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

Volume 2D: Instruction Set Reference, W-Z

NOTE: The *Intel® 64 and IA-32 Architectures Software Developer's Manual* consists of ten volumes: *Basic Architecture*, Order Number 253665; *Instruction Set Reference, A-L*, Order Number 253666; *Instruction Set Reference, M-U*, Order Number 253667; *Instruction Set Reference, V*, Order Number 326018; *Instruction Set Reference*, W-Z, Order Number 334569; *System Programming Guide, Part 1*, Order Number 253668; *System Programming Guide, Part 2*, Order Number 253669; *System Programming Guide, Part 3*, Order Number 326019; *System Programming Guide, Part 4*, Order Number 332831; *Model-Specific Registers,* Order Number 335592. Refer to all ten volumes when evaluating your design needs.

Order Number: 334569-079US
March 2023

CHAPTER 6
INSTRUCTION SET REFERENCE, W-Z

6.1 INSTRUCTIONS (W-Z)

Chapter 6 continues an alphabetical discussion of Intel® 64 and IA-32 instructions (W-Z). See also: Chapter 3, "Instruction Set Reference, A-L," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A; Chapter 4, "Instruction Set Reference, M-U," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B; and Chapter 5, "Instruction Set Reference, V," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2D.

WAIT/FWAIT—Wait

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
9B	WAIT	ZO	Valid	Valid	Check pending unmasked floating-point exceptions.
9B	FWAIT	ZO	Valid	Valid	Check pending unmasked floating-point exceptions.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	N/A	N/A	N/A	N/A

Description

Causes the processor to check for and handle pending, unmasked, floating-point exceptions before proceeding. (FWAIT is an alternate mnemonic for WAIT.)

This instruction is useful for synchronizing exceptions in critical sections of code. Coding a WAIT instruction after a floating-point instruction ensures that any unmasked floating-point exceptions the instruction may raise are handled before the processor can modify the instruction’s results. See the section titled “Floating-Point Exception Synchronization” in Chapter 8 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, for more information on using the WAIT/FWAIT instruction.

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

Operation

```
CheckForPendingUnmaskedFloatingPointExceptions;
```

FPU Flags Affected

The C0, C1, C2, and C3 flags are undefined.

Floating-Point Exceptions

None.

Protected Mode Exceptions

#NM	If CR0.MP[bit 1] = 1 and CR0.TS[bit 3] = 1.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

WBINVD—Write Back and Invalidate Cache

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 09	WBINVD	ZO	Valid	Valid	Write back and flush Internal caches; initiate writing-back and flushing of external caches.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	N/A	N/A	N/A	N/A

Description

Writes back all modified cache lines in the processor's internal cache to main memory and invalidates (flushes) the internal caches. The instruction then issues a special-function bus cycle that directs external caches to also write back modified data and another bus cycle to indicate that the external caches should be invalidated.

After executing this instruction, the processor does not wait for the external caches to complete their write-back and flushing operations before proceeding with instruction execution. It is the responsibility of hardware to respond to the cache write-back and flush signals. The amount of time or cycles for WBINVD to complete will vary due to size and other factors of different cache hierarchies. As a consequence, the use of the WBINVD instruction can have an impact on logical processor interrupt/event response time. Additional information of WBINVD behavior in a cache hierarchy with hierarchical sharing topology can be found in Chapter 2 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A.

The WBINVD instruction is a privileged instruction. When the processor is running in protected mode, the CPL of a program or procedure must be 0 to execute this instruction. This instruction is also a serializing instruction (see "Serializing Instructions" in Chapter 9 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A).

In situations where cache coherency with main memory is not a concern, software can use the INVD instruction.

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

IA-32 Architecture Compatibility

The WBINVD instruction is implementation dependent, and its function may be implemented differently on future Intel 64 and IA-32 processors. The instruction is not supported on IA-32 processors earlier than the Intel486 processor.

Operation

```
WriteBack(InternalCaches);
Flush(InternalCaches);
SignalWriteBack(ExternalCaches);
SignalFlush(ExternalCaches);
Continue; (* Continue execution *)
```

Intel C/C++ Compiler Intrinsic Equivalent

WBINVD void _wbinvd(void);

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If the current privilege level is not 0.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	WBINVD cannot be executed at the virtual-8086 mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

WBNOINVD—Write Back and Do Not Invalidate Cache

Opcode / Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 09 WBNOINVD	ZO	V/V	WBNOINVD	Write back and do not flush internal caches; initiate writing-back without flushing of external caches.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
ZO	N/A	N/A	N/A	N/A	N/A

Description

The WBNOINVD instruction writes back all modified cache lines in the processor's internal cache to main memory but does not invalidate (flush) the internal caches.

After executing this instruction, the processor does not wait for the external caches to complete their write-back operation before proceeding with instruction execution. It is the responsibility of hardware to respond to the cache write-back signal. The amount of time or cycles for WBNOINVD to complete will vary due to size and other factors of different cache hierarchies. As a consequence, the use of the WBNOINVD instruction can have an impact on logical processor interrupt/event response time.

The WBNOINVD instruction is a privileged instruction. When the processor is running in protected mode, the CPL of a program or procedure must be 0 to execute this instruction. This instruction is also a serializing instruction (see "Serializing Instructions" in Chapter 9 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A).

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

Operation

```
WriteBack(InternalCaches);
Continue; (* Continue execution *)
```

Intel C/C++ Compiler Intrinsic Equivalent

WBNOINVD void _wbnoinvd(void);

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If the current privilege level is not 0.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	WBNOINVD cannot be executed at the virtual-8086 mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

WRFSBASE/WRGSBASE—Write FS/GS Segment Base

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F3 0F AE /2 WRFSBASE r32	M	V/I	FSGSBASE	Load the FS base address with the 32-bit value in the source register.
F3 REX.W 0F AE /2 WRFSBASE r64	M	V/I	FSGSBASE	Load the FS base address with the 64-bit value in the source register.
F3 0F AE /3 WRGSBASE r32	M	V/I	FSGSBASE	Load the GS base address with the 32-bit value in the source register.
F3 REX.W 0F AE /3 WRGSBASE r64	M	V/I	FSGSBASE	Load the GS base address with the 64-bit value in the source register.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r)	N/A	N/A	N/A

Description

Loads the FS or GS segment base address with the general-purpose register indicated by the modR/M:r/m field.

The source operand may be either a 32-bit or a 64-bit general-purpose register. The REX.W prefix indicates the operand size is 64 bits. If no REX.W prefix is used, the operand size is 32 bits; the upper 32 bits of the source register are ignored and upper 32 bits of the base address (for FS or GS) are cleared.

This instruction is supported only in 64-bit mode.

Operation

```
FS/GS segment base address := SRC;
```

Flags Affected

None.

C/C++ Compiler Intrinsic Equivalent

```
WRFSBASE  void _writefsbase_u32( unsigned int );
WRFSBASE  _writefsbase_u64( unsigned __int64 );
WRGSBASE  void _writegsbase_u32( unsigned int );
WRGSBASE  _writegsbase_u64( unsigned __int64 );
```

Protected Mode Exceptions

#UD The WRFSBASE and WRGSBASE instructions are not recognized in protected mode.

Real-Address Mode Exceptions

#UD The WRFSBASE and WRGSBASE instructions are not recognized in real-address mode.

Virtual-8086 Mode Exceptions

#UD The WRFSBASE and WRGSBASE instructions are not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

#UD The WRFSBASE and WRGSBASE instructions are not recognized in compatibility mode.

64-Bit Mode Exceptions

#UD	If the LOCK prefix is used.
	If CR4.FSGSBASE[bit 16] = 0.
	If CPUID.07H.0H:EBX.FSGSBASE[bit 0] = 0
#GP(0)	If the source register contains a non-canonical address.

WRMSR—Write to Model Specific Register

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 30	WRMSR	ZO	Valid	Valid	Write the value in EDX:EAX to MSR specified by ECX.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	N/A	N/A	N/A	N/A

Description

Writes the contents of registers EDX:EAX into the 64-bit model specific register (MSR) specified in the ECX register. (On processors that support the Intel 64 architecture, the high-order 32 bits of RCX are ignored.) The contents of the EDX register are copied to high-order 32 bits of the selected MSR and the contents of the EAX register are copied to low-order 32 bits of the MSR. (On processors that support the Intel 64 architecture, the high-order 32 bits of each of RAX and RDX are ignored.) Undefined or reserved bits in an MSR should be set to values previously read.

This instruction must be executed at privilege level 0 or in real-address mode; otherwise, a general protection exception #GP(0) is generated. Specifying a reserved or unimplemented MSR address in ECX will also cause a general protection exception. The processor will also generate a general protection exception if software attempts to write to bits in a reserved MSR.

When the WRMSR instruction is used to write to an MTRR, the TLBs are invalidated. This includes global entries (see "Translation Lookaside Buffers (TLBs)" in Chapter 3 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A).

MSRs control functions for testability, execution tracing, performance-monitoring and machine check errors. Chapter 2, "Model-Specific Registers (MSRs)," of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4, lists all MSRs that can be written with this instruction and their addresses. Note that each processor family has its own set of MSRs.

The WRMSR instruction is a serializing instruction (see "Serializing Instructions" in Chapter 9 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A). Note that WRMSR to the IA32_TSC_DEADLINE MSR (MSR index 6E0H) and the X2APIC MSRs (MSR indices 802H to 83FH) are not serializing.

The CPUID instruction should be used to determine whether MSRs are supported (CPUID.01H:EDX[5] = 1) before using this instruction.

IA-32 Architecture Compatibility

The MSRs and the ability to read them with the WRMSR instruction were introduced into the IA-32 architecture with the Pentium processor. Execution of this instruction by an IA-32 processor earlier than the Pentium processor results in an invalid opcode exception #UD.

Operation

```
MSR[ECX] := EDX:EAX;
```

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If the current privilege level is not 0.
	If the value in ECX specifies a reserved or unimplemented MSR address.
	If the value in EDX:EAX sets bits that are reserved in the MSR specified by ECX.
	If the source register contains a non-canonical address and ECX specifies one of the following MSRs: IA32_DS_AREA, IA32_FS_BASE, IA32_GS_BASE, IA32_KERNEL_GS_BASE, IA32_LSTAR, IA32_SYSENTER_EIP, IA32_SYSENTER_ESP.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If the value in ECX specifies a reserved or unimplemented MSR address.
	If the value in EDX:EAX sets bits that are reserved in the MSR specified by ECX.
	If the source register contains a non-canonical address and ECX specifies one of the following MSRs: IA32_DS_AREA, IA32_FS_BASE, IA32_GS_BASE, IA32_KERNEL_GS_BASE, IA32_LSTAR, IA32_SYSENTER_EIP, IA32_SYSENTER_ESP.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	The WRMSR instruction is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

WRPKRU—Write Data to User Page Key Register

Opcode/ Instruction	Op/ En	64/32bit Mode Support	CPUID Feature Flag	Description
NP 0F 01 EF WRPKRU	ZO	V/V	OSPKE	Writes EAX into PKRU.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	N/A	N/A	N/A	N/A

Description

Writes the value of EAX into PKRU. ECX and EDX must be 0 when WRPKRU is executed; otherwise, a general-protection exception (#GP) occurs.

WRPKRU can be executed only if CR4.PKE = 1; otherwise, an invalid-opcode exception (#UD) occurs. Software can discover the value of CR4.PKE by examining CPUID.(EAX=07H,ECX=0H):ECX.OSPKE [bit 4].

On processors that support the Intel 64 Architecture, the high-order 32-bits of RCX, RDX, and RAX are ignored.

WRPKRU will never execute speculatively. Memory accesses affected by PKRU register will not execute (even speculatively) until all prior executions of WRPKRU have completed execution and updated the PKRU register.

Operation

```
IF (ECX = 0 AND EDX = 0)
    THEN PKRU := EAX;
    ELSE #GP(0);
FI;
```

Flags Affected

None.

C/C++ Compiler Intrinsic Equivalent

WRPKRU void _wrpkru(uint32_t);

Protected Mode Exceptions

#GP(0)	If ECX ≠ 0.
	If EDX ≠ 0.
#UD	If the LOCK prefix is used.
	If CR4.PKE = 0.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

WRSSD/WRSSQ—Write to Shadow Stack

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F 38 F6 !(11):rrr:bbb WRSSD m32, r32	MR	V/V	CET_SS	Write 4 bytes to shadow stack.
REX.W 0F 38 F6 !(11):rrr:bbb WRSSQ m64, r64	MR	V/N.E.	CET_SS	Write 8 bytes to shadow stack.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MR	ModRM:r/m (w)	ModRM:reg (r)	N/A	N/A

Description

Writes bytes in register source to the shadow stack.

Operation

```
IF CPL = 3
    IF (CR4.CET & IA32_U_CET.SH_STK_EN) = 0
        THEN #UD; FI;
    IF (IA32_U_CET.WR_SHSTK_EN) = 0
        THEN #UD; FI;
ELSE
    IF (CR4.CET & IA32_S_CET.SH_STK_EN) = 0
        THEN #UD; FI;
    IF (IA32_S_CET.WR_SHSTK_EN) = 0
        THEN #UD; FI;
FI;
DEST_LA = Linear_Address(mem operand)
IF (operand size is 64 bit)
    THEN
        (* Destination not 8B aligned *)
        IF DEST_LA[2:0]
            THEN GP(0); FI;
        Shadow_stack_store 8 bytes of SRC to DEST_LA;
    ELSE
        (* Destination not 4B aligned *)
        IF DEST_LA[1:0]
            THEN GP(0); FI;
        Shadow_stack_store 4 bytes of SRC[31:0] to DEST_LA;
FI;
```

Flags Affected

None.

C/C++ Compiler Intrinsic Equivalent

```
WRSSD void _wrssd(__int32, void *);
WRSSQ void _wrssq(__int64, void *);
```

Protected Mode Exceptions

#UD	If the LOCK prefix is used.
	If CR4.CET = 0.
	If CPL = 3 and IA32_U_CET.SH_STK_EN = 0.
	If CPL < 3 and IA32_S_CET.SH_STK_EN = 0.
	If CPL = 3 and IA32_U_CET.WR_SHSTK_EN = 0.
	If CPL < 3 and IA32_S_CET.WR_SHSTK_EN = 0.
#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If destination is located in a non-writeable segment.
	If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
	If linear address of destination is not 4 byte aligned.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs if destination is not a user shadow stack when CPL3 and not a supervisor shadow stack when CPL < 3.
	Other terminal and non-terminal faults.

Real-Address Mode Exceptions

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

Virtual-8086 Mode Exceptions

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

Compatibility Mode Exceptions

#UD	If the LOCK prefix is used.
	If CR4.CET = 0.
	If CPL = 3 and IA32_U_CET.SH_STK_EN = 0.
	If CPL < 3 and IA32_S_CET.SH_STK_EN = 0.
	If CPL = 3 and IA32_U_CET.WR_SHSTK_EN = 0.
	If CPL < 3 and IA32_S_CET.WR_SHSTK_EN = 0.
#PF(fault-code)	If a page fault occurs if destination is not a user shadow stack when CPL3 and not a supervisor shadow stack when CPL < 3.
	Other terminal and non-terminal faults.

64-Bit Mode Exceptions

#UD	If the LOCK prefix is used.
	If CR4.CET = 0.
	If CPL = 3 and IA32_U_CET.SH_STK_EN = 0.
	If CPL < 3 and IA32_S_CET.SH_STK_EN = 0.
	If CPL = 3 and IA32_U_CET.WR_SHSTK_EN = 0.
	If CPL < 3 and IA32_S_CET.WR_SHSTK_EN = 0.
#GP(0)	If a memory address is in a non-canonical form.
	If linear address of destination is not 4 byte aligned.
#PF(fault-code)	If a page fault occurs if destination is not a user shadow stack when CPL3 and not a supervisor shadow stack when CPL < 3.
	Other terminal and non-terminal faults.

WRUSSD/WRUSSQ—Write to User Shadow Stack

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 F5 !(11):rrr:bbb WRUSSD m32, r32	MR	V/V	CET_SS	Write 4 bytes to shadow stack.
66 REX.W 0F 38 F5 !(11):rrr:bbb WRUSSQ m64, r64	MR	V/N.E.	CET_SS	Write 8 bytes to shadow stack.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MR	ModRM:r/m (w)	ModRM:reg (r)	N/A	N/A

Description

Writes bytes in register source to a user shadow stack page. The WRUSS instruction can be executed only if CPL = 0, however the processor treats its shadow-stack accesses as user accesses.

Operation

```
IF CR4.CET = 0
    THEN #UD; FI;
IF CPL > 0
    THEN #GP(0); FI;
DEST_LA = Linear_Address(mem operand)
IF (operand size is 64 bit)
    THEN
        (* Destination not 8B aligned *)
        IF DEST_LA[2:0]
            THEN GP(0); FI;
        Shadow_stack_store 8 bytes of SRC to DEST_LA as user-mode access;
    ELSE
        (* Destination not 4B aligned *)
        IF DEST_LA[1:0]
            THEN GP(0); FI;
        Shadow_stack_store 4 bytes of SRC[31:0] to DEST_LA as user-mode access;
FI;
```

Flags Affected

None.

C/C++ Compiler Intrinsic Equivalent

WRUSSD void _wrussd(__int32, void *);
WRUSSQ void _wrussq(__int64, void *);

Protected Mode Exceptions

#UD	If the LOCK prefix is used.
	If CR4.CET = 0.
#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If destination is located in a non-writeable segment.
	If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
	If linear address of destination is not 4 byte aligned.
	If CPL is not 0.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If destination is not a user shadow stack.
	Other terminal and non-terminal faults.

Real-Address Mode Exceptions

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

Virtual-8086 Mode Exceptions

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

Compatibility Mode Exceptions

#UD	If the LOCK prefix is used.
	If CR4.CET = 0.
#GP(0)	If a memory address is in a non-canonical form.
	If linear address of destination is not 4 byte aligned.
	If CPL is not 0.
#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#PF(fault-code)	If destination is not a user shadow stack.
	Other terminal and non-terminal faults.

64-Bit Mode Exceptions

#UD	If the LOCK prefix is used.
	If CR4.CET = 0.
#GP(0)	If a memory address is in a non-canonical form.
	If linear address of destination is not 4 byte aligned.
	If CPL is not 0.
#PF(fault-code)	If destination is not a user shadow stack.
	Other terminal and non-terminal faults.

XABORT—Transactional Abort

Opcode/Instruction	Op/ En	64/32bit Mode Support	CPUID Feature Flag	Description
C6 F8 ib XABORT imm8	A	V/V	RTM	Causes an RTM abort if in RTM execution.

Instruction Operand Encoding

Op/En	Operand 1	Operand2	Operand3	Operand4
A	imm8	N/A	N/A	N/A

Description

XABORT forces an RTM abort. Following an RTM abort, the logical processor resumes execution at the fallback address computed through the outermost XBEGIN instruction. The EAX register is updated to reflect an XABORT instruction caused the abort, and the imm8 argument will be provided in bits 31:24 of EAX.

Operation

XABORT

```
IF RTM_ACTIVE = 0
    THEN
        Treat as NOP;
    ELSE
        GOTO RTM_ABORT_PROCESSING;
FI;

(* For any RTM abort condition encountered during RTM execution *)
RTM_ABORT_PROCESSING:
    Restore architectural register state;
    Discard memory updates performed in transaction;
    Update EAX with status and XABORT argument;
    RTM_NEST_COUNT:= 0;
    RTM_ACTIVE:= 0;
    SUSLDTRK_ACTIVE := 0;
    IF 64-bit Mode
        THEN
            RIP:= fallbackRIP;
        ELSE
            EIP := fallbackEIP;
    FI;
END
```

Flags Affected

None.

Intel C/C++ Compiler Intrinsic Equivalent

XABORT void _xabort(unsigned int);

SIMD Floating-Point Exceptions

None.

Other Exceptions

#UD	CPUID.(EAX=7, ECX=0):EBX.RTM[bit 11] = 0.
	If LOCK prefix is used.

XACQUIRE/XRELEASE—Hardware Lock Elision Prefix Hints

Opcode/Instruction	64/32bit Mode Support	CPUID Feature Flag	Description
F2 XACQUIRE	V/V	HLE[1]	A hint used with an "XACQUIRE-enabled" instruction to start lock elision on the instruction memory operand address.
F3 XRELEASE	V/V	HLE	A hint used with an "XRELEASE-enabled" instruction to end lock elision on the instruction memory operand address.

NOTES:

1. Software is not required to check the HLE feature flag to use XACQUIRE or XRELEASE, as they are treated as regular prefix if HLE feature flag reports 0.

Description

The XACQUIRE prefix is a hint to start lock elision on the memory address specified by the instruction and the XRELEASE prefix is a hint to end lock elision on the memory address specified by the instruction.

The XACQUIRE prefix hint can only be used with the following instructions (these instructions are also referred to as XACQUIRE-enabled when used with the XACQUIRE prefix):

- Instructions with an explicit LOCK prefix (F0H) prepended to forms of the instruction where the destination operand is a memory operand: ADD, ADC, AND, BTC, BTR, BTS, CMPXCHG, CMPXCHG8B, DEC, INC, NEG, NOT, OR, SBB, SUB, XOR, XADD, and XCHG.
- The XCHG instruction either with or without the presence of the LOCK prefix.

The XRELEASE prefix hint can only be used with the following instructions (also referred to as XRELEASE-enabled when used with the XRELEASE prefix):

- Instructions with an explicit LOCK prefix (F0H) prepended to forms of the instruction where the destination operand is a memory operand: ADD, ADC, AND, BTC, BTR, BTS, CMPXCHG, CMPXCHG8B, DEC, INC, NEG, NOT, OR, SBB, SUB, XOR, XADD, and XCHG.
- The XCHG instruction either with or without the presence of the LOCK prefix.
- The "MOV mem, reg" (Opcode 88H/89H) and "MOV mem, imm" (Opcode C6H/C7H) instructions. In these cases, the XRELEASE is recognized without the presence of the LOCK prefix.

The lock variables must satisfy the guidelines described in Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, Section 16.3.3, for elision to be successful, otherwise an HLE abort may be signaled.

If an encoded byte sequence that meets XACQUIRE/XRELEASE requirements includes both prefixes, then the HLE semantic is determined by the prefix byte that is placed closest to the instruction opcode. For example, an F3F2C6 will not be treated as a XRELEASE-enabled instruction since the F2H (XACQUIRE) is closest to the instruction opcode C6. Similarly, an F2F3F0 prefixed instruction will be treated as a XRELEASE-enabled instruction since F3H (XRELEASE) is closest to the instruction opcode.

Intel 64 and IA-32 Compatibility

The effect of the XACQUIRE/XRELEASE prefix hint is the same in non-64-bit modes and in 64-bit mode.

For instructions that do not support the XACQUIRE hint, the presence of the F2H prefix behaves the same way as prior hardware, according to

- REPNE/REPNZ semantics for string instructions,
- Serve as SIMD prefix for legacy SIMD instructions operating on XMM register
- Cause #UD if prepending the VEX prefix.
- Undefined for non-string instructions or other situations.

For instructions that do not support the XRELEASE hint, the presence of the F3H prefix behaves the same way as in prior hardware, according to

- REP/REPE/REPZ semantics for string instructions,
- Serve as SIMD prefix for legacy SIMD instructions operating on XMM register
- Cause #UD if prepending the VEX prefix.
- Undefined for non-string instructions or other situations.

Operation

XACQUIRE

```
IF XACQUIRE-enabled instruction
    THEN
        IF (HLE_NEST_COUNT < MAX_HLE_NEST_COUNT) THEN
            HLE_NEST_COUNT++
            IF (HLE_NEST_COUNT = 1) THEN
                HLE_ACTIVE := 1
                IF 64-bit mode
                    THEN
                        restartRIP := instruction pointer of the XACQUIRE-enabled instruction
                    ELSE
                        restartEIP := instruction pointer of the XACQUIRE-enabled instruction
                FI;
                Enter HLE Execution (* record register state, start tracking memory state *)
            FI; (* HLE_NEST_COUNT = 1*)
            IF ElisionBufferAvailable
                THEN
                    Allocate elision buffer
                    Record address and data for forwarding and commit checking
                    Perform elision
                ELSE
                    Perform lock acquire operation transactionally but without elision
            FI;
        ELSE (* HLE_NEST_COUNT = MAX_HLE_NEST_COUNT*)
                GOTO HLE_ABORT_PROCESSING
        FI;
    ELSE
        Treat instruction as non-XACQUIRE F2H prefixed legacy instruction
FI;
```

XRELEASE

```
IF XRELEASE-enabled instruction
    THEN
        IF (HLE_NEST_COUNT > 0)
            THEN
                HLE_NEST_COUNT--
                IF lock address matches in elision buffer THEN
                    IF lock satisfies address and value requirements THEN
                        Deallocate elision buffer
                    ELSE
                        GOTO HLE_ABORT_PROCESSING
                    FI;
                FI;
                IF (HLE_NEST_COUNT = 0)
                    THEN
                        IF NoAllocatedElisionBuffer
                            THEN
                                Try to commit transactional execution
                                IF fail to commit transactional execution
                                    THEN
                                        GOTO HLE_ABORT_PROCESSING;
                                    ELSE (* commit success *)
                                        HLE_ACTIVE := 0
                                FI;
                            ELSE
                                GOTO HLE_ABORT_PROCESSING
                        FI;
                FI;
        FI; (* HLE_NEST_COUNT > 0 *)
    ELSE
        Treat instruction as non-XRELEASE F3H prefixed legacy instruction
FI;

(* For any HLE abort condition encountered during HLE execution *)
HLE_ABORT_PROCESSING:
    HLE_ACTIVE := 0
    HLE_NEST_COUNT := 0
    Restore architectural register state
    Discard memory updates performed in transaction
    Free any allocated lock elision buffers
    IF 64-bit mode
        THEN
            RIP := restartRIP
        ELSE
            EIP := restartEIP
    FI;
    Execute and retire instruction at RIP (or EIP) and ignore any HLE hint
END
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

#GP(0)	If the use of prefix causes instruction length to exceed 15 bytes.

XADD—Exchange and Add

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F C0 /r	XADD r/m8, r8	MR	Valid	Valid	Exchange r8 and r/m8; load sum into r/m8.
REX + 0F C0 /r	XADD r/m8*, r8*	MR	Valid	N.E.	Exchange r8 and r/m8; load sum into r/m8.
0F C1 /r	XADD r/m16, r16	MR	Valid	Valid	Exchange r16 and r/m16; load sum into r/m16.
0F C1 /r	XADD r/m32, r32	MR	Valid	Valid	Exchange r32 and r/m32; load sum into r/m32.
REX.W + 0F C1 /r	XADD r/m64, r64	MR	Valid	N.E.	Exchange r64 and r/m64; load sum into r/m64.

NOTES:

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

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MR	ModRM:r/m (r, w)	ModRM:reg (r, w)	N/A	N/A

Description

Exchanges the first operand (destination operand) with the second operand (source operand), then loads the sum of the two values into the destination operand. The destination operand can be a register or a memory location; the source operand is a register.

In 64-bit mode, the instruction's default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.

IA-32 Architecture Compatibility

IA-32 processors earlier than the Intel486 processor do not recognize this instruction. If this instruction is used, you should provide an equivalent code sequence that runs on earlier processors.

Operation

```
TEMP := SRC + DEST;
SRC := DEST;
DEST := TEMP;
```

Flags Affected

The CF, PF, AF, SF, ZF, and OF flags are set according to the result of the addition, which is stored in the destination operand.

Protected Mode Exceptions

#GP(0)	If the destination is located in a non-writable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

XBEGIN—Transactional Begin

Opcode/Instruction	Op/ En	64/32bit Mode Support	CPUID Feature Flag	Description
C7 F8 XBEGIN rel16	A	V/V	RTM	Specifies the start of an RTM region. Provides a 16-bit relative offset to compute the address of the fallback instruction address at which execution resumes following an RTM abort.
C7 F8 XBEGIN rel32	A	V/V	RTM	Specifies the start of an RTM region. Provides a 32-bit relative offset to compute the address of the fallback instruction address at which execution resumes following an RTM abort.

Instruction Operand Encoding

Op/En	Operand 1	Operand2	Operand3	Operand4
A	Offset	N/A	N/A	N/A

Description

The XBEGIN instruction specifies the start of an RTM code region. If the logical processor was not already in transactional execution, then the XBEGIN instruction causes the logical processor to transition into transactional execution. The XBEGIN instruction that transitions the logical processor into transactional execution is referred to as the outermost XBEGIN instruction. The instruction also specifies a relative offset to compute the address of the fallback code path following a transactional abort. (Use of the 16-bit operand size does not cause this address to be truncated to 16 bits, unlike a near jump to a relative offset.)

On an RTM abort, the logical processor discards all architectural register and memory updates performed during the RTM execution and restores architectural state to that corresponding to the outermost XBEGIN instruction. The fallback address following an abort is computed from the outermost XBEGIN instruction.

Execution of XBEGIN while in a suspend read address tracking region causes a transactional abort.

Operation

XBEGIN

```
IF RTM_NEST_COUNT < MAX_RTM_NEST_COUNT AND SUSLDTRK_ACTIVE = 0
    THEN
        RTM_NEST_COUNT++
        IF RTM_NEST_COUNT = 1 THEN
            IF 64-bit Mode
                THEN
                    IF OperandSize = 16
                        THEN fallbackRIP := RIP + SignExtend64(rel16);
                        ELSE fallbackRIP := RIP + SignExtend64(rel32);
                    FI;
                    IF fallbackRIP is not canonical
                        THEN #GP(0);
                    FI;
                ELSE
                    IF OperandSize = 16
                        THEN fallbackEIP := EIP + SignExtend32(rel16);
                        ELSE fallbackEIP := EIP + rel32;
                    FI;
                    IF fallbackEIP outside code segment limit
                        THEN #GP(0);
                    FI;
            FI;
```

```
                RTM_ACTIVE := 1
                Enter RTM Execution (* record register state, start tracking memory state*)
            FI; (* RTM_NEST_COUNT = 1 *)
    ELSE (* RTM_NEST_COUNT = MAX_RTM_NEST_COUNT OR SUSLDTRK_ACTIVE = 1 *)
            GOTO RTM_ABORT_PROCESSING
FI;

(* For any RTM abort condition encountered during RTM execution *)
RTM_ABORT_PROCESSING:
    Restore architectural register state
    Discard memory updates performed in transaction
    Update EAX with status
    RTM_NEST_COUNT := 0
    RTM_ACTIVE := 0
    SUSLDTRK_ACTIVE := 0
    IF 64-bit mode
        THEN
            RIP := fallbackRIP
        ELSE
            EIP := fallbackEIP
    FI;
END
```

Flags Affected

None.

Intel C/C++ Compiler Intrinsic Equivalent

XBEGIN unsigned int _xbegin(void);

SIMD Floating-Point Exceptions

None.

Protected Mode Exceptions

#UD	CPUID.(EAX=7, ECX=0):EBX.RTM[bit 11]=0.
	If LOCK prefix is used.
#GP(0)	If the fallback address is outside the CS segment.

Real-Address Mode Exceptions

#GP(0)	If the fallback address is outside the address space 0000H and FFFFH.
#UD	CPUID.(EAX=7, ECX=0):EBX.RTM[bit 11]=0.
	If LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If the fallback address is outside the address space 0000H and FFFFH.
#UD	CPUID.(EAX=7, ECX=0):EBX.RTM[bit 11]=0.
	If LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-bit Mode Exceptions

#UD	CPUID.(EAX=7, ECX=0):EBX.RTM[bit 11] = 0.
	If LOCK prefix is used.
#GP(0)	If the fallback address is non-canonical.

XCHG—Exchange Register/Memory With Register

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
90+rw	XCHG AX, r16	O	Valid	Valid	Exchange r16 with AX.
90+rw	XCHG r16, AX	O	Valid	Valid	Exchange AX with r16.
90+rd	XCHG EAX, r32	O	Valid	Valid	Exchange r32 with EAX.
REX.W + 90+rd	XCHG RAX, r64	O	Valid	N.E.	Exchange r64 with RAX.
90+rd	XCHG r32, EAX	O	Valid	Valid	Exchange EAX with r32.
REX.W + 90+rd	XCHG r64, RAX	O	Valid	N.E.	Exchange RAX with r64.
86 /r	XCHG r/m8, r8	MR	Valid	Valid	Exchange r8 (byte register) with byte from r/m8.
REX + 86 /r	XCHG r/m8*, r8*	MR	Valid	N.E.	Exchange r8 (byte register) with byte from r/m8.
86 /r	XCHG r8, r/m8	RM	Valid	Valid	Exchange byte from r/m8 with r8 (byte register).
REX + 86 /r	XCHG r8*, r/m8*	RM	Valid	N.E.	Exchange byte from r/m8 with r8 (byte register).
87 /r	XCHG r/m16, r16	MR	Valid	Valid	Exchange r16 with word from r/m16.
87 /r	XCHG r16, r/m16	RM	Valid	Valid	Exchange word from r/m16 with r16.
87 /r	XCHG r/m32, r32	MR	Valid	Valid	Exchange r32 with doubleword from r/m32.
REX.W + 87 /r	XCHG r/m64, r64	MR	Valid	N.E.	Exchange r64 with quadword from r/m64.
87 /r	XCHG r32, r/m32	RM	Valid	Valid	Exchange doubleword from r/m32 with r32.
REX.W + 87 /r	XCHG r64, r/m64	RM	Valid	N.E.	Exchange quadword from r/m64 with r64.

NOTES:

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

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
O	AX/EAX/RAX (r, w)	opcode + rd (r, w)	N/A	N/A
O	opcode + rd (r, w)	AX/EAX/RAX (r, w)	N/A	N/A
MR	ModRM:r/m (r, w)	ModRM:reg (r)	N/A	N/A
RM	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Exchanges the contents of the destination (first) and source (second) operands. The operands can be two general-purpose registers or a register and a memory location. If a memory operand is referenced, the processor's locking protocol is automatically implemented for the duration of the exchange operation, regardless of the presence or absence of the LOCK prefix or of the value of the IOPL. (See the LOCK prefix description in this chapter for more information on the locking protocol.)

This instruction is useful for implementing semaphores or similar data structures for process synchronization. (See "Bus Locking" in Chapter 9 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A, for more information on bus locking.)

The XCHG instruction can also be used instead of the BSWAP instruction for 16-bit operands.

In 64-bit mode, the instruction's default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

NOTE

XCHG (E)AX, (E)AX (encoded instruction byte is 90H) is an alias for NOP regardless of data size prefixes, including REX.W.

Operation

```
TEMP := DEST;
DEST := SRC;
SRC := TEMP;
```

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If either operand is in a non-writable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

XEND—Transactional End

Opcode/Instruction	Op/ En	64/32bit Mode Support	CPUID Feature Flag	Description
NP 0F 01 D5 XEND	A	V/V	RTM	Specifies the end of an RTM code region.

Instruction Operand Encoding

Op/En	Operand 1	Operand2	Operand3	Operand4
A	N/A	N/A	N/A	N/A

Description

The instruction marks the end of an RTM code region. If this corresponds to the outermost scope (that is, including this XEND instruction, the number of XBEGIN instructions is the same as number of XEND instructions), the logical processor will attempt to commit the logical processor state atomically. If the commit fails, the logical processor will rollback all architectural register and memory updates performed during the RTM execution. The logical processor will resume execution at the fallback address computed from the outermost XBEGIN instruction. The EAX register is updated to reflect RTM abort information.

Execution of XEND outside a transactional region causes a general-protection exception (#GP). Execution of XEND while in a suspend read address tracking region causes a transactional abort.

Operation

XEND

```
IF (RTM_ACTIVE = 0) THEN
    SIGNAL #GP
ELSE
    IF SUSLDTRK_ACTIVE = 1
        THEN GOTO RTM_ABORT_PROCESSING;
    FI;
    RTM_NEST_COUNT--
    IF (RTM_NEST_COUNT = 0) THEN
        Try to commit transaction
        IF fail to commit transactional execution
            THEN
                GOTO RTM_ABORT_PROCESSING;
            ELSE (* commit success *)
                RTM_ACTIVE := 0
        FI;
    FI;
FI;

(* For any RTM abort condition encountered during RTM execution *)
RTM_ABORT_PROCESSING:
    Restore architectural register state
    Discard memory updates performed in transaction
    Update EAX with status
    RTM_NEST_COUNT := 0
    RTM_ACTIVE := 0
    SUSLDTRK_ACTIVE := 0
    IF 64-bit Mode
        THEN
```

```
            RIP := fallbackRIP
        ELSE
            EIP := fallbackEIP
    FI;
END
```

Flags Affected

None.

Intel C/C++ Compiler Intrinsic Equivalent

XEND void _xend(void);

SIMD Floating-Point Exceptions

None.

Other Exceptions

#UD	CPUID.(EAX=7, ECX=0):EBX.RTM[bit 11] = 0.
	If LOCK prefix is used.
#GP(0)	If RTM_ACTIVE = 0.

XGETBV—Get Value of Extended Control Register

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
NP 0F 01 D0	XGETBV	ZO	Valid	Valid	Reads an XCR specified by ECX into EDX:EAX.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	N/A	N/A	N/A	N/A

Description

Reads the contents of the extended control register (XCR) specified in the ECX register into registers EDX:EAX. (On processors that support the Intel 64 architecture, the high-order 32 bits of RCX are ignored.) The EDX register is loaded with the high-order 32 bits of the XCR and the EAX register is loaded with the low-order 32 bits. (On processors that support the Intel 64 architecture, the high-order 32 bits of each of RAX and RDX are cleared.) If fewer than 64 bits are implemented in the XCR being read, the values returned to EDX:EAX in unimplemented bit locations are undefined.

XCR0 is supported on any processor that supports the XGETBV instruction. If CPUID.(EAX=0DH,ECX=1):EAX.XG1[bit 2] = 1, executing XGETBV with ECX = 1 returns in EDX:EAX the logical-AND of XCR0 and the current value of the XINUSE state-component bitmap. This allows software to discover the state of the init optimization used by XSAVEOPT and XSAVES. See Chapter 13, "Managing State Using the XSAVE Feature Set," in Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1.

Use of any other value for ECX results in a general-protection (#GP) exception.

Operation

```
EDX:EAX := XCR[ECX];
```

Flags Affected

None.

Intel C/C++ Compiler Intrinsic Equivalent

XGETBV unsigned __int64 _xgetbv(unsigned int);

Protected Mode Exceptions

#GP(0) If an invalid XCR is specified in ECX (includes ECX = 1 if CPUID.(EAX=0DH,ECX=1):EAX.XG1[bit 2] = 0).

#UD If CPUID.01H:ECX.XSAVE[bit 26] = 0.
If CR4.OSXSAVE[bit 18] = 0.
If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP(0) If an invalid XCR is specified in ECX (includes ECX = 1 if CPUID.(EAX=0DH,ECX=1):EAX.XG1[bit 2] = 0).

#UD If CPUID.01H:ECX.XSAVE[bit 26] = 0.
If CR4.OSXSAVE[bit 18] = 0.
If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

XLAT/XLATB—Table Look-up Translation

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
D7	XLAT m8	ZO	Valid	Valid	Set AL to memory byte DS:[(E)BX + unsigned AL].
D7	XLATB	ZO	Valid	Valid	Set AL to memory byte DS:[(E)BX + unsigned AL].
REX.W + D7	XLATB	ZO	Valid	N.E.	Set AL to memory byte [RBX + unsigned AL].

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	N/A	N/A	N/A	N/A

Description

Locates a byte entry in a table in memory, using the contents of the AL register as a table index, then copies the contents of the table entry back into the AL register. The index in the AL register is treated as an unsigned integer. The XLAT and XLATB instructions get the base address of the table in memory from either the DS:EBX or the DS:BX registers (depending on the address-size attribute of the instruction, 32 or 16, respectively). (The DS segment may be overridden with a segment override prefix.)

At the assembly-code level, two forms of this instruction are allowed: the “explicit-operand” form and the “no-operand” form. The explicit-operand form (specified with the XLAT mnemonic) allows the base address of the table to be specified explicitly with a symbol. This explicit-operands form is provided to allow documentation; however, note that the documentation provided by this form can be misleading. That is, the symbol does not have to specify the correct base address. The base address is always specified by the DS:(E)BX registers, which must be loaded correctly before the XLAT instruction is executed.

The no-operands form (XLATB) provides a “short form” of the XLAT instructions. Here also the processor assumes that the DS:(E)BX registers contain the base address of the table.

In 64-bit mode, operation is similar to that in legacy or compatibility mode. AL is used to specify the table index (the operand size is fixed at 8 bits). RBX, however, is used to specify the table’s base address. See the summary chart at the beginning of this section for encoding data and limits.

Operation

```
IF AddressSize = 16
    THEN
        AL := (DS:BX + ZeroExtend(AL));
    ELSE IF (AddressSize = 32)
        AL := (DS:EBX + ZeroExtend(AL)); FI;
    ELSE (AddressSize = 64)
        AL := (RBX + ZeroExtend(AL));
FI;
```

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#UD	If the LOCK prefix is used.

XOR—Logical Exclusive OR

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
34 ib	XOR AL, imm8	I	Valid	Valid	AL XOR imm8.
35 iw	XOR AX, imm16	I	Valid	Valid	AX XOR imm16.
35 id	XOR EAX, imm32	I	Valid	Valid	EAX XOR imm32.
REX.W + 35 id	XOR RAX, imm32	I	Valid	N.E.	RAX XOR imm32 (sign-extended).
80 /6 ib	XOR r/m8, imm8	MI	Valid	Valid	r/m8 XOR imm8.
REX + 80 /6 ib	XOR r/m8*, imm8	MI	Valid	N.E.	r/m8 XOR imm8.
81 /6 iw	XOR r/m16, imm16	MI	Valid	Valid	r/m16 XOR imm16.
81 /6 id	XOR r/m32, imm32	MI	Valid	Valid	r/m32 XOR imm32.
REX.W + 81 /6 id	XOR r/m64, imm32	MI	Valid	N.E.	r/m64 XOR imm32 (sign-extended).
83 /6 ib	XOR r/m16, imm8	MI	Valid	Valid	r/m16 XOR imm8 (sign-extended).
83 /6 ib	XOR r/m32, imm8	MI	Valid	Valid	r/m32 XOR imm8 (sign-extended).
REX.W + 83 /6 ib	XOR r/m64, imm8	MI	Valid	N.E.	r/m64 XOR imm8 (sign-extended).
30 /r	XOR r/m8, r8	MR	Valid	Valid	r/m8 XOR r8.
REX + 30 /r	XOR r/m8*, r8*	MR	Valid	N.E.	r/m8 XOR r8.
31 /r	XOR r/m16, r16	MR	Valid	Valid	r/m16 XOR r16.
31 /r	XOR r/m32, r32	MR	Valid	Valid	r/m32 XOR r32.
REX.W + 31 /r	XOR r/m64, r64	MR	Valid	N.E.	r/m64 XOR r64.
32 /r	XOR r8, r/m8	RM	Valid	Valid	r8 XOR r/m8.
REX + 32 /r	XOR r8*, r/m8*	RM	Valid	N.E.	r8 XOR r/m8.
33 /r	XOR r16, r/m16	RM	Valid	Valid	r16 XOR r/m16.
33 /r	XOR r32, r/m32	RM	Valid	Valid	r32 XOR r/m32.
REX.W + 33 /r	XOR r64, r/m64	RM	Valid	N.E.	r64 XOR r/m64.

NOTES:

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

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
I	AL/AX/EAX/RAX	imm8/16/32	N/A	N/A
MI	ModRM:r/m (r, w)	imm8/16/32	N/A	N/A
MR	ModRM:r/m (r, w)	ModRM:reg (r)	N/A	N/A
RM	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A

Description

Performs a bitwise exclusive OR (XOR) operation on the destination (first) and source (second) operands and stores the result in the destination operand location. The source operand can be an immediate, a register, or a memory location; the destination operand can be a register or a memory location. (However, two memory operands cannot be used in one instruction.) Each bit of the result is 1 if the corresponding bits of the operands are different; each bit is 0 if the corresponding bits are the same.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.

In 64-bit mode, using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

Operation

```
DEST := DEST XOR SRC;
```

Flags Affected

The OF and CF flags are cleared; the SF, ZF, and PF flags are set according to the result. The state of the AF flag is undefined.

Protected Mode Exceptions

#GP(0)	If the destination operand points to a non-writable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

XORPD—Bitwise Logical XOR of Packed Double Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 57/r XORPD xmm1, xmm2/m128	A	V/V	SSE2	Return the bitwise logical XOR of packed double precision floating-point values in xmm1 and xmm2/mem.
VEX.128.66.0F.WIG 57 /r VXORPD xmm1,xmm2, xmm3/m128	B	V/V	AVX	Return the bitwise logical XOR of packed double precision floating-point values in xmm2 and xmm3/mem.
VEX.256.66.0F.WIG 57 /r VXORPD ymm1, ymm2, ymm3/m256	B	V/V	AVX	Return the bitwise logical XOR of packed double precision floating-point values in ymm2 and ymm3/mem.
EVEX.128.66.0F.W1 57 /r VXORPD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	AVX512VL AVX512DQ	Return the bitwise logical XOR of packed double precision floating-point values in xmm2 and xmm3/m128/m64bcst subject to writemask k1.
EVEX.256.66.0F.W1 57 /r VXORPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	AVX512VL AVX512DQ	Return the bitwise logical XOR of packed double precision floating-point values in ymm2 and ymm3/m256/m64bcst subject to writemask k1.
EVEX.512.66.0F.W1 57 /r VXORPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512DQ	Return the bitwise logical XOR of packed double precision floating-point values in zmm2 and zmm3/m512/m64bcst subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a bitwise logical XOR of the two, four or eight packed double precision floating-point values from the first source operand and the second source operand, and stores the result in the destination operand.

EVEX.512 encoded version: The first source operand is a ZMM register. The second source operand can be a ZMM register or a vector memory location. The destination operand is a ZMM register conditionally updated with write-mask k1.

VEX.256 and EVEX.256 encoded versions: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register (conditionally updated with writemask k1 in case of EVEX). The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 and EVEX.128 encoded versions: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register (conditionally updated with writemask k1 in case of EVEX). The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding register destination are unmodified.

Operation

VXORPD (EVEX Encoded Versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b == 1) AND (SRC2 *is memory*)
                THEN DEST[i+63:i] := SRC1[i+63:i] BITWISE XOR SRC2[63:0];
                ELSE DEST[i+63:i] := SRC1[i+63:i] BITWISE XOR SRC2[i+63:i];
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+63:i] = 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VXORPD (VEX.256 Encoded Version)

```
DEST[63:0] := SRC1[63:0] BITWISE XOR SRC2[63:0]
DEST[127:64] := SRC1[127:64] BITWISE XOR SRC2[127:64]
DEST[191:128] := SRC1[191:128] BITWISE XOR SRC2[191:128]
DEST[255:192] := SRC1[255:192] BITWISE XOR SRC2[255:192]
DEST[MAXVL-1:256] := 0
```

VXORPD (VEX.128 Encoded Version)

```
DEST[63:0] := SRC1[63:0] BITWISE XOR SRC2[63:0]
DEST[127:64] := SRC1[127:64] BITWISE XOR SRC2[127:64]
DEST[MAXVL-1:128] := 0
```

XORPD (128-bit Legacy SSE Version)

```
DEST[63:0] := DEST[63:0] BITWISE XOR SRC[63:0]
DEST[127:64] := DEST[127:64] BITWISE XOR SRC[127:64]
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VXORPD __m512d _mm512_xor_pd (__m512d a, __m512d b);
VXORPD __m512d _mm512_mask_xor_pd (__m512d a, __mmask8 m, __m512d b);
VXORPD __m512d _mm512_maskz_xor_pd (__mmask8 m, __m512d a);
VXORPD __m256d _mm256_xor_pd (__m256d a, __m256d b);
VXORPD __m256d _mm256_mask_xor_pd (__m256d a, __mmask8 m, __m256d b);
VXORPD __m256d _mm256_maskz_xor_pd (__mmask8 m, __m256d a);
XORPD __m128d _mm_xor_pd (__m128d a, __m128d b);
VXORPD __m128d _mm_mask_xor_pd (__m128d a, __mmask8 m, __m128d b);
VXORPD __m128d _mm_maskz_xor_pd (__mmask8 m, __m128d a);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instructions, see Table 2-21, “Type 4 Class Exception Conditions.”

EVEX-encoded instructions, see Table 2-49, “Type E4 Class Exception Conditions.”

XORPS—Bitwise Logical XOR of Packed Single Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 57 /r XORPS xmm1, xmm2/m128	A	V/V	SSE	Return the bitwise logical XOR of packed single-precision floating-point values in xmm1 and xmm2/mem.
VEX.128.0F.WIG 57 /r VXORPS xmm1,xmm2, xmm3/m128	B	V/V	AVX	Return the bitwise logical XOR of packed single-precision floating-point values in xmm2 and xmm3/mem.
VEX.256.0F.WIG 57 /r VXORPS ymm1, ymm2, ymm3/m256	B	V/V	AVX	Return the bitwise logical XOR of packed single-precision floating-point values in ymm2 and ymm3/mem.
EVEX.128.0F.W0 57 /r VXORPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	AVX512VL AVX512DQ	Return the bitwise logical XOR of packed single-precision floating-point values in xmm2 and xmm3/m128/m32bcst subject to writemask k1.
EVEX.256.0F.W0 57 /r VXORPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	AVX512VL AVX512DQ	Return the bitwise logical XOR of packed single-precision floating-point values in ymm2 and ymm3/m256/m32bcst subject to writemask k1.
EVEX.512.0F.W0 57 /r VXORPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	C	V/V	AVX512DQ	Return the bitwise logical XOR of packed single-precision floating-point values in zmm2 and zmm3/m512/m32bcst subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A
B	N/A	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Performs a bitwise logical XOR of the four, eight or sixteen packed single-precision floating-point values from the first source operand and the second source operand, and stores the result in the destination operand

EVEX.512 encoded version: The first source operand is a ZMM register. The second source operand can be a ZMM register or a vector memory location. The destination operand is a ZMM register conditionally updated with writemask k1.

VEX.256 and EVEX.256 encoded versions: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register (conditionally updated with writemask k1 in case of EVEX). The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 and EVEX.128 encoded versions: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register (conditionally updated with writemask k1 in case of EVEX). The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding register destination are unmodified.

Operation

VXORPS (EVEX Encoded Versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b == 1) AND (SRC2 *is memory*)
                THEN DEST[i+31:i] := SRC1[i+31:i] BITWISE XOR SRC2[31:0];
                ELSE DEST[i+31:i] := SRC1[i+31:i] BITWISE XOR SRC2[i+31:i];
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*            ; zeroing-masking
                    DEST[i+31:i] = 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

VXORPS (VEX.256 Encoded Version)

```
DEST[31:0] := SRC1[31:0] BITWISE XOR SRC2[31:0]
DEST[63:32] := SRC1[63:32] BITWISE XOR SRC2[63:32]
DEST[95:64] := SRC1[95:64] BITWISE XOR SRC2[95:64]
DEST[127:96] := SRC1[127:96] BITWISE XOR SRC2[127:96]
DEST[159:128] := SRC1[159:128] BITWISE XOR SRC2[159:128]
DEST[191:160] := SRC1[191:160] BITWISE XOR SRC2[191:160]
DEST[223:192] := SRC1[223:192] BITWISE XOR SRC2[223:192]
DEST[255:224] := SRC1[255:224] BITWISE XOR SRC2[255:224].
DEST[MAXVL-1:256] := 0
```

VXORPS (VEX.128 Encoded Version)

```
DEST[31:0] := SRC1[31:0] BITWISE XOR SRC2[31:0]
DEST[63:32] := SRC1[63:32] BITWISE XOR SRC2[63:32]
DEST[95:64] := SRC1[95:64] BITWISE XOR SRC2[95:64]
DEST[127:96] := SRC1[127:96] BITWISE XOR SRC2[127:96]
DEST[MAXVL-1:128] := 0
```

XORPS (128-bit Legacy SSE Version)

```
DEST[31:0] := SRC1[31:0] BITWISE XOR SRC2[31:0]
DEST[63:32] := SRC1[63:32] BITWISE XOR SRC2[63:32]
DEST[95:64] := SRC1[95:64] BITWISE XOR SRC2[95:64]
DEST[127:96] := SRC1[127:96] BITWISE XOR SRC2[127:96]
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VXORPS __m512 _mm512_xor_ps (__m512 a, __m512 b);
VXORPS __m512 _mm512_mask_xor_ps (__m512 a, __mmask16 m, __m512 b);
VXORPS __m512 _mm512_maskz_xor_ps (__mmask16 m, __m512 a);
VXORPS __m256 _mm256_xor_ps (__m256 a, __m256 b);
VXORPS __m256 _mm256_mask_xor_ps (__m256 a, __mmask8 m, __m256 b);
VXORPS __m256 _mm256_maskz_xor_ps (__mmask8 m, __m256 a);
XORPS __m128 _mm_xor_ps (__m128 a, __m128 b);
VXORPS __m128 _mm_mask_xor_ps (__m128 a, __mmask8 m, __m128 b);
VXORPS __m128 _mm_maskz_xor_ps (__mmask8 m, __m128 a);

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instructions, see Table 2-21, “Type 4 Class Exception Conditions.”
EVEX-encoded instructions, see Table 2-49, “Type E4 Class Exception Conditions.”

XRESLDTRK—Resume Tracking Load Addresses

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 01 E9 XRESLDTRK	ZO	V/V	TSXLDTRK	Specifies the end of an Intel TSX suspend read address tracking region.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
ZO	N/A	N/A	N/A	N/A	N/A

Description

The instruction marks the end of an Intel TSX (RTM) suspend load address tracking region. If the instruction is used inside a suspend load address tracking region it will end the suspend region and all following load addresses will be added to the transaction read set. If this instruction is used inside an active transaction but not in a suspend region it will cause transaction abort.

If the instruction is used outside of a transactional region it behaves like a NOP.

Chapter 16, "Programming with Intel® Transactional Synchronization Extensions" in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1 provides additional information on Intel® TSX Suspend Load Address Tracking.

Operation

XRESLDTRK

```
IF RTM_ACTIVE = 1:
    IF SUSLDTRK_ACTIVE = 1:
        SUSLDTRK_ACTIVE := 0
    ELSE:
        RTM_ABORT
ELSE:
    NOP
```

Flags Affected

None.

Intel C/C++ Compiler Intrinsic Equivalent

XRESLDTRK void _xresldtrk(void);

SIMD Floating-Point Exceptions

None.

Other Exceptions

#UD	If CPUID.(EAX=7, ECX=0):EDX.TSXLDTRK[bit 16] = 0.
	If the LOCK prefix is used.

XRSTOR—Restore Processor Extended States

Opcode / Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F AE /5 XRSTOR mem	M	V/V	XSAVE	Restore state components specified by EDX:EAX from mem.
NP REX.W + 0F AE /5 XRSTOR64 mem	M	V/N.E.	XSAVE	Restore state components specified by EDX:EAX from mem.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r)	N/A	N/A	N/A

Description

Performs a full or partial restore of processor state components from the XSAVE area located at the memory address specified by the source operand. The implicit EDX:EAX register pair specifies a 64-bit instruction mask. The specific state components restored correspond to the bits set in the requested-feature bitmap (RFBM), which is the logical-AND of EDX:EAX and XCR0.

The format of the XSAVE area is detailed in Section 13.4, "XSAVE Area," of Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1. Like FXRSTOR and FXSAVE, the memory format used for x87 state depends on a REX.W prefix; see Section 13.5.1, "x87 State" of Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1.

Section 13.8, "Operation of XRSTOR," of Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1 provides a detailed description of the operation of the XRSTOR instruction. The following items provide a high-level outline:

- Execution of XRSTOR may take one of two forms: standard and compacted. Bit 63 of the XCOMP_BV field in the XSAVE header determines which form is used: value 0 specifies the standard form, while value 1 specifies the compacted form.
- If RFBM[*i*] = 0, XRSTOR does not update state component *i*.[1]
- If RFBM[*i*] = 1 and bit *i* is clear in the XSTATE_BV field in the XSAVE header, XRSTOR initializes state component *i*.
- If RFBM[*i*] = 1 and XSTATE_BV[*i*] = 1, XRSTOR loads state component *i* from the XSAVE area.
- The standard form of XRSTOR treats MXCSR (which is part of state component 1 — SSE) differently from the XMM registers. If either form attempts to load MXCSR with an illegal value, a general-protection exception (#GP) occurs.
- XRSTOR loads the internal value XRSTOR_INFO, which may be used to optimize a subsequent execution of XSAVEOPT or XSAVES.
- Immediately following an execution of XRSTOR, the processor tracks as in-use (not in initial configuration) any state component *i* for which RFBM[*i*] = 1 and XSTATE_BV[*i*] = 1; it tracks as modified any state component *i* for which RFBM[*i*] = 0.

Use of a source operand not aligned to 64-byte boundary (for 64-bit and 32-bit modes) results in a general-protection (#GP) exception. In 64-bit mode, the upper 32 bits of RDX and RAX are ignored.

See Section 13.6, "Processor Tracking of XSAVE-Managed State," of Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1 for discussion of the bitmaps XINUSE and XMODIFIED and of the quantity XRSTOR_INFO.

1. There is an exception if RFBM[1] = 0 and RFBM[2] = 1. In this case, the standard form of XRSTOR will load MXCSR from memory, even though MXCSR is part of state component 1 — SSE. The compacted form of XRSTOR does not make this exception.

Operation

```
RFBM := XCR0 AND EDX:EAX;     /* bitwise logical AND */
COMPMASK := XCOMP_BV field from XSAVE header;
RSTORMASK := XSTATE_BV field from XSAVE header;

IF COMPMASK[63] = 0
    THEN
        /* Standard form of XRSTOR */
        TO_BE_RESTORED := RFBM AND RSTORMASK;
        TO_BE_INITIALIZED := RFBM AND NOT RSTORMASK;

        IF TO_BE_RESTORED[0] = 1
            THEN
                XINUSE[0] := 1;
                load x87 state from legacy region of XSAVE area;
        ELSIF TO_BE_INITIALIZED[0] = 1
            THEN
                XINUSE[0] := 0;
                initialize x87 state;
        FI;

        IF RFBM[1] = 1 OR RFBM[2] = 1
            THEN load MXCSR from legacy region of XSAVE area;
        FI;

        IF TO_BE_RESTORED[1] = 1
            THEN
                XINUSE[1] := 1;
                load XMM registers from legacy region of XSAVE area; // this step does not load MXCSR
        ELSIF TO_BE_INITIALIZED[1] = 1
            THEN
                XINUSE[1] := 0;
                set all XMM registers to 0; // this step does not initialize MXCSR
        FI;

        FOR i := 2 TO 62
            IF TO_BE_RESTORED[i] = 1
                THEN
                    XINUSE[i] := 1;
                    load XSAVE state component i at offset n from base of XSAVE area;
                        // n enumerated by CPUID(EAX=0DH,ECX=i):EBX)
            ELSIF TO_BE_INITIALIZED[i] = 1
                THEN
                    XINUSE[i] := 0;
                    initialize XSAVE state component i;
            FI;
        ENDFOR;

    ELSE
        /* Compacted form of XRSTOR */
        IF CPUID.(EAX=0DH,ECX=1):EAX.XSAVEC[bit 1] = 0
            THEN     /* compacted form not supported */
                #GP(0);
        FI;
```

```
        FORMAT = COMPMASK AND 7FFFFFFF_FFFFFFFFH;
        RESTORE_FEATURES = FORMAT AND RFBM;
        TO_BE_RESTORED := RESTORE_FEATURES AND RSTORMASK;
        FORCE_INIT := RFBM AND NOT FORMAT;
        TO_BE_INITIALIZED = (RFBM AND NOT RSTORMASK) OR FORCE_INIT;

        IF TO_BE_RESTORED[0] = 1
            THEN
                XINUSE[0] := 1;
                load x87 state from legacy region of XSAVE area;
        ELSIF TO_BE_INITIALIZED[0] = 1
            THEN
                XINUSE[0] := 0;
                initialize x87 state;
        FI;

        IF TO_BE_RESTORED[1] = 1
            THEN
                XINUSE[1] := 1;
                load SSE state from legacy region of XSAVE area; // this step loads the XMM registers and MXCSR
        ELSIF TO_BE_INITIALIZED[1] = 1
            THEN
                set all XMM registers to 0;
                XINUSE[1] := 0;
                MXCSR := 1F80H;
        FI;

        NEXT_FEATURE_OFFSET = 576;          // Legacy area and XSAVE header consume 576 bytes
        FOR i := 2 TO 62
            IF FORMAT[i] = 1
                THEN
                    IF TO_BE_RESTORED[i] = 1
                        THEN
                            XINUSE[i] := 1;
                            load XSAVE state component i at offset NEXT_FEATURE_OFFSET from base of XSAVE area;
                    FI;
                    NEXT_FEATURE_OFFSET = NEXT_FEATURE_OFFSET + n (n enumerated by CPUID(EAX=0DH,ECX=i):EAX);
            FI;
            IF TO_BE_INITIALIZED[i] = 1
                THEN
                    XINUSE[i] := 0;
                    initialize XSAVE state component i;
            FI;
        ENDFOR;
FI;

XMODIFIED := NOT RFBM;

IF in VMX non-root operation
    THEN VMXNR := 1;
    ELSE VMXNR := 0;
FI;
LAXA := linear address of XSAVE area;
```

XRSTOR_INFO := ⟨CPL,VMXNR,LAXA,COMPMASK⟩;

Flags Affected

None.

Intel C/C++ Compiler Intrinsic Equivalent

XRSTOR void _xrstor(void * , unsigned __int64);
XRSTOR void _xrstor64(void * , unsigned __int64);

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
	If bit 63 of the XCOMP_BV field of the XSAVE header is 1 and CPUID.(EAX=0DH,ECX=1):EAX.XSAVEC[bit 1] = 0.
	If the standard form is executed and a bit in XCR0 is 0 and the corresponding bit in the XSTATE_BV field of the XSAVE header is 1.
	If the standard form is executed and bytes 23:8 of the XSAVE header are not all zero.
	If the compacted form is executed and a bit in XCR0 is 0 and the corresponding bit in the XCOMP_BV field of the XSAVE header is 1.
	If the compacted form is executed and a bit in the XCOMP_BV field in the XSAVE header is 0 and the corresponding bit in the XSTATE_BV field is 1.
	If the compacted form is executed and bytes 63:16 of the XSAVE header are not all zero.
	If attempting to write any reserved bits of the MXCSR register with 1.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If the LOCK prefix is used.
#AC	If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 64-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).

Real-Address Mode Exceptions

#GP	If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
	If any part of the operand lies outside the effective address space from 0 to FFFFH.
	If bit 63 of the XCOMP_BV field of the XSAVE header is 1 and CPUID.(EAX=0DH,ECX=1):EAX.XSAVEC[bit 1] = 0.
	If the standard form is executed and a bit in XCR0 is 0 and the corresponding bit in the XSTATE_BV field of the XSAVE header is 1.
	If the standard form is executed and bytes 23:8 of the XSAVE header are not all zero.
	If the compacted form is executed and a bit in XCR0 is 0 and the corresponding bit in the XCOMP_BV field of the XSAVE header is 1.

	If the compacted form is executed and a bit in the XCOMP_BV field in the XSAVE header is 0 and the corresponding bit in the XSTATE_BV field is 1.
	If the compacted form is executed and bytes 63:16 of the XSAVE header are not all zero.
	If attempting to write any reserved bits of the MXCSR register with 1.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0)	If a memory address is in a non-canonical form.
	If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
	If bit 63 of the XCOMP_BV field of the XSAVE header is 1 and CPUID.(EAX=0DH,ECX=1):EAX.XSAVEC[bit 1] = 0.
	If the standard form is executed and a bit in XCR0 is 0 and the corresponding bit in the XSTATE_BV field of the XSAVE header is 1.
	If the standard form is executed and bytes 23:8 of the XSAVE header are not all zero.
	If the compacted form is executed and a bit in XCR0 is 0 and the corresponding bit in the XCOMP_BV field of the XSAVE header is 1.
	If the compacted form is executed and a bit in the XCOMP_BV field in the XSAVE header is 0 and the corresponding bit in the XSTATE_BV field is 1.
	If the compacted form is executed and bytes 63:16 of the XSAVE header are not all zero.
	If attempting to write any reserved bits of the MXCSR register with 1.
#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If the LOCK prefix is used.
#AC	If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 64-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).

XRSTORS—Restore Processor Extended States Supervisor

Opcode / Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F C7 /3 XRSTORS mem	M	V/V	XSS	Restore state components specified by EDX:EAX from mem.
NP REX.W + 0F C7 /3 XRSTORS64 mem	M	V/N.E.	XSS	Restore state components specified by EDX:EAX from mem.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r)	N/A	N/A	N/A

Description

Performs a full or partial restore of processor state components from the XSAVE area located at the memory address specified by the source operand. The implicit EDX:EAX register pair specifies a 64-bit instruction mask. The specific state components restored correspond to the bits set in the requested-feature bitmap (RFBM), which is the logical-AND of EDX:EAX and the logical-OR of XCR0 with the IA32_XSS MSR. XRSTORS may be executed only if CPL = 0.

The format of the XSAVE area is detailed in Section 13.4, "XSAVE Area," of Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1. Like FXRSTOR and FXSAVE, the memory format used for x87 state depends on a REX.W prefix; see Section 13.5.1, "x87 State" of Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1.

Section 13.12, "Operation of XRSTORS," of Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1 provides a detailed description of the operation of the XRSTOR instruction. The following items provide a high-level outline:

- Execution of XRSTORS is similar to that of the compacted form of XRSTOR; XRSTORS cannot restore from an XSAVE area in which the extended region is in the standard format (see Section 13.4.3, "Extended Region of an XSAVE Area" of Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1).
- XRSTORS differs from XRSTOR in that it can restore state components corresponding to bits set in the IA32_XSS MSR.
- If RFBM[*i*] = 0, XRSTORS does not update state component *i*.
- If RFBM[*i*] = 1 and bit *i* is clear in the XSTATE_BV field in the XSAVE header, XRSTORS initializes state component *i*.
- If RFBM[*i*] = 1 and XSTATE_BV[*i*] = 1, XRSTORS loads state component *i* from the XSAVE area.
- If XRSTORS attempts to load MXCSR with an illegal value, a general-protection exception (#GP) occurs.
- XRSTORS loads the internal value XRSTOR_INFO, which may be used to optimize a subsequent execution of XSAVEOPT or XSAVES.
- Immediately following an execution of XRSTORS, the processor tracks as in-use (not in initial configuration) any state component *i* for which RFBM[*i*] = 1 and XSTATE_BV[*i*] = 1; it tracks as modified any state component *i* for which RFBM[*i*] = 0.

Use of a source operand not aligned to 64-byte boundary (for 64-bit and 32-bit modes) results in a general-protection (#GP) exception. In 64-bit mode, the upper 32 bits of RDX and RAX are ignored.

See Section 13.6, "Processor Tracking of XSAVE-Managed State," of Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1 for discussion of the bitmaps XINUSE and XMODIFIED and of the quantity XRSTOR_INFO.

Operation

```
RFBM := (XCR0 OR IA32_XSS) AND EDX:EAX;            /* bitwise logical OR and AND */
COMPMASK := XCOMP_BV field from XSAVE header;
RSTORMASK := XSTATE_BV field from XSAVE header;

FORMAT = COMPMASK AND 7FFFFFFF_FFFFFFFFH;
RESTORE_FEATURES = FORMAT AND RFBM;
TO_BE_RESTORED := RESTORE_FEATURES AND RSTORMASK;
FORCE_INIT := RFBM AND NOT FORMAT;
TO_BE_INITIALIZED = (RFBM AND NOT RSTORMASK) OR FORCE_INIT;

IF TO_BE_RESTORED[0] = 1
    THEN
        XINUSE[0] := 1;
        load x87 state from legacy region of XSAVE area;
ELSIF TO_BE_INITIALIZED[0] = 1
    THEN
        XINUSE[0] := 0;
        initialize x87 state;
FI;

IF TO_BE_RESTORED[1] = 1
    THEN
        XINUSE[1] := 1;
        load SSE state from legacy region of XSAVE area; // this step loads the XMM registers and MXCSR
ELSIF TO_BE_INITIALIZED[1] = 1
    THEN
        set all XMM registers to 0;
        XINUSE[1] := 0;
        MXCSR := 1F80H;
FI;

NEXT_FEATURE_OFFSET = 576;         // Legacy area and XSAVE header consume 576 bytes
FOR i := 2 TO 62
    IF FORMAT[i] = 1
        THEN
            IF TO_BE_RESTORED[i] = 1
                THEN
                    XINUSE[i] := 1;
                    load XSAVE state component i at offset NEXT_FEATURE_OFFSET from base of XSAVE area;
            FI;
            NEXT_FEATURE_OFFSET = NEXT_FEATURE_OFFSET + n (n enumerated by CPUID(EAX=0DH,ECX=i):EAX);
    FI;
    IF TO_BE_INITIALIZED[i] = 1
        THEN
            XINUSE[i] := 0;
            initialize XSAVE state component i;
    FI;
ENDFOR;

XMODIFIED := NOT RFBM;

IF in VMX non-root operation
    THEN VMXNR := 1;
```

```
    ELSE VMXNR := 0;
FI;
LAXA := linear address of XSAVE area;
XRSTOR_INFO := ⟨CPL,VMXNR,LAXA,COMPMASK⟩;
```

Flags Affected

None.

Intel C/C++ Compiler Intrinsic Equivalent

XRSTORS void _xrstors(void * , unsigned __int64);
XRSTORS64 void _xrstors64(void * , unsigned __int64);

Protected Mode Exceptions

#GP(0)	If CPL > 0.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
	If bit 63 of the XCOMP_BV field of the XSAVE header is 0.
	If a bit in XCR0|IA32_XSS is 0 and the corresponding bit in the XCOMP_BV field of the XSAVE header is 1.
	If a bit in the XCOMP_BV field in the XSAVE header is 0 and the corresponding bit in the XSTATE_BV field is 1.
	If bytes 63:16 of the XSAVE header are not all zero.
	If attempting to write any reserved bits of the MXCSR register with 1.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSS[bit 3] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
	If any part of the operand lies outside the effective address space from 0 to FFFFH.
	If bit 63 of the XCOMP_BV field of the XSAVE header is 0.
	If a bit in XCR0|IA32_XSS is 0 and the corresponding bit in the XCOMP_BV field of the XSAVE header is 1.
	If a bit in the XCOMP_BV field in the XSAVE header is 0 and the corresponding bit in the XSTATE_BV field is 1.
	If bytes 63:16 of the XSAVE header are not all zero.
	If attempting to write any reserved bits of the MXCSR register with 1.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSS[bit 3] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0) If CPL > 0.

If a memory address is in a non-canonical form.

If a memory operand is not aligned on a 64-byte boundary, regardless of segment.

If bit 63 of the XCOMP_BV field of the XSAVE header is 0.

If a bit in XCR0|IA32_XSS is 0 and the corresponding bit in the XCOMP_BV field of the XSAVE header is 1.

If a bit in the XCOMP_BV field in the XSAVE header is 0 and the corresponding bit in the XSTATE_BV field is 1.

If bytes 63:16 of the XSAVE header are not all zero.

If attempting to write any reserved bits of the MXCSR register with 1.

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

#PF(fault-code) If a page fault occurs.

#NM If CR0.TS[bit 3] = 1.

#UD If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSS[bit 3] = 0.

If CR4.OSXSAVE[bit 18] = 0.

If the LOCK prefix is used.

XSAVE—Save Processor Extended States

Opcode / Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F AE /4 XSAVE mem	M	V/V	XSAVE	Save state components specified by EDX:EAX to mem.
NP REX.W + 0F AE /4 XSAVE64 mem	M	V/N.E.	XSAVE	Save state components specified by EDX:EAX to mem.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r, w)	N/A	N/A	N/A

Description

Performs a full or partial save of processor state components to the XSAVE area located at the memory address specified by the destination operand. The implicit EDX:EAX register pair specifies a 64-bit instruction mask. The specific state components saved correspond to the bits set in the requested-feature bitmap (RFBM), which is the logical-AND of EDX:EAX and XCR0.

The format of the XSAVE area is detailed in Section 13.4, “XSAVE Area,” of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1. Like FXRSTOR and FXSAVE, the memory format used for x87 state depends on a REX.W prefix; see Section 13.5.1, “x87 State” of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1.

Section 13.7, “Operation of XSAVE,” of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1 provides a detailed description of the operation of the XSAVE instruction. The following items provide a high-level outline:

- XSAVE saves state component *i* if and only if RFBM[*i*] = 1.[1]
- XSAVE does not modify bytes 511:464 of the legacy region of the XSAVE area (see Section 13.4.1, “Legacy Region of an XSAVE Area” of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1).
- XSAVE reads the XSTATE_BV field of the XSAVE header (see Section 13.4.2, “XSAVE Header” of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1) and writes a modified value back to memory as follows. If RFBM[*i*] = 1, XSAVE writes XSTATE_BV[*i*] with the value of XINUSE[*i*]. (XINUSE is a bitmap by which the processor tracks the status of various state components. See Section 13.6, “Processor Tracking of XSAVE-Managed State” of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1.) If RFBM[*i*] = 0, XSAVE writes XSTATE_BV[*i*] with the value that it read from memory (it does not modify the bit). XSAVE does not write to any part of the XSAVE header other than the XSTATE_BV field.
- XSAVE always uses the standard format of the extended region of the XSAVE area (see Section 13.4.3, “Extended Region of an XSAVE Area” of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1).

Use of a destination operand not aligned to 64-byte boundary (in either 64-bit or 32-bit modes) results in a general-protection (#GP) exception. In 64-bit mode, the upper 32 bits of RDX and RAX are ignored.

1. An exception is made for MXCSR and MXCSR_MASK, which belong to state component 1 — SSE. XSAVE saves these values to memory if either RFBM[1] or RFBM[2] is 1.

Operation

```
RFBM := XCR0 AND EDX:EAX;    /* bitwise logical AND */
OLD_BV := XSTATE_BV field from XSAVE header;

IF RFBM[0] = 1
    THEN store x87 state into legacy region of XSAVE area;
FI;

IF RFBM[1] = 1
    THEN store XMM registers into legacy region of XSAVE area; // this step does not save MXCSR or MXCSR_MASK
FI;

IF RFBM[1] = 1 OR RFBM[2] = 1
    THEN store MXCSR and MXCSR_MASK into legacy region of XSAVE area;
FI;

FOR i := 2 TO 62
    IF RFBM[i] = 1
        THEN save XSAVE state component i at offset n from base of XSAVE area (n enumerated by CPUID(EAX=0DH,ECX=i):EBX);
    FI;
ENDFOR;

XSTATE_BV field in XSAVE header := (OLD_BV AND NOT RFBM) OR (XINUSE AND RFBM);
```

Flags Affected

None.

Intel C/C++ Compiler Intrinsic Equivalent

```
XSAVE void _xsave( void * , unsigned __int64);
XSAVE void _xsave64( void * , unsigned __int64);
```

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If the LOCK prefix is used.
#AC	If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 64-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).

Real-Address Mode Exceptions

#GP	If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0)	If the memory address is in a non-canonical form.
	If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If the LOCK prefix is used.
#AC	If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 64-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).

XSAVEC—Save Processor Extended States With Compaction

Opcode / Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F C7 /4 XSAVEC mem	M	V/V	XSAVEC	Save state components specified by EDX:EAX to mem with compaction.
NP REX.W + 0F C7 /4 XSAVEC64 mem	M	V/N.E.	XSAVEC	Save state components specified by EDX:EAX to mem with compaction.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (w)	N/A	N/A	N/A

Description

Performs a full or partial save of processor state components to the XSAVE area located at the memory address specified by the destination operand. The implicit EDX:EAX register pair specifies a 64-bit instruction mask. The specific state components saved correspond to the bits set in the requested-feature bitmap (RFBM), which is the logical-AND of EDX:EAX and XCR0.

The format of the XSAVE area is detailed in Section 13.4, "XSAVE Area," of Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1. Like FXRSTOR and FXSAVE, the memory format used for x87 state depends on a REX.W prefix; see Section 13.5.1, "x87 State" of Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1.

Section 13.10, "Operation of XSAVEC," of Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1 provides a detailed description of the operation of the XSAVEC instruction. The following items provide a high-level outline:

- Execution of XSAVEC is similar to that of XSAVE. XSAVEC differs from XSAVE in that it uses compaction and that it may use the init optimization.
- XSAVEC saves state component *i* if and only if RFBM[*i*] = 1 and XINUSE[*i*] = 1.[1] (XINUSE is a bitmap by which the processor tracks the status of various state components. See Section 13.6, "Processor Tracking of XSAVE-Managed State" of Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1.)
- XSAVEC does not modify bytes 511:464 of the legacy region of the XSAVE area (see Section 13.4.1, "Legacy Region of an XSAVE Area" of Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1).
- XSAVEC writes the logical AND of RFBM and XINUSE to the XSTATE_BV field of the XSAVE header.[2,3] (See Section 13.4.2, "XSAVE Header" of Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1.) XSAVEC sets bit 63 of the XCOMP_BV field and sets bits 62:0 of that field to RFBM[62:0]. XSAVEC does not write to any parts of the XSAVE header other than the XSTATE_BV and XCOMP_BV fields.
- XSAVEC always uses the compacted format of the extended region of the XSAVE area (see Section 13.4.3, "Extended Region of an XSAVE Area" of Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1).

Use of a destination operand not aligned to 64-byte boundary (in either 64-bit or 32-bit modes) results in a general-protection (#GP) exception. In 64-bit mode, the upper 32 bits of RDX and RAX are ignored.

1. There is an exception for state component 1 (SSE). MXCSR is part of SSE state, but XINUSE[1] may be 0 even if MXCSR does not have its initial value of 1F80H. In this case, XSAVEC saves SSE state as long as RFBM[1] = 1.
2. Unlike XSAVE and XSAVEOPT, XSAVEC clears bits in the XSTATE_BV field that correspond to bits that are clear in RFBM.
3. There is an exception for state component 1 (SSE). MXCSR is part of SSE state, but XINUSE[1] may be 0 even if MXCSR does not have its initial value of 1F80H. In this case, XSAVEC sets XSTATE_BV[1] to 1 as long as RFBM[1] = 1.

Operation

```
RFBM := XCR0 AND EDX:EAX;               /* bitwise logical AND */
TO_BE_SAVED := RFBM AND XINUSE;         /* bitwise logical AND */
If MXCSR ≠ 1F80H AND RFBM[1]
    TO_BE_SAVED[1] = 1;
FI;

IF TO_BE_SAVED[0] = 1
    THEN store x87 state into legacy region of XSAVE area;
FI;

IF TO_BE_SAVED[1] = 1
    THEN store SSE state into legacy region of XSAVE area; // this step saves the XMM registers, MXCSR, and MXCSR_MASK
FI;

NEXT_FEATURE_OFFSET = 576;              // Legacy area and XSAVE header consume 576 bytes
FOR i := 2 TO 62
    IF RFBM[i] = 1
        THEN
            IF TO_BE_SAVED[i]
                THEN save XSAVE state component i at offset NEXT_FEATURE_OFFSET from base of XSAVE area;
            FI;
            NEXT_FEATURE_OFFSET = NEXT_FEATURE_OFFSET + n (n enumerated by CPUID(EAX=0DH,ECX=i):EAX);
    FI;
ENDFOR;

XSTATE_BV field in XSAVE header := TO_BE_SAVED;
XCOMP_BV field in XSAVE header := RFBM OR 80000000_00000000H;
```

Flags Affected

None.

Intel C/C++ Compiler Intrinsic Equivalent

```
XSAVEC void _xsavec( void * , unsigned __int64);
XSAVEC64 void _xsavec64( void * , unsigned __int64);
```

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSAVEC[bit 1] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If the LOCK prefix is used.

#AC	If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 64-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).

Real-Address Mode Exceptions

#GP	If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSAVEC[bit 1] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0)	If the memory address is in a non-canonical form.
	If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSAVEC[bit 1] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If the LOCK prefix is used.
#AC	If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 64-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).

XSAVEOPT—Save Processor Extended States Optimized

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F AE /6 XSAVEOPT mem	M	V/V	XSAVEOPT	Save state components specified by EDX:EAX to mem, optimizing if possible.
NP REX.W + 0F AE /6 XSAVEOPT64 mem	M	V/V	XSAVEOPT	Save state components specified by EDX:EAX to mem, optimizing if possible.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r, w)	N/A	N/A	N/A

Description

Performs a full or partial save of processor state components to the XSAVE area located at the memory address specified by the destination operand. The implicit EDX:EAX register pair specifies a 64-bit instruction mask. The specific state components saved correspond to the bits set in the requested-feature bitmap (RFBM), which is the logical-AND of EDX:EAX and XCR0.

The format of the XSAVE area is detailed in Section 13.4, "XSAVE Area," of Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1. Like FXRSTOR and FXSAVE, the memory format used for x87 state depends on a REX.W prefix; see Section 13.5.1, "x87 State" of Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1.

Section 13.9, "Operation of XSAVEOPT," of Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1 provides a detailed description of the operation of the XSAVEOPT instruction. The following items provide a high-level outline:

- Execution of XSAVEOPT is similar to that of XSAVE. XSAVEOPT differs from XSAVE in that it may use the init and modified optimizations. The performance of XSAVEOPT will be equal to or better than that of XSAVE.
- XSAVEOPT saves state component *i* only if RFBM[*i*] = 1 and XINUSE[*i*] = 1.[1] (XINUSE is a bitmap by which the processor tracks the status of various state components. See Section 13.6, "Processor Tracking of XSAVE-Managed State"of Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1.) Even if both bits are 1, XSAVEOPT may optimize and not save state component *i* if (1) state component *i* has not been modified since the last execution of XRSTOR or XRSTORS; and (2) this execution of XSAVES corresponds to that last execution of XRSTOR or XRSTORS as determined by the internal value XRSTOR_INFO (see the Operation section below).
- XSAVEOPT does not modify bytes 511:464 of the legacy region of the XSAVE area (see Section 13.4.1, "Legacy Region of an XSAVE Area" of Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1).
- XSAVEOPT reads the XSTATE_BV field of the XSAVE header (see Section 13.4.2, "XSAVE Header" of Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1) and writes a modified value back to memory as follows. If RFBM[*i*] = 1, XSAVEOPT writes XSTATE_BV[*i*] with the value of XINUSE[*i*]. If RFBM[*i*] = 0, XSAVEOPT writes XSTATE_BV[*i*] with the value that it read from memory (it does not modify the bit). XSAVEOPT does not write to any part of the XSAVE header other than the XSTATE_BV field.
- XSAVEOPT always uses the standard format of the extended region of the XSAVE area (see Section 13.4.3, "Extended Region of an XSAVE Area" of Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1).

Use of a destination operand not aligned to 64-byte boundary (in either 64-bit or 32-bit modes) will result in a general-protection (#GP) exception. In 64-bit mode, the upper 32 bits of RDX and RAX are ignored.

1. There is an exception made for MXCSR and MXCSR_MASK, which belong to state component 1 — SSE. XSAVEOPT always saves these to memory if RFBM[1] = 1 or RFBM[2] = 1, regardless of the value of XINUSE.

See Section 13.6, “Processor Tracking of XSAVE-Managed State,” of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1 for discussion of the bitmap XMODIFIED and of the quantity XRSTOR_INFO.

Operation

```
RFBM := XCR0 AND EDX:EAX;    /* bitwise logical AND */
OLD_BV := XSTATE_BV field from XSAVE header;
TO_BE_SAVED := RFBM AND XINUSE;

IF in VMX non-root operation
    THEN VMXNR := 1;
    ELSE VMXNR := 0;
FI;
LAXA := linear address of XSAVE area;
IF XRSTOR_INFO = 〈CPL,VMXNR,LAXA,00000000_00000000H〉
    THEN TO_BE_SAVED := TO_BE_SAVED AND XMODIFIED;
FI;

IF TO_BE_SAVED[0] = 1
    THEN store x87 state into legacy region of XSAVE area;
FI;

IF TO_BE_SAVED[1]
    THEN store XMM registers into legacy region of XSAVE area; // this step does not save MXCSR or MXCSR_MASK
FI;

IF RFBM[1] = 1 or RFBM[2] = 1
    THEN store MXCSR and MXCSR_MASK into legacy region of XSAVE area;
FI;

FOR i := 2 TO 62
    IF TO_BE_SAVED[i] = 1
        THEN save XSAVE state component i at offset n from base of XSAVE area (n enumerated by CPUID(EAX=0DH,ECX=i):EBX);
    FI;
ENDFOR;

XSTATE_BV field in XSAVE header := (OLD_BV AND NOT RFBM) OR (XINUSE AND RFBM);
```

Flags Affected

None.

Intel C/C++ Compiler Intrinsic Equivalent

```
XSAVEOPT void _xsaveopt( void * , unsigned __int64);
XSAVEOPT void _xsaveopt64( void * , unsigned __int64);
```

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#NM	If CR0.TS[bit 3] = 1.

#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSAVEOPT[bit 0] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If the LOCK prefix is used.
#AC	If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 64-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).

Real-Address Mode Exceptions

#GP	If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSAVEOPT[bit 0] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
	If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
#PF(fault-code)	If a page fault occurs.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSAVEOPT[bit 0] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If the LOCK prefix is used.
#AC	If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 64-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).

XSAVES—Save Processor Extended States Supervisor

Opcode / Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F C7 /5 XSAVES mem	M	V/V	XSS	Save state components specified by EDX:EAX to mem with compaction, optimizing if possible.
NP REX.W + 0F C7 /5 XSAVES64 mem	M	V/N.E.	XSS	Save state components specified by EDX:EAX to mem with compaction, optimizing if possible.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (w)	N/A	N/A	N/A

Description

Performs a full or partial save of processor state components to the XSAVE area located at the memory address specified by the destination operand. The implicit EDX:EAX register pair specifies a 64-bit instruction mask. The specific state components saved correspond to the bits set in the requested-feature bitmap (RFBM), the logical-AND of EDX:EAX and the logical-OR of XCR0 with the IA32_XSS MSR. XSAVES may be executed only if CPL = 0.

The format of the XSAVE area is detailed in Section 13.4, “XSAVE Area,” of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1. Like FXRSTOR and FXSAVE, the memory format used for x87 state depends on a REX.W prefix; see Section 13.5.1, “x87 State” of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1.

Section 13.11, “Operation of XSAVES,” of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1 provides a detailed description of the operation of the XSAVES instruction. The following items provide a high-level outline:

- Execution of XSAVES is similar to that of XSAVEC. XSAVES differs from XSAVEC in that it can save state components corresponding to bits set in the IA32_XSS MSR and that it may use the modified optimization.
- XSAVES saves state component *i* only if RFBM[*i*] = 1 and XINUSE[*i*] = 1.[1] (XINUSE is a bitmap by which the processor tracks the status of various state components. See Section 13.6, “Processor Tracking of XSAVE-Managed State” of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1.) Even if both bits are 1, XSAVES may optimize and not save state component *i* if (1) state component *i* has not been modified since the last execution of XRSTOR or XRSTORS; and (2) this execution of XSAVES correspond to that last execution of XRSTOR or XRSTORS as determined by XRSTOR_INFO (see the Operation section below).
- XSAVES does not modify bytes 511:464 of the legacy region of the XSAVE area (see Section 13.4.1, “Legacy Region of an XSAVE Area” of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1).
- XSAVES writes the logical AND of RFBM and XINUSE to the XSTATE_BV field of the XSAVE header.[2] (See Section 13.4.2, “XSAVE Header” of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1.) XSAVES sets bit 63 of the XCOMP_BV field and sets bits 62:0 of that field to RFBM[62:0]. XSAVES does not write to any parts of the XSAVE header other than the XSTATE_BV and XCOMP_BV fields.
- XSAVES always uses the compacted format of the extended region of the XSAVE area (see Section 13.4.3, “Extended Region of an XSAVE Area” of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1).

Use of a destination operand not aligned to 64-byte boundary (in either 64-bit or 32-bit modes) results in a general-protection (#GP) exception. In 64-bit mode, the upper 32 bits of RDX and RAX are ignored.

1. There is an exception for state component 1 (SSE). MXCSR is part of SSE state, but XINUSE[1] may be 0 even if MXCSR does not have its initial value of 1F80H. In this case, the init optimization does not apply and XSAVEC will save SSE state as long as RFBM[1] = 1 and the modified optimization is not being applied.
2. There is an exception for state component 1 (SSE). MXCSR is part of SSE state, but XINUSE[1] may be 0 even if MXCSR does not have its initial value of 1F80H. In this case, XSAVES sets XSTATE_BV[1] to 1 as long as RFBM[1] = 1.

See Section 13.6, “Processor Tracking of XSAVE-Managed State,” of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1 for discussion of the bitmap XMODIFIED and of the quantity XRSTOR_INFO.

Operation

```
RFBM := (XCR0 OR IA32_XSS) AND EDX:EAX;                /* bitwise logical OR and AND */
IF in VMX non-root operation
    THEN VMXNR := 1;
    ELSE VMXNR := 0;
FI;
LAXA := linear address of XSAVE area;
COMPMASK := RFBM OR 80000000_00000000H;
TO_BE_SAVED := RFBM AND XINUSE;
IF XRSTOR_INFO = 〈CPL,VMXNR,LAXA,COMPMASK〉
    THEN TO_BE_SAVED := TO_BE_SAVED AND XMODIFIED;
FI;
IF MXCSR ≠ 1F80H AND RFBM[1]
    THEN TO_BE_SAVED[1] = 1;
FI;

IF TO_BE_SAVED[0] = 1
    THEN store x87 state into legacy region of XSAVE area;
FI;

IF TO_BE_SAVED[1] = 1
    THEN store SSE state into legacy region of XSAVE area; // this step saves the XMM registers, MXCSR, and MXCSR_MASK
FI;

NEXT_FEATURE_OFFSET = 576;            // Legacy area and XSAVE header consume 576 bytes
FOR i := 2 TO 62
    IF RFBM[i] = 1
        THEN
            IF TO_BE_SAVED[i]
                THEN
                    save XSAVE state component i at offset NEXT_FEATURE_OFFSET from base of XSAVE area;
                    IF i = 8          // state component 8 is for PT state
                        THEN IA32_RTIT_CTL.TraceEn[bit 0] := 0;
                    FI;
            FI;
            NEXT_FEATURE_OFFSET = NEXT_FEATURE_OFFSET + n (n enumerated by CPUID(EAX=0DH,ECX=i):EAX);
    FI;
ENDFOR;

NEW_HEADER := RFBM AND XINUSE;
IF MXCSR ≠ 1F80H AND RFBM[1]
    THEN NEW_HEADER[1] = 1;
FI;
XSTATE_BV field in XSAVE header := NEW_HEADER;
XCOMP_BV field in XSAVE header := COMPMASK;
```

Flags Affected

None.

Intel C/C++ Compiler Intrinsic Equivalent

XSAVES void _xsaves(void * , unsigned __int64);
XSAVES64 void _xsaves64(void * , unsigned __int64);

Protected Mode Exceptions

#GP(0)	If CPL > 0.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSS[bit 3] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSS[bit 3] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0)	If CPL > 0.
	If the memory address is in a non-canonical form.
	If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSS[bit 3] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If the LOCK prefix is used.

XSETBV—Set Extended Control Register

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
NP 0F 01 D1	XSETBV	ZO	Valid	Valid	Write the value in EDX:EAX to the XCR specified by ECX.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	N/A	N/A	N/A	N/A

Description

Writes the contents of registers EDX:EAX into the 64-bit extended control register (XCR) specified in the ECX register. (On processors that support the Intel 64 architecture, the high-order 32 bits of RCX are ignored.) The contents of the EDX register are copied to high-order 32 bits of the selected XCR and the contents of the EAX register are copied to low-order 32 bits of the XCR. (On processors that support the Intel 64 architecture, the high-order 32 bits of each of RAX and RDX are ignored.) Undefined or reserved bits in an XCR should be set to values previously read.

This instruction must be executed at privilege level 0 or in real-address mode; otherwise, a general protection exception #GP(0) is generated. Specifying a reserved or unimplemented XCR in ECX will also cause a general protection exception. The processor will also generate a general protection exception if software attempts to write to reserved bits in an XCR.

Currently, only XCR0 is supported. Thus, all other values of ECX are reserved and will cause a #GP(0). Note that bit 0 of XCR0 (corresponding to x87 state) must be set to 1; the instruction will cause a #GP(0) if an attempt is made to clear this bit. In addition, the instruction causes a #GP(0) if an attempt is made to set XCR0[2] (AVX state) while clearing XCR0[1] (SSE state); it is necessary to set both bits to use AVX instructions; Section 13.3, “Enabling the XSAVE Feature Set and XSAVE-Enabled Features,” of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1.

Operation

```
XCR[ECX] := EDX:EAX;
```

Flags Affected

None.

Intel C/C++ Compiler Intrinsic Equivalent

XSETBV void _xsetbv(unsigned int, unsigned __int64);

Protected Mode Exceptions

#GP(0)	If the current privilege level is not 0.
	If an invalid XCR is specified in ECX.
	If the value in EDX:EAX sets bits that are reserved in the XCR specified by ECX.
	If an attempt is made to clear bit 0 of XCR0.
	If an attempt is made to set XCR0[2:1] to 10b.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If an invalid XCR is specified in ECX.
	If the value in EDX:EAX sets bits that are reserved in the XCR specified by ECX.
	If an attempt is made to clear bit 0 of XCR0.
	If an attempt is made to set XCR0[2:1] to 10b.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	The XSETBV instruction is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

XSUSLDTRK—Suspend Tracking Load Addresses

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 01 E8 XSUSLDTRK	ZO	V/V	TSXLDTRK	Specifies the start of an Intel TSX suspend read address tracking region.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
ZO	N/A	N/A	N/A	N/A	N/A

Description

The instruction marks the start of an Intel TSX (RTM) suspend load address tracking region. If the instruction is used inside a transactional region, subsequent loads are not added to the read set of the transaction. If the instruction is used inside a suspend load address tracking region it will cause transaction abort.

If the instruction is used outside of a transactional region it behaves like a NOP.

Chapter 16, “Programming with Intel® Transactional Synchronization Extensions” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1 provides additional information on Intel® TSX Suspend Load Address Tracking.

Operation

XSUSLDTRK

```
IF RTM_ACTIVE = 1:
    IF SUSLDTRK_ACTIVE = 0:
        SUSLDTRK_ACTIVE := 1
    ELSE:
        RTM_ABORT
ELSE:
    NOP
```

Flags Affected

None.

Intel C/C++ Compiler Intrinsic Equivalent

XSUSLDTRK void _xsusldtrk(void);

SIMD Floating-Point Exceptions

None.

Other Exceptions

#UD	If CPUID.(EAX=7, ECX=0):EDX.TSXLDTRK[bit 16] = 0. If the LOCK prefix is used.

XTEST—Test if in Transactional Execution

Opcode/Instruction	Op/ En	64/32bit Mode Support	CPUID Feature Flag	Description
NP 0F 01 D6 XTEST	ZO	V/V	HLE or RTM	Test if executing in a transactional region.

Instruction Operand Encoding

Op/En	Operand 1	Operand2	Operand3	Operand4
ZO	N/A	N/A	N/A	N/A

Description

The XTEST instruction queries the transactional execution status. If the instruction executes inside a transactionally executing RTM region or a transactionally executing HLE region, then the ZF flag is cleared, else it is set.

Operation

XTEST

```
IF (RTM_ACTIVE = 1 OR HLE_ACTIVE = 1)
    THEN
        ZF := 0
    ELSE
        ZF := 1
FI;
```

Flags Affected

The ZF flag is cleared if the instruction is executed transactionally; otherwise it is set to 1. The CF, OF, SF, PF, and AF, flags are cleared.

Intel C/C++ Compiler Intrinsic Equivalent

XTEST int _xtest(void);

SIMD Floating-Point Exceptions

None.

Other Exceptions

#UD	CPUID.(EAX=7, ECX=0):EBX.HLE[bit 4] = 0 and CPUID.(EAX=7, ECX=0):EBX.RTM[bit 11] = 0.
	If LOCK prefix is used.

CHAPTER 7
SAFER MODE EXTENSIONS REFERENCE

7.1 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)[1]. The system executive may be based on a Virtual Machine Monitor (VMM), a measured VMM is referred to as MVMM[2].
- 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.

7.2 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).

7.2.1 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 7-1.

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

Table 7-1. 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].
16	Reserved
17	SGX Launch Control Enable: Must be set to '1' to enable runtime re-configuration of SGX Launch Control via the IA32_SGXLEPUBKEYHASHn MSR.
18	SGX Global Enable: Must be set to '1' to enable Intel SGX leaf functions.
19	Reserved
20	LMCE On: When set, system software can program the MSRs associated with LMCE to configure delivery of some machine check exceptions to a single logical processor.
63:21	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 7, "Safer Mode Extensions 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.

7.2.2 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.

7.2.2.1 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 7-2 lists the currently available GETSEC leaf functions.

Table 7-2. 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	SENTER	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

7.2.2.2 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.

7.2.2.3 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.

7.2.2.4 GETSEC[SENTER]

The GETSEC[SENTER] leaf function is used by the initiating logical processor (ILP) to launch an MLE. GETSEC[SENTER] 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[SENTER] 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[SENTER]. The AC module responsible for the launch of a measured environment and loaded by GETSEC[SENTER] 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[SENTER].

7.2.2.5 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.

7.2.2.6 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.

7.2.2.7 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.

7.2.2.8 GETSEC[WAKEUP]

Responding logical processors (RLPs) are placed in the SENTER sleep state after the initiating logical processor executes GETSEC[SENTER]. 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 an chipset register LT.MLE.JOIN) in TXT configuration space.

7.2.3 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. The 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[SENTER] 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[SENTER].

After the GETSEC[SENTER] 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 wake up 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 configuration 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.

7.3 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[1] passed as input parameters to GETSEC functions and to register results returned as output parameters.

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 MVMM 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.

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[1] 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.

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[SENTER] or GTSEC[SEXIT] is transparent to the retired instruction count on the ILP.

1. 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.

GETSEC[CAPABILITIES]—Report the SMX Capabilities

Opcode	Instruction	Description
NP 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 7-3.

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 7-3. 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.
SENTER	4	GETSEC[SENTER] is available.
SEXIT	5	GETSEC[SEXIT] is available.
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 (SENTER Available)
            THEN BitVector[SENTER] := 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.
NP	66/F2/F3 prefixes are not allowed.
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
NP 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[SENTER] 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 ensure consistent handling of SIPI messages, the processor executing the GETSEC[ENTERACCS] instruction must also be designated the BSP (boot-strap processor) as defined by IA32_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[SENTER] 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[SENTER]). 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[SENTER], 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 BASE=0, LIMIT=FFFFFh, G=1, D=1, P=1, 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 7-4. 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 7-4. Register State Initialization After GETSEC[ENTERACCS]

Register State	Initialization Status	Comment
CR0	PG←0, AM←0, WP←0: Others unchanged	Paging, Alignment Check, Write-protection are disabled.
CR4	MCE←0, CET←0, PCIDE←0: Others unchanged	Machine Check Exceptions, Control-flow Enforcement Technology, and Process-context Identifiers 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	
GDTR	Base= AC.base (EBX) + [GDTBasePtr], Limit=[GDTLimit]	
DR7	00000400H	
IA32_DEBUGCTL	0H	
IA32_MISC_ENABLE	See Table 7-5 for example.	The number of initialized fields may change due to processor implementation.
Performance counters and counter control registers	0H	

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.

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

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 7-5.). 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 7-5. IA32_MISC_ENABLE MSR Initialization[1] 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.[2]
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_MC[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].RPL≠0))
    THEN TXT-SHUTDOWN(#BadACMFormat);
CR0.[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 := FFFFFh;
CS.G := 1;
CS.D := 1;
CS.AR := 9Bh;
DS.SEL := ACRAM[SegSel]+8;
```

```
DS.BASE := 0;
DS.LIMIT := FFFFFh;
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.
NP	66/F2/F3 prefixes are not allowed.
Segment overrides	Ignored.
Address size	Ignored.
REX	Ignored.

Protected Mode Exceptions

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

Real-Address Mode Exceptions

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

Virtual-8086 Mode Exceptions

#UD If CR4.SMXE = 0.

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

#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
NP 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[SENTER]. 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 unmasks 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[SENTER] or GETSEC[ENTERACCS]. If the processor is in authenticated code execution mode due to the execution of GETSEC[SENTER], 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 3C. 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;
IF IA32_EFER.LMA == 1
    THEN CR3 := R8;
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.
NP	66/F2/F3 prefixes are not allowed.
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].
#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[SENTER]—Enter a Measured Environment

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

Description

The GETSEC[SENTER] instruction initiates the launch of a measured environment and places the initiating logical processor (ILP) into the authenticated code execution mode. The SENTER 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[SENTER] 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[SENTER] instruction.

There are restrictions enforced by the processor for execution of the GETSEC[SENTER] 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[SENTER] 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 ensure consistent handling of SIPI messages, the processor executing the GETSEC[SENTER] instruction must also be designated the BSP (boot-strap processor) as defined by IA32_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[SENTER]) 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 3C 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[SENTER] 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.ERRORCODE 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 a trusted storage facility in the platform. The following trusted storage facility are supported:

- If the platform register FTM_INTERFACE_ID.[bits 3:0] = 0, the computed hash is stored to the platform's 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[SENTER]. 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.
- If the platform register FTM_INTERFACE_ID.[bits 3:0] = 1, the computed hash is stored in a firmware trusted module (FTM) using a modified protocol similar to the protocol used to write to TPM's PCR17.

After successful execution of SENTER, either PCR17 (if FTM is not enabled) or the FTM (if enabled) 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 7-6 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 7-6. Register State Initialization After GETSEC[SENTER] and GETSEC[WAKEUP]

Register State	ILP after GETSEC[SENTER]	RLP after GETSEC[WAKEUP]
CR0	PG←0, AM←0, WP←0; Others unchanged	PG←0, CD←0, NW←0, AM←0, WP←0; PE←1, NE←1
CR4	00004000H	00004000H
EFLAGS	00000002H	00000002H
IA32_EFER	0H	0
EIP	[EntryPoint from MLE header[1]]	[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 = FFFFFH, 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 = FFFFFH, G = 1, D = 1, AR = 93H
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 7-5	See Table 7-5
IA32_SMM_MONITOR_CTL	Bit 2←0	Bit 2←0

NOTES:

1. See the 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 7-6. 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 7-5 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 7-1.

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
    (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
    (SENTERFLAG=1) or (ACMODEFLAG=1) or (IN_SMM=1) or
    (TPM interface is not present) or
    (EDX ≠ (SENTER_EDX_support_mask & EDX)) or
    (IA32_FEATURE_CONTROL[0]=0) or (IA32_FEATURE_CONTROL[15]=0) or
    ((IA32_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;
SENTERFLAG := 1;
SignalTXTMsg(SENTERAck);
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];
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[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);

IF (FTM_INTERFACE_ID.[3:0] = 1 ) (* Alternate FTM Interface has been enabled *)
    THEN (* TPM_LOC_CTRL_4 is located at 0FED44008H, TMP_DATA_BUFFER_4 is located at 0FED44080H *)
        WRITE(TPM_LOC_CTRL_4) := 01H; (* Modified HASH.START protocol *)
        (* Write to firmware storage *)
        WRITE(TPM_DATA_BUFFER_4) := SIGNATURE_LEN_CONST + 4;
        FOR I=0 to SIGNATURE_LEN_CONST - 1 DO
            WRITE(TPM_DATA_BUFFER_4 + 2 + I ) := ACRAM[SCRATCH.I];
        WRITE(TPM_DATA_BUFFER_4 + 2 + SIGNATURE_LEN_CONST) := EDX;
        WRITE(FTM.LOC_CTRL) := 06H; (* Modified protocol combining HASH.DATA and HASH.END *)
    ELSE IF (FTM_INTERFACE_ID.[3:0] = 0 ) (* Use standard TPM Interface *)
        ACRAM[SCRATCH.SIGNATURE_LEN_CONST] := EDX;
        WRITE(TPM.HASH.START) := 0;
        FOR I=0 to SIGNATURE_LEN_CONST + 3 DO
            WRITE(TPM.HASH.DATA) := ACRAM[SCRATCH.I];
        WRITE(TPM.HASH.END) := 0;
FI;
```

```
ACMODEFLAG := 1;
CR0.[PG.AM.WP] := 0;
CR4 := 00004000h;
EFLAGS := 00000002h;
IA32_EFER := 0;
EBP := ACBASE;
GDTR.BASE := ACBASE+ACRAM[GDTBasePtr];
GDTR.LIMIT := ACRAM[GDTLimit];
CS.SEL := ACRAM[SegSel];
CS.BASE := 0;
CS.LIMIT := FFFFFh;
CS.G := 1;
CS.D := 1;
CS.AR := 9Bh;
DS.SEL := ACRAM[SegSel]+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;
SignalTXTMsg(UnlockSMRAM);
SignalTXTMsg(OpenPrivate);
SignalTXTMsg(OpenLocality3);
EIP := ACEntryPoint;
END;
```

RLP_SENTER_ROUTINE: (RLP 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.
NP	66/F2/F3 prefixes are not allowed.
Segment overrides	Ignored.
Address size	Ignored.
REX	Ignored.

Protected Mode Exceptions

#UD	If CR4.SMXE = 0.
	If GETSEC[SENTER] is not reported as supported by GETSEC[CAPABILITIES].
#GP(0)	If CR0.CD = 1 or CR0.NW = 1 or CR0.NE = 0 or CR0.PE = 0 or CPL > 0 or EFLAGS.VM = 1.
	If in VMX root operation.
	If the initiating processor is not designated as the bootstrap processor via the MSR bit IA32_APIC_BASE.BSP.
	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[I]_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
NP 0F 37 (EAX=5)	GETSEC[SEXIT]	Exit measured environment.

Description

The GETSEC[SEXIT] instruction initiates an exit of a measured environment established by GETSEC[SENTER]. 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[SENTER] instruction, but not still in authenticated code execution mode.
- To avoid potential interoperability conflicts between modes, the processor is not allowed to execute this instruction if it currently is in SMM or in VMX operation.
- To ensure 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[SENTER]. 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
    (SENTERFLAG=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 SignalSENTER 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);
SENTERFLAG := 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.
NP	66/F2/F3 prefixes are not allowed.
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 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
NP 0F 37 (EAX=6)	GETSEC[PARAMETERS]	Report the SMX parameters. The parameters index is input in EBX with the result returned in EAX, EBX, and ECX.

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

Table 7-7. 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)
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)	Reserved	Reserved
6-31	Undefined	Reserved (unmodified)	Reserved (unmodified)	Reserved (unmodified)

Table 7-8. 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 7-9.

The parameter type of 4 is used for enumerating the availability of selective GETSEC[SENTER] 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 SENTER. If no selective disable capability for SENTER exists as enumerated, then the corresponding bits in the IA32_FEATURE_CONTROL MSR bits 14:8 must also be programmed to 1 if the SENTER global enable bit 15 of the MSR is set. This is required to enable future extensibility of SENTER selective disable capability with respect to potentially separate software initialization of the MSR.

Table 7-9. 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 7-10.

Table 7-10. 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 SENTER 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;
ESE IF (EBX= 2)
    (* example of a processor supporting external memory types of UC and WC *)
    THEN EAX := 00000303h;
ESE IF (EBX= other value(s) less than unsupported index value)
    (* EAX value varies. Consult Table 7-7 and Table *)
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.
NP	66/F2/F3 prefixes are not allowed.
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
NP 0F 37 (EAX = 7)	GETSEC[SMCTRL]	Perform specified SMX mode control as selected with the input EBX.

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[SENTER] 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 7-11). 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[SENTER], 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 32, “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 7-11 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 7-11. 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/REPNZ and REP/REPE/REPZ).
Operand size	Causes #UD.
NP	66/F2/F3 prefixes are not allowed.
Segment overrides	Ignored.
Address size	Ignored.
REX	Ignored.

Protected Mode Exceptions

#UD	If CR4.SMXE = 0.
	If GETSEC[SMCTRL] 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 a protected partition is not already active or the processor is currently in authenticated code mode.
	If the processor is in SMM.
	If the SMM monitor is not configured.

Real-Address Mode Exceptions

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

Virtual-8086 Mode Exceptions

#UD	If CR4.SMXE = 0.
	If GETSEC[SMCTRL] is not reported as supported by GETSEC[CAPABILITIES].
#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
NP 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 7-12.

Table 7-12. 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 7-6). Certain states from the content of Table 7-12 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 7-12).

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_SIPI_WAKEUP_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;
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);
CR0.[PG,CD,NW,AM,WP] := 0;
CR0.[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.
NP	66/F2/F3 prefixes are not allowed.
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.

CHAPTER 8
INSTRUCTION SET REFERENCE UNIQUE TO INTEL® XEON PHI™ PROCESSORS

This chapter describes the instruction set that is unique to Intel® Xeon Phi™ Processors based on the Knights Landing and Knights Mill microarchitectures. The set is not supported in any other Intel processors. Included are Intel® AVX-512 instructions. For additional instructions supported on these processors, see Chapter 3, “Instruction Set Reference, A-L”; Chapter 4, “Instruction Set Reference, M-U”; Chapter 5, “Instruction Set Reference, V”; and Chapter 6, “Instruction Set Reference, W-Z”.

PREFETCHWT1—Prefetch Vector Data Into Caches With Intent to Write and T1 Hint

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F 0D /2 PREFETCHWT1 m8	M	V/V	PREFETCHWT1	Move data from m8 closer to the processor using T1 hint with intent to write.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r)	N/A	N/A	N/A

Description

Fetches the line of data from memory that contains the byte specified with the source operand to a location in the cache hierarchy specified by an intent to write hint (so that data is brought into 'Exclusive' state via a request for ownership) and a locality hint:

- T1 (temporal data with respect to first level cache)—prefetch data into the second level cache.

The source operand is a byte memory location. (The locality hints are encoded into the machine level instruction using bits 3 through 5 of the ModR/M byte. Use of any ModR/M value other than the specified ones will lead to unpredictable behavior.)

If the line selected is already present in the cache hierarchy at a level closer to the processor, no data movement occurs. Prefetches from uncacheable or WC memory are ignored.

The PREFETCHWT1 instruction is merely a hint and does not affect program behavior. If executed, this instruction moves data closer to the processor in anticipation of future use.

The implementation of prefetch locality hints is implementation-dependent, and can be overloaded or ignored by a processor implementation. The amount of data prefetched is also processor implementation-dependent. It will, however, be a minimum of 32 bytes. Additional details of the implementation-dependent locality hints are described in Section 9.5, "Memory Optimization Using Prefetch" of the Intel® 64 and IA-32 Architectures Optimization Reference Manual.

It should be noted that processors are free to speculatively fetch and cache data from system memory regions that are assigned a memory-type that permits speculative reads (that is, the WB, WC, and WT memory types). A PREFETCHWT1 instruction is considered a hint to this speculative behavior. Because this speculative fetching can occur at any time and is not tied to instruction execution, a PREFETCHWT1 instruction is not ordered with respect to the fence instructions (MFENCE, SFENCE, and LFENCE) or locked memory references. A PREFETCHWT1 instruction is also unordered with respect to CLFLUSH and CLFLUSHOPT instructions, other PREFETCHWT1 instructions, or any other general instruction. It is ordered with respect to serializing instructions such as CPUID, WRMSR, OUT, and MOV CR.

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

Operation

PREFETCH(mem, Level, State) Prefetches a byte memory location pointed by 'mem' into the cache level specified by 'Level'; a request for exclusive/ownership is done if 'State' is 1. Note that the memory location ignore cache line splits. This operation is considered a hint for the processor and may be skipped depending on implementation.

```
Prefetch (m8, Level = 1, EXCLUSIVE=1);
```

Flags Affected

All flags are affected.

C/C++ Compiler Intrinsic Equivalent

```
void _mm_prefetch( char const *, int hint= _MM_HINT_ET1);
```

Protected Mode Exceptions

#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

#UD	If the LOCK prefix is used.

64-Bit Mode Exceptions

#UD	If the LOCK prefix is used.

V4FMADDPS/V4FNMADDPS—Packed Single Precision Floating-Point Fused Multiply-Add (4-Iterations)

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.512.F2.0F38.W0 9A /r V4FMADDPS zmm1{k1}{z}, zmm2+3, m128	A	V/V	AVX512_4FMAPS	Multiply packed single-precision floating-point values from source register block indicated by zmm2 by values from m128 and accumulate the result in zmm1.
EVEX.512.F2.0F38.W0 AA /r V4FNMADDPS zmm1{k1}{z}, zmm2+3, m128	A	V/V	AVX512_4FMAPS	Multiply and negate packed single-precision floating-point values from source register block indicated by zmm2 by values from m128 and accumulate the result in zmm1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1_4X	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction computes 4 sequential packed fused single-precision floating-point multiply-add instructions with a sequentially selected memory operand in each of the four steps.

In the above box, the notation of “+3” is used to denote that the instruction accesses 4 source registers based on that operand; sources are consecutive, start in a multiple of 4 boundary, and contain the encoded register operand.

This instruction supports memory fault suppression. The entire memory operand is loaded if any of the 16 lowest significant mask bits is set to 1 or if a “no masking” encoding is used.

The tuple type Tuple1_4X implies that four 32-bit elements (16 bytes) are referenced by the memory operation portion of this instruction.

Rounding is performed at every FMA (fused multiply and add) boundary. Exceptions are also taken sequentially. Pre- and post-computational exceptions of the first FMA take priority over the pre- and post-computational exceptions of the second FMA, etc.

Operation

src_reg_id is the 5 bit index of the vector register specified in the instruction as the src1 register.

```
define NFMA_PS(kl, vl, dest, k1, msrc, regs_loaded, src_base, posneg):
    tmpdest := dest

    // reg[] is an array representing the SIMD register file.
    FOR j := 0 to regs_loaded-1:
        FOR i := 0 to kl-1:
            IF k1[i] or *no writemask*:
                IF posneg = 0:
                    tmpdest.single[i] := RoundFPControl_MXCSR(tmpdest.single[i] - reg[src_base + j ].single[i] * msrc.single[j])
                ELSE:
                    tmpdest.single[i] := RoundFPControl_MXCSR(tmpdest.single[i] + reg[src_base + j ].single[i] * msrc.single[j])
            ELSE IF *zeroing*:
                tmpdest.single[i] := 0
    dest := tmpdst
    dest[MAX_VL-1:VL] := 0

V4FMADDPS and V4FNMADDPS dest{k1}, src1, msrc (AVX512)
KL, VL = (16,512)

regs_loaded := 4
src_base := src_reg_id & ~3 // for src1 operand
posneg := 0 if negative form, 1 otherwise
NFMA_PS(kl, vl, dest, k1, msrc, regs_loaded, src_base, posneg)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
V4FMADDPS __m512 _mm512_4fmadd_ps( __m512, __m512x4, __m128 *);
V4FMADDPS __m512 _mm512_mask_4fmadd_ps(__m512, __mmask16, __m512x4, __m128 *);
V4FMADDPS __m512 _mm512_maskz_4fmadd_ps(__mmask16, __m512, __m512x4, __m128 *);
V4FNMADDPS __m512 _mm512_4fnmadd_ps(__m512, __m512x4, __m128 *);
V4FNMADDPS __m512 _mm512_mask_4fnmadd_ps(__m512, __mmask16, __m512x4, __m128 *);
V4FNMADDPS __m512 _mm512_maskz_4fnmadd_ps(__mmask16, __m512, __m512x4, __m128 *);
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

See Type E2; additionally:

#UD	If the EVEX broadcast bit is set to 1.
#UD	If the MODRM.mod = 0b11.

V4FMADDSS/V4FNMADDSS—Scalar Single Precision Floating-Point Fused Multiply-Add (4-Iterations)

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.F2.0F38.W0 9B /r V4FMADDSS xmm1{k1}{z}, xmm2+3, m128	A	V/V	AVX512_4FMAPS	Multiply scalar single-precision floating-point values from source register block indicated by xmm2 by values from m128 and accumulate the result in xmm1.
EVEX.LLIG.F2.0F38.W0 AB /r V4FNMADDSS xmm1{k1}{z}, xmm2+3, m128	A	V/V	AVX512_4FMAPS	Multiply and negate scalar single-precision floating-point values from source register block indicated by xmm2 by values from m128 and accumulate the result in xmm1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1_4X	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction computes 4 sequential scalar fused single-precision floating-point multiply-add instructions with a sequentially selected memory operand in each of the four steps.

In the above box, the notation of "+3" is used to denote that the instruction accesses 4 source registers based that operand; sources are consecutive, start in a multiple of 4 boundary, and contain the encoded register operand.

This instruction supports memory fault suppression. The entire memory operand is loaded if the least significant mask bit is set to 1 or if a "no masking" encoding is used.

The tuple type Tuple1_4X implies that four 32-bit elements (16 bytes) are referenced by the memory operation portion of this instruction.

Rounding is performed at every FMA boundary. Exceptions are also taken sequentially. Pre- and post-computational exceptions of the first FMA take priority over the pre- and post-computational exceptions of the second FMA, etc.

Operation

src_reg_id is the 5 bit index of the vector register specified in the instruction as the src1 register.

```
define NFMA_SS(vl, dest, k1, msrc, regs_loaded, src_base, posneg):
    tmpdest := dest
    // reg[] is an array representing the SIMD register file.
    IF k1[0] or *no writemask*:
        FOR j := 0 to regs_loaded - 1:
            IF posneg = 0:
                tmpdest.single[0] := RoundFPControl_MXCSR(tmpdest.single[0] - reg[src_base + j ].single[0] * msrc.single[j])
            ELSE:
                tmpdest.single[0] := RoundFPControl_MXCSR(tmpdest.single[0] + reg[src_base + j ].single[0] * msrc.single[j])
    ELSE IF *zeroing*:
        tmpdest.single[0] := 0
    dest := tmpdst
    dest[MAX_VL-1:VL] := 0
```

V4FMADDSS and V4FNMADDSS dest{k1}, src1, msrc (AVX512)

VL = 128

regs_loaded := 4
src_base := src_reg_id & ~3 // for src1 operand
posneg := 0 if negative form, 1 otherwise
NFMA_SS(vl, dest, k1, msrc, regs_loaded, src_base, posneg)

Intel C/C++ Compiler Intrinsic Equivalent

V4FMADDSS __m128 _mm_4fmadd_ss(__m128, __m128x4, __m128 *);
V4FMADDSS __m128 _mm_mask_4fmadd_ss(__m128, __mmask8, __m128x4, __m128 *);
V4FMADDSS __m128 _mm_maskz_4fmadd_ss(__mmask8, __m128, __m128x4, __m128 *);
V4FNMADDSS __m128 _mm_4fnmadd_ss(__m128, __m128x4, __m128 *);
V4FNMADDSS __m128 _mm_mask_4fnmadd_ss(__m128, __mmask8, __m128x4, __m128 *);
V4FNMADDSS __m128 _mm_maskz_4fnmadd_ss(__mmask8, __m128, __m128x4, __m128 *);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

See Type E2; additionally:

#UD	If the EVEX broadcast bit is set to 1.
#UD	If the MODRM.mod = 0b11.

VEXP2PD—Approximation to the Exponential 2^x of Packed Double Precision Floating-Point Values With Less Than 2^-23 Relative Error

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.512.66.0F38.W1 C8 /r VEXP2PD zmm1 {k1}{z}, zmm2/m512/m64bcst {sae}	A	V/V	AVX512ER	Computes approximations to the exponential 2^x (with less than 2^-23 of maximum relative error) of the packed double precision floating-point values from zmm2/m512/m64bcst and stores the floating-point result in zmm1with writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A

Description

Computes the approximate base-2 exponential evaluation of the double precision floating-point values in the source operand (the second operand) and stores the results to the destination operand (the first operand) using the writemask k1. The approximate base-2 exponential is evaluated with less than 2^-23 of relative error.

Denormal input values are treated as zeros and do not signal #DE, irrespective of MXCSR.DAZ. Denormal results are flushed to zeros and do not signal #UE, irrespective of MXCSR.FTZ.

The source operand is a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

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

A numerically exact implementation of VEXP2xx can be found at https://software.intel.com/en-us/articles/reference-implementations-for-IA-approximation-instructions-vrcp14-vrsqrt14-vrcp28-vrsqrt28-vexp2.

Operation

VEXP2PD

```
(KL, VL) = (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC *is memory*)
                THEN DEST[i+63:i] := EXP2_23_DP(SRC[63:0])
                ELSE DEST[i+63:i] := EXP2_23_DP(SRC[i+63:i])
            FI;
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[i+63:i] remains unchanged*
            ELSE                            ; zeroing-masking
                DEST[i+63:i] := 0
        FI;
    FI;
ENDFOR;
```

Table 8-1. Special Values Behavior

Source Input	Result	Comments
NaN	QNaN(src)	If (SRC = SNaN) then #I
+∞	+∞	
+/-0	1.0f	Exact result
-∞	+0.0f	
Integral value N	2^ (N)	Exact result

Intel C/C++ Compiler Intrinsic Equivalent

VEXP2PD __m512d _mm512_exp2a23_round_pd (__m512d a, int sae);
VEXP2PD __m512d _mm512_mask_exp2a23_round_pd (__m512d a, __mmask8 m, __m512d b, int sae);
VEXP2PD __m512d _mm512_maskz_exp2a23_round_pd (__mmask8 m, __m512d b, int sae);

SIMD Floating-Point Exceptions

Invalid (if SNaN input), Overflow.

Other Exceptions

See Table 2-46, “Type E2 Class Exception Conditions.”

VEXP2PS—Approximation to the Exponential 2^x of Packed Single Precision Floating-Point Values With Less Than 2^-23 Relative Error

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.512.66.0F38.W0 C8 /r VEXP2PS zmm1 {k1}{z}, zmm2/m512/m32bcst {sae}	A	V/V	AVX512ER	Computes approximations to the exponential 2^x (with less than 2^-23 of maximum relative error) of the packed single-precision floating-point values from zmm2/m512/m32bcst and stores the floating-point result in zmm1with writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (r, w)	ModRM:r/m (r)	N/A	N/A

Description

Computes the approximate base-2 exponential evaluation of the single-precision floating-point values in the source operand (the second operand) and store the results in the destination operand (the first operand) using the write-mask k1. The approximate base-2 exponential is evaluated with less than 2^-23 of relative error.

Denormal input values are treated as zeros and do not signal #DE, irrespective of MXCSR.DAZ. Denormal results are flushed to zeros and do not signal #UE, irrespective of MXCSR.FTZ.

The source operand is a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

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

A numerically exact implementation of VEXP2xx can be found at https://software.intel.com/en-us/articles/reference-implementations-for-IA-approximation-instructions-vrcp14-vrsqrt14-vrcp28-vrsqrt28-vexp2.

Operation

VEXP2PS

```
(KL, VL) = (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC *is memory*)
                THEN DEST[i+31:i] := EXP2_23_SP(SRC[31:0])
                ELSE DEST[i+31:i] := EXP2_23_SP(SRC[i+31:i])
            FI;
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[i+31:i] remains unchanged*
            ELSE                            ; zeroing-masking
                DEST[i+31:i] := 0
        FI;
    FI;
ENDFOR;
```

Table 8-2. Special Values Behavior

Source Input	Result	Comments
NaN	QNaN(src)	If (SRC = SNaN) then #I
+∞	+∞	
+/-0	1.0f	Exact result
-∞	+0.0f	
Integral value N	2^ (N)	Exact result

Intel C/C++ Compiler Intrinsic Equivalent

VEXP2PS __m512 _mm512_exp2a23_round_ps (__m512 a, int sae);
VEXP2PS __m512 _mm512_mask_exp2a23_round_ps (__m512 a, __mmask16 m, __m512 b, int sae);
VEXP2PS __m512 _mm512_maskz_exp2a23_round_ps (__mmask16 m, __m512 b, int sae);

SIMD Floating-Point Exceptions

Invalid (if SNaN input), Overflow.

Other Exceptions

See Table 2-46, “Type E2 Class Exception Conditions.”

VGATHERPF0DPS/VGATHERPF0QPS/VGATHERPF0DPD/VGATHERPF0QPD—Sparse Prefetch Packed SP/DP Data Values With Signed Dword, Signed Qword Indices Using T0 Hint

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.512.66.0F38.W0 C6 /1 /vsib VGATHERPF0DPS vm32z {k1}	A	V/V	AVX512PF	Using signed dword indices, prefetch sparse byte memory locations containing single-precision data using opmask k1 and T0 hint.
EVEX.512.66.0F38.W0 C7 /1 /vsib VGATHERPF0QPS vm64z {k1}	A	V/V	AVX512PF	Using signed qword indices, prefetch sparse byte memory locations containing single-precision data using opmask k1 and T0 hint.
EVEX.512.66.0F38.W1 C6 /1 /vsib VGATHERPF0DPD vm32y {k1}	A	V/V	AVX512PF	Using signed dword indices, prefetch sparse byte memory locations containing double precision data using opmask k1 and T0 hint.
EVEX.512.66.0F38.W1 C7 /1 /vsib VGATHERPF0QPD vm64z {k1}	A	V/V	AVX512PF	Using signed qword indices, prefetch sparse byte memory locations containing double precision data using opmask k1 and T0 hint.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	BaseReg (R): VSIB:base, VectorReg(R): VSIB:index	N/A	N/A	N/A

Description

The instruction conditionally prefetches up to sixteen 32-bit or eight 64-bit integer byte data elements. The elements are specified via the VSIB (i.e., the index register is an zmm, holding packed indices). Elements will only be prefetched if their corresponding mask bit is one.

Lines prefetched are loaded into to a location in the cache hierarchy specified by a locality hint (T0):

- T0 (temporal data)—prefetch data into the first level cache.

[PS data] For dword indices, the instruction will prefetch sixteen memory locations. For qword indices, the instruction will prefetch eight values.

[PD data] For dword and qword indices, the instruction will prefetch eight memory locations.

Note that:

(1) The prefetches may happen in any order (or not at all). The instruction is a hint.

(2) The mask is left unchanged.

(3) Not valid with 16-bit effective addresses. Will deliver a #UD fault.

(4) No FP nor memory faults may be produced by this instruction.

(5) Prefetches do not handle cache line splits

(6) A #UD is signaled if the memory operand is encoded without the SIB byte.

Operation

BASE_ADDR stands for the memory operand base address (a GPR); may not exist.
VINDEX stands for the memory operand vector of indices (a vector register).
SCALE stands for the memory operand scalar (1, 2, 4 or 8).
DISP is the optional 1, 2 or 4 byte displacement.
PREFETCH(mem, Level, State) Prefetches a byte memory location pointed by 'mem' into the cache level specified by 'Level'; a request for exclusive/ownership is done if 'State' is 1. Note that the memory location ignore cache line splits. This operation is considered a hint for the processor and may be skipped depending on implementation.

VGATHERPF0DPS (EVEX Encoded Version)

```
(KL, VL) = (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j]
         Prefetch( [BASE_ADDR + SignExtend(VINDEX[i+31:i]) * SCALE + DISP], Level=0, RFO = 0)
    FI;
ENDFOR
```

VGATHERPF0DPD (EVEX Encoded Version)

```
(KL, VL) = (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    k := j * 32
    IF k1[j]
         Prefetch( [BASE_ADDR + SignExtend(VINDEX[k+31:k]) * SCALE + DISP], Level=0, RFO = 0)
    FI;
ENDFOR
```

VGATHERPF0QPS (EVEX Encoded Version)

```
(KL, VL) = (8, 256)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j]
         Prefetch( [BASE_ADDR + SignExtend(VINDEX[i+63:i]) * SCALE + DISP], Level=0, RFO = 0)
    FI;
ENDFOR
```

VGATHERPF0QPD (EVEX Encoded Version)

```
(KL, VL) = (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    k := j * 64
    IF k1[j]
         Prefetch( [BASE_ADDR + SignExtend(VINDEX[k+63:k]) * SCALE + DISP], Level=0, RFO = 0)
    FI;
ENDFOR
```

Intel C/C++ Compiler Intrinsic Equivalent

VGATHERPF0DPD void _mm512_mask_prefetch_i32gather_pd(__m256i vdx, __mmask8 m, void * base, int scale, int hint);
VGATHERPF0DPS void _mm512_mask_prefetch_i32gather_ps(__m512i vdx, __mmask16 m, void * base, int scale, int hint);
VGATHERPF0QPD void _mm512_mask_prefetch_i64gather_pd(__m512i vdx, __mmask8 m, void * base, int scale, int hint);
VGATHERPF0QPS void _mm512_mask_prefetch_i64gather_ps(__m512i vdx, __mmask8 m, void * base, int scale, int hint);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-62, “Type E12NP Class Exception Conditions.”

VGATHERPF1DPS/VGATHERPF1QPS/VGATHERPF1DPD/VGATHERPF1QPD—Sparse Prefetch Packed SP/DP Data Values With Signed Dword, Signed Qword Indices Using T1 Hint

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.512.66.0F38.W0 C6 /2 /vsib VGATHERPF1DPS vm32z {k1}	A	V/V	AVX512PF	Using signed dword indices, prefetch sparse byte memory locations containing single-precision data using opmask k1 and T1 hint.
EVEX.512.66.0F38.W0 C7 /2 /vsib VGATHERPF1QPS vm64z {k1}	A	V/V	AVX512PF	Using signed qword indices, prefetch sparse byte memory locations containing single-precision data using opmask k1 and T1 hint.
EVEX.512.66.0F38.W1 C6 /2 /vsib VGATHERPF1DPD vm32y {k1}	A	V/V	AVX512PF	Using signed dword indices, prefetch sparse byte memory locations containing double precision data using opmask k1 and T1 hint.
EVEX.512.66.0F38.W1 C7 /2 /vsib VGATHERPF1QPD vm64z {k1}	A	V/V	AVX512PF	Using signed qword indices, prefetch sparse byte memory locations containing double precision data using opmask k1 and T1 hint.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	BaseReg (R): VSIB:base, VectorReg(R): VSIB:index	N/A	N/A	N/A

Description

The instruction conditionally prefetches up to sixteen 32-bit or eight 64-bit integer byte data elements. The elements are specified via the VSIB (i.e., the index register is an zmm, holding packed indices). Elements will only be prefetched if their corresponding mask bit is one.

Lines prefetched are loaded into to a location in the cache hierarchy specified by a locality hint (T1):

• T1 (temporal data)—prefetch data into the second level cache.

[PS data] For dword indices, the instruction will prefetch sixteen memory locations. For qword indices, the instruction will prefetch eight values.

[PD data] For dword and qword indices, the instruction will prefetch eight memory locations.

Note that:

(1) The prefetches may happen in any order (or not at all). The instruction is a hint.

(2) The mask is left unchanged.

(3) Not valid with 16-bit effective addresses. Will deliver a #UD fault.

(4) No FP nor memory faults may be produced by this instruction.

(5) Prefetches do not handle cache line splits

(6) A #UD is signaled if the memory operand is encoded without the SIB byte.

Operation

BASE_ADDR stands for the memory operand base address (a GPR); may not exist.
VINDEX stands for the memory operand vector of indices (a vector register).
SCALE stands for the memory operand scalar (1, 2, 4 or 8).
DISP is the optional 1, 2 or 4 byte displacement.
PREFETCH(mem, Level, State) Prefetches a byte memory location pointed by 'mem' into the cache level specified by 'Level'; a request for exclusive/ownership is done if 'State' is 1. Note that the memory location ignore cache line splits. This operation is considered a hint for the processor and may be skipped depending on implementation.

VGATHERPF1DPS (EVEX Encoded Version)

```
(KL, VL) = (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j]
        Prefetch( [BASE_ADDR + SignExtend(VINDEX[i+31:i]) * SCALE + DISP], Level=1, RFO = 0)
    FI;
ENDFOR
```

VGATHERPF1DPD (EVEX Encoded Version)

```
(KL, VL) = (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    k := j * 32
    IF k1[j]
        Prefetch( [BASE_ADDR + SignExtend(VINDEX[k+31:k]) * SCALE + DISP], Level=1, RFO = 0)
    FI;
ENDFOR
```

VGATHERPF1QPS (EVEX Encoded Version)

```
(KL, VL) = (8, 256)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j]
        Prefetch( [BASE_ADDR + SignExtend(VINDEX[i+63:i]) * SCALE + DISP], Level=1, RFO = 0)
    FI;
ENDFOR
```

VGATHERPF1QPD (EVEX Encoded Version)

```
(KL, VL) = (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    k := j * 64
    IF k1[j]
        Prefetch( [BASE_ADDR + SignExtend(VINDEX[k+63:k]) * SCALE + DISP], Level=1, RFO = 0)
    FI;
ENDFOR
```

Intel C/C++ Compiler Intrinsic Equivalent

VGATHERPF1DPD void _mm512_mask_prefetch_i32gather_pd(__m256i vdx, __mmask8 m, void * base, int scale, int hint);
VGATHERPF1DPS void _mm512_mask_prefetch_i32gather_ps(__m512i vdx, __mmask16 m, void * base, int scale, int hint);
VGATHERPF1QPD void _mm512_mask_prefetch_i64gather_pd(__m512i vdx, __mmask8 m, void * base, int scale, int hint);
VGATHERPF1QPS void _mm512_mask_prefetch_i64gather_ps(__m512i vdx, __mmask8 m, void * base, int scale, int hint);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-62, "Type E12NP Class Exception Conditions."

VP4DPWSSDS—Dot Product of Signed Words With Dword Accumulation and Saturation (4-Iterations)

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.512.F2.0F38.W0 53 /r VP4DPWSSDS zmm1{k1}{z}, zmm2+3, m128	A	V/V	AVX512_4VNNIW	Multiply signed words from source register block indicated by zmm2 by signed words from m128 and accumulate the resulting dword results with signed saturation in zmm1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1_4X	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction computes 4 sequential register source-block dot-products of two signed word operands with doubleword accumulation and signed saturation. The memory operand is sequentially selected in each of the four steps.

In the above box, the notation of "+3" is used to denote that the instruction accesses 4 source registers based on that operand; sources are consecutive, start in a multiple of 4 boundary, and contain the encoded register operand.

This instruction supports memory fault suppression. The entire memory operand is loaded if any bit of the lowest 16-bits of the mask is set to 1 or if a "no masking" encoding is used.

The tuple type Tuple1_4X implies that four 32-bit elements (16 bytes) are referenced by the memory operation portion of this instruction.

Operation

src_reg_id is the 5 bit index of the vector register specified in the instruction as the src1 register.

VP4DPWSSDS dest, src1, src2

```
(KL,VL) = (16,512)
N := 4

ORIGDEST := DEST
src_base := src_reg_id & ~ (N-1) // for src1 operand

FOR i := 0 to KL-1:
    IF k1[i] or *no writemask*:
        FOR m := 0 to N-1:
            t := SRC2.dword[m]
            p1dword := reg[src_base+m].word[2*i] * t.word[0]
            p2dword := reg[src_base+m].word[2*i+1] * t.word[1]
            DEST.dword[i] := SIGNED_DWORD_SATURATE(DEST.dword[i] + p1dword + p2dword)
    ELSE IF *zeroing*:
        DEST.dword[i] := 0
    ELSE
        DEST.dword[i] := ORIGDEST.dword[i]
DEST[MAX_VL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VP4DPWSSDS __m512i _mm512_4dpwssds_epi32(__m512i, __m512ix4, __m128i *);
VP4DPWSSDS __m512i _mm512_mask_4dpwssds_epi32(__m512i, __mmask16, __m512ix4, __m128i *);
VP4DPWSSDS __m512i _mm512_maskz_4dpwssds_epi32(__mmask16, __m512i, __m512ix4, __m128i *);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Type E4; additionally:

#UD	If the EVEX broadcast bit is set to 1.
#UD	If the MODRM.mod = 0b11.

VP4DPWSSD—Dot Product of Signed Words With Dword Accumulation (4-Iterations)

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.512.F2.0F38.W0 52 /r VP4DPWSSD zmm1{k1}{z}, zmm2+3, m128	A	V/V	AVX512_4VNNIW	Multiply signed words from source register block indicated by zmm2 by signed words from m128 and accumulate resulting signed dwords in zmm1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1_4X	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

This instruction computes 4 sequential register source-block dot-products of two signed word operands with doubleword accumulation; see Figure 8-1 below. The memory operand is sequentially selected in each of the four steps.

In the above box, the notation of "+3"' is used to denote that the instruction accesses 4 source registers based on that operand; sources are consecutive, start in a multiple of 4 boundary, and contain the encoded register operand.

This instruction supports memory fault suppression. The entire memory operand is loaded if any bit of the lowest 16-bits of the mask is set to 1 or if a "no masking" encoding is used.

The tuple type Tuple1_4X implies that four 32-bit elements (16 bytes) are referenced by the memory operation portion of this instruction.

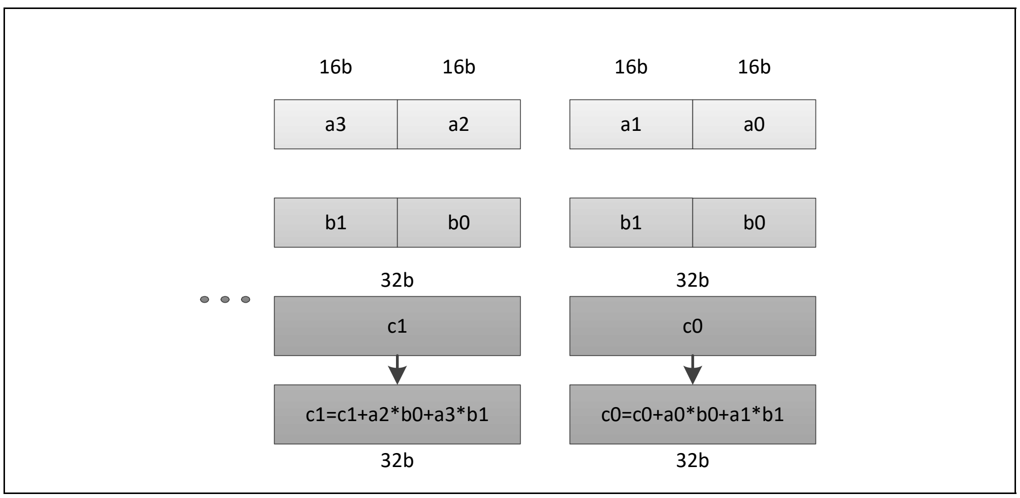

Figure 8-1. Register Source-Block Dot Product of Two Signed Word Operands With Doubleword Accumulation[1]

NOTES:

1. For illustration purposes, one source-block dot product instance is shown out of the four.

Operation

src_reg_id is the 5 bit index of the vector register specified in the instruction as the src1 register.

```
VP4DPWSSD dest, src1, src2
(KL,VL) = (16,512)
N := 4

ORIGDEST := DEST
src_base := src_reg_id & ~ (N-1) // for src1 operand

FOR i := 0 to KL-1:
    IF k1[i] or *no writemask*:
        FOR m := 0 to N-1:
            t := SRC2.dword[m]
            p1dword := reg[src_base+m].word[2*i] * t.word[0]
            p2dword := reg[src_base+m].word[2*i+1] * t.word[1]
            DEST.dword[i] := DEST.dword[i] + p1dword + p2dword
    ELSE IF *zeroing*:
        DEST.dword[i] := 0
    ELSE
        DEST.dword[i] := ORIGDEST.dword[i]
DEST[MAX_VL-1:VL] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VP4DPWSSD __m512i _mm512_4dpwssd_epi32(__m512i, __m512ix4, __m128i *);
VP4DPWSSD __m512i _mm512_mask_4dpwssd_epi32(__m512i, __mmask16, __m512ix4, __m128i *);
VP4DPWSSD __m512i _mm512_maskz_4dpwssd_epi32(__mmask16, __m512i, __m512ix4, __m128i *);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Type E4; additionally:

#UD	If the EVEX broadcast bit is set to 1.
#UD	If the MODRM.mod = 0b11.

VRCP28PD—Approximation to the Reciprocal of Packed Double Precision Floating-Point Values With Less Than 2^-28 Relative Error

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.512.66.0F38.W1 CA /r VRCP28PD zmm1 {k1}{z}, zmm2/m512/m64bcst {sae}	A	V/V	AVX512ER	Computes the approximate reciprocals (< 2^-28 relative error) of the packed double precision floating-point values in zmm2/m512/m64bcst and stores the results in zmm1. Under writemask.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Computes the reciprocal approximation of the float64 values in the source operand (the second operand) and store the results to the destination operand (the first operand). The approximate reciprocal is evaluated with less than 2^-28 of maximum relative error.

Denormal input values are treated as zeros and do not signal #DE, irrespective of MXCSR.DAZ. Denormal results are flushed to zeros and do not signal #UE, irrespective of MXCSR.FTZ.

If any source element is NaN, the quietized NaN source value is returned for that element. If any source element is ±∞, ±0.0 is returned for that element. Also, if any source element is ±0.0, ±∞ is returned for that element.

The source operand is a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

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

A numerically exact implementation of VRCP28xx can be found at https://software.intel.com/en-us/articles/reference-implementations-for-IA-approximation-instructions-vrcp14-vrsqrt14-vrcp28-vrsqrt28-vexp2.

Operation

VRCP28PD (EVEX Encoded Versions)

```
(KL, VL) = (8, 512)

FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC *is memory*)
                THEN DEST[i+63:i] := RCP_28_DP(1.0/SRC[63:0]);
                ELSE DEST[i+63:i] := RCP_28_DP(1.0/SRC[i+63:i]);
            FI;
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[i+63:i] remains unchanged*
            ELSE                            ; zeroing-masking
                DEST[i+63:i] := 0
        FI;
    FI;
ENDFOR;
```

Table 8-3. VRCP28PD Special Cases

Input Value	Result Value	Comments
NAN	QNAN(input)	If (SRC = SNaN) then #I
$0 \le X < 2^{-1022}$	INF	Positive input denormal or zero; #Z
$-2^{-1022} < X \le -0$	-INF	Negative input denormal or zero; #Z
$X > 2^{1022}$	+0.0f	
$X < -2^{1022}$	-0.0f	
$X = +\infty$	+0.0f	
$X = -\infty$	-0.0f	
$X = 2^{-n}$	2^{n}	Exact result (unless input/output is a denormal)
$X = -2^{-n}$	-2^{n}	Exact result (unless input/output is a denormal)

Intel C/C++ Compiler Intrinsic Equivalent

VRCP28PD __m512d _mm512_rcp28_round_pd (__m512d a, int sae);
VRCP28PD __m512d _mm512_mask_rcp28_round_pd(__m512d a, __mmask8 m, __m512d b, int sae);
VRCP28PD __m512d _mm512_maskz_rcp28_round_pd(__mmask8 m, __m512d b, int sae);

SIMD Floating-Point Exceptions

Invalid (if SNaN input), Divide-by-zero.

Other Exceptions

See Table 2-46, "Type E2 Class Exception Conditions."

VRCP28SD—Approximation to the Reciprocal of Scalar Double Precision Floating-Point Value With Less Than 2^-28 Relative Error

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.66.0F38.W1 CB /r VRCP28SD xmm1 {k1}{z}, xmm2, xmm3/m64 {sae}	A	V/V	AVX512ER	Computes the approximate reciprocal (< 2^-28 relative error) of the scalar double precision floating-point value in xmm3/m64 and stores the results in xmm1. Under writemask. Also, upper double precision floating-point value (bits[127:64]) from xmm2 is copied to xmm1[127:64].

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Computes the reciprocal approximation of the low float64 value in the second source operand (the third operand) and store the result to the destination operand (the first operand). The approximate reciprocal is evaluated with less than 2^-28 of maximum relative error. The result is written into the low float64 element of the destination operand according to the writemask k1. Bits 127:64 of the destination is copied from the corresponding bits of the first source operand (the second operand).

A denormal input value is treated as zero and does not signal #DE, irrespective of MXCSR.DAZ. A denormal result is flushed to zero and does not signal #UE, irrespective of MXCSR.FTZ.

If any source element is NaN, the quietized NaN source value is returned for that element. If any source element is ±∞, ±0.0 is returned for that element. Also, if any source element is ±0.0, ±∞ is returned for that element.

The first source operand is an XMM register. The second source operand is an XMM register or a 64-bit memory location. The destination operand is a XMM register, conditionally updated using writemask k1.

A numerically exact implementation of VRCP28xx can be found at https://software.intel.com/en-us/articles/reference-implementations-for-IA-approximation-instructions-vrcp14-vrsqrt14-vrcp28-vrsqrt28-vexp2.

Operation

VRCP28SD ((EVEX Encoded Versions)

```
IF k1[0] OR *no writemask* THEN
            DEST[63: 0] := RCP_28_DP(1.0/SRC2[63: 0]);
ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[63: 0] remains unchanged*
            ELSE                                ; zeroing-masking
                DEST[63: 0] := 0
        FI;
FI;
ENDFOR;
DEST[127:64] := SRC1[127: 64]
DEST[MAXVL-1:128] := 0
```

Table 8-4. VRCP28SD Special Cases

Input Value	Result Value	Comments
NAN	QNAN(input)	If (SRC = SNaN) then #I
$0 \le X < 2^{-1022}$	INF	Positive input denormal or zero; #Z
$-2^{-1022} < X \le -0$	-INF	Negative input denormal or zero; #Z
$X > 2^{1022}$	+0.0f	
$X < -2^{1022}$	-0.0f	
$X = +\infty$	+0.0f	
$X = -\infty$	-0.0f	
$X = 2^{-n}$	2^n	Exact result (unless input/output is a denormal)
$X = -2^{-n}$	-2^n	Exact result (unless input/output is a denormal)

Intel C/C++ Compiler Intrinsic Equivalent

```
VRCP28SD __m128d _mm_rcp28_round_sd ( __m128d a, __m128d b, int sae);
VRCP28SD __m128d _mm_mask_rcp28_round_sd(__m128d s, __mmask8 m, __m128d a, __m128d b, int sae);
VRCP28SD __m128d _mm_maskz_rcp28_round_sd(__mmask8 m, __m128d a, __m128d b, int sae);
```

SIMD Floating-Point Exceptions

Invalid (if SNaN input), Divide-by-zero.

Other Exceptions

See Table 2-47, “Type E3 Class Exception Conditions.”

VRCP28PS—Approximation to the Reciprocal of Packed Single Precision Floating-Point Values With Less Than 2^-28 Relative Error

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.512.66.0F38.W0 CA /r VRCP28PS zmm1 {k1}{z}, zmm2/m512/m32bcst {sae}	A	V/V	AVX512ER	Computes the approximate reciprocals (< 2^-28 relative error) of the packed single-precision floating-point values in zmm2/m512/m32bcst and stores the results in zmm1. Under writemask.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Computes the reciprocal approximation of the float32 values in the source operand (the second operand) and store the results to the destination operand (the first operand) using the writemask k1. The approximate reciprocal is evaluated with less than 2^-28 of maximum relative error prior to final rounding. The final results are rounded to < 2^-23 relative error before written to the destination.

Denormal input values are treated as zeros and do not signal #DE, irrespective of MXCSR.DAZ. Denormal results are flushed to zeros and do not signal #UE, irrespective of MXCSR.FTZ.

If any source element is NaN, the quietized NaN source value is returned for that element. If any source element is ±∞, ±0.0 is returned for that element. Also, if any source element is ±0.0, ±∞ is returned for that element.

The source operand is a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

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

A numerically exact implementation of VRCP28xx can be found at https://software.intel.com/en-us/articles/reference-implementations-for-IA-approximation-instructions-vrcp14-vrsqrt14-vrcp28-vrsqrt28-vexp2.

Operation

VRCP28PS (EVEX Encoded Versions)

```
(KL, VL) = (16, 512)

FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC *is memory*)
                THEN DEST[i+31:i] := RCP_28_SP(1.0/SRC[31:0]);
                ELSE DEST[i+31:i] := RCP_28_SP(1.0/SRC[i+31:i]);
            FI;
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[i+31:i] remains unchanged*
            ELSE                            ; zeroing-masking
                DEST[i+31:i] := 0
        FI;
    FI;
ENDFOR;
```

Table 8-5. VRCP28PS Special Cases

Input Value	Result Value	Comments
NAN	QNAN(input)	If (SRC = SNaN) then #I
$0 \le X < 2^{-126}$	INF	Positive input denormal or zero; #Z
$-2^{-126} < X \le -0$	-INF	Negative input denormal or zero; #Z
$X > 2^{126}$	+0.0f	
$X < -2^{126}$	-0.0f	
$X = +\infty$	+0.0f	
$X = -\infty$	-0.0f	
$X = 2^{-n}$	2^{n}	Exact result (unless input/output is a denormal)
$X = -2^{-n}$	-2^{n}	Exact result (unless input/output is a denormal)

Intel C/C++ Compiler Intrinsic Equivalent

VRCP28PS _mm512_rcp28_round_ps (__m512 a, int sae);
VRCP28PS __m512 _mm512_mask_rcp28_round_ps(__m512 s, __mmask16 m, __m512 a, int sae);
VRCP28PS __m512 _mm512_maskz_rcp28_round_ps(__mmask16 m, __m512 a, int sae);

SIMD Floating-Point Exceptions

Invalid (if SNaN input), Divide-by-zero.

Other Exceptions

See Table 2-46, “Type E2 Class Exception Conditions.”

VRCP28SS—Approximation to the Reciprocal of Scalar Single Precision Floating-Point Value With Less Than 2^-28 Relative Error

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.66.0F38.W0 CB /r VRCP28SS xmm1 {k1}{z}, xmm2, xmm3/m32 {sae}	A	V/V	AVX512ER	Computes the approximate reciprocal (< 2^-28 relative error) of the scalar single-precision floating-point value in xmm3/m32 and stores the results in xmm1. Under writemask. Also, upper 3 single-precision floating-point values (bits[127:32]) from xmm2 is copied to xmm1[127:32].

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Computes the reciprocal approximation of the low float32 value in the second source operand (the third operand) and store the result to the destination operand (the first operand). The approximate reciprocal is evaluated with less than 2^-28 of maximum relative error prior to final rounding. The final result is rounded to < 2^-23 relative error before written into the low float32 element of the destination according to writemask k1. Bits 127:32 of the destination is copied from the corresponding bits of the first source operand (the second operand).

A denormal input value is treated as zero and does not signal #DE, irrespective of MXCSR.DAZ. A denormal result is flushed to zero and does not signal #UE, irrespective of MXCSR.FTZ.

If any source element is NaN, the quietized NaN source value is returned for that element. If any source element is ±∞, ±0.0 is returned for that element. Also, if any source element is ±0.0, ±∞ is returned for that element.

The first source operand is an XMM register. The second source operand is an XMM register or a 32-bit memory location. The destination operand is a XMM register, conditionally updated using writemask k1.

A numerically exact implementation of VRCP28xx can be found at https://software.intel.com/en-us/articles/reference-implementations-for-IA-approximation-instructions-vrcp14-vrsqrt14-vrcp28-vrsqrt28-vexp2.

Operation

VRCP28SS ((EVEX Encoded Versions)

```
IF k1[0] OR *no writemask* THEN
            DEST[31: 0] := RCP_28_SP(1.0/SRC2[31: 0]);
ELSE
        IF *merging-masking*                  ; merging-masking
            THEN *DEST[31: 0] remains unchanged*
            ELSE                              ; zeroing-masking
                DEST[31: 0] := 0
        FI;
FI;
ENDFOR;
DEST[127:32] := SRC1[127: 32]
DEST[MAXVL-1:128] := 0
```

Table 8-6. VRCP28SS Special Cases

Input Value	Result Value	Comments
NAN	QNAN(input)	If (SRC = SNaN) then #I
$0 \le X < 2^{-126}$	INF	Positive input denormal or zero; #Z
$-2^{-126} < X \le -0$	-INF	Negative input denormal or zero; #Z
$X > 2^{126}$	+0.0f	
$X < -2^{126}$	-0.0f	
$X = +\infty$	+0.0f	
$X = -\infty$	-0.0f	
$X = 2^{-n}$	2^n	Exact result (unless input/output is a denormal)
$X = -2^{-n}$	-2^n	Exact result (unless input/output is a denormal)

Intel C/C++ Compiler Intrinsic Equivalent

VRCP28SS __m128 _mm_rcp28_round_ss (__m128 a, __m128 b, int sae);
VRCP28SS __m128 _mm_mask_rcp28_round_ss(__m128 s, __mmask8 m, __m128 a, __m128 b, int sae);
VRCP28SS __m128 _mm_maskz_rcp28_round_ss(__mmask8 m, __m128 a, __m128 b, int sae);

SIMD Floating-Point Exceptions

Invalid (if SNaN input), Divide-by-zero.

Other Exceptions

See Table 2-47, “Type E3 Class Exception Conditions.”

VRSQRT28PD—Approximation to the Reciprocal Square Root of Packed Double Precision Floating-Point Values With Less Than 2^-28 Relative Error

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.512.66.0F38.W1 CC /r VRSQRT28PD zmm1 {k1}{z}, zmm2/m512/m64bcst {sae}	A	V/V	AVX512ER	Computes approximations to the Reciprocal square root (<2^-28 relative error) of the packed double precision floating-point values from zmm2/m512/m64bcst and stores result in zmm1with writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Computes the reciprocal square root of the float64 values in the source operand (the second operand) and store the results to the destination operand (the first operand). The approximate reciprocal is evaluated with less than 2^-28 of maximum relative error.

If any source element is NaN, the quietized NaN source value is returned for that element. Negative (non-zero) source numbers, as well as -∞, return the canonical NaN and set the Invalid Flag (#I).

A value of -0 must return -∞ and set the DivByZero flags (#Z). Negative numbers should return NaN and set the Invalid flag (#I). Note however that the instruction flush input denormals to zero of the same sign, so negative denormals return -∞ and set the DivByZero flag.

The source operand is a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

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

A numerically exact implementation of VRSQRT28xx can be found at https://software.intel.com/en-us/articles/reference-implementations-for-IA-approximation-instructions-vrcp14-vrsqrt14-vrcp28-vrsqrt28-vexp2.

Operation

VRSQRT28PD (EVEX Encoded Versions)

```
(KL, VL) = (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC *is memory*)
                THEN DEST[i+63:i] := (1.0/ SQRT(SRC[63:0]));
                ELSE DEST[i+63:i] := (1.0/ SQRT(SRC[i+63:i]));
            FI;
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[i+63:i] remains unchanged*
            ELSE                            ; zeroing-masking
                DEST[i+63:i] := 0
        FI;
    FI;
ENDFOR;
```

Table 8-7. VRSQRT28PD Special Cases

Input Value	Result Value	Comments
NAN	QNAN(input)	If (SRC = SNaN) then #I
$X = 2^{-2n}$	2^n	
X < 0	QNaN_Indefinite	Including -INF
X = -0 or negative denormal	-INF	#Z
X = +0 or positive denormal	+INF	#Z
X = +INF	+0	

Intel C/C++ Compiler Intrinsic Equivalent

VRSQRT28PD __m512d _mm512_rsqrt28_round_pd(__m512d a, int sae);
VRSQRT28PD __m512d _mm512_mask_rsqrt28_round_pd(__m512d s, __mmask8 m,__m512d a, int sae);
VRSQRT28PD __m512d _mm512_maskz_rsqrt28_round_pd(__mmask8 m,__m512d a, int sae);

SIMD Floating-Point Exceptions

Invalid (if SNaN input), Divide-by-zero.

Other Exceptions

See Table 2-46, "Type E2 Class Exception Conditions."

VRSQRT28SD—Approximation to the Reciprocal Square Root of Scalar Double Precision Floating-Point Value With Less Than 2^-28 Relative Error

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.66.0F38.W1 CD /r VRSQRT28SD xmm1 {k1}{z}, xmm2, xmm3/m64 {sae}	A	V/V	AVX512ER	Computes approximate reciprocal square root (<2^-28 relative error) of the scalar double precision floating-point value from xmm3/m64 and stores result in xmm1with writemask k1. Also, upper double precision floating-point value (bits[127:64]) from xmm2 is copied to xmm1[127:64].

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Computes the reciprocal square root of the low float64 value in the second source operand (the third operand) and store the result to the destination operand (the first operand). The approximate reciprocal square root is evaluated with less than 2^-28 of maximum relative error. The result is written into the low float64 element of xmm1 according to the writemask k1. Bits 127:64 of the destination is copied from the corresponding bits of the first source operand (the second operand).

If any source element is NaN, the quietized NaN source value is returned for that element. Negative (non-zero) source numbers, as well as -∞, return the canonical NaN and set the Invalid Flag (#I).

A value of -0 must return -∞ and set the DivByZero flags (#Z). Negative numbers should return NaN and set the Invalid flag (#I). Note however that the instruction flush input denormals to zero of the same sign, so negative denormals return -∞ and set the DivByZero flag.

The first source operand is an XMM register. The second source operand is an XMM register or a 64-bit memory location. The destination operand is a XMM register.

A numerically exact implementation of VRSQRT28xx can be found at https://software.intel.com/en-us/articles/reference-implementations-for-IA-approximation-instructions-vrcp14-vrsqrt14-vrcp28-vrsqrt28-vexp2.

Operation

VRSQRT28SD (EVEX Encoded Versions)

```
    IF k1[0] OR *no writemask* THEN
            DEST[63: 0] := (1.0/ SQRT(SRC[63: 0]));
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[63: 0] remains unchanged*
            ELSE                            ; zeroing-masking
                DEST[63: 0] := 0
        FI;
    FI;
ENDFOR;
DEST[127:64] := SRC1[127: 64]
DEST[MAXVL-1:128] := 0
```

Table 8-8. VRSQRT28SD Special Cases

Input Value	Result Value	Comments
NAN	QNAN(input)	If (SRC = SNaN) then #I
$X = 2^{-2n}$	2^{n}	
X < 0	QNaN_Indefinite	Including -INF
X = -0 or negative denormal	-INF	#Z
X = +0 or positive denormal	+INF	#Z
X = +INF	+0	

Intel C/C++ Compiler Intrinsic Equivalent

VRSQRT28SD __m128d _mm_rsqrt28_round_sd(__m128d a, __m128d b, int rounding);
VRSQRT28SD __m128d _mm_mask_rsqrt28_round_sd(__m128d s, __mmask8 m,__m128d a, __m128d b, int rounding);
VRSQRT28SD __m128d _mm_maskz_rsqrt28_round_sd(__mmask8 m,__m128d a, __m128d b, int rounding);

SIMD Floating-Point Exceptions

Invalid (if SNaN input), Divide-by-zero.

Other Exceptions

See Table 2-47, “Type E3 Class Exception Conditions.”

VRSQRT28PS—Approximation to the Reciprocal Square Root of Packed Single Precision Floating-Point Values With Less Than 2^-28 Relative Error

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.512.66.0F38.W0 CC /r VRSQRT28PS zmm1 {k1}{z}, zmm2/m512/m32bcst {sae}	A	V/V	AVX512ER	Computes approximations to the Reciprocal square root (<2^-28 relative error) of the packed single-precision floating-point values from zmm2/m512/m32bcst and stores result in zmm1with writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	N/A	N/A

Description

Computes the reciprocal square root of the float32 values in the source operand (the second operand) and store the results to the destination operand (the first operand). The approximate reciprocal is evaluated with less than 2^-28 of maximum relative error prior to final rounding. The final results is rounded to < 2^-23 relative error before written to the destination.

If any source element is NaN, the quietized NaN source value is returned for that element. Negative (non-zero) source numbers, as well as -∞, return the canonical NaN and set the Invalid Flag (#I).

A value of -0 must return -∞ and set the DivByZero flags (#Z). Negative numbers should return NaN and set the Invalid flag (#I). Note however that the instruction flush input denormals to zero of the same sign, so negative denormals return -∞ and set the DivByZero flag.

The source operand is a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

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

A numerically exact implementation of VRSQRT28xx can be found at https://software.intel.com/en-us/articles/reference-implementations-for-IA-approximation-instructions-vrcp14-vrsqrt14-vrcp28-vrsqrt28-vexp2.

Operation

VRSQRT28PS (EVEX Encoded Versions)

```
(KL, VL) = (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC *is memory*)
                THEN DEST[i+31:i] := (1.0/ SQRT(SRC[31:0]));
                ELSE DEST[i+31:i] := (1.0/ SQRT(SRC[i+31:i]));
            FI;
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[i+31:i] remains unchanged*
            ELSE                            ; zeroing-masking
                DEST[i+31:i] := 0
        FI;
    FI;
ENDFOR;
```

Table 8-9. VRSQRT28PS Special Cases

Input Value	Result Value	Comments
NAN	QNAN(input)	If (SRC = SNaN) then #I
$X = 2^{-2n}$	2^n	
X < 0	QNaN_Indefinite	Including -INF
X = -0 or negative denormal	-INF	#Z
X = +0 or positive denormal	+INF	#Z
X = +INF	+0	

Intel C/C++ Compiler Intrinsic Equivalent

VRSQRT28PS __m512 _mm512_rsqrt28_round_ps(__m512 a, int sae);
VRSQRT28PS __m512 _mm512_mask_rsqrt28_round_ps(__m512 s, __mmask16 m,__m512 a, int sae);
VRSQRT28PS __m512 _mm512_maskz_rsqrt28_round_ps(__mmask16 m,__m512 a, int sae);

SIMD Floating-Point Exceptions

Invalid (if SNaN input), Divide-by-zero.

Other Exceptions

See Table 2-46, “Type E2 Class Exception Conditions.”

VRSQRT28SS—Approximation to the Reciprocal Square Root of Scalar Single Precision Floating-Point Value With Less Than 2^-28 Relative Error

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.66.0F38.W0 CD /r VRSQRT28SS xmm1 {k1}{z}, xmm2, xmm3/m32 {sae}	A	V/V	AVX512ER	Computes approximate reciprocal square root (<2^-28 relative error) of the scalar single-precision floating-point value from xmm3/m32 and stores result in xmm1with writemask k1. Also, upper 3 single-precision floating-point value (bits[127:32]) from xmm2 is copied to xmm1[127:32].

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Computes the reciprocal square root of the low float32 value in the second source operand (the third operand) and store the result to the destination operand (the first operand). The approximate reciprocal square root is evaluated with less than 2^-28 of maximum relative error prior to final rounding. The final result is rounded to < 2^-23 relative error before written to the low float32 element of the destination according to the writemask k1. Bits 127:32 of the destination is copied from the corresponding bits of the first source operand (the second operand).

If any source element is NaN, the quietized NaN source value is returned for that element. Negative (non-zero) source numbers, as well as -∞, return the canonical NaN and set the Invalid Flag (#I).

A value of -0 must return -∞ and set the DivByZero flags (#Z). Negative numbers should return NaN and set the Invalid flag (#I). Note however that the instruction flush input denormals to zero of the same sign, so negative denormals return -∞ and set the DivByZero flag.

The first source operand is an XMM register. The second source operand is an XMM register or a 32-bit memory location. The destination operand is a XMM register.

A numerically exact implementation of VRSQRT28xx can be found at https://software.intel.com/en-us/articles/reference-implementations-for-IA-approximation-instructions-vrcp14-vrsqrt14-vrcp28-vrsqrt28-vexp2.

Operation

VRSQRT28SS (EVEX Encoded Versions)

```
    IF k1[0] OR *no writemask* THEN
            DEST[31: 0] := (1.0/ SQRT(SRC[31: 0]));
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[31: 0] remains unchanged*
            ELSE                            ; zeroing-masking
                DEST[31: 0] := 0
        FI;
    FI;
ENDFOR;
DEST[127:32] := SRC1[127: 32]
DEST[MAXVL-1:128] := 0
```

Table 8-10. VRSQRT28SS Special Cases

Input Value	Result Value	Comments
NAN	QNAN(input)	If (SRC = SNaN) then #I
$X = 2^{-2n}$	2^n	
X < 0	QNaN_Indefinite	Including -INF
X = -0 or negative denormal	-INF	#Z
X = +0 or positive denormal	+INF	#Z
X = +INF	+0	

Intel C/C++ Compiler Intrinsic Equivalent

VRSQRT28SS __m128 _mm_rsqrt28_round_ss(__m128 a, __m128 b, int rounding);
VRSQRT28SS __m128 _mm_mask_rsqrt28_round_ss(__m128 s, __mmask8 m,__m128 a,__m128 b, int rounding);
VRSQRT28SS __m128 _mm_maskz_rsqrt28_round_ss(__mmask8 m,__m128 a,__m128 b, int rounding);

SIMD Floating-Point Exceptions

Invalid (if SNaN input), Divide-by-zero.

Other Exceptions

See Table 2-47, "Type E3 Class Exception Conditions."

VSCATTERPF0DPS/VSCATTERPF0QPS/VSCATTERPF0DPD/VSCATTERPF0QPD—Sparse Prefetch Packed SP/DP Data Values with Signed Dword, Signed Qword Indices Using T0 Hint With Intent to Write

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.512.66.0F38.W0 C6 /5 /vsib VSCATTERPF0DPS vm32z {k1}	A	V/V	AVX512PF	Using signed dword indices, prefetch sparse byte memory locations containing single-precision data using writemask k1 and T0 hint with intent to write.
EVEX.512.66.0F38.W0 C7 /5 /vsib VSCATTERPF0QPS vm64z {k1}	A	V/V	AVX512PF	Using signed qword indices, prefetch sparse byte memory locations containing single-precision data using writemask k1 and T0 hint with intent to write.
EVEX.512.66.0F38.W1 C6 /5 /vsib VSCATTERPF0DPD vm32y {k1}	A	V/V	AVX512PF	Using signed dword indices, prefetch sparse byte memory locations containing double precision data using writemask k1 and T0 hint with intent to write.
EVEX.512.66.0F38.W1 C7 /5 /vsib VSCATTERPF0QPD vm64z {k1}	A	V/V	AVX512PF	Using signed qword indices, prefetch sparse byte memory locations containing double precision data using writemask k1 and T0 hint with intent to write.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	BaseReg (R): VSIB:base, VectorReg(R): VSIB:index	N/A	N/A	N/A

Description

The instruction conditionally prefetches up to sixteen 32-bit or eight 64-bit integer byte data elements. The elements are specified via the VSIB (i.e., the index register is an zmm, holding packed indices). Elements will only be prefetched if their corresponding mask bit is one.

cache lines will be brought into exclusive state (RFO) specified by a locality hint (T0):

• T0 (temporal data)—prefetch data into the first level cache.

[PS data] For dword indices, the instruction will prefetch sixteen memory locations. For qword indices, the instruction will prefetch eight values.

[PD data] For dword and qword indices, the instruction will prefetch eight memory locations.

Note that:

(1) The prefetches may happen in any order (or not at all). The instruction is a hint.

(2) The mask is left unchanged.

(3) Not valid with 16-bit effective addresses. Will deliver a #UD fault.

(4) No FP nor memory faults may be produced by this instruction.

(5) Prefetches do not handle cache line splits

(6) A #UD is signaled if the memory operand is encoded without the SIB byte.

Operation

BASE_ADDR stands for the memory operand base address (a GPR); may not exist.
VINDEX stands for the memory operand vector of indices (a vector register).
SCALE stands for the memory operand scalar (1, 2, 4 or 8).
DISP is the optional 1, 2 or 4 byte displacement.
PREFETCH(mem, Level, State) Prefetches a byte memory location pointed by 'mem' into the cache level specified by 'Level'; a request for exclusive/ownership is done if 'State' is 1. Note that the memory location ignore cache line splits. This operation is considered a hint for the processor and may be skipped depending on implementation.

VSCATTERPF0DPS (EVEX Encoded Version)

```
(KL, VL) = (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j]
        Prefetch( [BASE_ADDR + SignExtend(VINDEX[i+31:i]) * SCALE + DISP], Level=0, RFO = 1)
    FI;
ENDFOR
```

VSCATTERPF0DPD (EVEX Encoded Version)

```
(KL, VL) = (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    k := j * 32
    IF k1[j]
        Prefetch( [BASE_ADDR + SignExtend(VINDEX[k+31:k]) * SCALE + DISP], Level=0, RFO = 1)
    FI;
ENDFOR
```

VSCATTERPF0QPS (EVEX Encoded Version)

```
(KL, VL) = (8, 256)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j]
        Prefetch( [BASE_ADDR + SignExtend(VINDEX[i+63:i]) * SCALE + DISP], Level=0, RFO = 1)
    FI;
ENDFOR
```

VSCATTERPF0QPD (EVEX Encoded Version)

```
(KL, VL) = (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    k := j * 64
    IF k1[j]
        Prefetch( [BASE_ADDR + SignExtend(VINDEX[k+63:k]) * SCALE + DISP], Level=0, RFO = 1)
    FI;
ENDFOR
```

Intel C/C++ Compiler Intrinsic Equivalent

VSCATTERPF0DPD void _mm512_prefetch_i32scatter_pd(void *base, __m256i vdx, int scale, int hint);
VSCATTERPF0DPD void _mm512_mask_prefetch_i32scatter_pd(void *base, __mmask8 m, __m256i vdx, int scale, int hint);
VSCATTERPF0DPS void _mm512_prefetch_i32scatter_ps(void *base, __m512i vdx, int scale, int hint);
VSCATTERPF0DPS void _mm512_mask_prefetch_i32scatter_ps(void *base, __mmask16 m, __m512i vdx, int scale, int hint);
VSCATTERPF0QPD void _mm512_prefetch_i64scatter_pd(void * base, __m512i vdx, int scale, int hint);
VSCATTERPF0QPD void _mm512_mask_prefetch_i64scatter_pd(void * base, __mmask8 m, __m512i vdx, int scale, int hint);
VSCATTERPF0QPS void _mm512_prefetch_i64scatter_ps(void * base, __m512i vdx, int scale, int hint);
VSCATTERPF0QPS void _mm512_mask_prefetch_i64scatter_ps(void * base, __mmask8 m, __m512i vdx, int scale, int hint);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-62, “Type E12NP Class Exception Conditions.”

VSCATTERPF1DPS/VSCATTERPF1QPS/VSCATTERPF1DPD/VSCATTERPF1QPD—Sparse Prefetch Packed SP/DP Data Values With Signed Dword, Signed Qword Indices Using T1 Hint With Intent to Write

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.512.66.0F38.W0 C6 /6 /vsib VSCATTERPF1DPS vm32z {k1}	A	V/V	AVX512PF	Using signed dword indices, prefetch sparse byte memory locations containing single-precision data using writemask k1 and T1 hint with intent to write.
EVEX.512.66.0F38.W0 C7 /6 /vsib VSCATTERPF1QPS vm64z {k1}	A	V/V	AVX512PF	Using signed qword indices, prefetch sparse byte memory locations containing single-precision data using writemask k1 and T1 hint with intent to write.
EVEX.512.66.0F38.W1 C6 /6 /vsib VSCATTERPF1DPD vm32y {k1}	A	V/V	AVX512PF	Using signed dword indices, prefetch sparse byte memory locations containing double precision data using writemask k1 and T1 hint with intent to write.
EVEX.512.66.0F38.W1 C7 /6 /vsib VSCATTERPF1QPD vm64z {k1}	A	V/V	AVX512PF	Using signed qword indices, prefetch sparse byte memory locations containing double precision data using writemask k1 and T1 hint with intent to write.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	BaseReg (R): VSIB:base, VectorReg(R): VSIB:index	N/A	N/A	N/A

Description

The instruction conditionally prefetches up to sixteen 32-bit or eight 64-bit integer byte data elements. The elements are specified via the VSIB (i.e., the index register is an zmm, holding packed indices). Elements will only be prefetched if their corresponding mask bit is one.

cache lines will be brought into exclusive state (RFO) specified by a locality hint (T1):

• T1 (temporal data)—prefetch data into the second level cache.

[PS data] For dword indices, the instruction will prefetch sixteen memory locations. For qword indices, the instruction will prefetch eight values.

[PD data] For dword and qword indices, the instruction will prefetch eight memory locations.

Note that:

(1) The prefetches may happen in any order (or not at all). The instruction is a hint.

(2) The mask is left unchanged.

(3) Not valid with 16-bit effective addresses. Will deliver a #UD fault.

(4) No FP nor memory faults may be produced by this instruction.

(5) Prefetches do not handle cache line splits

(6) A #UD is signaled if the memory operand is encoded without the SIB byte.

Operation

BASE_ADDR stands for the memory operand base address (a GPR); may not exist.
VINDEX stands for the memory operand vector of indices (a vector register).
SCALE stands for the memory operand scalar (1, 2, 4 or 8).
DISP is the optional 1, 2 or 4 byte displacement.
PREFETCH(mem, Level, State) Prefetches a byte memory location pointed by 'mem' into the cache level specified by 'Level'; a request for exclusive/ownership is done if 'State' is 1. Note that the memory location ignore cache line splits. This operation is considered a hint for the processor and may be skipped depending on implementation.

VSCATTERPF1DPS (EVEX Encoded Version)

```
(KL, VL) = (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j]
        Prefetch( [BASE_ADDR + SignExtend(VINDEX[i+31:i]) * SCALE + DISP], Level=1, RFO = 1)
    FI;
ENDFOR
```

VSCATTERPF1DPD (EVEX Encoded Version)

```
(KL, VL) = (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    k := j * 32
    IF k1[j]
        Prefetch( [BASE_ADDR + SignExtend(VINDEX[k+31:k]) * SCALE + DISP], Level=1, RFO = 1)
    FI;
ENDFOR
```

VSCATTERPF1QPS (EVEX Encoded Version)

```
(KL, VL) = (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j]
        Prefetch( [BASE_ADDR + SignExtend(VINDEX[i+63:i]) * SCALE + DISP], Level=1, RFO = 1)
    FI;
ENDFOR
```

VSCATTERPF1QPD (EVEX Encoded Version)

```
(KL, VL) = (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    k := j * 64
    IF k1[j]
        Prefetch( [BASE_ADDR + SignExtend(VINDEX[k+63:k]) * SCALE + DISP], Level=1, RFO = 1)
    FI;
ENDFOR
```

Intel C/C++ Compiler Intrinsic Equivalent

VSCATTERPF1DPD void _mm512_prefetch_i32scatter_pd(void *base, __m256i vdx, int scale, int hint);
VSCATTERPF1DPD void _mm512_mask_prefetch_i32scatter_pd(void *base, __mmask8 m, __m256i vdx, int scale, int hint);
VSCATTERPF1DPS void _mm512_prefetch_i32scatter_ps(void *base, __m512i vdx, int scale, int hint);
VSCATTERPF1DPS void _mm512_mask_prefetch_i32scatter_ps(void *base, __mmask16 m, __m512i vdx, int scale, int hint);
VSCATTERPF1QPD void _mm512_prefetch_i64scatter_pd(void * base, __m512i vdx, int scale, int hint);
VSCATTERPF1QPD void _mm512_mask_prefetch_i64scatter_pd(void * base, __mmask8 m, __m512i vdx, int scale, int hint);
VSCATTERPF1QPS void _mm512_prefetch_i64scatter_ps(void *base, __m512i vdx, int scale, int hint);
VSCATTERPF1QPS void _mm512_mask_prefetch_i64scatter_ps(void *base, __mmask8 m, __m512i vdx, int scale, int hint);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-62, "Type E12NP Class Exception Conditions."

APPENDIX A
OPCODE MAP

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. Cells marked "Reserved-NOP" are also reserved but may behave as NOP on certain processors. Software should not use opcodes corresponding blank cells or cells marked "Reserved-NOP" nor depend on the current behavior of those opcodes.

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

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

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

A.2 KEY TO ABBREVIATIONS

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

A.2.1 Codes for Addressing Method

The following abbreviations are used to document addressing methods:

A Direct address: the instruction has no ModR/M byte; the address of the operand is encoded in the instruction. No base register, index register, or scaling factor can be applied (for example, far JMP (EA)).

B The VEX.vvvv field of the VEX prefix selects a general purpose register.

C The reg field of the ModR/M byte selects a control register (for example, MOV (0F20, 0F22)).

D The reg field of the ModR/M byte selects a debug register (for example, MOV (0F21,0F23)).

E A ModR/M byte follows the opcode and specifies the operand. The operand is either a general-purpose register or a memory address. If it is a memory address, the address is computed from a segment register and any of the following values: a base register, an index register, a scaling factor, a displacement.

F EFLAGS/RFLAGS Register.

G The reg field of the ModR/M byte selects a general register (for example, AX (000)).

H The VEX.vvvv field of the VEX prefix selects a 128-bit XMM register or a 256-bit YMM register, determined by operand type. For legacy SSE encodings this operand does not exist, changing the instruction to destructive form.

I Immediate data: the operand value is encoded in subsequent bytes of the instruction.

J The instruction contains a relative offset to be added to the instruction pointer register (for example, JMP (0E9), LOOP).

L The upper 4 bits of the 8-bit immediate selects a 128-bit XMM register or a 256-bit YMM register, determined by operand type. (the MSB is ignored in 32-bit mode)

M The ModR/M byte may refer only to memory (for example, BOUND, LES, LDS, LSS, LFS, LGS, CMPXCHG8B).

N The R/M field of the ModR/M byte selects a packed-quadword, MMX technology register.

O The instruction has no ModR/M byte. The offset of the operand is coded as a word or double word (depending on address size attribute) in the instruction. No base register, index register, or scaling factor can be applied (for example, MOV (A0–A3)).

P The reg field of the ModR/M byte selects a packed quadword MMX technology register.

Q A ModR/M byte follows the opcode and specifies the operand. The operand is either an MMX technology register or a memory address. If it is a memory address, the address is computed from a segment register and any of the following values: a base register, an index register, a scaling factor, and a displacement.

R The R/M field of the ModR/M byte may refer only to a general register (for example, MOV (0F20-0F23)).

S The reg field of the ModR/M byte selects a segment register (for example, MOV (8C,8E)).

U The R/M field of the ModR/M byte selects a 128-bit XMM register or a 256-bit YMM register, determined by operand type.

V The reg field of the ModR/M byte selects a 128-bit XMM register or a 256-bit YMM register, determined by operand type.

W A ModR/M byte follows the opcode and specifies the operand. The operand is either a 128-bit XMM register, a 256-bit YMM register (determined by operand type), or a memory address. If it is a memory address, the address is computed from a segment register and any of the following values: a base register, an index register, a scaling factor, and a displacement.

X Memory addressed by the DS:rSI register pair (for example, MOVS, CMPS, OUTS, or LODS).

Y Memory addressed by the ES:rDI register pair (for example, MOVS, CMPS, INS, STOS, or SCAS).

A.2.2 Codes for Operand Type

The following abbreviations are used to document operand types:

a Two one-word operands in memory or two double-word operands in memory, depending on operand-size attribute (used only by the BOUND instruction).

b Byte, regardless of operand-size attribute.

c Byte or word, depending on operand-size attribute.

d Doubleword, regardless of operand-size attribute.

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

A.2.3 Register Codes

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

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

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

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

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

A.2.4.1 One-Byte Opcode Instructions

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

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

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

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

- A ModR/M byte is required and includes an opcode extension in the reg field in the ModR/M byte. Use Table A-6 when interpreting the ModR/M byte.
- Use of the ModR/M byte is reserved or undefined. This applies to entries that represent an instruction prefix or entries for instructions without operands that use ModR/M (for example: 60H, 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 0FA4050000000003H 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), the 0FB9H opcode (UD1 instruction), or the 0FFFH opcode (UD0 instruction) when deliberately trying to generate an invalid opcode exception (#UD).
1C	Some instructions use the same two-byte opcode. If the instruction has variations, or the opcode represents different instructions, the ModR/M byte will be used to differentiate the instruction. For the value of the ModR/M byte needed to decode the instruction, see Table A-6.
i64	The instruction is invalid or not encodable in 64-bit mode. 40 through 4F (single-byte INC and DEC) are REX prefix combinations when in 64-bit mode (use FE/FF Grp 4 and 5 for INC and DEC).
o64	Instruction is only available when in 64-bit mode.
d64	When in 64-bit mode, instruction defaults to 64-bit operand size and cannot encode 32-bit operand size.
f64	The operand size is forced to a 64-bit operand size when in 64-bit mode (prefixes that change operand size are ignored for this instruction in 64-bit mode).
v	VEX form only exists. There is no legacy SSE form of the instruction. For Integer GPR instructions it means VEX prefix required.
v1	VEX128 & SSE forms only exist (no VEX256), when can't be inferred from the data size.

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

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

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

	0	1	2	3	4	5	6	7
0	ADD						PUSH ES^{i64}	POP ES^{i64}
	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz		
1	ADC						PUSH SS^{i64}	POP SS^{i64}
	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz		
2	AND						SEG=ES (Prefix)	DAA^{i64}
	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz		
3	XOR						SEG=SS (Prefix)	AAA^{i64}
	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz		
4	INC^{i64} general register / REX^{o64} 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^{d64} general register							
	rAX/r8	rCX/r9	rDX/r10	rBX/r11	rSP/r12	rBP/r13	rSI/r14	rDI/r15
6	PUSHA^{i64}/ PUSHAD^{i64}	POPA^{i64}/ POPAD^{i64}	BOUND^{i64} Gv, Ma	ARPL^{i64} Ew, Gw MOVSXD^{o64} Gv, Ev	SEG=FS (Prefix)	SEG=GS (Prefix)	Operand Size (Prefix)	Address Size (Prefix)
7	Jcc^{f64}, 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^{1A}				TEST		XCHG	
	Eb, Ib	Ev, Iz	Eb, Ib^{i64}	Ev, Ib	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/R8B, Ib	CL/R9B, Ib	DL/R10B, Ib	BL/R11B, Ib	AH/R12B, Ib	CH/R13B, Ib	DH/R14B, Ib	BH/R15B, Ib
C	Shift Grp 2^{1A}		near RET^{f64} Iw	near RET^{f64}	LES^{i64} Gz, Mp VEX+2byte	LDS^{i64} Gz, Mp VEX+1byte	Grp 11^{1A} - MOV	
	Eb, Ib	Ev, Ib					Eb, Ib	Ev, Iz
D	Shift Grp 2^{1A}				AAM^{i64} Ib	AAD^{i64} Ib		XLAT/ XLATB
	Eb, 1	Ev, 1	Eb, CL	Ev, CL				
E	LOOPNE^{f64}/ LOOPNZ^{f64} Jb	LOOPE^{f64}/ LOOPZ^{f64} Jb	LOOP^{f64} Jb	JrCXZ^{f64}/ Jb	IN		OUT	
					AL, Ib	eAX, Ib	Ib, AL	Ib, eAX
F	LOCK (Prefix)	INT1	REPNE XACQUIRE (Prefix)	REP/REPE XRELEASE (Prefix)	HLT	CMC	Unary Grp 3^{1A}	
							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, Ib	rAX, Iz	PUSH CS[i64]	2-byte escape (Table A-3)
1	SBB Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz	PUSH DS[i64]	POP DS[i64]
2	SUB Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz	SEG=CS (Prefix)	DAS[i64]
3	CMP Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz	SEG=DS (Prefix)	AAS[i64]
4	DEC[i64] general register / REX[o64] 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[d64] into general register rAX/r8	rCX/r9	rDX/r10	rBX/r11	rSP/r12	rBP/r13	rSI/r14	rDI/r15
6	PUSH[d64] Iz	IMUL Gv, Ev, Iz	PUSH[d64] Ib	IMUL Gv, Ev, Ib	INS/ INSB Yb, DX	INS/ INSW/ INSD Yz, DX	OUTS/ OUTSB DX, Xb	OUTS/ OUTSW/ OUTSD DX, Xz
7	Jcc[f64], 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[1A] POP[d64] Ev
9	CBW/ CWDE/ CDQE	CWD/ CDQ/ CQO	far CALL[i64] Ap	FWAIT/ WAIT	PUSHF/D/Q [d64]/ Fv	POPF/D/Q [d64]/ Fv	SAHF	LAHF
A	TEST AL, Ib	rAX, Iz	STOS/B Yb, AL	STOS/W/D/Q Yv, rAX	LODS/B AL, Xb	LODS/W/D/Q rAX, Xv	SCAS/B AL, Yb	SCAS/W/D/Q rAX, Yv
B	MOV immediate word or double into word, double, or quad register rAX/r8, Iv	rCX/r9, Iv	rDX/r10, Iv	rBX/r11, Iv	rSP/r12, Iv	rBP/r13, Iv	rSI/r14, Iv	rDI/r15 , Iv
C	ENTER Iw, Ib	LEAVE[d64]	far RET Iw	far RET	INT3	INT Ib	INTO[i64]	IRET/D/Q
D	ESC (Escape to coprocessor instruction set)							
E	near CALL[f64] Jz	JMP near[f64] Jz	far[i64] Ap	short[f64] Jb	IN AL, DX	eAX, DX	OUT DX, AL	DX, eAX
F	CLC	STC	CLI	STI	CLD	STD	INC/DEC Grp 4[1A]	INC/DEC Grp 5[1A]

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) *

	pfx	0	1	2	3	4	5	6	7
0		Grp 6[1A]	Grp 7[1A]	LAR Gv, Ew	LSL Gv, Ew		SYSCALL[o64]	CLTS	SYSRET[o64]
1		vmovups Vps, Wps	vmovups Wps, Vps	vmovlps Vq, Hq, Mq vmovhlps Vq, Hq, Uq	vmovlps Mq, Vq	vunpcklps Vx, Hx, Wx	vunpckhps Vx, Hx, Wx	vmovhps[v1] Vdq, Hq, Mq vmovlhps Vdq, Hq, Uq	vmovhps[v1] Mq, Vq
	66	vmovupd Vpd, Wpd	vmovupd Wpd,Vpd	vmovlpd Vq, Hq, Mq	vmovlpd Mq, Vq	vunpcklpd Vx,Hx,Wx	vunpckhpd Vx,Hx,Wx	vmovhpd[v1] Vdq, Hq, Mq	vmovhpd[v1] Mq, Vq
	F3	vmovss Vx, Hx, Wss	vmovss Wss, Hx, Vss	vmovsldup Vx, Wx				vmovshdup Vx, Wx	
	F2	vmovsd Vx, Hx, Wsd	vmovsd Wsd, Hx, Vsd	vmovddup Vx, Wx					
2		MOV Rd, Cd	MOV Rd, Dd	MOV Cd, Rd	MOV Dd, Rd				
3		WRMSR	RDTSC	RDMSR	RDPMC	SYSENTER	SYSEXIT		GETSEC
4		CMOVcc, (Gv, Ev) - Conditional Move							
		O	NO	B/C/NAE	AE/NB/NC	E/Z	NE/NZ	BE/NA	A/NBE
5		vmovmskps Gy, Ups	vsqrtps Vps, Wps	vrsqrtps Vps, Wps	vrcpps Vps, Wps	vandps Vps, Hps, Wps	vandnps Vps, Hps, Wps	vorps Vps, Hps, Wps	vxorps Vps, Hps, Wps
	66	vmovmskpd Gy,Upd	vsqrtpd Vpd, Wpd			vandpd Vpd, Hpd, Wpd	vandnpd Vpd, Hpd, Wpd	vorpd Vpd, Hpd, Wpd	vxorpd Vpd, Hpd, Wpd
	F3		vsqrtss Vss, Hss, Wss	vrsqrtss Vss, Hss, Wss	vrcpss Vss, Hss, Wss				
	F2		vsqrtsd Vsd, Hsd, Wsd						
6		punpcklbw Pq, Qd	punpcklwd Pq, Qd	punpckldq Pq, Qd	packsswb Pq, Qq	pcmpgtb Pq, Qq	pcmpgtw Pq, Qq	pcmpgtd Pq, Qq	packuswb Pq, Qq
	66	vpunpcklbw Vx, Hx, Wx	vpunpcklwd Vx, Hx, Wx	vpunpckldq Vx, Hx, Wx	vpacksswb Vx, Hx, Wx	vpcmpgtb Vx, Hx, Wx	vpcmpgtw Vx, Hx, Wx	vpcmpgtd Vx, Hx, Wx	vpackuswb Vx, Hx, Wx
	F3								
7		pshufw Pq, Qq, Ib	(Grp 12[1A])	(Grp 13[1A])	(Grp 14[1A])	pcmpeqb Pq, Qq	pcmpeqw Pq, Qq	pcmpeqd Pq, Qq	emms vzeroupper[v] vzeroall[v]
	66	vpshufd Vx, Wx, Ib				vpcmpeqb Vx, Hx, Wx	vpcmpeqw Vx, Hx, Wx	vpcmpeqd Vx, Hx, Wx	
	F3	vpshufhw Vx, Wx, Ib							
	F2	vpshuflw Vx, Wx, Ib							

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

	pfx	8	9	A	B	C	D	E	F
0		INVD	WBINVD		2-byte Illegal Opcodes UD2[1B]		prefetchw(/1) Ev		
1		Prefetch[1C] (Grp 16[1A])	Reserved-NOP	bndldx	bndstx	Reserved-NOP			NOP /0 Ev
	66			bndmov	bndmov				
	F3			bndcl	bndmk				
	F2			bndcu	bndcn				
2		vmovaps Vps, Wps	vmovaps Wps, Vps	cvtpi2ps Vps, Qpi	vmovntps Mps, Vps	cvttps2pi Ppi, Wps	cvtps2pi Ppi, Wps	vucomiss Vss, Wss	vcomiss Vss, Wss
	66	vmovapd Vpd, Wpd	vmovapd Wpd,Vpd	cvtpi2pd Vpd, Qpi	vmovntpd Mpd, Vpd	cvttpd2pi Ppi, Wpd	cvtpd2pi Qpi, Wpd	vucomisd Vsd, Wsd	vcomisd Vsd, Wsd
	F3			vcvtsi2ss Vss, Hss, Ey		vcvttss2si Gy, Wss	vcvtss2si Gy, Wss		
	F2			vcvtsi2sd Vsd, Hsd, Ey		vcvttsd2si Gy, Wsd	vcvtsd2si Gy, Wsd		
3		3-byte escape (Table A-4)		3-byte escape (Table A-5)					
4		CMOVcc(Gv, Ev) - Conditional Move							
		S	NS	P/PE	NP/PO	L/NGE	NL/GE	LE/NG	NLE/G
5		vaddps Vps, Hps, Wps	vmulps Vps, Hps, Wps	vcvtps2pd Vpd, Wps	vcvtdq2ps Vps, Wdq	vsubps Vps, Hps, Wps	vminps Vps, Hps, Wps	vdivps Vps, Hps, Wps	vmaxps Vps, Hps, Wps
	66	vaddpd Vpd, Hpd, Wpd	vmulpd Vpd, Hpd, Wpd	vcvtpd2ps Vps, Wpd	vcvtps2dq Vdq, Wps	vsubpd Vpd, Hpd, Wpd	vminpd Vpd, Hpd, Wpd	vdivpd Vpd, Hpd, Wpd	vmaxpd Vpd, Hpd, Wpd
	F3	vaddss Vss, Hss, Wss	vmulss Vss, Hss, Wss	vcvtss2sd Vsd, Hx, Wss	vcvttps2dq Vdq, Wps	vsubss Vss, Hss, Wss	vminss Vss, Hss, Wss	vdivss Vss, Hss, Wss	vmaxss Vss, Hss, Wss
	F2	vaddsd Vsd, Hsd, Wsd	vmulsd Vsd, Hsd, Wsd	vcvtsd2ss Vss, Hx, Wsd		vsubsd Vsd, Hsd, Wsd	vminsd Vsd, Hsd, Wsd	vdivsd Vsd, Hsd, Wsd	vmaxsd Vsd, Hsd, Wsd
6		punpckhbw Pq, Qd	punpckhwd Pq, Qd	punpckhdq Pq, Qd	packssdw Pq, Qd			movd/q Pd, Ey	movq Pq, Qq
	66	vpunpckhbw Vx, Hx, Wx	vpunpckhwd Vx, Hx, Wx	vpunpckhdq Vx, Hx, Wx	vpackssdw Vx, Hx, Wx	vpunpcklqdq Vx, Hx, Wx	vpunpckhqdq Vx, Hx, Wx	vmovd/q Vy, Ey	vmovdqa Vx, Wx
	F3								vmovdqu Vx, Wx
7		VMREAD Ey, Gy	VMWRITE Gy, Ey					movd/q Ey, Pd	movq Qq, Pq
	66					vhaddpd Vpd, Hpd, Wpd	vhsubpd Vpd, Hpd, Wpd	vmovd/q Ey, Vy	vmovdqa Wx,Vx
	F3							vmovq Vq, Wq	vmovdqu Wx,Vx
	F2					vhaddps Vps, Hps, Wps	vhsubps Vps, Hps, Wps		

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

	pfx	0	1	2	3	4	5	6	7
8		Jcc^f64, Jz - Long-displacement jump on condition							
		O	NO	B/CNAE	AE/NB/NC	E/Z	NE/NZ	BE/NA	A/NBE
9		SETcc, Eb - Byte Set on condition							
		O	NO	B/C/NAE	AE/NB/NC	E/Z	NE/NZ	BE/NA	A/NBE
A		PUSH^d64 FS	POP^d64 FS	CPUID	BT Ev, Gv	SHLD Ev, Gv, Ib	SHLD Ev, Gv, CL		
B		CMPXCHG		LSS Gv, Mp	BTR Ev, Gv	LFS Gv, Mp	LGS Gv, Mp	MOVZX	
		Eb, Gb	Ev, Gv					Gv, Eb	Gv, Ew
C		XADD Eb, Gb	XADD Ev, Gv	vcmpps Vps,Hps,Wps,Ib	movnti My, Gy	pinsrw Pq,Ry/Mw,Ib	pextrw Gd, Nq, Ib	vshufps Vps,Hps,Wps,Ib	Grp 9^1A
	66			vcmppd 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			vcmpsd Vsd,Hsd,Wsd,Ib					
D			psrlw Pq, Qq	psrld Pq, Qq	psrlq Pq, Qq	paddq Pq, Qq	pmullw Pq, Qq		pmovmskb Gd, Nq
	66	vaddsubpd Vpd, Hpd, Wpd	vpsrlw Vx, Hx, Wx	vpsrld Vx, Hx, Wx	vpsrlq Vx, Hx, Wx	vpaddq Vx, Hx, Wx	vpmullw Vx, Hx, Wx	vmovq Wq, Vq	vpmovmskb Gd, Ux
	F3							movq2dq Vdq, Nq	
	F2	vaddsubps Vps, Hps, Wps						movdq2q Pq, Uq	
E		pavgb Pq, Qq	psraw Pq, Qq	psrad Pq, Qq	pavgw Pq, Qq	pmulhuw 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	vpmulhuw Vx, Hx, Wx	vpmulhw Vx, Hx, Wx	vcvttpd2dq Vx, Wpd	vmovntdq Mx, Vx
	F3							vcvtdq2pd Vx, Wpd	
	F2							vcvtpd2dq Vx, Wpd	
F			psllw Pq, Qq	pslld Pq, Qq	psllq Pq, Qq	pmuludq Pq, Qq	pmaddwd Pq, Qq	psadbw Pq, Qq	maskmovq Pq, Nq
	66		vpsllw Vx, Hx, Wx	vpslld Vx, Hx, Wx	vpsllq Vx, Hx, Wx	vpmuludq Vx, Hx, Wx	vpmaddwd Vx, Hx, Wx	vpsadbw Vx, Hx, Wx	vmaskmovdqu Vdq, Udq
	F2	vlddqu Vx, Mx							

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

	pfx	8	9	A	B	C	D	E	F
8		Jcc^f64, 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^d64 GS	POP^d64 GS	RSM	BTS Ev, Gv	SHRD Ev, Gv, Ib	SHRD Ev, Gv, CL	(Grp 15^1A)^1C	IMUL Gv, Ev
B		*JMPE* (reserved for emulator on IPF)	Grp 10^1A Invalid Opcode^1B	Grp 8^1A Ev, Ib	BTC Ev, Gv	BSF Gv, Ev	BSR Gv, Ev	MOVSX Gv, Eb	Gv, Ew
	F3	POPCNT Gv, Ev				TZCNT Gv, Ev	LZCNT Gv, Ev		
C		BSWAP RAX/EAX/ R8/R8D	RCX/ECX/ R9/R9D	RDX/EDX/ R10/R10D	RBX/EBX/ R11/R11D	RSP/ESP/ R12/R12D	RBP/EBP/ R13/R13D	RSI/ESI/ R14/R14D	RDI/EDI/ R15/R15D
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	UD0
	66	vpsubb Vx, Hx, Wx	vpsubw Vx, Hx, Wx	vpsubd Vx, Hx, Wx	vpsubq Vx, Hx, Wx	vpaddb Vx, Hx, Wx	vpaddw Vx, Hx, Wx	vpaddd Vx, Hx, Wx	
	F2								

NOTES:

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

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

	pfx	0	1	2	3	4	5	6	7
0		pshufb Pq, Qq	phaddw Pq, Qq	phaddd 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	vphaddd 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[v] Vx, Wx, Ib	blendvps Vdq, Wdq	blendvpd Vdq, Wdq	vpermps[v] Vqq, Hqq, Wqq	vptest Vx, Wx
2	66	vpmovsxbw Vx, Ux/Mq	vpmovsxbd Vx, Ux/Md	vpmovsxbq Vx, Ux/Mw	vpmovsxwd Vx, Ux/Mq	vpmovsxwq Vx, Ux/Md	vpmovsxdq Vx, Ux/Mq		
3	66	vpmovzxbw Vx, Ux/Mq	vpmovzxbd Vx, Ux/Md	vpmovzxbq Vx, Ux/Mw	vpmovzxwd Vx, Ux/Mq	vpmovzxwq Vx, Ux/Md	vpmovzxdq Vx, Ux/Mq	vpermd[v] Vqq, Hqq, Wqq	vpcmpgtq Vx, Hx, Wx
4	66	vpmulld Vx, Hx, Wx	vphminposuw Vdq, Wdq				vpsrlvd/q[v] Vx, Hx, Wx	vpsravd[v] Vx, Hx, Wx	vpsllvd/q[v] Vx, Hx, Wx
5									
6									
7									
8	66	INVEPT Gy, Mdq	INVVPID Gy, Mdq	INVPCID Gy, Mdq					
9	66	vgatherdd/q[v] Vx,Hx,Wx	vgatherqd/q[v] Vx,Hx,Wx	vgatherdps/d[v] Vx,Hx,Wx	vgatherqps/d[v] Vx,Hx,Wx			vfmaddsub132ps/d[v] Vx,Hx,Wx	vfmsubadd132ps/d[v] Vx,Hx,Wx
A	66							vfmaddsub213ps/d[v] Vx,Hx,Wx	vfmsubadd213ps/d[v] Vx,Hx,Wx
B	66							vfmaddsub231ps/d[v] Vx,Hx,Wx	vfmsubadd231ps/d[v] Vx,Hx,Wx
C									
D									
E									
F		MOVBE Gy, My	MOVBE My, Gy	ANDN[v] Gy, By, Ey			BZHI[v] Gy, Ey, By		BEXTR[v] Gy, Ey, By
	66	MOVBE Gw, Mw	MOVBE Mw, Gw					ADCX Gy, Ey	SHLX[v] Gy, Ey, By
	F3				Grp 17[1A]		PEXT[v] Gy, By, Ey	ADOX Gy, Ey	SARX[v] Gy, Ey, By
	F2	CRC32 Gd, Eb	CRC32 Gd, Ey				PDEP[v] Gy, By, Ey	MULX[v] By,Gy,rDX,Ey	SHRX[v] 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) *

	pfx	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^{v} Vx,Hx,Wx	vpermilpd^{v} Vx,Hx,Wx	vtestps^{v} Vx, Wx	vtestpd^{v} Vx, Wx
1						pabsb Pq, Qq	pabsw Pq, Qq	pabsd Pq, Qq	
	66	vbroadcastss^{v} Vx, Wd	vbroadcastsd^{v} Vqq, Wq	vbroadcastf128^{v} Vqq, Mdq		vpabsb Vx, Wx	vpabsw Vx, Wx	vpabsd Vx, Wx	
2	66	vpmuldq Vx, Hx, Wx	vpcmpeqq Vx, Hx, Wx	vmovntdqa Vx, Mx	vpackusdw Vx, Hx, Wx	vmaskmovps^{v} Vx,Hx,Mx	vmaskmovpd^{v} Vx,Hx,Mx	vmaskmovps^{v} Mx,Hx,Vx	vmaskmovpd^{v} Mx,Hx,Vx
3	66	vpminsb Vx, Hx, Wx	vpminsd 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	vpbroadcastd^{v} Vx, Wx	vpbroadcastq^{v} Vx, Wx	vbroadcasti128^{v} Vqq, Mdq					
6									
7	66	vpbroadcastb^{v} Vx, Wx	vpbroadcastw^{v} Vx, Wx						
8	66					vpmaskmovd/q^{v} Vx,Hx,Mx		vpmaskmovd/q^{v} Mx,Vx,Hx	
9	66	vfmadd132ps/d^{v} Vx, Hx, Wx	vfmadd132ss/d^{v} Vx, Hx, Wx	vfmsub132ps/d^{v} Vx, Hx, Wx	vfmsub132ss/d^{v} Vx, Hx, Wx	vfnmadd132ps/d^{v} Vx, Hx, Wx	vfnmadd132ss/d^{v} Vx, Hx, Wx	vfnmsub132ps/d^{v} Vx, Hx, Wx	vfnmsub132ss/d^{v} Vx, Hx, Wx
A	66	vfmadd213ps/d^{v} Vx, Hx, Wx	vfmadd213ss/d^{v} Vx, Hx, Wx	vfmsub213ps/d^{v} Vx, Hx, Wx	vfmsub213ss/d^{v} Vx, Hx, Wx	vfnmadd213ps/d^{v} Vx, Hx, Wx	vfnmadd213ss/d^{v} Vx, Hx, Wx	vfnmsub213ps/d^{v} Vx, Hx, Wx	vfnmsub213ss/d^{v} Vx, Hx, Wx
B	66	vfmadd231ps/d^{v} Vx, Hx, Wx	vfmadd231ss/d^{v} Vx, Hx, Wx	vfmsub231ps/d^{v} Vx, Hx, Wx	vfmsub231ss/d^{v} Vx, Hx, Wx	vfnmadd231ps/d^{v} Vx, Hx, Wx	vfnmadd231ss/d^{v} Vx, Hx, Wx	vfnmsub231ps/d^{v} Vx, Hx, Wx	vfnmsub231ss/d^{v} Vx, Hx, Wx
C		sha1nexte Vdq,Wdq	sha1msg1 Vdq,Wdq	sha1msg2 Vdq,Wdq	sha256rnds2 Vdq,Wdq	sha256msg1 Vdq,Wdq	sha256msg2 Vdq,Wdq		
	66								
D	66				VAESIMC Vdq, Wdq	VAESENC Vdq,Hdq,Wdq	VAESENCLAST 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) *

	pfx	0	1	2	3	4	5	6	7
0	66	vpermq[v] Vqq, Wqq, Ib	vpermpd[v] Vqq, Wqq, Ib	vpblendd[v] Vx,Hx,Wx,Ib		vpermilps[v] Vx, Wx, Ib	vpermilpd[v] Vx, Wx, Ib	vperm2f128[v] Vqq,Hqq,Wqq,Ib	
1	66					vpextrb Rd/Mb, Vdq, Ib	vpextrw Rd/Mw, Vdq, Ib	vpextrd/q Ey, Vdq, Ib	vextractps Ed, Vdq, Ib
2	66	vpinsrb Vdq,Hdq,Ry/Mb,Ib	vinsertps Vdq,Hdq,Udq/Md,Ib	vpinsrd/q Vdq,Hdq,Ey,Ib					
3									
4	66	vdpps Vx,Hx,Wx,Ib	vdppd Vdq,Hdq,Wdq,Ib	vmpsadbw Vx,Hx,Wx,Ib		vpclmulqdq Vdq,Hdq,Wdq,Ib		vperm2i128[v] Vqq,Hqq,Wqq,Ib	
5									
6	66	vpcmpestrm Vdq, Wdq, Ib	vpcmpestri Vdq, Wdq, Ib	vpcmpistrm Vdq, Wdq, Ib	vpcmpistri Vdq, Wdq, Ib				
7									
8									
9									
A									
B									
C									
D									
E									
F	F2	RORX[v] Gy, Ey, Ib							

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

	pfx	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	vinsertf128[v] Vqq,Hqq,Wqq,Ib	vextractf128[v] Wdq,Vqq,Ib				vcvtps2ph[v] Wx, Vx, Ib		
2									
3	66	vinserti128[v] Vqq,Hqq,Wqq,Ib	vextracti128[v] Wdq,Vqq,Ib						
4	66			vblendvps[v] Vx,Hx,Wx,Lx	vblendvpd[v] Vx,Hx,Wx,Lx	vpblendvb[v] Vx,Hx,Wx,Lx			
5									
6									
7									
8									
9									
A									
B									
C						sha1rnds4 Vdq,Wdq,Ib			
D	66								VAESKEYGEN Vdq, Wdq, Ib
E									
F									

NOTES:

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

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

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

mod	nnn	R/M

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

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

A.4.1 Opcode Look-up Examples Using Opcode Extensions

An Example is provided below.

Example A-4. Interpreting an ADD Instruction

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

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

Example A-5. Looking Up 0F01C3H

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

- 0F indicates 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 Ib/Iz		NOT	NEG	MUL AL/rAX	IMUL AL/rAX	DIV AL/rAX	IDIV AL/rAX
FE	4	mem, 11B		INC Eb	DEC Eb						
FF	5	mem, 11B		INC Ev	DEC Ev	near CALL[f64] Ev	far CALL Ep	near JMP[f64] Ev	far JMP Mp	PUSH[d64] Ev	
0F 00	6	mem, 11B		SLDT Rv/Mw	STR Rv/Mw	LLDT Ew	LTR Ew	VERR Ew	VERW Ew		
0F 01	7	mem		SGDT Ms	SIDT Ms	LGDT Ms	LIDT Ms	SMSW Mw/Rv		LMSW Ew	INVLPG Mb
		11B		VMCALL (001) VMLAUNCH (010) VMRESUME (011) VMXOFF (100)	MONITOR (000) MWAIT (001) CLAC (010) STAC (011) ENCLS (111)	XGETBV (000) XSETBV (001) VMFUNC (100) XEND (101) XTEST (110) ENCLU(111)					SWAPGS [o64](000) RDTSCP (001)
0F BA	8	mem, 11B						BT	BTS	BTR	BTC
0F C7	9	mem			CMPXCH8B Mq CMPXCHG16B Mdq					VMPTRLD Mq	VMPTRST Mq
			66							VMCLEAR Mq	
			F3							VMXON Mq	
		11B								RDRAND Rv	RDSEED Rv
			F3								RDPID Rd/q
0F B9	10	mem		UD1							
		11B									
C6	11	mem		MOV Eb, Ib							
		11B									XABORT (000) Ib
C7		mem		MOV Ev, Iz							
		11B									XBEGIN (000) Jz
0F 71	12	mem									
		11B				psrlw Nq, Ib		psraw Nq, Ib		psllw Nq, Ib	
			66			vpsrlw Hx,Ux,Ib		vpsraw Hx,Ux,Ib		vpsllw Hx,Ux,Ib	
0F 72	13	mem									
		11B				psrld Nq, Ib		psrad Nq, Ib		pslld Nq, Ib	
			66			vpsrld Hx,Ux,Ib		vpsrad Hx,Ux,Ib		vpslld Hx,Ux,Ib	
0F 73	14	mem									
		11B				psrlq Nq, Ib				psllq Nq, Ib	
			66			vpsrlq Hx,Ux,Ib	vpsrldq Hx,Ux,Ib			vpsllq Hx,Ux,Ib	vpslldq Hx,Ux,Ib

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

Opcode	Group	Mod 7,6	pfx	Encoding of Bits 5,4,3 of the ModR/M Byte (bits 2,1,0 in parenthesis)							
				000	001	010	011	100	101	110	111
0F AE	15	mem		fxsave	fxrstor	ldmxcsr	stmxcsr	XSAVE	XRSTOR	XSAVEOPT	clflush
		11B							lfence	mfence	sfence
			F3	RDFSBASE Ry	RDGSBASE Ry	WRFSBASE Ry	WRGSBASE Ry				
0F 18	16	mem		prefetch NTA	prefetch T0	prefetch T1	prefetch T2	Reserved NOP			
		11B		Reserved NOP							
VEX.0F38 F3	17	mem			BLSR[v] By, Ey	BLSMSK[v] By, Ey	BLSI[v] 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-12 shows the map if the ModR/M byte is outside the range of 00H-BFH. Here, the first digit of the ModR/M byte selects the table row and the second digit selects the column.

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

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

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

NOTES:

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

A.5.2.4 Escape Opcodes with DB as First Byte

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

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

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

NOTES:

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

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

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

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

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

NOTES:

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

A.5.2.5 Escape Opcodes with DC as First Byte

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

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

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

NOTES:

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

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

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

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

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

NOTES:

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

A.5.2.6 Escape Opcodes with DD as First Byte

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

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

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

NOTES:

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

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

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

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

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

NOTES:

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

A.5.2.7 Escape Opcodes with DE as First Byte

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

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

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

NOTES:

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

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

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

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

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

NOTES:

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

A.5.2.8 Escape Opcodes with DF As First Byte

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

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

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

NOTES:

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

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

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

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

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

NOTES:

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

This page was intentionally left blank.

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)

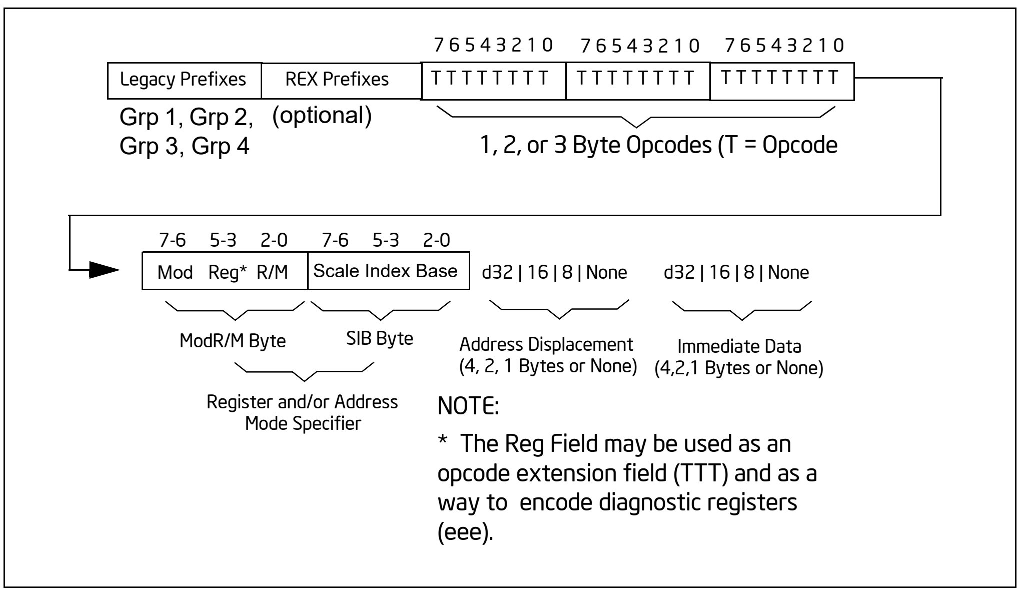

Figure B-1. General Machine Instruction Format

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 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
tttn	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		
reg	Function of w Field	
	When w = 0	When w = 1
000	AL	AX
001	CL	CX
010	DL	DX
011	BL	BX
100	AH	SP
101	CH	BP
110	DH	SI
111	BH	DI

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

Register Specified by reg Field During 32-Bit Data Operations		
reg	Function of w Field	
	When w = 0	When w = 1
000	AL	EAX
001	CL	ECX
010	DL	EDX
011	BL	EBX
100	AH*	ESP
101	CH*	EBP
110	DH*	ESI
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

s	Effect on 8-Bit Immediate Data	Effect on 16- or 32-Bit Immediate Data
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

2-Bit sreg2 Field	Segment Register Selected
00	ES
01	CS
10	SS
11	DS

3-Bit sreg3 Field	Segment Register Selected
000	ES
001	CS
010	SS
011	DS
100	FS
101	GS
110	Reserved[1]
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 though 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[1]	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 (tttn) Field

For conditional instructions (such as conditional jumps and set on condition), the condition test field (tttn) 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 tttn field is located in bits 3, 2, 1, and 0 of the opcode byte.
- For 2-byte primary opcodes, the tttn field is located in bits 3, 2, 1, and 0 of the second opcode byte.

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

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

t t t n	Mnemonic	Condition
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

d	Source	Destination
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
AAA – ASCII Adjust after Addition	0011 0111
AAD – ASCII Adjust AX before Division	1101 0101 : 0000 1010
AAM – ASCII Adjust AX after Multiply	1101 0100 : 0000 1010
AAS – ASCII Adjust AL after Subtraction	0011 1111
ADC – ADD with Carry	
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
ADD – Add	
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
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
AND – Logical AND	
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

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

Instruction and Format	Encoding
ARPL – Adjust RPL Field of Selector	
from register	0110 0011 : 11 reg1 reg2
from memory	0110 0011 : mod reg r/m
BOUND – Check Array Against Bounds	0110 0010 : mod[A] reg r/m
BSF – Bit Scan Forward	
register1, register2	0000 1111 : 1011 1100 : 11 reg1 reg2
memory, register	0000 1111 : 1011 1100 : mod reg r/m
BSR – Bit Scan Reverse	
register1, register2	0000 1111 : 1011 1101 : 11 reg1 reg2
memory, register	0000 1111 : 1011 1101 : mod reg r/m
BSWAP – Byte Swap	0000 1111 : 1100 1 reg
BT – Bit Test	
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
BTC – Bit Test and Complement	
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
BTR – Bit Test and Reset	
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
BTS – Bit Test and Set	
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
CALL – Call Procedure (in same segment)	
direct	1110 1000 : full displacement
register indirect	1111 1111 : 11 010 reg
memory indirect	1111 1111 : mod 010 r/m
CALL – Call Procedure (in other segment)	
direct	1001 1010 : unsigned full offset, selector
indirect	1111 1111 : mod 011 r/m

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

Instruction and Format	Encoding
CBW – Convert Byte to Word	1001 1000
CDQ – Convert Doubleword to Qword	1001 1001
CLC – Clear Carry Flag	1111 1000
CLD – Clear Direction Flag	1111 1100
CLI – Clear Interrupt Flag	1111 1010
CLTS – Clear Task-Switched Flag in CR0	0000 1111 : 0000 0110
CMC – Complement Carry Flag	1111 0101
CMP – Compare Two Operands	
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
CMPS/CMPSB/CMPSW/CMPSD – Compare String Operands	1010 011w
CMPXCHG – Compare and Exchange	
register1, register2	0000 1111 : 1011 000w : 11 reg2 reg1
memory, register	0000 1111 : 1011 000w : mod reg r/m
CPUID – CPU Identification	0000 1111 : 1010 0010
CWD – Convert Word to Doubleword	1001 1001
CWDE – Convert Word to Doubleword	1001 1000
DAA – Decimal Adjust AL after Addition	0010 0111
DAS – Decimal Adjust AL after Subtraction	0010 1111
DEC – Decrement by 1	
register	1111 111w : 11 001 reg
register (alternate encoding)	0100 1 reg
memory	1111 111w : mod 001 r/m
DIV – Unsigned Divide	
AL, AX, or EAX by register	1111 011w : 11 110 reg
AL, AX, or EAX by memory	1111 011w : mod 110 r/m
HLT – Halt	1111 0100
IDIV – Signed Divide	
AL, AX, or EAX by register	1111 011w : 11 111 reg
AL, AX, or EAX by memory	1111 011w : mod 111 r/m

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

Instruction and Format	Encoding
IMUL – Signed Multiply	
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
IN – Input From Port	
fixed port	1110 010w : port number
variable port	1110 110w
INC – Increment by 1	
reg	1111 111w : 11 000 reg
reg (alternate encoding)	0100 0 reg
memory	1111 111w : mod 000 r/m
INS – Input from DX Port	0110 110w
INT n – Interrupt Type n	1100 1101 : type
INT – Single-Step Interrupt 3	1100 1100
INTO – Interrupt 4 on Overflow	1100 1110
INVD – Invalidate Cache	0000 1111 : 0000 1000
INVLPG – Invalidate TLB Entry	0000 1111 : 0000 0001 : mod 111 r/m
INVPCID – Invalidate Process-Context Identifier	0110 0110:0000 1111:0011 1000:1000 0010: mod reg r/m
IRET/IRETD – Interrupt Return	1100 1111
***Jcc* – Jump if Condition is Met**	
8-bit displacement	0111 tttn : 8-bit displacement
full displacement	0000 1111 : 1000 tttn : full displacement
JCXZ/JECXZ – Jump on CX/ECX Zero Address-size prefix differentiates JCXZ and JECXZ	1110 0011 : 8-bit displacement
JMP – Unconditional Jump (to same segment)	
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
JMP – Unconditional Jump (to other segment)	
direct intersegment	1110 1010 : unsigned full offset, selector
indirect intersegment	1111 1111 : mod 101 r/m
LAHF – Load Flags into AHRegister	1001 1111

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

Instruction and Format	Encoding
LAR – Load Access Rights Byte	
from register	0000 1111 : 0000 0010 : 11 reg1 reg2
from memory	0000 1111 : 0000 0010 : mod reg r/m
LDS – Load Pointer to DS	1100 0101 : mod^{A,B} reg r/m
LEA – Load Effective Address	1000 1101 : mod^{A} reg r/m
LEAVE – High Level Procedure Exit	1100 1001
LES – Load Pointer to ES	1100 0100 : mod^{A,B} reg r/m
LFS – Load Pointer to FS	0000 1111 : 1011 0100 : mod^{A} reg r/m
LGDT – Load Global Descriptor Table Register	0000 1111 : 0000 0001 : mod^{A} 010 r/m
LGS – Load Pointer to GS	0000 1111 : 1011 0101 : mod^{A} reg r/m
LIDT – Load Interrupt Descriptor Table Register	0000 1111 : 0000 0001 : mod^{A} 011 r/m
LLDT – Load Local Descriptor Table Register	
LDTR from register	0000 1111 : 0000 0000 : 11 010 reg
LDTR from memory	0000 1111 : 0000 0000 : mod 010 r/m
LMSW – Load Machine Status Word	
from register	0000 1111 : 0000 0001 : 11 110 reg
from memory	0000 1111 : 0000 0001 : mod 110 r/m
LOCK – Assert LOCK# Signal Prefix	1111 0000
LODS/LODSB/LODSW/LODSD – Load String Operand	1010 110w
LOOP – Loop Count	1110 0010 : 8-bit displacement
LOOPZ/LOOPE – Loop Count while Zero/Equal	1110 0001 : 8-bit displacement
LOOPNZ/LOOPNE – Loop Count while not Zero/Equal	1110 0000 : 8-bit displacement
LSL – Load Segment Limit	
from register	0000 1111 : 0000 0011 : 11 reg1 reg2
from memory	0000 1111 : 0000 0011 : mod reg r/m
LSS – Load Pointer to SS	0000 1111 : 1011 0010 : mod^{A} reg r/m
LTR – Load Task Register	
from register	0000 1111 : 0000 0000 : 11 011 reg
from memory	0000 1111 : 0000 0000 : mod 011 r/m
MOV – Move Data	
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

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

Instruction and Format	Encoding
AL, AX, or EAX to memory	1010 001w : full displacement
MOV – Move to/from Control Registers	
CR0 from register	0000 1111 : 0010 0010 : -- 000 reg
CR2 from register	0000 1111 : 0010 0010 : -- 010reg
CR3 from register	0000 1111 : 0010 0010 : -- 011 reg
CR4 from register	0000 1111 : 0010 0010 : -- 100 reg
register from CR0-CR4	0000 1111 : 0010 0000 : -- eee reg
MOV – Move to/from Debug Registers	
DR0-DR3 from register	0000 1111 : 0010 0011 : -- eee reg
DR4-DR5 from register	0000 1111 : 0010 0011 : -- eee reg
DR6-DR7 from register	0000 1111 : 0010 0011 : -- eee reg
register from DR6-DR7	0000 1111 : 0010 0001 : -- eee reg
register from DR4-DR5	0000 1111 : 0010 0001 : -- eee reg
register from DR0-DR3	0000 1111 : 0010 0001 : -- eee reg
MOV – Move to/from Segment Registers	
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
MOVBE – Move data after swapping bytes	
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
MOVS/MOVSB/MOVSW/MOVSD – Move Data from String to String	1010 010w
MOVSX – Move with Sign-Extend	
memory to reg	0000 1111 : 1011 111w : mod reg r/m
MOVZX – Move with Zero-Extend	
register2 to register1	0000 1111 : 1011 011w : 11 reg1 reg2
memory to register	0000 1111 : 1011 011w : mod reg r/m
MUL – Unsigned Multiply	
AL, AX, or EAX with register	1111 011w : 11 100 reg
AL, AX, or EAX with memory	1111 011w : mod 100 r/m
NEG – Two's Complement Negation	
register	1111 011w : 11 011 reg
memory	1111 011w : mod 011 r/m
NOP – No Operation	1001 0000

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

Instruction and Format	Encoding
NOP – Multi-byte No Operation[1]	
register	0000 1111 0001 1111 : 11 000 reg
memory	0000 1111 0001 1111 : mod 000 r/m
NOT – One's Complement Negation	
register	1111 011w : 11 010 reg
memory	1111 011w : mod 010 r/m
OR – Logical Inclusive OR	
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
OUT – Output to Port	
fixed port	1110 011w : port number
variable port	1110 111w
OUTS – Output to DX Port	0110 111w
POP – Pop a Word from the Stack	
register	1000 1111 : 11 000 reg
register (alternate encoding)	0101 1 reg
memory	1000 1111 : mod 000 r/m
POP – Pop a Segment Register from the Stack (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
POPA/POPAD – Pop All General Registers	0110 0001
POPF/POPFD – Pop Stack into FLAGS or EFLAGS Register	1001 1101
PUSH – Push Operand onto the Stack	
register	1111 1111 : 11 110 reg
register (alternate encoding)	0101 0 reg
memory	1111 1111 : mod 110 r/m
immediate	0110 10s0 : immediate data
PUSH – Push Segment Register onto the Stack	
segment register CS,DS,ES,SS	000 sreg2 110
segment register FS,GS	0000 1111: 10 sreg3 000
PUSHA/PUSHAD – Push All General Registers	0110 0000

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

Instruction and Format	Encoding
PUSHF/PUSHFD – Push Flags Register onto the Stack	1001 1100
RCL – Rotate thru Carry Left	
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
RCR – Rotate thru Carry Right	
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
RDMSR – Read from Model-Specific Register	0000 1111 : 0011 0010
RDPMC – Read Performance Monitoring Counters	0000 1111 : 0011 0011
RDTSC – Read Time-Stamp Counter	0000 1111 : 0011 0001
RDTSCP – Read Time-Stamp Counter and Processor ID	0000 1111 : 0000 0001: 1111 1001
REP INS – Input String	1111 0011 : 0110 110w
REP LODS – Load String	1111 0011 : 1010 110w
REP MOVS – Move String	1111 0011 : 1010 010w
REP OUTS – Output String	1111 0011 : 0110 111w
REP STOS – Store String	1111 0011 : 1010 101w
REPE CMPS – Compare String	1111 0011 : 1010 011w
REPE SCAS – Scan String	1111 0011 : 1010 111w
REPNE CMPS – Compare String	1111 0010 : 1010 011w
REPNE SCAS – Scan String	1111 0010 : 1010 111w
RET – Return from Procedure (to same segment)	
no argument	1100 0011
adding immediate to SP	1100 0010 : 16-bit displacement
RET – Return from Procedure (to other segment)	
intersegment	1100 1011
adding immediate to SP	1100 1010 : 16-bit displacement

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

Instruction and Format	Encoding
ROL – Rotate Left	
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
ROR – Rotate Right	
register by 1	1101 000w : 11 001 reg
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
RSM – Resume from System Management Mode	0000 1111 : 1010 1010
SAHF – Store AH into Flags	1001 1110
SAL – Shift Arithmetic Left	same instruction as SHL
SAR – Shift Arithmetic Right	
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
SBB – Integer Subtraction with Borrow	
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
SCAS/SCASB/SCASW/SCASD – Scan String	1010 111w
SETcc – Byte Set on Condition	
register	0000 1111 : 1001 tttn : 11 000 reg
memory	0000 1111 : 1001 tttn : mod 000 r/m
SGDT – Store Global Descriptor Table Register	0000 1111 : 0000 0001 : mod^A 000 r/m

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

Instruction and Format	Encoding
SHL – Shift Left	
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
SHLD – Double Precision Shift Left	
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
SHR – Shift Right	
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
SHRD – Double Precision Shift Right	
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
SIDT – Store Interrupt Descriptor Table Register	0000 1111 : 0000 0001 : mod[A] 001 r/m
SLDT – Store Local Descriptor Table Register	
to register	0000 1111 : 0000 0000 : 11 000 reg
to memory	0000 1111 : 0000 0000 : mod 000 r/m
SMSW – Store Machine Status Word	
to register	0000 1111 : 0000 0001 : 11 100 reg
to memory	0000 1111 : 0000 0001 : mod 100 r/m
STC – Set Carry Flag	1111 1001
STD – Set Direction Flag	1111 1101
STI – Set Interrupt Flag	1111 1011
STOS/STOSB/STOSW/STOSD – Store String Data	1010 101w
STR – Store Task Register	
to register	0000 1111 : 0000 0000 : 11 001 reg
to memory	0000 1111 : 0000 0000 : mod 001 r/m

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

Instruction and Format	Encoding
SUB – Integer Subtraction	
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
TEST – Logical Compare	
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
UD0 – Undefined instruction	0000 1111 : 1111 1111
UD1 – Undefined instruction	0000 1111 : 0000 1011
UD2 – Undefined instruction	0000 FFFF : 0000 1011
VERR – Verify a Segment for Reading	
register	0000 1111 : 0000 0000 : 11 100 reg
memory	0000 1111 : 0000 0000 : mod 100 r/m
VERW – Verify a Segment for Writing	
register	0000 1111 : 0000 0000 : 11 101 reg
memory	0000 1111 : 0000 0000 : mod 101 r/m
WAIT – Wait	1001 1011
WBINVD – Writeback and Invalidate Data Cache	0000 1111 : 0000 1001
WRMSR – Write to Model-Specific Register	0000 1111 : 0011 0000
XADD – Exchange and Add	
register1, register2	0000 1111 : 1100 000w : 11 reg2 reg1
memory, reg	0000 1111 : 1100 000w : mod reg r/m
XCHG – Exchange Register/Memory with Register	
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
XLAT/XLATB – Table Look-up Translation	1101 0111
XOR – Logical Exclusive OR	
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

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

Instruction and Format	Encoding
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
Prefix Bytes	
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

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
ADC – ADD with Carry	
register1 to register2	0100 0R0B : 0001 000w : 11 reg1 reg2
qwordregister1 to qwordregister2	0100 1R0B : 0001 0001 : 11 qwordreg1 qwordreg2
register2 to register1	0100 0R0B : 0001 001w : 11 reg1 reg2
qwordregister1 to qwordregister2	0100 1R0B : 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
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

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

Instruction and Format	Encoding
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
ADD – Add	
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
immediate32 to memory64	0100 10XB : 1000 0001 : mod 010 r/m : imm32
immediate8 to memory64	0100 10XB : 1000 0011 : mod 010 r/m : imm8
AND – Logical AND	
register1 to register2	0100 0R0B 0010 000w : 11 reg1 reg2
qwordregister1 to qwordregister2	0100 1R0B 0010 0001 : 11 qwordreg1 qwordreg2
register2 to register1	0100 0R0B 0010 001w : 11 reg1 reg2
register1 to register2	0100 1R0B 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
BSF – Bit Scan Forward	

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

Instruction and Format	Encoding
register1, register2	0100 0R0B 0000 1111 : 1011 1100 : 11 reg1 reg2
qwordregister1, qwordregister2	0100 1R0B 0000 1111 : 1011 1100 : 11 qwordreg1 qwordreg2
memory, register	0100 0RXB 0000 1111 : 1011 1100 : mod reg r/m
memory64, qwordregister	0100 1RXB 0000 1111 : 1011 1100 : mod qwordreg r/m
BSR – Bit Scan Reverse	
register1, register2	0100 0R0B 0000 1111 : 1011 1101 : 11 reg1 reg2
qwordregister1, qwordregister2	0100 1R0B 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
BSWAP – Byte Swap	0000 1111 : 1100 1 reg
BSWAP – Byte Swap	0100 100B 0000 1111 : 1100 1 qwordreg
BT – Bit Test	
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 0R0B 0000 1111 : 1010 0011 : 11 reg2 reg1
qwordregister1, qwordregister2	0100 1R0B 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
BTC – Bit Test and Complement	
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 0R0B 0000 1111 : 1011 1011 : 11 reg2 reg1
qwordregister1, qwordregister2	0100 1R0B 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
BTR – Bit Test and Reset	
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 0R0B 0000 1111 : 1011 0011 : 11 reg2 reg1

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

Instruction and Format	Encoding
qwordregister1, qwordregister2	0100 1R0B 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
BTS – Bit Test and Set	
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
CALL – Call Procedure (in same segment)	
direct	1110 1000 : displacement32
register indirect	0100 WR00^{W} 1111 1111 : 11 010 reg
memory indirect	0100 W0XB^{W} 1111 1111 : mod 010 r/m
CALL – Call Procedure (in other segment)	
indirect	1111 1111 : mod 011 r/m
indirect	0100 10XB 0100 1000 1111 1111 : mod 011 r/m
CBW – Convert Byte to Word	1001 1000
CDQ – Convert Doubleword to Qword+	1001 1001
CDQE – RAX, Sign-Extend of EAX	0100 1000 1001 1001
CLC – Clear Carry Flag	1111 1000
CLD – Clear Direction Flag	1111 1100
CLI – Clear Interrupt Flag	1111 1010
CLTS – Clear Task-Switched Flag in CR0	0000 1111 : 0000 0110
CMC – Complement Carry Flag	1111 0101
CMP – Compare Two Operands	
register1 with register2	0100 0R0B 0011 100w : 11 reg1 reg2
qwordregister1 with qwordregister2	0100 1R0B 0011 1001 : 11 qwordreg1 qwordreg2
register2 with register1	0100 0R0B 0011 101w : 11 reg1 reg2
qwordregister2 with qwordregister1	0100 1R0B 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

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

Instruction and Format	Encoding
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
CMPS/CMPSB/CMPSW/CMPSD/CMPSQ – Compare String Operands	
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
CMPXCHG – Compare and Exchange	
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
memory8, byteregister	0100 00XB 0000 1111 : 1011 0000 : mod bytereg r/m
memory64, qwordregister	0100 10XB 0000 1111 : 1011 0001 : mod qwordreg r/m
CPUID – CPU Identification	0000 1111 : 1010 0010
CQO – Sign-Extend RAX	0100 1000 1001 1001
CWD – Convert Word to Doubleword	1001 1001
CWDE – Convert Word to Doubleword	1001 1000
DEC – Decrement by 1	
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
DIV – Unsigned Divide	
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
ENTER – Make Stack Frame for High Level Procedure	1100 1000 : 16-bit displacement : 8-bit level (L)
HLT – Halt	1111 0100
IDIV – Signed Divide	
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
IMUL – Signed Multiply	

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

Instruction and Format	Encoding
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
register1 with register2	0000 1111 : 1010 1111 : 11 : reg1 reg2
qwordregister1 := qwordregister1 with qwordregister2	0100 1R0B 0000 1111 : 1010 1111 : 11 : qwordreg1 qwordreg2
register with memory	0100 0RXB 0000 1111 : 1010 1111 : mod reg r/m
qwordregister := qwordregister with memory64	0100 1RXB 0000 1111 : 1010 1111 : mod qwordreg r/m
register1 with immediate to register2	0100 0R0B 0110 10s1 : 11 reg1 reg2 : imm
qwordregister1 := qwordregister2 with sign-extended immediate8	0100 1R0B 0110 1011 : 11 qwordreg1 qwordreg2 : imm8
qwordregister1 := qwordregister2 with immediate32	0100 1R0B 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
IN – Input From Port	
fixed port	1110 010w : port number
variable port	1110 110w
INC – Increment by 1	
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
INS – Input from DX Port	0110 110w
INT n – Interrupt Type n	1100 1101 : type
INT – Single-Step Interrupt 3	1100 1100
INTO – Interrupt 4 on Overflow	1100 1110
INVD – Invalidate Cache	0000 1111 : 0000 1000
INVLPG – Invalidate TLB Entry	0000 1111 : 0000 0001 : mod 111 r/m
INVPCID – Invalidate Process-Context Identifier	0110 0110:0000 1111:0011 1000:1000 0010: mod reg r/m
IRETO – Interrupt Return	1100 1111
***Jcc* – Jump if Condition is Met**	
8-bit displacement	0111 tttn : 8-bit displacement
displacements (excluding 16-bit relative offsets)	0000 1111 : 1000 tttn : displacement32
JCXZ/JECXZ – Jump on CX/ECX Zero	
Address-size prefix differentiates JCXZ and JECXZ	1110 0011 : 8-bit displacement
JMP – Unconditional Jump (to same segment)	
short	1110 1011 : 8-bit displacement

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

Instruction and Format	Encoding
direct	1110 1001 : displacement32
register indirect	0100 W00B^{W} : 1111 1111 : 11 100 reg
memory indirect	0100 W0XB^{W} : 1111 1111 : mod 100 r/m
JMP – Unconditional Jump (to other segment)	
indirect intersegment	0100 00XB : 1111 1111 : mod 101 r/m
64-bit indirect intersegment	0100 10XB : 1111 1111 : mod 101 r/m
LAR – Load Access Rights Byte	
from register	0100 0R0B : 0000 1111 : 0000 0010 : 11 reg1 reg2
from dwordregister to qwordregister, masked by 00FxFF00H	0100 WR0B : 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
LEA – Load Effective Address	
in wordregister/dwordregister	0100 0RXB : 1000 1101 : mod^{A} reg r/m
in qwordregister	0100 1RXB : 1000 1101 : mod^{A} qwordreg r/m
LEAVE – High Level Procedure Exit	1100 1001
LFS – Load Pointer to FS	
FS:r16/r32 with far pointer from memory	0100 0RXB : 0000 1111 : 1011 0100 : mod^{A} reg r/m
FS:r64 with far pointer from memory	0100 1RXB : 0000 1111 : 1011 0100 : mod^{A} qwordreg r/m
LGDT – Load Global Descriptor Table Register	0100 10XB : 0000 1111 : 0000 0001 : mod^{A} 010 r/m
LGS – Load Pointer to GS	
GS:r16/r32 with far pointer from memory	0100 0RXB : 0000 1111 : 1011 0101 : mod^{A} reg r/m
GS:r64 with far pointer from memory	0100 1RXB : 0000 1111 : 1011 0101 : mod^{A} qwordreg r/m
LIDT – Load Interrupt Descriptor Table Register	0100 10XB : 0000 1111 : 0000 0001 : mod^{A} 011 r/m
LLDT – Load Local Descriptor Table Register	
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
LMSW – Load Machine Status Word	
from register	0100 000B : 0000 1111 : 0000 0001 : 11 110 reg
from memory	0100 00XB :0000 1111 : 0000 0001 : mod 110 r/m
LOCK – Assert LOCK# Signal Prefix	1111 0000
LODS/LODSB/LODSW/LODSD/LODSQ – Load String Operand	
at DS:(E)SI to AL/EAX/EAX	1010 110w
at (R)SI to RAX	0100 1000 1010 1101
LOOP – Loop Count	
if count ≠ 0, 8-bit displacement	1110 0010
if count ≠ 0, RIP + 8-bit displacement sign-extended to 64-bits	0100 1000 1110 0010
LOOPE – Loop Count while Zero/Equal	

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
LOOPNE/LOOPNZ – Loop Count while not Zero/Equal	
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
LSL – Load Segment Limit	
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
LSS – Load Pointer to SS	
SS:r16/r32 with far pointer from memory	0100 0RXB : 0000 1111 : 1011 0010 : mod[A] reg r/m
SS:r64 with far pointer from memory	0100 1WXB : 0000 1111 : 1011 0010 : mod[A] qwordreg r/m
LTR – Load Task Register	
from register	0100 0R00 : 0000 1111 : 0000 0000 : 11 011 reg
from memory	0100 00XB : 0000 1111 : 0000 0000 : mod 011 r/m
MOV – Move Data	
register1 to register2	0100 0R0B : 1000 100w : 11 reg1 reg2
qwordregister1 to qwordregister2	0100 1R0B 1000 1001 : 11 qwordeg1 qwordreg2
register2 to register1	0100 0R0B : 1000 101w : 11 reg1 reg2
qwordregister2 to qwordregister1	0100 1R0B 1000 1011 : 11 qwordreg1 qwordreg2
memory to reg	0100 0RXB : 1000 101w : mod reg r/m
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
MOV – Move to/from Control Registers	
CR0-CR4 from register	0100 0R0B : 0000 1111 : 0010 0010 : 11 eee reg (eee = CR#)

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

Instruction and Format	Encoding
CRx from qwordregister	0100 1R0B : 0000 1111 : 0010 0010 : 11 eee qwordreg (Reee = CR#)
register from CR0-CR4	0100 0R0B : 0000 1111 : 0010 0000 : 11 eee reg (eee = CR#)
qwordregister from CRx	0100 1R0B 0000 1111 : 0010 0000 : 11 eee qwordreg (Reee = CR#)
MOV – Move to/from Debug Registers	
DR0-DR7 from register	0000 1111 : 0010 0011 : 11 eee reg (eee = DR#)
DR0-DR7 from quadregister	0100 10OB 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 10OB 0000 1111 : 0010 0001 : 11 eee quadreg (eee = DR#)
MOV – Move to/from Segment Registers	
register to segment register	0100 W00B^{W} : 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
MOVBE – Move data after swapping bytes	
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
MOVS/MOVSB/MOVSW/MOVSD/MOVSQ – Move Data from String to String	
Move data from string to string	1010 010w
Move data from string to string (qword)	0100 1000 1010 0101
MOVSX/MOVSXD – Move with Sign-Extend	
register2 to register1	0100 0R0B : 0000 1111 : 1011 111w : 11 reg1 reg2
byteregister2 to qwordregister1 (sign-extend)	0100 1R0B 0000 1111 : 1011 1110 : 11 quadreg1 bytereg2
wordregister2 to qwordregister1	0100 1R0B 0000 1111 : 1011 1111 : 11 quadreg1 wordreg2
dwordregister2 to qwordregister1	0100 1R0B 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
MOVZX – Move with Zero-Extend	

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

Instruction and Format	Encoding
register2 to register1	0100 0R0B : 0000 1111 : 1011 011w : 11 reg1 reg2
dwordregister2 to qwordregister1	0100 1R0B 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
MUL – Unsigned Multiply	
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
NEG – Two's Complement Negation	
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
NOP – No Operation	1001 0000
NOT – One's Complement Negation	
register	0100 000B : 1111 011w : 11 010 reg
qwordregister	0100 000B 1111 0111 : 11 010 qwordreg
memory	0100 00XB : 1111 011w : mod 010 r/m
memory64	0100 1RXB 1111 0111 : mod 010 r/m
OR – Logical Inclusive OR	
register1 to register2	0000 100w : 11 reg1 reg2
byteregister1 to byteregister2	0100 0R0B 0000 1000 : 11 bytereg1 bytereg2
qwordregister1 to qwordregister2	0100 1R0B 0000 1001 : 11 qwordreg1 qwordreg2
register2 to register1	0000 101w : 11 reg1 reg2
byteregister2 to byteregister1	0100 0R0B 0000 1010 : 11 bytereg1 bytereg2
qwordregister2 to qwordregister1	0100 0R0B 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

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

Instruction and Format	Encoding
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
OUT – Output to Port	
fixed port	1110 011w : port number
variable port	1110 111w
OUTS – Output to DX Port	
output to DX Port	0110 111w
POP – Pop a Value from the Stack	
wordregister	0101 0101 : 0100 000B : 1000 1111 : 11 000 reg16
qwordregister	0100 W00B^{S} : 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^{S} : 1000 1111 : mod 000 r/m
memory16	0101 0101 : 0100 00XB 1000 1111 : mod 000 r/m
POP – Pop a Segment Register from the Stack (Note: CS cannot be sreg2 in this usage.)	
segment register FS, GS	0000 1111: 10 sreg3 001
POPF/POPFQ – Pop Stack into FLAGS/RFLAGS Register	
pop stack to FLAGS register	0101 0101 : 1001 1101
pop Stack to RFLAGS register	0100 1000 1001 1101
PUSH – Push Operand onto the Stack	
wordregister	0101 0101 : 0100 000B : 1111 1111 : 11 110 reg16
qwordregister	0100 W00B^{S} : 1111 1111 : 11 110 reg64
wordregister (alternate encoding)	0101 0101 : 0100 000B : 0101 0 reg16
qwordregister (alternate encoding)	0100 W00B^{S} : 0101 0 reg64
memory16	0101 0101 : 0100 000B : 1111 1111 : mod 110 r/m
memory64	0100 W00B^{S} : 1111 1111 : mod 110 r/m
immediate8	0110 1010 : imm8
immediate16	0101 0101 : 0110 1000 : imm16
immediate64	0110 1000 : imm64
PUSH – Push Segment Register onto the Stack	
segment register FS,GS	0000 1111: 10 sreg3 000
PUSHF/PUSHFD – Push Flags Register onto the Stack	1001 1100
RCL – Rotate thru Carry Left	
register by 1	0100 000B : 1101 000w : 11 010 reg
qwordregister by 1	0100 100B 1101 0001 : 11 010 qwordreg

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

Instruction and Format	Encoding
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
RCR – Rotate thru Carry Right	
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
qwordregister 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
RDMSR – Read from Model-Specific Register	
load ECX-specified register into EDX:EAX	0000 1111 : 0011 0010
RDPMC – Read Performance Monitoring Counters	
load ECX-specified performance counter into EDX:EAX	0000 1111 : 0011 0011
RDTSC – Read Time-Stamp Counter	
read time-stamp counter into EDX:EAX	0000 1111 : 0011 0001
RDTSCP – Read Time-Stamp Counter and Processor ID	0000 1111 : 0000 0001: 1111 1001
REP INS – Input String	
REP LODS – Load String	
REP MOVS – Move String	
REP OUTS – Output String	
REP STOS – Store String	
REPE CMPS – Compare String	
REPE SCAS – Scan String	
REPNE CMPS – Compare String	

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

Instruction and Format	Encoding
REPNE SCAS – Scan String	
RET – Return from Procedure (to same segment)	
no argument	1100 0011
adding immediate to SP	1100 0010 : 16-bit displacement
RET – Return from Procedure (to other segment)	
intersegment	1100 1011
adding immediate to SP	1100 1010 : 16-bit displacement
ROL – Rotate Left	
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
ROR – Rotate Right	
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
byteregister by CL	0100 000B 1101 0010 : 11 001 bytereg
qwordregister 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

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

Instruction and Format	Encoding
memory64 by CL	0100 10XB 1101 0011 : mod 001 r/m
register by immediate count	0100 000B 1100 000w : 11 001 reg : imm8
byteregister by immediate count	0100 000B 1100 0000 : 11 001 reg : imm8
qwordregister 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
RSM – Resume from System Management Mode	0000 1111 : 1010 1010
SAL – Shift Arithmetic Left	same instruction as SHL
SAR – Shift Arithmetic Right	
register by 1	0100 000B 1101 000w : 11 111 reg
byteregister by 1	0100 000B 1101 0000 : 11 111 bytereg
qwordregister 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
byteregister by CL	0100 000B 1101 0010 : 11 111 bytereg
qwordregister 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
byteregister by immediate count	0100 000B 1100 0000 : 11 111 bytereg : imm8
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
SBB – Integer Subtraction with Borrow	
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

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

Instruction and Format	Encoding
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
immediate8 to memory64	0100 10XB 1000 0011 : mod 011 r/m : imm8
SCAS/SCASB/SCASW/SCASD – Scan String	
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
SETcc – Byte Set on Condition	
register	0100 000B 0000 1111 : 1001 tttn : 11 000 reg
register	0100 0000 0000 1111 : 1001 tttn : 11 000 reg
memory	0100 00XB 0000 1111 : 1001 tttn : mod 000 r/m
memory	0100 0000 0000 1111 : 1001 tttn : mod 000 r/m
SGDT – Store Global Descriptor Table Register	0000 1111 : 0000 0001 : mod[A] 000 r/m
SHL – Shift Left	
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
SHLD – Double Precision Shift Left	
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
SHR – Shift Right	
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
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
SHRD – Double Precision Shift Right	
register by immediate count	0100 0R0B 0000 1111 : 1010 1100 : 11 reg2 reg1 : imm8
qwordregister by immediate8	0100 1R0B 0000 1111 : 1010 1100 : 11 qwordreg2 qwordreg1 : imm8
memory by immediate count	0100 00XB 0000 1111 : 1010 1100 : mod reg r/m : imm8

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

Instruction and Format	Encoding
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 1R0B 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
SIDT – Store Interrupt Descriptor Table Register	0000 1111 : 0000 0001 : mod[A] 001 r/m
SLDT – Store Local Descriptor Table Register	
to register	0100 000B 0000 1111 : 0000 0000 : 11 000 reg
to memory	0100 00XB 0000 1111 : 0000 0000 : mod 000 r/m
SMSW – Store Machine Status Word	
to register	0100 000B 0000 1111 : 0000 0001 : 11 100 reg
to memory	0100 00XB 0000 1111 : 0000 0001 : mod 100 r/m
STC – Set Carry Flag	1111 1001
STD – Set Direction Flag	1111 1101
STI – Set Interrupt Flag	1111 1011
STOS/STOSB/STOSW/STOSD/STOSQ – Store String Data	
store string data	1010 101w
store string data (RAX at address RDI)	0100 1000 1010 1011
STR – Store Task Register	
to register	0100 000B 0000 1111 : 0000 0000 : 11 001 reg
to memory	0100 00XB 0000 1111 : 0000 0000 : mod 001 r/m
SUB – Integer Subtraction	
register1 from register2	0100 0R0B 0010 100w : 11 reg1 reg2
byteregister1 from byteregister2	0100 0R0B 0010 1000 : 11 bytereg1 bytereg2
qwordregister1 from qwordregister2	0100 1R0B 0010 1000 : 11 qwordreg1 qwordreg2
register2 from register1	0100 0R0B 0010 101w : 11 reg1 reg2
byteregister2 from byteregister1	0100 0R0B 0010 1010 : 11 bytereg1 bytereg2
qwordregister2 from qwordregister1	0100 1R0B 0010 1011 : 11 qwordreg1 qwordreg2
memory from register	0100 00XB 0010 101w : mod reg r/m
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

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

Instruction and Format	Encoding
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
SWAPGS – Swap GS Base Register	
Exchanges the current GS base register value for value in MSR C0000102H	0000 1111 0000 0001 1111 1000
SYSCALL – Fast System Call	
fast call to privilege level 0 system procedures	0000 1111 0000 0101
SYSRET – Return From Fast System Call	
return from fast system call	0000 1111 0000 0111
TEST – Logical Compare	
register1 and register2	0100 0R0B 1000 010w : 11 reg1 reg2
byteregister1 and byteregister2	0100 0R0B 1000 0100 : 11 bytereg1 bytereg2
qwordregister1 and qwordregister2	0100 1R0B 1000 0101 : 11 qwordreg1 qwordreg2
memory and register	0100 0R0B 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
UD2 – Undefined instruction	0000 FFFF : 0000 1011
VERR – Verify a Segment for Reading	
register	0100 000B 0000 1111 : 0000 0000 : 11 100 reg
memory	0100 00XB 0000 1111 : 0000 0000 : mod 100 r/m
VERW – Verify a Segment for Writing	
register	0100 000B 0000 1111 : 0000 0000 : 11 101 reg
memory	0100 00XB 0000 1111 : 0000 0000 : mod 101 r/m
WAIT – Wait	1001 1011
WBINVD – Writeback and Invalidate Data Cache	0000 1111 : 0000 1001

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

Instruction and Format	Encoding
WRMSR – Write to Model-Specific Register	
write EDX:EAX to ECX specified MSR	0000 1111 : 0011 0000
write RDX[31:0]:RAX[31:0] to RCX specified MSR	0100 1000 0000 1111 : 0011 0000
XADD – Exchange and Add	
register1, register2	0100 0R0B 0000 1111 : 1100 000w : 11 reg2 reg1
byteregister1, byteregister2	0100 0R0B 0000 1111 : 1100 0000 : 11 bytereg2 bytereg1
qwordregister1, qwordregister2	0100 0R0B 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
XCHG – Exchange Register/Memory with Register	
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
XLAT/XLATB – Table Look-up Translation	
AL to byte DS:[(E)BX + unsigned AL]	1101 0111
AL to byte DS:[RBX + unsigned AL]	0100 1000 1101 0111
XOR – Logical Exclusive OR	
register1 to register2	0100 0RXB 0011 000w : 11 reg1 reg2
byteregister1 to byteregister2	0100 0R0B 0011 0000 : 11 bytereg1 bytereg2
qwordregister1 to qwordregister2	0100 1R0B 0011 0001 : 11 qwordreg1 qwordreg2
register2 to register1	0100 0R0B 0011 001w : 11 reg1 reg2
byteregister2 to byteregister1	0100 0R0B 0011 0010 : 11 bytereg1 bytereg2
qwordregister2 to qwordregister1	0100 1R0B 0011 0011 : 11 qwordreg1 qwordreg2
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

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 110 r/m : imm32
immediate8 to memory64	0100 10XB 1000 0011 : mod 110 r/m : imm8
Prefix Bytes	
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
CMPXCHG8B – Compare and Exchange 8 Bytes	
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
CMPXCHG8B/CMPXCHG16B – Compare and Exchange Bytes	
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).

1. 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.

2. 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

Instruction and Format	Encoding
EMMS – Empty MMX technology state	0000 1111:01110111
MOVD – Move doubleword	
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
MOVQ – Move quadword	
mmxreg2 to mmxreg1	0000 1111:0110 1111: 11 mmxreg1 mmxreg2
mmxreg2 from mmxreg1	0000 1111:0111 1111: 11 mmxreg1 mmxreg2

Table B-19. MMX Instruction Formats and Encodings (Contd.)

Instruction and Format	Encoding
mem to mmxreg	0000 1111:0110 1111: mod mmxreg r/m
mem from mmxreg	0000 1111:0111 1111: mod mmxreg r/m
PACKSSDW[1] – Pack dword to word data (signed with saturation)	
mmxreg2 to mmxreg1	0000 1111:0110 1011: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111:0110 1011: mod mmxreg r/m
PACKSSWB[1] – Pack word to byte data (signed with saturation)	
mmxreg2 to mmxreg1	0000 1111:0110 0011: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111:0110 0011: mod mmxreg r/m
PACKUSWB[1] – Pack word to byte data (unsigned with saturation)	
mmxreg2 to mmxreg1	0000 1111:0110 0111: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111:0110 0111: mod mmxreg r/m
PADD – Add with wrap-around	
mmxreg2 to mmxreg1	0000 1111: 1111 11gg: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111: 1111 11gg: mod mmxreg r/m
PADDS – Add signed with saturation	
mmxreg2 to mmxreg1	0000 1111: 1110 11gg: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111: 1110 11gg: mod mmxreg r/m
PADDUS – Add unsigned with saturation	
mmxreg2 to mmxreg1	0000 1111: 1101 11gg: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111: 1101 11gg: mod mmxreg r/m
PAND – Bitwise And	
mmxreg2 to mmxreg1	0000 1111:1101 1011: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111:1101 1011: mod mmxreg r/m
PANDN – Bitwise AndNot	
mmxreg2 to mmxreg1	0000 1111:1101 1111: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111:1101 1111: mod mmxreg r/m
PCMPEQ – Packed compare for equality	
mmxreg1 with mmxreg2	0000 1111:0111 01gg: 11 mmxreg1 mmxreg2
mmxreg with memory	0000 1111:0111 01gg: mod mmxreg r/m
PCMPGT – Packed compare greater (signed)	
mmxreg1 with mmxreg2	0000 1111:0110 01gg: 11 mmxreg1 mmxreg2
mmxreg with memory	0000 1111:0110 01gg: mod mmxreg r/m
PMADDWD – Packed multiply add	
mmxreg2 to mmxreg1	0000 1111:1111 0101: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111:1111 0101: mod mmxreg r/m
PMULHUW – Packed multiplication, store high word (unsigned)	
mmxreg2 to mmxreg1	0000 1111: 1110 0100: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111: 1110 0100: mod mmxreg r/m

Table B-19. MMX Instruction Formats and Encodings (Contd.)

Instruction and Format	Encoding
PMULHW – Packed multiplication, store high word	
mmxreg2 to mmxreg1	0000 1111:1110 0101: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111:1110 0101: mod mmxreg r/m
PMULLW – Packed multiplication, store low word	
mmxreg2 to mmxreg1	0000 1111:1101 0101: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111:1101 0101: mod mmxreg r/m
POR – Bitwise Or	
mmxreg2 to mmxreg1	0000 1111:1110 1011: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111:1110 1011: mod mmxreg r/m
PSLL[2] – Packed shift left logical	
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
PSRA[2] – Packed shift right arithmetic	
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
PSRL[2] – Packed shift right logical	
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
PSUB – Subtract with wrap-around	
mmxreg2 from mmxreg1	0000 1111:1111 10gg: 11 mmxreg1 mmxreg2
memory from mmxreg	0000 1111:1111 10gg: mod mmxreg r/m
PSUBS – Subtract signed with saturation	
mmxreg2 from mmxreg1	0000 1111:1110 10gg: 11 mmxreg1 mmxreg2
memory from mmxreg	0000 1111:1110 10gg: mod mmxreg r/m
PSUBUS – Subtract unsigned with saturation	
mmxreg2 from mmxreg1	0000 1111:1101 10gg: 11 mmxreg1 mmxreg2
memory from mmxreg	0000 1111:1101 10gg: mod mmxreg r/m
PUNPCKH – Unpack high data to next larger type	
mmxreg2 to mmxreg1	0000 1111:0110 10gg: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111:0110 10gg: mod mmxreg r/m
PUNPCKL – Unpack low data to next larger type	
mmxreg2 to mmxreg1	0000 1111:0110 00gg: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111:0110 00gg: mod mmxreg r/m

Table B-19. MMX Instruction Formats and Encodings (Contd.)

Instruction and Format	Encoding
PXOR – Bitwise Xor	
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
XGETBV – Get Value of Extended Control Register	0000 1111:0000 0001: 1101 0000
XRSTOR – Restore Processor Extended States[1]	0000 1111:1010 1110: mod[A] 101 r/m
XSAVE – Save Processor Extended States[1]	0000 1111:1010 1110: mod[A] 100 r/m
XSETBV – Set Extended Control Register	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
CMOVcc – Conditional Move	
register2 to register1	0000 1111: 0100 tttn : 11 reg1 reg2
memory to register	0000 1111 : 0100 tttn : mod reg r/m
FCMOVcc – Conditional Move on EFLAG Register Condition Codes	
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)

Table B-21. Formats and Encodings of P6 Family Instructions (Contd.)

Instruction and Format	Encoding
move if not unordered (NU)	11011 011 : 11 011 ST(i)
FCOMI – Compare Real and Set EFLAGS	11011 011 : 11 110 ST(i)
FXRSTOR – Restore x87 FPU, MMX, SSE, and SSE2 State[1]	0000 1111:1010 1110: mod[A] 001 r/m
FXSAVE – Save x87 FPU, MMX, SSE, and SSE2 State[1]	0000 1111:1010 1110: mod[A] 000 r/m
SYSENTER – Fast System Call	0000 1111:0011 0100
SYSEXIT – Fast Return from Fast System Call	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
ADDPS—Add Packed Single Precision Floating-Point Values	
xmmreg2 to xmmreg1	0000 1111:0101 1000:11 xmmreg1 xmmreg2
mem to xmmreg	0000 1111:0101 1000: mod xmmreg r/m
ADDSS—Add Scalar Single Precision Floating-Point Values	
xmmreg2 to xmmreg1	1111 0011:0000 1111:01011000:11 xmmreg1 xmmreg2
mem to xmmreg	1111 0011:0000 1111:01011000: mod xmmreg r/m
ANDNPS—Bitwise Logical AND NOT of Packed Single Precision Floating-Point Values	
xmmreg2 to xmmreg1	0000 1111:0101 0101:11 xmmreg1 xmmreg2
mem to xmmreg	0000 1111:0101 0101: mod xmmreg r/m
ANDPS—Bitwise Logical AND of Packed Single Precision Floating-Point Values	
xmmreg2 to xmmreg1	0000 1111:0101 0100:11 xmmreg1 xmmreg2
mem to xmmreg	0000 1111:0101 0100: mod xmmreg r/m
CMPPS—Compare Packed Single Precision Floating-Point Values	
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
CMPSS—Compare Scalar Single Precision Floating-Point Values	
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
COMISS—Compare Scalar Ordered Single Precision Floating-Point Values and Set EFLAGS	
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
CVTPI2PS—Convert Packed Doubleword Integers to Packed Single Precision Floating-Point Values	
mmreg to xmmreg	0000 1111:0010 1010:11 xmmreg1 mmreg1
mem to xmmreg	0000 1111:0010 1010: mod xmmreg r/m
CVTPS2PI—Convert Packed Single Precision Floating-Point Values to Packed Doubleword Integers	
xmmreg to mmreg	0000 1111:0010 1101:11 mmreg1 xmmreg1
mem to mmreg	0000 1111:0010 1101: mod mmreg r/m
CVTSI2SS—Convert Doubleword Integer to Scalar Single Precision Floating-Point Value	
r32 to xmmreg1	1111 0011:0000 1111:00101010:11 xmmreg1 r32
mem to xmmreg	1111 0011:0000 1111:00101010: mod xmmreg r/m
CVTSS2SI—Convert Scalar Single Precision Floating-Point Value to Doubleword Integer	
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
CVTTPS2PI—Convert with Truncation Packed Single Precision Floating-Point Values to Packed Doubleword Integers	
xmmreg to mmreg	0000 1111:0010 1100:11 mmreg1 xmmreg1
mem to mmreg	0000 1111:0010 1100: mod mmreg r/m
CVTTSS2SI—Convert with Truncation Scalar Single Precision Floating-Point Value to Doubleword Integer	
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
DIVPS—Divide Packed Single Precision Floating-Point Values	
xmmreg2 to xmmreg1	0000 1111:0101 1110:11 xmmreg1 xmmreg2
mem to xmmreg	0000 1111:0101 1110: mod xmmreg r/m
DIVSS—Divide Scalar Single Precision Floating-Point Values	
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
LDMXCSR—Load MXCSR Register State	
m32 to MXCSR	0000 1111:1010 1110:mod[A] 010 mem
MAXPS—Return Maximum Packed Single Precision Floating-Point Values	
xmmreg2 to xmmreg1	0000 1111:0101 1111:11 xmmreg1 xmmreg2
mem to xmmreg	0000 1111:0101 1111: mod xmmreg r/m
MAXSS—Return Maximum Scalar Double Precision Floating-Point Value	
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
MINPS—Return Minimum Packed Double Precision Floating-Point Values	
xmmreg2 to xmmreg1	0000 1111:0101 1101:11 xmmreg1 xmmreg2
mem to xmmreg	0000 1111:0101 1101: mod xmmreg r/m
MINSS—Return Minimum Scalar Double Precision Floating-Point Value	
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

Table B-22. Formats and Encodings of SSE Floating-Point Instructions (Contd.)

Instruction and Format	Encoding
MOVAPS—Move Aligned Packed Single Precision Floating-Point Values	
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
MOVHLPS—Move Packed Single Precision Floating-Point Values High to Low	
xmmreg2 to xmmreg1	0000 1111:0001 0010:11 xmmreg1 xmmreg2
MOVHPS—Move High Packed Single Precision Floating-Point Values	
mem to xmmreg	0000 1111:0001 0110: mod xmmreg r/m
xmmreg to mem	0000 1111:0001 0111: mod xmmreg r/m
MOVLHPS—Move Packed Single Precision Floating-Point Values Low to High	
xmmreg2 to xmmreg1	0000 1111:00010110:11 xmmreg1 xmmreg2
MOVLPS—Move Low Packed Single Precision Floating-Point Values	
mem to xmmreg	0000 1111:0001 0010: mod xmmreg r/m
xmmreg to mem	0000 1111:0001 0011: mod xmmreg r/m
MOVMSKPS—Extract Packed Single Precision Floating-Point Sign Mask	
xmmreg to r32	0000 1111:0101 0000:11 r32 xmmreg
MOVSS—Move Scalar Single Precision Floating-Point Values	
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
MOVUPS—Move Unaligned Packed Single Precision Floating-Point Values	
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
MULPS—Multiply Packed Single Precision Floating-Point Values	
xmmreg2 to xmmreg1	0000 1111:0101 1001:11 xmmreg1 xmmreg2
mem to xmmreg	0000 1111:0101 1001: mod xmmreg r/m
MULSS—Multiply Scalar Single Precision Floating-Point Values	
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
ORPS—Bitwise Logical OR of Single Precision Floating-Point Values	
xmmreg2 to xmmreg1	0000 1111:0101 0110:11 xmmreg1 xmmreg2
mem to xmmreg	0000 1111:0101 0110: mod xmmreg r/m
RCPPS—Compute Reciprocals of Packed Single Precision Floating-Point Values	
xmmreg2 to xmmreg1	0000 1111:0101 0011:11 xmmreg1 xmmreg2

Table B-22. Formats and Encodings of SSE Floating-Point Instructions (Contd.)

Instruction and Format	Encoding
mem to xmmreg	0000 1111:0101 0011: mod xmmreg r/m
RCPSS—Compute Reciprocals of Scalar Single Precision Floating-Point Value	
xmmreg2 to xmmreg1	1111 0011:0000 1111:01010011:11 xmmreg1 xmmreg2
mem to xmmreg	1111 0011:0000 1111:01010011: mod xmmreg r/m
RSQRTPS—Compute Reciprocals of Square Roots of Packed Single Precision Floating-Point Values	
xmmreg2 to xmmreg1	0000 1111:0101 0010:11 xmmreg1 xmmreg2
mem to xmmreg	0000 1111:0101 0010: mode xmmreg r/m
RSQRTSS—Compute Reciprocals of Square Roots of Scalar Single Precision Floating-Point Value	
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
SHUFPS—Shuffle Packed Single Precision Floating-Point Values	
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
SQRTPS—Compute Square Roots of Packed Single Precision Floating-Point Values	
xmmreg2 to xmmreg1	0000 1111:0101 0001:11 xmmreg1 xmmreg2
mem to xmmreg	0000 1111:0101 0001: mod xmmreg r/m
SQRTSS—Compute Square Root of Scalar Single Precision Floating-Point Value	
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
STMXCSR—Store MXCSR Register State	
MXCSR to mem	0000 1111:1010 1110:mod[A] 011 mem
SUBPS—Subtract Packed Single Precision Floating-Point Values	
xmmreg2 to xmmreg1	0000 1111:0101 1100:11 xmmreg1 xmmreg2
mem to xmmreg	0000 1111:0101 1100:mod xmmreg r/m
SUBSS—Subtract Scalar Single Precision Floating-Point Values	
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
UCOMISS—Unordered Compare Scalar Ordered Single Precision Floating-Point Values and Set EFLAGS	
xmmreg2 to xmmreg1	0000 1111:0010 1110:11 xmmreg1 xmmreg2
mem to xmmreg	0000 1111:0010 1110: mod xmmreg r/m
UNPCKHPS—Unpack and Interleave High Packed Single Precision Floating-Point Values	
xmmreg2 to xmmreg1	0000 1111:0001 0101:11 xmmreg1 xmmreg2
mem to xmmreg	0000 1111:0001 0101: mod xmmreg r/m
UNPCKLPS—Unpack and Interleave Low Packed Single Precision Floating-Point Values	
xmmreg2 to xmmreg1	0000 1111:0001 0100:11 xmmreg1 xmmreg2
mem to xmmreg	0000 1111:0001 0100: mod xmmreg r/m
XORPS—Bitwise Logical XOR of Single Precision Floating-Point Values	
xmmreg2 to xmmreg1	0000 1111:0101 0111:11 xmmreg1 xmmreg2

Table B-22. Formats and Encodings of SSE Floating-Point Instructions (Contd.)

Instruction and Format	Encoding
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
PAVGB/PAVGW—Average Packed Integers	
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
PEXTRW—Extract Word	
mmreg to reg32, imm8	0000 1111:1100 0101:11 r32 mmreg: imm8
PINSRW—Insert Word	
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
PMAXSW—Maximum of Packed Signed Word Integers	
mmreg2 to mmreg1	0000 1111:1110 1110:11 mmreg1 mmreg2
mem to mmreg	0000 1111:1110 1110: mod mmreg r/m
PMAXUB—Maximum of Packed Unsigned Byte Integers	
mmreg2 to mmreg1	0000 1111:1101 1110:11 mmreg1 mmreg2
mem to mmreg	0000 1111:1101 1110: mod mmreg r/m
PMINSW—Minimum of Packed Signed Word Integers	
mmreg2 to mmreg1	0000 1111:1110 1010:11 mmreg1 mmreg2
mem to mmreg	0000 1111:1110 1010: mod mmreg r/m
PMINUB—Minimum of Packed Unsigned Byte Integers	
mmreg2 to mmreg1	0000 1111:1101 1010:11 mmreg1 mmreg2
mem to mmreg	0000 1111:1101 1010: mod mmreg r/m
PMOVMSKB—Move Byte Mask To Integer	
mmreg to reg32	0000 1111:1101 0111:11 r32 mmreg
PMULHUW—Multiply Packed Unsigned Integers and Store High Result	
mmreg2 to mmreg1	0000 1111:1110 0100:11 mmreg1 mmreg2
mem to mmreg	0000 1111:1110 0100: mod mmreg r/m
PSADBW—Compute Sum of Absolute Differences	
mmreg2 to mmreg1	0000 1111:1111 0110:11 mmreg1 mmreg2
mem to mmreg	0000 1111:1111 0110: mod mmreg r/m
PSHUFW—Shuffle Packed Words	
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
MASKMOVQ—Store Selected Bytes of Quadword	
mmreg2 to mmreg1	0000 1111:1111 0111:11 mmreg1 mmreg2
MOVNTPS—Store Packed Single Precision Floating-Point Values Using Non-Temporal Hint	
xmmreg to mem	0000 1111:0010 1011: mod xmmreg r/m
MOVNTQ—Store Quadword Using Non-Temporal Hint	
mmreg to mem	0000 1111:1110 0111: mod mmreg r/m
PREFETCHT0—Prefetch Temporal to All Cache Levels	0000 1111:0001 1000:mod[A] 001 mem
PREFETCHT1—Prefetch Temporal to First Level Cache	0000 1111:0001 1000:mod[A] 010 mem
PREFETCHT2—Prefetch Temporal to Second Level Cache	0000 1111:0001 1000:mod[A] 011 mem
PREFETCHNTA—Prefetch Non-Temporal to All Cache Levels	0000 1111:0001 1000:mod[A] 000 mem
SFENCE—Store Fence	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

Instruction and Format	Encoding
ADDPD—Add Packed Double Precision Floating-Point Values	
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
ADDSD—Add Scalar Double Precision Floating-Point Values	
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
ANDNPD—Bitwise Logical AND NOT of Packed Double Precision Floating-Point Values	
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
ANDPD—Bitwise Logical AND of Packed Double Precision Floating-Point Values	
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
CMPPD—Compare Packed Double Precision Floating-Point Values	
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
CMPSD—Compare Scalar Double Precision Floating-Point Values	
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
COMISD—Compare Scalar Ordered Double Precision Floating-Point Values and Set EFLAGS	
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
CVTPI2PD—Convert Packed Doubleword Integers to Packed Double Precision Floating-Point Values	
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
CVTPD2PI—Convert Packed Double Precision Floating-Point Values to Packed Doubleword Integers	
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
CVTSI2SD—Convert Doubleword Integer to Scalar Double Precision Floating-Point Value	
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
CVTSD2SI—Convert Scalar Double Precision Floating-Point Value to Doubleword Integer	
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
CVTTPD2PI—Convert with Truncation Packed Double Precision Floating-Point Values to Packed Doubleword Integers	
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
CVTTSD2SI—Convert with Truncation Scalar Double Precision Floating-Point Value to Doubleword Integer	
xmmreg to r32	1111 0010:0000 1111:0010 1100:11 r32 xmmreg

Table B-26. Formats and Encodings of SSE2 Floating-Point Instructions (Contd.)

Instruction and Format	Encoding
mem to r32	1111 0010:0000 1111:0010 1100: mod r32 r/m
CVTPD2PS—Covert Packed Double Precision Floating-Point Values to Packed Single Precision Floating-Point Values	
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
CVTPS2PD—Covert Packed Single Precision Floating-Point Values to Packed Double Precision Floating-Point Values	
xmmreg2 to xmmreg1	0000 1111:0101 1010:11 xmmreg1 xmmreg2
mem to xmmreg	0000 1111:0101 1010: mod xmmreg r/m
CVTSD2SS—Covert Scalar Double Precision Floating-Point Value to Scalar Single Precision Floating-Point Value	
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
CVTSS2SD—Covert Scalar Single Precision Floating-Point Value to Scalar Double Precision Floating-Point Value	
xmmreg2 to xmmreg1	1111 0011:0000 1111:0101 1010:11 xmmreg1 xmmreg2
mem to xmmreg	1111 0011:00001 111:0101 1010: mod xmmreg r/m
CVTPD2DQ—Convert Packed Double Precision Floating-Point Values to Packed Doubleword Integers	
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
CVTTPD2DQ—Convert With Truncation Packed Double Precision Floating-Point Values to Packed Doubleword Integers	
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
CVTDQ2PD—Convert Packed Doubleword Integers to Packed Single Precision Floating-Point Values	
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
CVTPS2DQ—Convert Packed Single Precision Floating-Point Values to Packed Doubleword Integers	
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
CVTTPS2DQ—Convert With Truncation Packed Single Precision Floating-Point Values to Packed Doubleword Integers	
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
CVTDQ2PS—Convert Packed Doubleword Integers to Packed Double Precision Floating-Point Values	
xmmreg2 to xmmreg1	0000 1111:0101 1011:11 xmmreg1 xmmreg2
mem to xmmreg	0000 1111:0101 1011: mod xmmreg r/m
DIVPD—Divide Packed Double Precision Floating-Point Values	
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
DIVSD—Divide Scalar Double Precision Floating-Point Values	
xmmreg2 to xmmreg1	1111 0010:0000 1111:0101 1110:11 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 1110: mod xmmreg r/m
MAXPD—Return Maximum Packed Double Precision Floating-Point Values	
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
MAXSD—Return Maximum Scalar Double Precision Floating-Point Value	
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
MINPD—Return Minimum Packed Double Precision Floating-Point Values	
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
MINSD—Return Minimum Scalar Double Precision Floating-Point Value	
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
MOVAPD—Move Aligned Packed Double Precision Floating-Point Values	
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
MOVHPD—Move High Packed Double Precision Floating-Point Values	
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
MOVLPD—Move Low Packed Double Precision Floating-Point Values	
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
MOVMSKPD—Extract Packed Double Precision Floating-Point Sign Mask	
xmmreg to r32	0110 0110:0000 1111:0101 0000:11 r32 xmmreg
MOVSD—Move Scalar Double Precision Floating-Point Values	
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
MOVUPD—Move Unaligned Packed Double Precision Floating-Point Values	
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
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
MULPD—Multiply Packed Double Precision Floating-Point Values	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0101 1001:11 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 1001: mod xmmreg r/m
MULSD—Multiply Scalar Double Precision Floating-Point Values	
xmmreg2 to xmmreg1	1111 0010:00001111:01011001:11 xmmreg1 xmmreg2
mem to xmmreg	1111 0010:00001111:01011001: mod xmmreg r/m
ORPD—Bitwise Logical OR of Double Precision Floating-Point Values	
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
SHUFPD—Shuffle Packed Double Precision Floating-Point Values	
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
SQRTPD—Compute Square Roots of Packed Double Precision Floating-Point Values	
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
SQRTSD—Compute Square Root of Scalar Double Precision Floating-Point Value	
xmmreg2 to xmmreg1	1111 0010:0000 1111:0101 0001:11 xmmreg1 xmmreg2
mem to xmmreg	1111 0010:0000 1111:0101 0001: mod xmmreg r/m
SUBPD—Subtract Packed Double Precision Floating-Point Values	
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
SUBSD—Subtract Scalar Double Precision Floating-Point Values	
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
UCOMISD—Unordered Compare Scalar Ordered Double Precision Floating-Point Values and Set EFLAGS	
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
UNPCKHPD—Unpack and Interleave High Packed Double Precision Floating-Point Values	
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
UNPCKLPD—Unpack and Interleave Low Packed Double Precision Floating-Point Values	
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
XORPD—Bitwise Logical OR of Double Precision Floating-Point Values	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0101 0111:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0101 0111: mod xmmreg r/m

Table B-27. Formats and Encodings of SSE2 Integer Instructions

Instruction and Format	Encoding
MOVD—Move Doubleword	
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
MOVDQA—Move Aligned Double Quadword	
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
MOVDQU—Move Unaligned Double Quadword	
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
MOVQ2DQ—Move Quadword from MMX to XMM Register	
mmreg to xmmreg	1111 0011:0000 1111:1101 0110:11 mmreg1 mmreg2
MOVDQ2Q—Move Quadword from XMM to MMX Register	
xmmreg to mmreg	1111 0010:0000 1111:1101 0110:11 mmreg1 mmreg2
MOVQ—Move Quadword	
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
mem from xmmreg	0110 0110:0000 1111:1101 0110: mod xmmreg r/m
PACKSSDW[1]—Pack Dword To Word Data (signed with saturation)	
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
PACKSSWB—Pack Word To Byte Data (signed with saturation)	
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
PACKUSWB—Pack Word To Byte Data (unsigned with saturation)	
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
PADDQ—Add Packed Quadword Integers	
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

Table B-27. Formats and Encodings of SSE2 Integer Instructions (Contd.)

Instruction and Format	Encoding
PADD—Add With Wrap-around	
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
PADDS—Add Signed With Saturation	
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
PADDUS—Add Unsigned With Saturation	
xmmreg2 to xmmreg1	0110 0110:0000 1111: 1101 11gg: 11 xmmreg1 xmmreg2
memory to xmmreg	0110 0110:0000 1111: 1101 11gg: mod xmmreg r/m
PAND—Bitwise And	
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
PANDN—Bitwise AndNot	
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
PAVGB—Average Packed Integers	
xmmreg2 to xmmreg1	0110 0110:0000 1111:11100 000:11 xmmreg1 xmmreg2
mem to xmmreg	01100110:00001111:11100000 mod xmmreg r/m
PAVGW—Average Packed Integers	
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
PCMPEQ—Packed Compare For Equality	
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
PCMPGT—Packed Compare Greater (signed)	
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
PEXTRW—Extract Word	
xmmreg to reg32, imm8	0110 0110:0000 1111:1100 0101:11 r32 xmmreg: imm8
PINSRW—Insert Word	
reg32 to xmmreg, imm8	0110 0110:0000 1111:1100 0100:11 xmmreg r32: imm8
m16 to xmmreg, imm8	0110 0110:0000 1111:1100 0100: mod xmmreg r/m: imm8
PMADDWD—Packed Multiply Add	
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
PMAXSW—Maximum of Packed Signed Word Integers	
xmmreg2 to xmmreg1	0110 0110:0000 1111:1110 1110:11 xmmreg1 xmmreg2
mem to xmmreg	01100110:00001111:11101110: mod xmmreg r/m

Table B-27. Formats and Encodings of SSE2 Integer Instructions (Contd.)

Instruction and Format	Encoding
PMAXUB—Maximum of Packed Unsigned Byte Integers	
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
PMINSW—Minimum of Packed Signed Word Integers	
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
PMINUB—Minimum of Packed Unsigned Byte Integers	
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
PMOVMSKB—Move Byte Mask To Integer	
xmmreg to reg32	0110 0110:0000 1111:1101 0111:11 r32 xmmreg
PMULHUW—Packed multiplication, store high word (unsigned)	
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
PMULHW—Packed Multiplication, store high word	
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
PMULLW—Packed Multiplication, store low word	
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
PMULUDQ—Multiply Packed Unsigned Doubleword Integers	
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
POR—Bitwise Or	
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
PSADBW—Compute Sum of Absolute Differences	
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
PSHUFLW—Shuffle Packed Low Words	
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
PSHUFHW—Shuffle Packed High Words	
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
PSHUFD—Shuffle Packed Doublewords	

Table B-27. Formats and Encodings of SSE2 Integer Instructions (Contd.)

Instruction and Format	Encoding
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
PSLLDQ—Shift Double Quadword Left Logical	
xmmreg, imm8	0110 0110:0000 1111:0111 0011:11 111 xmmreg: imm8
PSLL—Packed Shift Left Logical	
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
PSRA—Packed Shift Right Arithmetic	
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
PSRLDQ—Shift Double Quadword Right Logical	
xmmreg, imm8	0110 0110:00001111:01110011:11 011 xmmreg: imm8
PSRL—Packed Shift Right Logical	
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
xmmreg by immediate	0110 0110:0000 1111:0111 00gg: 11 010 xmmreg: imm8
PSUBQ—Subtract Packed Quadword Integers	
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
PSUB—Subtract With Wrap-around	
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
PSUBS—Subtract Signed With Saturation	
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
PSUBUS—Subtract Unsigned With Saturation	
xmmreg2 from xmmreg1	0000 1111:1101 10gg: 11 xmmreg1 xmmreg2
memory from xmmreg	0000 1111:1101 10gg: mod xmmreg r/m
PUNPCKH—Unpack High Data To Next Larger Type	
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
PUNPCKHQDQ—Unpack High Data	
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

Table B-27. Formats and Encodings of SSE2 Integer Instructions (Contd.)

Instruction and Format	Encoding
PUNPCKL—Unpack Low Data To Next Larger Type	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0110 00gg:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0110 00gg: mod xmmreg r/m
PUNPCKLQDQ—Unpack Low Data	
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
PXOR—Bitwise Xor	
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
MASKMOVDQU—Store Selected Bytes of Double Quadword	
xmmreg2 to xmmreg1	0110 0110:0000 1111:1111 0111:11 xmmreg1 xmmreg2
CLFLUSH—Flush Cache Line	
mem	0000 1111:1010 1110: mod 111 r/m
MOVNTPD—Store Packed Double Precision Floating-Point Values Using Non-Temporal Hint	
xmmreg to mem	0110 0110:0000 1111:0010 1011: mod xmmreg r/m
MOVNTDQ—Store Double Quadword Using Non-Temporal Hint	
xmmreg to mem	0110 0110:0000 1111:1110 0111: mod xmmreg r/m
MOVNTI—Store Doubleword Using Non-Temporal Hint	
reg to mem	0000 1111:1100 0011: mod reg r/m
PAUSE—Spin Loop Hint	1111 0011:1001 0000
LFENCE—Load Fence	0000 1111:1010 1110: 11 101 000
MFENCE—Memory Fence	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
ADDSUBPD—Add /Sub packed DP FP numbers from XMM2/Mem to XMM1	
xmmreg2 to xmmreg1	01100110:00001111:11010000:11 xmmreg1 xmmreg2
mem to xmmreg	01100110:00001111:11010000: mod xmmreg r/m
ADDSUBPS—Add /Sub packed SP FP numbers from XMM2/Mem to XMM1	
xmmreg2 to xmmreg1	11110010:00001111:11010000:11 xmmreg1 xmmreg2
mem to xmmreg	11110010:00001111:11010000: mod xmmreg r/m
HADDPD—Add horizontally packed DP FP numbers XMM2/Mem to XMM1	
xmmreg2 to xmmreg1	01100110:00001111:01111100:11 xmmreg1 xmmreg2
mem to xmmreg	01100110:00001111:01111100: mod xmmreg r/m
HADDPS—Add horizontally packed SP FP numbers XMM2/Mem to XMM1	
xmmreg2 to xmmreg1	11110010:00001111:01111100:11 xmmreg1 xmmreg2
mem to xmmreg	11110010:00001111:01111100: mod xmmreg r/m
HSUBPD—Sub horizontally packed DP FP numbers XMM2/Mem to XMM1	
xmmreg2 to xmmreg1	01100110:00001111:01111101:11 xmmreg1 xmmreg2
mem to xmmreg	01100110:00001111:01111101: mod xmmreg r/m
HSUBPS—Sub horizontally packed SP FP numbers XMM2/Mem to XMM1	
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
MONITOR—Set up a linear address range to be monitored by hardware	
eax, ecx, edx	0000 1111 : 0000 0001:11 001 000
MWAIT—Wait until write-back store performed within the range specified by the instruction MONITOR	
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
FISTTP—Store ST in int16 (chop) and pop	
m16int	11011 111 : mod^A 001 r/m
FISTTP—Store ST in int32 (chop) and pop	
m32int	11011 011 : mod^A 001 r/m
FISTTP—Store ST in int64 (chop) and pop	

Table B-31. Formats and Encodings for SSE3 Integer and Move Instructions (Contd.)

Instruction and Format	Encoding
m64int	11011 101 : mod[A] 001 r/m
LDDQU—Load unaligned integer 128-bit	
xmm, m128	11110010:00001111:11110000: mod[A] xmmreg r/m
MOVDDUP—Move 64 bits representing one DP data from XMM2/Mem to XMM1 and duplicate	
xmmreg2 to xmmreg1	11110010:00001111:00010010:11 xmmreg1 xmmreg2
mem to xmmreg	11110010:00001111:00010010: mod xmmreg r/m
MOVSHDUP—Move 128 bits representing 4 SP data from XMM2/Mem to XMM1 and duplicate high	
xmmreg2 to xmmreg1	11110011:00001111:00010110:11 xmmreg1 xmmreg2
mem to xmmreg	11110011:00001111:00010110: mod xmmreg r/m
MOVSLDUP—Move 128 bits representing 4 SP data from XMM2/Mem to XMM1 and duplicate low	
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
PABSB—Packed Absolute Value Bytes	
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
PABSD—Packed Absolute Value Double Words	
mmreg2 to mmreg1	0000 1111:0011 1000: 0001 1110:11 mmreg1 mmreg2
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
PABSW—Packed Absolute Value Words	
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
PALIGNR—Packed Align Right	
mmreg2 to mmreg1, imm8	0000 1111:0011 1010: 0000 1111:11 mmreg1 mmreg2: imm8

Table B-32. Formats and Encodings for SSSE3 Instructions (Contd.)

Instruction and Format	Encoding
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
PHADDD—Packed Horizontal Add Double Words	
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
PHADDSW—Packed Horizontal Add and Saturate	
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
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0000 0011: mod xmmreg r/m
PHADDW—Packed Horizontal Add Words	
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
PHSUBD—Packed Horizontal Subtract Double Words	
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
PHSUBSW—Packed Horizontal Subtract and Saturate	
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
PHSUBW—Packed Horizontal Subtract Words	
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

Table B-32. Formats and Encodings for SSSE3 Instructions (Contd.)

Instruction and Format	Encoding
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0000 0101: mod xmmreg r/m
PMADDUBSW—Multiply and Add Packed Signed and Unsigned Bytes	
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
PMULHRSW—Packed Multiply HIgn with Round and Scale	
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
PSHUFB—Packed Shuffle Bytes	
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
PSIGNB—Packed Sign Bytes	
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
PSIGND—Packed Sign Double Words	
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
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
PSIGNW—Packed Sign Words	
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
AESDEC—Perform One Round of an AES Decryption Flow	
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
AESDECLAST—Perform Last Round of an AES Decryption Flow	
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
AESENC—Perform One Round of an AES Encryption Flow	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000:1101 1100:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000:1101 1100: mod xmmreg r/m
AESENCLAST—Perform Last Round of an AES Encryption Flow	
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
AESIMC—Perform the AES InvMixColumn Transformation	
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
AESKEYGENASSIST—AES Round Key Generation Assist	
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
PCLMULQDQ—Carry-Less Multiplication Quadword	
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

Instruction and Format	Encoding
CMOVcc—Conditional Move	

Table B-34. Special Case Instructions Promoted Using REX.W (Contd.)

Instruction and Format	Encoding
register2 to register1	0100 0R0B 0000 1111: 0100 tttn : 11 reg1 reg2
qwordregister2 to qwordregister1	0100 1R0B 0000 1111: 0100 tttn : 11 qwordreg1 qwordreg2
memory to register	0100 0RXB 0000 1111 : 0100 tttn : mod reg r/m
memory64 to qwordregister	0100 1RXB 0000 1111 : 0100 tttn : mod qwordreg r/m
CVTSD2SI—Convert Scalar Double Precision Floating-Point Value to Doubleword Integer	
xmmreg to r32	0100 0R0B 1111 0010:0000 1111:0010 1101:11 r32 xmmreg
xmmreg to r64	0100 1R0B 1111 0010:0000 1111:0010 1101:11 r64 xmmreg
mem64 to r32	0100 0R0XB 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
CVTSI2SS—Convert Doubleword Integer to Scalar Single Precision Floating-Point Value	
r32 to xmmreg1	0100 0R0B 1111 0011:0000 1111:0010 1010:11 xmmreg r32
r64 to xmmreg1	0100 1R0B 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
CVTSI2SD—Convert Doubleword Integer to Scalar Double Precision Floating-Point Value	
r32 to xmmreg1	0100 0R0B 1111 0010:0000 1111:0010 1010:11 xmmreg r32
r64 to xmmreg1	0100 1R0B 1111 0010:0000 1111:0010 1010:11 xmmreg r64
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
CVTSS2SI—Convert Scalar Single Precision Floating-Point Value to Doubleword Integer	
xmmreg to r32	0100 0R0B 1111 0011:0000 1111:0010 1101:11 r32 xmmreg
xmmreg to r64	0100 1R0B 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
CVTTSD2SI—Convert with Truncation Scalar Double Precision Floating-Point Value to Doubleword Integer	
xmmreg to r32	0100 0R0B 11110010:00001111:00101100:11 r32 xmmreg
xmmreg to r64	0100 1R0B 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

Table B-34. Special Case Instructions Promoted Using REX.W (Contd.)

Instruction and Format	Encoding
CVTTSS2SI—Convert with Truncation Scalar Single Precision Floating-Point Value to Doubleword Integer	
xmmreg to r32	0100 0R0B 1111 0011:0000 1111:0010 1100:11 r32 xmmreg1
xmmreg to r64	0100 1R0B 1111 0011:0000 1111:0010 1100:11 r64 xmmreg1
mem to r32	0100 0RXB 1111 0011:0000 1111:0010 1100: mod r32 r/m
mem32 to r64	0100 1RXB 1111 0011:0000 1111:0010 1100: mod r64 r/m
MOVD/MOVQ—Move doubleword	
reg to mmxreg	0100 0R0B 0000 1111:0110 1110: 11 mmxreg reg
qwordreg to mmxreg	0100 1R0B 0000 1111:0110 1110: 11 mmxreg qwordreg
reg from mmxreg	0100 0R0B 0000 1111:0111 1110: 11 mmxreg reg
qwordreg from mmxreg	0100 1R0B 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
MOVMSKPS—Extract Packed Single Precision Floating-Point Sign Mask	
xmmreg to r32	0100 0R0B 0000 1111:0101 0000:11 r32 xmmreg
xmmreg to r64	0100 1R0B 00001111:01010000:11 r64 xmmreg
PEXTRW—Extract Word	
mmreg to reg32, imm8	0100 0R0B 0000 1111:1100 0101:11 r32 mmreg: imm8
mmreg to reg64, imm8	0100 1R0B 0000 1111:1100 0101:11 r64 mmreg: imm8
xmmreg to reg32, imm8	0100 0R0B 0110 0110 0000 1111:1100 0101:11 r32 xmmreg: imm8
xmmreg to reg64, imm8	0100 1R0B 0110 0110 0000 1111:1100 0101:11 r64 xmmreg: imm8
PINSRW—Insert Word	
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
PMOVMSKB—Move Byte Mask To Integer	

Table B-34. Special Case Instructions Promoted Using REX.W (Contd.)

Instruction and Format	Encoding
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
BLENDPD — Blend Packed Double Precision Floats	
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
BLENDPS — Blend Packed Single Precision Floats	
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
BLENDVPD — Variable Blend Packed Double Precision Floats	
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
BLENDVPS — Variable Blend Packed Single Precision Floats	
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
DPPD — Packed Double Precision Dot Products	
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
DPPS — Packed Single Precision Dot Products	
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
EXTRACTPS — Extract From Packed Single Precision Floats	
reg from xmmreg , imm8	0110 0110:0000 1111:0011 1010: 0001 0111:11 xmmreg reg: imm8

Table B-35. Encodings of SSE4.1 instructions

Instruction and Format	Encoding
mem from xmmreg , imm8	0110 0110:0000 1111:0011 1010: 0001 0111: mod xmmreg r/m: imm8
INSERTPS — Insert Into Packed Single Precision Floats	
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
MOVNTDQA — Load Double Quadword Non-temporal Aligned	
m128 to xmmreg	0110 0110:0000 1111:0011 1000: 0010 1010:11 r/m xmmreg2
MPSADBW — Multiple Packed Sums of Absolute Difference	
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
PACKUSDW — Pack with Unsigned Saturation	
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
PBLENDVB — Variable Blend Packed Bytes	
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
PBLENDW — Blend Packed Words	
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
PCMPEQQ — Compare Packed Qword Data of Equal	
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
PEXTRB — Extract Byte	
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
PEXTRD — Extract DWord	
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

Table B-35. Encodings of SSE4.1 instructions

Instruction and Format	Encoding
PEXTRQ — Extract QWord	
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
PEXTRW — Extract Word	
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
PHMINPOSUW — Packed Horizontal Word Minimum	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000: 0100 0001:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0100 0001: mod xmmreg r/m
PINSRB — Extract Byte	
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
PINSRD — Extract DWord	
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
PINSRQ — Extract QWord	
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
PMAXSB — Maximum of Packed Signed Byte Integers	
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
PMAXSD — Maximum of Packed Signed Dword Integers	
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
PMAXUD — Maximum of Packed Unsigned Dword Integers	
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
PMAXUW — Maximum of Packed Unsigned Word Integers	

Table B-35. Encodings of SSE4.1 instructions

Instruction and Format	Encoding
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
PMINSB — Minimum of Packed Signed Byte Integers	
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
PMINSD — Minimum of Packed Signed Dword Integers	
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
PMINUD — Minimum of Packed Unsigned Dword Integers	
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
PMINUW — Minimum of Packed Unsigned Word Integers	
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
PMOVSXBD — Packed Move Sign Extend - Byte to Dword	
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
PMOVSXBQ — Packed Move Sign Extend - Byte to Qword	
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
PMOVSXBW — Packed Move Sign Extend - Byte to Word	
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
PMOVSXWD — Packed Move Sign Extend - Word to Dword	
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
PMOVSXWQ — Packed Move Sign Extend - Word to Qword	
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
PMOVSXDQ — Packed Move Sign Extend - Dword to Qword	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000: 0010 0101:11 xmmreg1 xmmreg2

Table B-35. Encodings of SSE4.1 instructions

Instruction and Format	Encoding
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0010 0101: mod xmmreg r/m
PMOVZXBD — Packed Move Zero Extend - Byte to Dword	
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
PMOVZXBQ — Packed Move Zero Extend - Byte to Qword	
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
PMOVZXBW — Packed Move Zero Extend - Byte to Word	
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
PMOVZXWD — Packed Move Zero Extend - Word to Dword	
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
PMOVZXWQ — Packed Move Zero Extend - Word to Qword	
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
PMOVZXDQ — Packed Move Zero Extend - Dword to Qword	
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
PMULDQ — Multiply Packed Signed Dword Integers	
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
PMULLD — Multiply Packed Signed Dword Integers, Store low Result	
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
PTEST — Logical Compare	
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
ROUNDPD — Round Packed Double Precision Values	
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

Table B-35. Encodings of SSE4.1 instructions

Instruction and Format	Encoding
ROUNDPS — Round Packed Single Precision Values	
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
ROUNDSD — Round Scalar Double Precision Value	
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
ROUNDSS — Round Scalar Single Precision Value	
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
CRC32 — Accumulate CRC32	
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 WR0B:0000 1111:0011 1000: 1111 0000 :11 reg1 bytereg2
m8 to reg	1111 0010:0100 WR0B:0000 1111:0011 1000: 1111 0000 : mod reg r/m
qwreg2 to qwreg1	1111 0010:0100 1R0B:0000 1111:0011 1000: 1111 0001 :11 qwreg1 qwreg2
mem64 to qwreg	1111 0010:0100 1R0B:0000 1111:0011 1000: 1111 0001 : mod qwreg r/m
PCMPESTRI— Packed Compare Explicit-Length Strings To Index	
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
PCMPESTRM— Packed Compare Explicit-Length Strings To Mask	
xmmreg2 to xmmreg1, imm8	0110 0110:0000 1111:0011 1010: 0110 0000:11 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 0000: mod xmmreg r/m
PCMPISTRI— Packed Compare Implicit-Length String To Index	
xmmreg2 to xmmreg1, imm8	0110 0110:0000 1111:0011 1010: 0110 0011:11 xmmreg1 xmmreg2: imm8
mem to xmmreg	0110 0110:0000 1111:0011 1010: 0110 0011: mod xmmreg r/m
PCMPISTRM— Packed Compare Implicit-Length Strings To Mask	
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
PCMPGTQ— Packed Compare Greater Than	
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
POPCNT— Return Number of Bits Set to 1	
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:

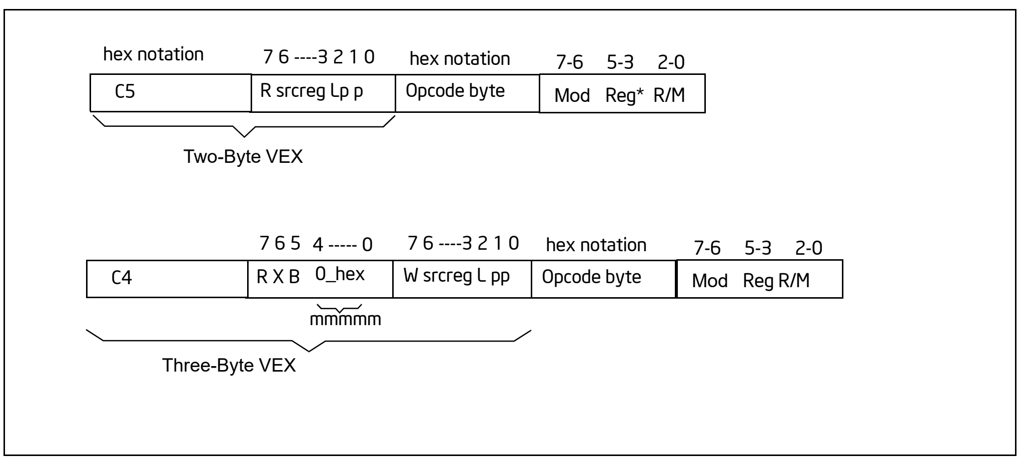

Figure B-2. Hybrid Notation of VEX-Encoded Key Instruction Bytes

Table B-37. Encodings of AVX instructions

Instruction and Format	Encoding
VBLENDPD — Blend Packed Double Precision Floats	
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
VBLENDPS — Blend Packed Single Precision Floats	
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
VBLENDVPD — Variable Blend Packed Double Precision Floats	
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
ymmreg2 with ymmreg3 into ymmreg1 using ymmreg4 as mask	C4: rxb0_3: 0 ymmreg2 101:4B:11 ymmreg1 ymmreg3: ymmreg4
ymmreg2 with mem to ymmreg1 using ymmreg4 as mask	C4: rxb0_3: 0 ymmreg2 101:4B:mod ymmreg1 r/m: ymmreg4
VBLENDVPS — Variable Blend Packed Single Precision Floats	
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
ymmreg2 with ymmreg3 into ymmreg1 using ymmreg4 as mask	C4: rxb0_3: 0 ymmreg2 101:4A:11 ymmreg1 ymmreg3: ymmreg4
ymmreg2 with mem to ymmreg1 using ymmreg4 as mask	C4: rxb0_3: 0 ymmreg2 101:4A:mod ymmreg1 r/m: ymmreg4
VDPPD — Packed Double Precision Dot Products	
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
VDPPS — Packed Single Precision Dot Products	
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
ymmreg2 with ymmreg3 into ymmreg1	C4: rxb0_3: w ymmreg2 101:40:11 ymmreg1 ymmreg3: imm
ymmreg2 with mem to ymmreg1	C4: rxb0_3: w ymmreg2 101:40:mod ymmreg1 r/m: imm
VEXTRACTPS — Extract From Packed Single Precision Floats	
reg from xmmreg1 using imm	C4: rxb0_3: w_F 001:17:11 xmmreg1 reg: imm
mem from xmmreg1 using imm	C4: rxb0_3: w_F 001:17:mod xmmreg1 r/m: imm
VINSERTPS — Insert Into Packed Single Precision Floats	
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
VMOVNTDQA — Load Double Quadword Non-temporal Aligned	
m128 to xmmreg1	C4: rxb0_2: w_F 001:2A:11 xmmreg1 r/m

Instruction and Format	Encoding
VPMPSADBW — Multiple Packed Sums of Absolute Difference	
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
VPACKUSDW — Pack with Unsigned Saturation	
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
VPBLENDVB — Variable Blend Packed Bytes	
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
VPBLENDW — Blend Packed Words	
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
VPCMPEQQ — Compare Packed Qword Data of Equal	
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:
VPEXTRB — Extract Byte	
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
VPEXTRD — Extract DWord	
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
VPEXTRQ — Extract QWord	
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
VPEXTRW — Extract Word	
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
VPHMINPOSUW — Packed Horizontal Word Minimum	
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
VPINSRB — Insert Byte	
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
VPINSRD — Insert DWord	
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
VPINSRQ — Insert QWord	
r64 with xmmreg2 to xmmreg1, imm8	C4: rxb0_3: 1 xmmreg2 001:22:11 xmmreg1 reg: imm

Instruction and Format	Encoding
m64 with xmmreg2 to xmmreg1, imm8	C4: rxb0_3: 1 xmmreg2 001:22:mod xmmreg1 r/m: imm
VPMAXSB — Maximum of Packed Signed Byte Integers	
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
VPMAXSD — Maximum of Packed Signed Dword Integers	
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
VPMAXUD — Maximum of Packed Unsigned Dword Integers	
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
VPMAXUW — Maximum of Packed Unsigned Word Integers	
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
VPMINSB — Minimum of Packed Signed Byte Integers	
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
VPMINSD — Minimum of Packed Signed Dword Integers	
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
VPMINUD — Minimum of Packed Unsigned Dword Integers	
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
VPMINUW — Minimum of Packed Unsigned Word Integers	
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
VPMOVSXBD — Packed Move Sign Extend - Byte to Dword	
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
VPMOVSXBQ — Packed Move Sign Extend - Byte to Qword	
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
VPMOVSXBW — Packed Move Sign Extend - Byte to Word	
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
VPMOVSXWD — Packed Move Sign Extend - Word to Dword	
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
VPMOVSXWQ — Packed Move Sign Extend - Word to Qword	
xmmreg2 to xmmreg1	C4: rxb0_2: w_F 001:24:11 xmmreg1 xmmreg2

Instruction and Format	Encoding
mem to xmmreg1	C4: rxb0_2: w_F 001:24:mod xmmreg1 r/m
VPMOVSXDQ — Packed Move Sign Extend - Dword to Qword	
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
VPMOVZXBD — Packed Move Zero Extend - Byte to Dword	
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
VPMOVZXBQ — Packed Move Zero Extend - Byte to Qword	
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
VPMOVZXBW — Packed Move Zero Extend - Byte to Word	
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
VPMOVZXWD — Packed Move Zero Extend - Word to Dword	
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
VPMOVZXWQ — Packed Move Zero Extend - Word to Qword	
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
VPMOVZXDQ — Packed Move Zero Extend - Dword to Qword	
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
VPMULDQ — Multiply Packed Signed Dword Integers	
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
VPMULLD — Multiply Packed Signed Dword Integers, Store low Result	
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
VPTEST — Logical Compare	
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
ymmreg2 to ymmreg1	C4: rxb0_2: w_F 101:17:11 ymmreg1 ymmreg2
mem to ymmreg	C4: rxb0_2: w_F 101:17:mod ymmreg1 r/m
VROUNDPD — Round Packed Double Precision Values	
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
ymmreg2 to ymmreg1, imm8	C4: rxb0_3: w_F 101:09:11 ymmreg1 ymmreg2: imm
mem to ymmreg1, imm8	C4: rxb0_3: w_F 101:09:mod ymmreg1 r/m: imm
VROUNDPS — Round Packed Single Precision Values	

Instruction and Format	Encoding
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
VROUNDSD — Round Scalar Double Precision Value	
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
VROUNDSS — Round Scalar Single Precision Value	
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
VPCMPESTRI — Packed Compare Explicit Length Strings, Return Index	
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
VPCMPESTRM — Packed Compare Explicit Length Strings, Return Mask	
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
VPCMPGTQ — Compare Packed Data for Greater Than	
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
VPCMPISTRI — Packed Compare Implicit Length Strings, Return Index	
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
VPCMPISTRM — Packed Compare Implicit Length Strings, Return Mask	
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
VAESDEC — Perform One Round of an AES Decryption Flow	
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
VAESDECLAST — Perform Last Round of an AES Decryption Flow	
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
VAESENC — Perform One Round of an AES Encryption Flow	
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
VAESENCLAST — Perform Last Round of an AES Encryption Flow	
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
VAESIMC — Perform the AES InvMixColumn Transformation	
xmmreg2 to xmmreg1	C4: rxb0_2: w_F 001:DB:11 xmmreg1 xmmreg2

Instruction and Format	Encoding
mem to xmmreg1	C4: rxb0_2: w_F 001:DB:mod xmmreg1 r/m
VAESKEYGENASSIST — AES Round Key Generation Assist	
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
VPABSB — Packed Absolute Value	
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
VPABSD — Packed Absolute Value	
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
VPABSW — Packed Absolute Value	
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
VPALIGNR — Packed Align Right	
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
VPHADDD — Packed Horizontal Add	
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
VPHADDW — Packed Horizontal Add	
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
VPHADDSW — Packed Horizontal Add and Saturate	
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
VPHSUBD — Packed Horizontal Subtract	
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
VPHSUBW — Packed Horizontal Subtract	
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
VPHSUBSW — Packed Horizontal Subtract and Saturate	
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
VPMADDUBSW — Multiply and Add Packed Signed and Unsigned Bytes	
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
VPMULHRSW — Packed Multiply High with Round and Scale	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_2: w xmmreg2 001:0B:11 xmmreg1 xmmreg3

Instruction and Format	Encoding
xmmreg2 with mem to xmmreg1	C4: rxb0_2: w xmmreg2 001:0B:mod xmmreg1 r/m
VPSHUFB — Packed Shuffle Bytes	
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
VPSIGNB — Packed SIGN	
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
VPSIGND — Packed SIGN	
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
VPSIGNW — Packed SIGN	
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
VADDSUBPD — Packed Double-FP Add/Subtract	
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[1] 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
ymmreg2 with ymmreg3 to ymmreg1	C4: rxb0_1: w ymmreg2 101:D0:11 ymmreg1 ymmreg3
ymmreg2 with mem to ymmreg1	C4: rxb0_1: w ymmreg2 101:D0:mod ymmreg1 r/m
ymmreglo2 with ymmreglo3 to ymmreg1	C5: r_ymmreglo2 101:D0:11 ymmreg1 ymmreglo3
ymmreglo2 with mem to ymmreg1	C5: r_ymmreglo2 101:D0:mod ymmreg1 r/m
VADDSUBPS — Packed Single-FP Add/Subtract	
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
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
VHADDPD — Packed Double-FP Horizontal Add	
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

Instruction and Format	Encoding
ymmreglo2 with mem to ymmreg1	C5: r_ymmreglo2 101:7C:mod ymmreg1 r/m
VHADDPS — Packed Single-FP Horizontal Add	
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
VHSUBPD — Packed Double-FP Horizontal Subtract	
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
ymmreg2 with ymmreg3 to ymmreg1	C4: rxb0_1: w ymmreg2 101:7D:11 ymmreg1 ymmreg3
ymmreg2 with mem to ymmreg1	C4: rxb0_1: w ymmreg2 101:7D:mod ymmreg1 r/m
ymmreglo2 with ymmreglo3 to ymmreg1	C5: r_ymmreglo2 101:7D:11 ymmreg1 ymmreglo3
ymmreglo2 with mem to ymmreg1	C5: r_ymmreglo2 101:7D:mod ymmreg1 r/m
VHSUBPS — Packed Single-FP Horizontal Subtract	
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
ymmreg2 with ymmreg3 to ymmreg1	C4: rxb0_1: w ymmreg2 111:7D:11 ymmreg1 ymmreg3
ymmreg2 with mem to ymmreg1	C4: rxb0_1: w ymmreg2 111:7D:mod ymmreg1 r/m
ymmreglo2 with ymmreglo3 to ymmreg1	C5: r_ymmreglo2 111:7D:11 ymmreg1 ymmreglo3
ymmreglo2 with mem to ymmreg1	C5: r_ymmreglo2 111:7D:mod ymmreg1 r/m
VLDDQU — Load Unaligned Integer 128 Bits	
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 ymmreg1	C4: rxb0_1: w_F 111:F0:mod ymmreg1 r/m
mem to ymmreg1	C5: r_F 111:F0:mod ymmreg1 r/m
VMOVDDUP — Move One Double-FP and Duplicate	
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
mem to xmmreg1	C5: r_F 011:12:mod xmmreg1 r/m
ymmreg2 to ymmreg1	C4: rxb0_1: w_F 111:12:11 ymmreg1 ymmreg2

Instruction and Format	Encoding
mem to ymmreg1	C4: rxb0_1: w_F 111:12:mod ymmreg1 r/m
ymmreglo to ymmreg1	C5: r_ F 111:12:11 ymmreg1 ymmreglo
mem to ymmreg1	C5: r_F 111:12:mod ymmreg1 r/m
VMOVHLPS — Move Packed Single Precision Floating-Point Values High to Low	
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
VMOVSHDUP — Move Packed Single-FP High and Duplicate	
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
ymmreg2 to ymmreg1	C4: rxb0_1: w_F 110:16:11 ymmreg1 ymmreg2
mem to ymmreg1	C4: rxb0_1: w_F 110:16:mod ymmreg1 r/m
ymmreglo to ymmreg1	C5: r_F 110:16:11 ymmreg1 ymmreglo
mem to ymmreg1	C5: r_F 110:16:mod ymmreg1 r/m
VMOVSLDUP — Move Packed Single-FP Low and Duplicate	
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
ymmreg2 to ymmreg1	C4: rxb0_1: w_F 110:12:11 ymmreg1 ymmreg2
mem to ymmreg1	C4: rxb0_1: w_F 110:12:mod ymmreg1 r/m
ymmreglo to ymmreg1	C5: r_F 110:12:11 ymmreg1 ymmreglo
mem to ymmreg1	C5: r_F 110:12:mod ymmreg1 r/m
VADDPD — Add Packed Double Precision Floating-Point Values	
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
VADDSD — Add Scalar Double Precision Floating-Point Values	
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
VANDPD — Bitwise Logical AND of Packed Double Precision Floating-Point Values	

Instruction and Format	Encoding
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
VANDNPD — Bitwise Logical AND NOT of Packed Double Precision Floating-Point Values	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:55:11 xmmreg1 xmmreg3
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
ymmreg2 with ymmreg3 to ymmreg1	C4: rxb0_1: w ymmreg2 101:55:11 ymmreg1 ymmreg3
ymmreg2 with mem to ymmreg1	C4: rxb0_1: w ymmreg2 101:55:mod ymmreg1 r/m
ymmreglo2 with ymmreglo3 to ymmreg1	C5: r_ymmreglo2 101:55:11 ymmreg1 ymmreglo3
ymmreglo2 with mem to ymmreg1	C5: r_ymmreglo2 101:55:mod ymmreg1 r/m
VCMPPD — Compare Packed Double Precision Floating-Point Values	
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
ymmreg2 with ymmreg3 to ymmreg1	C4: rxb0_1: w ymmreg2 101:C2:11 ymmreg1 ymmreg3: imm
ymmreg2 with mem to ymmreg1	C4: rxb0_1: w ymmreg2 101:C2:mod ymmreg1 r/m: imm
ymmreglo2 with ymmreglo3 to ymmreg1	C5: r_ymmreglo2 101:C2:11 ymmreg1 ymmreglo3: imm
ymmreglo2 with mem to ymmreg1	C5: r_ymmreglo2 101:C2:mod ymmreg1 r/m: imm
VCMPSD — Compare Scalar Double Precision Floating-Point Values	
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
VCOMISD — Compare Scalar Ordered Double Precision Floating-Point Values and Set EFLAGS	
xmmreg2 to xmmreg1	C4: rxb0_1: w_F 001:2F:11 xmmreg1 xmmreg2
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
VCVTDQ2PD— Convert Packed Dword Integers to Packed Double Precision FP Values	
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

Instruction and Format	Encoding
xmmreglo to xmmreg1	C5: r_F 010:E6:11 xmmreg1 xmmreglo
mem to xmmreg1	C5: r_F 010:E6:mod xmmreg1 r/m
ymmreg2 to ymmreg1	C4: rxb0_1: w_F 110:E6:11 ymmreg1 ymmreg2
mem to ymmreg1	C4: rxb0_1: w_F 110:E6:mod ymmreg1 r/m
ymmreglo to ymmreg1	C5: r_F 110:E6:11 ymmreg1 ymmreglo
mem to ymmreg1	C5: r_F 110:E6:mod ymmreg1 r/m
VCVTDQ2PS— Convert Packed Dword Integers to Packed Single Precision FP Values	
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
ymmreg2 to ymmreg1	C4: rxb0_1: w_F 100:5B:11 ymmreg1 ymmreg2
mem to ymmreg1	C4: rxb0_1: w_F 100:5B:mod ymmreg1 r/m
ymmreglo to ymmreg1	C5: r_F 100:5B:11 ymmreg1 ymmreglo
mem to ymmreg1	C5: r_F 100:5B:mod ymmreg1 r/m
VCVTPD2DQ— Convert Packed Double Precision FP Values to Packed Dword Integers	
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
ymmreg2 to ymmreg1	C4: rxb0_1: w_F 111:E6:11 ymmreg1 ymmreg2
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
VCVTPD2PS— Convert Packed Double Precision FP Values to Packed Single Precision FP Values	
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
VCVTPS2DQ— Convert Packed Single Precision FP Values to Packed Dword Integers	
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

Instruction and Format	Encoding
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
VCVTPS2PD— Convert Packed Single Precision FP Values to Packed Double Precision FP Values	
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
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
VCVTSD2SI— Convert Scalar Double Precision FP Value to Integer	
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
VCVTSD2SS — Convert Scalar Double Precision FP Value to Scalar Single Precision FP Value	
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
VCVTSI2SD— Convert Dword Integer to Scalar Double Precision FP Value	
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
VCVTSS2SD — Convert Scalar Single Precision FP Value to Scalar Double Precision FP Value	
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
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 010:5A:mod xmmreg1 r/m
VCVTTPD2DQ— Convert with Truncation Packed Double Precision FP Values to Packed Dword Integers	
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

Instruction and Format	Encoding
xmmreglo to xmmreg1	C5: r_F 001:E6:11 xmmreg1 xmmreglo
mem to xmmreg1	C5: r_F 001:E6:mod xmmreg1 r/m
ymmreg2 to ymmreg1	C4: rxb0_1: w_F 101:E6:11 ymmreg1 ymmreg2
mem to ymmreg1	C4: rxb0_1: w_F 101:E6:mod ymmreg1 r/m
ymmreglo to ymmreg1	C5: r_F 101:E6:11 ymmreg1 ymmreglo
mem to ymmreg1	C5: r_F 101:E6:mod ymmreg1 r/m
VCVTTPS2DQ— Convert with Truncation Packed Single Precision FP Values to Packed Dword Integers	
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
ymmreg2 to ymmreg1	C4: rxb0_1: w_F 110:5B:11 ymmreg1 ymmreg2
mem to ymmreg1	C4: rxb0_1: w_F 110:5B:mod ymmreg1 r/m
ymmreglo to ymmreg1	C5: r_F 110:5B:11 ymmreg1 ymmreglo
mem to ymmreg1	C5: r_F 110:5B:mod ymmreg1 r/m
VCVTTSD2SI— Convert with Truncation Scalar Double Precision FP Value to Signed Integer	
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
VDIVPD — Divide Packed Double Precision Floating-Point Values	
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
ymmreg2 with ymmreg3 to ymmreg1	C4: rxb0_1: w ymmreg2 101:5E:11 ymmreg1 ymmreg3
ymmreg2 with mem to ymmreg1	C4: rxb0_1: w ymmreg2 101:5E:mod ymmreg1 r/m
ymmreglo2 with ymmreglo3 to ymmreg1	C5: r_ymmreglo2 101:5E:11 ymmreg1 ymmreglo3
ymmreglo2 with mem to ymmreg1	C5: r_ymmreglo2 101:5E:mod ymmreg1 r/m
VDIVSD — Divide Scalar Double Precision Floating-Point Values	
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
VMASKMOVDQU— Store Selected Bytes of Double Quadword	
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

Instruction and Format	Encoding
VMAXPD — Return Maximum Packed Double Precision Floating-Point Values	
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
ymmreg2 with ymmreg3 to ymmreg1	C4: rxb0_1: w ymmreg2 101:5F:11 ymmreg1 ymmreg3
ymmreg2 with mem to ymmreg1	C4: rxb0_1: w ymmreg2 101:5F:mod ymmreg1 r/m
ymmreglo2 with ymmreglo3 to ymmreg1	C5: r_ymmreglo2 101:5F:11 ymmreg1 ymmreglo3
ymmreglo2 with mem to ymmreg1	C5: r_ymmreglo2 101:5F:mod ymmreg1 r/m
VMAXSD — Return Maximum Scalar Double Precision Floating-Point Value	
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
VMINPD — Return Minimum Packed Double Precision Floating-Point Values	
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
ymmreg2 with ymmreg3 to ymmreg1	C4: rxb0_1: w ymmreg2 101:5D:11 ymmreg1 ymmreg3
ymmreg2 with mem to ymmreg1	C4: rxb0_1: w ymmreg2 101:5D:mod ymmreg1 r/m
ymmreglo2 with ymmreglo3 to ymmreg1	C5: r_ymmreglo2 101:5D:11 ymmreg1 ymmreglo3
ymmreglo2 with mem to ymmreg1	C5: r_ymmreglo2 101:5D:mod ymmreg1 r/m
VMINSD — Return Minimum Scalar Double Precision Floating-Point Value	
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
VMOVAPD — Move Aligned Packed Double Precision Floating-Point Values	
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
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
ymmreg2 to ymmreg1	C4: rxb0_1: w_F 101:28:11 ymmreg1 ymmreg2
mem to ymmreg1	C4: rxb0_1: w_F 101:28:mod ymmreg1 r/m

Instruction and Format	Encoding
ymmreglo to ymmreg1	C5: r_F 101:28:11 ymmreg1 ymmreglo
mem to ymmreg1	C5: r_F 101:28:mod ymmreg1 r/m
ymmreg1 to ymmreg2	C4: rxb0_1: w_F 101:29:11 ymmreg2 ymmreg1
ymmreg1 to mem	C4: rxb0_1: w_F 101:29:mod r/m ymmreg1
ymmreg1 to ymmreglo	C5: r_F 101:29:11 ymmreglo ymmreg1
ymmreg1 to mem	C5: r_F 101:29:mod r/m ymmreg1
VMOVD — Move Doubleword	
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
VMOVQ — Move Quadword	
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
VMOVDQA — Move Aligned Double Quadword	
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
xmmreglo to xmmreg1	C5: r_F 001:6F:11 xmmreg1 xmmreglo
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 xmmreglo	C5: r_F 001:7F:11 xmmreglo xmmreg1
xmmreg1 to mem	C5: r_F 001:7F:mod r/m xmmreg1
ymmreg2 to ymmreg1	C4: rxb0_1: w_F 101:6F:11 ymmreg1 ymmreg2
mem to ymmreg1	C4: rxb0_1: w_F 101:6F:mod ymmreg1 r/m
ymmreglo to ymmreg1	C5: r_F 101:6F:11 ymmreg1 ymmreglo
mem to ymmreg1	C5: r_F 101:6F:mod ymmreg1 r/m
ymmreg1 to ymmreg2	C4: rxb0_1: w_F 101:7F:11 ymmreg2 ymmreg1
ymmreg1 to mem	C4: rxb0_1: w_F 101:7F:mod r/m ymmreg1
ymmreg1 to ymmreglo	C5: r_F 101:7F:11 ymmreglo ymmreg1
ymmreg1 to mem	C5: r_F 101:7F:mod r/m ymmreg1
VMOVDQU — Move Unaligned Double Quadword	
xmmreg2 to xmmreg1	C4: rxb0_1: w_F 010:6F:11 xmmreg1 xmmreg2

Instruction and Format	Encoding
mem to xmmreg1	C4: rxb0_1: w_F 010:6F:mod xmmreg1 r/m
xmmreglo to xmmreg1	C5: r_F 010:6F:11 xmmreg1 xmmreglo
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 xmmreglo	C5: r_F 010:7F:11 xmmreglo xmmreg1
xmmreg1 to mem	C5: r_F 010:7F:mod r/m xmmreg1
ymmreg2 to ymmreg1	C4: rxb0_1: w_F 110:6F:11 ymmreg1 ymmreg2
mem to ymmreg1	C4: rxb0_1: w_F 110:6F:mod ymmreg1 r/m
ymmreglo to ymmreg1	C5: r_F 110:6F:11 ymmreg1 ymmreglo
mem to ymmreg1	C5: r_F 110:6F:mod ymmreg1 r/m
ymmreg1 to ymmreg2	C4: rxb0_1: w_F 110:7F:11 ymmreg2 ymmreg1
ymmreg1 to mem	C4: rxb0_1: w_F 110:7F:mod r/m ymmreg1
ymmreg1 to ymmreglo	C5: r_F 110:7F:11 ymmreglo ymmreg1
ymmreg1 to mem	C5: r_F 110:7F:mod r/m ymmreg1
VMOVHPD — Move High Packed Double Precision Floating-Point Value	
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
VMOVLPD — Move Low Packed Double Precision Floating-Point Value	
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
VMOVMSKPD — Extract Packed Double Precision Floating-Point Sign Mask	
xmmreg2 to reg	C4: rxb0_1: w_F 001:50:11 reg xmmreg1
xmmreglo to reg	C5: r_F 001:50:11 reg xmmreglo
ymmreg2 to reg	C4: rxb0_1: w_F 101:50:11 reg ymmreg1
ymmreglo to reg	C5: r_F 101:50:11 reg ymmreglo
VMOVNTDQ — Store Double Quadword Using Non-Temporal Hint	
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
ymmreg1 to mem	C4: rxb0_1: w_F 101:E7:11 r/m ymmreg1
ymmreglo to mem	C5: r_F 101:E7:11 r/m ymmreglo
VMOVNTPD — Store Packed Double Precision Floating-Point Values Using Non-Temporal Hint	
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
ymmreg1 to mem	C4: rxb0_1: w_F 101:2B:11r/m ymmreg1

Instruction and Format	Encoding
ymmreglo to mem	C5: r_F 101:2B:11r/m ymmreglo
VMOVSD — Move Scalar Double Precision Floating-Point Value	
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
VMOVUPD — Move Unaligned Packed Double Precision Floating-Point Values	
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
ymmreg2 to ymmreg1	C4: rxb0_1: w_F 101:10:11 ymmreg1 ymmreg2
mem to ymmreg1	C4: rxb0_1: w_F 101:10:mod ymmreg1 r/m
ymmreglo to ymmreg1	C5: r_F 101:10:11 ymmreg1 ymmreglo
mem to ymmreg1	C5: r_F 101:10:mod ymmreg1 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
ymmreg1 to ymmreg2	C4: rxb0_1: w_F 101:11:11 ymmreg2 ymmreg1
ymmreg1 to mem	C4: rxb0_1: w_F 101:11:mod r/m ymmreg1
ymmreg1 to ymmreglo	C5: r_F 101:11:11 ymmreglo ymmreg1
ymmreg1 to mem	C5: r_F 101:11:mod r/m ymmreg1
VMULPD — Multiply Packed Double Precision Floating-Point Values	
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
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
ymmreg2 with ymmreg3 to ymmreg1	C4: rxb0_1: w ymmreg2 101:59:11 ymmreg1 ymmreg3
ymmreg2 with mem to ymmreg1	C4: rxb0_1: w ymmreg2 101:59:mod ymmreg1 r/m
ymmreglo2 with ymmreglo3 to ymmreg1	C5: r_ymmreglo2 101:59:11 ymmreg1 ymmreglo3
ymmreglo2 with mem to ymmreg1	C5: r_ymmreglo2 101:59:mod ymmreg1 r/m
VMULSD — Multiply Scalar Double Precision Floating-Point Values	
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

Instruction and Format	Encoding
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
VORPD — Bitwise Logical OR of Double Precision Floating-Point Values	
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
ymmreg2 with ymmreg3 to ymmreg1	C4: rxb0_1: w ymmreg2 101:56:11 ymmreg1 ymmreg3
ymmreg2 with mem to ymmreg1	C4: rxb0_1: w ymmreg2 101:56:mod ymmreg1 r/m
ymmreglo2 with ymmreglo3 to ymmreg1	C5: r_ymmreglo2 101:56:11 ymmreg1 ymmreglo3
ymmreglo2 with mem to ymmreg1	C5: r_ymmreglo2 101:56:mod ymmreg1 r/m
VPACKSSWB— Pack with Signed Saturation	
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
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:63:mod xmmreg1 r/m
VPACKSSDW— Pack with Signed Saturation	
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
VPACKUSWB— Pack with Unsigned Saturation	
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
VPADDB — Add Packed Integers	
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
VPADDW — Add Packed Integers	
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
VPADDD — Add Packed Integers	
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

Instruction and Format	Encoding
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
VPADDQ — Add Packed Quadword Integers	
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
VPADDSB — Add Packed Signed Integers with Signed Saturation	
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
VPADDSW — Add Packed Signed Integers with Signed Saturation	
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
VPADDUSB — Add Packed Unsigned Integers with Unsigned Saturation	
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
VPADDUSW — Add Packed Unsigned Integers with Unsigned Saturation	
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
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:DD:mod xmmreg1 r/m
VPAND — Logical AND	
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
VPANDN — Logical AND NOT	
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
VPAVGB — Average Packed Integers	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:E0:11 xmmreg1 xmmreg3

Instruction and Format	Encoding
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
VPAVGW — Average Packed Integers	
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
VPCMPEQB — Compare Packed Data for Equal	
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
VPCMPEQW — Compare Packed Data for Equal	
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
VPCMPEQD — Compare Packed Data for Equal	
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
VPCMPGTB — Compare Packed Signed Integers for Greater Than	
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
VPCMPGTW — Compare Packed Signed Integers for Greater Than	
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
VPCMPGTD — Compare Packed Signed Integers for Greater Than	
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
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:66:mod xmmreg1 r/m
VPEXTRW — Extract Word	

Instruction and Format	Encoding
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
VPINSRW — Insert Word	
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
VPMADDWD — Multiply and Add Packed Integers	
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
VPMAXSW — Maximum of Packed Signed Word Integers	
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
VPMAXUB — Maximum of Packed Unsigned Byte Integers	
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
VPMINSW — Minimum of Packed Signed Word Integers	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 001:EA:11 xmmreg1 xmmreg3
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
VPMINUB — Minimum of Packed Unsigned Byte Integers	
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
VPMOVMSKB — Move Byte Mask	
xmmreg1 to reg	C4: rxb0_1: w_F 001:D7:11 reg xmmreg1
xmmreg1 to reg	C5: r_F 001:D7:11 reg xmmreg1
VPMULHUW — Multiply Packed Unsigned Integers and Store High Result	
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

Instruction and Format	Encoding
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:E4:mod xmmreg1 r/m
VPMULHW — Multiply Packed Signed Integers and Store High Result	
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
VPMULLW — Multiply Packed Signed Integers and Store Low Result	
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
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
VPMULUDQ — Multiply Packed Unsigned Doubleword Integers	
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
VPOR — Bitwise Logical OR	
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
VPSADBW — Compute Sum of Absolute Differences	
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
VPSHUFD — Shuffle Packed Doublewords	
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
VPSHUFHW — Shuffle Packed High Words	
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
mem to xmmreg1 using imm	C5: r_F 010:70:mod xmmreg1 r/m: imm
VPSHUFLW — Shuffle Packed Low Words	
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

Instruction and Format	Encoding
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
VPSLLDQ — Shift Double Quadword Left Logical	
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
VPSLLW — Shift Packed Data Left Logical	
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
VPSLLD — Shift Packed Data Left Logical	
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
VPSLLQ — Shift Packed Data Left Logical	
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
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
VPSRAW — Shift Packed Data Right Arithmetic	
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
VPSRAD — Shift Packed Data Right Arithmetic	
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

Instruction and Format	Encoding
xmmreglo to xmmreg1 using imm8	C5: r_F 001:72:11 xmmreg1 xmmreglo: imm
VPSRLDQ — Shift Double Quadword Right Logical	
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
VPSRLW — Shift Packed Data Right Logical	
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
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
VPSRLD — Shift Packed Data Right Logical	
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
VPSRLQ — Shift Packed Data Right Logical	
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
VPSUBB — Subtract Packed Integers	
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
VPSUBW — Subtract Packed Integers	
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
xmmrelog2 with mem to xmmreg1	C5: r_xmmreglo2 001:F9:mod xmmreg1 r/m
VPSUBD — Subtract Packed Integers	
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

Instruction and Format	Encoding
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:FA:mod xmmreg1 r/m
VPSUBQ — Subtract Packed Quadword Integers	
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
VPSUBSB — Subtract Packed Signed Integers with Signed Saturation	
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
VPSUBSW — Subtract Packed Signed Integers with Signed Saturation	
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
VPSUBUSB — Subtract Packed Unsigned Integers with Unsigned Saturation	
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
VPSUBUSW — Subtract Packed Unsigned Integers with Unsigned Saturation	
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
VPUNPCKHBW — Unpack High Data	
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
VPUNPCKHWD — Unpack High Data	
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
VPUNPCKHDQ — Unpack High Data	
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

Instruction and Format	Encoding
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
VPUNPCKHQDQ — Unpack High Data	
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
VPUNPCKLBW — Unpack Low Data	
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
VPUNPCKLWD — Unpack Low Data	
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
VPUNPCKLDQ — Unpack Low Data	
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
VPUNPCKLQDQ — Unpack Low Data	
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
VPXOR — Logical Exclusive OR	
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
VSHUFPD — Shuffle Packed Double Precision Floating-Point Values	
xmmreg2 with xmmreg3 to xmmreg1 using imm8	C4: rxb0_1: w xmmreg2 001:C6:11 xmmreg1 xmmreg3: imm
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
ymmreg2 with ymmreg3 to ymmreg1 using imm8	C4: rxb0_1: w ymmreg2 101:C6:11 ymmreg1 ymmreg3: imm
ymmreg2 with mem to ymmreg1 using imm8	C4: rxb0_1: w ymmreg2 101:C6:mod ymmreg1 r/m: imm

Instruction and Format	Encoding
ymmreglo2 with ymmreglo3 to ymmreg1 using imm8	C5: r_ymmreglo2 101:C6:11 ymmreg1 ymmreglo3: imm
ymmreglo2 with mem to ymmreg1 using imm8	C5: r_ymmreglo2 101:C6:mod ymmreg1 r/m: imm
VSQRTPD — Compute Square Roots of Packed Double Precision Floating-Point Values	
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
ymmreg2 to ymmreg1	C4: rxb0_1: w_F 101:51:11 ymmreg1 ymmreg2
mem to ymmreg1	C4: rxb0_1: w_F 101:51:mod ymmreg1 r/m
ymmreglo to ymmreg1	C5: r_F 101:51:11 ymmreg1 ymmreglo
mem to ymmreg1	C5: r_F 101:51:mod ymmreg1 r/m
VSQRTSD — Compute Square Root of Scalar Double Precision Floating-Point Value	
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
VSUBPD — Subtract Packed Double Precision Floating-Point Values	
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
ymmreg2 with ymmreg3 to ymmreg1	C4: rxb0_1: w ymmreg2 101:5C:11 ymmreg1 ymmreg3
ymmreg2 with mem to ymmreg1	C4: rxb0_1: w ymmreg2 101:5C:mod ymmreg1 r/m
ymmreglo2 with ymmreglo3 to ymmreg1	C5: r_ymmreglo2 101:5C:11 ymmreg1 ymmreglo3
ymmreglo2 with mem to ymmreg1	C5: r_ymmreglo2 101:5C:mod ymmreg1 r/m
VSUBSD — Subtract Scalar Double Precision Floating-Point Values	
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
VUCOMISD — Unordered Compare Scalar Double Precision Floating-Point Values and Set EFLAGS	
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
VUNPCKHPD — Unpack and Interleave High Packed Double Precision Floating-Point Values	
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

Instruction and Format	Encoding
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 001:15:mod xmmreg1 r/m
ymmreg2 with ymmreg3 to ymmreg1	C4: rxb0_1: w ymmreg2 101:15:11 ymmreg1 ymmreg3
ymmreg2 with mem to ymmreg1	C4: rxb0_1: w ymmreg2 101:15:mod ymmreg1 r/m
ymmreglo2 with ymmreglo3 to ymmreg1	C5: r_ymmreglo2 101:15:11 ymmreg1 ymmreglo3
ymmreglo2 with mem to ymmreg1	C5: r_ymmreglo2 101:15:mod ymmreg1 r/m
VUNPCKHPS — Unpack and Interleave High Packed Single Precision Floating-Point Values	
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
ymmreg2 with ymmreg3 to ymmreg1	C4: rxb0_1: w ymmreg2 100:15:11 ymmreg1 ymmreg3
ymmreg2 with mem to ymmreg1	C4: rxb0_1: w ymmreg2 100:15:mod ymmreg1 r/m
ymmreglo2 with ymmreglo3 to ymmreg1	C5: r_ymmreglo2 100:15:11 ymmreg1 ymmreglo3
ymmreglo2 with mem to ymmreg1	C5: r_ymmreglo2 100:15:mod ymmreg1 r/m
VUNPCKLPD — Unpack and Interleave Low Packed Double Precision Floating-Point Values	
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
ymmreg2 with ymmreg3 to ymmreg1	C4: rxb0_1: w ymmreg2 101:14:11 ymmreg1 ymmreg3
ymmreg2 with mem to ymmreg1	C4: rxb0_1: w ymmreg2 101:14:mod ymmreg1 r/m
ymmreglo2 with ymmreglo3 to ymmreg1	C5: r_ymmreglo2 101:14:11 ymmreg1 ymmreglo3
ymmreglo2 with mem to ymmreg1	C5: r_ymmreglo2 101:14:mod ymmreg1 r/m
VUNPCKLPS — Unpack and Interleave Low Packed Single Precision Floating-Point Values	
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
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
VXORPD — Bitwise Logical XOR for Double Precision Floating-Point Values	
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

Instruction and Format	Encoding
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
VADDPS — Add Packed Single Precision Floating-Point Values	
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
VADDSS — Add Scalar Single Precision Floating-Point Values	
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
VANDPS — Bitwise Logical AND of Packed Single Precision Floating-Point Values	
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
VANDNPS — Bitwise Logical AND NOT of Packed Single Precision Floating-Point Values	
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
VCMPPS — Compare Packed Single Precision Floating-Point Values	
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

Instruction and Format	Encoding
ymmreg2 with ymmreg3 to ymmreg1	C4: rxb0_1: w ymmreg2 100:C2:11 ymmreg1 ymmreg3: imm
ymmreg2 with mem to ymmreg1	C4: rxb0_1: w ymmreg2 100:C2:mod ymmreg1 r/m: imm
ymmreglo2 with ymmreglo3 to ymmreg1	C5: r_ymmreglo2 100:C2:11 ymmreg1 ymmreglo3: imm
ymmreglo2 with mem to ymmreg1	C5: r_ymmreglo2 100:C2:mod ymmreg1 r/m: imm
VCMPSS — Compare Scalar Single Precision Floating-Point Values	
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
VCOMISS — Compare Scalar Ordered Single Precision Floating-Point Values and Set EFLAGS	
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
VCVTSI2SS — Convert Dword Integer to Scalar Single Precision FP Value	
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
xmmreg2 with mem to xmmreg1	C4: rxb0_1: 1 xmmreg2 010:2A:mod xmmreg1 r/m
VCVTSS2SI — Convert Scalar Single Precision FP Value to Dword Integer	
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
VCVTTSS2SI — Convert with Truncation Scalar Single Precision FP Value to Dword Integer	
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
VDIVPS — Divide Packed Single Precision Floating-Point Values	
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

Instruction and Format	Encoding
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 000:5E:mod xmmreg1 r/m
ymmreg2 with ymmreg3 to ymmreg1	C4: rxb0_1: w ymmreg2 100:5E:11 ymmreg1 ymmreg3
ymmreg2 with mem to ymmreg1	C4: rxb0_1: w ymmreg2 100:5E:mod ymmreg1 r/m
ymmreglo2 with ymmreglo3 to ymmreg1	C5: r_ymmreglo2 100:5E:11 ymmreg1 ymmreglo3
ymmreglo2 with mem to ymmreg1	C5: r_ymmreglo2 100:5E:mod ymmreg1 r/m
VDIVSS — Divide Scalar Single Precision Floating-Point Values	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 010:5E:11 xmmreg1 xmmreg3
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
VLDMXCSR — Load MXCSR Register	
mem to MXCSR reg	C4: rxb0_1: w_F 000:AEmod 011 r/m
mem to MXCSR reg	C5: r_F 000:AEmod 011 r/m
VMAXPS — Return Maximum Packed Single Precision Floating-Point Values	
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
ymmreg2 with ymmreg3 to ymmreg1	C4: rxb0_1: w ymmreg2 100:5F:11 ymmreg1 ymmreg3
ymmreg2 with mem to ymmreg1	C4: rxb0_1: w ymmreg2 100:5F:mod ymmreg1 r/m
ymmreglo2 with ymmreglo3 to ymmreg1	C5: r_ymmreglo2 100:5F:11 ymmreg1 ymmreglo3
ymmreglo2 with mem to ymmreg1	C5: r_ymmreglo2 100:5F:mod ymmreg1 r/m
VMAXSS — Return Maximum Scalar Single Precision Floating-Point Value	
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
VMINPS — Return Minimum Packed Single Precision Floating-Point Values	
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
xmmreglo2 with mem to xmmreg1	C5: r_xmmreglo2 000:5D:mod xmmreg1 r/m
ymmreg2 with ymmreg3 to ymmreg1	C4: rxb0_1: w ymmreg2 100:5D:11 ymmreg1 ymmreg3
ymmreg2 with mem to ymmreg1	C4: rxb0_1: w ymmreg2 100:5D:mod ymmreg1 r/m
ymmreglo2 with ymmreglo3 to ymmreg1	C5: r_ymmreglo2 100:5D:11 ymmreg1 ymmreglo3
ymmreglo2 with mem to ymmreg1	C5: r_ymmreglo2 100:5D:mod ymmreg1 r/m
VMINSS — Return Minimum Scalar Single Precision Floating-Point Value	
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

Instruction and Format	Encoding
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
VMOVAPS— Move Aligned Packed Single Precision Floating-Point Values	
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
ymmreg2 to ymmreg1	C4: rxb0_1: w_F 100:28:11 ymmreg1 ymmreg2
mem to ymmreg1	C4: rxb0_1: w_F 100:28:mod ymmreg1 r/m
ymmreglo to ymmreg1	C5: r_F 100:28:11 ymmreg1 ymmreglo
mem to ymmreg1	C5: r_F 100:28:mod ymmreg1 r/m
ymmreg1 to ymmreg2	C4: rxb0_1: w_F 100:29:11 ymmreg2 ymmreg1
ymmreg1 to mem	C4: rxb0_1: w_F 100:29:mod r/m ymmreg1
ymmreg1 to ymmreglo	C5: r_F 100:29:11 ymmreglo ymmreg1
ymmreg1 to mem	C5: r_F 100:29:mod r/m ymmreg1
VMOVHPS — Move High Packed Single Precision Floating-Point Values	
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
VMOVLHPS — Move Packed Single Precision Floating-Point Values Low to High	
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
VMOVLPS — Move Low Packed Single Precision Floating-Point Values	
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
VMOVMSKPS — Extract Packed Single Precision Floating-Point Sign Mask	
xmmreg2 to reg	C4: rxb0_1: w_F 000:50:11 reg xmmreg2
xmmreglo to reg	C5: r_F 000:50:11 reg xmmreglo
ymmreg2 to reg	C4: rxb0_1: w_F 100:50:11 reg ymmreg2
ymmreglo to reg	C5: r_F 100:50:11 reg ymmreglo
VMOVNTPS — Store Packed Single Precision Floating-Point Values Using Non-Temporal Hint	
xmmreg1 to mem	C4: rxb0_1: w_F 000:2B:mod r/m xmmreg1

Instruction and Format	Encoding
xmmreglo to mem	C5: r_F 000:2B:mod r/m xmmreglo
ymmreg1 to mem	C4: rxb0_1: w_F 100:2B:mod r/m ymmreg1
ymmreglo to mem	C5: r_F 100:2B:mod r/m ymmreglo
VMOVSS — Move Scalar Single Precision Floating-Point Values	
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
VMOVUPS— Move Unaligned Packed Single Precision Floating-Point Values	
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
ymmreg2 to ymmreg1	C4: rxb0_1: w_F 100:10:11 ymmreg1 ymmreg2
mem to ymmreg1	C4: rxb0_1: w_F 100:10:mod ymmreg1 r/m
ymmreglo to ymmreg1	C5: r_F 100:10:11 ymmreg1 ymmreglo
mem to ymmreg1	C5: r_F 100:10:mod ymmreg1 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
ymmreg1 to ymmreg2	C4: rxb0_1: w_F 100:11:11 ymmreg2 ymmreg1
ymmreg1 to mem	C4: rxb0_1: w_F 100:11:mod r/m ymmreg1
ymmreg1 to ymmreglo	C5: r_F 100:11:11 ymmreglo ymmreg1
ymmreg1 to mem	C5: r_F 100:11:mod r/m ymmreg1
VMULPS — Multiply Packed Single Precision Floating-Point Values	
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
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
VMULSS — Multiply Scalar Single Precision Floating-Point Values	

Instruction and Format	Encoding
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
VORPS — Bitwise Logical OR of Single Precision Floating-Point Values	
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
VRCPPS — Compute Reciprocals of Packed Single Precision Floating-Point Values	
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
mem to xmmreg1	C5: r_F 000:53:mod xmmreg1 r/m
ymmreg2 to ymmreg1	C4: rxb0_1: w_F 100:53:11 ymmreg1 ymmreg2
mem to ymmreg1	C4: rxb0_1: w_F 100:53:mod ymmreg1 r/m
ymmreglo to ymmreg1	C5: r_F 100:53:11 ymmreg1 ymmreglo
mem to ymmreg1	C5: r_F 100:53:mod ymmreg1 r/m
VRCPSS — Compute Reciprocal of Scalar Single Precision Floating-Point Values	
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
VRSQRTPS — Compute Reciprocals of Square Roots of Packed Single Precision Floating-Point Values	
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
ymmreg2 to ymmreg1	C4: rxb0_1: w_F 100:52:11 ymmreg1 ymmreg2
mem to ymmreg1	C4: rxb0_1: w_F 100:52:mod ymmreg1 r/m
ymmreglo to ymmreg1	C5: r_F 100:52:11 ymmreg1 ymmreglo
mem to ymmreg1	C5: r_F 100:52:mod ymmreg1 r/m
VRSQRTSS — Compute Reciprocal of Square Root of Scalar Single Precision Floating-Point Value	
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

Instruction and Format	Encoding
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
VSHUFPS — Shuffle Packed Single Precision Floating-Point Values	
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
VSQRTPS — Compute Square Roots of Packed Single Precision Floating-Point Values	
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
VSQRTSS — Compute Square Root of Scalar Single Precision Floating-Point Value	
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
VSTMXCSR — Store MXCSR Register State	
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
VSUBPS — Subtract Packed Single Precision Floating-Point Values	
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
ymmreg2 with ymmreg3 to ymmreg1	C4: rxb0_1: w ymmreg2 100:5C:11 ymmreg1 ymmreg3
ymmreg2 with mem to ymmreg1	C4: rxb0_1: w ymmreg2 100:5C:mod ymmreg1 r/m
ymmreglo2 with ymmreglo3 to ymmreg1	C5: r_ymmreglo2 100:5C:11 ymmreg1 ymmreglo3
ymmreglo2 with mem to ymmreg1	C5: r_ymmreglo2 100:5C:mod ymmreg1 r/m
VSUBSS — Subtract Scalar Single Precision Floating-Point Values	
xmmreg2 with xmmreg3 to xmmreg1	C4: rxb0_1: w xmmreg2 010:5C:11 xmmreg1 xmmreg3

Instruction and Format	Encoding
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
VUCOMISS — Unordered Compare Scalar Single Precision Floating-Point Values and Set EFLAGS	
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
UNPCKHPS — Unpack and Interleave High Packed Single Precision Floating-Point Values	
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 000:15mod xmmreg1 r/m
ymmreg2 with ymmreg3 to ymmreg1	C4: rxb0_1: w ymmreg2 100:15:11 ymmreg1 ymmreg3
ymmreg2 with mem to ymmreg1	C4: rxb0_1: w ymmreg2 100:15mod ymmreg1 r/m
UNPCKLPS — Unpack and Interleave Low Packed Single Precision Floating-Point Value	
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
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 100:14mod ymmreg1 r/m
VXORPS — Bitwise Logical XOR for Single Precision Floating-Point Values	
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
ymmreg2 with ymmreg3 to ymmreg1	C4: rxb0_1: w ymmreg2 100:57:11 ymmreg1 ymmreg3
ymmreg2 with mem to ymmreg1	C4: rxb0_1: w ymmreg2 100:57:mod ymmreg1 r/m
ymmreglo2 with ymmreglo3 to ymmreg1	C5: r_ymmreglo2 100:57:11 ymmreg1 ymmreglo3
ymmreglo2 with mem to ymmreg1	C5: r_ymmreglo2 100:57:mod ymmreg1 r/m
VBROADCAST —Load with Broadcast	
mem to xmmreg1	C4: rxb0_2: 0_F 001:18:mod xmmreg1 r/m
mem to ymmreg1	C4: rxb0_2: 0_F 101:18:mod ymmreg1 r/m
mem to ymmreg1	C4: rxb0_2: 0_F 101:19:mod ymmreg1 r/m
mem to ymmreg1	C4: rxb0_2: 0_F 101:1A:mod ymmreg1 r/m
VEXTRACTF128 — Extract Packed Floating-Point Values	
ymmreg2 to xmmreg1, imm8	C4: rxb0_3: 0_F 001:19:11 xmmreg1 ymmreg2: imm
ymmreg2 to mem, imm8	C4: rxb0_3: 0_F 001:19:mod r/m ymmreg2: imm
VINSERTF128 — Insert Packed Floating-Point Values	
xmmreg3 and merge with ymmreg2 to ymmreg1, imm8	C4: rxb0_3: 0 ymmreg2101:18:11 ymmreg1 xmmreg3: imm
mem and merge with ymmreg2 to ymmreg1, imm8	C4: rxb0_3: 0 ymmreg2 101:18:mod ymmreg1 r/m: imm
VPERMILPD — Permute Double Precision Floating-Point Values	

Instruction and Format	Encoding
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
VPERMILPS — Permute Single Precision Floating-Point Values	
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
VPERM2F128 — Permute Floating-Point Values	
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
VTESTPD/VTESTPS — Packed Bit Test	
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
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-38 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-39 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 $2^{ST(0)}$ – 1	11011 001 : 1111 0000
FABS – Absolute Value	11011 001 : 1110 0001
FADD – Add	
ST(0) := ST(0) + 32-bit memory	11011 000 : mod 000 r/m
ST(0) := ST(0) + 64-bit memory	11011 100 : mod 000 r/m
ST(d) := ST(0) + ST(i)	11011 d00 : 11 000 ST(i)
FADDP – Add and Pop	
ST(0) := ST(0) + ST(i)	11011 110 : 11 000 ST(i)
FBLD – Load Binary Coded Decimal	11011 111 : mod 100 r/m
FBSTP – Store Binary Coded Decimal and Pop	11011 111 : mod 110 r/m
FCHS – Change Sign	11011 001 : 1110 0000
FCLEX – Clear Exceptions	11011 011 : 1110 0010
FCOM – Compare Real	

Table B-39. Floating-Point Instruction Formats and Encodings (Contd.)

Instruction and Format	Encoding
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)
FCOMP – Compare Real and Pop	
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)
FCOMPP – Compare Real and Pop Twice	11011 110 : 11 011 001
FCOMIP – Compare Real, Set EFLAGS, and Pop	11011 111 : 11 110 ST(i)
FCOS – Cosine of ST(0)	11011 001 : 1111 1111
FDECSTP – Decrement Stack-Top Pointer	11011 001 : 1111 0110
FDIV – Divide	
ST(0) := ST(0) ÷ 32-bit memory	11011 000 : mod 110 r/m
ST(0) := ST(0) ÷ 64-bit memory	11011 100 : mod 110 r/m
ST(d) := ST(0) ÷ ST(i)	11011 d00 : 1111 R ST(i)
FDIVP – Divide and Pop	
ST(0) := ST(0) ÷ ST(i)	11011 110 : 1111 1 ST(i)
FDIVR – Reverse Divide	
ST(0) := 32-bit memory ÷ ST(0)	11011 000 : mod 111 r/m
ST(0) := 64-bit memory ÷ ST(0)	11011 100 : mod 111 r/m
ST(d) := ST(i) ÷ ST(0)	11011 d00 : 1111 R ST(i)
FDIVRP – Reverse Divide and Pop	
ST(0) := ST(i) ÷ ST(0)	11011 110 : 1111 0 ST(i)
FFREE – Free ST(i) Register	11011 101 : 1100 0 ST(i)
FIADD – Add Integer	
ST(0) := ST(0) + 16-bit memory	11011 110 : mod 000 r/m
ST(0) := ST(0) + 32-bit memory	11011 010 : mod 000 r/m
FICOM – Compare Integer	
16-bit memory	11011 110 : mod 010 r/m
32-bit memory	11011 010 : mod 010 r/m
FICOMP – Compare Integer and Pop	
16-bit memory	11011 110 : mod 011 r/m
32-bit memory	11011 010 : mod 011 r/m
FIDIV – Divide	
ST(0) := ST(0) ÷ 16-bit memory	11011 110 : mod 110 r/m
ST(0) := ST(0) ÷ 32-bit memory	11011 010 : mod 110 r/m
FIDIVR – Reverse Divide	
ST(0) := 16-bit memory ÷ ST(0)	11011 110 : mod 111 r/m

Table B-39. Floating-Point Instruction Formats and Encodings (Contd.)

Instruction and Format	Encoding
ST(0) := 32-bit memory ÷ ST(0)	11011 010 : mod 111 r/m
FILD – Load Integer	
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
FIMUL– Multiply	
ST(0) := ST(0) × 16-bit memory	11011 110 : mod 001 r/m
ST(0) := ST(0) × 32-bit memory	11011 010 : mod 001 r/m
FINCSTP – Increment Stack Pointer	11011 001 : 1111 0111
FINIT – Initialize Floating-Point Unit	
FIST – Store Integer	
16-bit memory	11011 111 : mod 010 r/m
32-bit memory	11011 011 : mod 010 r/m
FISTP – Store Integer and Pop	
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
FISUB – Subtract	
ST(0) := ST(0) - 16-bit memory	11011 110 : mod 100 r/m
ST(0) := ST(0) - 32-bit memory	11011 010 : mod 100 r/m
FISUBR – Reverse Subtract	
ST(0) := 16-bit memory – ST(0)	11011 110 : mod 101 r/m
ST(0) := 32-bit memory – ST(0)	11011 010 : mod 101 r/m
FLD – Load Real	
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)
FLD1 – Load +1.0 into ST(0)	11011 001 : 1110 1000
FLDCW – Load Control Word	11011 001 : mod 101 r/m
FLDENV – Load FPU Environment	11011 001 : mod 100 r/m
FLDL2E – Load $\log_2(\varepsilon)$ into ST(0)	11011 001 : 1110 1010
FLDL2T – Load $\log_2(10)$ into ST(0)	11011 001 : 1110 1001
FLDLG2 – Load $\log_{10}(2)$ into ST(0)	11011 001 : 1110 1100
FLDLN2 – Load $\log_\varepsilon(2)$ into ST(0)	11011 001 : 1110 1101
FLDPI – Load π into ST(0)	11011 001 : 1110 1011
FLDZ – Load +0.0 into ST(0)	11011 001 : 1110 1110
FMUL – Multiply	

Table B-39. Floating-Point Instruction Formats and Encodings (Contd.)

Instruction and Format	Encoding
ST(0) := ST(0) × 32-bit memory	11011 000 : mod 001 r/m
ST(0) := ST(0) × 64-bit memory	11011 100 : mod 001 r/m
ST(d) := ST(0) × ST(i)	11011 d00 : 1100 1 ST(i)
FMULP – Multiply	
ST(i) := ST(0) × ST(i)	11011 110 : 1100 1 ST(i)
FNOP – No Operation	11011 001 : 1101 0000
FPATAN – Partial Arctangent	11011 001 : 1111 0011
FPREM – Partial Remainder	11011 001 : 1111 1000
FPREM1 – Partial Remainder (IEEE)	11011 001 : 1111 0101
FPTAN – Partial Tangent	11011 001 : 1111 0010
FRNDINT – Round to Integer	11011 001 : 1111 1100
FRSTOR – Restore FPU State	11011 101 : mod 100 r/m
FSAVE – Store FPU State	11011 101 : mod 110 r/m
FSCALE – Scale	11011 001 : 1111 1101
FSIN – Sine	11011 001 : 1111 1110
FSINCOS – Sine and Cosine	11011 001 : 1111 1011
FSQRT – Square Root	11011 001 : 1111 1010
FST – Store Real	
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)
FSTCW – Store Control Word	11011 001 : mod 111 r/m
FSTENV – Store FPU Environment	11011 001 : mod 110 r/m
FSTP – Store Real and Pop	
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)
FSTSW – Store Status Word into AX	11011 111 : 1110 0000
FSTSW – Store Status Word into Memory	11011 101 : mod 111 r/m
FSUB – Subtract	
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)
FSUBP – Subtract and Pop	
ST(0) := ST(0) – ST(i)	11011 110 : 1110 1 ST(i)
FSUBR – Reverse Subtract	
ST(0) := 32-bit memory – ST(0)	11011 000 : mod 101 r/m

Table B-39. Floating-Point Instruction Formats and Encodings (Contd.)

Instruction and Format	Encoding
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)
FSUBRP – Reverse Subtract and Pop	
ST(i) := ST(i) – ST(0)	11011 110 : 1110 0 ST(i)
FTST – Test	11011 001 : 1110 0100
FUCOM – Unordered Compare Real	11011 101 : 1110 0 ST(i)
FUCOMP – Unordered Compare Real and Pop	11011 101 : 1110 1 ST(i)
FUCOMPP – Unordered Compare Real and Pop Twice	11011 010 : 1110 1001
FUCOMI – Unorderd Compare Real and Set EFLAGS	11011 011 : 11 101 ST(i)
FUCOMIP – Unorderd Compare Real, Set EFLAGS, and Pop	11011 111 : 11 101 ST(i)
FXAM – Examine	11011 001 : 1110 0101
FXCH – Exchange ST(0) and ST(i)	11011 001 : 1100 1 ST(i)
FXTRACT – Extract Exponent and Significand	11011 001 : 1111 0100
FYL2X – ST(1) $\times$ log$_2$(ST(0))	11011 001 : 1111 0001
FYL2XP1 – ST(1) $\times$ log$_2$(ST(0) + 1.0)	11011 001 : 1111 1001
FWAIT – Wait until FPU Ready	1001 1011 (same instruction as WAIT)

B.18 VMX INSTRUCTIONS

Table B-40 describes virtual-machine extensions (VMX).

Table B-40. Encodings for VMX Instructions

Instruction and Format	Encoding
INVEPT—Invalidate Cached EPT Mappings	
Descriptor m128 according to reg	01100110 00001111 00111000 10000000: mod reg r/m
INVVPID—Invalidate Cached VPID Mappings	
Descriptor m128 according to reg	01100110 00001111 00111000 10000001: mod reg r/m
VMCALL—Call to VM Monitor	
Call VMM: causes VM exit.	00001111 00000001 11000001
VMCLEAR—Clear Virtual-Machine Control Structure	
mem32:VMCS_data_ptr	01100110 00001111 11000111: mod 110 r/m
mem64:VMCS_data_ptr	01100110 00001111 11000111: mod 110 r/m
VMFUNC—Invoke VM Function	
Invoke VM function specified in EAX	00001111 00000001 11010100
VMLAUNCH—Launch Virtual Machine	
Launch VM managed by Current_VMCS	00001111 00000001 11000010
VMRESUME—Resume Virtual Machine	
Resume VM managed by Current_VMCS	00001111 00000001 11000011
VMPTRLD—Load Pointer to Virtual-Machine Control Structure	
mem32 to Current_VMCS_ptr	00001111 11000111: mod 110 r/m

Table B-40. Encodings for VMX Instructions

Instruction and Format	Encoding
mem64 to Current_VMCS_ptr	00001111 11000111: mod 110 r/m
VMPTRST—Store Pointer to Virtual-Machine Control Structure	
Current_VMCS_ptr to mem32	00001111 11000111: mod 111 r/m
Current_VMCS_ptr to mem64	00001111 11000111: mod 111 r/m
VMREAD—Read Field from Virtual-Machine Control Structure	
r32 *(VMCS_fieldn)* to r32 r32 *(VMCS_fieldn)* to mem32 r64 *(VMCS_fieldn)* to r64 r64 *(VMCS_fieldn)* to mem64	00001111 01111000: 11 reg2 reg1 00001111 01111000: mod r32 r/m 00001111 01111000: 11 reg2 reg1 00001111 01111000: mod r64 r/m
VMWRITE—Write Field to Virtual-Machine Control Structure	
r32 to r32 *(VMCS_fieldn)* mem32 to r32 *(VMCS_fieldn)* r64 to r64 *(VMCS_fieldn)* mem64 to r64 *(VMCS_fieldn)*	00001111 01111001: 11 reg1 reg2 00001111 01111001: mod r32 r/m 00001111 01111001: 11 reg1 reg2 00001111 01111001: mod r64 r/m
VMXOFF—Leave VMX Operation	
Leave VMX.	00001111 00000001 11000100
VMXON—Enter VMX Operation	
Enter VMX.	11110011 000011111 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
GETSEC—GETSEC leaf functions are selected by the value in EAX on input	
GETSEC[CAPABILITIES]	00001111 00110111 (EAX= 0)
GETSEC[ENTERACCS]	00001111 00110111 (EAX= 2)
GETSEC[EXITAC]	00001111 00110111 (EAX= 3)
GETSEC[SENTER]	00001111 00110111 (EAX= 4)
GETSEC[SEXIT]	00001111 00110111 (EAX= 5)
GETSEC[PARAMETERS]	00001111 00110111 (EAX= 6)
GETSEC[SMCTRL]	00001111 00110111 (EAX= 7)
GETSEC[WAKEUP]	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/performancetools/.

Table C-1 presents simple intrinsics and Table C-2 presents composite intrinsics. Some intrinsics are “composites” because they require more than one instruction to implement them.

Intel C/C++ Compiler intrinsic names reflect the following naming conventions:

```
_mm_<intrin_op>_<suffix>
```

where:

<intrin_op>	Indicates the intrinsics basic operation; for example, add for addition and sub for subtraction
<suffix>	Denotes the type of data operated on by the instruction. The first one or two letters of each suffix denotes whether the data is packed (p), extended packed (ep), or scalar (s).

The remaining letters denote the type:

s	single-precision floating-point
d	double precision floating-point
i128	signed 128-bit integer
i64	signed 64-bit integer
u64	unsigned 64-bit integer
i32	signed 32-bit integer
u32	unsigned 32-bit integer
i16	signed 16-bit integer
u16	unsigned 16-bit integer
i8	signed 8-bit integer
u8	unsigned 8-bit integer

The variable r is generally used for the intrinsic's return value. A number appended to a variable name indicates the element of a packed object. For example, r0 is the lowest word of r.

The packed values are represented in right-to-left order, with the lowest value being used for scalar operations. Consider the following example operation:

```
double a[2] = {1.0, 2.0};
__m128d t = _mm_load_pd(a);
```

The result is the same as either of the following:

```
__m128d t = _mm_set_pd(2.0, 1.0);
__m128d t = _mm_setr_pd(1.0, 2.0);
```

In other words, the XMM register that holds the value t will look as follows:

2.0	1.0
127 — 64	63 — 0

The “scalar” element is 1.0. Due to the nature of the instruction, some intrinsics require their arguments to be immediates (constant integer literals).

To use an intrinsic in your code, insert a line with the following syntax:

data_type intrinsic_name (parameters)

Where:

data_type	Is the return data type, which can be either void, int, __m64, __m128, __m128d, or __m128i. Only the _mm_empty intrinsic returns void.
intrinsic_name	Is the name of the intrinsic, which behaves like a function that you can use in your C/C++ code instead of in-lining the actual instruction.
parameters	Represents the parameters required by each intrinsic.

C.1 SIMPLE INTRINSICS

NOTE

For detailed descriptions of the intrinsics in Table C-1, see the corresponding mnemonic in Chapter 3, “Instruction Set Reference, A-L” of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A; Chapter 4, “Instruction Set Reference, M-U” of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2B; Chapter 5, “Instruction Set Reference, V,” of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2C; or Chapter 6, “Instruction Set Reference, W-Z,” of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2D.

Table C-1. Simple Intrinsics

Mnemonic	Intrinsic
ADDPD	__m128d _mm_add_pd(__m128d a, __m128d b)
ADDPS	__m128 _mm_add_ps(__m128 a, __m128 b)
ADDSD	__m128d _mm_add_sd(__m128d a, __m128d b)
ADDSS	__m128 _mm_add_ss(__m128 a, __m128 b)
ADDSUBPD	__m128d _mm_addsub_pd(__m128d a, __m128d b)
ADDSUBPS	__m128 _mm_addsub_ps(__m128 a, __m128 b)
AESDEC	__m128i _mm_aesdec (__m128i, __m128i)
AESDECLAST	__m128i _mm_aesdeclast (__m128i, __m128i)
AESENC	__m128i _mm_aesenc (__m128i, __m128i)
AESENCLAST	__m128i _mm_aesenclast (__m128i, __m128i)
AESIMC	__m128i _mm_aesimc (__m128i)
AESKEYGENASSIST	__m128i _mm_aesimc (__m128i, const int)
ANDNPD	__m128d _mm_andnot_pd(__m128d a, __m128d b)
ANDNPS	__m128 _mm_andnot_ps(__m128 a, __m128 b)
ANDPD	__m128d _mm_and_pd(__m128d a, __m128d b)
ANDPS	__m128 _mm_and_ps(__m128 a, __m128 b)
BLENDPD	__m128d _mm_blend_pd(__m128d v1, __m128d v2, const int mask)
BLENDPS	__m128 _mm_blend_ps(__m128 v1, __m128 v2, const int mask)
BLENDVPD	__m128d _mm_blendv_pd(__m128d v1, __m128d v2, __m128d v3)
BLENDVPS	__m128 _mm_blendv_ps(__m128 v1, __m128 v2, __m128 v3)
CLFLUSH	void _mm_clflush(void const *p)

Table C-1. Simple Intrinsics (Contd.)

Mnemonic	Intrinsic
CMPPD	__m128d _mm_cmpeq_pd(__m128d a, __m128d b)
	__m128d _mm_cmplt_pd(__m128d a, __m128d b)
	__m128d _mm_cmple_pd(__m128d a, __m128d b)
	__m128d _mm_cmpgt_pd(__m128d a, __m128d b)
	__m128d _mm_cmpge_pd(__m128d a, __m128d b)
	__m128d _mm_cmpneq_pd(__m128d a, __m128d b)
	__m128d _mm_cmpnlt_pd(__m128d a, __m128d b)
	__m128d _mm_cmpngt_pd(__m128d a, __m128d b)
	__m128d _mm_cmpnge_pd(__m128d a, __m128d b)
	__m128d _mm_cmpord_pd(__m128d a, __m128d b)
	__m128d _mm_cmpunord_pd(__m128d a, __m128d b)
	__m128d _mm_cmpnle_pd(__m128d a, __m128d b)
CMPPS	__m128 _mm_cmpeq_ps(__m128 a, __m128 b)
	__m128 _mm_cmplt_ps(__m128 a, __m128 b)
	__m128 _mm_cmple_ps(__m128 a, __m128 b)
	__m128 _mm_cmpgt_ps(__m128 a, __m128 b)
	__m128 _mm_cmpge_ps(__m128 a, __m128 b)
	__m128 _mm_cmpneq_ps(__m128 a, __m128 b)
	__m128 _mm_cmpnlt_ps(__m128 a, __m128 b)
	__m128 _mm_cmpngt_ps(__m128 a, __m128 b)
	__m128 _mm_cmpnge_ps(__m128 a, __m128 b)
	__m128 _mm_cmpord_ps(__m128 a, __m128 b)
	__m128 _mm_cmpunord_ps(__m128 a, __m128 b)
	__m128 _mm_cmpnle_ps(__m128 a, __m128 b)
CMPSD	__m128d _mm_cmpeq_sd(__m128d a, __m128d b)
	__m128d _mm_cmplt_sd(__m128d a, __m128d b)
	__m128d _mm_cmple_sd(__m128d a, __m128d b)
	__m128d _mm_cmpgt_sd(__m128d a, __m128d b)
	__m128d _mm_cmpge_sd(__m128d a, __m128d b)
	__m128 _mm_cmpneq_sd(__m128d a, __m128d b)
	__m128 _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)

Table C-1. Simple Intrinsics (Contd.)

Mnemonic	Intrinsic
	__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)
	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)

Table C-1. Simple Intrinsics (Contd.)

Mnemonic	Intrinsic
CVTSS2SI	int _mm_cvt_ss2si(__m128 a) int _mm_cvtss_si32(__m128 a)
CVTTPD2DQ	__m128i _mm_cvttpd_epi32(__m128d a)
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_cvttsd_si32(__m128d a)
CVTTSS2SI	int _mm_cvtt_ss2si(__m128 a) int _mm_cvttss_si32(__m128 a) __m64 _mm_cvtsi32_si64(int i) int _mm_cvtsi64_si32(__m64 m)
DIVPD	__m128d _mm_div_pd(__m128d a, __m128d b)
DIVPS	__m128 _mm_div_ps(__m128 a, __m128 b)
DIVSD	__m128d _mm_div_sd(__m128d a, __m128d b)
DIVSS	__m128 _mm_div_ss(__m128 a, __m128 b)
DPPD	__m128d _mm_dp_pd(__m128d a, __m128d b, const int mask)
DPPS	__m128 _mm_dp_ps(__m128 a, __m128 b, const int mask)
EMMS	void _mm_empty()
EXTRACTPS	int _mm_extract_ps(__m128 src, const int ndx)
HADDPD	__m128d _mm_hadd_pd(__m128d a, __m128d b)
HADDPS	__m128 _mm_hadd_ps(__m128 a, __m128 b)
HSUBPD	__m128d _mm_hsub_pd(__m128d a, __m128d b)
HSUBPS	__m128 _mm_hsub_ps(__m128 a, __m128 b)
INSERTPS	__m128 _mm_insert_ps(__m128 dst, __m128 src, const int ndx)
LDDQU	__m128i _mm_lddqu_si128(__m128i const *p)
LDMXCSR	__mm_setcsr(unsigned int i)
LFENCE	void _mm_lfence(void)
MASKMOVDQU	void _mm_maskmoveu_si128(__m128i d, __m128i n, char *p)
MASKMOVQ	void _mm_maskmove_si64(__m64 d, __m64 n, char *p)
MAXPD	__m128d _mm_max_pd(__m128d a, __m128d b)
MAXPS	__m128 _mm_max_ps(__m128 a, __m128 b)
MAXSD	__m128d _mm_max_sd(__m128d a, __m128d b)
MAXSS	__m128 _mm_max_ss(__m128 a, __m128 b)
MFENCE	void _mm_mfence(void)
MINPD	__m128d _mm_min_pd(__m128d a, __m128d b)
MINPS	__m128 _mm_min_ps(__m128 a, __m128 b)
MINSD	__m128d _mm_min_sd(__m128d a, __m128d b)
MINSS	__m128 _mm_min_ss(__m128 a, __m128 b)
MONITOR	void _mm_monitor(void const *p, unsigned extensions, unsigned hints)
MOVAPD	__m128d _mm_load_pd(double * p)
	void_mm_store_pd(double *p, __m128d a)

Table C-1. Simple Intrinsics (Contd.)

Mnemonic	Intrinsic
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)
MOVDQA	__m128i _mm_load_si128(__m128i * p)
	void_mm_store_si128(__m128i *p, __m128i a)
MOVDQU	__m128i _mm_loadu_si128(__m128i * p)
	void_mm_storeu_si128(__m128i *p, __m128i a)
MOVDQ2Q	__m64 _mm_movepi64_pi64(__m128i a)
MOVHLPS	__m128 _mm_movehl_ps(__m128 a, __m128 b)
MOVHPD	__m128d _mm_loadh_pd(__m128d a, double * p)
	void _mm_storeh_pd(double * p, __m128d a)
MOVHPS	__m128 _mm_loadh_pi(__m128 a, __m64 * p)
	void _mm_storeh_pi(__m64 * p, __m128 a)
MOVLPD	__m128d _mm_loadl_pd(__m128d a, double * p)
	void _mm_storel_pd(double * p, __m128d a)
MOVLPS	__m128 _mm_loadl_pi(__m128 a, __m64 *p)
	void_mm_storel_pi(__m64 * p, __m128 a)
MOVLHPS	__m128 _mm_movelh_ps(__m128 a, __m128 b)
MOVMSKPD	int _mm_movemask_pd(__m128d a)
MOVMSKPS	int _mm_movemask_ps(__m128 a)
MOVNTDQA	__m128i _mm_stream_load_si128(__m128i *p)
MOVNTDQ	void_mm_stream_si128(__m128i * p, __m128i a)
MOVNTPD	void_mm_stream_pd(double * p, __m128d a)
MOVNTPS	void_mm_stream_ps(float * p, __m128 a)
MOVNTI	void_mm_stream_si32(int * p, int a)
MOVNTQ	void_mm_stream_pi(__m64 * p, __m64 a)
MOVQ	__m128i _mm_loadl_epi64(__m128i * p)
	void_mm_storel_epi64(_m128i * p, __m128i a) __m128i _mm_move_epi64(__m128i a)
MOVQ2DQ	__m128i _mm_movpi64_epi64(__m64 a)
MOVSD	__m128d _mm_load_sd(double * p)
	void_mm_store_sd(double * p, __m128d a)
	__m128d _mm_move_sd(__m128d a, __m128d b)
MOVSHDUP	__m128 _mm_movehdup_ps(__m128 a)
MOVSLDUP	__m128 _mm_moveldup_ps(__m128 a)
MOVSS	__m128 _mm_load_ss(float * p)

Table C-1. Simple Intrinsics (Contd.)

Mnemonic	Intrinsic
	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)
PACKUSWB	__m64 _mm_packs_pu16(__m64 m1, __m64 m2)
PADDB	__m128i _mm_add_epi8(__m128i m1, __m128i m2)
PADDB	__m64 _mm_add_pi8(__m64 m1, __m64 m2)
PADDW	__m128i _mm_add_epi16(__m128i m1, __m128i m2)
PADDW	__m64 _mm_add_pi16(__m64 m1, __m64 m2)
PADDD	__m128i _mm_add_epi32(__m128i m1, __m128i m2)
PADDD	__m64 _mm_add_pi32(__m64 m1, __m64 m2)
PADDQ	__m128i _mm_add_epi64(__m128i m1, __m128i m2)
PADDQ	__m64 _mm_add_si64(__m64 m1, __m64 m2)
PADDSB	__m128i _mm_adds_epi8(__m128i m1, __m128i m2)
PADDSB	__m64 _mm_adds_pi8(__m64 m1, __m64 m2)
PADDSW	__m128i _mm_adds_epi16(__m128i m1, __m128i m2)
PADDSW	__m64 _mm_adds_pi16(__m64 m1, __m64 m2)
PADDUSB	__m128i _mm_adds_epu8(__m128i m1, __m128i m2)
PADDUSB	__m64 _mm_adds_pu8(__m64 m1, __m64 m2)

Table C-1. Simple Intrinsics (Contd.)

Mnemonic	Intrinsic
PADDUSW	__m128i _mm_adds_epu16(__m128i m1, __m128i m2)
PADDUSW	__m64 _mm_adds_pu16(__m64 m1, __m64 m2)
PALIGNR	__m64 _mm_alignr_pi8 (__m64 a, __m64 b, int n)
	__m128i _mm_alignr_epi8 (__m128i a, __m128i b, int n)
PAND	__m128i _mm_and_si128(__m128i m1, __m128i m2)
PAND	__m64 _mm_and_si64(__m64 m1, __m64 m2)
PANDN	__m128i _mm_andnot_si128(__m128i m1, __m128i m2)
PANDN	__m64 _mm_andnot_si64(__m64 m1, __m64 m2)
PAUSE	void _mm_pause(void)
PAVGB	__m128i _mm_avg_epu8(__m128i a, __m128i b)
PAVGB	__m64 _mm_avg_pu8(__m64 a, __m64 b)
PAVGW	__m128i _mm_avg_epu16(__m128i a, __m128i b)
PAVGW	__m64 _mm_avg_pu16(__m64 a, __m64 b)
PBLENDVB	__m128i _mm_blendv_epi (__m128i v1, __m128i v2, __m128i mask)
PBLENDW	__m128i _mm_blend_epi16(__m128i v1, __m128i v2, const int mask)
PCLMULQDQ	__m128i _mm_clmulepi64_si128 (__m128i, __m128i, const int)
PCMPEQB	__m128i _mm_cmpeq_epi8(__m128i m1, __m128i m2)
PCMPEQB	__m64 _mm_cmpeq_pi8(__m64 m1, __m64 m2)
PCMPEQQ	__m128i _mm_cmpeq_epi64(__m128i a, __m128i b)
PCMPEQW	__m128i _mm_cmpeq_epi16 (__m128i m1, __m128i m2)
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)

Table C-1. Simple Intrinsics (Contd.)

Mnemonic	Intrinsic
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)
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)

Table C-1. Simple Intrinsics (Contd.)

Mnemonic	Intrinsic
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)
PMINSB	_m128i _mm_min_epi8(__m128i a, __m128i b)
PMINSD	__m128i _mm_min_epi32(__m128i a, __m128i b)
PMINSW	__m128i _mm_min_epi16(__m128i a, __m128i b)
PMINSW	__m64 _mm_min_pi16(__m64 a, __m64 b)
PMINUB	__m128i _mm_min_epu8(__m128i a, __m128i b)
PMINUB	__m64 _mm_min_pu8(__m64 a, __m64 b)
PMINUD	__m128i _mm_min_epu32 (__m128i a, __m128i b)
PMINUW	__m128i _mm_min_epu16 (__m128i a, __m128i b)
PMOVMSKB	int _mm_movemask_epi8(__m128i a)
PMOVMSKB	int _mm_movemask_pi8(__m64 a)
PMOVSXBW	__m128i _mm_ cvtepi8_epi16(__m128i a)
PMOVSXBD	__m128i _mm_ cvtepi8_epi32(__m128i a)
PMOVSXBQ	__m128i _mm_ cvtepi8_epi64(__m128i a)
PMOVSXWD	__m128i _mm_ cvtepi16_epi32(__m128i a)
PMOVSXWQ	__m128i _mm_ cvtepi16_epi64(__m128i a)
PMOVSXDQ	__m128i _mm_ cvtepi32_epi64(__m128i a)
PMOVZXBW	__m128i _mm_ cvtepu8_epi16(__m128i a)
PMOVZXBD	__m128i _mm_ cvtepu8_epi32(__m128i a)
PMOVZXBQ	__m128i _mm_ cvtepu8_epi64(__m128i a)
PMOVZXWD	__m128i _mm_ cvtepu16_epi32(__m128i a)
PMOVZXWQ	__m128i _mm_ cvtepu16_epi64(__m128i a)
PMOVZXDQ	__m128i _mm_ cvtepu32_epi64(__m128i a)
PMULDQ	__m128i _mm_mul_epi32(__m128i a, __m128i b)
PMULHRSW	__m64 _mm_mulhrs_pi16 (__m64 a, __m64 b)
	__m128i _mm_mulhrs_epi16 (__m128i a, __m128i b)
PMULHUW	__m128i _mm_mulhi_epu16(__m128i a, __m128i b)
PMULHUW	__m64 _mm_mulhi_pu16(__m64 a, __m64 b)
PMULHW	__m128i _mm_mulhi_epi16(__m128i m1, __m128i m2)
PMULHW	__m64 _mm_mulhi_pi16(__m64 m1, __m64 m2)
PMULLUD	__m128i _mm_mullo_epi32(__m128i a, __m128i b)
PMULLW	__m128i _mm_mullo_epi16(__m128i m1, __m128i m2)
PMULLW	__m64 _mm_mullo_pi16(__m64 m1, __m64 m2)

Table C-1. Simple Intrinsics (Contd.)

Mnemonic	Intrinsic
PMULUDQ	__m64 _mm_mul_su32(__m64 m1, __m64 m2)
	__m128i _mm_mul_epu32(__m128i m1, __m128i m2)
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)
PSHUFHW	__m128i _mm_shufflehi_epi16(__m128i a, int n)
PSHUFLW	__m128i _mm_shufflelo_epi16(__m128i a, int n)
PSHUFW	__m64 _mm_shuffle_pi16(__m64 a, int n)
PSIGNB	__m64 _mm_sign_pi8 (__m64 a, __m64 b)
	__m128i _mm_sign_epi8 (__m128i a, __m128i b)
PSIGND	__m64 _mm_sign_pi32 (__m64 a, __m64 b)
	__m128i _mm_sign_epi32 (__m128i a, __m128i b)
PSIGNW	__m64 _mm_sign_pi16 (__m64 a, __m64 b)
	__m128i _mm_sign_epi16 (__m128i a, __m128i b)
PSLLW	__m128i _mm_sll_epi16(__m128i m, __m128i count)
PSLLW	__m128i _mm_slli_epi16(__m128i m, int count)
PSLLW	__m64 _mm_sll_pi16(__m64 m, __m64 count)
	__m64 _mm_slli_pi16(__m64 m, int count)
PSLLD	__m128i _mm_slli_epi32(__m128i m, int count)
	__m128i _mm_sll_epi32(__m128i m, __m128i count)
PSLLD	__m64 _mm_slli_pi32(__m64 m, int count)
	__m64 _mm_sll_pi32(__m64 m, __m64 count)
PSLLQ	__m64 _mm_sll_si64(__m64 m, __m64 count)
	__m64 _mm_slli_si64(__m64 m, int count)
PSLLQ	__m128i _mm_sll_epi64(__m128i m, __m128i count)
	__m128i _mm_slli_epi64(__m128i m, int count)
PSLLDQ	__m128i _mm_slli_si128(__m128i m, int imm)
PSRAW	__m128i _mm_sra_epi16(__m128i m, __m128i count)
	__m128i _mm_srai_epi16(__m128i m, int count)
PSRAW	__m64 _mm_sra_pi16(__m64 m, __m64 count)
	__m64 _mm_srai_pi16(__m64 m, int count)
PSRAD	__m128i _mm_sra_epi32 (__m128i m, __m128i count)
	__m128i _mm_srai_epi32 (__m128i m, int count)
PSRAD	__m64 _mm_sra_pi32 (__m64 m, __m64 count)

Table C-1. Simple Intrinsics (Contd.)

Mnemonic	Intrinsic
	__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)
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)

Table C-1. Simple Intrinsics (Contd.)

Mnemonic	Intrinsic
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_rsqrt_ps(__m128 a)
RSQRTSS	__m128 _mm_rsqrt_ss(__m128 a)
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)

Table C-1. Simple Intrinsics (Contd.)

Mnemonic	Intrinsic
	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)

Table C-2. Composite Intrinsics (Contd.)

Mnemonic	Intrinsic
(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)
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

A

B

C

G

H

I

N

O

R

S

T

U

V

W

X

Z