn</i> )	00001111 01111001: 11 reg1 reg2 00001111 01111001: mod r32 r/m 00001111 01111001: 11 reg1 reg2 00001111 01111001: mod r64 r/m
<b>VMXOFF—Leave VMX Operation</b>	
Leave VMX.	00001111 00000001 11000100
<b>VMXON—Enter VMX Operation</b>	
Enter VMX.	11110011 00001111 11000111: mod 110 r/m

## B.19 SMX INSTRUCTIONS

Table B-38 describes Safer Mode extensions (VMX). **GETSEC leaf functions are selected by a valid value in EAX on input.**

**Table B-41. Encodings for SMX Instructions**

Instruction and Format	Encoding
<b>GETSEC—GETSEC leaf functions are selected by the value in EAX on input</b>	
<i>GETSEC[CAPABILITIES].</i>	00001111 00110111 (EAX= 0)
<i>GETSEC[ENTERACCS].</i>	00001111 00110111 (EAX= 2)
<i>GETSEC[EXITAC].</i>	00001111 00110111 (EAX= 3)
<i>GETSEC[SENDER].</i>	00001111 00110111 (EAX= 4)
<i>GETSEC[SEXIT].</i>	00001111 00110111 (EAX= 5)
<i>GETSEC[PARAMETERS].</i>	00001111 00110111 (EAX= 6)
<i>GETSEC[SMCTRL].</i>	00001111 00110111 (EAX= 7)
<i>GETSEC[WAKEUP].</i>	00001111 00110111 (EAX= 8)



# APPENDIX C

## INTEL® C/C++ COMPILER INTRINSICS AND FUNCTIONAL EQUIVALENTS

---

The two tables in this appendix itemize the Intel C/C++ compiler intrinsics and functional equivalents for the Intel MMX technology, SSE, SSE2, SSE3, and SSSE3 instructions.

There may be additional intrinsics that do not have an instruction equivalent. It is strongly recommended that the reader reference the compiler documentation for the complete list of supported intrinsics. Please refer to <http://www.intel.com/support/performance/tools/>.

Table C-1 presents simple intrinsics and Table C-2 presents composite intrinsics. Some intrinsics are “composites” because they require more than one instruction to implement them.

Intel C/C++ Compiler intrinsic names reflect the following naming conventions:

`_mm_<intrin_op>_<suffix>`

where:

<code>&lt;intrin_op&gt;</code>	Indicates the intrinsics basic operation; for example, add for addition and sub for subtraction
<code>&lt;suffix&gt;</code>	Denotes the type of data operated on by the instruction. The first one or two letters of each suffix denotes whether the data is packed (p), extended packed (ep), or scalar (s).

The remaining letters denote the type:

s	single-precision floating point
d	double-precision floating point
i128	signed 128-bit integer
i64	signed 64-bit integer
u64	unsigned 64-bit integer
i32	signed 32-bit integer
u32	unsigned 32-bit integer
i16	signed 16-bit integer
u16	unsigned 16-bit integer
i8	signed 8-bit integer
u8	unsigned 8-bit integer

The variable `r` is generally used for the intrinsic's return value. A number appended to a variable name indicates the element of a packed object. For example, `r0` is the lowest word of `r`.



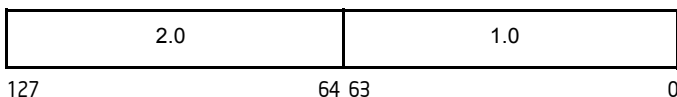
The packed values are represented in right-to-left order, with the lowest value being used for scalar operations. Consider the following example operation:

```
double a[2] = {1.0, 2.0};
__m128d t = _mm_load_pd(a);
```

The result is the same as either of the following:

```
__m128d t = _mm_set_pd(2.0, 1.0);
__m128d t = _mm_setr_pd(1.0, 2.0);
```

In other words, the XMM register that holds the value t will look as follows:



The “scalar” element is 1.0. Due to the nature of the instruction, some intrinsics require their arguments to be immediates (constant integer literals).

To use an intrinsic in your code, insert a line with the following syntax:

```
data_type intrinsic_name (parameters)
```

Where:

- data\_type
Is the return data type, which can be either void, int, \_\_m64, \_\_m128, \_\_m128d, or \_\_m128i. Only the \_\_mm\_empty intrinsic returns void.
- intrinsic\_name
Is the name of the intrinsic, which behaves like a function that you can use in your C/C++ code instead of in-lining the actual instruction.
- parameters
Represents the parameters required by each intrinsic.

## C.1 SIMPLE INTRINSICS

### NOTE

For detailed descriptions of the intrinsics in Table C-1, see the corresponding mnemonic in Chapter 3 in the “Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A”, or Chapter 4, “Instruction Set Reference, M-Z” in the “Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2B”.

Table C-1. Simple Intrinsics

Mnemonic	Intrinsic
ADDDP	<code>__m128d _mm_add_pd(__m128d a, __m128d b)</code>
ADDPS	<code>__m128 _mm_add_ps(__m128 a, __m128 b)</code>
ADDSD	<code>__m128d _mm_add_sd(__m128d a, __m128d b)</code>
ADDSS	<code>__m128 _mm_add_ss(__m128 a, __m128 b)</code>
ADDSUBPD	<code>__m128d _mm_addsub_pd(__m128d a, __m128d b)</code>
ADDSUBPS	<code>__m128 _mm_addsub_ps(__m128 a, __m128 b)</code>
AESDEC	<code>__m128i _mm_aesdec (__m128i, __m128i)</code>
AESDECLAST	<code>__m128i _mm_aesdeclast (__m128i, __m128i)</code>
AESENC	<code>__m128i _mm_aesenc (__m128i, __m128i)</code>
AESENCLAST	<code>__m128i _mm_aesenclast (__m128i, __m128i)</code>
AESIMC	<code>__m128i _mm_aesimc (__m128i)</code>
AESKEYGENASSIST	<code>__m128i _mm_aesimc (__m128i, const int)</code>
ANDNPD	<code>__m128d _mm_andnot_pd(__m128d a, __m128d b)</code>
ANDNPS	<code>__m128 _mm_andnot_ps(__m128 a, __m128 b)</code>
ANDPD	<code>__m128d _mm_and_pd(__m128d a, __m128d b)</code>
ANDPS	<code>__m128 _mm_and_ps(__m128 a, __m128 b)</code>
BLENDDP	<code>__m128d _mm_blend_pd(__m128d v1, __m128d v2, const int mask)</code>
BLENDPS	<code>__m128 _mm_blend_ps(__m128 v1, __m128 v2, const int mask)</code>
BLENDVPD	<code>__m128d _mm_blendv_pd(__m128d v1, __m128d v2, __m128d v3)</code>
BLENDVPS	<code>__m128 _mm_blendv_ps(__m128 v1, __m128 v2, __m128 v3)</code>
CLFLUSH	<code>void _mm_clflush(void const *p)</code>
CMPDP	<code>__m128d _mm_cmpeq_pd(__m128d a, __m128d b)</code>
	<code>__m128d _mm_cmplt_pd(__m128d a, __m128d b)</code>
	<code>__m128d _mm_cmple_pd(__m128d a, __m128d b)</code>
	<code>__m128d _mm_cmpgt_pd(__m128d a, __m128d b)</code>
	<code>__m128d _mm_cmpge_pd(__m128d a, __m128d b)</code>
	<code>__m128d _mm_cmpneq_pd(__m128d a, __m128d b)</code>
	<code>__m128d _mm_cmpnlt_pd(__m128d a, __m128d b)</code>
	<code>__m128d _mm_cmpngt_pd(__m128d a, __m128d b)</code>
	<code>__m128d _mm_cmpnge_pd(__m128d a, __m128d b)</code>
	<code>__m128d _mm_cmpord_pd(__m128d a, __m128d b)</code>
	<code>__m128d _mm_cmpunord_pd(__m128d a, __m128d b)</code>
<code>__m128d _mm_cmpnle_pd(__m128d a, __m128d b)</code>	
CMPPS	<code>__m128 _mm_cmpeq_ps(__m128 a, __m128 b)</code>
	<code>__m128 _mm_cmplt_ps(__m128 a, __m128 b)</code>

**Table C-1. Simple Intrinsics (Contd.)**

Mnemonic	Intrinsic
	__m128_mm_cmple_ps(__m128 a, __m128 b)
	__m128_mm_cmpgt_ps(__m128 a, __m128 b)
	__m128_mm_cmpge_ps(__m128 a, __m128 b)
	__m128_mm_cmpneq_ps(__m128 a, __m128 b)
	__m128_mm_cmpnlt_ps(__m128 a, __m128 b)
	__m128_mm_cmpngt_ps(__m128 a, __m128 b)
	__m128_mm_cmpnge_ps(__m128 a, __m128 b)
	__m128_mm_cmpord_ps(__m128 a, __m128 b)
	__m128_mm_cmpunord_ps(__m128 a, __m128 b)
	__m128_mm_cmpnle_ps(__m128 a, __m128 b)
CMPSD	__m128d_mm_cmpeq_sd(__m128d a, __m128d b)
	__m128d_mm_cmplt_sd(__m128d a, __m128d b)
	__m128d_mm_cmple_sd(__m128d a, __m128d b)
	__m128d_mm_cmpgt_sd(__m128d a, __m128d b)
	__m128d_mm_cmpge_sd(__m128d a, __m128d b)
	__m128d_mm_cmpneq_sd(__m128d a, __m128d b)
	__m128d_mm_cmpnlt_sd(__m128d a, __m128d b)
	__m128d_mm_cmpnle_sd(__m128d a, __m128d b)
	__m128d_mm_cmpngt_sd(__m128d a, __m128d b)
	__m128d_mm_cmpnge_sd(__m128d a, __m128d b)
	__m128d_mm_cmpord_sd(__m128d a, __m128d b)
	__m128d_mm_cmpunord_sd(__m128d a, __m128d b)
CMPSS	__m128_mm_cmpeq_ss(__m128 a, __m128 b)
	__m128_mm_cmplt_ss(__m128 a, __m128 b)
	__m128_mm_cmple_ss(__m128 a, __m128 b)
	__m128_mm_cmpgt_ss(__m128 a, __m128 b)
	__m128_mm_cmpge_ss(__m128 a, __m128 b)
	__m128_mm_cmpneq_ss(__m128 a, __m128 b)
	__m128_mm_cmpnlt_ss(__m128 a, __m128 b)
	__m128_mm_cmpnle_ss(__m128 a, __m128 b)
	__m128_mm_cmpngt_ss(__m128 a, __m128 b)
	__m128_mm_cmpnge_ss(__m128 a, __m128 b)
	__m128_mm_cmpord_ss(__m128 a, __m128 b)
	__m128_mm_cmpunord_ss(__m128 a, __m128 b)
COMISD	int_mm_comieq_sd(__m128d a, __m128d b)

Table C-1. Simple Intrinsics (Contd.)

Mnemonic	Intrinsic
	int __mm_comilt_sd(__m128d a, __m128d b)
	int __mm_comile_sd(__m128d a, __m128d b)
	int __mm_comigt_sd(__m128d a, __m128d b)
	int __mm_comige_sd(__m128d a, __m128d b)
	int __mm_comineq_sd(__m128d a, __m128d b)
COMISS	int __mm_comieq_ss(__m128 a, __m128 b)
	int __mm_comilt_ss(__m128 a, __m128 b)
	int __mm_comile_ss(__m128 a, __m128 b)
	int __mm_comigt_ss(__m128 a, __m128 b)
	int __mm_comige_ss(__m128 a, __m128 b)
	int __mm_comineq_ss(__m128 a, __m128 b)
CRC32	unsigned int __mm_crc32_u8(unsigned int crc, unsigned char data)
	unsigned int __mm_crc32_u16(unsigned int crc, unsigned short data)
	unsigned int __mm_crc32_u32(unsigned int crc, unsigned int data)
	unsigned __int64 __mm_crc32_u64(unsigned __int64 crc, unsigned __int64 data)
CVTDQ2PD	__m128d __mm_cvtepi32_pd(__m128i a)
CVTDQ2PS	__m128 __mm_cvtepi32_ps(__m128i a)
CVTPD2DQ	__m128i __mm_cvtpd_epi32(__m128d a)
CVTPD2PI	__m64 __mm_cvtpd_pi32(__m128d a)
CVTPD2PS	__m128 __mm_cvtpd_ps(__m128d a)
CVTPI2PD	__m128d __mm_cvtpi32_pd(__m64 a)
CVTPI2PS	__m128 __mm_cvt_pi2ps(__m128 a, __m64 b) __m128 __mm_cvtpi32_ps(__m128 a, __m64 b)
CVTPS2DQ	__m128i __mm_cvtps_epi32(__m128 a)
CVTPS2PD	__m128d __mm_cvtps_pd(__m128 a)
CVTPS2PI	__m64 __mm_cvt_ps2pi(__m128 a) __m64 __mm_cvtps_pi32(__m128 a)
CVTSD2SI	int __mm_cvtsd_si32(__m128d a)
CVTSD2SS	__m128 __mm_cvtsd_ss(__m128 a, __m128d b)
CVTSI2SD	__m128d __mm_cvtsi32_sd(__m128d a, int b)
CVTSI2SS	__m128 __mm_cvt_si2ss(__m128 a, int b) __m128 __mm_cvtsi32_ss(__m128 a, int b) __m128 __mm_cvtsi64_ss(__m128 a, __int64 b)
CVTSS2SD	__m128d __mm_cvtss_sd(__m128d a, __m128 b)
CVTSS2SI	int __mm_cvt_ss2si(__m128 a) int __mm_cvtss_si32(__m128 a)
CVTTPD2DQ	__m128i __mm_cvttpd_epi32(__m128d a)

Table C-1. Simple Intrinsics (Contd.)

Mnemonic	Intrinsic
CVTTPD2PI	__m64 _mm_cvttpd_pi32(__m128d a)
CVTTPS2DQ	__m128i _mm_cvttps_epi32(__m128 a)
CVTTPS2PI	__m64 _mm_cvtt_ps2pi(__m128 a) __m64 _mm_cvttps_pi32(__m128 a)
CVTTSD2SI	int _mm_cvttss_si32(__m128d a)
CVTTSS2SI	int _mm_cvtt_ss2si(__m128 a) int _mm_cvttss_si32(__m128 a) __m64 _mm_cvtsi32_si64(int i) int _mm_cvtsi64_si32(__m64 m)
DIVPD	__m128d _mm_div_pd(__m128d a, __m128d b)
DIVPS	__m128 _mm_div_ps(__m128 a, __m128 b)
DIVSD	__m128d _mm_div_sd(__m128d a, __m128d b)
DIVSS	__m128 _mm_div_ss(__m128 a, __m128 b)
DPPD	__m128d _mm_dp_pd(__m128d a, __m128d b, const int mask)
DPPS	__m128 _mm_dp_ps(__m128 a, __m128 b, const int mask)
EMMS	void _mm_empty()
EXTRACTPS	int _mm_extract_ps(__m128 src, const int ndx)
HADDPD	__m128d _mm_hadd_pd(__m128d a, __m128d b)
HADDPS	__m128 _mm_hadd_ps(__m128 a, __m128 b)
HSUBPD	__m128d _mm_hsub_pd(__m128d a, __m128d b)
HSUBPS	__m128 _mm_hsub_ps(__m128 a, __m128 b)
INSERTPS	__m128 _mm_insert_ps(__m128 dst, __m128 src, const int ndx)
LDDQU	__m128i _mm_lddqu_si128(__m128i const *p)
LDMXCSR	__mm_setcsr(unsigned int i)
LFENCE	void _mm_lfence(void)
MASKMOVDQU	void _mm_maskmoveu_si128(__m128i d, __m128i n, char *p)
MASKMOVQ	void _mm_maskmove_si64(__m64 d, __m64 n, char *p)
MAXPD	__m128d _mm_max_pd(__m128d a, __m128d b)
MAXPS	__m128 _mm_max_ps(__m128 a, __m128 b)
MAXSD	__m128d _mm_max_sd(__m128d a, __m128d b)
MAXSS	__m128 _mm_max_ss(__m128 a, __m128 b)
MFENCE	void _mm_mfence(void)
MINPD	__m128d _mm_min_pd(__m128d a, __m128d b)
MINPS	__m128 _mm_min_ps(__m128 a, __m128 b)
MINSD	__m128d _mm_min_sd(__m128d a, __m128d b)
MINSS	__m128 _mm_min_ss(__m128 a, __m128 b)

Table C-1. Simple Intrinsics (Contd.)

Mnemonic	Intrinsic
MONITOR	void __mm_monitor(void const *p, unsigned extensions, unsigned hints)
MOVAPD	__m128d __mm_load_pd(double * p)
	void __mm_store_pd(double *p, __m128d a)
MOVAPS	__m128 __mm_load_ps(float * p)
	void __mm_store_ps(float *p, __m128 a)
MOVD	__m128i __mm_cvtsi32_si128(int a)
	int __mm_cvtsi128_si32(__m128i a)
	__m64 __mm_cvtsi32_si64(int a)
	int __mm_cvtsi64_si32(__m64 a)
MOVDDUP	__m128d __mm_movedup_pd(__m128d a)
	__m128d __mm_loaddup_pd(double const * dp)
MOVDDQA	__m128i __mm_load_si128(__m128i * p)
	void __mm_store_si128(__m128i *p, __m128i a)
MOVDDQU	__m128i __mm_loadu_si128(__m128i * p)
	void __mm_storeu_si128(__m128i *p, __m128i a)
MOVDDQ2Q	__m64 __mm_movepi64_pi64(__m128i a)
MOVHLPD	__m128 __mm_movehl_ps(__m128 a, __m128 b)
MOVHPD	__m128d __mm_loadh_pd(__m128d a, double * p)
	void __mm_storeh_pd(double * p, __m128d a)
MOVHPS	__m128 __mm_loadh_pi(__m128 a, __m64 * p)
	void __mm_storeh_pi(__m64 * p, __m128 a)
MOVLPD	__m128d __mm_loadl_pd(__m128d a, double * p)
	void __mm_storel_pd(double * p, __m128d a)
MOVLPS	__m128 __mm_loadl_pi(__m128 a, __m64 * p)
	void __mm_storel_pi(__m64 * p, __m128 a)
MOVLHPS	__m128 __mm_movelh_ps(__m128 a, __m128 b)
MOVMSKPD	int __mm_movemask_pd(__m128d a)
MOVMSKPS	int __mm_movemask_ps(__m128 a)
MOVNTDQA	__m128i __mm_stream_load_si128(__m128i *p)
MOVNTDQ	void __mm_stream_si128(__m128i * p, __m128i a)
MOVNTPD	void __mm_stream_pd(double * p, __m128d a)
MOVNTPS	void __mm_stream_ps(float * p, __m128 a)
MOVNTI	void __mm_stream_si32(int * p, int a)
MOVNTQ	void __mm_stream_pi(__m64 * p, __m64 a)
MOVQ	__m128i __mm_loadl_epi64(__m128i * p)

Table C-1. Simple Intrinsics (Contd.)

Mnemonic	Intrinsic
	void_mm_store_epi64(__m128i * p, __m128i a) __m128i_mm_move_epi64(__m128i a)
MOVQ2DQ	__m128i_mm_movpi64_epi64(__m64 a)
MOVSD	__m128d_mm_load_sd(double * p) void_mm_store_sd(double * p, __m128d a) __m128d_mm_move_sd(__m128d a, __m128d b)
MOVSHDUP	__m128_mm_movehdup_ps(__m128 a)
MOVSLDUP	__m128_mm_moveldup_ps(__m128 a)
MOVSS	__m128_mm_load_ss(float * p) void_mm_store_ss(float * p, __m128 a) __m128_mm_move_ss(__m128 a, __m128 b)
MOVUPD	__m128d_mm_loadu_pd(double * p) void_mm_storeu_pd(double *p, __m128d a)
MOVUPS	__m128_mm_loadu_ps(float * p) void_mm_storeu_ps(float *p, __m128 a)
MPSADBW	__m128i_mm_mpsadbw_epu8(__m128i s1, __m128i s2, const int mask)
MULPD	__m128d_mm_mul_pd(__m128d a, __m128d b)
MULPS	__m128_mm_mul_ss(__m128 a, __m128 b)
MULSD	__m128d_mm_mul_sd(__m128d a, __m128d b)
MULSS	__m128_mm_mul_ss(__m128 a, __m128 b)
MWAIT	void_mm_mwait(unsigned extensions, unsigned hints)
ORPD	__m128d_mm_or_pd(__m128d a, __m128d b)
ORPS	__m128_mm_or_ps(__m128 a, __m128 b)
PABSB	__m64_mm_abs_pi8 (__m64 a) __m128i_mm_abs_epi8 (__m128i a)
PABSD	__m64_mm_abs_pi32 (__m64 a) __m128i_mm_abs_epi32 (__m128i a)
PABSW	__m64_mm_abs_pi16 (__m64 a) __m128i_mm_abs_epi16 (__m128i a)
PACKSSWB	__m128i_mm_packs_epi16(__m128i m1, __m128i m2)
PACKSSWB	__m64_mm_packs_pi16(__m64 m1, __m64 m2)
PACKSSDW	__m128i_mm_packs_epi32 (__m128i m1, __m128i m2)
PACKSSDW	__m64_mm_packs_pi32 (__m64 m1, __m64 m2)
PACKUSDW	__m128i_mm_packus_epi32(__m128i m1, __m128i m2)
PACKUSWB	__m128i_mm_packus_epi16(__m128i m1, __m128i m2)

Table C-1. Simple Intrinsics (Contd.)

Mnemonic	Intrinsic
PACKUSWB	<code>__m64 _mm_packs_pu16(__m64 m1, __m64 m2)</code>
PADDB	<code>__m128i _mm_add_epi8(__m128i m1, __m128i m2)</code>
PADDB	<code>__m64 _mm_add_pi8(__m64 m1, __m64 m2)</code>
PADDW	<code>__m128i _mm_add_epi16(__m128i m1, __m128i m2)</code>
PADDW	<code>__m64 _mm_add_pi16(__m64 m1, __m64 m2)</code>
PADD	<code>__m128i _mm_add_epi32(__m128i m1, __m128i m2)</code>
PADD	<code>__m64 _mm_add_pi32(__m64 m1, __m64 m2)</code>
PADDQ	<code>__m128i _mm_add_epi64(__m128i m1, __m128i m2)</code>
PADDQ	<code>__m64 _mm_add_si64(__m64 m1, __m64 m2)</code>
PADDSB	<code>__m128i _mm_adds_epi8(__m128i m1, __m128i m2)</code>
PADDSB	<code>__m64 _mm_adds_pi8(__m64 m1, __m64 m2)</code>
PADDSW	<code>__m128i _mm_adds_epi16(__m128i m1, __m128i m2)</code>
PADDSW	<code>__m64 _mm_adds_pi16(__m64 m1, __m64 m2)</code>
PADDUSB	<code>__m128i _mm_adds_epu8(__m128i m1, __m128i m2)</code>
PADDUSB	<code>__m64 _mm_adds_pu8(__m64 m1, __m64 m2)</code>
PADDUSW	<code>__m128i _mm_adds_epu16(__m128i m1, __m128i m2)</code>
PADDUSW	<code>__m64 _mm_adds_pu16(__m64 m1, __m64 m2)</code>
PALIGNR	<code>__m64 _mm_alignr_pi8 (__m64 a, __m64 b, int n)</code> <code>__m128i _mm_alignr_epi8 (__m128i a, __m128i b, int n)</code>
PAND	<code>__m128i _mm_and_si128(__m128i m1, __m128i m2)</code>
PAND	<code>__m64 _mm_and_si64(__m64 m1, __m64 m2)</code>
PANDN	<code>__m128i _mm_andnot_si128(__m128i m1, __m128i m2)</code>
PANDN	<code>__m64 _mm_andnot_si64(__m64 m1, __m64 m2)</code>
PAUSE	<code>void _mm_pause(void)</code>
PAVGB	<code>__m128i _mm_avg_epu8(__m128i a, __m128i b)</code>
PAVGB	<code>__m64 _mm_avg_pu8(__m64 a, __m64 b)</code>
PAVGW	<code>__m128i _mm_avg_epu16(__m128i a, __m128i b)</code>
PAVGW	<code>__m64 _mm_avg_pu16(__m64 a, __m64 b)</code>
PBLENDVB	<code>__m128i _mm_blendv_epi (__m128i v1, __m128i v2, __m128i mask)</code>
PBLENDW	<code>__m128i _mm_blend_epi16(__m128i v1, __m128i v2, const int mask)</code>
PCLMULQDQ	<code>__m128i _mm_clmulepi64_si128 (__m128i, __m128i, const int)</code>
PCMPEQB	<code>__m128i _mm_cmpeq_epi8(__m128i m1, __m128i m2)</code>
PCMPEQB	<code>__m64 _mm_cmpeq_pi8(__m64 m1, __m64 m2)</code>
PCMPEQQ	<code>__m128i _mm_cmpeq_epi64(__m128i a, __m128i b)</code>
PCMPEQW	<code>__m128i _mm_cmpeq_epi16 (__m128i m1, __m128i m2)</code>



**Table C-1. Simple Intrinsics (Contd.)**

<b>Mnemonic</b>	<b>Intrinsic</b>
PCMPEQW	__m64 _mm_cmpeq_pi16 (__m64 m1, __m64 m2)
PCMPEQD	__m128i _mm_cmpeq_epi32(__m128i m1, __m128i m2)
PCMPEQD	__m64 _mm_cmpeq_pi32(__m64 m1, __m64 m2)
PCMPESTRI	int _mm_cmpestri (__m128i a, int la, __m128i b, int lb, const int mode)
	int _mm_cmpestra (__m128i a, int la, __m128i b, int lb, const int mode)
	int _mm_cmpestrc (__m128i a, int la, __m128i b, int lb, const int mode)
	int _mm_cmpestro (__m128i a, int la, __m128i b, int lb, const int mode)
	int _mm_cmpestrs (__m128i a, int la, __m128i b, int lb, const int mode)
	int _mm_cmpestrz (__m128i a, int la, __m128i b, int lb, const int mode)
PCMPESTRM	__m128i _mm_cmpestrm (__m128i a, int la, __m128i b, int lb, const int mode)
	int _mm_cmpestra (__m128i a, int la, __m128i b, int lb, const int mode)
	int _mm_cmpestrc (__m128i a, int la, __m128i b, int lb, const int mode)
	int _mm_cmpestro (__m128i a, int la, __m128i b, int lb, const int mode)
	int _mm_cmpestrs (__m128i a, int la, __m128i b, int lb, const int mode)
	int _mm_cmpestrz (__m128i a, int la, __m128i b, int lb, const int mode)
PCMPGTB	__m128i _mm_cmpgt_epi8 (__m128i m1, __m128i m2)
PCMPGTB	__m64 _mm_cmpgt_pi8 (__m64 m1, __m64 m2)
PCMPGTW	__m128i _mm_cmpgt_epi16(__m128i m1, __m128i m2)
PCMPGTW	__m64 _mm_cmpgt_pi16 (__m64 m1, __m64 m2)
PCMPGTD	__m128i _mm_cmpgt_epi32(__m128i m1, __m128i m2)
PCMPGTD	__m64 _mm_cmpgt_pi32(__m64 m1, __m64 m2)
PCMPISTRI	__m128i _mm_cmpestrm (__m128i a, int la, __m128i b, int lb, const int mode)
	int _mm_cmpestra (__m128i a, int la, __m128i b, int lb, const int mode)
	int _mm_cmpestrc (__m128i a, int la, __m128i b, int lb, const int mode)
	int _mm_cmpestro (__m128i a, int la, __m128i b, int lb, const int mode)
	int _mm_cmpestrs (__m128i a, int la, __m128i b, int lb, const int mode)
	int _mm_cmpistrz (__m128i a, __m128i b, const int mode)
PCMPISTRM	__m128i _mm_cmpistrm (__m128i a, __m128i b, const int mode)
	int _mm_cmpistra (__m128i a, __m128i b, const int mode)
	int _mm_cmpistrc (__m128i a, __m128i b, const int mode)
	int _mm_cmpistro (__m128i a, __m128i b, const int mode)
	int _mm_cmpistrs (__m128i a, __m128i b, const int mode)
	int _mm_cmpistrz (__m128i a, __m128i b, const int mode)
PCMPGTQ	__m128i _mm_cmpgt_epi64(__m128i a, __m128i b)
PEXTRB	int _mm_extract_epi8 (__m128i src, const int ndx)

Table C-1. Simple Intrinsics (Contd.)

Mnemonic	Intrinsic
PEXTRD	int_mm_extract_epi32 (__m128i src, const int ndx)
PEXTRQ	__int64_mm_extract_epi64 (__m128i src, const int ndx)
PEXTRW	int_mm_extract_epi16(__m128i a, int n)
PEXTRW	int_mm_extract_pi16(__m64 a, int n)
	int_mm_extract_epi16 (__m128i src, int ndx)
PHADDD	__m64_mm_hadd_pi32 (__m64 a, __m64 b)
	__m128i_mm_hadd_epi32 (__m128i a, __m128i b)
PHADDSW	__m64_mm_hadds_pi16 (__m64 a, __m64 b)
	__m128i_mm_hadds_epi16 (__m128i a, __m128i b)
PHADDW	__m64_mm_hadd_pi16 (__m64 a, __m64 b)
	__m128i_mm_hadd_epi16 (__m128i a, __m128i b)
PHMINPOSUW	__m128i_mm_minpos_epu16( __m128i packed_words)
PHSUBD	__m64_mm_hsub_pi32 (__m64 a, __m64 b)
	__m128i_mm_hsub_epi32 (__m128i a, __m128i b)
PHSUBSW	__m64_mm_hsubs_pi16 (__m64 a, __m64 b)
	__m128i_mm_hsubs_epi16 (__m128i a, __m128i b)
PHSUBW	__m64_mm_hsub_pi16 (__m64 a, __m64 b)
	__m128i_mm_hsub_epi16 (__m128i a, __m128i b)
PINSRB	__m128i_mm_insert_epi8(__m128i s1, int s2, const int ndx)
PINSRD	__m128i_mm_insert_epi32(__m128i s2, int s, const int ndx)
PINSRQ	__m128i_mm_insert_epi64(__m128i s2, __int64 s, const int ndx)
PINSRW	__m128i_mm_insert_epi16(__m128i a, int d, int n)
PINSRW	__m64_mm_insert_pi16(__m64 a, int d, int n)
PMADDUBSW	__m64_mm_maddubs_pi16 (__m64 a, __m64 b)
	__m128i_mm_maddubs_epi16 (__m128i a, __m128i b)
PMADDWD	__m128i_mm_madd_epi16(__m128i m1 __m128i m2)
PMADDWD	__m64_mm_madd_pi16(__m64 m1, __m64 m2)
PMAXSB	__m128i_mm_max_epi8( __m128i a, __m128i b)
PMAXSD	__m128i_mm_max_epi32( __m128i a, __m128i b)
PMAXSW	__m128i_mm_max_epi16(__m128i a, __m128i b)
PMAXSW	__m64_mm_max_pi16(__m64 a, __m64 b)
PMAXUB	__m128i_mm_max_epu8(__m128i a, __m128i b)
PMAXUB	__m64_mm_max_pu8(__m64 a, __m64 b)
PMAXUD	__m128i_mm_max_epu32( __m128i a, __m128i b)
PMAXUW	__m128i_mm_max_epu16( __m128i a, __m128i b)

**Table C-1. Simple Intrinsics (Contd.)**

<b>Mnemonic</b>	<b>Intrinsic</b>
PMINSB	<code>__m128i_mm_min_epi8(__m128i a, __m128i b)</code>
PMINSD	<code>__m128i_mm_min_epi32(__m128i a, __m128i b)</code>
PMINSW	<code>__m128i_mm_min_epi16(__m128i a, __m128i b)</code>
PMINSW	<code>__m64_mm_min_pi16(__m64 a, __m64 b)</code>
PMINUB	<code>__m128i_mm_min_epu8(__m128i a, __m128i b)</code>
PMINUB	<code>__m64_mm_min_pu8(__m64 a, __m64 b)</code>
PMINUD	<code>__m128i_mm_min_epu32(__m128i a, __m128i b)</code>
PMINUW	<code>__m128i_mm_min_epu16(__m128i a, __m128i b)</code>
PMOVMSKB	<code>int_mm_movemask_epi8(__m128i a)</code>
PMOVMSKB	<code>int_mm_movemask_pi8(__m64 a)</code>
PMOVSXBW	<code>__m128i_mm_cvtepi8_epi16(__m128i a)</code>
PMOVSXBD	<code>__m128i_mm_cvtepi8_epi32(__m128i a)</code>
PMOVSXBQ	<code>__m128i_mm_cvtepi8_epi64(__m128i a)</code>
PMOVSXWD	<code>__m128i_mm_cvtepi16_epi32(__m128i a)</code>
PMOVSXWQ	<code>__m128i_mm_cvtepi16_epi64(__m128i a)</code>
PMOVSDQ	<code>__m128i_mm_cvtepi32_epi64(__m128i a)</code>
PMOVZXBW	<code>__m128i_mm_cvtepu8_epi16(__m128i a)</code>
PMOVZXBD	<code>__m128i_mm_cvtepu8_epi32(__m128i a)</code>
PMOVZXBQ	<code>__m128i_mm_cvtepu8_epi64(__m128i a)</code>
PMOVZXWD	<code>__m128i_mm_cvtepu16_epi32(__m128i a)</code>
PMOVZXWQ	<code>__m128i_mm_cvtepu16_epi64(__m128i a)</code>
PMOVZXDQ	<code>__m128i_mm_cvtepu32_epi64(__m128i a)</code>
PMULDQ	<code>__m128i_mm_mul_epi32(__m128i a, __m128i b)</code>
PMULHRSW	<code>__m64_mm_mulhrs_pi16(__m64 a, __m64 b)</code>
	<code>__m128i_mm_mulhrs_epi16(__m128i a, __m128i b)</code>
PMULHUW	<code>__m128i_mm_mulhi_epu16(__m128i a, __m128i b)</code>
PMULHUW	<code>__m64_mm_mulhi_pu16(__m64 a, __m64 b)</code>
PMULHW	<code>__m128i_mm_mulhi_epi16(__m128i m1, __m128i m2)</code>
PMULHW	<code>__m64_mm_mulhi_pi16(__m64 m1, __m64 m2)</code>
PMULLUD	<code>__m128i_mm_mullo_epi32(__m128i a, __m128i b)</code>
PMULLW	<code>__m128i_mm_mullo_epi16(__m128i m1, __m128i m2)</code>
PMULLW	<code>__m64_mm_mullo_pi16(__m64 m1, __m64 m2)</code>
PMULUDQ	<code>__m64_mm_mul_su32(__m64 m1, __m64 m2)</code>
	<code>__m128i_mm_mul_epu32(__m128i m1, __m128i m2)</code>

Table C-1. Simple Intrinsics (Contd.)

Mnemonic	Intrinsic
POPCNT	int __mm_popcnt_u32(unsigned int a)
	int64_t __mm_popcnt_u64(unsigned __int64 a)
POR	__m64 __mm_or_si64(__m64 m1, __m64 m2)
POR	__m128i __mm_or_si128(__m128i m1, __m128i m2)
PREFETCHh	void __mm_prefetch(char *a, int sel)
PSADBw	__m128i __mm_sad_epu8(__m128i a, __m128i b)
PSADBw	__m64 __mm_sad_pu8(__m64 a, __m64 b)
PSHUFB	__m64 __mm_shuffle_pi8 (__m64 a, __m64 b)
	__m128i __mm_shuffle_epi8 (__m128i a, __m128i b)
PSHUFD	__m128i __mm_shuffle_epi32(__m128i a, int n)
PSHUFW	__m128i __mm_shufflehi_epi16(__m128i a, int n)
PSHUFLW	__m128i __mm_shufflelo_epi16(__m128i a, int n)
PSHUFW	__m64 __mm_shuffle_pi16(__m64 a, int n)
PSIGNB	__m64 __mm_sign_pi8 (__m64 a, __m64 b)
	__m128i __mm_sign_epi8 (__m128i a, __m128i b)
PSIGND	__m64 __mm_sign_pi32 (__m64 a, __m64 b)
	__m128i __mm_sign_epi32 (__m128i a, __m128i b)
PSIGNW	__m64 __mm_sign_pi16 (__m64 a, __m64 b)
	__m128i __mm_sign_epi16 (__m128i a, __m128i b)
PSLLW	__m128i __mm_sll_epi16(__m128i m, __m128i count)
PSLLW	__m128i __mm_slli_epi16(__m128i m, int count)
PSLLW	__m64 __mm_sll_pi16(__m64 m, __m64 count)
	__m64 __mm_slli_pi16(__m64 m, int count)
PSLLD	__m128i __mm_slli_epi32(__m128i m, int count)
	__m128i __mm_sll_epi32(__m128i m, __m128i count)
PSLLD	__m64 __mm_slli_pi32(__m64 m, int count)
	__m64 __mm_sll_pi32(__m64 m, __m64 count)
PSLLQ	__m64 __mm_sll_si64(__m64 m, __m64 count)
	__m64 __mm_slli_si64(__m64 m, int count)
PSLLQ	__m128i __mm_sll_epi64(__m128i m, __m128i count)
	__m128i __mm_slli_epi64(__m128i m, int count)
PSLLDQ	__m128i __mm_slli_si128(__m128i m, int imm)
PSRAW	__m128i __mm_sra_epi16(__m128i m, __m128i count)
	__m128i __mm_srai_epi16(__m128i m, int count)

Table C-1. Simple Intrinsics (Contd.)

Mnemonic	Intrinsic
PSRAW	__m64 __mm_sra_pi16(__m64 m, __m64 count)
	__m64 __mm_srai_pi16(__m64 m, int count)
PSRAD	__m128i __mm_sra_epi32 (__m128i m, __m128i count)
	__m128i __mm_srai_epi32 (__m128i m, int count)
PSRAD	__m64 __mm_sra_pi32 (__m64 m, __m64 count)
	__m64 __mm_srai_pi32 (__m64 m, int count)
PSRLW	__m128i __mm_srl_epi16 (__m128i m, __m128i count)
	__m128i __mm_srli_epi16 (__m128i m, int count)
	__m64 __mm_srl_pi16 (__m64 m, __m64 count)
	__m64 __mm_srli_pi16(__m64 m, int count)
PSRLD	__m128i __mm_srl_epi32 (__m128i m, __m128i count)
	__m128i __mm_srli_epi32 (__m128i m, int count)
PSRLD	__m64 __mm_srl_pi32 (__m64 m, __m64 count)
	__m64 __mm_srli_pi32 (__m64 m, int count)
PSRLQ	__m128i __mm_srl_epi64 (__m128i m, __m128i count)
	__m128i __mm_srli_epi64 (__m128i m, int count)
PSRLQ	__m64 __mm_srl_si64 (__m64 m, __m64 count)
	__m64 __mm_srli_si64 (__m64 m, int count)
PSRLDQ	__m128i __mm_srli_si128(__m128i m, int imm)
PSUBB	__m128i __mm_sub_epi8(__m128i m1, __m128i m2)
PSUBB	__m64 __mm_sub_pi8(__m64 m1, __m64 m2)
PSUBW	__m128i __mm_sub_epi16(__m128i m1, __m128i m2)
PSUBW	__m64 __mm_sub_pi16(__m64 m1, __m64 m2)
PSUBD	__m128i __mm_sub_epi32(__m128i m1, __m128i m2)
PSUBD	__m64 __mm_sub_pi32(__m64 m1, __m64 m2)
PSUBQ	__m128i __mm_sub_epi64(__m128i m1, __m128i m2)
PSUBQ	__m64 __mm_sub_si64(__m64 m1, __m64 m2)
PSUBSB	__m128i __mm_subs_epi8(__m128i m1, __m128i m2)
PSUBSB	__m64 __mm_subs_pi8(__m64 m1, __m64 m2)
PSUBSW	__m128i __mm_subs_epi16(__m128i m1, __m128i m2)
PSUBSW	__m64 __mm_subs_pi16(__m64 m1, __m64 m2)
PSUBUSB	__m128i __mm_subs_epu8(__m128i m1, __m128i m2)
PSUBUSB	__m64 __mm_subs_pu8(__m64 m1, __m64 m2)
PSUBUSW	__m128i __mm_subs_epu16(__m128i m1, __m128i m2)
PSUBUSW	__m64 __mm_subs_pu16(__m64 m1, __m64 m2)

Table C-1. Simple Intrinsics (Contd.)

Mnemonic	Intrinsic
PTEST	int_mm_testz_si128(__m128i s1, __m128i s2)
	int_mm_testc_si128(__m128i s1, __m128i s2)
	int_mm_testnzc_si128(__m128i s1, __m128i s2)
PUNPCKHBW	__m64_mm_unpackhi_pi8(__m64 m1, __m64 m2)
PUNPCKHBW	__m128i_mm_unpackhi_epi8(__m128i m1, __m128i m2)
PUNPCKHWD	__m64_mm_unpackhi_pi16(__m64 m1, __m64 m2)
PUNPCKHWD	__m128i_mm_unpackhi_epi16(__m128i m1, __m128i m2)
PUNPCKHDQ	__m64_mm_unpackhi_pi32(__m64 m1, __m64 m2)
PUNPCKHDQ	__m128i_mm_unpackhi_epi32(__m128i m1, __m128i m2)
PUNPCKHQDQ	__m128i_mm_unpackhi_epi64(__m128i m1, __m128i m2)
PUNPCKLBW	__m64_mm_unpacklo_pi8(__m64 m1, __m64 m2)
PUNPCKLBW	__m128i_mm_unpacklo_epi8(__m128i m1, __m128i m2)
PUNPCKLWD	__m64_mm_unpacklo_pi16(__m64 m1, __m64 m2)
PUNPCKLWD	__m128i_mm_unpacklo_epi16(__m128i m1, __m128i m2)
PUNPCKLDQ	__m64_mm_unpacklo_pi32(__m64 m1, __m64 m2)
PUNPCKLDQ	__m128i_mm_unpacklo_epi32(__m128i m1, __m128i m2)
PUNPCKLQDQ	__m128i_mm_unpacklo_epi64(__m128i m1, __m128i m2)
PXOR	__m64_mm_xor_si64(__m64 m1, __m64 m2)
PXOR	__m128i_mm_xor_si128(__m128i m1, __m128i m2)
RCPPS	__m128_mm_rcp_ps(__m128 a)
RCPSS	__m128_mm_rcp_ss(__m128 a)
ROUNDPD	__m128 mm_round_pd(__m128d s1, int iRoundMode)
	__m128 mm_floor_pd(__m128d s1)
	__m128 mm_ceil_pd(__m128d s1)
ROUNDPS	__m128 mm_round_ps(__m128 s1, int iRoundMode)
	__m128 mm_floor_ps(__m128 s1)
	__m128 mm_ceil_ps(__m128 s1)
ROUNDSD	__m128d mm_round_sd(__m128d dst, __m128d s1, int iRoundMode)
	__m128d mm_floor_sd(__m128d dst, __m128d s1)
	__m128d mm_ceil_sd(__m128d dst, __m128d s1)
ROUNDSS	__m128 mm_round_ss(__m128 dst, __m128 s1, int iRoundMode)
	__m128 mm_floor_ss(__m128 dst, __m128 s1)
	__m128 mm_ceil_ss(__m128 dst, __m128 s1)
RSQRTPS	__m128_mm_rsqrtps(__m128 a)
RSQRTSS	__m128_mm_rsqrtps(__m128 a)

**Table C-1. Simple Intrinsics (Contd.)**

<b>Mnemonic</b>	<b>Intrinsic</b>
SFENCE	void_mm_sfence(void)
SHUFPD	__m128d_mm_shuffle_pd(__m128d a, __m128d b, unsigned int imm8)
SHUFPS	__m128_mm_shuffle_ps(__m128 a, __m128 b, unsigned int imm8)
SQRTPD	__m128d_mm_sqrt_pd(__m128d a)
SQRTPS	__m128_mm_sqrt_ps(__m128 a)
SQRTSD	__m128d_mm_sqrt_sd(__m128d a)
SQRTSS	__m128_mm_sqrt_ss(__m128 a)
STMXCSR	__mm_getcsr(void)
SUBPD	__m128d_mm_sub_pd(__m128d a, __m128d b)
SUBPS	__m128_mm_sub_ps(__m128 a, __m128 b)
SUBSD	__m128d_mm_sub_sd(__m128d a, __m128d b)
SUBSS	__m128_mm_sub_ss(__m128 a, __m128 b)
UCOMISD	int_mm_ucomieq_sd(__m128d a, __m128d b)
	int_mm_ucomilt_sd(__m128d a, __m128d b)
	int_mm_ucomile_sd(__m128d a, __m128d b)
	int_mm_ucomigt_sd(__m128d a, __m128d b)
	int_mm_ucomige_sd(__m128d a, __m128d b)
	int_mm_ucomineq_sd(__m128d a, __m128d b)
UCOMISS	int_mm_ucomieq_ss(__m128 a, __m128 b)
	int_mm_ucomilt_ss(__m128 a, __m128 b)
	int_mm_ucomile_ss(__m128 a, __m128 b)
	int_mm_ucomigt_ss(__m128 a, __m128 b)
	int_mm_ucomige_ss(__m128 a, __m128 b)
	int_mm_ucomineq_ss(__m128 a, __m128 b)
UNPCKHPD	__m128d_mm_unpackhi_pd(__m128d a, __m128d b)
UNPCKHPS	__m128_mm_unpackhi_ps(__m128 a, __m128 b)
UNPCKLPD	__m128d_mm_unpacklo_pd(__m128d a, __m128d b)
UNPCKLPS	__m128_mm_unpacklo_ps(__m128 a, __m128 b)
XORPD	__m128d_mm_xor_pd(__m128d a, __m128d b)
XORPS	__m128_mm_xor_ps(__m128 a, __m128 b)

## C.2 COMPOSITE INTRINSICS

Table C-2. Composite Intrinsics

Mnemonic	Intrinsic
(composite)	__m128i _mm_set_epi64(__m64 q1, __m64 q0)
(composite)	__m128i _mm_set_epi32(int i3, int i2, int i1, int i0)
(composite)	__m128i _mm_set_epi16(short w7,short w6, short w5, short w4, short w3, short w2, short w1,short w0)
(composite)	__m128i _mm_set_epi8(char w15,char w14, char w13, char w12, char w11, char w10, char w9, char w8, char w7,char w6, char w5, char w4, char w3, char w2,char w1, char w0)
(composite)	__m128i _mm_set1_epi64(__m64 q)
(composite)	__m128i _mm_set1_epi32(int a)
(composite)	__m128i _mm_set1_epi16(short a)
(composite)	__m128i _mm_set1_epi8(char a)
(composite)	__m128i _mm_setr_epi64(__m64 q1, __m64 q0)
(composite)	__m128i _mm_setr_epi32(int i3, int i2, int i1, int i0)
(composite)	__m128i _mm_setr_epi16(short w7,short w6, short w5, short w4, short w3, short w2, short w, short w0)
(composite)	__m128i _mm_setr_epi8(char w15,char w14, char w13, char w12, char w11, char w10, char w9, char w8,char w7, char w6,char w5, char w4, char w3, char w2,char w1,char w0)
(composite)	__m128i _mm_setzero_si128()
(composite)	__m128 _mm_set_ps1(float w) __m128 _mm_set1_ps(float w)
(composite)	__m128cmm_set1_pd(double w)
(composite)	__m128d _mm_set_sd(double w)
(composite)	__m128d _mm_set_pd(double z, double y)
(composite)	__m128 _mm_set_ps(float z, float y, float x, float w)
(composite)	__m128d _mm_setr_pd(double z, double y)
(composite)	__m128 _mm_setr_ps(float z, float y, float x, float w)
(composite)	__m128d _mm_setzero_pd(void)
(composite)	__m128 _mm_setzero_ps(void)
MOVSD + shuffle	__m128d _mm_load_pd(double * p) __m128d _mm_load1_pd(double *p)
MOVSS + shuffle	__m128 _mm_load_ps1(float * p) __m128 _mm_load1_ps(float *p)
MOVAPD + shuffle	__m128d _mm_loadr_pd(double * p)
MOVAPS + shuffle	__m128 _mm_loadr_ps(float * p)
MOVSD + shuffle	void _mm_store1_pd(double *p, __m128d a)



**Table C-2. Composite Intrinsic (Contd.)**

<b>Mnemonic</b>	<b>Intrinsic</b>
MOVSS + shuffle	void __mm_store_ps1(float * p, __m128 a) void __mm_store1_ps(float *p, __m128 a)
MOVAPD + shuffle	__mm_storer_pd(double * p, __m128d a)
MOVAPS + shuffle	__mm_storer_ps(float * p, __m128 a)

## Numerics

0000, B-59

64-bit mode

- control and debug registers, 2-15
- default operand size, 2-15
- direct memory-offset MOVs, 2-13
- general purpose encodings, B-24
- immediates, 2-14
- introduction, 2-9
- machine instructions, B-1
- reg (reg) field, B-4
- REX prefixes, 2-9, B-2
- RIP-relative addressing, 2-14
- SIMD encodings, B-54
- special instruction encodings, B-94
- summary table notation, 3-10

## A

- AAA instruction, 3-23, 3-25
- AAD instruction, 3-25
- AAM instruction, 3-27
- AAS instruction, 3-29
- Access rights, segment descriptor, 3-565
- ADC instruction, 3-31, 3-594
- ADD instruction, 3-23, 3-35, 3-292, 3-594
- ADDPD instruction, 3-38
- ADDPS instruction, 3-40
- Addressing methods
  - RIP-relative, 2-14
- Addressing, segments, 1-6
- ADDSD instruction, 3-42
- ADDSS instruction, 3-44
- ADDSUBPD instruction, 3-46
- ADDSUBPS instruction, 3-49
- AESDEC/AESDECLAST- Perform One Round of an AES
  - Decryption Flow, 3-58
- AESIMC- Perform the AES InvMixColumn
  - Transformation, 3-54
- AESKEYGENASSIST - AES Round Key Generation
  - Assist, 3-62
- AND instruction, 3-64, 3-594
- ANDNPD instruction, 3-71
- ANDNPS instruction, 3-73
- ANDPD instruction, 3-67
- ANDPS instruction, 3-69
- Arctangent, x87 FPU operation, 3-395
- ARPL instruction, 3-75
- authenticated code execution mode, 6-4

## B

- Base (operand addressing), 2-4
- BCD integers
  - packed, 3-292, 3-294, 3-333, 3-335
  - unpacked, 3-23, 3-25, 3-27, 3-29
- Binary numbers, 1-6
- Bit order, 1-4

- bootstrap processor, 6-21, 6-28, 6-37, 6-39, 6-40
- BOUND instruction, 3-89, 4-621
- BOUND range exceeded exception (#BR), 3-89, 4-621
- Branch hints, 2-2
- Brand information, 3-226
  - processor brand index, 3-229
  - processor brand string, 3-226
- BSF instruction, 3-92
- BSR instruction, 3-95
- BSWAP instruction, 3-98
- BT instruction, 3-100
- BTC instruction, 3-103, 3-594
- BTR instruction, 3-106, 3-594
- BTS instruction, 3-109, 3-594
- Byte order, 1-4

## C

- Cache and TLB information, 3-219
- Cache Inclusiveness, 3-201
- Caches, invalidating (flushing), 3-529, 4-653
- CALL instruction, 3-112
- GETSEC, 6-3
- CBW instruction, 3-131
- CDQ instruction, 3-290
- CDQE instruction, 3-131
- CF (carry) flag, EFLAGS register, 3-35, 3-100, 3-103, 3-106, 3-109, 3-133, 3-142, 3-296, 3-495, 3-501, 4-146, 4-443, 4-512, 4-530, 4-534, 4-561, 4-575
- CLC instruction, 3-133
- CLD instruction, 3-134
- CLFLUSH instruction, 3-135
  - CPUID flag, 3-218
- CLI instruction, 3-137
- CLTS instruction, 3-140
- CMC instruction, 3-142
- CMOVcc flag, 3-218
- CMOVcc instructions, 3-143
  - CPUID flag, 3-218
- CMP instruction, 3-150
- CMPPD instruction, 3-153
- CMPPS instruction, 3-163
- CMPS instruction, 3-170, 4-468
- CMPSB instruction, 3-170
- CMPSD instruction, 3-170, 3-176
- CMPSQ instruction, 3-170
- CMPS instruction, 3-182
- CMPSW instruction, 3-170
- CMPXCHG instruction, 3-188, 3-594
- CMPXCHG16B instruction, 3-191
  - CPUID bit, 3-214
- CMPXCHG8B instruction, 3-191
  - CPUID flag, 3-217
- COMISD instruction, 3-194
- COMISS instruction, 3-196
- Compatibility mode

## INDEX

- introduction, 2-9
  - see 64-bit mode
  - summary table notation, 3-11
- Compatibility, software, 1-5
- compilers
  - documentation, 1-9
- Condition code flags, EFLAGS register, 3-143
- Condition code flags, x87 FPU status word
  - flags affected by instructions, 3-18
  - setting, 3-445, 3-447, 3-450
- Conditional jump, 3-544
- Conforming code segment, 3-566
- Constants (floating point), loading, 3-383
- Control registers, moving values to and from, 4-49
- Cosine, x87 FPU operation, 3-351, 3-420
- CPL, 3-137, 4-625
- CPUID instruction, 3-198, 3-218
  - 36-bit page size extension, 3-218
  - AP-485, 1-9
  - APIC on-chip, 3-217
  - basic CPUID information, 3-199
  - cache and TLB characteristics, 3-200
  - CLFLUSH flag, 3-218
  - CLFLUSH instruction cache line size, 3-211
  - CMPXCHG16B flag, 3-214
  - CMPXCHG8B flag, 3-217
  - CPL qualified debug store, 3-214
  - debug extensions, CR4.DE, 3-217
  - debug store supported, 3-218
  - deterministic cache parameters leaf, 3-200, 3-202, 3-204, 3-205, 3-206
  - extended function information, 3-206
  - feature information, 3-216
  - FPU on-chip, 3-217
  - FSAVE flag, 3-218
  - FXRSTOR flag, 3-218
  - HT technology flag, 3-219
  - IA-32e mode available, 3-207
  - input limits for EAX, 3-208
  - L1 Context ID, 3-214
  - local APIC physical ID, 3-211
  - machine check architecture, 3-218
  - machine check exception, 3-217
  - memory type range registers, 3-217
  - MONITOR feature information, 3-224
  - MONITOR/MWAIT flag, 3-214
  - MONITOR/MWAIT leaf, 3-201, 3-202, 3-203, 3-204
  - MWAIT feature information, 3-224
  - page attribute table, 3-218
  - page size extension, 3-217
  - performance monitoring features, 3-225
  - physical address bits, 3-208
  - physical address extension, 3-217
  - power management, 3-224, 3-225
  - processor brand index, 3-211, 3-226
  - processor brand string, 3-207, 3-226
  - processor serial number, 3-200, 3-218
  - processor type field, 3-210
  - RDMSR flag, 3-217
  - returned in EBX, 3-211
  - returned in ECX & EDX, 3-212
  - self snoop, 3-219
  - SpeedStep technology, 3-214
  - SS2 extensions flag, 3-219
  - SSE extensions flag, 3-219
  - SSE3 extensions flag, 3-214
  - SSSE3 extensions flag, 3-214
  - SYSENTER flag, 3-217
  - SYSEXIT flag, 3-217
  - thermal management, 3-224, 3-225
  - thermal monitor, 3-214, 3-218, 3-219
  - time stamp counter, 3-217
  - using CPUID, 3-198
  - vendor ID string, 3-208
  - version information, 3-199, 3-224
  - virtual 8086 Mode flag, 3-217
  - virtual address bits, 3-208
  - WRMSR flag, 3-217
- CQO instruction, 3-290
- CRO control register, 4-549
- CS register, 3-114, 3-514, 3-533, 3-553, 4-44, 4-341
- CVTDQ2PD instruction, 3-236
- CVTDQ2PS instruction, 3-243
- CVTPD2DQ instruction, 3-245
- CVTPD2PI instruction, 3-248
- CVTPD2PS instruction, 3-250
- CVTPI2PD instruction, 3-253
- CVTPI2PS instruction, 3-255
- CVTPS2DQ instruction, 3-257, 3-258
- CVTPS2PD instruction, 3-259
- CVTPS2PI instruction, 3-262
- CVTSD2SI instruction, 3-264
- CVTSD2SS instruction, 3-266
- CVTSI2SD instruction, 3-268
- CVTSI2SS instruction, 3-270
- CVTSS2SD instruction, 3-272
- CVTSS2SI instruction, 3-274
- CVTTPD2DQ instruction, 3-276
- CVTTPD2PI instruction, 3-276, 3-279
- CVTTPS2DQ instruction, 3-281
- CVTTPS2PI instruction, 3-284
- CVTTS2SI instruction, 3-286
- CVTSS2SI instruction, 3-288
- CWD instruction, 3-290
- CWDE instruction, 3-131
- C/C++ compiler intrinsics
  - compiler functional equivalents, C-1
  - composite, C-17
  - description of, 3-15
  - lists of, C-1
  - simple, C-2

**D**

D (default operation size) flag, segment descriptor, 4-346

DAA instruction, 3-292

DAS instruction, 3-294

Debug registers, moving value to and from, 4-53

DEC instruction, 3-296, 3-594

Denormalized finite number, 3-450

Detecting and Enabling SMX  
level 2, 6-2

DF (direction) flag, EFLAGS register, 3-134, 3-172, 3-505, 3-597, 4-114, 4-179, 4-516, 4-562

Displacement (operand addressing), 2-4

DIV instruction, 3-299

Divide error exception (#DE), 3-299

DIVPD instruction, 3-303

DIVPS instruction, 3-305

DIVSD instruction, 3-307

DIVSS instruction, 3-309

DS register, 3-171, 3-574, 3-597, 4-114, 4-178, 4-179

**E**

EDI register, 4-515, 4-562, 4-567

Effective address, 3-579

EFLAGS register  
condition codes, 3-147, 3-342, 3-348  
flags affected by instructions, 3-18  
popping, 4-351  
popping on return from interrupt, 3-533  
pushing, 4-434  
pushing on interrupts, 3-514  
saving, 4-501  
status flags, 3-150, 3-549, 4-522, 4-602

EIP register, 3-113, 3-514, 3-533, 3-553

EMMS instruction, 3-317

Encodings  
See machine instructions, opcodes

ENTER instruction, 3-319

GETSEC, 6-4, 6-12

Error numbers  
VM-instruction error field, 5-34

ES register, 3-574, 4-179, 4-515, 4-516, 4-567, 4-568

ESI register, 3-171, 3-597, 4-114, 4-178, 4-562

ESP register, 3-114, 4-341

Exceptions  
BOUND range exceeded (#BR), 3-89, 4-621  
notation, 1-7  
overflow exception (#OF), 3-513  
returning from, 3-533

GETSEC, 6-4, 6-6

Exponent, extracting from floating-point number, 3-468

Extract exponent and significand, x87 FPU operation, 3-468

**F**

F2XM1 instruction, 3-325, 3-468

FABS instruction, 3-327

FADD instruction, 3-329

FADDP instruction, 3-329

Far pointer, loading, 3-573

Far return, RET instruction, 4-471

FBLD instruction, 3-333

FBSTP instruction, 3-335

FCHS instruction, 3-338

FCLEX instruction, 3-340

FCMOVcc instructions, 3-342

FCOM instruction, 3-344

FCOMI instruction, 3-348

FCOMIP instruction, 3-348

FCOMP instruction, 3-344

FCOMPP instruction, 3-344

FCOS instruction, 3-351

FDECSTP instruction, 3-353

FDIV instruction, 3-355

FDIVP instruction, 3-355

FDIVR instruction, 3-359

FDIVRP instruction, 3-359

Feature information, processor, 3-198

FFREE instruction, 3-363

FIADD instruction, 3-329

FICOM instruction, 3-364

FICOMP instruction, 3-364

FIDIV instruction, 3-355

FIDIVR instruction, 3-359

FILD instruction, 3-367

FIMUL instruction, 3-390

FINCSTP instruction, 3-369

FINIT instruction, 3-371

FINIT/FNINIT instructions, 3-412

FIST instruction, 3-373

FISTP instruction, 3-373

FISTTP instruction, 3-377

FISUB instruction, 3-437

FISUBR instruction, 3-441

FLD instruction, 3-380

FLD1 instruction, 3-383

FLDCW instruction, 3-385

FLDENV instruction, 3-387

FLDL2E instruction, 3-383

FLDL2T instruction, 3-383

FLDLG2 instruction, 3-383

FLDLN2 instruction, 3-383

FLDPI instruction, 3-383

FLDZ instruction, 3-383

Floating point instructions  
machine encodings, B-95

Floating-point exceptions  
SSE and SSE2 SIMD, 3-21  
x87 FPU, 3-21

Flushing  
caches, 3-529, 4-653  
TLB entry, 3-531

## INDEX

FMUL instruction, 3-390  
FMULP instruction, 3-390  
FNCLEX instruction, 3-340  
FNINIT instruction, 3-371  
FNOP instruction, 3-394  
FNSAVE instruction, 3-412  
FNSTCW instruction, 3-428  
FNSTENV instruction, 3-387, 3-431  
FNSTSW instruction, 3-434  
FPATAN instruction, 3-395  
FPREM instruction, 3-398  
FPREM1 instruction, 3-401  
FPTAN instruction, 3-404  
FRNDINT instruction, 3-407  
FRSTOR instruction, 3-409  
FS register, 3-574  
FSAVE instruction, 3-412  
FSAVE/FNSAVE instructions, 3-409  
FSCALE instruction, 3-416  
FSIN instruction, 3-418  
FSINCOS instruction, 3-420  
FSQRT instruction, 3-423  
FST instruction, 3-425  
FSTCW instruction, 3-428  
FSTENV instruction, 3-431  
FSTP instruction, 3-425  
FSTSW instruction, 3-434  
FSUB instruction, 3-437  
FSUBP instruction, 3-437  
FSUBR instruction, 3-441  
FSUBRP instruction, 3-441  
FTST instruction, 3-445  
FUCOM instruction, 3-447  
FUCOMI instruction, 3-348  
FUCOMIP instruction, 3-348  
FUCOMP instruction, 3-447  
FUCOMPP instruction, 3-447  
FXAM instruction, 3-450  
FXCH instruction, 3-452  
FXRSTOR instruction, 3-454  
    CPUID flag, 3-218  
FXSAVE instruction, 3-458, 4-647, 4-649, 4-663,  
    4-675, 4-682, 4-686, 4-691  
    CPUID flag, 3-218  
FXTTRACT instruction, 3-416, 3-468  
FYL2X instruction, 3-470  
FYL2XP1 instruction, 3-472

## G

GDT (global descriptor table), 3-586, 3-589  
GDTR (global descriptor table register), 3-586, 4-526  
General-purpose instructions  
    64-bit encodings, B-24  
    non-64-bit encodings, B-9  
General-purpose registers  
    moving value to and from, 4-44  
    popping all, 4-346

    pushing all, 4-431  
GETSEC, 6-1, 6-3, 6-7  
GS register, 3-574

## H

HADDDPD instruction, 3-474, 3-476  
HADDDPS instruction, 3-477  
Hexadecimal numbers, 1-6  
HLT instruction, 3-481  
HSUBPD instruction, 3-483  
HSUBPS instruction, 3-486  
Hyper-Threading Technology  
    CPUID flag, 3-219

## I

IA-32e mode  
    CPUID flag, 3-207  
    introduction, 2-9, 2-16  
    see 64-bit mode  
    see compatibility mode  
IA32\_SYSENTER\_CS MSR, 4-593, 4-598  
IA32\_SYSENTER\_EIP MSR, 4-593  
IA32\_SYSENTER\_ESP MSR, 4-593  
IDIV instruction, 3-490  
IDT (interrupt descriptor table), 3-514, 3-586  
IDTR (interrupt descriptor table register), 3-586,  
    4-544  
IF (interrupt enable) flag, EFLAGS register, 3-137,  
    4-563  
Immediate operands, 2-4  
IMUL instruction, 3-494  
IN instruction, 3-499  
INC instruction, 3-501, 3-594  
Index (operand addressing), 2-4  
Initialization x87 FPU, 3-371  
initiating logical processor, 6-4, 6-6, 6-12, 6-13, 6-27,  
    6-28  
INS instruction, 3-504, 4-468  
INSB instruction, 3-504  
INSD instruction, 3-504  
instruction encodings, B-89, B-98  
Instruction format  
    base field, 2-4  
    description of reference information, 3-1  
    displacement, 2-4  
    immediate, 2-4  
    index field, 2-4  
    Mod field, 2-4  
    ModR/M byte, 2-4  
    opcode, 2-3  
    operands, 1-6  
    prefixes, 2-1  
    reg/opcode field, 2-4  
    r/m field, 2-4  
    scale field, 2-4  
    SIB byte, 2-4

See also: machine instructions, opcodes  
 Instruction reference, nomenclature, 3-1  
 Instruction set, reference, 3-1  
 INSW instruction, 3-504  
 INT 3 instruction, 3-513  
 Integer, storing, x87 FPU data type, 3-373  
 Intel 64 architecture  
   definition of, 1-3  
   instruction format, 2-1  
   relation to IA-32, 1-3  
 Intel developer link, 1-9  
 Intel NetBurst microarchitecture, 1-2  
 Intel software network link, 1-9  
 Intel VTune Performance Analyzer  
   related information, 1-9  
 Intel Xeon processor, 1-1  
 Intel® Trusted Execution Technology, 6-4  
 Inter-privilege level  
   call, CALL instruction, 3-113  
   return, RET instruction, 4-471  
 Interrupts  
   interrupt vector 4, 3-513  
   returning from, 3-533  
   software, 3-513  
 INTn instruction, 3-513  
 INTO instruction, 3-513  
 Intrinsic  
   compiler functional equivalents, C-1  
   composite, C-17  
   description of, 3-15  
   list of, C-1  
   simple, C-2  
 INVD instruction, 3-529  
 INVLPG instruction, 3-531  
 IOPL (I/O privilege level) field, EFLAGS register, 3-137,  
   4-434, 4-563  
 IRET instruction, 3-533  
 IRETD instruction, 3-533

## J

Jcc instructions, 3-544  
 JMP instruction, 3-552  
 Jump operation, 3-552

## L

L1 Context ID, 3-214  
 LAHF instruction, 3-563  
 LAR instruction, 3-565  
 Last branch  
   interrupt & exception recording  
     description of, 4-485  
 LDDQU instruction, 3-569  
 LDMXCSR instruction, 3-571  
 LDS instruction, 3-573  
 LDT (local descriptor table), 3-589  
 LDTR (local descriptor table register), 3-589, 4-547

LEA instruction, 3-579  
 LEAVE instruction, 3-582  
 LES instruction, 3-573  
 LFENCE instruction, 3-584  
 LFS instruction, 3-573  
 LGDT instruction, 3-586  
 LGS instruction, 3-573  
 LIDT instruction, 3-586  
 LLDT instruction, 3-589  
 LMSW instruction, 3-592  
 Load effective address operation, 3-579  
 LOCK prefix, 3-32, 3-36, 3-65, 3-104, 3-107, 3-110,  
   3-189, 3-296, 3-501, 3-594, 4-161,  
   4-166, 4-169, 4-513, 4-575, 4-657,  
   4-661, 4-669  
 Locking operation, 3-594  
 LODS instruction, 3-596, 4-468  
 LODSB instruction, 3-596  
 LODSD instruction, 3-596  
 LODSQ instruction, 3-596  
 LODSW instruction, 3-596  
 Log epsilon, x87 FPU operation, 3-470  
 Log (base 2), x87 FPU operation, 3-472  
 LOOP instructions, 3-600  
 LOOPcc instructions, 3-600  
 LSL instruction, 3-603  
 LSS instruction, 3-573  
 LTR instruction, 3-607

## M

Machine check architecture  
   CPUID flag, 3-218  
   description, 3-218  
 Machine instructions  
   64-bit mode, B-1  
   condition test (tttn) field, B-7  
   direction bit (d) field, B-8  
   floating-point instruction encodings, B-95  
   general description, B-1  
   general-purpose encodings, B-9–B-53  
   legacy prefixes, B-2  
   MMX encodings, B-55–B-58  
   opcode fields, B-2  
   operand size (w) bit, B-5  
   P6 family encodings, B-59  
   Pentium processor family encodings, B-53  
   reg (reg) field, B-3, B-4  
   REX prefixes, B-2  
   segment register (sreg) field, B-6  
   sign-extend (s) bit, B-5  
   SIMD 64-bit encodings, B-54  
   special 64-bit encodings, B-94  
   special fields, B-2  
   special-purpose register (eee) field, B-6  
   SSE encodings, B-60–B-69  
   SSE2 encodings, B-69–B-86  
   SSE3 encodings, B-87–B-89

## INDEX

- SSSE3 encodings, B-89–B-93
- VMX encodings, B-115–B-116, B-117–??
- See also: opcodes
- Machine status word, CR0 register, 3-592, 4-549
- MASKMOVDQU instruction, 4-53
- MASKMOVQ instruction, 4-11
- MAXPD instruction, 4-17
- MAXPS instruction, 4-20
- MAXSD instruction, 4-23
- MAXSS instruction, 4-25
- measured environment, 6-1
- Measured Launched Environment, 6-1, 6-33
- MFENCE instruction, 4-27
- MINPD instruction, 4-29
- MINPS instruction, 4-32
- MINSD instruction, 4-35
- MINSS instruction, 4-37
- MLE, 6-1
- MMX instructions
  - CPUID flag for technology, 3-218
  - encodings, B-55
- Mod field, instruction format, 2-4
- Model & family information, 3-224
- ModR/M byte, 2-4
  - 16-bit addressing forms, 2-6
  - 32-bit addressing forms of, 2-7
  - description of, 2-4
- MONITOR instruction, 4-39
  - CPUID flag, 3-214
  - feature data, 3-224
- MOV instruction, 4-42
- MOV instruction (control registers), 4-49
- MOV instruction (debug registers), 4-53, 4-62
- MOVAPD instruction, 4-56
- MOVAPS instruction, 4-59
- MOVD instruction, 4-62
- MOVDDUP instruction, 4-68
- MOVQ2Q instruction, 4-77
- MOVQQA instruction, 4-71
- MOVQDU instruction, 4-74
- MOVHLPS instruction, 4-79
- MOVHPD instruction, 4-81
- MOVHPS instruction, 4-83
- MOVLHP instruction, 4-85
- MOVLHPS instruction, 4-85
- MOVLPD instruction, 4-87
- MOVLPS instruction, 4-89
- MOVMSKPD instruction, 4-91
- MOVMSKPS instruction, 4-93
- MOVNTDQ instruction, 4-112
- MOVNTI instruction, 4-112
- MOVNTPD instruction, 4-103
- MOVNTPS instruction, 4-105
- MOVNTQ instruction, 4-107
- MOVQ instruction, 4-62, 4-109
- MOVQ2DQ instruction, 4-111
- MOVSD instruction, 4-113, 4-468
- MOVSB instruction, 4-113

- MOVSD instruction, 4-113, 4-118
- MOVSHDUP instruction, 4-121
- MOVSLDUP instruction, 4-124
- MOVSSQ instruction, 4-113
- MOVSS instruction, 4-127
- MOVSW instruction, 4-113
- MOVSD instruction, 4-130
- MOVSSXD instruction, 4-130
- MOVUPD instruction, 4-133
- MOVUPS instruction, 4-136
- MOVZX instruction, 4-139
- MSRs (model specific registers)
  - reading, 4-452
- MUL instruction, 3-27, 4-146
- MULPD instruction, 4-149
- MULPS instruction, 4-151
- MULSD instruction, 4-153
- MULSS instruction, 4-155
- Multi-byte no operation, 4-161, 4-164, B-16
- MVMM, 6-1, 6-6, 6-7, 6-49
- MWAIT instruction, 4-157
  - CPUID flag, 3-214
  - feature data, 3-224

## N

- NaN. testing for, 3-445
- Near
  - return, RET instruction, 4-471
- NEG instruction, 3-594, 4-161
- NetBurst microarchitecture (see Intel NetBurst microarchitecture)
- No operation, 4-161, 4-164, B-16
- Nomenclature, used in instruction reference pages, 3-1
- NOP instruction, 4-164
- NOT instruction, 3-594, 4-166
- Notation
  - bit and byte order, 1-4
  - exceptions, 1-7
  - hexadecimal and binary numbers, 1-6
  - instruction operands, 1-6
  - reserved bits, 1-5
  - segmented addressing, 1-6
- Notational conventions, 1-4
- NT (nested task) flag, EFLAGS register, 3-533

## O

- OF (carry) flag, EFLAGS register, 3-495
- OF (overflow) flag, EFLAGS register, 3-35, 3-513, 4-146, 4-512, 4-530, 4-534, 4-575
- Opcode format, 2-3
- Opcodes
  - addressing method codes for, A-2
  - extensions, A-20
  - extensions tables, A-21
  - group numbers, A-20

- integers
    - one-byte opcodes, A-10
    - two-byte opcodes, A-12
  - key to abbreviations, A-2
  - look-up examples, A-5, A-20, A-23
  - ModR/M byte, A-20
  - one-byte opcodes, A-5, A-10
  - opcode maps, A-1
  - operand type codes for, A-3
  - register codes for, A-4
  - superscripts in tables, A-8
  - two-byte opcodes, A-5, A-6, A-7, A-12
  - VMX instructions, B-115, B-117
  - x87 ESC instruction opcodes, A-23
- Operands, 1-6
- OR instruction, 3-594, 4-168
- ORPD instruction, 4-171
- ORPS instruction, 4-173
- OUT instruction, 4-175
- OUTS instruction, 4-178, 4-468
- OUTSB instruction, 4-178
- OUTSD instruction, 4-178
- OUTSW instruction, 4-178
- Overflow exception (#OF), 3-513
- P**
- P6 family processors
  - description of, 1-1
  - machine encodings, B-59
- PABSB instruction, 4-184
- PABSD instruction, 4-184
- PABSW instruction, 4-184
- PACKSSDW instruction, 4-188
- PACKSSWB instruction, 4-188
- PACKUSWB instruction, 4-195
- PADDB instruction, 4-198
- PADD instruction, 4-198
- PADDQ instruction, 4-202
- PADDSB instruction, 4-204
- PADDSW instruction, 4-204
- PADDUSB instruction, 4-207
- PADDUSW instruction, 4-207
- PADDW instruction, 4-198
- PALIGNR instruction, 4-210
- PAND instruction, 4-212
- PANDN instruction, 4-214
- GETSEC, 6-5
- PAUSE instruction, 4-216
- PAVGB instruction, 4-218
- PAVGW instruction, 4-218
- PCE flag, CR4 register, 4-456
- PCMPEQB instruction, 4-231
- PCMPEQD instruction, 4-231
- PCMPEQW instruction, 4-231
- PCMPGTB instruction, 4-241
- PCMPGTD instruction, 4-241
- PCMPGTW instruction, 4-241
- PE (protection enable) flag, CR0 register, 3-592
- Pending break enable, 3-219
- Pentium 4 processor, 1-1
- Pentium II processor, 1-2
- Pentium III processor, 1-2
- Pentium Pro processor, 1-2
- Pentium processor, 1-1
- Pentium processor family processors
  - machine encodings, B-53
- Performance-monitoring counters
  - CPUID inquiry for, 3-225
- PEXTRW instruction, 4-254
- PHADD instruction, 4-257
- PHADDSW instruction, 4-260
- PHADDW instruction, 4-257
- PHSUBD instruction, 4-264
- PHSUBSW instruction, 4-267
- PHSUBW instruction, 4-264
- Pi, 3-383
- PINSRW instruction, 4-272, 4-374
- PMADDUBSW instruction, 4-275
- PMADDUSW instruction, 4-275
- PMADDWD instruction, 4-277
- PMAXSW instruction, 4-285
- PMAXUB instruction, 4-288
- PMINSW instruction, 4-300
- PMINUB instruction, 4-303
- PMOVMASKB instruction, 4-306
- PMULHRSW instruction, 4-322
- PMULHUW instruction, 4-325
- PMULHW instruction, 4-328
- PMULLW instruction, 4-333
- PMULUDQ instruction, 4-336
- POP instruction, 4-339
- POPA instruction, 4-346
- POPAD instruction, 4-346
- POPF instruction, 4-351
- POPFD instruction, 4-351
- POPfq instruction, 4-351
- POR instruction, 4-355
- PREFETCHh instruction, 4-357
- Prefixes
  - Address-size override prefix, 2-2
  - Branch hints, 2-2
  - branch hints, 2-2
  - instruction, description of, 2-1
  - legacy prefix encodings, B-2
  - LOCK, 2-2, 3-594
  - Operand-size override prefix, 2-2
  - REP or REPE/REPZ, 2-2
  - REPNE/REPZ, 2-2
  - REP/REPE/REPZ/REPNE/REPZ, 4-466
  - REX prefix encodings, B-2
  - Segment override prefixes, 2-2
- PSADBW instruction, 4-360
- Pseudo-functions
  - VMfail, 5-2
  - VMfailinvalid, 5-2



## INDEX

- VMfailValid, 5-2
- VMsucceed, 5-2
- PSHUFB instruction, 4-363
- PSHUFD instruction, 4-366
- PSHUFHW instruction, 4-368
- PSHUFW instruction, 4-370
- PSHUFw instruction, 4-372
- PSIGNB instruction, 4-374
- PSIGND instruction, 4-374
- PSIGNW instruction, 4-374
- PSLLD instruction, 4-381
- PSLLDQ instruction, 4-379
- PSLLQ instruction, 4-381
- PSLLW instruction, 4-381
- PSRAD instruction, 4-388
- PSRAW instruction, 4-388
- PSRLD instruction, 4-395
- PSRLDQ instruction, 4-393
- PSRLQ instruction, 4-395
- PSRLW instruction, 4-395
- PSUBB instruction, 4-401
- PSUBD instruction, 4-401
- PSUBQ instruction, 4-405
- PSUBSB instruction, 4-407
- PSUBSW instruction, 4-407
- PSUBUSB instruction, 4-410
- PSUBUSW instruction, 4-410
- PSUBW instruction, 4-401
- PUNPCKHBW instruction, 4-415
- PUNPCKHDQ instruction, 4-415
- PUNPCKHQDQ instruction, 4-415
- PUNPCKHWD instruction, 4-415
- PUNPCKLBW instruction, 4-421
- PUNPCKLDQ instruction, 4-421
- PUNPCKLQDQ instruction, 4-421
- PUNPCKLWD instruction, 4-421
- PUSH instruction, 4-426
- PUSHA instruction, 4-431
- PUSHAD instruction, 4-431
- PUSHF instruction, 4-434
- PUSHFD instruction, 4-434
- PXOR instruction, 4-437

**R**

- RC (rounding control) field, x87 FPU control word, 3-374, 3-383, 3-425
- RCL instruction, 4-439
- RCPPS instruction, 4-447
- RCPSS instruction, 4-450
- RCR instruction, 4-439
- RDMSR instruction, 4-452, 4-456, 4-462
  - CPUID flag, 3-217
- RDPMC instruction, 4-454
- RDTSC instruction, 4-460, 4-462, 4-464
- Reg/opcode field, instruction format, 2-4
- Related literature, 1-8
- Remainder, x87 FPU operation, 3-401

- REP/REPE/REPZ/REPNE/REPNZ prefixes, 3-172, 3-505, 4-179, 4-466
- Reserved
  - use of reserved bits, 1-5
- Responding logical processor, 6-6
- responding logical processor, 6-4, 6-5, 6-6
- RET instruction, 4-471
- REX prefixes
  - addressing modes, 2-11
  - and INC/DEC, 2-10
  - encodings, 2-10, B-2
  - field names, 2-11
  - ModR/M byte, 2-10
  - overview, 2-9
  - REX.B, 2-10
  - REX.R, 2-10
  - REX.W, 2-10
  - special encodings, 2-13
- RIP-relative addressing, 2-14
- ROL instruction, 4-439
- ROR instruction, 4-439
- Rounding
  - modes, floating-point operations, 4-485
- Rounding control (RC) field
  - MXCSR register, 4-485
  - x87 FPU control word, 4-485
- Rounding, round to integer, x87 FPU operation, 3-407
- RPL field, 3-75
- RSM instruction, 4-494
- RSQRTPS instruction, 4-496
- RSQRTSS instruction, 4-499
- R/m field, instruction format, 2-4

## S

- Safer Mode Extensions, 6-1
- SAHF instruction, 4-501
- SAL instruction, 4-503
- SAR instruction, 4-503
- SBB instruction, 3-594, 4-511
- Scale (operand addressing), 2-4
- Scale, x87 FPU operation, 3-416
- Scan string instructions, 4-515
- SCAS instruction, 4-468, 4-515
- SCASB instruction, 4-515
- SCASD instruction, 4-515
- SCASW instruction, 4-515
- Segment
  - descriptor, segment limit, 3-603
  - limit, 3-603
  - registers, moving values to and from, 4-44
  - selector, RPL field, 3-75
- Segmented addressing, 1-6
- Self Snoop, 3-219
- GETSEC, 6-2, 6-4, 6-6
- SENTER sleep state, 6-12
- SETcc instructions, 4-520
- GETSEC, 6-5

- SF (sign) flag, EFLAGS register, 3-35
  - SFENCE instruction, 4-525
  - SGDT instruction, 4-526
  - SHAF instruction, 4-501
  - Shift instructions, 4-503
  - SHL instruction, 4-503
  - SHLD instruction, 4-529
  - SHR instruction, 4-503
  - SHRD instruction, 4-533
  - SHUFPD instruction, 4-537
  - SHUFPS instruction, 4-540
  - SIB byte, 2-4
    - 32-bit addressing forms of, 2-8
    - description of, 2-4
  - SIDT instruction, 4-526, 4-544
  - Significand, extracting from floating-point number, 3-468
  - SIMD floating-point exceptions, unmasking, effects of, 3-571
  - Sine, x87 FPU operation, 3-418, 3-420
  - SINIT, 6-5
  - SLDT instruction, 4-547
  - GETSEC, 6-5
  - SMSW instruction, 4-549
  - SpeedStep technology, 3-214
  - SQRTPD instruction, 4-552
  - SQRTPS instruction, 4-554
  - SQRTSD instruction, 4-557
  - SQRTSS instruction, 4-559
  - Square root, Fx87 PU operation, 3-423
  - SS register, 3-574, 4-44, 4-341
  - SSE extensions
    - cacheability instruction encodings, B-68
    - CPUID flag, 3-219
    - floating-point encodings, B-60
    - instruction encodings, B-60
    - integer instruction encodings, B-67
    - memory ordering encodings, B-68
  - SSE2 extensions
    - cacheability instruction encodings, B-86
    - CPUID flag, 3-219
    - floating-point encodings, B-70
    - integer instruction encodings, B-79
  - SSE3
    - CPUID flag, 3-214
  - SSE3 extensions
    - CPUID flag, 3-214
    - event mgmt instruction encodings, B-88
    - floating-point instruction encodings, B-87
    - integer instruction encodings, B-88, B-89
  - SSSE3 extensions, B-89, B-98
    - CPUID flag, 3-214
  - Stack, pushing values on, 4-426
  - Status flags, EFLAGS register, 3-147, 3-150, 3-342, 3-348, 3-549, 4-522, 4-602
  - STC instruction, 4-561
  - STD instruction, 4-562
  - Stepping information, 3-224
  - STI instruction, 4-563
  - STMXCSR instruction, 4-566
  - STOS instruction, 4-468, 4-567
  - STOSB instruction, 4-567
  - STOSD instruction, 4-567
  - STOSQ instruction, 4-567
  - STOSW instruction, 4-567
  - STR instruction, 4-572
  - String instructions, 3-170, 3-504, 3-596, 4-113, 4-178, 4-515, 4-567
  - SUB instruction, 3-29, 3-294, 3-594, 4-574
  - SUBPD instruction, 4-577
  - SUBSS instruction, 4-584
  - SWAPGS instruction, 4-586
  - SYSCALL instruction, 4-588
  - SYSENTER instruction, 4-590
    - CPUID flag, 3-217
  - SYSEXIT instruction, 4-595
    - CPUID flag, 3-217
  - SYSRET instruction, 4-599
- ## T
- Tangent, x87 FPU operation, 3-404
  - Task register
    - loading, 3-607
    - storing, 4-572
  - Task switch
    - CALL instruction, 3-113
    - return from nested task, IRET instruction, 3-533
  - TEST instruction, 4-601, 4-643
  - Thermal Monitor
    - CPUID flag, 3-219
  - Thermal Monitor 2, 3-214
    - CPUID flag, 3-214
  - Time Stamp Counter, 3-217
  - Time-stamp counter, reading, 4-462, 4-464
  - TLB entry, invalidating (flushing), 3-531
  - Trusted Platform Module, 6-6
  - TS (task switched) flag, CR0 register, 3-140
  - TSD flag, CR4 register, 4-462, 4-464
  - TSS, relationship to task register, 4-572
- ## U
- UCOMISD instruction, 4-604
  - UCOMISS instruction, 4-606
  - UD2 instruction, 4-608
  - Undefined, format opcodes, 3-445
  - Unordered values, 3-344, 3-445, 3-447
  - UNPCKHPD instruction, 4-609
  - UNPCKHPS instruction, 4-612
  - UNPCKLPD instruction, 4-615
  - UNPCKLPS instruction, 4-618
- ## V
- VERR instruction, 4-625
  - Version information, processor, 3-198

## INDEX

- VERW instruction, 4-625
- VEX, 3-4
- VEX.B, 3-4
- VEX.L, 3-4
- VEX.mmmm, 3-4
- VEX.pp, 3-5
- VEX.R, 3-6
- VEX.vvvv, 3-4
- VEX.W, 3-4
- VEX.X, 3-4
- Virtual Machine Monitor, 6-1
- VM (virtual 8086 mode) flag, EFLAGS register, 3-533
- VMCALL instruction, 5-1
- VMCLEAR instruction, 5-1
- VMCS
  - error numbers, 5-34
  - VM-instruction error field, 5-34
- VMLAUNCH instruction, 5-1
- VMM, 6-1
- VMPTRLD instruction, 5-1
- VMPTRST instruction, 5-1
- VMREAD instruction, 5-1
- VMRESUME instruction, 5-1, 5-2
- VMWRITE instruction, 5-1
- VMXOFF instruction, 5-1
- VMXON instruction, 5-2
  - restoring, 3-409
  - saving, 3-412, 3-431, 3-434
  - TOP field, 3-369
  - x87 FPU flags affected by instructions, 3-18
- x87 FPU tag word, 3-387, 3-409, 3-412, 3-431
- XADD instruction, 3-594, 4-657
- XCHG instruction, 3-594, 4-660
- XFEATURE\_ENABLED\_MASK, 4-663, 4-676, 4-682, 4-686, 4-687, 4-691, 4-692
- XGETBV, 4-663, 4-675, B-59
- XLAB instruction, 4-665
- XLAT instruction, 4-665
- XOR instruction, 3-594, 4-668
- XORPD instruction, 4-671
- XORPS instruction, 4-673
- XRSTOR, 4-683, 4-687, B-59
- XSAVE, 4-663, 4-664, 4-675, 4-676, 4-677, 4-678, 4-679, 4-680, 4-682, 4-683, 4-684, 4-685, 4-686, 4-687, 4-689, 4-690, 4-692, B-59
- XSETBV, 4-686, 4-691, B-59

## Z

- ZF (zero) flag, EFLAGS register, 3-188, 3-565, 3-600, 3-603, 4-468, 4-625

## W

- WAIT/FWAIT instructions, 4-651
- GETSEC, 6-6
- WBINVD instruction, 4-653
- wBINVD/INVD bit, 3-201
- Write-back and invalidate caches, 4-653
- WRMSR instruction, 4-655
  - CPUID flag, 3-217

## X

- x87 FPU
  - checking for pending x87 FPU exceptions, 4-651
  - constants, 3-383
  - initialization, 3-371
  - instruction opcodes, A-23
- x87 FPU control word
  - loading, 3-385, 3-387
  - RC field, 3-374, 3-383, 3-425
  - restoring, 3-409
  - saving, 3-412, 3-431
  - storing, 3-428
- x87 FPU data pointer, 3-387, 3-409, 3-412, 3-431
- x87 FPU instruction pointer, 3-387, 3-409, 3-412, 3-431
- x87 FPU last opcode, 3-387, 3-409, 3-412, 3-431
- x87 FPU status word
  - condition code flags, 3-344, 3-364, 3-445, 3-447, 3-450
  - loading, 3-387