IA-32 Intel® Architecture Software Developer's Manual

Volume 2B: Instruction Set Reference, N-Z

NOTE: The *IA-32 Intel Architecture Software Developer's Manual* consists of four volumes: *Basic Architecture*, Order Number 253665; *Instruction Set Reference A-M,* Order Number 253666; *Instruction Set Reference N-Z,* Order Number 253667; and the *System Programming Guide,* Order Number 253668. Refer to all four volumes when evaluating your design needs.

2004

4

Instruction Set Reference, N-Z

CHAPTER 4
INSTRUCTION SET REFERENCE, N-Z

Chapter 4 continues the alphabetical discussion of IA-32 instructions (N-Z) started in Chapter 3. To access information on the remainder of the IA-32 instructions (A-M), see *IA-32 Intel Architecture Software Developer's Manual, Volume 2A*.

NEG—Two's Complement Negation

Opcode	Instruction	Description
F6 /3	NEG *r/m8*	Two's complement negate *r/m8.*
F7 /3	NEG *r/m16*	Two's complement negate *r/m16.*
F7 /3	NEG *r/m32*	Two's complement negate *r/m32.*

Description

Replaces the value of operand (the destination operand) with its two's complement. (This operation is equivalent to subtracting the operand from 0.) The destination operand is located in a general-purpose register or a memory location.

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

Operation

```
IF DEST = 0
    THEN CF ← 0
    ELSE CF ← 1;
FI;
DEST ← – (DEST)
```

Flags Affected

The CF flag set to 0 if the source operand is 0; otherwise it is set to 1. The OF, SF, ZF, AF, and PF flags are set according to the result.

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.

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.

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.

NOP—No Operation

Opcode	Instruction	Description
90	NOP	No operation.

Description

Performs no operation. This instruction is a one-byte instruction that takes up space in the instruction stream but does not affect the machine context, except the EIP register.

The NOP instruction is an alias mnemonic for the XCHG (E)AX, (E)AX instruction.

Flags Affected

None.

Exceptions (All Operating Modes)

None.

NOT—One's Complement Negation

Opcode	Instruction	Description
F6 /2	NOT *r/m8*	Reverse each bit of *r/m8.*
F7 /2	NOT *r/m16*	Reverse each bit of *r/m16.*
F7 /2	NOT *r/m32*	Reverse each bit of *r/m32.*

Description

Performs a bitwise NOT operation (each 1 is set to 0, and each 0 is set to 1) on the destination operand and stores the result in the destination operand location. The destination operand can be a register or a memory location.

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

Operation

```
DEST ← NOT DEST;
```

Flags Affected

None.

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.

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.

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.

OR—Logical Inclusive OR

Opcode	Instruction	Description
0C *ib*	OR AL,*imm8*	AL OR *imm8.*
0D *iw*	OR AX,*imm16*	AX OR *imm16.*
0D *id*	OR EAX,*imm32*	EAX OR *imm32.*
80 /1 *ib*	OR *r/m8,imm8*	*r/m8* OR *imm8.*
81 /1 *iw*	OR *r/m16,imm16*	*r/m16* OR *imm16.*
81 /1 *id*	OR *r/m32,imm32*	*r/m32* OR *imm32*
83 /1 *ib*	OR *r/m16,imm8*	*r/m16* OR *imm8 (sign-extended).*
83 /1 *ib*	OR *r/m32,imm8*	*r/m32* OR *imm8 (sign-extended).*
08 */r*	OR *r/m8,r8*	*r/m8* OR *r8.*
09 */r*	OR *r/m16,r16*	*r/m16* OR *r16.*
09 */r*	OR *r/m32,r32*	*r/m32* OR *r32.*
0A */r*	OR *r8,r/m8*	*r8* OR *r/m8.*
0B */r*	OR *r16,r/m16*	*r16* OR *r/m16.*
0B */r*	OR *r32,r/m32*	*r32* OR *r/m32.*

Description

Performs a bitwise inclusive OR operation between 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 of the OR instruction is set to 0 if both corresponding bits of the first and second operands are 0; otherwise, each bit is set to 1.

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

Operation

```
DEST ← DEST OR 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.

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.

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.

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

Opcode	Instruction	Description
66 0F 56 /r	ORPD *xmm1*, *xmm2/m128*	Bitwise OR of *xmm2/m128* and *xmm1*.

Description

Performs a bitwise logical OR of the two packed double-precision floating-point values from the source operand (second operand) and the destination operand (first operand), and stores the result in the destination operand. The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register.

Operation

```
DEST[127-0] ← DEST[127-0] BitwiseOR SRC[127-0];
```

Intel C/C++ Compiler Intrinsic Equivalent

```
ORPD            __m128d _mm_or_pd(__m128d a, __m128d b)
```

SIMD Floating-Point Exceptions

None.

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
	If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS in CR0 is set.
#UD	If EM in CR0 is set.
	If OSFXSR in CR4 is 0.
	If CPUID feature flag SSE2 is 0.

Real-Address Mode Exceptions

#GP(0)	If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If TS in CR0 is set.
#UD	If EM in CR0 is set.
	If OSFXSR in CR4 is 0.
	If CPUID feature flag SSE2 is 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.

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

Opcode	Instruction	Description
0F 56 /r	ORPS *xmm1*, *xmm2/m128*	Bitwise OR of *xmm2/m128* and *xmm1.*

Description

Performs a bitwise logical OR of the four packed single-precision floating-point values from the source operand (second operand) and the destination operand (first operand), and stores the result in the destination operand. The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register.

Operation

```
DEST[127-0] ← DEST[127-0] BitwiseOR SRC[127-0];
```

Intel C/C++ Compiler Intrinsic Equivalent

```
ORPS            __m128 _mm_or_ps(__m128 a, __m128 b)
```

SIMD Floating-Point Exceptions

None.

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
	If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS in CR0 is set.
#UD	If EM in CR0 is set.
	If OSFXSR in CR4 is 0.
	If CPUID feature flag SSE is 0.

Real-Address Mode Exceptions

#GP(0)	If a memory operand is not aligned on a 16-byte boundary, regardless of segment.

	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If TS in CR0 is set.
#UD	If EM in CR0 is set.
	If OSFXSR in CR4 is 0.
	If CPUID feature flag SSE is 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.

OUT—Output to Port

Opcode	Instruction	Description
E6 *ib*	OUT *imm8*, AL	Output byte in AL to I/O port address *imm8*.
E7 *ib*	OUT *imm8*, AX	Output word in AX to I/O port address *imm8.*
E7 *ib*	OUT *imm8*, EAX	Output doubleword in EAX to I/O port address *imm8.*
EE	OUT DX, AL	Output byte in AL to I/O port address in DX.
EF	OUT DX, AX	Output word in AX to I/O port address in DX.
EF	OUT DX, EAX	Output doubleword in EAX to I/O port address in DX.

Description

Copies the value from the second operand (source operand) to the I/O port specified with the destination operand (first operand). The source operand can be register AL, AX, or EAX, depending on the size of the port being accessed (8, 16, or 32 bits, respectively); the destination operand can be a byte-immediate or the DX register. Using a byte immediate allows I/O port addresses 0 to 255 to be accessed; using the DX register as a source operand allows I/O ports from 0 to 65,535 to be accessed.

The size of the I/O port being accessed is determined by the opcode for an 8-bit I/O port or by the operand-size attribute of the instruction for a 16- or 32-bit I/O port.

At the machine code level, I/O instructions are shorter when accessing 8-bit I/O ports. Here, the upper eight bits of the port address will be 0.

This instruction is only useful for accessing I/O ports located in the processor's I/O address space. See Chapter 13, *Input/Output*, in the *IA-32 Intel Architecture Software Developer's Manual, Volume 1*, for more information on accessing I/O ports in the I/O address space.

IA-32 Architecture Compatibility

After executing an OUT instruction, the Pentium processor insures that the EWBE# pin has been sampled active before it begins to execute the next instruction. (Note that the instruction can be prefetched if EWBE# is not active, but it will not be executed until the EWBE# pin is sampled active.) Only the Pentium processor family has the EWBE# pin; the other IA-32 processors do not.

Operation

```
IF ((PE = 1) AND ((CPL > IOPL) OR (VM = 1)))
    THEN (* Protected mode with CPL > IOPL or virtual-8086 mode *)
        IF (Any I/O Permission Bit for I/O port being accessed = 1)
            THEN (* I/O operation is not allowed *)
                #GP(0);
            ELSE ( * I/O operation is allowed *)
                DEST ← SRC; (* Writes to selected I/O port *)
        FI;
    ELSE (Real Mode or Protected Mode with CPL ≤ IOPL *)
        DEST ← SRC; (* Writes to selected I/O port *)
FI;
```

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If the CPL is greater than (has less privilege) the I/O privilege level (IOPL) and any of the corresponding I/O permission bits in TSS for the I/O port being accessed is 1.

Real-Address Mode Exceptions

None.

Virtual-8086 Mode Exceptions

#GP(0)	If any of the I/O permission bits in the TSS for the I/O port being accessed is 1.

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

Opcode	Instruction	Description
6E	OUTS DX, m8	Output byte from memory location specified in DS:(E)SI to I/O port specified in DX.
6F	OUTS DX, m16	Output word from memory location specified in DS:(E)SI to I/O port specified in DX.
6F	OUTS DX, m32	Output doubleword from memory location specified in DS:(E)SI to I/O port specified in DX.
6E	OUTSB	Output byte from memory location specified in DS:(E)SI to I/O port specified in DX.
6F	OUTSW	Output word from memory location specified in DS:(E)SI to I/O port specified in DX.
6F	OUTSD	Output doubleword from memory location specified in DS:(E)SI to I/O port specified in DX.

Description

Copies data from the source operand (second operand) to the I/O port specified with the destination operand (first operand). The source operand is a memory location, the address of which is read from either the DS:ESI or the DS:SI 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.) The destination operand is an I/O port address (from 0 to 65,535) that is read from the DX register. The size of the I/O port being accessed (that is, the size of the source and destination operands) is determined by the opcode for an 8-bit I/O port or by the operand-size attribute of the instruction for a 16- or 32-bit I/O port.

At the assembly-code level, two forms of this instruction are allowed: the “explicit-operands” form and the “no-operands” form. The explicit-operands form (specified with the OUTS mnemonic) allows the source and destination operands to be specified explicitly. Here, the source operand should be a symbol that indicates the size of the I/O port and the source address, and the destination operand must be DX. This explicit-operands form is provided to allow documentation; however, note that the documentation provided by this form can be misleading. That is, the source operand symbol must specify the correct **type** (size) of the operand (byte, word, or doubleword), but it does not have to specify the correct **location**. The location is always specified by the DS:(E)SI registers, which must be loaded correctly before the OUTS instruction is executed.

The no-operands form provides “short forms” of the byte, word, and doubleword versions of the OUTS instructions. Here also DS:(E)SI is assumed to be the source operand and DX is assumed to be the destination operand. The size of the I/O port is specified with the choice of mnemonic: OUTSB (byte), OUTSW (word), or OUTSD (doubleword).

After the byte, word, or doubleword is transferred from the memory location to the I/O port, the (E)SI register is incremented or decremented automatically according to the setting of the DF flag in the EFLAGS register. (If the DF flag is 0, the (E)SI register is incremented; if the DF flag is 1, the (E)SI register is decremented.) The (E)SI register is incremented or decremented by 1 for byte operations, by 2 for word operations, or by 4 for doubleword operations.

The OUTS, OUTSB, OUTSW, and OUTSD instructions can be preceded by the REP prefix for block input of ECX bytes, words, or doublewords. See "REP/REPE/REPZ/REPNE /REPNZ—Repeat String Operation Prefix" in this chapter for a description of the REP prefix. This instruction is only useful for accessing I/O ports located in the processor's I/O address space. See Chapter 13, *Input/Output*, in the *IA-32 Intel Architecture Software Developer's Manual, Volume 1*, for more information on accessing I/O ports in the I/O address space.

IA-32 Architecture Compatibility

After executing an OUTS, OUTSB, OUTSW, or OUTSD instruction, the Pentium processor insures that the EWBE# pin has been sampled active before it begins to execute the next instruction. (Note that the instruction can be prefetched if EWBE# is not active, but it will not be executed until the EWBE# pin is sampled active.) Only the Pentium processor family has the EWBE# pin; the other IA-32 processors do not. For the Pentium 4, Intel Xeon, and P6 family processors, upon execution of an OUTS, OUTSB, OUTSW, or OUTSD instruction, the processor will not execute the next instruction until the data phase of the transaction is complete.

Operation

```
IF ((PE = 1) AND ((CPL > IOPL) OR (VM = 1)))
    THEN (* Protected mode with CPL > IOPL or virtual-8086 mode *)
        IF (Any I/O Permission Bit for I/O port being accessed = 1)
            THEN (* I/O operation is not allowed *)
                #GP(0);
            ELSE ( * I/O operation is allowed *)
                DEST ← SRC; (* Writes to I/O port *)
        FI;
    ELSE (Real Mode or Protected Mode with CPL ≤ IOPL *)
        DEST ← SRC; (* Writes to I/O port *)
FI;
IF (byte transfer)
    THEN IF DF = 0
        THEN (E)SI ← (E)SI + 1;
        ELSE (E)SI ← (E)SI – 1;
    FI;
    ELSE IF (word transfer)
        THEN IF DF = 0
            THEN (E)SI ← (E)SI + 2;
            ELSE (E)SI ← (E)SI – 2;
        FI;
        ELSE (* doubleword transfer *)
            THEN IF DF = 0
                THEN (E)SI ← (E)SI + 4;
                ELSE (E)SI ← (E)SI – 4;
            FI; FI; FI;
```

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If the CPL is greater than (has less privilege) the I/O privilege level (IOPL) and any of the corresponding I/O permission bits in TSS for the I/O port being accessed is 1.
	If a memory operand effective address is outside the limit of the CS, DS, ES, FS, or GS segment.
	If the segment register contains a null segment selector.
#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.

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.

Virtual-8086 Mode Exceptions

#GP(0)	If any of the I/O permission bits in the TSS for the I/O port being accessed is 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

PACKSSWB/PACKSSDW—Pack with Signed Saturation

Opcode	Instruction	Description
0F 63 /r	PACKSSWB *mm1, mm2/m64*	Converts 4 packed signed word integers from *mm1* and from *mm2/m64* into 8 packed signed byte integers in *mm1* using signed saturation.
66 0F 63 /r	PACKSSWB *xmm1, xmm2/m128*	Converts 8 packed signed word integers from *xmm1* and from *xmm2/m128* into 16 packed signed byte integers in *xmm1* using signed saturation.
0F 6B /r	PACKSSDW *mm1, mm2/m64*	Converts 2 packed signed doubleword integers from *mm1* and from *mm2/m64* into 4 packed signed word integers in *mm1* using signed saturation.
66 0F 6B /r	PACKSSDW *xmm1, xmm2/m128*	Converts 4 packed signed doubleword integers from *xmm1* and from *xmm2/m128* into 8 packed signed word integers in *xmm1* using signed saturation.

Description

Converts packed signed word integers into packed signed byte integers (PACKSSWB) or converts packed signed doubleword integers into packed signed word integers (PACKSSDW), using saturation to handle overflow conditions. See Figure 4-1 for an example of the packing operation.

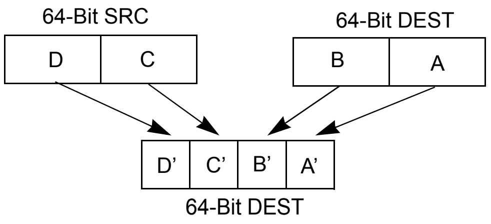


Figure 4-1. Operation of the PACKSSDW Instruction Using 64-bit Operands.

The PACKSSWB instruction converts 4 or 8 signed word integers from the destination operand (first operand) and 4 or 8 signed word integers from the source operand (second operand) into 8 or 16 signed byte integers and stores the result in the destination operand. If a signed word integer value is beyond the range of a signed byte integer (that is, greater than 7FH for a positive integer or greater than 80H for a negative integer), the saturated signed byte integer value of 7FH or 80H, respectively, is stored in the destination.

The PACKSSDW instruction packs 2 or 4 signed doublewords from the destination operand (first operand) and 2 or 4 signed doublewords from the source operand (second operand) into 4 or 8 signed words in the destination operand (see Figure 4-1). If a signed doubleword integer value is beyond the range of a signed word (that is, greater than 7FFFH for a positive integer or greater than 8000H for a negative integer), the saturated signed word integer value of 7FFFH or 8000H, respectively, is stored into the destination.

The PACKSSWB and PACKSSDW instructions operate on either 64-bit or 128-bit operands. When operating on 64-bit operands, the destination operand must be an MMX technology register and the source operand can be either an MMX technology register or a 64-bit memory location. When operating on 128-bit operands, the destination operand must be an XMM register and the source operand can be either an XMM register or a 128-bit memory location.

Operation

```
PACKSSWB instruction with 64-bit operands
    DEST[7..0] ← SaturateSignedWordToSignedByte DEST[15..0];
    DEST[15..8] ← SaturateSignedWordToSignedByte DEST[31..16];
    DEST[23..16] ← SaturateSignedWordToSignedByte DEST[47..32];
    DEST[31..24] ← SaturateSignedWordToSignedByte DEST[63..48];
    DEST[39..32] ← SaturateSignedWordToSignedByte SRC[15..0];
    DEST[47..40] ← SaturateSignedWordToSignedByte SRC[31..16];
    DEST[55..48] ← SaturateSignedWordToSignedByte SRC[47..32];
    DEST[63..56] ← SaturateSignedWordToSignedByte SRC[63..48];

PACKSSDW instruction with 64-bit operands
    DEST[15..0] ← SaturateSignedDoublewordToSignedWord DEST[31..0];
    DEST[31..16] ← SaturateSignedDoublewordToSignedWord DEST[63..32];
    DEST[47..32] ← SaturateSignedDoublewordToSignedWord SRC[31..0];
    DEST[63..48] ← SaturateSignedDoublewordToSignedWord SRC[63..32];

PACKSSWB instruction with 128-bit operands
    DEST[7-0]   ← SaturateSignedWordToSignedByte (DEST[15-0]);
    DEST[15-8]  ← SaturateSignedWordToSignedByte (DEST[31-16]);
    DEST[23-16] ← SaturateSignedWordToSignedByte (DEST[47-32]);
    DEST[31-24] ← SaturateSignedWordToSignedByte (DEST[63-48]);
    DEST[39-32] ← SaturateSignedWordToSignedByte (DEST[79-64]);
    DEST[47-40] ← SaturateSignedWordToSignedByte (DEST[95-80]);
    DEST[55-48] ← SaturateSignedWordToSignedByte (DEST[111-96]);
    DEST[63-56] ← SaturateSignedWordToSignedByte (DEST[127-112]);
    DEST[71-64] ← SaturateSignedWordToSignedByte (SRC[15-0]);
    DEST[79-72] ← SaturateSignedWordToSignedByte (SRC[31-16]);
    DEST[87-80] ← SaturateSignedWordToSignedByte (SRC[47-32]);
    DEST[95-88] ← SaturateSignedWordToSignedByte (SRC[63-48]);
    DEST[103-96]  ← SaturateSignedWordToSignedByte (SRC[79-64]);
    DEST[111-104] ← SaturateSignedWordToSignedByte (SRC[95-80]);
    DEST[119-112] ← SaturateSignedWordToSignedByte (SRC[111-96]);
    DEST[127-120] ← SaturateSignedWordToSignedByte (SRC[127-112]);

PACKSSDW instruction with 128-bit operands
    DEST[15-0]  ← SaturateSignedDwordToSignedWord (DEST[31-0]);
    DEST[31-16] ← SaturateSignedDwordToSignedWord (DEST[63-32]);
    DEST[47-32] ← SaturateSignedDwordToSignedWord (DEST[95-64]);
    DEST[63-48] ← SaturateSignedDwordToSignedWord (DEST[127-96]);
    DEST[79-64] ← SaturateSignedDwordToSignedWord (SRC[31-0]);
    DEST[95-80] ← SaturateSignedDwordToSignedWord (SRC[63-32]);
    DEST[111-96]  ← SaturateSignedDwordToSignedWord (SRC[95-64]);
    DEST[127-112] ← SaturateSignedDwordToSignedWord (SRC[127-96]);
```

Intel C/C++ Compiler Intrinsic Equivalents

```
__m64 _mm_packs_pi16(__m64 m1, __m64 m2)
__m64 _mm_packs_pi32 (__m64 m1, __m64 m2)
```

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.
	(128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If EM in CR0 is set.
	(128-bit operations only) If OSFXSR in CR4 is 0.
#NM	If TS in CR0 is set.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#GP(0)	(128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
	If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set.
	(128-bit operations only) If OSFXSR in CR4 is 0.

#NM	If TS in CR0 is set.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made.

PACKUSWB—Pack with Unsigned Saturation

Opcode	Instruction	Description
0F 67 /r	PACKUSWB *mm, mm/m64*	Converts 4 signed word integers from *mm* and 4 signed word integers from *mm/m64* into 8 unsigned byte integers in *mm* using unsigned saturation.
66 0F 67 /r	PACKUSWB *xmm1*, *xmm2/m128*	Converts 8 signed word integers from *xmm1* and 8 signed word integers from *xmm2/m128* into 16 unsigned byte integers in *xmm1* using unsigned saturation.

Description

Converts 4 or 8 signed word integers from the destination operand (first operand) and 4 or 8 signed word integers from the source operand (second operand) into 8 or 16 unsigned byte integers and stores the result in the destination operand. (See Figure 4-1 for an example of the packing operation.) If a signed word integer value is beyond the range of an unsigned byte integer (that is, greater than FFH or less than 00H), the saturated unsigned byte integer value of FFH or 00H, respectively, is stored in the destination.

The PACKUSWB instruction operates on either 64-bit or 128-bit operands. When operating on 64-bit operands, the destination operand must be an MMX technology register and the source operand can be either an MMX technology register or a 64-bit memory location. When operating on 128-bit operands, the destination operand must be an XMM register and the source operand can be either an XMM register or a 128-bit memory location.

Operation

```
PACKUSWB instruction with 64-bit operands:
    DEST[7..0] ← SaturateSignedWordToUnsignedByte DEST[15..0];
    DEST[15..8] ← SaturateSignedWordToUnsignedByte DEST[31..16];
    DEST[23..16] ← SaturateSignedWordToUnsignedByte DEST[47..32];
    DEST[31..24] ← SaturateSignedWordToUnsignedByte DEST[63..48];
    DEST[39..32] ← SaturateSignedWordToUnsignedByte SRC[15..0];
    DEST[47..40] ← SaturateSignedWordToUnsignedByte SRC[31..16];
    DEST[55..48] ← SaturateSignedWordToUnsignedByte SRC[47..32];
    DEST[63..56] ← SaturateSignedWordToUnsignedByte SRC[63..48];

PACKUSWB instruction with 128-bit operands:
    DEST[7-0]   ← SaturateSignedWordToUnsignedByte (DEST[15-0]);
    DEST[15-8]  ← SaturateSignedWordToUnsignedByte (DEST[31-16]);
    DEST[23-16] ← SaturateSignedWordToUnsignedByte (DEST[47-32]);
    DEST[31-24] ← SaturateSignedWordToUnsignedByte (DEST[63-48]);
    DEST[39-32] ← SaturateSignedWordToUnsignedByte (DEST[79-64]);
    DEST[47-40] ← SaturateSignedWordToUnsignedByte (DEST[95-80]);
    DEST[55-48] ← SaturateSignedWordToUnsignedByte (DEST[111-96]);
    DEST[63-56] ← SaturateSignedWordToUnsignedByte (DEST[127-112]);
    DEST[71-64] ← SaturateSignedWordToUnsignedByte (SRC[15-0]);
    DEST[79-72] ← SaturateSignedWordToUnsignedByte (SRC[31-16]);
    DEST[87-80] ← SaturateSignedWordToUnsignedByte (SRC[47-32]);
    DEST[95-88] ← SaturateSignedWordToUnsignedByte (SRC[63-48]);
    DEST[103-96]  ← SaturateSignedWordToUnsignedByte (SRC[79-64]);
    DEST[111-104] ← SaturateSignedWordToUnsignedByte (SRC[95-80]);
    DEST[119-112] ← SaturateSignedWordToUnsignedByte (SRC[111-96]);
    DEST[127-120] ← SaturateSignedWordToUnsignedByte (SRC[127-112]);
```

Intel C/C++ Compiler Intrinsic Equivalent

__m64 _mm_packs_pu16(__m64 m1, __m64 m2)

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.
	(128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If EM in CR0 is set.
	128-bit operations will generate #UD only if OSFXSR in CR4 is 0. Execution of 128-bit instructions on a non-SSE2 capable processor (one that is MMX technology capable) will result in the instruction operating on the mm registers, not #UD.
#NM	If TS in CR0 is set.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#GP(0)	(128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
	If any part of the operand lies outside of the effective address space from 0 to FFFFH.

#UD	If EM in CR0 is set.
	128-bit operations will generate #UD only if OSFXSR in CR4 is 0. Execution of 128-bit instructions on a non-SSE2 capable processor (one that is MMX technology capable) will result in the instruction operating on the mm registers, not #UD.
#NM	If TS in CR0 is set.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made.

PADDB/PADDW/PADDD—Add Packed Integers

Opcode	Instruction	Description
0F FC /r	PADDB *mm, mm/m64*	Add packed byte integers from *mm/m64* and *mm*.
66 0F FC /r	PADDB *xmm1,xmm2/m128*	Add packed byte integers from *xmm2/m128* and *xmm1*.
0F FD /r	PADDW *mm, mm/m64*	Add packed word integers from *mm/m64* and *mm*.
66 0F FD /r	PADDW *xmm1, xmm2/m128*	Add packed word integers from *xmm2/m128* and *xmm1*.
0F FE /r	PADDD *mm, mm/m64*	Add packed doubleword integers from *mm/m64* and *mm*.
66 0F FE /r	PADDD *xmm1, xmm2/m128*	Add packed doubleword integers from *xmm2/m128* and *xmm1*.

Description

Performs an SIMD add of the packed integers from the source operand (second operand) and the destination operand (first operand), and stores the packed integer results in the destination operand. See Figure 9-4 in the *IA-32 Intel Architecture Software Developer's Manual, Volume 1* for an illustration of an SIMD operation. Overflow is handled with wraparound, as described in the following paragraphs.

These instructions can operate on either 64-bit or 128-bit operands. When operating on 64-bit operands, the destination operand must be an MMX technology register and the source operand can be either an MMX technology register or a 64-bit memory location. When operating on 128-bit operands, the destination operand must be an XMM register and the source operand can be either an XMM register or a 128-bit memory location.

The PADDB instruction adds packed byte integers. When an individual result is too large to be represented in 8 bits (overflow), the result is wrapped around and the low 8 bits are written to the destination operand (that is, the carry is ignored).

The PADDW instruction adds packed word integers. When an individual result is too large to be represented in 16 bits (overflow), the result is wrapped around and the low 16 bits are written to the destination operand.

The PADDD instruction adds packed doubleword integers. When an individual result is too large to be represented in 32 bits (overflow), the result is wrapped around and the low 32 bits are written to the destination operand.

Note that the PADDB, PADDW, and PADDD instructions can operate on either unsigned or signed (two's complement notation) packed integers; however, it does not set bits in the EFLAGS register to indicate overflow and/or a carry. To prevent undetected overflow conditions, software must control the ranges of values operated on.

Operation

```
PADDB instruction with 64-bit operands:
    DEST[7..0] ← DEST[7..0] + SRC[7..0];
    * repeat add operation for 2nd through 7th byte *;
    DEST[63..56] ← DEST[63..56] + SRC[63..56];

PADDB instruction with 128-bit operands:
    DEST[7-0] ← DEST[7-0] + SRC[7-0];
    * repeat add operation for 2nd through 14th byte *;
    DEST[127-120] ← DEST[111-120] + SRC[127-120];

PADDW instruction with 64-bit operands:
    DEST[15..0] ← DEST[15..0] + SRC[15..0];
    * repeat add operation for 2nd and 3th word *;
    DEST[63..48] ← DEST[63..48] + SRC[63..48];

PADDW instruction with 128-bit operands:
    DEST[15-0]  ← DEST[15-0] + SRC[15-0];
    * repeat add operation for 2nd through 7th word *;
    DEST[127-112] ← DEST[127-112] + SRC[127-112];

PADDD instruction with 64-bit operands:
    DEST[31..0] ← DEST[31..0] + SRC[31..0];
    DEST[63..32] ← DEST[63..32] + SRC[63..32];

PADDD instruction with 128-bit operands:
    DEST[31-0]  ← DEST[31-0]  + SRC[31-0];
    * repeat add operation for 2nd and 3th doubleword *;
    DEST[127-96] ← DEST[127-96] + SRC[127-96];
```

Intel C/C++ Compiler Intrinsic Equivalents

PADDB	__m64 _mm_add_pi8(__m64 m1, __m64 m2)
PADDB	__m128i_mm_add_epi8 (__m128ia,__m128ib)
PADDW	__m64 _mm_addw_pi16(__m64 m1, __m64 m2)
PADDW	__m128i _mm_add_epi16 (__m128i a, __m128i b)
PADDD	__m64 _mm_add_pi32(__m64 m1, __m64 m2)
PADDD	__m128i _mm_add_epi32 (__m128i a, __m128i b)

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.
	(128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment.

#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If EM in CR0 is set.
	128-bit operations will generate #UD only if OSFXSR in CR4 is 0. Execution of 128-bit instructions on a non-SSE2 capable processor (one that is MMX technology capable) will result in the instruction operating on the mm registers, not #UD.
#NM	If TS in CR0 is set.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#GP(0)	(128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
	If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set.
	128-bit operations will generate #UD only if OSFXSR in CR4 is 0. Execution of 128-bit instructions on a non-SSE2 capable processor (one that is MMX technology capable) will result in the instruction operating on the mm registers, not #UD.
#NM	If TS in CR0 is set.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made.

PADDQ—Add Packed Quadword Integers

Opcode	Instruction	Description
0F D4 /r	PADDQ *mm1,mm2/m64*	Add quadword integer *mm2/m64* to *mm1.*
66 0F D4 /r	PADDQ *xmm1,xmm2/m128*	Add packed quadword integers *xmm2/m128* to *xmm1.*

Description

Adds the first operand (destination operand) to the second operand (source operand) and stores the result in the destination operand. The source operand can be a quadword integer stored in an MMX technology register or a 64-bit memory location, or it can be two packed quadword integers stored in an XMM register or an 128-bit memory location. The destination operand can be a quadword integer stored in an MMX technology register or two packed quadword integers stored in an XMM register. When packed quadword operands are used, an SIMD add is performed. When a quadword result is too large to be represented in 64 bits (overflow), the result is wrapped around and the low 64 bits are written to the destination element (that is, the carry is ignored).

Note that the PADDQ instruction can operate on either unsigned or signed (two's complement notation) integers; however, it does not set bits in the EFLAGS register to indicate overflow and/or a carry. To prevent undetected overflow conditions, software must control the ranges of the values operated on.

Operation

```
PADDQ instruction with 64-Bit operands:
    DEST[63-0] ¨ DEST[63-0] + SRC[63-0];

PADDQ instruction with 128-Bit operands:
    DEST[63-0] ¨ DEST[63-0] + SRC[63-0];
    DEST[127-64] ¨ DEST[127-64] + SRC[127-64];
```

Intel C/C++ Compiler Intrinsic Equivalents

PADDQ	__m64 _mm_add_si64 (__m64 a, __m64 b)
PADDQ	__m128i _mm_add_epi64 (__m128i a, __m128i b)

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.
	(128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment.

#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If EM in CR0 is set.
	(128-bit operations only) If OSFXSR in CR4 is 0.
	(128-bit operations only) If CPUID feature flag SSE2 is 0.
#NM	If TS in CR0 is set.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#GP(0)	(128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
	If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set.
	(128-bit operations only) If OSFXSR in CR4 is 0.
	(128-bit operations only) If CPUID feature flag SSE2 is 0.
#NM	If TS in CR0 is set.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made.

Numeric Exceptions

None.

PADDSB/PADDSW—Add Packed Signed Integers with Signed Saturation

Opcode	Instruction	Description
0F EC /r	PADDSB *mm, mm/m64*	Add packed signed byte integers from *mm/m64 and mm* and saturate the results.
66 0F EC /r	PADDSB *xmm1,*	Add packed signed byte integers from *xmm2/m128* and *xmm1* saturate the results.
0F ED /r	PADDSW *mm, mm/m64*	Add packed signed word integers from *mm/m64 and mm* and saturate the results.
66 0F ED /r	PADDSW *xmm1, xmm2/m128*	Add packed signed word integers from *xmm2/m128* and *xmm1* and saturate the results.

Description

Performs an SIMD add of the packed signed integers from the source operand (second operand) and the destination operand (first operand), and stores the packed integer results in the destination operand. See Figure 9-4 in the *IA-32 Intel Architecture Software Developer's Manual, Volume 1* for an illustration of an SIMD operation. Overflow is handled with signed saturation, as described in the following paragraphs.

These instructions can operate on either 64-bit or 128-bit operands. When operating on 64-bit operands, the destination operand must be an MMX technology register and the source operand can be either an MMX technology register or a 64-bit memory location. When operating on 128-bit operands, the destination operand must be an XMM register and the source operand can be either an XMM register or a 128-bit memory location.

The PADDSB instruction adds packed signed byte integers. When an individual byte result is beyond the range of a signed byte integer (that is, greater than 7FH or less than 80H), the saturated value of 7FH or 80H, respectively, is written to the destination operand.

The PADDSW instruction adds packed signed word integers. When an individual word result is beyond the range of a signed word integer (that is, greater than 7FFFH or less than 8000H), the saturated value of 7FFFH or 8000H, respectively, is written to the destination operand.

Operation

```
PADDSB instruction with 64-bit operands:
    DEST[7..0] ← SaturateToSignedByte(DEST[7..0] + SRC (7..0]) ;
    * repeat add operation for 2nd through 7th bytes *;
    DEST[63..56] ← SaturateToSignedByte(DEST[63..56] + SRC[63..56] );

PADDSB instruction with 128-bit operands:
    DEST[7-0] ← SaturateToSignedByte (DEST[7-0] + SRC[7-0]);
    * repeat add operation for 2nd through 14th bytes *;
    DEST[127-120] ← SaturateToSignedByte (DEST[111-120] + SRC[127-120]);
PADDSW instruction with 64-bit operands
    DEST[15..0] ¨ SaturateToSignedWord(DEST[15..0] + SRC[15..0] );
    * repeat add operation for 2nd and 7th words *;
    DEST[63..48] ¨ SaturateToSignedWord(DEST[63..48] + SRC[63..48] );

PADDSW instruction with 128-bit operands
    DEST[15-0]  ← SaturateToSignedWord (DEST[15-0] + SRC[15-0]);
    * repeat add operation for 2nd through 7th words *;
    DEST[127-112] ← SaturateToSignedWord (DEST[127-112] + SRC[127-112]);
```

Intel C/C++ Compiler Intrinsic Equivalents

PADDSB	__m64 _mm_adds_pi8(__m64 m1, __m64 m2)
PADDSB	__m128i _mm_adds_epi8 (__m128i a, __m128i b)
PADDSW	__m64 _mm_adds_pi16(__m64 m1, __m64 m2)
PADDSW	__m128i _mm_adds_epi16 (__m128i a, __m128i b)

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.
	(128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If EM in CR0 is set.
	128-bit operations will generate #UD only if OSFXSR in CR4 is 0. Execution of 128-bit instructions on a non-SSE2 capable processor (one that is MMX technology capable) will result in the instruction operating on the mm registers, not #UD.
#NM	If TS in CR0 is set.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#GP(0)	(128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment. If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set. 128-bit operations will generate #UD only if OSFXSR in CR4 is 0. Execution of 128-bit instructions on a non-SSE2 capable processor (one that is MMX technology capable) will result in the instruction operating on the mm registers, not #UD.
#NM	If TS in CR0 is set.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made.

PADDUSB/PADDUSW—Add Packed Unsigned Integers with Unsigned Saturation

Opcode	Instruction	Description
0F DC /r	PADDUSB *mm, mm/m64*	Add packed unsigned byte integers from *mm/m64 and mm* and saturate the results.
66 0F DC /r	PADDUSB xmm1, xmm2/m128	Add packed unsigned byte integers from *xmm2/m128* and *xmm1* saturate the results.
0F DD /r	PADDUSW *mm, mm/m64*	Add packed unsigned word integers from *mm/m64 and mm* and saturate the results.
66 0F DD /r	PADDUSW xmm1, xmm2/m128	Add packed unsigned word integers from *xmm2/m128* to *xmm1* and saturate the results.

Description

Performs an SIMD add of the packed unsigned integers from the source operand (second operand) and the destination operand (first operand), and stores the packed integer results in the destination operand. See Figure 9-4 in the *IA-32 Intel Architecture Software Developer's Manual, Volume 1* for an illustration of an SIMD operation. Overflow is handled with unsigned saturation, as described in the following paragraphs.

These instructions can operate on either 64-bit or 128-bit operands. When operating on 64-bit operands, the destination operand must be an MMX technology register and the source operand can be either an MMX technology register or a 64-bit memory location. When operating on 128-bit operands, the destination operand must be an XMM register and the source operand can be either an XMM register or a 128-bit memory location.

The PADDUSB instruction adds packed unsigned byte integers. When an individual byte result is beyond the range of an unsigned byte integer (that is, greater than FFH), the saturated value of FFH is written to the destination operand.

The PADDUSW instruction adds packed unsigned word integers. When an individual word result is beyond the range of an unsigned word integer (that is, greater than FFFFH), the saturated value of FFFFH is written to the destination operand.

Operation

```
PADDUSB instruction with 64-bit operands:
    DEST[7..0] ← SaturateToUnsignedByte(DEST[7..0] + SRC (7..0] );
    * repeat add operation for 2nd through 7th bytes *:
    DEST[63..56] ← SaturateToUnsignedByte(DEST[63..56] + SRC[63..56]

PADDUSB instruction with 128-bit operands:
    DEST[7-0] ← SaturateToUnsignedByte (DEST[7-0] + SRC[7-0]);
    * repeat add operation for 2nd through 14th bytes *:
    DEST[127-120] ← SaturateToUnSignedByte (DEST[127-120] + SRC[127-120]);
PADDUSW instruction with 64-bit operands:
    DEST[15..0] ¨ SaturateToUnsignedWord(DEST[15..0] + SRC[15..0] );
    * repeat add operation for 2nd and 3rd words *:
    DEST[63..48] ¨ SaturateToUnsignedWord(DEST[63..48] + SRC[63..48] );

PADDUSW instruction with 128-bit operands:
    DEST[15-0]  ¨ SaturateToUnsignedWord (DEST[15-0] + SRC[15-0]);
    * repeat add operation for 2nd through 7th words *:
    DEST[127-112] ← SaturateToUnSignedWord (DEST[127-112] + SRC[127-112]);
```

Intel C/C++ Compiler Intrinsic Equivalents

PADDUSB	__m64 _mm_adds_pu8(__m64 m1, __m64 m2)
PADDUSW	__m64 _mm_adds_pu16(__m64 m1, __m64 m2)
PADDUSB	__m128i _mm_adds_epu8 (__m128i a, __m128i b)
PADDUSW	__m128i _mm_adds_epu16 (__m128i a, __m128i b)

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.
	(128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If EM in CR0 is set.
	128-bit operations will generate #UD only if OSFXSR in CR4 is 0. Execution of 128-bit instructions on a non-SSE2 capable processor (one that is MMX technology capable) will result in the instruction operating on the mm registers, not #UD.
#NM	If TS in CR0 is set.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#GP(0)	(128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment. If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set. 128-bit operations will generate #UD only if OSFXSR in CR4 is 0. Execution of 128-bit instructions on a non-SSE2 capable processor (one that is MMX technology capable) will result in the instruction operating on the mm registers, not #UD.
#NM	If TS in CR0 is set.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made.

Numeric Exceptions

None.

PAND—Logical AND

Opcode	Instruction	Description
0F DB /r	PAND *mm, mm/m64*	Bitwise AND *mm/m64* and mm.
66 0F DB /r	PAND *xmm1*, xmm2/m128	Bitwise AND of *xmm2/m128* and *xmm1*.

Description

Performs a bitwise logical AND operation on the source operand (second operand) and the destination operand (first operand) and stores the result in the destination operand. The source operand can be an MMX technology register or a 64-bit memory location or it can be an XMM register or a 128-bit memory location. The destination operand can be an MMX technology register or an XMM register. Each bit of the result is set to 1 if the corresponding bits of the first and second operands are 1; otherwise, it is set to 0.

Operation

```
DEST ← DEST AND SRC;
```

Intel C/C++ Compiler Intrinsic Equivalent

PAND	__m64 _mm_and_si64 (__m64 m1, __m64 m2)
PAND	__m128i _mm_and_si128 (__m128i a, __m128i b)

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.
	(128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If EM in CR0 is set.
	128-bit operations will generate #UD only if OSFXSR in CR4 is 0. Execution of 128-bit instructions on a non-SSE2 capable processor (one that is MMX technology capable) will result in the instruction operating on the mm registers, not #UD.
#NM	If TS in CR0 is set.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.

#PF(fault-code)	If a page fault occurs.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#GP(0)	(128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment. If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set. 128-bit operations will generate #UD only if OSFXSR in CR4 is 0. Execution of 128-bit instructions on a non-SSE2 capable processor (one that is MMX technology capable) will result in the instruction operating on the mm registers, not #UD.
#NM	If TS in CR0 is set.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made.

Numeric Exceptions

None.

PANDN—Logical AND NOT

Opcode	Instruction	Description
0F DF /r	PANDN *mm, mm/m64*	Bitwise AND NOT of *mm/m64* and *mm*.
66 0F DF /r	PANDN *xmm1*, *xmm2/m128*	Bitwise AND NOT of *xmm2/m128* and *xmm1*.

Description

Performs a bitwise logical NOT of the destination operand (first operand), then performs a bitwise logical AND of the source operand (second operand) and the inverted destination operand. The result is stored in the destination operand. The source operand can be an MMX technology register or a 64-bit memory location or it can be an XMM register or a 128-bit memory location. The destination operand can be an MMX technology register or an XMM register. Each bit of the result is set to 1 if the corresponding bit in the first operand is 0 and the corresponding bit in the second operand is 1; otherwise, it is set to 0.

Operation

```
DEST ← (NOT DEST) AND SRC;
```

Intel C/C++ Compiler Intrinsic Equivalent

PANDN	__m64 _mm_andnot_si64 (__m64 m1, __m64 m2)
PANDN	__m128i _mm_andnot_si128 (__m128i a, __m128i b)

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.
	(128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If EM in CR0 is set.
	128-bit operations will generate #UD only if OSFXSR in CR4 is 0. Execution of 128-bit instructions on a non-SSE2 capable processor (one that is MMX technology capable) will result in the instruction operating on the mm registers, not #UD.
#NM	If TS in CR0 is set.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.

#PF(fault-code)	If a page fault occurs.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#GP(0)	(128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
	If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set.
	128-bit operations will generate #UD only if OSFXSR in CR4 is 0. Execution of 128-bit instructions on a non-SSE2 capable processor (one that is MMX technology capable) will result in the instruction operating on the mm registers, not #UD.
#NM	If TS in CR0 is set.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made.

Numeric Exceptions

None.

PAUSE—Spin Loop Hint

Opcode	Instruction	Description
F3 90	PAUSE	Gives hint to processor that improves performance of spin-wait loops.

Description

Improves the performance of spin-wait loops. When executing a “spin-wait loop,” a Pentium 4 or Intel Xeon processor suffers a severe performance penalty when exiting the loop because it detects a possible memory order violation. The PAUSE instruction provides a hint to the processor that the code sequence is a spin-wait loop. The processor uses this hint to avoid the memory order violation in most situations, which greatly improves processor performance. For this reason, it is recommended that a PAUSE instruction be placed in all spin-wait loops.

An additional function of the PAUSE instruction is to reduce the power consumed by a Pentium 4 processor while executing a spin loop. The Pentium 4 processor can execute a spin-wait loop extremely quickly, causing the processor to consume a lot of power while it waits for the resource it is spinning on to become available. Inserting a pause instruction in a spin-wait loop greatly reduces the processor’s power consumption.

This instruction was introduced in the Pentium 4 processors, but is backward compatible with all IA-32 processors. In earlier IA-32 processors, the PAUSE instruction operates like a NOP instruction. The Pentium 4 and Intel Xeon processors implement the PAUSE instruction as a pre-defined delay. The delay is finite and can be zero for some processors. This instruction does not change the architectural state of the processor (that is, it performs essentially a delaying no-op operation).

Operation

```
Execute_Next_Instruction(DELAY);
```

Protected Mode Exceptions

None.

Real-Address Mode Exceptions

None.

Virtual-8086 Mode Exceptions

None.

Numeric Exceptions

None.

PAVGB/PAVGW—Average Packed Integers

Opcode	Instruction	Description
0F E0 /r	PAVGB *mm1, mm2/m64*	Average packed unsigned byte integers from *mm2/m64* and *mm1* with rounding.
66 0F E0, /r	PAVGB *xmm1*, *xmm2/m128*	Average packed unsigned byte integers from *xmm2/m128* and *xmm1* with rounding.
0F E3 /r	PAVGW *mm1, mm2/m64*	Average packed unsigned word integers from *mm2/m64* and *mm1* with rounding.
66 0F E3 /r	PAVGW *xmm1*, *xmm2/m128*	Average packed unsigned word integers from *xmm2/m128* and *xmm1* with rounding.

Description

Performs an SIMD average of the packed unsigned integers from the source operand (second operand) and the destination operand (first operand), and stores the results in the destination operand. For each corresponding pair of data elements in the first and second operands, the elements are added together, a 1 is added to the temporary sum, and that result is shifted right one bit position. The source operand can be an MMX technology register or a 64-bit memory location or it can be an XMM register or a 128-bit memory location. The destination operand can be an MMX technology register or an XMM register.

The PAVGB instruction operates on packed unsigned bytes and the PAVGW instruction operates on packed unsigned words.

Operation

```
PAVGB instruction with 64-bit operands:
    SRC[7-0) ← (SRC[7-0) + DEST[7-0) + 1) >> 1; * temp sum before shifting is 9 bits *
    * repeat operation performed for bytes 2 through 6;
    SRC[63-56) ← (SRC[63-56) + DEST[63-56) + 1) >> 1;

PAVGW instruction with 64-bit operands:
    SRC[15-0) ← (SRC[15-0) + DEST[15-0) + 1) >> 1; * temp sum before shifting is 17 bits *
    * repeat operation performed for words 2 and 3;
    SRC[63-48) ← (SRC[63-48) + DEST[63-48) + 1) >> 1;

PAVGB instruction with 128-bit operands:
    SRC[7-0) ← (SRC[7-0) + DEST[7-0) + 1) >> 1; * temp sum before shifting is 9 bits *
    * repeat operation performed for bytes 2 through 14;
    SRC[63-56) ← (SRC[63-56) + DEST[63-56) + 1) >> 1;

PAVGW instruction with 128-bit operands:
    SRC[15-0) ← (SRC[15-0) + DEST[15-0) + 1) >> 1; * temp sum before shifting is 17 bits *
    * repeat operation performed for words 2 through 6;
    SRC[127-48) ← (SRC[127-112) + DEST[127-112) + 1) >> 1;
```

Intel C/C++ Compiler Intrinsic Equivalent

PAVGB	__m64_mm_avg_pu8 (__m64 a, __m64 b)
PAVGW	__m64_mm_avg_pu16 (__m64 a, __m64 b)
PAVGB	__m128i _mm_avg_epu8 (__m128i a, __m128i b)
PAVGW	__m128i _mm_avg_epu16 (__m128i a, __m128i b)

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.
	(128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If EM in CR0 is set.
	(128-bit operations only) If OSFXSR in CR4 is 0.
	(128-bit operations only) If CPUID feature flag SSE2 is 0.
#NM	If TS in CR0 is set.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#GP(0)	(128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
	If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set.
	(128-bit operations only) If OSFXSR in CR4 is 0.
	(128-bit operations only) If CPUID feature flag SSE2 is 0.
#NM	If TS in CR0 is set.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made.

Numeric Exceptions

None.

PCMPEQB/PCMPEQW/PCMPEQD— Compare Packed Data for Equal

Opcode	Instruction	Description
0F 74 /r	PCMPEQB *mm, mm/m64*	Compare packed bytes in *mm/m64* and *mm* for equality.
66 0F 74 /r	PCMPEQB *xmm1*, *xmm2/m128*	Compare packed bytes in *xmm2/m128* and *xmm1* for equality.
0F 75 /r	PCMPEQW *mm, mm/m64*	Compare packed words in *mm/m64* and *mm* for equality.
66 0F 75 /r	PCMPEQW *xmm1*, *xmm2/m128*	Compare packed words in *xmm2/m128* and *xmm1* for equality.
0F 76 /r	PCMPEQD *mm, mm/m64*	Compare packed doublewords in *mm/m64* and *mm* for equality.
66 0F 76 /r	PCMPEQD *xmm1*, *xmm2/m128*	Compare packed doublewords in *xmm2/m128* and *xmm1* for equality.

Description

Performs an SIMD compare for equality of the packed bytes, words, or doublewords in the destination operand (first operand) and the source operand (second operand). If a pair of data elements is equal, the corresponding data element in the destination operand is set to all 1s; otherwise, it is set to all 0s. The source operand can be an MMX technology register or a 64-bit memory location, or it can be an XMM register or a 128-bit memory location. The destination operand can be an MMX technology register or an XMM register.

The PCMPEQB instruction compares the corresponding bytes in the destination and source operands; the PCMPEQW instruction compares the corresponding words in the destination and source operands; and the PCMPEQD instruction compares the corresponding doublewords in the destination and source operands.

Operation

```
PCMPEQB instruction with 64-bit operands:
   IF DEST[7..0] = SRC[7..0]
        THEN DEST[7  0) ← FFH;
        ELSE DEST[7..0] ← 0;
   * Continue comparison of 2nd through 7th bytes in DEST and SRC *
   IF DEST[63..56] = SRC[63..56]
        THEN DEST[63..56] ← FFH;
        ELSE DEST[63..56] ← 0;

PCMPEQB instruction with 128-bit operands:
   IF DEST[7..0] = SRC[7..0]
        THEN DEST[7  0) ← FFH;
        ELSE DEST[7..0] ← 0;
   * Continue comparison of 2nd through 15th bytes in DEST and SRC *
```

```
    IF DEST[63..56] = SRC[63..56]
        THEN DEST[63..56] ← FFH;
        ELSE DEST[63..56] ← 0;

PCMPEQW instruction with 64-bit operands:
    IF DEST[15..0] = SRC[15..0]
        THEN DEST[15..0] ← FFFFH;
        ELSE DEST[15..0] ← 0;
    * Continue comparison of 2nd and 3rd words in DEST and SRC *
    IF DEST[63..48] = SRC[63..48]
        THEN DEST[63..48] ← FFFFH;
        ELSE DEST[63..48] ← 0;

PCMPEQW instruction with 128-bit operands:
    IF DEST[15..0] = SRC[15..0]
        THEN DEST[15..0] ← FFFFH;
        ELSE DEST[15..0] ← 0;
    * Continue comparison of 2nd through 7th words in DEST and SRC *
    IF DEST[63..48] = SRC[63..48]
        THEN DEST[63..48] ← FFFFH;
        ELSE DEST[63..48] ← 0;

PCMPEQD instruction with 64-bit operands:
    IF DEST[31..0] = SRC[31..0]
        THEN DEST[31..0] ← FFFFFFFFH;
        ELSE DEST[31..0] ← 0;
    IF DEST[63..32] = SRC[63..32]
        THEN DEST[63..32] ← FFFFFFFFH;
        ELSE DEST[63..32] ← 0;

PCMPEQD instruction with 128-bit operands:
    IF DEST[31..0] = SRC[31..0]
        THEN DEST[31..0] ← FFFFFFFFH;
        ELSE DEST[31..0] ← 0;
    * Continue comparison of 2nd and 3rd doublewords in DEST and SRC *
    IF DEST[63..32] = SRC[63..32]
        THEN DEST[63..32] ← FFFFFFFFH;
        ELSE DEST[63..32] ← 0;
```

Intel C/C++ Compiler Intrinsic Equivalents

PCMPEQB	__m64 _mm_cmpeq_pi8 (__m64 m1, __m64 m2)
PCMPEQW	__m64 _mm_cmpeq_pi16 (__m64 m1, __m64 m2)
PCMPEQD	__m64 _mm_cmpeq_pi32 (__m64 m1, __m64 m2)
PCMPEQB	__m128i _mm_cmpeq_epi8 (__m128i a, __m128i b)

PCMPEQW	__m128i _mm_cmpeq_epi16 (__m128i a, __m128i b)
PCMPEQD	__m128i _mm_cmpeq_epi32 (__m128i a, __m128i b)

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.
	(128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If EM in CR0 is set.
	128-bit operations will generate #UD only if OSFXSR in CR4 is 0. Execution of 128-bit instructions on a non-SSE2 capable processor (one that is MMX technology capable) will result in the instruction operating on the mm registers, not #UD.
#NM	If TS in CR0 is set.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#GP(0)	(128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
	If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set.
	128-bit operations will generate #UD only if OSFXSR in CR4 is 0. Execution of 128-bit instructions on a non-SSE2 capable processor (one that is MMX technology capable) will result in the instruction operating on the mm registers, not #UD.
#NM	If TS in CR0 is set.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made.

Numeric Exceptions

None.

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

Opcode	Instruction	Description
0F 64 /r	PCMPGTB *mm, mm/m64*	Compare packed signed byte integers in *mm* and *mm/m64* for greater than.
66 0F 64 /r	PCMPGTB *xmm1*, *xmm2/m128*	Compare packed signed byte integers in *xmm1* and *xmm2/m128* for greater than.
0F 65 /r	PCMPGTW *mm, mm/m64*	Compare packed signed word integers in *mm* and *mm/m64* for greater than.
66 0F 65 /r	PCMPGTW *xmm1*, *xmm2/m128*	Compare packed signed word integers in *xmm1* and *xmm2/m128* for greater than.
0F 66 /r	PCMPGTD *mm, mm/m64*	Compare packed signed doubleword integers in *mm* and *mm/m64* for greater than.
66 0F 66 /r	PCMPGTD *xmm1*, *xmm2/m128*	Compare packed signed doubleword integers in *xmm1* and *xmm2/m128* for greater than.

Description

Performs an SIMD signed compare for the greater value of the packed byte, word, or doubleword integers in the destination operand (first operand) and the source operand (second operand). If a data element in the destination operand is greater than the corresponding date element in the source operand, the corresponding data element in the destination operand is set to all 1s; otherwise, it is set to all 0s. The source operand can be an MMX technology register or a 64-bit memory location, or it can be an XMM register or a 128-bit memory location. The destination operand can be an MMX technology register or an XMM register.

The PCMPGTB instruction compares the corresponding signed byte integers in the destination and source operands; the PCMPGTW instruction compares the corresponding signed word integers in the destination and source operands; and the PCMPGTD instruction compares the corresponding signed doubleword integers in the destination and source operands.

Operation

```
PCMPGTB instruction with 64-bit operands:
   IF DEST[7..0] > SRC[7..0]
        THEN DEST[7  0) ← FFH;
        ELSE DEST[7..0] ← 0;
   * Continue comparison of 2nd through 7th bytes in DEST and SRC *
   IF DEST[63..56] > SRC[63..56]
        THEN DEST[63..56] ← FFH;
        ELSE DEST[63..56] ← 0;

PCMPGTB instruction with 128-bit operands:
   IF DEST[7..0] > SRC[7..0]
        THEN DEST[7  0) ← FFH;
        ELSE DEST[7..0] ← 0;
```

```
    * Continue comparison of 2nd through 15th bytes in DEST and SRC *
    IF DEST[63..56] > SRC[63..56]
        THEN DEST[63..56] ← FFH;
        ELSE DEST[63..56] ← 0;

PCMPGTW instruction with 64-bit operands:
    IF DEST[15..0] > SRC[15..0]
        THEN DEST[15..0] ← FFFFH;
        ELSE DEST[15..0] ← 0;
    * Continue comparison of 2nd and 3rd words in DEST and SRC *
    IF DEST[63..48] > SRC[63..48]
        THEN DEST[63..48] ← FFFFH;
        ELSE DEST[63..48] ← 0;

PCMPGTW instruction with 128-bit operands:
    IF DEST[15..0] > SRC[15..0]
        THEN DEST[15..0] ← FFFFH;
        ELSE DEST[15..0] ← 0;
    * Continue comparison of 2nd through 7th words in DEST and SRC *
    IF DEST[63..48] > SRC[63..48]
        THEN DEST[63..48] ← FFFFH;
        ELSE DEST[63..48] ← 0;

PCMPGTD instruction with 64-bit operands:
    IF DEST[31..0] > SRC[31..0]
        THEN DEST[31..0] ← FFFFFFFFH;
        ELSE DEST[31..0] ← 0;
    IF DEST[63..32] > SRC[63..32]
        THEN DEST[63..32] ← FFFFFFFFH;
        ELSE DEST[63..32] ← 0;

PCMPGTD instruction with 128-bit operands:
    IF DEST[31..0] > SRC[31..0]
        THEN DEST[31..0] ← FFFFFFFFH;
        ELSE DEST[31..0] ← 0;
    * Continue comparison of 2nd and 3rd doublewords in DEST and SRC *
    IF DEST[63..32] > SRC[63..32]
        THEN DEST[63..32] ← FFFFFFFFH;
        ELSE DEST[63..32] ← 0;
```

Intel C/C++ Compiler Intrinsic Equivalents

PCMPGTB	__m64 _mm_cmpgt_pi8 (__m64 m1, __m64 m2)
PCMPGTW	__m64 _mm_pcmpgt_pi16 (__m64 m1, __m64 m2)
DCMPGTD	__m64 _mm_pcmpgt_pi32 (__m64 m1, __m64 m2)
PCMPGTB	__m128i _mm_cmpgt_epi8 (__m128i a, __m128i b

PCMPGTW	__m128i _mm_cmpgt_epi16 (__m128i a, __m128i b
DCMPGTD	__m128i _mm_cmpgt_epi32 (__m128i a, __m128i b

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.
	(128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If EM in CR0 is set.
	128-bit operations will generate #UD only if OSFXSR in CR4 is 0. Execution of 128-bit instructions on a non-SSE2 capable processor (one that is MMX technology capable) will result in the instruction operating on the mm registers, not #UD.
#NM	If TS in CR0 is set.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#GP(0)	(128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
	If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set.
	128-bit operations will generate #UD only if OSFXSR in CR4 is 0. Execution of 128-bit instructions on a non-SSE2 capable processor (one that is MMX technology capable) will result in the instruction operating on the mm registers, not #UD.
#NM	If TS in CR0 is set.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made.

Numeric Exceptions

None.

PEXTRW—Extract Word

Opcode	Instruction	Description
0F C5 /r ib	PEXTRW *r32*, *mm*, *imm8*	Extract the word specified by *imm8* from *mm* and move it to *r32*.
66 0F C5 /r ib	PEXTRW *r32*, *xmm*, *imm8*	Extract the word specified by *imm8* from *xmm* and move it to a *r32*.

Description

Copies the word in the source operand (second operand) specified by the count operand (third operand) to the destination operand (first operand). The source operand can be an MMX technology register or an XMM register. The destination operand is the low word of a general-purpose register. The count operand is an 8-bit immediate. When specifying a word location in an MMX technology register, the 2 least-significant bits of the count operand specify the location; for an XMM register, the 3 least-significant bits specify the location. The high word of the destination operand is cleared (set to all 0s).

Operation

```
PEXTRW instruction with 64-bit source operand:
   SEL ← COUNT AND 3H;
   TEMP ← (SRC >> (SEL ∗ 16)) AND FFFFH;
   r32[15-0] ← TEMP[15-0];
   r32[31-16] ← 0000H;

PEXTRW instruction with 128-bit source operand:
   SEL ← COUNT AND 7H;
   TEMP ← (SRC >> (SEL ∗ 16)) AND FFFFH;
   r32[15-0] ← TEMP[15-0];
   r32[31-16] ← 0000H;
```

Intel C/C++ Compiler Intrinsic Equivalent

PEXTRW	int_mm_extract_pi16 (__m64 a, int n)
PEXTRW	int _mm_extract_epi16 (__m128i a, int imm)

Flags Affected

None.

Protected 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.
#UD	If EM in CR0 is set.
	(128-bit operations only) If OSFXSR in CR4 is 0.
	(128-bit operations only) If CPUID feature flag SSE2 is 0.
#NM	If TS in CR0 is set.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#GP(0)	If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set.
	(128-bit operations only) If OSFXSR in CR4 is 0.
	(128-bit operations only) If CPUID feature flag SSE2 is 0.
#NM	If TS in CR0 is set.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made.

Numeric Exceptions

None.

PINSRW—Insert Word

Opcode	Instruction	Description
0F C4 /r ib	PINSRW *mm*, *r32/m16, imm8*	Insert the low word from *r32* or from *m16* into *mm* at the word position specified by *imm8.*
66 0F C4 /r ib	PINSRW xmm, *r32/m16*, imm8	Move the low word of *r32* or from *m16* into *xmm* at the word position specified by *imm8*.

Description

Copies a word from the source operand (second operand) and inserts it in the destination operand (first operand) at the location specified with the count operand (third operand). (The other words in the destination register are left untouched.) The source operand can be a general-purpose register or a 16-bit memory location. (When the source operand is a general-purpose register, the low word of the register is copied.) The destination operand can be an MMX technology register or an XMM register. The count operand is an 8-bit immediate. When specifying a word location in an MMX technology register, the 2 least-significant bits of the count operand specify the location; for an XMM register, the 3 least-significant bits specify the location.

Operation

```
PINSRW instruction with 64-bit source operand:
   SEL ← COUNT AND 3H;
       CASE (determine word position) OF
           SEL ← 0:     MASK ← 000000000000FFFFH;
           SEL ← 1:     MASK ← 00000000FFFF0000H;
           SEL ← 2:     MASK ← 0000FFFF00000000H;
           SEL ← 3:     MASK ← FFFF000000000000H;
   DEST ← (DEST AND NOT MASK) OR (((SRC << (SEL ∗ 16)) AND MASK);

PINSRW instruction with 128-bit source operand:
   SEL ← COUNT AND 7H;
       CASE (determine word position) OF
           SEL ← 0:     MASK ← 0000000000000000000000000000FFFFH;
           SEL ← 1:     MASK ← 000000000000000000000000FFFF0000H;
           SEL ← 2:     MASK ← 00000000000000000000FFFF00000000H;
           SEL ← 3:     MASK ← 0000000000000000FFFF000000000000H;
           SEL ← 4:     MASK ← 000000000000FFFF0000000000000000H;
           SEL ← 5:     MASK ← 00000000FFFF00000000000000000000H;
           SEL ← 6:     MASK ← 0000FFFF000000000000000000000000H;
           SEL ← 7:     MASK ← FFFF0000000000000000000000000000H;
   DEST ← (DEST AND NOT MASK) OR (((SRC << (SEL ∗ 16)) AND MASK);
```

Intel C/C++ Compiler Intrinsic Equivalent

PINSRW	__m64 _mm_insert_pi16 (__m64 a, int d, int n)
PINSRW	__m128i _mm_insert_epi16 (__m128i a, int b, int imm)

Flags Affected

None.

Protected 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.
#UD	If EM in CR0 is set.
	(128-bit operations only) If OSFXSR in CR4 is 0.
	(128-bit operations only) If CPUID feature flag SSE2 is 0.
#NM	If TS in CR0 is set.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#GP(0)	If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set.
	(128-bit operations only) If OSFXSR in CR4 is 0.
	(128-bit operations only) If CPUID feature flag SSE2 is 0.
#NM	If TS in CR0 is set.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made.

Numeric Exceptions

None.

PMADDWD—Multiply and Add Packed Integers

Opcode	Instruction	Description
0F F5 /r	PMADDWD *mm, mm/m64*	Multiply the packed words in *mm* by the packed words in *mm/m64*, add adjacent doubleword results, and store in *mm*.
66 0F F5 /r	PMADDWD *xmm1*, *xmm2/m128*	Multiply the packed word integers in *xmm1* by the packed word integers in *xmm2/m128*, add adjacent doubleword results, and store in *xmm1*.

Description

Multiplies the individual signed words of the destination operand (first operand) by the corresponding signed words of the source operand (second operand), producing temporary signed, doubleword results. The adjacent doubleword results are then summed and stored in the destination operand. For example, the corresponding low-order words (15-0) and (31-16) in the source and destination operands are multiplied by one another and the doubleword results are added together and stored in the low doubleword of the destination register (31-0). The same operation is performed on the other pairs of adjacent words. (Figure 4-2 shows this operation when using 64-bit operands.) The source operand can be an MMX technology register or a 64-bit memory location, or it can be an XMM register or a 128-bit memory location. The destination operand can be an MMX technology register or an XMM register.

The PMADDWD instruction wraps around only in one situation: when the 2 pairs of words being operated on in a group are all 8000H. In this case, the result wraps around to 80000000H.

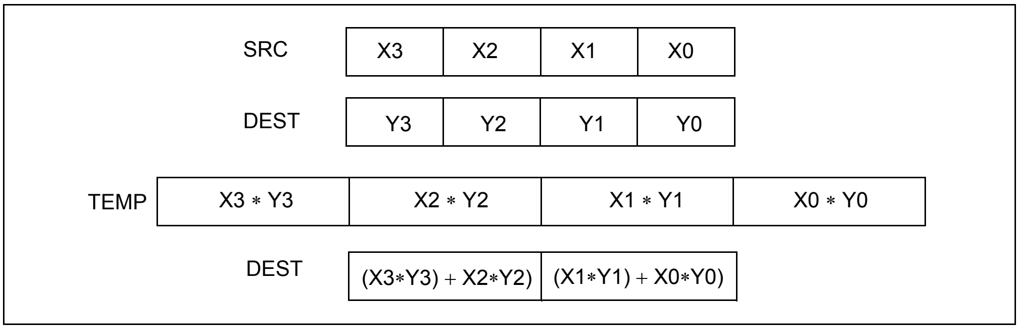


Figure 4-2. PMADDWD Execution Model Using 64-bit Operands

Operation

```
PMADDWD instruction with 64-bit operands:
    DEST[31..0] ← (DEST[15..0] ∗ SRC[15..0]) + (DEST[31..16] ∗ SRC[31..16]);
    DEST[63..32] ← (DEST[47..32] ∗ SRC[47..32]) + (DEST[63..48] ∗ SRC[63..48]);
```

```
PMADDWD instruction with 128-bit operands:
    DEST[31..0] ← (DEST[15..0] ∗ SRC[15..0]) + (DEST[31..16] ∗ SRC[31..16]);
    DEST[63..32] ← (DEST[47..32] ∗ SRC[47..32]) + (DEST[63..48] ∗ SRC[63..48]);
    DEST[95..64) ← (DEST[79..64) ∗ SRC[79..64)) + (DEST[95..80) ∗ SRC[95..80));
    DEST[127..96) ← (DEST[111..96) ∗ SRC[111..96)) + (DEST[127..112) ∗ SRC[127..112));
```

Intel C/C++ Compiler Intrinsic Equivalent

PMADDWD	__m64 _mm_madd_pi16(__m64 m1, __m64 m2)
PMADDWD	__m128i _mm_madd_epi16 (__m128i a, __m128i b)

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.
	(128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If EM in CR0 is set.
	128-bit operations will generate #UD only if OSFXSR in CR4 is 0. Execution of 128-bit instructions on a non-SSE2 capable processor (one that is MMX technology capable) will result in the instruction operating on the mm registers, not #UD.
#NM	If TS in CR0 is set.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#GP(0)	(128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
	If any part of the operand lies outside of the effective address space from 0 to FFFFH.

#UD	If EM in CR0 is set.
	128-bit operations will generate #UD only if OSFXSR in CR4 is 0. Execution of 128-bit instructions on a non-SSE2 capable processor (one that is MMX technology capable) will result in the instruction operating on the mm registers, not #UD.
#NM	If TS in CR0 is set.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made.

Numeric Exceptions

None.

PMAXSW—Maximum of Packed Signed Word Integers

Opcode	Instruction	Description
0F EE /r	PMAXSW *mm1, mm2/m64*	Compare signed word integers in *mm2/m64* and *mm1* and return maximum values.
66 0F EE /r	PMAXSW *xmm1*, *xmm2/m128*	Compare signed word integers in *xmm2/m128* and *xmm1* and return maximum values.

Description

Performs an SIMD compare of the packed signed word integers in the destination operand (first operand) and the source operand (second operand), and returns the maximum value for each pair of word integers to the destination operand. The source operand can be an MMX technology register or a 64-bit memory location, or it can be an XMM register or a 128-bit memory location. The destination operand can be an MMX technology register or an XMM register.

Operation

```
PMAXSW instruction for 64-bit operands:
    IF DEST[15-0] > SRC[15-0]) THEN
        (DEST[15-0] ← DEST[15-0];
    ELSE
        (DEST[15-0] ← SRC[15-0];
    FI
    * repeat operation for 2nd and 3rd words in source and destination operands *
    IF DEST[63-48] > SRC[63-48]) THEN
        (DEST[63-48] ← DEST[63-48];
    ELSE
        (DEST[63-48] ← SRC[63-48];
    FI
PMAXSW instruction for 128-bit operands:
    IF DEST[15-0] > SRC[15-0]) THEN
        (DEST[15-0] ← DEST[15-0];
    ELSE
        (DEST[15-0] ← SRC[15-0];
    FI
    * repeat operation for 2nd through 7th words in source and destination operands *
    IF DEST[127-112] > SRC[127-112]) THEN
        (DEST[127-112] ← DEST[127-112];
    ELSE
        (DEST[127-112] ← SRC[127-112];
    FI
```

Intel C/C++ Compiler Intrinsic Equivalent

PMAXSW	__m64 _mm_max_pi16(__m64 a, __m64 b)
PMAXSW	__m128i _mm_max_epi16 (__m128i a, __m128i b)

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.
	(128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If EM in CR0 is set.
	(128-bit operations only) If OSFXSR in CR4 is 0.
	(128-bit operations only) If CPUID feature flag SSE2 is 0.
#NM	If TS in CR0 is set.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#GP(0)	(128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
	If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set.
	(128-bit operations only) If OSFXSR in CR4 is 0.
	(128-bit operations only) If CPUID feature flag SSE2 is 0.
#NM	If TS in CR0 is set.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made.

Numeric Exceptions

None.

PMAXUB—Maximum of Packed Unsigned Byte Integers

Opcode	Instruction	Description
0F DE /r	PMAXUB *mm1, mm2/m64*	Compare unsigned byte integers in *mm2/m64* and *mm1* and returns maximum values.
66 0F DE /r	PMAXUB *xmm1*, *xmm2/m128*	Compare unsigned byte integers in *xmm2/m128* and *xmm1* and returns maximum values.

Description

Performs an SIMD compare of the packed unsigned byte integers in the destination operand (first operand) and the source operand (second operand), and returns the maximum value for each pair of byte integers to the destination operand. The source operand can be an MMX technology register or a 64-bit memory location, or it can be an XMM register or a 128-bit memory location. The destination operand can be an MMX technology register or an XMM register.

Operation

```
PMAXUB instruction for 64-bit operands:
   IF DEST[7-0] > SRC[17-0]) THEN
       (DEST[7-0] ← DEST[7-0];
   ELSE
       (DEST[7-0] ← SRC[7-0];
   FI
   * repeat operation for 2nd through 7th bytes in source and destination operands *
   IF DEST[63-56] > SRC[63-56]) THEN
       (DEST[63-56] ← DEST[63-56];
   ELSE
       (DEST[63-56] ← SRC[63-56];
   FI

PMAXUB instruction for 128-bit operands:
   IF DEST[7-0] > SRC[17-0]) THEN
       (DEST[7-0] ← DEST[7-0];
   ELSE
       (DEST[7-0] ← SRC[7-0];
   FI
   * repeat operation for 2nd through 15th bytes in source and destination operands *
   IF DEST[127-120] > SRC[127-120]) THEN
       (DEST[127-120] ← DEST[127-120];
   ELSE
       (DEST[127-120] ← SRC[127-120];
   FI
```

Intel C/C++ Compiler Intrinsic Equivalent

PMAXUB	__m64 _mm_max_pu8(__m64 a, __m64 b)
PMAXUB	__m128i _mm_max_epu8 (__m128i a, __m128i b)

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.
	(128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If EM in CR0 is set.
	(128-bit operations only) If OSFXSR in CR4 is 0.
	(128-bit operations only) If CPUID feature flag SSE2 is 0.
#NM	If TS in CR0 is set.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#GP(0)	(128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
	If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set.
	(128-bit operations only) If OSFXSR in CR4 is 0.
	(128-bit operations only) If CPUID feature flag SSE2 is 0.
#NM	If TS in CR0 is set.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made.

Numeric Exceptions

None.

PMINSW—Minimum of Packed Signed Word Integers

Opcode	Instruction	Description
0F EA /r	PMINSW *mm1, mm2/m64*	Compare signed word integers in *mm2/m64* and *mm1* and return minimum values.
66 0F EA /r	PMINSW *xmm1*, *xmm2/m128*	Compare signed word integers in *xmm2/m128* and *xmm1* and return minimum values.

Description

Performs an SIMD compare of the packed signed word integers in the destination operand (first operand) and the source operand (second operand), and returns the minimum value for each pair of word integers to the destination operand. The source operand can be an MMX technology register or a 64-bit memory location, or it can be an XMM register or a 128-bit memory location. The destination operand can be an MMX technology register or an XMM register.

Operation

```
PMINSW instruction for 64-bit operands:
    IF DEST[15-0] < SRC[15-0]) THEN
        (DEST[15-0] ← DEST[15-0];
    ELSE
        (DEST[15-0] ← SRC[15-0];
    FI
    * repeat operation for 2nd and 3rd words in source and destination operands *
    IF DEST[63-48] < SRC[63-48]) THEN
        (DEST[63-48] ← DEST[63-48];
    ELSE
        (DEST[63-48] ← SRC[63-48];
    FI
MINSW instruction for 128-bit operands:
    IF DEST[15-0] < SRC[15-0]) THEN
        (DEST[15-0] ← DEST[15-0];
    ELSE
        (DEST[15-0] ← SRC[15-0];
    FI
    * repeat operation for 2nd through 7th words in source and destination operands *
    IF DEST[127-112] < SRC/m64[127-112]) THEN
        (DEST[127-112] ← DEST[127-112];
    ELSE
        (DEST[127-112] ← SRC[127-112];
    FI
```

Intel C/C++ Compiler Intrinsic Equivalent

PMINSW	__m64 _mm_min_pi16 (__m64 a, __m64 b)
PMINSW	__m128i _mm_min_epi16 (__m128i a, __m128i b)

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.
	(128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If EM in CR0 is set.
	(128-bit operations only) If OSFXSR in CR4 is 0.
	(128-bit operations only) If CPUID feature flag SSE2 is 0.
#NM	If TS in CR0 is set.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#GP(0)	(128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
	If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set.
	(128-bit operations only) If OSFXSR in CR4 is 0.
	(128-bit operations only) If CPUID feature flag SSE2 is 0.
#NM	If TS in CR0 is set.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made.

Numeric Exceptions

None.

PMINUB—Minimum of Packed Unsigned Byte Integers

Opcode	Instruction	Description
0F DA /r	PMINUB *mm1, mm2/m64*	Compare unsigned byte integers in *mm2/m64* and *mm1* and return minimum values.
66 0F DA /r	PMINUB *xmm1*, *xmm2/m128*	Compare unsigned byte integers in *xmm2/m128* and *xmm1* and return minimum values.

Description

Performs an SIMD compare of the packed unsigned byte integers in the destination operand (first operand) and the source operand (second operand), and returns the minimum value for each pair of byte integers to the destination operand. The source operand can be an MMX technology register or a 64-bit memory location, or it can be an XMM register or a 128-bit memory location. The destination operand can be an MMX technology register or an XMM register.

Operation

```
PMINUB instruction for 64-bit operands:
    IF DEST[7-0] < SRC[17-0]) THEN
        (DEST[7-0] ← DEST[7-0];
    ELSE
        (DEST[7-0] ← SRC[7-0];
    FI
    * repeat operation for 2nd through 7th bytes in source and destination operands *
    IF DEST[63-56] < SRC[63-56]) THEN
        (DEST[63-56] ← DEST[63-56];
    ELSE
        (DEST[63-56] ← SRC[63-56];
    FI

PMINUB instruction for 128-bit operands:
    IF DEST[7-0] < SRC[17-0]) THEN
        (DEST[7-0] ← DEST[7-0];
    ELSE
        (DEST[7-0] ← SRC[7-0];
    FI
    * repeat operation for 2nd through 15th bytes in source and destination operands *
    IF DEST[127-120] < SRC[127-120]) THEN
        (DEST[127-120] ← DEST[127-120];
    ELSE
        (DEST[127-120] ← SRC[127-120];
    FI
```

Intel C/C++ Compiler Intrinsic Equivalent

PMINUB	__m64 _m_min_pu8 (__m64 a, __m64 b)
PMINUB	___m128i _mm_min_epu8 (__m128i a, __m128i b)

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.
	(128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If EM in CR0 is set.
	(128-bit operations only) If OSFXSR in CR4 is 0.
	(128-bit operations only) If CPUID feature flag SSE2 is 0.
#NM	If TS in CR0 is set.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#GP(0)	(128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
	If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set.
	(128-bit operations only) If OSFXSR in CR4 is 0.
	(128-bit operations only) If CPUID feature flag SSE2 is 0.
#NM	If TS in CR0 is set.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made.

Numeric Exceptions

None.

PMOVMSKB—Move Byte Mask

Opcode	Instruction	Description
0F D7 /r	PMOVMSKB *r32*, *mm*	Move a byte mask of *mm* to *r32*.
66 0F D7 /r	PMOVMSKB *r32*, *xmm*	Move a byte mask of *xmm* to *r32*.

Description

Creates a mask made up of the most significant bit of each byte of the source operand (second operand) and stores the result in the low byte or word of the destination operand (first operand). The source operand is an MMX technology register or an XMM register; the destination operand is a general-purpose register. When operating on 64-bit operands, the byte mask is 8 bits; when operating on 128-bit operands, the byte mask is 16-bits.

Operation

```
PMOVMSKB instruction with 64-bit source operand:
r32[0] ← SRC[7];
r32[1] ← SRC[15];
* repeat operation for bytes 2 through 6;
r32[7] ← SRC[63];
r32[31-8] ← 000000H;

PMOVMSKB instruction with 128-bit source operand:
r32[0] ← SRC[7];
r32[1] ← SRC[15];
* repeat operation for bytes 2 through 14;
r32[15] ← SRC[127];
r32[31-16] ← 0000H;
```

Intel C/C++ Compiler Intrinsic Equivalent

PMOVMSKB	int_mm_movemask_pi8(__m64 a)
PMOVMSKB	int _mm_movemask_epi8 (__m128i a)

Flags Affected

None.

Protected Mode Exceptions

#UD	If EM in CR0 is set.
	(128-bit operations only) If OSFXSR in CR4 is 0.
	(128-bit operations only) If CPUID feature flag SSE2 is 0.

#NM	If TS in CR0 is set.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.

Real-Address Mode Exceptions

Same exceptions as in Protected Mode

Virtual-8086 Mode Exceptions

Same exceptions as in Protected Mode

Numeric Exceptions

None.

PMULHUW—Multiply Packed Unsigned Integers and Store High Result

Opcode	Instruction	Description
0F E4 /r	PMULHUW *mm1*, *mm2/m64*	Multiply the packed unsigned word integers in *mm1* register and *mm2/m64*, and store the high 16 bits of the results in *mm1*.
66 0F E4 /r	PMULHUW *xmm1*, *xmm2/m128*	Multiply the packed unsigned word integers in *xmm1* and *xmm2/m128*, and store the high 16 bits of the results in *xmm1*.

Description

Performs an SIMD unsigned multiply of the packed unsigned word integers in the destination operand (first operand) and the source operand (second operand), and stores the high 16 bits of each 32-bit intermediate results in the destination operand. (Figure 4-3 shows this operation when using 64-bit operands.) The source operand can be an MMX technology register or a 64-bit memory location, or it can be an XMM register or a 128-bit memory location. The destination operand can be an MMX technology register or an XMM register.

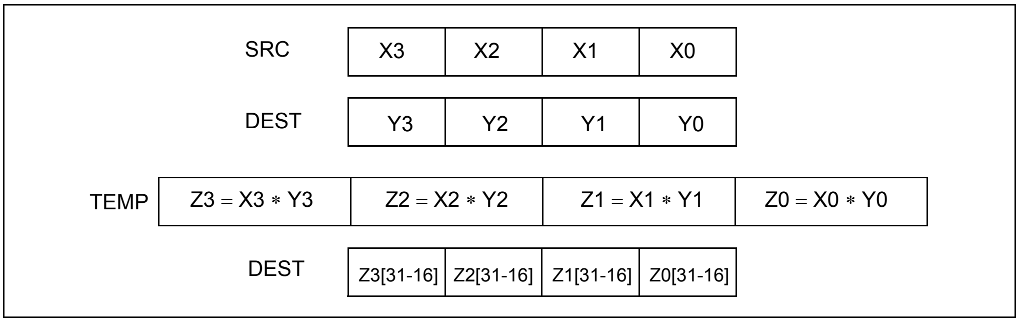


Figure 4-3. PMULHUW and PMULHW Instruction Operation Using 64-bit Operands

Operation

PMULHUW instruction with 64-bit operands:

```
    TEMP0[31-0] ←  DEST[15-0] ∗ SRC[15-0]; * Unsigned multiplication *
    TEMP1[31-0] ←  DEST[31-16] ∗ SRC[31-16];
    TEMP2[31-0] ←  DEST[47-32] ∗ SRC[47-32];
    TEMP3[31-0] ←  DEST[63-48] ∗ SRC[63-48];
    DEST[15-0] ←   TEMP0[31-16];
    DEST[31-16] ←  TEMP1[31-16];
    DEST[47-32] ←  TEMP2[31-16];
    DEST[63-48] ←  TEMP3[31-16];
```

PMULHUW instruction with 128-bit operands:

```
    TEMP0[31-0] ←  DEST[15-0] ∗ SRC[15-0]; * Unsigned multiplication *
    TEMP1[31-0] ←  DEST[31-16] ∗ SRC[31-16];
    TEMP2[31-0] ←  DEST[47-32] ∗ SRC[47-32];
    TEMP3[31-0] ←  DEST[63-48] ∗ SRC[63-48];
    TEMP4[31-0] ←  DEST[79-64] ∗ SRC[79-64];
    TEMP5[31-0] ←  DEST[95-80] ∗ SRC[95-80];
    TEMP6[31-0] ←  DEST[111-96] ∗ SRC[111-96];
    TEMP7[31-0] ←  DEST[127-112] ∗ SRC[127-112];
    DEST[15-0] ←    TEMP0[31-16];
    DEST[31-16] ←  TEMP1[31-16];
    DEST[47-32] ←  TEMP2[31-16];
    DEST[63-48] ←  TEMP3[31-16];
    DEST[79-64] ←  TEMP4[31-16];
    DEST[95-80] ←  TEMP5[31-16];
    DEST[111-96] ← TEMP6[31-16];
    DEST[127-112] ← TEMP7[31-16];
```

Intel C/C++ Compiler Intrinsic Equivalent

PMULHUW	__m64 _mm_mulhi_pu16(__m64 a, __m64 b)
PMULHUW	__m128i _mm_mulhi_epu16 (__m128i a, __m128i b)

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.
	(128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If EM in CR0 is set.
	(128-bit operations only) If OSFXSR in CR4 is 0.
	(128-bit operations only) If CPUID feature flag SSE2 is 0.
#NM	If TS in CR0 is set.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#GP(0)	(128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
	If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set.
	(128-bit operations only) If OSFXSR in CR4 is 0.
	(128-bit operations only) If CPUID feature flag SSE2 is 0.
#NM	If TS in CR0 is set.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made.

Numeric Exceptions

None.

PMULHW—Multiply Packed Signed Integers and Store High Result

Opcode	Instruction	Description
0F E5 /r	PMULHW *mm, mm/m64*	Multiply the packed signed word integers in *mm1* register and *mm2/m64*, and store the high 16 bits of the results in *mm1*.
66 0F E5 /r	PMULHW *xmm1*, *xmm2/m128*	Multiply the packed signed word integers in *xmm1* and *xmm2/m128*, and store the high 16 bits of the results in *xmm1*.

Description

Performs an SIMD signed multiply of the packed signed word integers in the destination operand (first operand) and the source operand (second operand), and stores the high 16 bits of each intermediate 32-bit result in the destination operand. (Figure 4-3 shows this operation when using 64-bit operands.) The source operand can be an MMX technology register or a 64-bit memory location, or it can be an XMM register or a 128-bit memory location. The destination operand can be an MMX technology register or an XMM register.

Operation

```
PMULHW instruction with 64-bit operands:
    TEMP0[31-0] ←  DEST[15-0] ∗ SRC[15-0]; * Signed multiplication *
    TEMP1[31-0] ←  DEST[31-16] ∗ SRC[31-16];
    TEMP2[31-0] ←  DEST[47-32] ∗ SRC[47-32];
    TEMP3[31-0] ←  DEST[63-48] ∗ SRC[63-48];
    DEST[15-0] ←   TEMP0[31-16];
    DEST[31-16] ←  TEMP1[31-16];
    DEST[47-32] ←  TEMP2[31-16];
    DEST[63-48] ←  TEMP3[31-16];

PMULHW instruction with 128-bit operands:
    TEMP0[31-0] ←  DEST[15-0] ∗ SRC[15-0]; * Signed multiplication *
    TEMP1[31-0] ←  DEST[31-16] ∗ SRC[31-16];
    TEMP2[31-0] ←  DEST[47-32] ∗ SRC[47-32];
    TEMP3[31-0] ←  DEST[63-48] ∗ SRC[63-48];
    TEMP4[31-0] ←  DEST[79-64] ∗ SRC[79-64];
    TEMP5[31-0] ←  DEST[95-80] ∗ SRC[95-80];
    TEMP6[31-0] ←  DEST[111-96] ∗ SRC[111-96];
    TEMP7[31-0] ←  DEST[127-112] ∗ SRC[127-112];
    DEST[15-0] ←   TEMP0[31-16];
    DEST[31-16] ←  TEMP1[31-16];
    DEST[47-32] ←  TEMP2[31-16];
    DEST[63-48] ←  TEMP3[31-16];
    DEST[79-64] ←  TEMP4[31-16];
    DEST[95-80] ←  TEMP5[31-16];
    DEST[111-96] ← TEMP6[31-16];
    DEST[127-112] ← TEMP7[31-16];
```

Intel C/C++ Compiler Intrinsic Equivalent

PMULHW	__m64 _mm_mulhi_pi16 (__m64 m1, __m64 m2)
PMULHW	__m128i _mm_mulhi_epi16 (__m128i a, __m128i b)

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. (128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If EM in CR0 is set. 128-bit operations will generate #UD only if OSFXSR in CR4 is 0. Execution of 128-bit instructions on a non-SSE2 capable processor (one that is MMX technology capable) will result in the instruction operating on the mm registers, not #UD.
#NM	If TS in CR0 is set.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#GP(0)	(128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment. If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set. 128-bit operations will generate #UD only if OSFXSR in CR4 is 0. Execution of 128-bit instructions on a non-SSE2 capable processor (one that is MMX technology capable) will result in the instruction operating on the mm registers, not #UD.

#NM	If TS in CR0 is set.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made.

Numeric Exceptions

None.

PMULLW—Multiply Packed Signed Integers and Store Low Result

Opcode	Instruction	Description
0F D5 /r	PMULLW *mm, mm/m64*	Multiply the packed signed word integers in *mm1* register and *mm2/m64*, and store the low 16 bits of the results in *mm1*.
66 0F D5 /r	PMULLW *xmm1*, *xmm2/m128*	Multiply the packed signed word integers in *xmm1* and *xmm2/m128*, and store the low 16 bits of the results in *xmm1*.

Description

Performs an SIMD signed multiply of the packed signed word integers in the destination operand (first operand) and the source operand (second operand), and stores the low 16 bits of each intermediate 32-bit result in the destination operand. (Figure 4-3 shows this operation when using 64-bit operands.) The source operand can be an MMX technology register or a 64-bit memory location, or it can be an XMM register or a 128-bit memory location. The destination operand can be an MMX technology register or an XMM register.

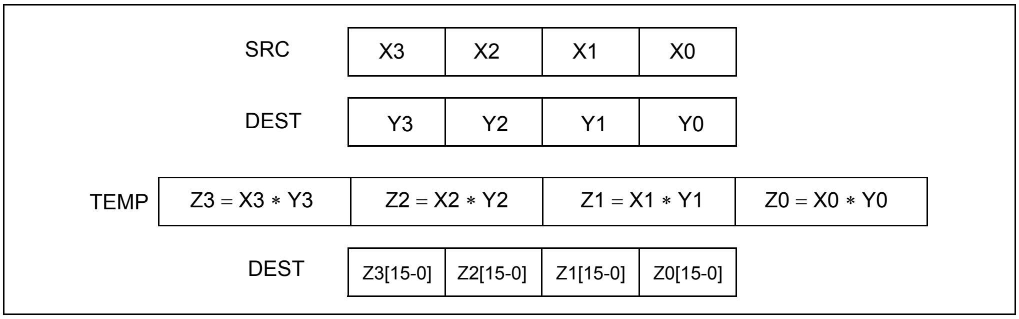


Figure 4-4. PMULLU Instruction Operation Using 64-bit Operands

Operation

```
PMULLW instruction with 64-bit operands:
    TEMP0[31-0] ←  DEST[15-0] ∗ SRC[15-0]; * Signed multiplication *
    TEMP1[31-0] ←  DEST[31-16] ∗ SRC[31-16];
    TEMP2[31-0] ←  DEST[47-32] ∗ SRC[47-32];
    TEMP3[31-0] ←  DEST[63-48] ∗ SRC[63-48];
    DEST[15-0] ←   TEMP0[15-0];
    DEST[31-16] ←  TEMP1[15-0];
    DEST[47-32] ←  TEMP2[15-0];
    DEST[63-48] ←  TEMP3[15-0];

PMULLW instruction with 64-bit operands:
    TEMP0[31-0] ←  DEST[15-0] ∗ SRC[15-0]; * Signed multiplication *
    TEMP1[31-0] ←  DEST[31-16] ∗ SRC[31-16];
    TEMP2[31-0] ←  DEST[47-32] ∗ SRC[47-32];
    TEMP3[31-0] ←  DEST[63-48] ∗ SRC[63-48];
    TEMP4[31-0] ←  DEST[79-64] ∗ SRC[79-64];
    TEMP5[31-0] ←  DEST[95-80] ∗ SRC[95-80];
    TEMP6[31-0] ←  DEST[111-96] ∗ SRC[111-96];
    TEMP7[31-0] ←  DEST[127-112] ∗ SRC[127-112];
    DEST[15-0] ←   TEMP0[15-0];
    DEST[31-16] ←  TEMP1[15-0];
    DEST[47-32] ←  TEMP2[15-0];
    DEST[63-48] ←  TEMP3[15-0];
    DEST[79-64] ←  TEMP4[15-0];
    DEST[95-80] ←  TEMP5[15-0];
    DEST[111-96] ← TEMP6[15-0];
    DEST[127-112] ← TEMP7[15-0];
```

Intel C/C++ Compiler Intrinsic Equivalent

PMULLW	__m64 _mm_mullo_pi16(__m64 m1, __m64 m2)
PMULLW	__m128i _mm_mullo_epi16 (__m128i a, __m128i b)

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.
	(128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If EM in CR0 is set.
	128-bit operations will generate #UD only if OSFXSR in CR4 is 0. Execution of 128-bit instructions on a non-SSE2 capable processor (one that is MMX technology capable) will result in the instruction operating on the mm registers, not #UD.
#NM	If TS in CR0 is set.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#GP(0)	(128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
	If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set.
	128-bit operations will generate #UD only if OSFXSR in CR4 is 0. Execution of 128-bit instructions on a non-SSE2 capable processor (one that is MMX technology capable) will result in the instruction operating on the mm registers, not #UD.
#NM	If TS in CR0 is set.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made.

Numeric Exceptions

None.

PMULUDQ—Multiply Packed Unsigned Doubleword Integers

Opcode	Instruction	Description
0F F4 /r	PMULUDQ *mm1*, *mm2/m64*	Multiply unsigned doubleword integer in *mm1* by unsigned doubleword integer in *mm2/m64*, and store the quadword result in *mm1*.
66 0F F4 /r	PMULUDQ *xmm1*, *xmm2/m128*	Multiply packed unsigned doubleword integers in *xmm1* by packed unsigned doubleword integers in *xmm2/m128*, and store the quadword results in *xmm1*.

Description

Multiplies the first operand (destination operand) by the second operand (source operand) and stores the result in the destination operand. The source operand can be an unsigned doubleword integer stored in the low doubleword of an MMX technology register or a 64-bit memory location, or it can be two packed unsigned doubleword integers stored in the first (low) and third doublewords of an XMM register or an 128-bit memory location. The destination operand can be an unsigned doubleword integer stored in the low doubleword an MMX technology register or two packed doubleword integers stored in the first and third doublewords of an XMM register. The result is an unsigned quadword integer stored in the destination an MMX technology register or two packed unsigned quadword integers stored in an XMM register. When a quadword result is too large to be represented in 64 bits (overflow), the result is wrapped around and the low 64 bits are written to the destination element (that is, the carry is ignored).

For 64-bit memory operands, 64 bits are fetched from memory, but only the low doubleword is used in the computation; for 128-bit memory operands, 128 bits are fetched from memory, but only the first and third doublewords are used in the computation.

Operation

```
PMULUDQ instruction with 64-Bit operands:
    DEST[63-0] ← DEST[31-0] ∗ SRC[31-0];

PMULUDQ instruction with 128-Bit operands:
    DEST[63-0] ← DEST[31-0] ∗ SRC[31-0];
    DEST[127-64] ← DEST[95-64] ∗ SRC[95-64];
```

Intel C/C++ Compiler Intrinsic Equivalent

PMULUDQ	__m64 _mm_mul_su32 (__m64 a, __m64 b)
PMULUDQ	__m128i _mm_mul_epu32 (__m128i a, __m128i b)

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.
	(128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If EM in CR0 is set.
	(128-bit operations only) If OSFXSR in CR4 is 0.
	(128-bit operations only) If CPUID feature flag SSE2 is 0.
#NM	If TS in CR0 is set.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#GP(0)	(128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
	If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set.
	(128-bit operations only) If OSFXSR in CR4 is 0.
	(128-bit operations only) If CPUID feature flag SSE2 is 0.
#NM	If TS in CR0 is set.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made.

POP—Pop a Value from the Stack

Opcode	Instruction	Description
8F /0	POP r/*m16*	Pop top of stack into *m16*; increment stack pointer.
8F /0	POP r/*m32*	Pop top of stack into *m32*; increment stack pointer.
58+ *rw*	POP *r16*	Pop top of stack into *r16*; increment stack pointer.
58+ *rd*	POP *r32*	Pop top of stack into *r32*; increment stack pointer.
1F	POP DS	Pop top of stack into DS; increment stack pointer.
07	POP ES	Pop top of stack into ES; increment stack pointer.
17	POP SS	Pop top of stack into SS; increment stack pointer.
0F A1	POP FS	Pop top of stack into FS; increment stack pointer.
0F A9	POP GS	Pop top of stack into GS; increment stack pointer.

Description

Loads the value from the top of the stack to the location specified with the destination operand and then increments the stack pointer. The destination operand can be a general-purpose register, memory location, or segment register.

The address-size attribute of the stack segment determines the stack pointer size (16 bits or 32 bits—the source address size), and the operand-size attribute of the current code segment determines the amount the stack pointer is incremented (2 bytes or 4 bytes). For example, if these address- and operand-size attributes are 32, the 32-bit ESP register (stack pointer) is incremented by 4 and, if they are 16, the 16-bit SP register is incremented by 2. (The B flag in the stack segment's segment descriptor determines the stack's address-size attribute, and the D flag in the current code segment's segment descriptor, along with prefixes, determines the operand-size attribute and also the address-size attribute of the destination operand.)

If the destination operand is one of the segment registers DS, ES, FS, GS, or SS, the value loaded into the register must be a valid segment selector. In protected mode, popping a segment selector into a segment register automatically causes the descriptor information associated with that segment selector to be loaded into the hidden (shadow) part of the segment register and causes the selector and the descriptor information to be validated (see the "Operation" section below).

A null value (0000-0003) may be popped into the DS, ES, FS, or GS register without causing a general protection fault. However, any subsequent attempt to reference a segment whose corresponding segment register is loaded with a null value causes a general protection exception (#GP). In this situation, no memory reference occurs and the saved value of the segment register is null.

The POP instruction cannot pop a value into the CS register. To load the CS register from the stack, use the RET instruction.

If the ESP register is used as a base register for addressing a destination operand in memory, the POP instruction computes the effective address of the operand after it increments the ESP register. For the case of a 16-bit stack where ESP wraps to 0h as a result of the POP instruction, the resulting location of the memory write is processor-family-specific.

The POP ESP instruction increments the stack pointer (ESP) before data at the old top of stack is written into the destination.

A POP SS instruction inhibits all interrupts, including the NMI interrupt, until after execution of the next instruction. This action allows sequential execution of POP SS and MOV ESP, EBP instructions without the danger of having an invalid stack during an interrupt[1]. However, use of the LSS instruction is the preferred method of loading the SS and ESP registers.

Operation

```
IF StackAddrSize = 32
    THEN
        IF OperandSize = 32
            THEN
                DEST ← SS:ESP; (* copy a doubleword *)
                ESP ← ESP + 4;
            ELSE (* OperandSize = 16*)
                DEST ← SS:ESP; (* copy a word *)
            ESP ← ESP + 2;
        FI;
    ELSE (* StackAddrSize = 16* )
        IF OperandSize = 16
            THEN
                DEST ← SS:SP; (* copy a word *)
                SP ← SP + 2;
            ELSE (* OperandSize = 32 *)
                DEST ← SS:SP; (* copy a doubleword *)
                SP ← SP + 4;
        FI;
FI;
```

Loading a segment register while in protected mode results in special actions, as described in the following listing. These checks are performed on the segment selector and the segment descriptor it points to.

```
IF SS is loaded;
    THEN
        IF segment selector is null
            THEN #GP(0);
        FI;
        IF segment selector index is outside descriptor table limits
```

1. Note that in a sequence of instructions that individually delay interrupts past the following instruction, only the first instruction in the sequence is guaranteed to delay the interrupt, but subsequent interrupt-delaying instructions may not delay the interrupt. Thus, in the following instruction sequence:

```
STI
POP SS
POP ESP
```

interrupts may be recognized before the POP ESP executes, because STI also delays interrupts for one instruction.

```
                OR segment selector's RPL ≠ CPL
                OR segment is not a writable data segment
                OR DPL ≠ CPL
                    THEN #GP(selector);
            FI;
            IF segment not marked present
                THEN #SS(selector);
        ELSE
            SS ← segment selector;
            SS ← segment descriptor;
        FI;
FI;
IF DS, ES, FS, or GS is loaded with non-null selector;
THEN
    IF segment selector index is outside descriptor table limits
        OR segment is not a data or readable code segment
        OR ((segment is a data or nonconforming code segment)
            AND (both RPL and CPL > DPL))
                THEN #GP(selector);
        IF segment not marked present
            THEN #NP(selector);
    ELSE
        SegmentRegister ← segment selector;
        SegmentRegister ← segment descriptor;
    FI;
FI;
IF DS, ES, FS, or GS is loaded with a null selector;
    THEN
        SegmentRegister ← segment selector;
        SegmentRegister ← segment descriptor;
FI;
```

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If attempt is made to load SS register with null segment selector.
	If the destination 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 is used to access memory and it contains a null segment selector.

#GP(selector)	If segment selector index is outside descriptor table limits.
	If the SS register is being loaded and the segment selector's RPL and the segment descriptor's DPL are not equal to the CPL.
	If the SS register is being loaded and the segment pointed to is a non-writable data segment.
	If the DS, ES, FS, or GS register is being loaded and the segment pointed to is not a data or readable code segment.
	If the DS, ES, FS, or GS register is being loaded and the segment pointed to is a data or nonconforming code segment, but both the RPL and the CPL are greater than the DPL.
#SS(0)	If the current top of stack is not within the stack segment.
	If a memory operand effective address is outside the SS segment limit.
#SS(selector)	If the SS register is being loaded and the segment pointed to is marked not present.
#NP	If the DS, ES, FS, or GS register is being loaded and the segment pointed to is marked not present.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory reference is made while the current privilege level is 3 and alignment checking is enabled.

Real-Address Mode Exceptions

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

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory reference is made while alignment checking is enabled.

POPA/POPAD—Pop All General-Purpose Registers

Opcode	Instruction	Description
61	POPA	Pop DI, SI, BP, BX, DX, CX, and AX.
61	POPAD	Pop EDI, ESI, EBP, EBX, EDX, ECX, and EAX.

Description

Pops doublewords (POPAD) or words (POPA) from the stack into the general-purpose registers. The registers are loaded in the following order: EDI, ESI, EBP, EBX, EDX, ECX, and EAX (if the operand-size attribute is 32) and DI, SI, BP, BX, DX, CX, and AX (if the operand-size attribute is 16). (These instructions reverse the operation of the PUSHA/PUSHAD instructions.) The value on the stack for the ESP or SP register is ignored. Instead, the ESP or SP register is incremented after each register is loaded.

The POPA (pop all) and POPAD (pop all double) mnemonics reference the same opcode. The POPA instruction is intended for use when the operand-size attribute is 16 and the POPAD instruction for when the operand-size attribute is 32. Some assemblers may force the operand size to 16 when POPA is used and to 32 when POPAD is used (using the operand-size override prefix [66H] if necessary). Others may treat these mnemonics as synonyms (POPA/POPAD) and use the current setting of the operand-size attribute to determine the size of values to be popped from the stack, regardless of the mnemonic used. (The D flag in the current code segment's segment descriptor determines the operand-size attribute.)

Operation

```
IF OperandSize = 32 (* instruction = POPAD *)
THEN
    EDI ← Pop();
    ESI ← Pop();
    EBP ← Pop();
    increment ESP by 4 (* skip next 4 bytes of stack *)
    EBX ← Pop();
    EDX ← Pop();
    ECX ← Pop();
    EAX ← Pop();
ELSE (* OperandSize = 16, instruction = POPA *)
    DI ← Pop();
    SI ← Pop();
    BP ← Pop();
    increment ESP by 2 (* skip next 2 bytes of stack *)
    BX ← Pop();
    DX ← Pop();
    CX ← Pop();
    AX ← Pop();
FI;
```

Flags Affected

None.

Protected Mode Exceptions

#SS(0)	If the starting or ending stack address is not within the stack segment.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory reference is made while the current privilege level is 3 and alignment checking is enabled.

Real-Address Mode Exceptions

#SS	If the starting or ending stack address is not within the stack segment.

Virtual-8086 Mode Exceptions

#SS(0)	If the starting or ending stack address is not within the stack segment.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory reference is made while alignment checking is enabled.

POPF/POPFD—Pop Stack into EFLAGS Register

Opcode	Instruction	Description
9D	POPF	Pop top of stack into lower 16 bits of EFLAGS.
9D	POPFD	Pop top of stack into EFLAGS.

Description

Pops a doubleword (POPFD) from the top of the stack (if the current operand-size attribute is 32) and stores the value in the EFLAGS register, or pops a word from the top of the stack (if the operand-size attribute is 16) and stores it in the lower 16 bits of the EFLAGS register (that is, the FLAGS register). These instructions reverse the operation of the PUSHF/PUSHFD instructions.

The POPF (pop flags) and POPFD (pop flags double) mnemonics reference the same opcode. The POPF instruction is intended for use when the operand-size attribute is 16 and the POPFD instruction for when the operand-size attribute is 32. Some assemblers may force the operand size to 16 when POPF is used and to 32 when POPFD is used. Others may treat these mnemonics as synonyms (POPF/POPFD) and use the current setting of the operand-size attribute to determine the size of values to be popped from the stack, regardless of the mnemonic used.

The effect of the POPF/POPFD instructions on the EFLAGS register changes slightly, depending on the mode of operation of the processor. When the processor is operating in protected mode at privilege level 0 (or in real-address mode, which is equivalent to privilege level 0), all the non-reserved flags in the EFLAGS register except the VIP, VIF, and VM flags can be modified. The VIP and VIF flags are cleared, and the VM flag is unaffected.

When operating in protected mode, with a privilege level greater than 0, but less than or equal to IOPL, all the flags can be modified except the IOPL field and the VIP, VIF, and VM flags. Here, the IOPL flags are unaffected, the VIP and VIF flags are cleared, and the VM flag is unaffected. The interrupt flag (IF) is altered only when executing at a level at least as privileged as the IOPL. If a POPF/POPFD instruction is executed with insufficient privilege, an exception does not occur, but the privileged bits do not change.

When operating in virtual-8086 mode, the I/O privilege level (IOPL) must be equal to 3 to use POPF/POPFD instructions and the VM, RF, IOPL, VIP, and VIF flags are unaffected. If the IOPL is less than 3, the POPF/POPFD instructions cause a general-protection exception (#GP).

See the section titled “EFLAGS Register” in Chapter 3 of the *IA-32 Intel Architecture Software Developer’s Manual, Volume 1*, for information about the EFLAGS registers.

Operation

```
IF VM=0 (* Not in Virtual-8086 Mode *)
    THEN IF CPL=0
        THEN
            IF OperandSize = 32;
                THEN
```

```
                    EFLAGS ← Pop();
                    (* All non-reserved flags except VIP, VIF, and VM can be modified; *)
                    (* VIP and VIF are cleared; VM is unaffected*)
                ELSE (* OperandSize = 16 *)
                    EFLAGS[15:0] ← Pop(); (* All non-reserved flags can be modified; *)
            FI;
        ELSE (* CPL > 0 *)
            IF OperandSize = 32;
                THEN
                    EFLAGS ← Pop()
                    (* All non-reserved bits except IOPL, VIP, and VIF can be modified; *)
                    (* IOPL is unaffected; VIP and VIF are cleared; VM is unaffected *)
                ELSE (* OperandSize = 16 *)
                    EFLAGS[15:0] ← Pop();
                    (* All non-reserved bits except IOPL can be modified *)
                    (* IOPL is unaffected *)
            FI;
    FI;
    ELSE  (* In Virtual-8086 Mode *)
        IF IOPL=3
            THEN IF OperandSize=32
                THEN
                    EFLAGS ← Pop()
                    (* All non-reserved bits except VM, RF, IOPL, VIP, and VIF *)
                    (* can be modified; VM, RF, IOPL, VIP, and VIF are unaffected *)
                ELSE
                    EFLAGS[15:0] ← Pop()
                    (* All non-reserved bits except IOPL can be modified *)
                    (* IOPL is unaffected *)
            FI;
            ELSE (* IOPL < 3 *)
                #GP(0);  (* trap to virtual-8086 monitor *)
        FI;
    FI;
FI;
```

Flags Affected

All flags except the reserved bits and the VM bit.

Protected Mode Exceptions

#SS(0)	If the top of stack is not within the stack segment.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory reference is made while the current privilege level is 3 and alignment checking is enabled.

Real-Address Mode Exceptions

#SS	If the top of stack is not within the stack segment.

Virtual-8086 Mode Exceptions

#GP(0)	If the I/O privilege level is less than 3.
	If an attempt is made to execute the POPF/POPFD instruction with an operand-size override prefix.
#SS(0)	If the top of stack is not within the stack segment.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory reference is made while alignment checking is enabled.

POR—Bitwise Logical OR

Opcode	Instruction	Description
0F EB /r	POR *mm, mm/m64*	Bitwise OR of *mm/m64* and *mm*.
66 0F EB /r	POR *xmm1*, *xmm2/m128*	Bitwise OR of *xmm2/m128* and *xmm1*.

Description

Performs a bitwise logical OR operation on the source operand (second operand) and the destination operand (first operand) and stores the result in the destination operand. The source operand can be an MMX technology register or a 64-bit memory location or it can be an XMM register or a 128-bit memory location. The destination operand can be an MMX technology register or an XMM register. Each bit of the result is set to 1 if either or both of the corresponding bits of the first and second operands are 1; otherwise, it is set to 0.

Operation

```
DEST ← DEST OR SRC;
```

Intel C/C++ Compiler Intrinsic Equivalent

POR	__m64 _mm_or_si64(__m64 m1, __m64 m2)
POR	__m128i _mm_or_si128(__m128i m1, __m128i m2)

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.
	(128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If EM in CR0 is set.
	128-bit operations will generate #UD only if OSFXSR in CR4 is 0. Execution of 128-bit instructions on a non-SSE2 capable processor (one that is MMX technology capable) will result in the instruction operating on the mm registers, not #UD.
#NM	If TS in CR0 is set.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.

#PF(fault-code)	If a page fault occurs.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#GP(0)	(128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
	If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set.
	128-bit operations will generate #UD only if OSFXSR in CR4 is 0. Execution of 128-bit instructions on a non-SSE2 capable processor (one that is MMX technology capable) will result in the instruction operating on the mm registers, not #UD.
#NM	If TS in CR0 is set.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made.

Numeric Exceptions

None.

PREFETCH*h*—Prefetch Data Into Caches

Opcode	Instruction	Description
0F 18 /1	PREFETCHT0 *m8*	Move data from *m8* closer to the processor using T0 hint.
0F 18 /2	PREFETCHT1 *m8*	Move data from *m8* closer to the processor using T1 hint.
0F 18 /3	PREFETCHT2 *m8*	Move data from *m8* closer to the processor using T2 hint.
0F 18 /0	PREFETCHNTA *m8*	Move data from *m8* closer to the processor using NTA hint.

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 a locality hint:

- T0 (temporal data)—prefetch data into all levels of the cache hierarchy.
  - Pentium III processor—1st- or 2nd-level cache.
  - Pentium 4 and Intel Xeon processors—2nd-level cache.
- T1 (temporal data with respect to first level cache)—prefetch data into level 2 cache and higher.
  - Pentium III processor—2nd-level cache.
  - Pentium 4 and Intel Xeon processors—2nd-level cache.
- T2 (temporal data with respect to second level cache)—prefetch data into level 2 cache and higher.
  - Pentium III processor—2nd-level cache.
  - Pentium 4 and Intel Xeon processors—2nd-level cache.
- NTA (non-temporal data with respect to all cache levels)—prefetch data into non-temporal cache structure and into a location close to the processor, minimizing cache pollution.
  - Pentium III processor—1st-level cache
  - Pentium 4 and Intel Xeon processors—2nd-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 PREFETCH*h* 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.

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 PREFETCH*h* 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 PREFETCH*h* instruction is not ordered with respect to the fence instructions (MFENCE, SFENCE, and LFENCE) or locked memory references. A PREFETCH*h* instruction is also unordered with respect to CLFLUSH instructions, other PREFETCH*h* instructions, or any other general instruction. It is ordered with respect to serializing instructions such as CPUID, WRMSR, OUT, and MOV CR.

Operation

```
FETCH (m8);
```

Intel C/C++ Compiler Intrinsic Equivalent

```
void_mm_prefetch(char *p, int i)
```

The argument "*p" gives the address of the byte (and corresponding cache line) to be prefetched. The value "i" gives a constant (_MM_HINT_T0, _MM_HINT_T1, _MM_HINT_T2, or _MM_HINT_NTA) that specifies the type of prefetch operation to be performed.

Numeric Exceptions

None.

Protected Mode Exceptions

None.

Real Address Mode Exceptions

None.

Virtual 8086 Mode Exceptions

None.

PSADBW—Compute Sum of Absolute Differences

Opcode	Instruction	Description
0F F6 /r	PSADBW *mm1, mm2/m64*	Computes the absolute differences of the packed unsigned byte integers from *mm2 /m64* and *mm1*; differences are then summed to produce an unsigned word integer result.
66 0F F6 /r	PSADBW *xmm1*, *xmm2/m128*	Computes the absolute differences of the packed unsigned byte integers from *xmm2 /m128* and *xmm1*; the 8 low differences and 8 high differences are then summed separately to produce two unsigned word integer results.

Description

Computes the absolute value of the difference of 8 unsigned byte integers from the source operand (second operand) and from the destination operand (first operand). These 8 differences are then summed to produce an unsigned word integer result that is stored in the destination operand. The source operand can be an MMX technology register or a 64-bit memory location or it can be an XMM register or a 128-bit memory location. The destination operand can be an MMX technology register or an XMM register. Figure 4-5 shows the operation of the PSADBW instruction when using 64-bit operands.

When operating on 64-bit operands, the word integer result is stored in the low word of the destination operand, and the remaining bytes in the destination operand are cleared to all 0s.

When operating on 128-bit operands, two packed results are computed. Here, the 8 low-order bytes of the source and destination operands are operated on to produce a word result that is stored in the low word of the destination operand, and the 8 high-order bytes are operated on to produce a word result that is stored in bits 64 through 79 of the destination operand. The remaining bytes of the destination operand are cleared.

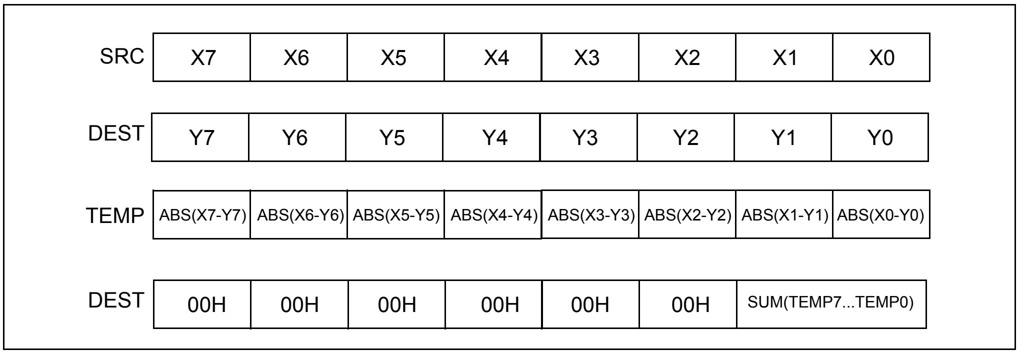


Figure 4-5. PSADBW Instruction Operation Using 64-bit Operands

Operation

```
PSADBW instructions when using 64-bit operands:
    TEMP0 ← ABS(DEST[7-0] – SRC[7-0]);
    * repeat operation for bytes 2 through 6 *;
    TEMP7 ← ABS(DEST[63-56] – SRC[63-56]);
    DEST[15:0] ← SUM(TEMP0...TEMP7);
    DEST[63:16] ← 000000000000H;

PSADBW instructions when using 128-bit operands:
    TEMP0 ← ABS(DEST[7-0] – SRC[7-0]);
    * repeat operation for bytes 2 through 14 *;
    TEMP15 ← ABS(DEST[127-120] – SRC[127-120]);
    DEST[15-0] ← SUM(TEMP0...TEMP7);
    DEST[63-6] ← 000000000000H;
    DEST[79-64] ← SUM(TEMP8...TEMP15);
    DEST[127-80] ← 000000000000H;
```

Intel C/C++ Compiler Intrinsic Equivalent

PSADBW	__m64_mm_sad_pu8(__m64 a,__m64 b)
PSADBW	__m128i _mm_sad_epu8(__m128i a, __m128i b)

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.
	(128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If EM in CR0 is set.
	(128-bit operations only) If OSFXSR in CR4 is 0.
	(128-bit operations only) If CPUID feature flag SSE2 is 0.
#NM	If TS in CR0 is set.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#GP(0)	(128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
	If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set.
	(128-bit operations only) If OSFXSR in CR4 is 0.
	(128-bit operations only) If CPUID feature flag SSE2 is 0.
#NM	If TS in CR0 is set.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made.

Numeric Exceptions

None.

PSHUFD—Shuffle Packed Doublewords

Opcode	Instruction	Description
66 0F 70 /r ib	PSHUFD *xmm1*, *xmm2/m128*, *imm8*	Shuffle the doublewords in *xmm2/m128* based on the encoding in *imm8* and store the result in *xmm1*.

Description

Copies doublewords from source operand (second operand) and inserts them in the destination operand (first operand) at the locations selected with the order operand (third operand). Figure 4-6 shows the operation of the PSHUFD instruction and the encoding of the order operand. Each 2-bit field in the order operand selects the contents of one doubleword location in the destination operand. For example, bits 0 and 1 of the order operand select the contents of doubleword 0 of the destination operand. The encoding of bits 0 and 1 of the order operand (see the field encoding in Figure 4-6) determines which doubleword from the source operand will be copied to doubleword 0 of the destination operand.

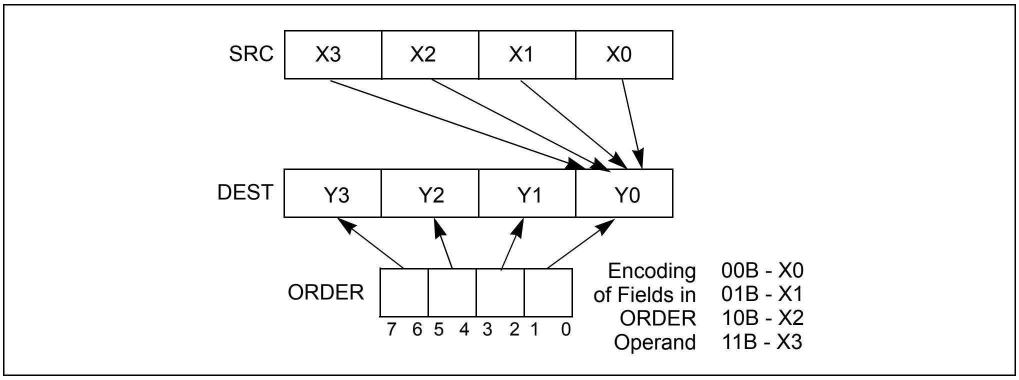


Figure 4-6. PSHUFD Instruction Operation

The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. The order operand is an 8-bit immediate.

Note that this instruction permits a doubleword in the source operand to be copied to more than one doubleword location in the destination operand.

Operation

```
DEST[31-0] ← (SRC >> (ORDER[1-0] ∗ 32) )[31-0]
DEST[63-32]  ← (SRC >> (ORDER[3-2] ∗ 32) )[31-0]
DEST[95-64]  ← (SRC >> (ORDER[5-4] ∗ 32) )[31-0]
DEST[127-96]  ← (SRC >> (ORDER[7-6] ∗ 32) )[31-0]
```

Intel C/C++ Compiler Intrinsic Equivalent

PSHUFD	__m128i _mm_shuffle_epi32(__m128i a, int n)

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 a memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If EM in CR0 is set.
	If OSFXSR in CR4 is 0.
	If CPUID feature flag SSE2 is 0.
#NM	If TS in CR0 is set.
#PF(fault-code)	If a page fault occurs.

Real-Address Mode Exceptions

#GP(0)	If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
	If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set.
	If OSFXSR in CR4 is 0.
	If CPUID feature flag SSE2 is 0.
#NM	If TS in CR0 is set.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.

Numeric Exceptions

None.

PSHUFHW—Shuffle Packed High Words

Opcode	Instruction	Description
F3 0F 70 /r ib	PSHUFHW *xmm1*, *xmm2/m128*, *imm8*	Shuffle the high words in *xmm2/m128* based on the encoding in *imm8* and store the result in *xmm1*.

Description

Copies words from the high quadword of the source operand (second operand) and inserts them in the high quadword of the destination operand (first operand) at word locations selected with the order operand (third operand). This operation is similar to the operation used by the PSHUFD instruction, which is illustrated in Figure 4-6. For the PSHUFHW instruction, each 2-bit field in the order operand selects the contents of one word location in the high quadword of the destination operand. The binary encodings of the order operand fields select words (0, 1, 2 or 3, 4) from the high quadword of the source operand to be copied to the destination operand. The low quadword of the source operand is copied to the low quadword of the destination operand.

The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. The order operand is an 8-bit immediate.

Note that this instruction permits a word in the high quadword of the source operand to be copied to more than one word location in the high quadword of the destination operand.

Operation

```
DEST[63-0] ← (SRC[63-0]
DEST[79-64] ← (SRC >> (ORDER[1-0] ∗ 16) )[79-64]
DEST[95-80] ← (SRC >> (ORDER[3-2] ∗ 16) )[79-64]
DEST[111-96]  ← (SRC >> (ORDER[5-4] ∗ 16) )[79-64]
DEST[127-112] ← (SRC >> (ORDER[7-6] ∗ 16) )[79-64]
```

Intel C/C++ Compiler Intrinsic Equivalent

```
PSHUFHW        __m128i _mm_shufflehi_epi16(__m128i a, int n)
```

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 a memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.

#UD	If EM in CR0 is set.
	If OSFXSR in CR4 is 0.
	If CPUID feature flag SSE2 is 0.
#NM	If TS in CR0 is set.
#PF(fault-code)	If a page fault occurs.

Real-Address Mode Exceptions

#GP(0)	If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
	If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set.
	If OSFXSR in CR4 is 0.
	If CPUID feature flag SSE2 is 0.
#NM	If TS in CR0 is set.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.

Numeric Exceptions

None.

PSHUFLW—Shuffle Packed Low Words

Opcode	Instruction	Description
F2 0F 70 /r ib	PSHUFLW *xmm1*, *xmm2/m128*, *imm8*	Shuffle the low words in *xmm2/m128* based on the encoding in *imm8* and store the result in *xmm1*.

Description

Copies words from the low quadword of the source operand (second operand) and inserts them in the low quadword of the destination operand (first operand) at word locations selected with the order operand (third operand). This operation is similar to the operation used by the PSHUFD instruction, which is illustrated in Figure 4-6. For the PSHUFLW instruction, each 2-bit field in the order operand selects the contents of one word location in the low quadword of the destination operand. The binary encodings of the order operand fields select words (0, 1, 2, or 3) from the low quadword of the source operand to be copied to the destination operand. The high quadword of the source operand is copied to the high quadword of the destination operand.

The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. The order operand is an 8-bit immediate.

Note that this instruction permits a word in the low quadword of the source operand to be copied to more than one word location in the low quadword of the destination operand.

Operation

```
DEST[15-0] ← (SRC >> (ORDER[1-0] ∗ 16) )[15-0]
DEST[31-16]  ← (SRC >> (ORDER[3-2] ∗ 16) )[15-0]
DEST[47-32]  ← (SRC >> (ORDER[5-4] ∗ 16) )[15-0]
DEST[63-48]  ← (SRC >> (ORDER[7-6] ∗ 16) )[15-0]
DEST[127-64]  ← (SRC[127-64]
```

Intel C/C++ Compiler Intrinsic Equivalent

PSHUFLW	__m128i _mm_shufflelo_epi16(__m128i a, int n)

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 a memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.

#UD	If EM in CR0 is set.
	If OSFXSR in CR4 is 0.
	If CPUID feature flag SSE2 is 0.
#NM	If TS in CR0 is set.
#PF(fault-code)	If a page fault occurs.

Real-Address Mode Exceptions

#GP(0)	If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
	If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set.
	If OSFXSR in CR4 is 0.
	If CPUID feature flag SSE2 is 0.
#NM	If TS in CR0 is set.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.

Numeric Exceptions

None.

PSHUFW—Shuffle Packed Words

Opcode	Instruction	Description
0F 70 /r ib	PSHUFW *mm1, mm2/m64, imm8*	Shuffle the words in *mm2/m64* based on the encoding in *imm8* and store the result in *mm1*.

Description

Copies words from the source operand (second operand) and inserts them in the destination operand (first operand) at word locations selected with the order operand (third operand). This operation is similar to the operation used by the PSHUFD instruction, which is illustrated in Figure 4-6. For the PSHUFW instruction, each 2-bit field in the order operand selects the contents of one word location in the destination operand. The encodings of the order operand fields select words from the source operand to be copied to the destination operand.

The source operand can be an MMX technology register or a 64-bit memory location. The destination operand is an MMX technology register. The order operand is an 8-bit immediate.

Note that this instruction permits a word in the source operand to be copied to more than one word location in the destination operand.

Operation

```
DEST[15-0] ← (SRC >> (ORDER[1-0] ∗ 16) )[15-0]
DEST[31-16] ← (SRC >> (ORDER[3-2] ∗ 16) )[15-0]
DEST[47-32] ← (SRC >> (ORDER[5-4] ∗ 16) )[15-0]
DEST[63-48] ← (SRC >> (ORDER[7-6] ∗ 16) )[15-0]
```

Intel C/C++ Compiler Intrinsic Equivalent

```
PSHUFW        __m64 _mm_shuffle_pi16(__m64 a, int n)
```

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.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If EM in CR0 is set.
#NM	If TS in CR0 is set.
#MF	If there is a pending x87 FPU exception.
#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.

Real-Address Mode Exceptions

#GP(0)	If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set.
#NM	If TS in CR0 is set.
#MF	If there is a pending x87 FPU exception.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

Numeric Exceptions

None.

PSLLDQ—Shift Double Quadword Left Logical

Opcode	Instruction	Description
66 0F 73 /7 ib	PSLLDQ *xmm1*, *imm8*	Shift *xmm1* left by *imm8* bytes while shifting in 0s.

Description

Shifts the destination operand (first operand) to the left by the number of bytes specified in the count operand (second operand). The empty low-order bytes are cleared (set to all 0s). If the value specified by the count operand is greater than 15, the destination operand is set to all 0s. The destination operand is an XMM register. The count operand is an 8-bit immediate.

Operation

```
TEMP ← COUNT;
if (TEMP > 15) TEMP ← 16;
DEST ← DEST << (TEMP ∗ 8);
```

Intel C/C++ Compiler Intrinsic Equivalent

```
PSLLDQ        __m128i _mm_slli_si128 ( __m128i a, int imm)
```

Flags Affected

None.

Protected Mode Exceptions

#UD	If EM in CR0 is set.
	If OSFXSR in CR4 is 0.
	If CPUID feature flag SSE2 is 0.
#NM	If TS in CR0 is set.

Real-Address Mode Exceptions

Same exceptions as in Protected Mode

Virtual-8086 Mode Exceptions

Same exceptions as in Protected Mode

Numeric Exceptions

None.

PSLLW/PSLLD/PSLLQ—Shift Packed Data Left Logical

Opcode	Instruction	Description
0F F1 /r	PSLLW *mm, mm/m64*	Shift words in *mm* left *mm/m64* while shifting in 0s.
66 0F F1 /r	PSLLW *xmm1*, *xmm2/m128*	Shift words in *xmm1* left by *xmm2/m128* while shifting in 0s.
0F 71 /6 ib	PSLLW *mm, imm8*	Shift words in *mm* left by *imm8* while shifting in 0s.
66 0F 71 /6 ib	PSLLW *xmm1*, *imm8*	Shift words in *xmm1* left by *imm8* while shifting in 0s.
0F F2 /r	PSLLD *mm, mm/m64*	Shift doublewords in *mm* left by *mm/m64* while shifting in 0s.
66 0F F2 /r	PSLLD *xmm1*, *xmm2/m128*	Shift doublewords in *xmm1* left by *xmm2/m128* while shifting in 0s.
0F 72 /6 ib	PSLLD *mm, imm8*	Shift doublewords in *mm* left by *imm8* while shifting in 0s.
66 0F 72 /6 ib	PSLLD *xmm1*, *imm8*	Shift doublewords in *xmm1* left by *imm8* while shifting in 0s.
0F F3 /r	PSLLQ *mm, mm/m64*	Shift quadword in *mm* left by *mm/m64* while shifting in 0s.
66 0F F3 /r	PSLLQ *xmm1*, *xmm2/m128*	Shift quadwords in *xmm1* left by *xmm2/m128* while shifting in 0s.
0F 73 /6 ib	PSLLQ *mm, imm8*	Shift quadword in *mm* left by *imm8* while shifting in 0s.
66 0F 73 /6 ib	PSLLQ *xmm1*, *imm8*	Shift quadwords in *xmm1* left by *imm8* while shifting in 0s.

Description

Shifts the bits in the individual data elements (words, doublewords, or quadword) in the destination operand (first operand) to the left by the number of bits specified in the count operand (second operand). As the bits in the data elements are shifted left, the empty low-order bits are cleared (set to 0). If the value specified by the count operand is greater than 15 (for words), 31 (for doublewords), or 63 (for a quadword), then the destination operand is set to all 0s. (Figure 4-7 gives an example of shifting words in a 64-bit operand.) The destination operand may be an MMX technology register or an XMM register; the count operand can be either an MMX technology register or an 64-bit memory location, an XMM register or a 128-bit memory location, or an 8-bit immediate.

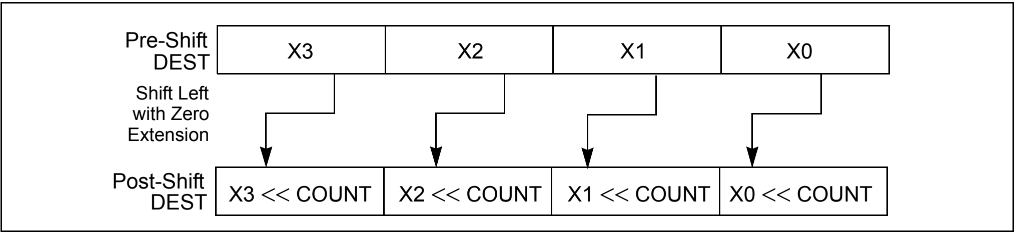


Figure 4-7. PSLLW, PSLLD, and PSLLQ Instruction Operation Using 64-bit Operand

The PSLLW instruction shifts each of the words in the destination operand to the left by the number of bits specified in the count operand; the PSLLD instruction shifts each of the doublewords in the destination operand; and the PSLLQ instruction shifts the quadword (or quadwords) in the destination operand.

Operation

```
PSLLW instruction with 64-bit operand:
    IF (COUNT > 15)
    THEN
         DEST[64..0] ← 0000000000000000H
    ELSE
         DEST[15..0] ← ZeroExtend(DEST[15..0] << COUNT);
         * repeat shift operation for 2nd and 3rd words *;
         DEST[63..48] ← ZeroExtend(DEST[63..48] << COUNT);
    FI;

PSLLD instruction with 64-bit operand:
    IF (COUNT > 31)
    THEN
         DEST[64..0] ← 0000000000000000H
    ELSE
         DEST[31..0] ← ZeroExtend(DEST[31..0] << COUNT);
         DEST[63..32] ← ZeroExtend(DEST[63..32] << COUNT);
    FI;

PSLLQ instruction with 64-bit operand:
    IF (COUNT > 63)
    THEN
         DEST[64..0] ← 0000000000000000H
    ELSE
         DEST ← ZeroExtend(DEST << COUNT);
    FI;

PSLLW instruction with 128-bit operand:
    IF (COUNT > 15)
    THEN
         DEST[128..0] ← 00000000000000000000000000000000H
    ELSE
         DEST[15-0]  ← ZeroExtend(DEST[15-0] << COUNT);
         * repeat shift operation for 2nd through 7th words *;
         DEST[127-112] ← ZeroExtend(DEST[127-112] << COUNT);
    FI;

PSLLD instruction with 128-bit operand:
    IF (COUNT > 31)
    THEN
```

```
        DEST[128..0] ← 00000000000000000000000000000000H
    ELSE
        DEST[31-0]  ← ZeroExtend(DEST[31-0] << COUNT);
        * repeat shift operation for 2nd and 3rd doublewords *;
        DEST[127-96] ← ZeroExtend(DEST[127-96] << COUNT);
    FI;

PSLLQ instruction with 128-bit operand:
    IF (COUNT > 63)
    THEN
        DEST[128..0] ← 00000000000000000000000000000000H
    ELSE
        DEST[63-0]  ← ZeroExtend(DEST[63-0] << COUNT);
        DEST[127-64] ← ZeroExtend(DEST[127-64] << COUNT);
    FI;
```

Intel C/C++ Compiler Intrinsic Equivalents

PSLLW	__m64 _mm_slli_pi16 (__m64 m, int count)
PSLLW	__m64 _mm_sll_pi16(__m64 m, __m64 count)
PSLLW	__m128i _mm_slli_pi16(__m64 m, int count)
PSLLW	__m128i _mm_slli_pi16(__m128i m, __m128i count)
PSLLD	__m64 _mm_slli_pi32(__m64 m, int count)
PSLLD	__m64 _mm_sll_pi32(__m64 m, __m64 count)
PSLLD	__m128i _mm_slli_epi32(__m128i m, int count)
PSLLD	__m128i _mm_sll_epi32(__m128i m, __m128i count)
PSLLQ	__m64 _mm_slli_si64(__m64 m, int count)
PSLLQ	__m64 _mm_sll_si64(__m64 m, __m64 count)
PSLLQ	__m128i _mm_slli_si64(__m128i m, int count)
PSLLQ	__m128i _mm_sll_si64(__m128i m, __m128i count)

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.
	(128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.

#UD	If EM in CR0 is set.
	(128-bit operations only) If OSFXSR in CR4 is 0.
	(128-bit operations only) If CPUID feature flag SSE2 is 0.
#NM	If TS in CR0 is set.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#GP(0)	(128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
	If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set.
	(128-bit operations only) If OSFXSR in CR4 is 0.
	(128-bit operations only) If CPUID feature flag SSE2 is 0.
#NM	If TS in CR0 is set.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made.

Numeric Exceptions

None.

PSRAW/PSRAD—Shift Packed Data Right Arithmetic

Opcode	Instruction	Description
0F E1 /r	PSRAW *mm, mm/m64*	Shift words in *mm* right by *mm/m64* while shifting in sign bits.
66 0F E1 /r	PSRAW *xmm1*, *xmm2/m128*	Shift words in *xmm1* right by *xmm2/m128* while shifting in sign bits.
0F 71 /4 ib	PSRAW *mm, imm8*	Shift words in *mm* right by *imm8* while shifting in sign bits
66 0F 71 /4 ib	PSRAW *xmm1*, imm8	Shift words in *xmm1* right by *imm8* while shifting in sign bits
0F E2 /r	PSRAD *mm, mm/m64*	Shift doublewords in *mm* right by *mm/m64* while shifting in sign bits.
66 0F E2 /r	PSRAD *xmm1*, *xmm2/m128*	Shift doubleword in *xmm1* right by *xmm2 /m128* while shifting in sign bits.
0F 72 /4 ib	PSRAD *mm, imm8*	Shift doublewords in *mm* right by *imm8* while shifting in sign bits.
66 0F 72 /4 ib	PSRAD *xmm1*, imm8	Shift doublewords in *xmm1* right by *imm8* while shifting in sign bits.

Description

Shifts the bits in the individual data elements (words or doublewords) in the destination operand (first operand) to the right by the number of bits specified in the count operand (second operand). As the bits in the data elements are shifted right, the empty high-order bits are filled with the initial value of the sign bit of the data element. If the value specified by the count operand is greater than 15 (for words) or 31 (for doublewords), each destination data element is filled with the initial value of the sign bit of the element. (Figure 4-8 gives an example of shifting words in a 64-bit operand.)

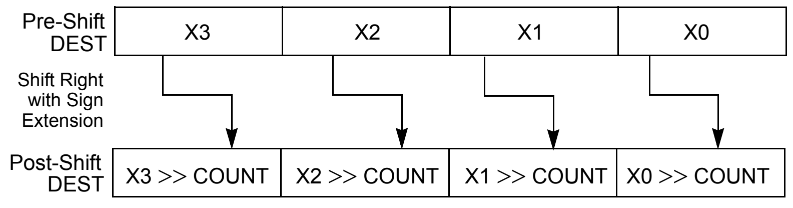


Figure 4-8. PSRAW and PSRAD Instruction Operation Using a 64-bit Operand

The destination operand may be an MMX technology register or an XMM register; the count operand can be either an MMX technology register or an 64-bit memory location, an XMM register or a 128-bit memory location, or an 8-bit immediate.

The PSRAW instruction shifts each of the words in the destination operand to the right by the number of bits specified in the count operand, and the PSRAD instruction shifts each of the doublewords in the destination operand.

Operation

```
PSRAW instruction with 64-bit operand:
    IF (COUNT > 15)
         THEN COUNT ← 16;
    FI;
         DEST[15..0] ← SignExtend(DEST[15..0] >> COUNT);
         * repeat shift operation for 2nd and 3rd words *;
         DEST[63..48] ← SignExtend(DEST[63..48] >> COUNT);

PSRAD instruction with 64-bit operand:
    IF (COUNT > 31)
         THEN COUNT ← 32;
    FI;
    ELSE
         DEST[31..0] ← SignExtend(DEST[31..0] >> COUNT);
         DEST[63..32] ← SignExtend(DEST[63..32] >> COUNT);

PSRAW instruction with 128-bit operand:
    IF (COUNT > 15)
         THEN COUNT ← 16;
    FI;
    ELSE
         DEST[15-0]  ← SignExtend(DEST[15-0] >> COUNT);
         * repeat shift operation for 2nd through 7th words *;
         DEST[127-112] ← SignExtend(DEST[127-112] >> COUNT);

PSRAD instruction with 128-bit operand:
    IF (COUNT > 31)
         THEN COUNT ← 32;
    FI;
    ELSE
         DEST[31-0]  ← SignExtend(DEST[31-0] >> COUNT);
         * repeat shift operation for 2nd and 3rd doublewords *;
         DEST[127-96] ← SignExtend(DEST[127-96] >>COUNT);
```

Intel C/C++ Compiler Intrinsic Equivalents

PSRAW	__m64 _mm_srai_pi16 (__m64 m, int count)
PSRAW	__m64 _mm_sraw_pi16 (__m64 m, __m64 count)
PSRAD	__m64 _mm_srai_pi32 (__m64 m, int count)
PSRAD	__m64 _mm_sra_pi32 (__m64 m, __m64 count)

PSRAW	__m128i _mm_srai_epi16(__m128i m, int count)
PSRAW	__m128i _mm_sra_epi16(__m128i m, __m128i count))
PSRAD	__m128i _mm_srai_epi32 (__m128i m, int count)
PSRAD	__m128i _mm_sra_epi32 (__m128i m, __m128i count)

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.
	(128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If EM in CR0 is set.
	(128-bit operations only) If OSFXSR in CR4 is 0.
	(128-bit operations only) If CPUID feature flag SSE2 is 0.
#NM	If TS in CR0 is set.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#GP(0)	(128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
	If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set.
	(128-bit operations only) If OSFXSR in CR4 is 0.
	(128-bit operations only) If CPUID feature flag SSE2 is 0.
#NM	If TS in CR0 is set.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made.

Numeric Exceptions

None.

PSRLDQ—Shift Double Quadword Right Logical

Opcode	Instruction	Description
66 0F 73 /3 ib	PSRLDQ *xmm1*, *imm8*	Shift *xmm1* right by *imm8* while shifting in 0s.

Description

Shifts the destination operand (first operand) to the right by the number of bytes specified in the count operand (second operand). The empty high-order bytes are cleared (set to all 0s). If the value specified by the count operand is greater than 15, the destination operand is set to all 0s. The destination operand is an XMM register. The count operand is an 8-bit immediate.

Operation

```
TEMP ← COUNT;
if (TEMP > 15) TEMP ← 16;
DEST ← DEST >> (temp ∗ 8);
```

Intel C/C++ Compiler Intrinsic Equivalents

PSRLDQ	__m128i _mm_srli_si128 (__m128i a, int imm)

Flags Affected

None.

Protected Mode Exceptions

#UD	If EM in CR0 is set.
	If OSFXSR in CR4 is 0.
	If CPUID feature flag SSE2 is 0.
#NM	If TS in CR0 is set.

Real-Address Mode Exceptions

Same exceptions as in Protected Mode

Virtual-8086 Mode Exceptions

Same exceptions as in Protected Mode

Numeric Exceptions

None.

PSRLW/PSRLD/PSRLQ—Shift Packed Data Right Logical

Opcode	Instruction	Description
0F D1 /r	PSRLW *mm, mm/m64*	Shift words in *mm* right by amount specified in *mm/m64* while shifting in 0s.
66 0F D1 /r	PSRLW *xmm1*, *xmm2/m128*	Shift words in *xmm1* right by amount specified in *xmm2/m128* while shifting in 0s.
0F 71 /2 ib	PSRLW *mm, imm8*	Shift words in *mm* right by *imm8* while shifting in 0s.
66 0F 71 /2 ib	PSRLW *xmm1*, *imm8*	Shift words in *xmm1* right by *imm8* while shifting in 0s.
0F D2 /r	PSRLD *mm, mm/m64*	Shift doublewords in *mm* right by amount specified in *mm/m64* while shifting in 0s.
66 0F D2 /r	PSRLD *xmm1*, *xmm2/m128*	Shift doublewords in *xmm1* right by amount specified in *xmm2 /m128* while shifting in 0s.
0F 72 /2 ib	PSRLD *mm, imm8*	Shift doublewords in *mm* right by *imm8* while shifting in 0s.
66 0F 72 /2 ib	PSRLD *xmm1*, imm8	Shift doublewords in *xmm1* right by imm8 while shifting in 0s.
0F D3 /r	PSRLQ *mm, mm/m64*	Shift *mm* right by amount specified in *mm/m64* while shifting in 0s.
66 0F D3 /r	PSRLQ *xmm1*, *xmm2/m128*	Shift quadwords in *xmm1* right by amount specified in *xmm2/m128* while shifting in 0s.
0F 73 /2 ib	PSRLQ *mm, imm8*	Shift *mm* right by *imm8* while shifting in 0s.
66 0F 73 /2 ib	PSRLQ *xmm1*, imm8	Shift quadwords in *xmm1* right by *imm8* while shifting in 0s.

Description

Shifts the bits in the individual data elements (words, doublewords, or quadword) in the destination operand (first operand) to the right by the number of bits specified in the count operand (second operand). As the bits in the data elements are shifted right, the empty high-order bits are cleared (set to 0). If the value specified by the count operand is greater than 15 (for words), 31 (for doublewords), or 63 (for a quadword), then the destination operand is set to all 0s. (Figure 4-9 gives an example of shifting words in a 64-bit operand.) The destination operand may be an MMX technology register or an XMM register; the count operand can be either an MMX technology register or an 64-bit memory location, an XMM register or a 128-bit memory location, or an 8-bit immediate.

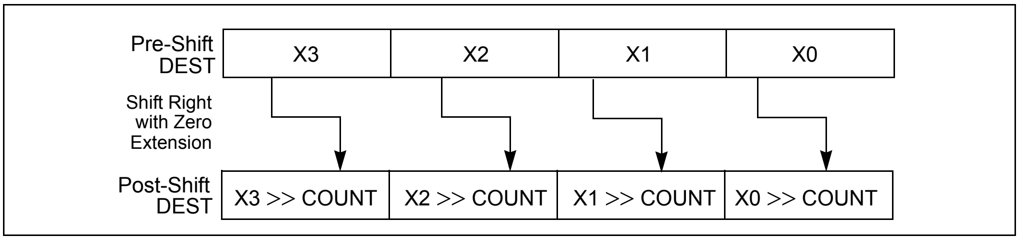


Figure 4-9. PSRLW, PSRLD, and PSRLQ Instruction Operation Using 64-bit Operand

The PSRLW instruction shifts each of the words in the destination operand to the right by the number of bits specified in the count operand; the PSRLD instruction shifts each of the doublewords in the destination operand; and the PSRLQ instruction shifts the quadword (or quadwords) in the destination operand.

Operation

```
PSRLW instruction with 64-bit operand:
    IF (COUNT > 15)
    THEN
         DEST[64..0] ← 0000000000000000H
    ELSE
         DEST[15..0] ← ZeroExtend(DEST[15..0] >> COUNT);
         * repeat shift operation for 2nd and 3rd words *;
         DEST[63..48] ← ZeroExtend(DEST[63..48] >> COUNT);
    FI;

PSRLD instruction with 64-bit operand:
    IF (COUNT > 31)
    THEN
         DEST[64..0] ← 0000000000000000H
    ELSE
         DEST[31..0] ← ZeroExtend(DEST[31..0] >> COUNT);
         DEST[63..32] ← ZeroExtend(DEST[63..32] >> COUNT);
    FI;

PSRLQ instruction with 64-bit operand:
    IF (COUNT > 63)
    THEN
         DEST[64..0] ← 0000000000000000H
    ELSE
         DEST ← ZeroExtend(DEST >> COUNT);
    FI;

PSRLW instruction with 128-bit operand:
    IF (COUNT > 15)
    THEN
         DEST[128..0] ← 00000000000000000000000000000000H
    ELSE
         DEST[15-0]  ← ZeroExtend(DEST[15-0] >> COUNT);
         * repeat shift operation for 2nd through 7th words *;
         DEST[127-112] ← ZeroExtend(DEST[127-112] >> COUNT);
    FI;

PSRLD instruction with 128-bit operand:
    IF (COUNT > 31)
    THEN
```

```
        DEST[128..0] ← 00000000000000000000000000000000H
    ELSE
        DEST[31-0]  ← ZeroExtend(DEST[31-0] >> COUNT);
        * repeat shift operation for 2nd and 3rd doublewords *;
        DEST[127-96] ← ZeroExtend(DEST[127-96] >> COUNT);
    FI;

PSRLQ instruction with 128-bit operand:
    IF (COUNT > 15)
    THEN
        DEST[128..0] ← 00000000000000000000000000000000H
    ELSE
        DEST[63-0]  ← ZeroExtend(DEST[63-0] >> COUNT);
        DEST[127-64] ← ZeroExtend(DEST[127-64] >> COUNT);
    FI;
```

Intel C/C++ Compiler Intrinsic Equivalents

PSRLW	__m64 _mm_srli_pi16(__m64 m, int count)
PSRLW	__m64 _mm_srl_pi16 (__m64 m, __m64 count)
PSRLW	__m128i _mm_srli_epi16 (__m128i m, int count)
PSRLW	__m128i _mm_srl_epi16 (__m128i m, __m128i count)
PSRLD	__m64 _mm_srli_pi32 (__m64 m, int count)
PSRLD	__m64 _mm_srl_pi32 (__m64 m, __m64 count)
PSRLD	__m128i _mm_srli_epi32 (__m128i m, int count)
PSRLD	__m128i _mm_srl_epi32 (__m128i m, __m128i count)
PSRLQ	__m64 _mm_srli_si64 (__m64 m, int count)
PSRLQ	__m64 _mm_srl_si64 (__m64 m, __m64 count)
PSRLQ	__m128i _mm_srli_epi64 (__m128i m, int count)
PSRLQ	__m128i _mm_srl_epi64 (__m128i m, __m128i count)

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.
	(128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.

#UD	If EM in CR0 is set.
	(128-bit operations only) If OSFXSR in CR4 is 0.
	(128-bit operations only) If CPUID feature flag SSE2 is 0.
#NM	If TS in CR0 is set.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#GP(0)	(128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
	If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set.
	(128-bit operations only) If OSFXSR in CR4 is 0.
	(128-bit operations only) If CPUID feature flag SSE2 is 0.
#NM	If TS in CR0 is set.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made.

Numeric Exceptions

None.

PSUBB/PSUBW/PSUBD—Subtract Packed Integers

Opcode	Instruction	Description
0F F8 /r	PSUBB *mm, mm/m64*	Subtract packed byte integers in *mm/m64* from packed byte integers in *mm*.
66 0F F8 /r	PSUBB *xmm1*, *xmm2/m128*	Subtract packed byte integers in *xmm2/m128* from packed byte integers in *xmm1*.
0F F9 /r	PSUBW *mm, mm/m64*	Subtract packed word integers in *mm/m64* from packed word integers in *mm*.
66 0F F9 /r	PSUBW *xmm1*, *xmm2/m128*	Subtract packed word integers in *xmm2/m128* from packed word integers in *xmm1*.
0F FA /r	PSUBD *mm, mm/m64*	Subtract packed doubleword integers in *mm/m64* from packed doubleword integers in *mm*.
66 0F FA /r	PSUBD *xmm1*, *xmm2/m128*	Subtract packed doubleword integers in *xmm2/mem128* from packed doubleword integers in *xmm1*.

Description

Performs an SIMD subtract of the packed integers of the source operand (second operand) from the packed integers of the destination operand (first operand), and stores the packed integer results in the destination operand. See Figure 9-4 in the *IA-32 Intel Architecture Software Developer's Manual, Volume 1* for an illustration of an SIMD operation. Overflow is handled with wraparound, as described in the following paragraphs.

These instructions can operate on either 64-bit or 128-bit operands. When operating on 64-bit operands, the destination operand must be an MMX technology register and the source operand can be either an MMX technology register or a 64-bit memory location. When operating on 128-bit operands, the destination operand must be an XMM register and the source operand can be either an XMM register or a 128-bit memory location.

The PSUBB instruction subtracts packed byte integers. When an individual result is too large or too small to be represented in a byte, the result is wrapped around and the low 8 bits are written to the destination element.

The PSUBW instruction subtracts packed word integers. When an individual result is too large or too small to be represented in a word, the result is wrapped around and the low 16 bits are written to the destination element.

The PSUBD instruction subtracts packed doubleword integers. When an individual result is too large or too small to be represented in a doubleword, the result is wrapped around and the low 32 bits are written to the destination element.

Note that the PSUBB, PSUBW, and PSUBD instructions can operate on either unsigned or signed (two's complement notation) packed integers; however, it does not set bits in the EFLAGS register to indicate overflow and/or a carry. To prevent undetected overflow conditions, software must control the ranges of values operated on.

Operation

```
PSUBB instruction with 64-bit operands:
    DEST[7..0] ← DEST[7..0] − SRC[7..0];
    * repeat subtract operation for 2nd through 7th byte *;
    DEST[63..56] ← DEST[63..56] − SRC[63..56];

PSUBB instruction with 128-bit operands:
    DEST[7-0] ← DEST[7-0] − SRC[7-0];
    * repeat subtract operation for 2nd through 14th byte *;
    DEST[127-120] ← DEST[111-120] − SRC[127-120];

PSUBW instruction with 64-bit operands:
    DEST[15..0] ← DEST[15..0] − SRC[15..0];
    * repeat subtract operation for 2nd and 3rd word *;
    DEST[63..48] ← DEST[63..48] − SRC[63..48];

PSUBW instruction with 128-bit operands:
    DEST[15-0]  ← DEST[15-0] − SRC[15-0];
    * repeat subtract operation for 2nd through 7th word *;
    DEST[127-112] ← DEST[127-112] − SRC[127-112];

PSUBD instruction with 64-bit operands:
    DEST[31..0] ← DEST[31..0] − SRC[31..0];
    DEST[63..32] ← DEST[63..32] − SRC[63..32];

PSUBD instruction with 128-bit operands:
    DEST[31-0]  ← DEST[31-0] − SRC[31-0];
    * repeat subtract operation for 2nd and 3rd doubleword *;
    DEST[127-96] ← DEST[127-96] − SRC[127-96];
```

Intel C/C++ Compiler Intrinsic Equivalents

PSUBB	__m64 _mm_sub_pi8(__m64 m1, __m64 m2)
PSUBW	__m64 _mm_sub_pi16(__m64 m1, __m64 m2)
PSUBD	__m64 _mm_sub_pi32(__m64 m1, __m64 m2)
PSUBB	__m128i _mm_sub_epi8 (__m128i a, __m128i b)
PSUBW	__m128i _mm_sub_epi16 (__m128i a, __m128i b)
PSUBD	__m128i _mm_sub_epi32 (__m128i a, __m128i b)

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.
	(128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If EM in CR0 is set.
	(128-bit operations only) If OSFXSR in CR4 is 0.
	(128-bit operations only) If CPUID feature flag SSE2 is 0.
#NM	If TS in CR0 is set.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#GP(0)	(128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
	If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set.
	(128-bit operations only) If OSFXSR in CR4 is 0.
	(128-bit operations only) If CPUID feature flag SSE2 is 0.
#NM	If TS in CR0 is set.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made.

Numeric Exceptions

None.

PSUBQ—Subtract Packed Quadword Integers

Opcode	Instruction	Description
0F FB /r	PSUBQ *mm1*, *mm2/m64*	Subtract quadword integer in *mm1* from *mm2 /m64*.
66 0F FB /r	PSUBQ *xmm1*, *xmm2/m128*	Subtract packed quadword integers in *xmm1* from *xmm2 /m128*.

Description

Subtracts the second operand (source operand) from the first operand (destination operand) and stores the result in the destination operand. The source operand can be a quadword integer stored in an MMX technology register or a 64-bit memory location, or it can be two packed quadword integers stored in an XMM register or an 128-bit memory location. The destination operand can be a quadword integer stored in an MMX technology register or two packed quadword integers stored in an XMM register. When packed quadword operands are used, an SIMD subtract is performed. When a quadword result is too large to be represented in 64 bits (overflow), the result is wrapped around and the low 64 bits are written to the destination element (that is, the carry is ignored).

Note that the PSUBQ instruction can operate on either unsigned or signed (two's complement notation) integers; however, it does not set bits in the EFLAGS register to indicate overflow and/or a carry. To prevent undetected overflow conditions, software must control the ranges of the values operated on.

Operation

```
PSUBQ instruction with 64-Bit operands:
    DEST[63-0] ← DEST[63-0] – SRC[63-0];

PSUBQ instruction with 128-Bit operands:
    DEST[63-0] ← DEST[63-0] – SRC[63-0];
    DEST[127-64] ← DEST[127-64] – SRC[127-64];
```

Intel C/C++ Compiler Intrinsic Equivalents

PSUBQ	__m64 _mm_sub_si64(__m64 m1, __m64 m2)
PSUBQ	__m128i _mm_sub_epi64(__m128i m1, __m128i m2)

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.
	(128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If EM in CR0 is set.
	(128-bit operations only) If OSFXSR in CR4 is 0.
	(128-bit operations only) If CPUID feature flag SSE2 is 0.
#NM	If TS in CR0 is set.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#GP(0)	(128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
	If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set.
	(128-bit operations only) If OSFXSR in CR4 is 0.
	(128-bit operations only) If CPUID feature flag SSE2 is 0.
#NM	If TS in CR0 is set.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made.

Numeric Exceptions

None.

PSUBSB/PSUBSW—Subtract Packed Signed Integers with Signed Saturation

Opcode	Instruction	Description
0F E8 /r	PSUBSB *mm, mm/m64*	Subtract signed packed bytes in *mm/m64* from signed packed bytes in *mm* and saturate results.
66 0F E8 /r	PSUBSB *xmm1*, *xmm2/m128*	Subtract packed signed byte integers in *xmm2/m128* from packed signed byte integers in *xmm1* and saturate results.
0F E9 /r	PSUBSW *mm, mm/m64*	Subtract signed packed words in *mm/m64* from signed packed words in *mm* and saturate results.
66 0F E9 /r	PSUBSW *xmm1*, *xmm2/m128*	Subtract packed signed word integers in *xmm2/m128* from packed signed word integers in *xmm1* and saturate results.

Description

Performs an SIMD subtract of the packed signed integers of the source operand (second operand) from the packed signed integers of the destination operand (first operand), and stores the packed integer results in the destination operand. See Figure 9-4 in the *IA-32 Intel Architecture Software Developer's Manual, Volume 1* for an illustration of an SIMD operation. Overflow is handled with signed saturation, as described in the following paragraphs.

These instructions can operate on either 64-bit or 128-bit operands. When operating on 64-bit operands, the destination operand must be an MMX technology register and the source operand can be either an MMX technology register or a 64-bit memory location. When operating on 128-bit operands, the destination operand must be an XMM register and the source operand can be either an XMM register or a 128-bit memory location.

The PSUBSB instruction subtracts packed signed byte integers. When an individual byte result is beyond the range of a signed byte integer (that is, greater than 7FH or less than 80H), the saturated value of 7FH or 80H, respectively, is written to the destination operand.

The PSUBSW instruction subtracts packed signed word integers. When an individual word result is beyond the range of a signed word integer (that is, greater than 7FFFH or less than 8000H), the saturated value of 7FFFH or 8000H, respectively, is written to the destination operand.

Operation

```
PSUBSB instruction with 64-bit operands:
    DEST[7..0] ← SaturateToSignedByte(DEST[7..0] – SRC (7..0]) ;
    * repeat subtract operation for 2nd through 7th bytes *;
    DEST[63..56] ← SaturateToSignedByte(DEST[63..56] – SRC[63..56] );
PSUBSB instruction with 128-bit operands:
    DEST[7-0] ← SaturateToSignedByte (DEST[7-0] – SRC[7-0]);
    * repeat subtract operation for 2nd through 14th bytes *;
    DEST[127-120] ← SaturateToSignedByte (DEST[111-120] – SRC[127-120]);
```

```
PSUBSW instruction with 64-bit operands
    DEST[15..0] ← SaturateToSignedWord(DEST[15..0] − SRC[15..0] );
    * repeat subtract operation for 2nd and 7th words *;
    DEST[63..48] ← SaturateToSignedWord(DEST[63..48] − SRC[63..48] );

PSUBSW instruction with 128-bit operands
    DEST[15-0]  ← SaturateToSignedWord (DEST[15-0] − SRC[15-0]);
    * repeat subtract operation for 2nd through 7th words *;
    DEST[127-112] ← SaturateToSignedWord (DEST[127-112] − SRC[127-112]);
```

Intel C/C++ Compiler Intrinsic Equivalents

PSUBSB	__m64 _mm_subs_pi8(__m64 m1, __m64 m2)
PSUBSB	__m128i _mm_subs_epi8(__m128i m1, __m128i m2)
PSUBSW	__m64 _mm_subs_pi16(__m64 m1, __m64 m2)
PSUBSW	__m128i _mm_subs_epi16(__m128i m1, __m128i m2)

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.
	(128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If EM in CR0 is set.
	(128-bit operations only) If OSFXSR in CR4 is 0.
	(128-bit operations only) If CPUID feature flag SSE2 is 0.
#NM	If TS in CR0 is set.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#GP(0)	(128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
	If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set.
	(128-bit operations only) If OSFXSR in CR4 is 0.
	(128-bit operations only) If CPUID feature flag SSE2 is 0.
#NM	If TS in CR0 is set.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made.

Numeric Exceptions

None.

PSUBUSB/PSUBUSW—Subtract Packed Unsigned Integers with Unsigned Saturation

Opcode	Instruction	Description
0F D8 /r	PSUBUSB *mm, mm/m64*	Subtract unsigned packed bytes in *mm/m64* from unsigned packed bytes in *mm* and saturate result.
66 0F D8 /r	PSUBUSB *xmm1*, *xmm2/m128*	Subtract packed unsigned byte integers in *xmm2/m128* from packed unsigned byte integers in *xmm1* and saturate result.
0F D9 /r	PSUBUSW *mm, mm/m64*	Subtract unsigned packed words in *mm/m64* from unsigned packed words in *mm* and saturate result.
66 0F D9 /r	PSUBUSW *xmm1*, *xmm2/m128*	Subtract packed unsigned word integers in *xmm2/m128* from packed unsigned word integers in *xmm1* and saturate result.

Description

Performs an SIMD subtract of the packed unsigned integers of the source operand (second operand) from the packed unsigned integers of the destination operand (first operand), and stores the packed unsigned integer results in the destination operand. See Figure 9-4 in the *IA-32 Intel Architecture Software Developer's Manual, Volume 1* for an illustration of an SIMD operation. Overflow is handled with unsigned saturation, as described in the following paragraphs.

These instructions can operate on either 64-bit or 128-bit operands. When operating on 64-bit operands, the destination operand must be an MMX technology register and the source operand can be either an MMX technology register or a 64-bit memory location. When operating on 128-bit operands, the destination operand must be an XMM register and the source operand can be either an XMM register or a 128-bit memory location.

The PSUBUSB instruction subtracts packed unsigned byte integers. When an individual byte result is less than zero, the saturated value of 00H is written to the destination operand.

The PSUBUSW instruction subtracts packed unsigned word integers. When an individual word result is less than zero, the saturated value of 0000H is written to the destination operand.

Operation

```
PSUBUSB instruction with 64-bit operands:
    DEST[7..0] ← SaturateToUnsignedByte(DEST[7..0] – SRC (7..0] );
    * repeat add operation for 2nd through 7th bytes *:
    DEST[63..56] ← SaturateToUnsignedByte(DEST[63..56] – SRC[63..56]

PSUBUSB instruction with 128-bit operands:
    DEST[7-0] ← SaturateToUnsignedByte (DEST[7-0] – SRC[7-0]);
    * repeat add operation for 2nd through 14th bytes *:
    DEST[127-120] ← SaturateToUnSignedByte (DEST[127-120] – SRC[127-120]);
PSUBUSW instruction with 64-bit operands:
    DEST[15..0] ← SaturateToUnsignedWord(DEST[15..0] – SRC[15..0] );
    * repeat add operation for 2nd and 3rd words *:
    DEST[63..48] ← SaturateToUnsignedWord(DEST[63..48] – SRC[63..48] );

PSUBUSW instruction with 128-bit operands:
    DEST[15-0]  ← SaturateToUnsignedWord (DEST[15-0] – SRC[15-0]);
    * repeat add operation for 2nd through 7th words *:
    DEST[127-112] ← SaturateToUnSignedWord (DEST[127-112] – SRC[127-112]);
```

Intel C/C++ Compiler Intrinsic Equivalents

PSUBUSB	__m64 _mm_sub_pu8(__m64 m1, __m64 m2)
PSUBUSB	__m128i _mm_sub_epu8(__m128i m1, __m128i m2)
PSUBUSW	__m64 _mm_sub_pu16(__m64 m1, __m64 m2)
PSUBUSW	__m128i _mm_sub_epu16(__m128i m1, __m128i m2)

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.
	(128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If EM in CR0 is set.
	(128-bit operations only) If OSFXSR in CR4 is 0.
	(128-bit operations only) If CPUID feature flag SSE2 is 0.
#NM	If TS in CR0 is set.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#GP(0)	(128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
	If any part of the operand lies outside of the effective address space from 0 to FFFFH.

#UD	If EM in CR0 is set.
	(128-bit operations only) If OSFXSR in CR4 is 0.
	(128-bit operations only) If CPUID feature flag SSE2 is 0.
#NM	If TS in CR0 is set.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made.

Numeric Exceptions

None.

PUNPCKHBW/PUNPCKHWD/PUNPCKHDQ/PUNPCKHQDQ— Unpack High Data

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

Description

Unpacks and interleaves the high-order data elements (bytes, words, doublewords, or quadwords) of the destination operand (first operand) and source operand (second operand) into the destination operand. (Figure 4-10 shows the unpack operation for bytes in 64-bit operands.). The low-order data elements are ignored.

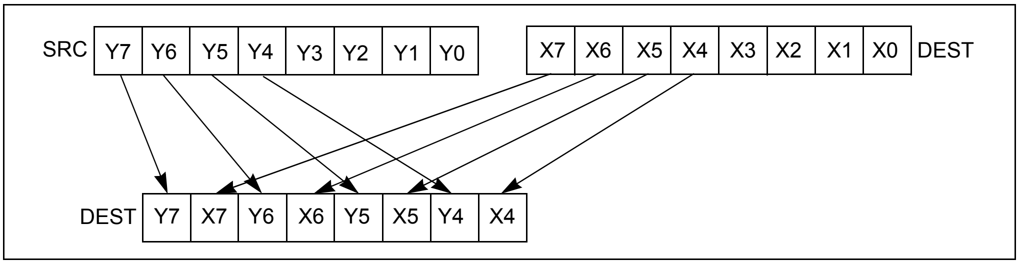


Figure 4-10. PUNPCKHBW Instruction Operation Using 64-bit Operands

The source operand can be an MMX technology register or a 64-bit memory location, or it can be an XMM register or a 128-bit memory location. The destination operand can be an MMX technology register or an XMM register. When the source data comes from a 64-bit memory operand, the full 64-bit operand is accessed from memory, but the instruction uses only the high-order 32 bits. When the source data comes from a 128-bit memory operand, an implementation may fetch only the appropriate 64 bits; however, alignment to a 16-byte boundary and normal segment checking will still be enforced.

The PUNPCKHBW instruction interleaves the high-order bytes of the source and destination operands, the PUNPCKHWD instruction interleaves the high-order words of the source and destination operands, the PUNPCKHDQ instruction interleaves the high-order doubleword (or doublewords) of the source and destination operands, and the PUNPCKHQDQ instruction interleaves the high-order quadwords of the source and destination operands.

These instructions can be used to convert bytes to words, words to doublewords, doublewords to quadwords, and quadwords to double quadwords, respectively, by placing all 0s in the source operand. Here, if the source operand contains all 0s, the result (stored in the destination operand) contains zero extensions of the high-order data elements from the original value in the destination operand. For example, with the PUNPCKHBW instruction the high-order bytes are zero extended (that is, unpacked into unsigned word integers), and with the PUNPCKHWD instruction, the high-order words are zero extended (unpacked into unsigned doubleword integers).

Operation

```
PUNPCKHBW instruction with 64-bit operands:
        DEST[7..0] ← DEST[39..32];
        DEST[15..8] ← SRC[39..32];
        DEST[23..16] ← DEST[47..40];
        DEST[31..24] ← SRC[47..40];
        DEST[39..32] ← DEST[55..48];
        DEST[47..40] ← SRC[55..48];
        DEST[55..48] ← DEST[63..56];
        DEST[63..56] ← SRC[63..56];

PUNPCKHW instruction with 64-bit operands:
        DEST[15..0] ← DEST[47..32];
        DEST[31..16] ← SRC[47..32];
        DEST[47..32] ← DEST[63..48];
        DEST[63..48] ← SRC[63..48];

PUNPCKHDQ instruction with 64-bit operands:
        DEST[31..0] ← DEST[63..32]
        DEST[63..32] ← SRC[63..32];

PUNPCKHBW instruction with 128-bit operands:
    DEST[7-0]   ← DEST[71-64];
    DEST[15-8]  ← SRC[71-64];
    DEST[23-16] ← DEST[79-72];
    DEST[31-24] ← SRC[79-72];
    DEST[39-32] ← DEST[87-80];
    DEST[47-40] ← SRC[87-80];
    DEST[55-48] ← DEST[95-88];
    DEST[63-56] ← SRC[95-88];
    DEST[71-64] ← DEST[103-96];
    DEST[79-72] ← SRC[103-96];
    DEST[87-80] ← DEST[111-104];
    DEST[95-88] ← SRC[111-104];
    DEST[103-96]  ← DEST[119-112];
    DEST[111-104] ← SRC[119-112];
    DEST[119-112] ← DEST[127-120];
    DEST[127-120] ← SRC[127-120];

PUNPCKHWD instruction with 128-bit operands:
    DEST[15-0]  ← DEST[79-64];
    DEST[31-16] ← SRC[79-64];
    DEST[47-32] ← DEST[95-80];
    DEST[63-48] ← SRC[95-80];
    DEST[79-64] ← DEST[111-96];
    DEST[95-80] ← SRC[111-96];
    DEST[111-96]  ← DEST[127-112];
    DEST[127-112] ← SRC[127-112];

PUNPCKHDQ instruction with 128-bit operands:
    DEST[31-0] ← DEST[95-64];
    DEST[63-32]  ← SRC[95-64];
    DEST[95-64]  ← DEST[127-96];
    DEST[127-96] ← SRC[127-96];

PUNPCKHQDQ instruction:
    DEST[63-0] ← DEST[127-64];
    DEST[127-64] ← SRC[127-64];
```

Intel C/C++ Compiler Intrinsic Equivalents

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 a, __m128i b)

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.
	(128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If EM in CR0 is set.
	(128-bit operations only) If OSFXSR in CR4 is 0.
	(128-bit operations only) If CPUID feature flag SSE2 is 0.
#NM	If TS in CR0 is set.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#GP(0)	If any part of the operand lies outside of the effective address space from 0 to FFFFH.
	(128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
#UD	If EM in CR0 is set.
	(128-bit operations only) If OSFXSR in CR4 is 0.
	(128-bit operations only) If CPUID feature flag SSE2 is 0.
#NM	If TS in CR0 is set.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made.

Numeric Exceptions

None.

PUNPCKLBW/PUNPCKLWD/PUNPCKLDQ/PUNPCKLQDQ— Unpack Low Data

Opcode	Instruction	Description
0F 60 /r	PUNPCKLBW *mm, mm/m32*	Interleave low-order bytes from *mm* and *mm/m32* into *mm*.
66 0F 60 /r	PUNPCKLBW *xmm1*, *xmm2/m128*	Interleave low-order bytes from *xmm1* and *xmm2/m128* into *xmm1*.
0F 61 /r	PUNPCKLWD *mm, mm/m32*	Interleave low-order words from *mm* and *mm/m32* into *mm*.
66 0F 61 /r	PUNPCKLWD *xmm1*, *xmm2/m128*	Interleave low-order words from *xmm1* and *xmm2/m128* into *xmm1*.
0F 62 /r	PUNPCKLDQ *mm, mm/m32*	Interleave low-order doublewords from *mm* and *mm/m32* into *mm*.
66 0F 62 /r	PUNPCKLDQ *xmm1*, *xmm2/m128*	Interleave low-order doublewords from *xmm1* and *xmm2/m128* into *xmm1*.
66 0F 6C /r	PUNPCKLQDQ *xmm1*, *xmm2/m128*	Interleave low-order quadwords from *xmm1* and *xmm2/m128* into *xmm1*.

Description

Unpacks and interleaves the low-order data elements (bytes, words, doublewords, and quadwords) of the destination operand (first operand) and source operand (second operand) into the destination operand. (Figure 4-11 shows the unpack operation for bytes in 64-bit operands.). The high-order data elements are ignored.

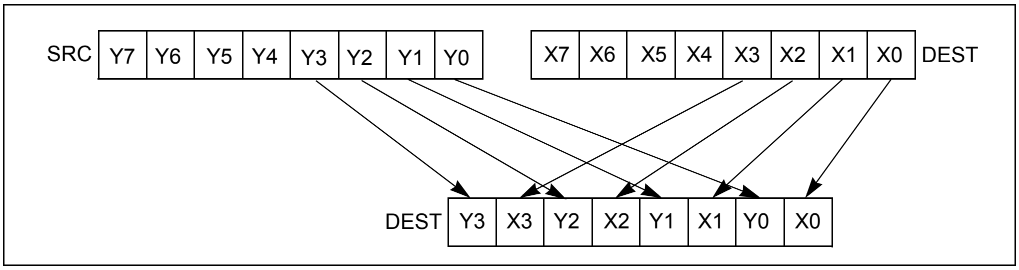


Figure 4-11. PUNPCKLBW Instruction Operation Using 64-bit Operands

The source operand can be an MMX technology register or a 32-bit memory location, or it can be an XMM register or a 128-bit memory location. The destination operand can be an MMX technology register or an XMM register. When the source data comes from a 128-bit memory operand, an implementation may fetch only the appropriate 64 bits; however, alignment to a 16-byte boundary and normal segment checking will still be enforced.

The PUNPCKLBW instruction interleaves the low-order bytes of the source and destination operands, the PUNPCKLWD instruction interleaves the low-order words of the source and destination operands, the PUNPCKLDQ instruction interleaves the low-order doubleword (or doublewords) of the source and destination operands, and the PUNPCKLQDQ instruction interleaves the low-order quadwords of the source and destination operands.

These instructions can be used to convert bytes to words, words to doublewords, doublewords to quadwords, and quadwords to double quadwords, respectively, by placing all 0s in the source operand. Here, if the source operand contains all 0s, the result (stored in the destination operand) contains zero extensions of the high-order data elements from the original value in the destination operand. For example, with the PUNPCKLBW instruction the high-order bytes are zero extended (that is, unpacked into unsigned word integers), and with the PUNPCKLWD instruction, the high-order words are zero extended (unpacked into unsigned doubleword integers).

Operation

```
PUNPCKLBW instruction with 64-bit operands:
        DEST[63..56] ← SRC[31..24];
        DEST[55..48] ← DEST[31..24];
        DEST[47..40] ← SRC[23..16];
        DEST[39..32] ← DEST[23..16];
        DEST[31..24] ← SRC[15..8];
        DEST[23..16] ← DEST[15..8];
        DEST[15..8] ← SRC[7..0];
        DEST[7..0] ← DEST[7..0];

PUNPCKLWD instruction with 64-bit operands:
        DEST[63..48] ← SRC[31..16];
        DEST[47..32] ← DEST[31..16];
        DEST[31..16] ← SRC[15..0];
        DEST[15..0] ← DEST[15..0];

PUNPCKLDQ instruction with 64-bit operands:
        DEST[63..32] ← SRC[31..0];
        DEST[31..0] ← DEST[31..0];

PUNPCKLBW instruction with 128-bit operands:
    DEST[7-0]   ← DEST[7-0];
    DEST[15-8]  ← SRC[7-0];
    DEST[23-16] ← DEST[15-8];
    DEST[31-24] ← SRC[15-8];
    DEST[39-32] ← DEST[23-16];
    DEST[47-40] ← SRC[23-16];
    DEST[55-48] ← DEST[31-24];
    DEST[63-56] ← SRC[31-24];
    DEST[71-64] ← DEST[39-32];
    DEST[79-72] ← SRC[39-32];
    DEST[87-80] ← DEST[47-40];
    DEST[95-88] ← SRC[47-40];
    DEST[103-96]  ← DEST[55-48];
    DEST[111-104] ← SRC[55-48];
    DEST[119-112] ← DEST[63-56];
    DEST[127-120] ← SRC[63-56];

PUNPCKLWD instruction with 128-bit operands:
    DEST[15-0]  ← DEST[15-0];
    DEST[31-16] ← SRC[15-0];
    DEST[47-32] ← DEST[31-16];
    DEST[63-48] ← SRC[31-16];
    DEST[79-64] ← DEST[47-32];
    DEST[95-80] ← SRC[47-32];
    DEST[111-96]  ← DEST[63-48];
    DEST[127-112] ← SRC[63-48];

PUNPCKLDQ instruction with 128-bit operands:
    DEST[31-0] ← DEST[31-0];
    DEST[63-32]  ← SRC[31-0];
    DEST[95-64]  ← DEST[63-32];
    DEST[127-96] ← SRC[63-32];
PUNPCKLQDQ
    DEST[63-0] ← DEST[63-0];
    DEST[127-64] ← SRC[63-0];
```

Intel C/C++ Compiler Intrinsic Equivalents

PUNPCKLBW	__m64 _mm_unpacklo_pi8 (__m64 m1, __m64 m2)
PUNPCKLBW	__m128i _mm_unpacklo_epi8 (__m128i m1, __m128i m2)
PUNPCKLWD	__m64 _mm_unpacklo_pi16 (__m64 m1, __m64 m2)
PUNPCKLWD	__m128i _mm_unpacklo_epi16 (__m128i m1, __m128i m2)
PUNPCKLDQ	__m64 _mm_unpacklo_pi32 (__m64 m1, __m64 m2)
PUNPCKLDQ	__m128i _mm_unpacklo_epi32 (__m128i m1, __m128i m2)
PUNPCKLQDQ	__m128i _mm_unpacklo_epi64 (__m128i m1, __m128i m2)

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.
	(128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.

#UD	If EM in CR0 is set.
	(128-bit operations only) If OSFXSR in CR4 is 0.
	(128-bit operations only) If CPUID feature flag SSE2 is 0.
#NM	If TS in CR0 is set.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#GP(0)	If any part of the operand lies outside of the effective address space from 0 to 0FFFFH.
	(128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
#UD	If EM in CR0 is set.
	(128-bit operations only) If OSFXSR in CR4 is 0.
	(128-bit operations only) If CPUID feature flag SSE2 is 0.
#NM	If TS in CR0 is set.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made.

Numeric Exceptions

None.

PUSH—Push Word or Doubleword Onto the Stack

Opcode	Instruction	Description
FF /6	PUSH *r/m16*	Push *r/m16.*
FF /6	PUSH *r/m32*	Push *r/m32.*
50+*rw*	PUSH *r16*	Push *r16.*
50+*rd*	PUSH *r32*	Push *r32.*
6A	PUSH *imm8*	Push *imm8.*
68	PUSH *imm16*	Push *imm16.*
68	PUSH *imm32*	Push *imm32.*
0E	PUSH CS	Push CS.
16	PUSH SS	Push SS.
1E	PUSH DS	Push DS.
06	PUSH ES	Push ES.
0F A0	PUSH FS	Push FS.
0F A8	PUSH GS	Push GS.

Description

Decrements the stack pointer and then stores the source operand on the top of the stack. The address-size attribute of the stack segment determines the stack pointer size (16 bits or 32 bits), and the operand-size attribute of the current code segment determines the amount the stack pointer is decremented (2 bytes or 4 bytes). For example, if these address- and operand-size attributes are 32, the 32-bit ESP register (stack pointer) is decremented by 4 and, if they are 16, the 16-bit SP register is decremented by 2. (The B flag in the stack segment's segment descriptor determines the stack's address-size attribute, and the D flag in the current code segment's segment descriptor, along with prefixes, determines the operand-size attribute and also the address-size attribute of the source operand.) Pushing a 16-bit operand when the stack address-size attribute is 32 can result in a misaligned the stack pointer (that is, the stack pointer is not aligned on a doubleword boundary).

The PUSH ESP instruction pushes the value of the ESP register as it existed before the instruction was executed. Thus, if a PUSH instruction uses a memory operand in which the ESP register is used as a base register for computing the operand address, the effective address of the operand is computed before the ESP register is decremented.

In the real-address mode, if the ESP or SP register is 1 when the PUSH instruction is executed, the processor shuts down due to a lack of stack space. No exception is generated to indicate this condition.

IA-32 Architecture Compatibility

For IA-32 processors from the Intel 286 on, the PUSH ESP instruction pushes the value of the ESP register as it existed before the instruction was executed. (This is also true in the real-address and virtual-8086 modes.) For the Intel 8086 processor, the PUSH SP instruction pushes the new value of the SP register (that is the value after it has been decremented by 2).

Operation

```
IF StackAddrSize = 32
THEN
    IF OperandSize = 32
        THEN
            ESP ← ESP − 4;
            SS:ESP ← SRC; (* push doubleword *)
        ELSE (* OperandSize = 16*)
            ESP ← ESP − 2;
            SS:ESP ← SRC; (* push word *)
    FI;
ELSE (* StackAddrSize = 16*)
    IF OperandSize = 16
        THEN
            SP ← SP − 2;
             SS:SP ← SRC; (* push word *)
        ELSE (* OperandSize = 32*)
            SP ← SP − 4;
            SS:SP ← SRC; (* push doubleword *)
    FI;
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 is used to access memory and it 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.

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.
	If the new value of the SP or ESP register is outside the stack segment limit.

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.

PUSHA/PUSHAD—Push All General-Purpose Registers

Opcode	Instruction	Description
60	PUSHA	Push AX, CX, DX, BX, original SP, BP, SI, and DI.
60	PUSHAD	Push EAX, ECX, EDX, EBX, original ESP, EBP, ESI, and EDI.

Description

Pushes the contents of the general-purpose registers onto the stack. The registers are stored on the stack in the following order: EAX, ECX, EDX, EBX, EBP, ESP (original value), EBP, ESI, and EDI (if the current operand-size attribute is 32) and AX, CX, DX, BX, SP (original value), BP, SI, and DI (if the operand-size attribute is 16). These instructions perform the reverse operation of the POPA/POPAD instructions. The value pushed for the ESP or SP register is its value before prior to pushing the first register (see the "Operation" section below).

The PUSHA (push all) and PUSHAD (push all double) mnemonics reference the same opcode. The PUSHA instruction is intended for use when the operand-size attribute is 16 and the PUSHAD instruction for when the operand-size attribute is 32. Some assemblers may force the operand size to 16 when PUSHA is used and to 32 when PUSHAD is used. Others may treat these mnemonics as synonyms (PUSHA/PUSHAD) and use the current setting of the operand-size attribute to determine the size of values to be pushed from the stack, regardless of the mnemonic used.

In the real-address mode, if the ESP or SP register is 1, 3, or 5 when the PUSHA/PUSHAD instruction is executed, the processor shuts down due to a lack of stack space. No exception is generated to indicate this condition.

Operation

```
IF OperandSize = 32 (* PUSHAD instruction *)
    THEN
        Temp ← (ESP);
        Push(EAX);
        Push(ECX);
        Push(EDX);
        Push(EBX);
        Push(Temp);
        Push(EBP);
        Push(ESI);
        Push(EDI);
    ELSE (* OperandSize = 16, PUSHA instruction *)
        Temp ← (SP);
        Push(AX);
        Push(CX);
        Push(DX);
        Push(BX);
        Push(Temp);
        Push(BP);
        Push(SI);
        Push(DI);
FI;
```

Flags Affected

None.

Protected Mode Exceptions

#SS(0)	If the starting or ending stack address is outside the stack segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory reference is made while the current privilege level is 3 and alignment checking is enabled.

Real-Address Mode Exceptions

#GP	If the ESP or SP register contains 7, 9, 11, 13, or 15.

Virtual-8086 Mode Exceptions

#GP(0)	If the ESP or SP register contains 7, 9, 11, 13, or 15.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory reference is made while alignment checking is enabled.

PUSHF/PUSHFD—Push EFLAGS Register onto the Stack

Opcode	Instruction	Description
9C	PUSHF	Push lower 16 bits of EFLAGS.
9C	PUSHFD	Push EFLAGS.

Description

Decrements the stack pointer by 4 (if the current operand-size attribute is 32) and pushes the entire contents of the EFLAGS register onto the stack, or decrements the stack pointer by 2 (if the operand-size attribute is 16) and pushes the lower 16 bits of the EFLAGS register (that is, the FLAGS register) onto the stack. (These instructions reverse the operation of the POPF/POPFD instructions.) When copying the entire EFLAGS register to the stack, the VM and RF flags (bits 16 and 17) are not copied; instead, the values for these flags are cleared in the EFLAGS image stored on the stack. See the section titled “EFLAGS Register” in Chapter 3 of the *IA-32 Intel Architecture Software Developer’s Manual, Volume 1*, for information about the EFLAGS registers.

The PUSHF (push flags) and PUSHFD (push flags double) mnemonics reference the same opcode. The PUSHF instruction is intended for use when the operand-size attribute is 16 and the PUSHFD instruction for when the operand-size attribute is 32. Some assemblers may force the operand size to 16 when PUSHF is used and to 32 when PUSHFD is used. Others may treat these mnemonics as synonyms (PUSHF/PUSHFD) and use the current setting of the operand-size attribute to determine the size of values to be pushed from the stack, regardless of the mnemonic used.

When in virtual-8086 mode and the I/O privilege level (IOPL) is less than 3, the PUSHF/PUSHFD instruction causes a general protection exception (#GP).

In the real-address mode, if the ESP or SP register is 1, 3, or 5 when the PUSHA/PUSHAD instruction is executed, the processor shuts down due to a lack of stack space. No exception is generated to indicate this condition.

Operation

```
IF (PE=0) OR (PE=1 AND ((VM=0) OR (VM=1 AND IOPL=3)))
(* Real-Address Mode, Protected mode, or Virtual-8086 mode with IOPL equal to 3 *)
   THEN
        IF OperandSize = 32
             THEN
                  push(EFLAGS AND 00FCFFFFH);
                  (* VM and RF EFLAG bits are cleared in image stored on the stack*)
             ELSE
                  push(EFLAGS); (* Lower 16 bits only *)
        FI;
```

```
    ELSE (* In Virtual-8086 Mode with IOPL less than 3 *)
        #GP(0); (* Trap to virtual-8086 monitor *)
FI;
```

Flags Affected

None.

Protected Mode Exceptions

#SS(0)	If the new value of the ESP register is outside the stack segment boundary.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory reference is made while the current privilege level is 3 and alignment checking is enabled.

Real-Address Mode Exceptions

None.

Virtual-8086 Mode Exceptions

#GP(0)	If the I/O privilege level is less than 3.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory reference is made while alignment checking is enabled.

PXOR—Logical Exclusive OR

Opcode	Instruction	Description
0F EF /r	PXOR *mm, mm/m64*	Bitwise XOR of *mm/m64* and *mm*.
66 0F EF /r	PXOR *xmm1*, *xmm2/m128*	Bitwise XOR of *xmm2/m128* and *xmm1*.

Description

Performs a bitwise logical exclusive-OR (XOR) operation on the source operand (second operand) and the destination operand (first operand) and stores the result in the destination operand. The source operand can be an MMX technology register or a 64-bit memory location or it can be an XMM register or a 128-bit memory location. The destination operand can be an MMX technology register or an XMM register. Each bit of the result is 1 if the corresponding bits of the two operands are different; each bit is 0 if the corresponding bits of the operands are the same.

Operation

```
DEST ← DEST XOR SRC;
```

Intel C/C++ Compiler Intrinsic Equivalent

PXOR	__m64 _mm_xor_si64 (__m64 m1, __m64 m2)
PXOR	__m128i _mm_xor_si128 (__m128i a, __m128i b)

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.
	(128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If EM in CR0 is set.
	(128-bit operations only) If OSFXSR in CR4 is 0.
	(128-bit operations only) If CPUID feature flag SSE2 is 0.
#NM	If TS in CR0 is set.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.

#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#GP(0)	(128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
	If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set.
	(128-bit operations only) If OSFXSR in CR4 is 0.
	(128-bit operations only) If CPUID feature flag SSE2 is 0.
#NM	If TS in CR0 is set.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made.

Numeric Exceptions

None.

RCL/RCR/ROL/ROR—Rotate

Opcode	Instruction	Description
D0 /2	RCL *r/m8*, 1	Rotate 9 bits (CF, *r/m8*) left once.
D2 /2	RCL *r/m8*, CL	Rotate 9 bits (CF, *r/m8*) left CL times.
C0 /2 *ib*	RCL *r/m8, imm8*	Rotate 9 bits (CF, *r/m8*) left *imm8* times.
D1 /2	RCL *r/m16*, 1	Rotate 17 bits (CF, *r/m16*) left once.
D3 /2	RCL *r/m16*, CL	Rotate 17 bits (CF, *r/m16*) left CL times.
C1 /2 *ib*	RCL *r/m16, imm8*	Rotate 17 bits (CF, *r/m16*) left *imm8* times.
D1 /2	RCL *r/m32*, 1	Rotate 33 bits (CF, *r/m32*) left once.
D3 /2	RCL *r/m32*, CL	Rotate 33 bits (CF, *r/m32*) left CL times.
C1 /2 *ib*	RCL *r/m32,i mm8*	Rotate 33 bits (CF, *r/m32*) left *imm8* times.
D0 /3	RCR *r/m8*, 1	Rotate 9 bits (CF, *r/m8*) right once.
D2 /3	RCR *r/m8*, CL	Rotate 9 bits (CF, *r/m8*) right CL times.
C0 /3 *ib*	RCR *r/m8, imm8*	Rotate 9 bits (CF, *r/m8*) right *imm8* times.
D1 /3	RCR *r/m16*, 1	Rotate 17 bits (CF, *r/m16*) right once.
D3 /3	RCR *r/m16*, CL	Rotate 17 bits (CF, *r/m16*) right CL times.
C1 /3 *ib*	RCR *r/m16, imm8*	Rotate 17 bits (CF, *r/m16*) right *imm8* times.
D1 /3	RCR *r/m32*, 1	Rotate 33 bits (CF, *r/m32*) right once.
D3 /3	RCR *r/m32*, CL	Rotate 33 bits (CF, *r/m32*) right CL times.
C1 /3 *ib*	RCR *r/m32, imm8*	Rotate 33 bits (CF, *r/m32*) right *imm8* times.
D0 /0	ROL *r/m8*, 1	Rotate 8 bits *r/m8* left once.
D2 /0	ROL *r/m8*, CL	Rotate 8 bits *r/m8* left CL times.
C0 /0 *ib*	ROL *r/m8, imm8*	Rotate 8 bits *r/m8* left *imm8* times.
D1 /0	ROL *r/m16*, 1	Rotate 16 bits *r/m16* left once.
D3 /0	ROL *r/m16*, CL	Rotate 16 bits *r/m16* left CL times.
C1 /0 *ib*	ROL *r/m16, imm8*	Rotate 16 bits *r/m16* left *imm8* times.
D1 /0	ROL *r/m32*, 1	Rotate 32 bits *r/m32* left once.
D3 /0	ROL *r/m32*, CL	Rotate 32 bits *r/m32* left CL times.
C1 /0 *ib*	ROL *r/m32, imm8*	Rotate 32 bits *r/m32* left *imm8* times.
D0 /1	ROR *r/m8*, 1	Rotate 8 bits *r/m8* right once.
D2 /1	ROR *r/m8*, CL	Rotate 8 bits *r/m8* right CL times.
C0 /1 *ib*	ROR *r/m8, imm8*	Rotate 8 bits *r/m16* right *imm8* times.
D1 /1	ROR *r/m16*, 1	Rotate 16 bits *r/m16* right once.
D3 /1	ROR *r/m16*, CL	Rotate 16 bits *r/m16* right CL times.
C1 /1 *ib*	ROR *r/m16, imm8*	Rotate 16 bits *r/m16* right *imm8* times.
D1 /1	ROR *r/m32*, 1	Rotate 32 bits *r/m32* right once.
D3 /1	ROR *r/m32*, CL	Rotate 32 bits *r/m32* right CL times.
C1 /1 *ib*	ROR *r/m32, imm8*	Rotate 32 bits *r/m32* right *imm8* times.

Description

Shifts (rotates) the bits of the first operand (destination operand) the number of bit positions specified in the second operand (count operand) and stores the result in the destination operand. The destination operand can be a register or a memory location; the count operand is an unsigned integer that can be an immediate or a value in the CL register. The processor restricts the count to a number between 0 and 31 by masking all the bits in the count operand except the 5 least-significant bits.

The rotate left (ROL) and rotate through carry left (RCL) instructions shift all the bits toward more-significant bit positions, except for the most-significant bit, which is rotated to the least-significant bit location (see Figure 7-11 in the *IA-32 Intel Architecture Software Developer's Manual, Volume 1*). The rotate right (ROR) and rotate through carry right (RCR) instructions shift all the bits toward less significant bit positions, except for the least-significant bit, which is rotated to the most-significant bit location (see Figure 7-11 in the *IA-32 Intel Architecture Software Developer's Manual, Volume 1*).

The RCL and RCR instructions include the CF flag in the rotation. The RCL instruction shifts the CF flag into the least-significant bit and shifts the most-significant bit into the CF flag (see Figure 7-11 in the *IA-32 Intel Architecture Software Developer's Manual, Volume 1*). The RCR instruction shifts the CF flag into the most-significant bit and shifts the least-significant bit into the CF flag (see Figure 7-11 in the *IA-32 Intel Architecture Software Developer's Manual, Volume 1*). For the ROL and ROR instructions, the original value of the CF flag is not a part of the result, but the CF flag receives a copy of the bit that was shifted from one end to the other.

The OF flag is defined only for the 1-bit rotates; it is undefined in all other cases (except that a zero-bit rotate does nothing, that is affects no flags). For left rotates, the OF flag is set to the exclusive OR of the CF bit (after the rotate) and the most-significant bit of the result. For right rotates, the OF flag is set to the exclusive OR of the two most-significant bits of the result.

IA-32 Architecture Compatibility

The 8086 does not mask the rotation count. However, all other IA-32 processors (starting with the Intel 286 processor) do mask the rotation count to 5 bits, resulting in a maximum count of 31. This masking is done in all operating modes (including the virtual-8086 mode) to reduce the maximum execution time of the instructions.

Operation

```
(* RCL and RCR instructions *)
SIZE ← OperandSize
CASE (determine count) OF
    SIZE ← 8:   tempCOUNT ← (COUNT AND 1FH) MOD 9;
    SIZE ← 16:  tempCOUNT ← (COUNT AND 1FH) MOD 17;
    SIZE ← 32:  tempCOUNT ← COUNT AND 1FH;
ESAC;

(* RCL instruction operation *)
WHILE (tempCOUNT ≠ 0)
    DO
```

```
            tempCF ← MSB(DEST);
            DEST ← (DEST ∗ 2) + CF;
            CF ← tempCF;
            tempCOUNT ← tempCOUNT – 1;
    OD;
ELIHW;
IF COUNT = 1
    THEN OF ← MSB(DEST) XOR CF;
    ELSE OF is undefined;
FI;
(* RCR instruction operation *)
IF COUNT = 1
    THEN OF ← MSB(DEST) XOR CF;
    ELSE OF is undefined;
FI;
WHILE (tempCOUNT ≠ 0)
    DO
            tempCF ← LSB(SRC);
            DEST ← (DEST / 2) + (CF * 2^SIZE);
            CF ← tempCF;
            tempCOUNT ← tempCOUNT – 1;
    OD;
(* ROL and ROR instructions *)
SIZE ← OperandSize
CASE (determine count) OF
    SIZE ← 8:     tempCOUNT ← COUNT MOD 8;
    SIZE ← 16:    tempCOUNT ← COUNT MOD 16;
    SIZE ← 32:    tempCOUNT ← COUNT MOD 32;
ESAC;
(* ROL instruction operation *)
WHILE (tempCOUNT ≠ 0)
    DO
            tempCF ← MSB(DEST);
            DEST ← (DEST ∗ 2) + tempCF;
            tempCOUNT ← tempCOUNT – 1;
    OD;
ELIHW;
CF ← LSB(DEST);
IF COUNT = 1
    THEN OF ← MSB(DEST) XOR CF;
    ELSE OF is undefined;
FI;
(* ROR instruction operation *)
WHILE (tempCOUNT ≠ 0)
    DO
            tempCF ← LSB(SRC);
            DEST ← (DEST / 2) + (tempCF ∗ 2^SIZE);
```

```
        tempCOUNT ← tempCOUNT – 1;
    OD;
ELIHW;
CF ← MSB(DEST);
IF COUNT = 1
    THEN OF ← MSB(DEST) XOR MSB − 1(DEST);
    ELSE OF is undefined;
FI;
```

Flags Affected

The CF flag contains the value of the bit shifted into it. The OF flag is affected only for single-bit rotates (see “Description” above); it is undefined for multi-bit rotates. The SF, ZF, AF, and PF flags are not affected.

Protected Mode Exceptions

#GP(0)	If the source operand is located in a non-writable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a null segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

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.

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.

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

Opcode	**Instruction**	**Description**
0F 53 /r	RCPPS *xmm1*, *xmm2/m128*	Compute the approximate reciprocals of the packed single-precision floating-point values in *xmm2/m128* and store the results in *xmm1*.

Description

Performs an SIMD computation of the approximate reciprocals of the four packed single-precision floating-point values in the source operand (second operand) stores the packed single-precision floating-point results in the destination operand. The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. See Figure 10-5 in the *IA-32 Intel Architecture Software Developer's Manual, Volume 1* for an illustration of an SIMD single-precision floating-point operation.

The relative error for this approximation is:

$|\text{Relative Error}| \leq 1.5 * 2^{-12}$

The RCPPS instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an ∞ of the sign of the source value is returned. A denormal source value is treated as a 0.0 (of the same sign). Tiny results are always flushed to 0.0, with the sign of the operand. (Input values greater than or equal to $|1.11111111110100000000000B*2^{125}|$ are guaranteed to not produce tiny results; input values less than or equal to $|1.00000000000110000000001B*2^{126}|$ are guaranteed to produce tiny results, which are in turn flushed to 0.0; and input values in between this range may or may not produce tiny results, depending on the implementation.) When a source value is an SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned.

Operation

```
DEST[31-0] ← APPROXIMATE(1.0/(SRC[31-0]));
DEST[63-32] ← APPROXIMATE(1.0/(SRC[63-32]));
DEST[95-64] ← APPROXIMATE(1.0/(SRC[95-64]));
DEST[127-96] ← APPROXIMATE(1.0/(SRC[127-96]));
```

Intel C/C++ Compiler Intrinsic Equivalent

```
RCCPS          __m128 _mm_rcp_ps(__m128 a)
```

SIMD Floating-Point Exceptions

None.

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
	If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS in CR0 is set.
#UD	If EM in CR0 is set.
	If OSFXSR in CR4 is 0.
	If CPUID feature flag SSE is 0.

Real-Address Mode Exceptions

#GP(0)	If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If TS in CR0 is set.
#UD	If EM in CR0 is set.
	If OSFXSR in CR4 is 0.
	If CPUID feature flag SSE is 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.

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

Opcode	Instruction	Description
F3 0F 53 /r	RCPSS *xmm1*, xmm2/m32	Compute the approximate reciprocal of the scalar single-precision floating-point value in *xmm2/m32* and store the result in *xmm1*.

Description

Computes of an approximate reciprocal of the low single-precision floating-point value in the source operand (second operand) and stores the single-precision floating-point result in the destination operand. The source operand can be an XMM register or a 32-bit memory location. The destination operand is an XMM register. The three high-order doublewords of the destination operand remain unchanged. See Figure 10-6 in the *IA-32 Intel Architecture Software Developer's Manual, Volume 1* for an illustration of a scalar single-precision floating-point operation.

The relative error for this approximation is:

$|\text{Relative Error}| \leq 1.5 * 2^{-12}$

The RCPSS instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an ∞ of the sign of the source value is returned. A denormal source value is treated as a 0.0 (of the same sign). Tiny results are always flushed to 0.0, with the sign of the operand. (Input values greater than or equal to $|1.11111111110100000000000B*2^{125}|$ are guaranteed to not produce tiny results; input values less than or equal to $|1.00000000000110000000001B*2^{126}|$ are guaranteed to produce tiny results, which are in turn flushed to 0.0; and input values in between this range may or may not produce tiny results, depending on the implementation.) When a source value is an SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned.

Operation

```
DEST[31-0] ← APPROX (1.0/(SRC[31-0]));
* DEST[127-32] remains unchanged *;
```

Intel C/C++ Compiler Intrinsic Equivalent

```
RCPSS          __m128 _mm_rcp_ss(__m128 a)
```

SIMD Floating-Point Exceptions

None.

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.

#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS in CR0 is set.
#UD	If EM in CR0 is set.
	If OSFXSR in CR4 is 0.
	If CPUID feature flag SSE is 0.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

GP(0)	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If TS in CR0 is set.
#UD	If EM in CR0 is set.
	If OSFXSR in CR4 is 0.
	If CPUID feature flag SSE is 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
#AC(0)	For unaligned memory reference.

RDMSR—Read from Model Specific Register

Opcode	Instruction	Description
0F 32	RDMSR	Load MSR specified by ECX into EDX:EAX.

Description

Loads the contents of a 64-bit model specific register (MSR) specified in the ECX register into registers EDX:EAX. The input value loaded into the ECX register is the address of the MSR to be read. The EDX register is loaded with the high-order 32 bits of the MSR and the EAX register is loaded with the low-order 32 bits. If fewer than 64 bits are implemented in the MSR being read, the values returned to EDX:EAX in unimplemented bit locations are undefined.

This instruction must be executed at privilege level 0 or in real-address mode; otherwise, a general protection exception #GP(0) will be generated. Specifying a reserved or unimplemented MSR address in ECX will also cause a general protection exception.

The MSRs control functions for testability, execution tracing, performance-monitoring, and machine check errors. Appendix B, *Model-Specific Registers (MSRs)*, in the *IA-32 Intel Architecture Software Developer's Manual, Volume 3*, lists all the MSRs that can be read with this instruction and their addresses. Note that each processor family has its own set of MSRs.

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

IA-32 Architecture Compatibility

The MSRs and the ability to read them with the RDMSR 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

```
EDX:EAX ← MSR[ECX];
```

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.

Real-Address Mode Exceptions

#GP	If the value in ECX specifies a reserved or unimplemented MSR address.

Virtual-8086 Mode Exceptions

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

RDPMC—Read Performance-Monitoring Counters

Opcode	Instruction	Description
0F 33	RDPMC	Read performance-monitoring counter specified by ECX into EDX:EAX.

Description

Loads the contents of the 40-bit performance-monitoring counter specified in the ECX register into registers EDX:EAX. The EDX register is loaded with the high-order 8 bits of the counter and the EAX register is loaded with the low-order 32 bits. The counter to be read is specified with an unsigned integer placed in the ECX register. The P6 family processors and Pentium processors with MMX technology have two performance-monitoring counters (0 and 1), which are specified by placing 0000H or 0001H, respectively, in the ECX register. The Pentium 4 and Intel Xeon processors have 18 counters (0 through 17), which are specified with 0000H through 0011H, respectively

The Pentium 4 and Intel Xeon processors also support “fast” (32-bit) and “slow” (40-bit) reads of the performance counters, selected with bit 31 of the ECX register. If bit 31 is set, the RDPMC instruction reads only the low 32 bits of the selected performance counter; if bit 31 is clear, all 40 bits of the counter are read. The 32-bit counter result is returned in the EAX register, and the EDX register is set to 0. A 32-bit read executes faster on a Pentium 4 or Intel Xeon processor than a full 40-bit read.

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

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

The performance-monitoring counters are event counters that can be programmed to count events such as the number of instructions decoded, number of interrupts received, or number of cache loads. Appendix A, *Performance-Monitoring Events*, in the *IA-32 Intel Architecture Software Developer's Manual, Volume 3*, lists the events that can be counted for the Pentium 4, Intel Xeon, and earlier IA-32 processors.

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

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

The RDPMC instruction can execute in 16-bit addressing mode or virtual-8086 mode; however, the full contents of the ECX register are used to select the counter, and the event count is stored in the full EAX and EDX registers.

The RDPMC instruction was introduced into the IA-32 Architecture in the Pentium Pro processor and the Pentium processor with MMX technology. The earlier Pentium processors have performance-monitoring counters, but they must be read with the RDMSR instruction.

Operation

```
(* P6 family processors and Pentium processor with MMX technology *)
IF (ECX=0 OR 1) AND ((CR4.PCE=1) OR (CPL=0) OR (CR0.PE=0))
    THEN
        EAX ← PMC(ECX)[31:0];
        EDX ← PMC(ECX)[39:32];
    ELSE (* ECX is not 0 or 1 or CR4.PCE is 0 and CPL is 1, 2, or 3 and CR0.PE is 1*)
        #GP(0); FI;
(* Pentium 4 and Intel Xeon processor *)
IF (ECX[30:0]=0 ... 17) AND ((CR4.PCE=1) OR (CPL=0) OR (CR0.PE=0))
    THEN IF ECX[31] = 0
        THEN
            EAX ← PMC(ECX[30:0])[31:0]; (* 40-bit read *);
            EDX ← PMC(ECX[30:0])[39:32];
        ELSE IF ECX[31] = 1
            THEN
                EAX ← PMC(ECX[30:0])[31:0]; (* 32-bit read *);
                EDX ← 0;
            FI;
        FI;
    ELSE (* ECX[30:0] is not 0...17 or CR4.PCE is 0 and CPL is 1, 2, or 3 and CR0.PE is 1 *)
        #GP(0); FI;
```

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If the current privilege level is not 0 and the PCE flag in the CR4 register is clear.
	(P6 family processors and Pentium processors with MMX technology) If the value in the ECX register is not 0 or 1.
	(Pentium 4 and Intel Xeon processors) If the value in ECX[30:0] is not within the range of 0 through 17.

Real-Address Mode Exceptions

#GP	(P6 family processors and Pentium processors with MMX technology) If the value in the ECX register is not 0 or 1.
	(Pentium 4 and Intel Xeon processors) If the value in ECX[30:0] is not within the range of 0 through 17.

Virtual-8086 Mode Exceptions

#GP(0)	If the PCE flag in the CR4 register is clear.
	(P6 family processors and Pentium processors with MMX technology) If the value in the ECX register is not 0 or 1.
	(Pentium 4 and Intel Xeon processors) If the value in ECX[30:0] is not within the range of 0 through 17.

RDTSC—Read Time-Stamp Counter

Opcode	Instruction	Description
0F 31	RDTSC	Read time-stamp counter into EDX:EAX.

Description

Loads the current value of the processor's time-stamp counter into the EDX:EAX registers. The time-stamp counter is contained in a 64-bit MSR. The high-order 32 bits of the MSR are loaded into the EDX register, and the low-order 32 bits are loaded into the EAX register. The processor monotonically increments the time-stamp counter MSR every clock cycle and resets it to 0 whenever the processor is reset. See "Time Stamp Counter" in Chapter 15 of the *IA-32 Intel Architecture Software Developer's Manual, Volume 3* for specific details of the time stamp counter behavior.

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

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

The RDTSC instruction is not a serializing instruction. Thus, it does not necessarily wait until all previous instructions have been executed before reading the counter. Similarly, subsequent instructions may begin execution before the read operation is performed.

This instruction was introduced into the IA-32 Architecture in the Pentium processor.

Operation

```
IF (CR4.TSD=0) OR (CPL=0) OR (CR0.PE=0)
    THEN
        EDX:EAX ← TimeStampCounter;
    ELSE (* CR4.TSD is 1 and CPL is 1, 2, or 3 and CR0.PE is 1 *)
        #GP(0)
FI;
```

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If the TSD flag in register CR4 is set and the CPL is greater than 0.

Real-Address Mode Exceptions

None.

Virtual-8086 Mode Exceptions

#GP(0)	If the TSD flag in register CR4 is set.

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

Opcode	Instruction	Description
F3 6C	REP INS *m8*, DX	Input (E)CX bytes from port DX into ES:[(E)DI].
F3 6D	REP INS *m16*, DX	Input (E)CX words from port DX into ES:[(E)DI].
F3 6D	REP INS *m32*, DX	Input (E)CX doublewords from port DX into ES:[(E)DI].
F3 A4	REP MOVS *m8, m8*	Move (E)CX bytes from DS:[(E)SI] to ES:[(E)DI].
F3 A5	REP MOVS *m16, m16*	Move (E)CX words from DS:[(E)SI] to ES:[(E)DI].
F3 A5	REP MOVS *m32, m32*	Move (E)CX doublewords from DS:[(E)SI] to ES:[(E)DI].
F3 6E	REP OUTS DX, *r/m8*	Output (E)CX bytes from DS:[(E)SI] to port DX.
F3 6F	REP OUTS DX, *r/m16*	Output (E)CX words from DS:[(E)SI] to port DX.
F3 6F	REP OUTS DX, *r/m32*	Output (E)CX doublewords from DS:[(E)SI] to port DX.
F3 AC	REP LODS AL	Load (E)CX bytes from DS:[(E)SI] to AL.
F3 AD	REP LODS AX	Load (E)CX words from DS:[(E)SI] to AX.
F3 AD	REP LODS EAX	Load (E)CX doublewords from DS:[(E)SI] to EAX.
F3 AA	REP STOS *m8*	Fill (E)CX bytes at ES:[(E)DI] with AL.
F3 AB	REP STOS *m16*	Fill (E)CX words at ES:[(E)DI] with AX.
F3 AB	REP STOS *m32*	Fill (E)CX doublewords at ES:[(E)DI] with EAX.
F3 A6	REPE CMPS *m8, m8*	Find nonmatching bytes in ES:[(E)DI] and DS:[(E)SI].
F3 A7	REPE CMPS *m16, m16*	Find nonmatching words in ES:[(E)DI] and DS:[(E)SI].
F3 A7	REPE CMPS *m32, m32*	Find nonmatching doublewords in ES:[(E)DI] and DS:[(E)SI].
F3 AE	REPE SCAS *m8*	Find non-AL byte starting at ES:[(E)DI].
F3 AF	REPE SCAS *m16*	Find non-AX word starting at ES:[(E)DI].
F3 AF	REPE SCAS *m32*	Find non-EAX doubleword starting at ES:[(E)DI].
F2 A6	REPNE CMPS *m8, m8*	Find matching bytes in ES:[(E)DI] and DS:[(E)SI].
F2 A7	REPNE CMPS *m16, m16*	Find matching words in ES:[(E)DI] and DS:[(E)SI].
F2 A7	REPNE CMPS *m32, m32*	Find matching doublewords in ES:[(E)DI] and DS:[(E)SI].
F2 AE	REPNE SCAS *m8*	Find AL, starting at ES:[(E)DI].
F2 AF	REPNE SCAS *m16*	Find AX, starting at ES:[(E)DI].
F2 AF	REPNE SCAS *m32*	Find EAX, starting at ES:[(E)DI].

Description

Repeats a string instruction the number of times specified in the count register ((E)CX) or until the indicated condition of the ZF flag is no longer met. The REP (repeat), REPE (repeat while equal), REPNE (repeat while not equal), REPZ (repeat while zero), and REPNZ (repeat while not zero) mnemonics are prefixes that can be added to one of the string instructions. The REP prefix can be added to the INS, OUTS, MOVS, LODS, and STOS instructions, and the REPE, REPNE, REPZ, and REPNZ prefixes can be added to the CMPS and SCAS instructions. (The REPZ and REPNZ prefixes are synonymous forms of the REPE and REPNE prefixes, respectively.) The behavior of the REP prefix is undefined when used with non-string instructions.

The REP prefixes apply only to one string instruction at a time. To repeat a block of instructions, use the LOOP instruction or another looping construct.

All of these repeat prefixes cause the associated instruction to be repeated until the count in register (E)CX is decremented to 0 (see Table 4-1). (If the current address-size attribute is 32, register ECX is used as a counter, and if the address-size attribute is 16, the CX register is used.) The REPE, REPNE, REPZ, and REPNZ prefixes also check the state of the ZF flag after each iteration and terminate the repeat loop if the ZF flag is not in the specified state. When both termination conditions are tested, the cause of a repeat termination can be determined either by testing the (E)CX register with a JECXZ instruction or by testing the ZF flag with a JZ, JNZ, and JNE instruction.

Table 4-1. Repeat Prefixes

Repeat Prefix	Termination Condition 1	Termination Condition 2
REP	ECX=0	None
REPE/REPZ	ECX=0	ZF=0
REPNE/REPNZ	ECX=0	ZF=1

When the REPE/REPZ and REPNE/REPNZ prefixes are used, the ZF flag does not require initialization because both the CMPS and SCAS instructions affect the ZF flag according to the results of the comparisons they make.

A repeating string operation can be suspended by an exception or interrupt. When this happens, the state of the registers is preserved to allow the string operation to be resumed upon a return from the exception or interrupt handler. The source and destination registers point to the next string elements to be operated on, the EIP register points to the string instruction, and the ECX register has the value it held following the last successful iteration of the instruction. This mechanism allows long string operations to proceed without affecting the interrupt response time of the system.

When a fault occurs during the execution of a CMPS or SCAS instruction that is prefixed with REPE or REPNE, the EFLAGS value is restored to the state prior to the execution of the instruction. Since the SCAS and CMPS instructions do not use EFLAGS as an input, the processor can resume the instruction after the page fault handler.

Use the REP INS and REP OUTS instructions with caution. Not all I/O ports can handle the rate at which these instructions execute.

A REP STOS instruction is the fastest way to initialize a large block of memory.

Operation

```
IF AddressSize = 16
    THEN
        use CX for CountReg;
    ELSE (* AddressSize = 32 *)
        use ECX for CountReg;
```

```
FI;
WHILE CountReg ≠ 0
   DO
      service pending interrupts (if any);
      execute associated string instruction;
      CountReg ← CountReg – 1;
      IF CountReg = 0
         THEN exit WHILE loop
      FI;
      IF (repeat prefix is REPZ or REPE) AND (ZF=0)
      OR (repeat prefix is REPNZ or REPNE) AND (ZF=1)
         THEN exit WHILE loop
      FI;
   OD;
```

Flags Affected

None; however, the CMPS and SCAS instructions do set the status flags in the EFLAGS register.

Exceptions (All Operating Modes)

None; however, exceptions can be generated by the instruction a repeat prefix is associated with.

RET—Return from Procedure

Opcode	Instruction	Description
C3	RET	Near return to calling procedure.
CB	RET	Far return to calling procedure.
C2 *iw*	RET *imm16*	Near return to calling procedure and pop *imm16* bytes from stack.
CA *iw*	RET *imm16*	Far return to calling procedure and pop *imm16* bytes from stack.

Description

Transfers program control to a return address located on the top of the stack. The address is usually placed on the stack by a CALL instruction, and the return is made to the instruction that follows the CALL instruction.

The optional source operand specifies the number of stack bytes to be released after the return address is popped; the default is none. This operand can be used to release parameters from the stack that were passed to the called procedure and are no longer needed. It must be used when the CALL instruction used to switch to a new procedure uses a call gate with a non-zero word count to access the new procedure. Here, the source operand for the RET instruction must specify the same number of bytes as is specified in the word count field of the call gate.

The RET instruction can be used to execute three different types of returns:

- Near return—A return to a calling procedure within the current code segment (the segment currently pointed to by the CS register), sometimes referred to as an intrasegment return.
- Far return—A return to a calling procedure located in a different segment than the current code segment, sometimes referred to as an intersegment return.
- Inter-privilege-level far return—A far return to a different privilege level than that of the currently executing program or procedure.

The inter-privilege-level return type can only be executed in protected mode. See the section titled "Calling Procedures Using Call and RET" in Chapter 6 of the *IA-32 Intel Architecture Software Developer's Manual, Volume 1*, for detailed information on near, far, and inter-privilege-level returns.

When executing a near return, the processor pops the return instruction pointer (offset) from the top of the stack into the EIP register and begins program execution at the new instruction pointer. The CS register is unchanged.

When executing a far return, the processor pops the return instruction pointer from the top of the stack into the EIP register, then pops the segment selector from the top of the stack into the CS register. The processor then begins program execution in the new code segment at the new instruction pointer.

The mechanics of an inter-privilege-level far return are similar to an intersegment return, except that the processor examines the privilege levels and access rights of the code and stack segments being returned to determine if the control transfer is allowed to be made. The DS, ES, FS, and GS segment registers are cleared by the RET instruction during an inter-privilege-level return if they refer to segments that are not allowed to be accessed at the new privilege level. Since a stack switch also occurs on an inter-privilege level return, the ESP and SS registers are loaded from the stack.

If parameters are passed to the called procedure during an inter-privilege level call, the optional source operand must be used with the RET instruction to release the parameters on the return. Here, the parameters are released both from the called procedure's stack and the calling procedure's stack (that is, the stack being returned to).

Operation

```
(* Near return *)
IF instruction = near return
   THEN;
      IF OperandSize = 32
         THEN
            IF top 12 bytes of stack not within stack limits THEN #SS(0); FI;
            EIP ← Pop();
         ELSE (* OperandSize = 16 *)
            IF top 6 bytes of stack not within stack limits
               THEN #SS(0)
            FI;
            tempEIP ← Pop();
            tempEIP ← tempEIP AND 0000FFFFH;
            IF tempEIP not within code segment limits THEN #GP(0); FI;
            EIP ← tempEIP;
      FI;
   IF instruction has immediate operand
      THEN IF StackAddressSize=32
         THEN
            ESP ← ESP + SRC; (* release parameters from stack *)
         ELSE (* StackAddressSize=16 *)
            SP ← SP + SRC; (* release parameters from stack *)
      FI;
   FI;

(* Real-address mode or virtual-8086 mode *)
IF ((PE = 0) OR (PE = 1 AND VM = 1)) AND instruction = far return
   THEN;

      IF OperandSize = 32
         THEN
            IF top 12 bytes of stack not within stack limits THEN #SS(0); FI;
            EIP ← Pop();
            CS ← Pop(); (* 32-bit pop, high-order 16 bits discarded *)
         ELSE (* OperandSize = 16 *)
            IF top 6 bytes of stack not within stack limits THEN #SS(0); FI;
```

```
                    tempEIP ← Pop();
                    tempEIP ← tempEIP AND 0000FFFFH;
                    IF tempEIP not within code segment limits THEN #GP(0); FI;
                    EIP ← tempEIP;
                    CS ← Pop(); (* 16-bit pop *)
          FI;
      IF instruction has immediate operand
          THEN
               SP ← SP + (SRC AND FFFFH); (* release parameters from stack *)
      FI;
FI;

(* Protected mode, not virtual-8086 mode *)
IF (PE = 1 AND VM = 0) AND instruction = far RET
      THEN
          IF OperandSize = 32
               THEN
                    IF second doubleword on stack is not within stack limits THEN #SS(0); FI;
               ELSE (* OperandSize = 16 *)
                    IF second word on stack is not within stack limits THEN #SS(0); FI;
          FI;
      IF return code segment selector is null THEN GP(0); FI;
      IF return code segment selector addrsses descriptor beyond diescriptor table limit
          THEN GP(selector; FI;
      Obtain descriptor to which return code segment selector points from descriptor table
      IF return code segment descriptor is not a code segment THEN #GP(selector); FI;
      if return code segment selector RPL < CPL THEN #GP(selector); FI;
      IF return code segment descriptor is conforming
          AND return code segment DPL > return code segment selector RPL
               THEN #GP(selector); FI;
      IF return code segment descriptor is not present THEN #NP(selector); FI:
      IF return code segment selector RPL > CPL
          THEN GOTO RETURN-OUTER-PRIVILEGE-LEVEL;
          ELSE GOTO RETURN-TO-SAME-PRIVILEGE-LEVEL
      FI;
END;FI;

RETURN-SAME-PRIVILEGE-LEVEL:
      IF the return instruction pointer is not within ther return code segment limit
          THEN #GP(0);
      FI;
      IF OperandSize=32
          THEN
               EIP ← Pop();
               CS ← Pop(); (* 32-bit pop, high-order 16 bits discarded *)
               ESP ← ESP + SRC; (* release parameters from stack *)
          ELSE (* OperandSize=16 *)
```

```
            EIP ← Pop();
            EIP ← EIP AND 0000FFFFH;
            CS ← Pop(); (* 16-bit pop *)
            ESP ← ESP + SRC; (* release parameters from stack *)
    FI;

RETURN-OUTER-PRIVILEGE-LEVEL:
    IF top (16 + SRC) bytes of stack are not within stack limits (OperandSize=32)
        OR top (8 + SRC) bytes of stack are not within stack limits (OperandSize=16)
            THEN #SS(0); FI;
    FI;
    Read return segment selector;
    IF stack segment selector is null THEN #GP(0); FI;
    IF return stack segment selector index is not within its descriptor table limits
            THEN #GP(selector); FI;
    Read segment descriptor pointed to by return segment selector;
    IF stack segment selector RPL ≠ RPL of the return code segment selector
        OR stack segment is not a writable data segment
        OR stack segment descriptor DPL ≠ RPL of the return code segment selector
            THEN #GP(selector); FI;
        IF stack segment not present THEN #SS(StackSegmentSelector); FI;
    IF the return instruction pointer is not within the return code segment limit THEN #GP(0); FI:
     CPL ← ReturnCodeSegmentSelector(RPL);
    IF OperandSize=32
        THEN
            EIP ← Pop();
            CS ← Pop(); (* 32-bit pop, high-order 16 bits discarded *)
             (* segment descriptor information also loaded *)
            CS(RPL) ← CPL;
            ESP ← ESP + SRC; (* release parameters from called procedure’s stack *)
            tempESP ← Pop();
            tempSS ← Pop(); (* 32-bit pop, high-order 16 bits discarded *)
             (* segment descriptor information also loaded *)
            ESP ← tempESP;
            SS ← tempSS;
        ELSE (* OperandSize=16 *)
            EIP ← Pop();
            EIP ← EIP AND 0000FFFFH;
            CS ← Pop(); (* 16-bit pop; segment descriptor information also loaded *)
            CS(RPL) ← CPL;
            ESP ← ESP + SRC; (* release parameters from called procedure’s stack *)
            tempESP ← Pop();
            tempSS ← Pop(); (* 16-bit pop; segment descriptor information also loaded *)
             (* segment descriptor information also loaded *)
            ESP ← tempESP;
            SS ← tempSS;
    FI;
```

```
FOR each of segment register (ES, FS, GS, and DS)
    DO;
        IF segment register points to data or non-conforming code segment
        AND CPL > segment descriptor DPL; (* DPL in hidden part of segment register *)
            THEN (* segment register invalid *)
                SegmentSelector ← 0; (* null segment selector *)
        FI;
    OD;
For each of ES, FS, GS, and DS
DO
    IF segment selector index is not within descriptor table limits
        OR segment descriptor indicates the segment is not a data or
            readable code segment
        OR if the segment is a data or non-conforming code segment and the segment
            descriptor's DPL < CPL or RPL of code segment's segment selector
            THEN
                segment selector register ← null selector;
OD;
ESP ← ESP + SRC; (* release parameters from calling procedure's stack *)
```

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If the return code or stack segment selector null.
	If the return instruction pointer is not within the return code segment limit
#GP(selector)	If the RPL of the return code segment selector is less then the CPL.
	If the return code or stack segment selector index is not within its descriptor table limits.
	If the return code segment descriptor does not indicate a code segment.
	If the return code segment is non-conforming and the segment selector's DPL is not equal to the RPL of the code segment's segment selector
	If the return code segment is conforming and the segment selector's DPL greater than the RPL of the code segment's segment selector
	If the stack segment is not a writable data segment.
	If the stack segment selector RPL is not equal to the RPL of the return code segment selector.
	If the stack segment descriptor DPL is not equal to the RPL of the return code segment selector.

#SS(0)	If the top bytes of stack are not within stack limits.
	If the return stack segment is not present.
#NP(selector)	If the return code segment is not present.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory access occurs when the CPL is 3 and alignment checking is enabled.

Real-Address Mode Exceptions

#GP	If the return instruction pointer is not within the return code segment limit
#SS	If the top bytes of stack are not within stack limits.

Virtual-8086 Mode Exceptions

#GP(0)	If the return instruction pointer is not within the return code segment limit
#SS(0)	If the top bytes of stack are not within stack limits.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory access occurs when alignment checking is enabled.

ROL/ROR—Rotate

See entry for RCL/RCR/ROL/ROR—Rotate.

RSM—Resume from System Management Mode

Opcode	Instruction	Description
0F AA	RSM	Resume operation of interrupted program.

Description

Returns program control from system management mode (SMM) to the application program or operating-system procedure that was interrupted when the processor received an SMM interrupt. The processor's state is restored from the dump created upon entering SMM. If the processor detects invalid state information during state restoration, it enters the shutdown state. The following invalid information can cause a shutdown:

- Any reserved bit of CR4 is set to 1.
- Any illegal combination of bits in CR0, such as (PG=1 and PE=0) or (NW=1 and CD=0).
- (Intel Pentium and Intel486 processors only.) The value stored in the state dump base field is not a 32-KByte aligned address.

The contents of the model-specific registers are not affected by a return from SMM.

See Chapter 13, *System Management Mode (SMM)*, in the *IA-32 Intel Architecture Software Developer's Manual, Volume 3*, for more information about SMM and the behavior of the RSM instruction.

Operation

```
ReturnFromSMM;
ProcessorState ← Restore(SMMDump);
```

Flags Affected

All.

Protected Mode Exceptions

#UD	If an attempt is made to execute this instruction when the processor is not in SMM.

Real-Address Mode Exceptions

#UD	If an attempt is made to execute this instruction when the processor is not in SMM.

Virtual-8086 Mode Exceptions

#UD	If an attempt is made to execute this instruction when the processor is not in SMM.

RSQRTPS—Compute Reciprocals of Square Roots of Packed Single-Precision Floating-Point Values

Opcode	Instruction	Description
0F 52 /r	RSQRTPS *xmm1*, *xmm2/m128*	Compute the approximate reciprocals of the square roots of the packed single-precision floating-point values in *xmm2/m128* and store the results in *xmm1*.

Description

Performs an SIMD computation of the approximate reciprocals of the square roots of the four packed single-precision floating-point values in the source operand (second operand) and stores the packed single-precision floating-point results in the destination operand. The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. See Figure 10-5 in the *IA-32 Intel Architecture Software Developer's Manual, Volume 1* for an illustration of an SIMD single-precision floating-point operation.

The relative error for this approximation is:

$|\text{Relative Error}| \leq 1.5 * 2^{-12}$

The RSQRTPS instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an ∞ of the sign of the source value is returned. A denormal source value is treated as a 0.0 (of the same sign). When a source value is a negative value (other than −0.0), a floating-point indefinite is returned. When a source value is an SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned.

Operation

```
DEST[31-0] ← APPROXIMATE(1.0/SQRT(SRC[31-0]));
DEST[63-32] ← APPROXIMATE(1.0/SQRT(SRC[63-32]));
DEST[95-64] ← APPROXIMATE(1.0/SQRT(SRC[95-64]));
DEST[127-96] ← APPROXIMATE(1.0/SQRT(SRC[127-96]));
```

Intel C/C++ Compiler Intrinsic Equivalent

```
RSQRTPS        __m128 _mm_rsqrt_ps(__m128 a)
```

SIMD Floating-Point Exceptions

None.

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
	If a memory operand is not aligned on a 16-byte boundary, regardless of segment.

#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS in CR0 is set.
#UD	If EM in CR0 is set.
	If OSFXSR in CR4 is 0.
	If CPUID feature flag SSE is 0.

Real-Address Mode Exceptions

#GP(0)	If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If TS in CR0 is set.
#UD	If EM in CR0 is set.
	If OSFXSR in CR4 is 0.
	If CPUID feature flag SSE is 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.

RSQRTSS—Compute Reciprocal of Square Root of Scalar Single-Precision Floating-Point Value

Opcode	Instruction	Description
F3 0F 52 /r	RSQRTSS *xmm1*, *xmm2/m32*	Computes the approximate reciprocal of the square root of the low single-precision floating-point value in *xmm2/m32* and stores the results in *xmm1*.

Description

Computes an approximate reciprocal of the square root of the low single-precision floating-point value in the source operand (second operand) stores the single-precision floating-point result in the destination operand. The source operand can be an XMM register or a 32-bit memory location. The destination operand is an XMM register. The three high-order doublewords of the destination operand remain unchanged. See Figure 10-6 in the *IA-32 Intel Architecture Software Developer's Manual, Volume 1* for an illustration of a scalar single-precision floating-point operation.

The relative error for this approximation is:

$|\text{Relative Error}| \leq 1.5 * 2^{-12}$

The RSQRTSS instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an ∞ of the sign of the source value is returned. A denormal source value is treated as a 0.0 (of the same sign). When a source value is a negative value (other than –0.0), a floating-point indefinite is returned. When a source value is an SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned.

Operation

```
DEST[31-0] ← APPROXIMATE(1.0/SQRT(SRC[31-0]));
* DEST[127-32] remains unchanged *;
```

Intel C/C++ Compiler Intrinsic Equivalent

```
RSQRTSS        __m128 _mm_rsqrt_ss(__m128 a)
```

SIMD Floating-Point Exceptions

None.

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.

#NM	If TS in CR0 is set.
#UD	If EM in CR0 is set.
	If OSFXSR in CR4 is 0.
	If CPUID feature flag SSE is 0.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

GP(0)	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If TS in CR0 is set.
#UD	If EM in CR0 is set.
	If OSFXSR in CR4 is 0.
	If CPUID feature flag SSE is 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

SAHF—Store AH into Flags

Opcode	Instruction	Description
9E	SAHF	Load SF, ZF, AF, PF, and CF from AH into EFLAGS register.

Description

Loads the SF, ZF, AF, PF, and CF flags of the EFLAGS register with values from the corresponding bits in the AH register (bits 7, 6, 4, 2, and 0, respectively). Bits 1, 3, and 5 of register AH are ignored; the corresponding reserved bits (1, 3, and 5) in the EFLAGS register remain as shown in the "Operation" section below.

Operation

```
EFLAGS(SF:ZF:0:AF:0:PF:1:CF) ← AH;
```

Flags Affected

The SF, ZF, AF, PF, and CF flags are loaded with values from the AH register. Bits 1, 3, and 5 of the EFLAGS register are unaffected, with the values remaining 1, 0, and 0, respectively.

Exceptions (All Operating Modes)

None.

SAL/SAR/SHL/SHR—Shift

Opcode	Instruction	Description
D0 /4	SAL *r/m8*	Multiply *r/m8* by 2, 1 time.
D2 /4	SAL *r/m8*,CL	Multiply *r/m8* by 2, CL times.
C0 /4 *ib*	SAL *r/m8,imm8*	Multiply *r/m8* by 2, *imm8* times.
D1 /4	SAL *r/m16*	Multiply *r/m16* by 2, 1 time.
D3 /4	SAL *r/m16*,CL	Multiply *r/m16* by 2, CL times.
C1 /4 *ib*	SAL *r/m16,imm8*	Multiply *r/m16* by 2, *imm8* times.
D1 /4	SAL *r/m32*	Multiply *r/m32* by 2, 1 time.
D3 /4	SAL *r/m32*,CL	Multiply *r/m32* by 2, CL times.
C1 /4 *ib*	SAL *r/m32,imm8*	Multiply *r/m32* by 2, *imm8* times.
D0 /7	SAR *r/m8*	Signed divide* *r/m8* by 2, 1 times.
D2 /7	SAR *r/m8*,CL	Signed divide* *r/m8* by 2, CL times.
C0 /7 *ib*	SAR *r/m8,imm8*	Signed divide* *r/m8* by 2, *imm8* times.
D1 /7	SAR *r/m16*	Signed divide* *r/m16* by 2, 1 time.
D3 /7	SAR *r/m16*,CL	Signed divide* *r/m16* by 2, CL times.
C1 /7 *ib*	SAR *r/m16,imm8*	Signed divide* *r/m16* by 2, *imm8* times.
D1 /7	SAR *r/m32*	Signed divide* *r/m32* by 2, 1 time.
D3 /7	SAR *r/m32*,CL	Signed divide* *r/m32* by 2, CL times.
C1 /7 *ib*	SAR *r/m32,imm8*	Signed divide* *r/m32* by 2, *imm8* times.
D0 /4	SHL *r/m8*	Multiply *r/m8* by 2, 1 time.
D2 /4	SHL *r/m8*,CL	Multiply *r/m8* by 2, CL times.
C0 /4 *ib*	SHL *r/m8,imm8*	Multiply *r/m8* by 2, *imm8* times.
D1 /4	SHL *r/m16*	Multiply *r/m16* by 2, 1 time.
D3 /4	SHL *r/m16*,CL	Multiply *r/m16* by 2, CL times.
C1 /4 *ib*	SHL *r/m16,imm8*	Multiply *r/m16* by 2, *imm8* times.
D1 /4	SHL *r/m32*	Multiply *r/m32* by 2, 1 time.
D3 /4	SHL *r/m32*,CL	Multiply *r/m32* by 2, CL times.
C1 /4 *ib*	SHL *r/m32,imm8*	Multiply *r/m32* by 2, *imm8* times.
D0 /5	SHR *r/m8*	Unsigned divide *r/m8* by 2, 1 time.
D2 /5	SHR *r/m8*,CL	Unsigned divide *r/m8* by 2, CL times.
C0 /5 *ib*	SHR *r/m8,imm8*	Unsigned divide *r/m8* by 2, *imm8* times.
D1 /5	SHR *r/m16*	Unsigned divide *r/m16* by 2, 1 time.
D3 /5	SHR *r/m16*,CL	Unsigned divide *r/m16* by 2, CL times.
C1 /5 *ib*	SHR *r/m16,imm8*	Unsigned divide *r/m16* by 2, *imm8* times.
D1 /5	SHR *r/m32*	Unsigned divide *r/m32* by 2, 1 time.
D3 /5	SHR *r/m32*,CL	Unsigned divide *r/m32* by 2, CL times.
C1 /5 *ib*	SHR *r/m32,imm8*	Unsigned divide *r/m32* by 2, *imm8* times.

NOTE:

* Not the same form of division as IDIV; rounding is toward negative infinity.

Description

Shifts the bits in the first operand (destination operand) to the left or right by the number of bits specified in the second operand (count operand). Bits shifted beyond the destination operand boundary are first shifted into the CF flag, then discarded. At the end of the shift operation, the CF flag contains the last bit shifted out of the destination operand.

The destination operand can be a register or a memory location. The count operand can be an immediate value or register CL. The count is masked to 5 bits, which limits the count range to 0 to 31. A special opcode encoding is provided for a count of 1.

The shift arithmetic left (SAL) and shift logical left (SHL) instructions perform the same operation; they shift the bits in the destination operand to the left (toward more significant bit locations). For each shift count, the most significant bit of the destination operand is shifted into the CF flag, and the least significant bit is cleared (see Figure 7-7 in the *IA-32 Intel Architecture Software Developer's Manual, Volume 1*).

The shift arithmetic right (SAR) and shift logical right (SHR) instructions shift the bits of the destination operand to the right (toward less significant bit locations). For each shift count, the least significant bit of the destination operand is shifted into the CF flag, and the most significant bit is either set or cleared depending on the instruction type. The SHR instruction clears the most significant bit (see Figure 7-8 in the *IA-32 Intel Architecture Software Developer's Manual, Volume 1*); the SAR instruction sets or clears the most significant bit to correspond to the sign (most significant bit) of the original value in the destination operand. In effect, the SAR instruction fills the empty bit position's shifted value with the sign of the unshifted value (see Figure 7-9 in the *IA-32 Intel Architecture Software Developer's Manual, Volume 1*).

The SAR and SHR instructions can be used to perform signed or unsigned division, respectively, of the destination operand by powers of 2. For example, using the SAR instruction to shift a signed integer 1 bit to the right divides the value by 2.

Using the SAR instruction to perform a division operation does not produce the same result as the IDIV instruction. The quotient from the IDIV instruction is rounded toward zero, whereas the "quotient" of the SAR instruction is rounded toward negative infinity. This difference is apparent only for negative numbers. For example, when the IDIV instruction is used to divide -9 by 4, the result is -2 with a remainder of -1. If the SAR instruction is used to shift -9 right by two bits, the result is -3 and the "remainder" is +3; however, the SAR instruction stores only the most significant bit of the remainder (in the CF flag).

The OF flag is affected only on 1-bit shifts. For left shifts, the OF flag is set to 0 if the most-significant bit of the result is the same as the CF flag (that is, the top two bits of the original operand were the same); otherwise, it is set to 1. For the SAR instruction, the OF flag is cleared for all 1-bit shifts. For the SHR instruction, the OF flag is set to the most-significant bit of the original operand.

IA-32 Architecture Compatibility

The 8086 does not mask the shift count. However, all other IA-32 processors (starting with the Intel 286 processor) do mask the shift count to 5 bits, resulting in a maximum count of 31. This masking is done in all operating modes (including the virtual-8086 mode) to reduce the maximum execution time of the instructions.

Operation

```
tempCOUNT ← (COUNT AND 1FH);
tempDEST ← DEST;
WHILE (tempCOUNT ≠ 0)
DO
    IF instruction is SAL or SHL
        THEN
            CF ← MSB(DEST);
        ELSE (* instruction is SAR or SHR *)
            CF ← LSB(DEST);
    FI;
    IF instruction is SAL or SHL
        THEN
            DEST ← DEST ∗ 2;
        ELSE
            IF instruction is SAR
                THEN
                    DEST ← DEST / 2 (*Signed divide, rounding toward negative infinity*);
                ELSE (* instruction is SHR *)
                    DEST ← DEST / 2 ; (* Unsigned divide *);
            FI;
    FI;
    tempCOUNT ← tempCOUNT – 1;
OD;
(* Determine overflow for the various instructions *)
IF (COUNT and 1FH) = 1
    THEN
        IF instruction is SAL or SHL
            THEN
                OF ← MSB(DEST) XOR CF;
            ELSE
                IF instruction is SAR
                    THEN
                        OF ← 0;
                    ELSE (* instruction is SHR *)
                        OF ← MSB(tempDEST);
                FI;
        FI;
    ELSE IF (COUNT AND 1FH) = 0
        THEN
            All flags remain unchanged;
        ELSE (* COUNT neither 1 or 0 *)
            OF ← undefined;
    FI;
FI;
```

Flags Affected

The CF flag contains the value of the last bit shifted out of the destination operand; it is undefined for SHL and SHR instructions where the count is greater than or equal to the size (in bits) of the destination operand. The OF flag is affected only for 1-bit shifts (see “Description” above); otherwise, it is undefined. The SF, ZF, and PF flags are set according to the result. If the count is 0, the flags are not affected. For a non-zero count, the AF flag is undefined.

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.

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.

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.

SBB—Integer Subtraction with Borrow

Opcode	Instruction	Description
1C *ib*	SBB AL,*imm8*	Subtract with borrow *imm8* from AL.
1D *iw*	SBB AX,*imm16*	Subtract with borrow *imm16* from AX.
1D *id*	SBB EAX,*imm32*	Subtract with borrow *imm32* from EAX.
80 /3 *ib*	SBB *r/m8,imm8*	Subtract with borrow *imm8* from *r/m8.*
81 /3 *iw*	SBB *r/m16,imm16*	Subtract with borrow *imm16* from *r/m16.*
81 /3 *id*	SBB *r/m32,imm32*	Subtract with borrow *imm32* from *r/m32.*
83 /3 *ib*	SBB *r/m16,imm8*	Subtract with borrow sign-extended *imm8* from *r/m16.*
83 /3 *ib*	SBB *r/m32,imm8*	Subtract with borrow sign-extended *imm8* from *r/m32.*
18 */r*	SBB *r/m8,r8*	Subtract with borrow *r8* from *r/m8.*
19 */r*	SBB *r/m16,r16*	Subtract with borrow *r16* from *r/m16.*
19 */r*	SBB *r/m32,r32*	Subtract with borrow *r32* from *r/m32.*
1A */r*	SBB *r8,r/m8*	Subtract with borrow *r/m8* from *r8.*
1B */r*	SBB *r16,r/m16*	Subtract with borrow *r/m16* from *r16.*
1B */r*	SBB *r32,r/m32*	Subtract with borrow *r/m32* from *r32.*

Description

Adds the source operand (second operand) and the carry (CF) flag, and subtracts the result from the destination operand (first operand). The result of the subtraction is stored in the destination operand. The destination operand can be a register or a memory location; the source operand can be an immediate, a register, or a memory location. (However, two memory operands cannot be used in one instruction.) The state of the CF flag represents a borrow from a previous subtraction.

When an immediate value is used as an operand, it is sign-extended to the length of the destination operand format.

The SBB instruction does not distinguish between signed or unsigned operands. Instead, the processor evaluates the result for both data types and sets the OF and CF flags to indicate a borrow in the signed or unsigned result, respectively. The SF flag indicates the sign of the signed result.

The SBB instruction is usually executed as part of a multibyte or multiword subtraction in which a SUB instruction is followed by a SBB instruction.

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

Operation

```
DEST ← DEST – (SRC + CF);
```

Flags Affected

The OF, SF, ZF, AF, PF, and CF flags are set according to the result.

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.

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.

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.

SCAS/SCASB/SCASW/SCASD—Scan String

Opcode	Instruction	Description
AE	SCAS m8	Compare AL with byte at ES:(E)DI and set status flags.
AF	SCAS m16	Compare AX with word at ES:(E)DI and set status flags.
AF	SCAS m32	Compare EAX with doubleword at ES(E)DI and set status flags.
AE	SCASB	Compare AL with byte at ES:(E)DI and set status flags.
AF	SCASW	Compare AX with word at ES:(E)DI and set status flags.
AF	SCASD	Compare EAX with doubleword at ES:(E)DI and set status flags.

Description

Compares the byte, word, or double word specified with the memory operand with the value in the AL, AX, or EAX register, and sets the status flags in the EFLAGS register according to the results. The memory operand address is read from either the ES:EDI or the ES:DI registers (depending on the address-size attribute of the instruction, 32 or 16, respectively). The ES segment cannot be overridden with a segment override prefix.

At the assembly-code level, two forms of this instruction are allowed: the “explicit-operands” form and the “no-operands” form. The explicit-operand form (specified with the SCAS mnemonic) allows the memory operand to be specified explicitly. Here, the memory operand should be a symbol that indicates the size and location of the operand value. The register operand is then automatically selected to match the size of the memory operand (the AL register for byte comparisons, AX for word comparisons, and EAX for doubleword comparisons). This explicit-operand form is provided to allow documentation; however, note that the documentation provided by this form can be misleading. That is, the memory operand symbol must specify the correct **type** (size) of the operand (byte, word, or doubleword), but it does not have to specify the correct **location**. The location is always specified by the ES:(E)DI registers, which must be loaded correctly before the compare string instruction is executed.

The no-operands form provides “short forms” of the byte, word, and doubleword versions of the SCAS instructions. Here also ES:(E)DI is assumed to be the memory operand and the AL, AX, or EAX register is assumed to be the register operand. The size of the two operands is selected with the mnemonic: SCASB (byte comparison), SCASW (word comparison), or SCASD (doubleword comparison).

After the comparison, the (E)DI register is incremented or decremented automatically according to the setting of the DF flag in the EFLAGS register. (If the DF flag is 0, the (E)DI register is incremented; if the DF flag is 1, the (E)DI register is decremented.) The (E)DI register is incremented or decremented by 1 for byte operations, by 2 for word operations, or by 4 for doubleword operations.

The SCAS, SCASB, SCASW, and SCASD instructions can be preceded by the REP prefix for block comparisons of ECX bytes, words, or doublewords. More often, however, these instructions will be used in a LOOP construct that takes some action based on the setting of the status flags before the next comparison is made. See “REP/REPE/REPZ/REPNE /REPNZ—Repeat String Operation Prefix” in this chapter for a description of the REP prefix.

Operation

```
IF (byte cmparison)
   THEN
      temp ← AL − SRC;
      SetStatusFlags(temp);
         THEN IF DF = 0
            THEN (E)DI ← (E)DI + 1;
            ELSE (E)DI ← (E)DI – 1;
         FI;
   ELSE IF (word comparison)
      THEN
         temp ← AX − SRC;
         SetStatusFlags(temp)
            THEN IF DF = 0
               THEN (E)DI ← (E)DI + 2;
               ELSE (E)DI ← (E)DI – 2;
            FI;
      ELSE (* doubleword comparison *)
         temp ← EAX − SRC;
         SetStatusFlags(temp)
            THEN IF DF = 0
               THEN (E)DI ← (E)DI + 4;
               ELSE (E)DI ← (E)DI – 4;
            FI;
   FI;
FI;
```

Flags Affected

The OF, SF, ZF, AF, PF, and CF flags are set according to the temporary result of the comparison.

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the limit of the ES segment.
	If the ES register contains a null segment selector.
	If an illegal memory operand effective address in the ES segment is given.
#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.

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.

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.

SET*cc*—Set Byte on Condition

Opcode	Instruction	Description
0F 97	SETA *r/m8*	Set byte if above (CF=0 and ZF=0).
0F 93	SETAE *r/m8*	Set byte if above or equal (CF=0).
0F 92	SETB *r/m8*	Set byte if below (CF=1).
0F 96	SETBE *r/m8*	Set byte if below or equal (CF=1 or ZF=1).
0F 92	SETC *r/m8*	Set if carry (CF=1).
0F 94	SETE *r/m8*	Set byte if equal (ZF=1).
0F 9F	SETG *r/m8*	Set byte if greater (ZF=0 and SF=OF).
0F 9D	SETGE *r/m8*	Set byte if greater or equal (SF=OF).
0F 9C	SETL *r/m8*	Set byte if less (SF<>OF).
0F 9E	SETLE *r/m8*	Set byte if less or equal (ZF=1 or SF<>OF).
0F 96	SETNA *r/m8*	Set byte if not above (CF=1 or ZF=1).
0F 92	SETNAE *r/m8*	Set byte if not above or equal (CF=1).
0F 93	SETNB *r/m8*	Set byte if not below (CF=0).
0F 97	SETNBE *r/m8*	Set byte if not below or equal (CF=0 and ZF=0).
0F 93	SETNC *r/m8*	Set byte if not carry (CF=0).
0F 95	SETNE *r/m8*	Set byte if not equal (ZF=0).
0F 9E	SETNG *r/m8*	Set byte if not greater (ZF=1 or SF<>OF).
0F 9C	SETNGE *r/m8*	Set if not greater or equal (SF<>OF).
0F 9D	SETNL *r/m8*	Set byte if not less (SF=OF).
0F 9F	SETNLE *r/m8*	Set byte if not less or equal (ZF=0 and SF=OF).
0F 91	SETNO *r/m8*	Set byte if not overflow (OF=0).
0F 9B	SETNP *r/m8*	Set byte if not parity (PF=0).
0F 99	SETNS *r/m8*	Set byte if not sign (SF=0).
0F 95	SETNZ *r/m8*	Set byte if not zero (ZF=0).
0F 90	SETO *r/m8*	Set byte if overflow (OF=1).
0F 9A	SETP *r/m8*	Set byte if parity (PF=1).
0F 9A	SETPE *r/m8*	Set byte if parity even (PF=1).
0F 9B	SETPO *r/m8*	Set byte if parity odd (PF=0).
0F 98	SETS *r/m8*	Set byte if sign (SF=1).
0F 94	SETZ *r/m8*	Set byte if zero (ZF=1).

Description

Sets the destination operand to 0 or 1 depending on the settings of the status flags (CF, SF, OF, ZF, and PF) in the EFLAGS register. The destination operand points to a byte register or a byte in memory. The condition code suffix (*cc*) indicates the condition being tested for.

The terms “above” and “below” are associated with the CF flag and refer to the relationship between two unsigned integer values. The terms “greater” and “less” are associated with the SF and OF flags and refer to the relationship between two signed integer values.

Many of the SET*cc* instruction opcodes have alternate mnemonics. For example, SETG (set byte if greater) and SETNLE (set if not less or equal) have the same opcode and test for the same condition: ZF equals 0 and SF equals OF. These alternate mnemonics are provided to make code more intelligible. Appendix B, *EFLAGS Condition Codes*, in the *IA-32 Intel Architecture Software Developer's Manual, Volume 1*, shows the alternate mnemonics for various test conditions.

Some languages represent a logical one as an integer with all bits set. This representation can be obtained by choosing the logically opposite condition for the SET*cc* instruction, then decrementing the result. For example, to test for overflow, use the SETNO instruction, then decrement the result.

Operation

```
IF condition
    THEN DEST ← 1
    ELSE DEST ← 0;
FI;
```

Flags Affected

None.

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.

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.

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.

SFENCE—Store Fence

Opcode	Instruction	Description
0F AE /7	SFENCE	Serializes store operations.

Description

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

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

Operation

```
Wait_On_Following_Stores_Until(preceding_stores_globally_visible);
```

Intel C/C++ Compiler Intrinsic Equivalent

```
void_mm_sfence(void)
```

Protected Mode Exceptions

None.

Real-Address Mode Exceptions

None.

Virtual-8086 Mode Exceptions

None.

SGDT—Store Global Descriptor Table Register

Opcode	Instruction	Description
0F 01 /0	SGDT *m*	Store GDTR to *m.*

Description

Stores the content of the global descriptor table register (GDTR) in the destination operand. The destination operand specifies a 6-byte memory location. If the operand-size attribute is 32 bits, the 16-bit limit field of the register is stored in the low 2 bytes of the memory location and the 32-bit base address is stored in the high 4 bytes. If the operand-size attribute is 16 bits, the limit is stored in the low 2 bytes and the 24-bit base address is stored in the third, fourth, and fifth byte, with the sixth byte filled with 0s.

SGDT is only useful in operating-system software; however, it can be used in application programs without causing an exception to be generated.

See "LGDT/LIDT—Load Global/Interrupt Descriptor Table Register" in Chapter 3 for information on loading the GDTR and IDTR.

IA-32 Architecture Compatibility

The 16-bit form of the SGDT is compatible with the Intel 286 processor if the upper 8 bits are not referenced. The Intel 286 processor fills these bits with 1s; the Pentium 4, Intel Xeon, P6 family, Pentium, Intel486, and Intel386 processors fill these bits with 0s.

Operation

```
IF instruction is SGDT
        IF OperandSize = 16
            THEN
                DEST[0:15] ← GDTR(Limit);
                DEST[16:39] ← GDTR(Base); (* 24 bits of base address loaded; *)
                DEST[40:47] ← 0;
            ELSE (* 32-bit Operand Size *)
                DEST[0:15] ← GDTR(Limit);
                DEST[16:47] ← GDTR(Base); (* full 32-bit base address loaded *)
        FI;
FI;
```

Flags Affected

None.

Protected Mode Exceptions

#UD	If the destination operand is a register.
#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 is used to access memory and it 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.

Real-Address Mode Exceptions

#UD	If the destination operand is a register.
#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.

Virtual-8086 Mode Exceptions

#UD	If the destination operand is a register.
#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.

SHL/SHR—Shift Instructions

See entry for SAL/SAR/SHL/SHR—Shift.

SHLD—Double Precision Shift Left

Opcode	Instruction	Description
0F A4	SHLD *r/m16, r16, imm8*	Shift *r/m16* to left *imm8* places while shifting bits from *r16* in from the right.
0F A5	SHLD *r/m16, r16*, CL	Shift *r/m16* to left CL places while shifting bits from *r16* in from the right.
0F A4	SHLD *r/m32, r32, imm8*	Shift *r/m32* to left *imm8* places while shifting bits from *r32* in from the right.
0F A5	SHLD *r/m32, r32*, CL	Shift *r/m32* to left CL places while shifting bits from *r32* in from the right.

Description

Shifts the first operand (destination operand) to the left the number of bits specified by the third operand (count operand). The second operand (source operand) provides bits to shift in from the right (starting with bit 0 of the destination operand). The destination operand can be a register or a memory location; the source operand is a register. The count operand is an unsigned integer that can be an immediate byte or the contents of the CL register. Only bits 0 through 4 of the count are used, which masks the count to a value between 0 and 31. If the count is greater than the operand size, the result in the destination operand is undefined.

If the count is 1 or greater, the CF flag is filled with the last bit shifted out of the destination operand. For a 1-bit shift, the OF flag is set if a sign change occurred; otherwise, it is cleared. If the count operand is 0, the flags are not affected.

The SHLD instruction is useful for multi-precision shifts of 64 bits or more.

Operation

```
COUNT ← COUNT MOD 32;
SIZE ← OperandSize
IF COUNT = 0
    THEN
        no operation
    ELSE
        IF COUNT > SIZE
            THEN (* Bad parameters *)
                DEST is undefined;
                CF, OF, SF, ZF, AF, PF are undefined;
            ELSE (* Perform the shift *)
                CF ← BIT[DEST, SIZE – COUNT];
                (* Last bit shifted out on exit *)
                FOR i ← SIZE – 1 DOWNTO COUNT
                DO
                    Bit(DEST, i) ← Bit(DEST, i – COUNT);
                OD;
            FOR i ← COUNT – 1 DOWNTO 0
            DO
                BIT[DEST, i] ← BIT[SRC, i – COUNT + SIZE];
            OD;
        FI;
FI;
```

Flags Affected

If the count is 1 or greater, the CF flag is filled with the last bit shifted out of the destination operand and the SF, ZF, and PF flags are set according to the value of the result. For a 1-bit shift, the OF flag is set if a sign change occurred; otherwise, it is cleared. For shifts greater than 1 bit, the OF flag is undefined. If a shift occurs, the AF flag is undefined. If the count operand is 0, the flags are not affected. If the count is greater than the operand size, the flags are undefined.

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.

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.

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.

SHRD—Double Precision Shift Right

Opcode	Instruction	Description
0F AC	SHRD *r/m16, r16, imm8*	Shift *r/m16* to right *imm8* places while shifting bits from *r16* in from the left.
0F AD	SHRD *r/m16, r16*, CL	Shift *r/m16* to right CL places while shifting bits from *r16* in from the left.
0F AC	SHRD *r/m32, r32, mm8*	Shift *r/m32* to right *imm8* places while shifting bits from *r32* in from the left.
0F AD	SHRD *r/m32, r32*, CL	Shift *r/m32* to right CL places while shifting bits from *r32* in from the left.

Description

Shifts the first operand (destination operand) to the right the number of bits specified by the third operand (count operand). The second operand (source operand) provides bits to shift in from the left (starting with the most significant bit of the destination operand). The destination operand can be a register or a memory location; the source operand is a register. The count operand is an unsigned integer that can be an immediate byte or the contents of the CL register. Only bits 0 through 4 of the count are used, which masks the count to a value between 0 and 31. If the count is greater than the operand size, the result in the destination operand is undefined.

If the count is 1 or greater, the CF flag is filled with the last bit shifted out of the destination operand. For a 1-bit shift, the OF flag is set if a sign change occurred; otherwise, it is cleared. If the count operand is 0, the flags are not affected.

The SHRD instruction is useful for multiprecision shifts of 64 bits or more.

Operation

```
COUNT ← COUNT MOD 32;
SIZE ← OperandSize
IF COUNT = 0
   THEN
       no operation
   ELSE
       IF COUNT > SIZE
           THEN (* Bad parameters *)
               DEST is undefined;
               CF, OF, SF, ZF, AF, PF are undefined;
           ELSE (* Perform the shift *)
               CF ← BIT[DEST, COUNT – 1]; (* last bit shifted out on exit *)
               FOR i ← 0 TO SIZE – 1 – COUNT
                   DO
                       BIT[DEST, i] ← BIT[DEST, i + COUNT];
                   OD;
               FOR i ← SIZE – COUNT TO SIZE – 1
```

```
                DO
                    BIT[DEST,i] ← BIT[SRC, i + COUNT – SIZE];
                OD;
      FI;
FI;
```

Flags Affected

If the count is 1 or greater, the CF flag is filled with the last bit shifted out of the destination operand and the SF, ZF, and PF flags are set according to the value of the result. For a 1-bit shift, the OF flag is set if a sign change occurred; otherwise, it is cleared. For shifts greater than 1 bit, the OF flag is undefined. If a shift occurs, the AF flag is undefined. If the count operand is 0, the flags are not affected. If the count is greater than the operand size, the flags are undefined.

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.

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.

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.

SHUFPD—Shuffle Packed Double-Precision Floating-Point Values

Opcode	Instruction	Description
66 0F C6 /r ib	SHUFPD *xmm1*, *xmm2/m128*, *imm8*	Shuffle packed double-precision floating-point values selected by *imm8* from *xmm1* and *xmm1/m128* to *xmm1*.

Description

Moves either of the two packed double-precision floating-point values from destination operand (first operand) into the low quadword of the destination operand; moves either of the two packed double-precision floating-point values from the source operand into to the high quadword of the destination operand (see Figure 4-12). The select operand (third operand) determines which values are moved to the destination operand.

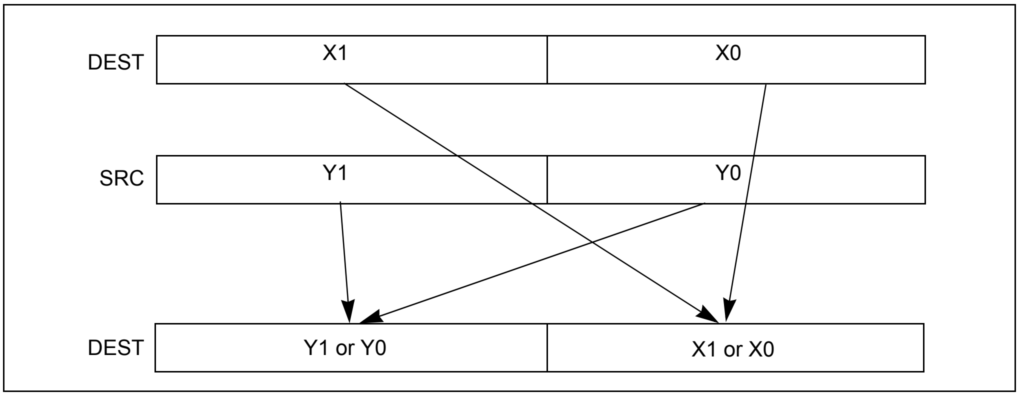


Figure 4-12. SHUFPD Shuffle Operation

The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. The select operand is an 8-bit immediate: bit 0 selects which value is moved from the destination operand to the result (where 0 selects the low quadword and 1 selects the high quadword) and bit 1 selects which value is moved from the source operand to the result. Bits 2 through 7 of the select operand are reserved and must be set to 0.

Operation

```
IF SELECT[0] = 0
    THEN DEST[63-0] ← DEST[63-0];
    ELSE DEST[63-0] ← DEST[127-64]; FI;
IF SELECT[1] = 0
    THEN DEST[127-64] ← SRC[63-0];
    ELSE DEST[127-64] ← SRC[127-64]; FI;
```

Intel C/C++ Compiler Intrinsic Equivalent

SHUFPD	__m128d _mm_shuffle_pd(__m128d a, __m128d b, unsigned int imm8)

SIMD Floating-Point Exceptions

None.

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
	If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS in CR0 is set.
#UD	If EM in CR0 is set.
	If OSFXSR in CR4 is 0.
	If CPUID feature flag SSE2 is 0.

Real-Address Mode Exceptions

#GP(0)	If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If TS in CR0 is set.
#UD	If EM in CR0 is set.
	If OSFXSR in CR4 is 0.
	If CPUID feature flag SSE2 is 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.

SHUFPS—Shuffle Packed Single-Precision Floating-Point Values

Opcode	Instruction	Description
0F C6 /r ib	SHUFPS *xmm1*, *xmm2/m128, imm8*	Shuffle packed single-precision floating-point values selected by *imm8* from *xmm1* and *xmm1/m128* to *xmm1*.

Description

Moves two of the four packed single-precision floating-point values from the destination operand (first operand) into the low quadword of the destination operand; moves two of the four packed single-precision floating-point values from the source operand (second operand) into to the high quadword of the destination operand (see Figure 4-13). The select operand (third operand) determines which values are moved to the destination operand.

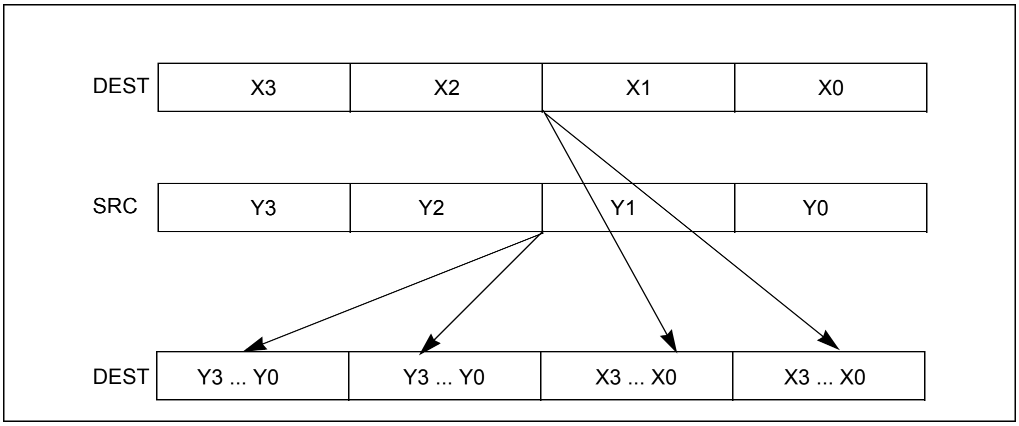


Figure 4-13. SHUFPS Shuffle Operation

The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. The select operand is an 8-bit immediate: bits 0 and 1 select the value to be moved from the destination operand to the low doubleword of the result, bits 2 and 3 select the value to be moved from the destination operand to the second doubleword of the result, bits 4 and 5 select the value to be moved from the source operand to the third doubleword of the result, and bits 6 and 7 select the value to be moved from the source operand to the high doubleword of the result.

Operation

```
CASE (SELECT[1-0]) OF
    0:   DEST[31-0]  ← DEST[31-0];
    1:   DEST[31-0]  ← DEST[63-32];
    2:   DEST[31-0]  ← DEST[95-64];
    3:   DEST[31-0]  ← DEST[127-96];
ESAC;
CASE (SELECT[3-2]) OF
    0:   DEST[63-32]  ← DEST[31-0];
    1:   DEST[63-32]  ← DEST[63-32];
    2:   DEST[63-32]  ← DEST[95-64];
    3:   DEST[63-32]  ← DEST[127-96];
ESAC;
CASE (SELECT[5-4]) OF
    0:   DEST[95-64]  ← SRC[31-0];
    1:   DEST[95-64]  ← SRC[63-32];
    2:   DEST[95-64]  ← SRC[95-64];
    3:   DEST[95-64]  ← SRC[127-96];
ESAC;
CASE (SELECT[7-6]) OF
    0:   DEST[127-96]  ← SRC[31-0];
    1:   DEST[127-96]  ← SRC[63-32];
    2:   DEST[127-96]  ← SRC[95-64];
    3:   DEST[127-96]  ← SRC[127-96];
ESAC;
```

Intel C/C++ Compiler Intrinsic Equivalent

SHUFPS	__m128 _mm_shuffle_ps(__m128 a, __m128 b, unsigned int imm8)

SIMD Floating-Point Exceptions

None.

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
	If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS in CR0 is set.

#UD	If EM in CR0 is set.
	If OSFXSR in CR4 is 0.
	If CPUID feature flag SSE is 0.

Real-Address Mode Exceptions

#GP(0)	If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If TS in CR0 is set.
#UD	If EM in CR0 is set.
	If OSFXSR in CR4 is 0.
	If CPUID feature flag SSE is 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.

SIDT—Store Interrupt Descriptor Table Register

Opcode	Instruction	Description
0F 01 /1	SIDT *m*	Store IDTR to *m.*

Description

Stores the content the interrupt descriptor table register (IDTR) in the destination operand. The destination operand specifies a 6-byte memory location. If the operand-size attribute is 32 bits, the 16-bit limit field of the register is stored in the low 2 bytes of the memory location and the 32-bit base address is stored in the high 4 bytes. If the operand-size attribute is 16 bits, the limit is stored in the low 2 bytes and the 24-bit base address is stored in the third, fourth, and fifth byte, with the sixth byte filled with 0s.

SIDT is only useful in operating-system software; however, it can be used in application programs without causing an exception to be generated.

See "LGDT/LIDT—Load Global/Interrupt Descriptor Table Register" in Chapter 4 for information on loading the GDTR and IDTR.

IA-32 Architecture Compatibility

The 16-bit form of SIDT is compatible with the Intel 286 processor if the upper 8 bits are not referenced. The Intel 286 processor fills these bits with 1s; the Pentium 4, Intel Xeon, P6 family, Pentium, Intel486, and Intel386 processors fill these bits with 0s.

Operation

```
IF instruction is SIDT
    THEN
        IF OperandSize = 16
            THEN
                DEST[0:15] ← IDTR(Limit);
                DEST[16:39] ← IDTR(Base); (* 24 bits of base address loaded; *)
                DEST[40:47] ← 0;
            ELSE (* 32-bit Operand Size *)
                DEST[0:15] ← IDTR(Limit);
                DEST[16:47] ← IDTR(Base); (* full 32-bit base address loaded *)
        FI;
FI;
```

Flags Affected

None.

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 is used to access memory and it 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.

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.

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.

SLDT—Store Local Descriptor Table Register

Opcode	Instruction	Description
0F 00 /0	SLDT *r/m16*	Store segment selector from LDTR in *r/m16*.

Description

Stores the segment selector from the local descriptor table register (LDTR) in the destination operand. The destination operand can be a general-purpose register or a memory location. The segment selector stored with this instruction points to the segment descriptor (located in the GDT) for the current LDT. This instruction can only be executed in protected mode.

When the destination operand is a 32-bit register, the 16-bit segment selector is copied into the lower-order 16 bits of the register. The high-order 16 bits of the register are cleared for the Pentium 4, Intel Xeon, and P6 family processors and are undefined for Pentium, Intel486, and Intel386 processors. When the destination operand is a memory location, the segment selector is written to memory as a 16-bit quantity, regardless of the operand size.

The SLDT instruction is only useful in operating-system software; however, it can be used in application programs.

Operation

```
DEST ← LDTR(SegmentSelector);
```

Flags Affected

None.

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 is used to access memory and it 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.

Real-Address Mode Exceptions

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

Virtual-8086 Mode Exceptions

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

SMSW—Store Machine Status Word

Opcode	Instruction	Description
0F 01 /4	SMSW *r/m16*	Store machine status word to *r/m16*.
0F 01 /4	SMSW *r32/m16*	Store machine status word in low-order 16 bits of *r32/m16*; high-order 16 bits of *r32* are undefined.

Description

Stores the machine status word (bits 0 through 15 of control register CR0) into the destination operand. The destination operand can be a 16-bit general-purpose register or a memory location.

When the destination operand is a 32-bit register, the low-order 16 bits of register CR0 are copied into the low-order 16 bits of the register and the upper 16 bits of the register are undefined. When the destination operand is a memory location, the low-order 16 bits of register CR0 are written to memory as a 16-bit quantity, regardless of the operand size.

The SMSW instruction is only useful in operating-system software; however, it is not a privileged instruction and can be used in application programs.

This instruction is provided for compatibility with the Intel 286 processor. Programs and procedures intended to run on the Pentium 4, Intel Xeon, P6 family, Pentium, Intel486, and Intel386 processors should use the MOV (control registers) instruction to load the machine status word.

Operation

```
DEST ← CR0[15:0]; (* Machine status word *);
```

Flags Affected

None.

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 is used to access memory and it 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.

Real-Address Mode Exceptions

#GP	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.

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.

SQRTPD—Compute Square Roots of Packed Double-Precision Floating-Point Values

Opcode	Instruction	Description
66 0F 51 /r	SQRTPD *xmm1*, *xmm2/m128*	Compute the square root of the packed double-precision floating-point values in *xmm2/m128* and store the results in *xmm1*.

Description

Performs an SIMD computation of the square roots of the two packed double-precision floating-point values in the source operand (second operand) stores the packed double-precision floating-point results in the destination operand. The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. See Figure 11-3 in the *IA-32 Intel Architecture Software Developer's Manual, Volume 1* for an illustration of an SIMD double-precision floating-point operation.

Operation

```
DEST[63-0] ← SQRT(SRC[63-0]);
DEST[127-64] ← SQRT(SRC[127-64]);
```

Intel C/C++ Compiler Intrinsic Equivalent

SQRTPD	__m128d _mm_sqrt_pd (m128d a)

SIMD Floating-Point Exceptions

Invalid, Precision, Denormal.

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
	If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0.
	If EM in CR0 is set.

	If OSFXSR in CR4 is 0.
	If CPUID feature flag SSE2 is 0.

Real-Address Mode Exceptions

#GP(0)	If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0.
	If EM in CR0 is set.
	If OSFXSR in CR4 is 0.
	If CPUID feature flag SSE2 is 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.

SQRTPS—Compute Square Roots of Packed Single-Precision Floating-Point Values

Opcode	Instruction	Description
0F 51 /r	SQRTPS *xmm1*, xmm2/m128	Compute the square root of the packed single-precision floating-point values in *xmm2/m128* and store the results in *xmm1.*

Description

Performs an SIMD computation of the square roots of the four packed single-precision floating-point values in the source operand (second operand) stores the packed single-precision floating-point results in the destination operand. The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. See Figure 10-5 in the *IA-32 Intel Architecture Software Developer's Manual, Volume 1* for an illustration of an SIMD single-precision floating-point operation.

Operation

```
DEST[31-0] ← SQRT(SRC[31-0]);
DEST[63-32] ← SQRT(SRC[63-32]);
DEST[95-64] ← SQRT(SRC[95-64]);
DEST[127-96] ← SQRT(SRC[127-96]);
```

Intel C/C++ Compiler Intrinsic Equivalent

SQRTPS	__m128 _mm_sqrt_ps(__m128 a)

SIMD Floating-Point Exceptions

Invalid, Precision, Denormal.

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
	If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.

#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0.
	If EM in CR0 is set.
	If OSFXSR in CR4 is 0.
	If CPUID feature flag SSE is 0.

Real-Address Mode Exceptions

#GP(0)	If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0.
	If EM in CR0 is set.
	If OSFXSR in CR4 is 0.
	If CPUID feature flag SSE is 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.

SQRTSD—Compute Square Root of Scalar Double-Precision Floating-Point Value

Opcode	Instruction	Description
F2 0F 51 /r	SQRTSD *xmm1*, *xmm2/m64*	Compute the square root of the low double-precision floating-point value in *xmm2/m64* and store the results in *xmm1*.

Description

Computes the square root of the low double-precision floating-point value in the source operand (second operand) and stores the double-precision floating-point result in the destination operand. The source operand can be an XMM register or a 64-bit memory location. The destination operand is an XMM register. The high quadword of the destination operand remains unchanged. See Figure 11-4 in the *IA-32 Intel Architecture Software Developer's Manual, Volume 1* for an illustration of a scalar double-precision floating-point operation.

Operation

```
DEST[63-0] ← SQRT(SRC[63-0]);
* DEST[127-64] remains unchanged *;
```

Intel C/C++ Compiler Intrinsic Equivalent

SQRTSD	__m128d _mm_sqrt_sd (m128d a)

SIMD Floating-Point Exceptions

Invalid, Precision, Denormal.

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0.
	If EM in CR0 is set.

	If OSFXSR in CR4 is 0.
	If CPUID feature flag SSE2 is 0.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

GP(0)	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0.
	If EM in CR0 is set.
	If OSFXSR in CR4 is 0.
	If CPUID feature flag SSE2 is 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

SQRTSS—Compute Square Root of Scalar Single-Precision Floating-Point Value

Opcode	Instruction	Description
F3 0F 51 /r	SQRTSS *xmm1*, *xmm2/m32*	Compute the square root of the low single-precision floating-point value in *xmm2/m32* and store the results in *xmm1*.

Description

Computes the square root of the low single-precision floating-point value in the source operand (second operand) and stores the single-precision floating-point result in the destination operand. The source operand can be an XMM register or a 32-bit memory location. The destination operand is an XMM register. The three high-order doublewords of the destination operand remain unchanged. See Figure 10-6 in the *IA-32 Intel Architecture Software Developer's Manual, Volume 1* for an illustration of a scalar single-precision floating-point operation.

Operation

```
DEST[31-0] ← SQRT (SRC[31-0]);
* DEST[127-64] remains unchanged *;
```

Intel C/C++ Compiler Intrinsic Equivalent

```
SQRTSS          __m128 _mm_sqrt_ss(__m128 a)
```

SIMD Floating-Point Exceptions

Invalid, Precision, Denormal.

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0.
	If EM in CR0 is set.

	If OSFXSR in CR4 is 0.
	If CPUID feature flag SSE is 0.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

GP(0)	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0.
	If EM in CR0 is set.
	If OSFXSR in CR4 is 0.
	If CPUID feature flag SSE is 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

STC—Set Carry Flag

Opcode	Instruction	Description
F9	STC	Set CF flag.

Description

Sets the CF flag in the EFLAGS register.

Operation

```
CF ← 1;
```

Flags Affected

The CF flag is set. The OF, ZF, SF, AF, and PF flags are unaffected.

Exceptions (All Operating Modes)

None.

STD—Set Direction Flag

Opcode	Instruction	Description
FD	STD	Set DF flag.

Description

Sets the DF flag in the EFLAGS register. When the DF flag is set to 1, string operations decrement the index registers (ESI and/or EDI).

Operation

```
DF ← 1;
```

Flags Affected

The DF flag is set. The CF, OF, ZF, SF, AF, and PF flags are unaffected.

Exceptions (All Operating Modes)

None.

STI—Set Interrupt Flag

Opcode	Instruction	Description
FB	STI	Set interrupt flag; external, maskable interrupts enabled at the end of the next instruction.

Description

If protected-mode virtual interrupts are not enabled, STI sets the interrupt flag (IF) in the EFLAGS register. After the IF flag is set, the processor begins responding to external, maskable interrupts after the next instruction is executed. The delayed effect of this instruction is provided to allow interrupts to be enabled just before returning from a procedure (or subroutine). For instance, if an STI instruction is followed by an RET instruction, the RET instruction is allowed to execute before external interrupts are recognized[2]. If the STI instruction is followed by a CLI instruction (which clears the IF flag), the effect of the STI instruction is negated.

The IF flag and the STI and CLI instructions do not prohibit the generation of exceptions and NMI interrupts. NMI interrupts may be blocked for one macroinstruction following an STI.

When protected-mode virtual interrupts are enabled, CPL is 3, and IOPL is less than 3; STI sets the VIF flag in the EFLAGS register, leaving IF unaffected.

Table 4-2 indicates the action of the STI instruction depending on the processor's mode of operation and the CPL/IOPL settings of the running program or procedure.

2. Note that in a sequence of instructions that individually delay interrupts past the following instruction, only the first instruction in the sequence is guaranteed to delay the interrupt, but subsequent interrupt-delaying instructions may not delay the interrupt. Thus, in the following instruction sequence:

```
STI
MOV SS, AX
MOV ESP, EBP
```

interrupts may be recognized before MOV ESP, EBP executes, even though MOV SS, AX normally delays interrupts for one instruction.

Table 4-2. Decision Table for STI Results

PE	VM	IOPL	CPL	PVI	VIP	VME	STI Result
0	X	X	X	X	X	X	**IF = 1**
1	0	≥ CPL	X	X	X	X	**IF = 1**
1	0	< CPL	3	1	0	X	**VIF = 1**
1	0	< CPL	< 3	X	X	X	**GP Fault**
1	0	< CPL	X	0	X	X	**GP Fault**
1	0	< CPL	X	X	1	X	**GP Fault**
1	1	3	X	X	X	X	**IF = 1**
1	1	< 3	X	X	0	1	**VIF = 1**
1	1	< 3	X	X	1	X	**GP Fault**
1	1	< 3	X	X	X	0	**GP Fault**
X = This setting has no impact.							

Operation

```
IF PE = 0  (* Executing in real-address mode *)
    THEN
        IF ← 1; (* Set Interrupt Flag *)
    ELSE  (* Executing in protected mode or virtual-8086 mode *)
        IF VM = 0  (* Executing in protected mode*)
            THEN
                IF IOPL ≥ CPL
                    THEN
                        IF ← 1;  (* Set Interrupt Flag *)
                ELSE
                    IF (IOPL < CPL) AND (CPL = 3) AND (VIP = 0)
                        THEN
                            VIF ← 1;  (* Set Virtual Interrupt Flag *)
                        ELSE
                            #GP(0);
                    FI;
                FI;
            ELSE  (* Executing in Virtual-8086 mode *)
                IF IOPL = 3
                    THEN
                        IF ← 1;  (* Set Interrupt Flag *)
                ELSE
                    IF ((IOPL < 3) AND (VIP = 0) AND (VME = 1))
                        THEN
                            VIF ← 1;  (* Set Virtual Interrupt Flag *)
                    ELSE
                        #GP(0); (* Trap to virtual-8086 monitor *)
                    FI;)
               FI;
          FI;
FI;
```

Flags Affected

The IF flag is set to 1; or the VIF flag is set to 1.

Protected Mode Exceptions

#GP(0)	If the CPL is greater (has less privilege) than the IOPL of the current program or procedure.

Real-Address Mode Exceptions

None.

Virtual-8086 Mode Exceptions

#GP(0)	If the CPL is greater (has less privilege) than the IOPL of the current program or procedure.

STMXCSR—Store MXCSR Register State

Opcode	Instruction	Description
0F AE /3	STMXCSR *m32*	Store contents of MXCSR register to *m32*.

Description

Stores the contents of the MXCSR control and status register to the destination operand. The destination operand is a 32-bit memory location. The reserved bits in the MXCSR register are stored as 0s.

Operation

```
m32 ← MXCSR;
```

Intel C/C++ Compiler Intrinsic Equivalent

_mm_getcsr(void)

Exceptions

None.

Numeric Exceptions

None.

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#UD	If CR0.EM = 1.
#NM	If TS bit in CR0 is set.
#AC	For unaligned memory reference. To enable #AC exceptions, three conditions must be true(CR0.AM is set; EFLAGS.AC is set; current CPL is 3).
#UD	If CR4.OSFXSR(bit 9) = 0.
	If CPUID.SSE(EDX bit 25) = 0.

Real Address Mode Exceptions

GP(0)	If any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.
#UD	If CR0.EM = 1.
#NM	If TS bit in CR0 is set.
#UD	If CR4.OSFXSR(bit 9) = 0.
	If CPUID.SSE(EDX bit 25) = 0.

Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode.

#PF(fault-code)	For a page fault.
#AC	For unaligned memory reference.

STOS/STOSB/STOSW/STOSD—Store String

Opcode	Instruction	Description
AA	STOS m8	Store AL at address ES:(E)DI.
AB	STOS m16	Store AX at address ES:(E)DI.
AB	STOS m32	Store EAX at address ES:(E)DI.
AA	STOSB	Store AL at address ES:(E)DI.
AB	STOSW	Store AX at address ES:(E)DI.
AB	STOSD	Store EAX at address ES:(E)DI.

Description

Stores a byte, word, or doubleword from the AL, AX, or EAX register, respectively, into the destination operand. The destination operand is a memory location, the address of which is read from either the ES:EDI or the ES:DI registers (depending on the address-size attribute of the instruction, 32 or 16, respectively). The ES segment cannot be overridden with a segment override prefix.

At the assembly-code level, two forms of this instruction are allowed: the “explicit-operands” form and the “no-operands” form. The explicit-operands form (specified with the STOS mnemonic) allows the destination operand to be specified explicitly. Here, the destination operand should be a symbol that indicates the size and location of the destination value. The source operand is then automatically selected to match the size of the destination operand (the AL register for byte operands, AX for word operands, and EAX for doubleword operands). This explicit-operands form is provided to allow documentation; however, note that the documentation provided by this form can be misleading. That is, the destination operand symbol must specify the correct **type** (size) of the operand (byte, word, or doubleword), but it does not have to specify the correct **location**. The location is always specified by the ES:(E)DI registers, which must be loaded correctly before the store string instruction is executed.

The no-operands form provides “short forms” of the byte, word, and doubleword versions of the STOS instructions. Here also ES:(E)DI is assumed to be the destination operand and the AL, AX, or EAX register is assumed to be the source operand. The size of the destination and source operands is selected with the mnemonic: STOSB (byte read from register AL), STOSW (word from AX), or STOSD (doubleword from EAX).

After the byte, word, or doubleword is transferred from the AL, AX, or EAX register to the memory location, the (E)DI register is incremented or decremented automatically according to the setting of the DF flag in the EFLAGS register. (If the DF flag is 0, the (E)DI register is incremented; if the DF flag is 1, the (E)DI register is decremented.) The (E)DI register is incremented or decremented by 1 for byte operations, by 2 for word operations, or by 4 for doubleword operations.

The STOS, STOSB, STOSW, and STOSD instructions can be preceded by the REP prefix for block loads of ECX bytes, words, or doublewords. More often, however, these instructions are used within a LOOP construct because data needs to be moved into the AL, AX, or EAX register before it can be stored. See “REP/REPE/REPZ/REPNE /REPNZ—Repeat String Operation Prefix” in this chapter for a description of the REP prefix.

Operation

```
IF (byte store)
   THEN
      DEST ← AL;
         THEN IF DF = 0
            THEN (E)DI ← (E)DI + 1;
            ELSE (E)DI ← (E)DI – 1;
         FI;
   ELSE IF (word store)
      THEN
         DEST ← AX;
            THEN IF DF = 0
               THEN (E)DI ← (E)DI + 2;
               ELSE (E)DI ← (E)DI – 2;
            FI;
      ELSE (* doubleword store *)
         DEST ← EAX;
            THEN IF DF = 0
               THEN (E)DI ← (E)DI + 4;
               ELSE (E)DI ← (E)DI – 4;
            FI;
   FI;
FI;
```

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If the destination is located in a non-writable segment.
	If a memory operand effective address is outside the limit of the ES segment.
	If the ES register contains a null segment selector.
#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.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the ES segment limit.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the ES segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

STR—Store Task Register

Opcode	Instruction	Description
0F 00 /1	STR *r/m16*	Stores segment selector from TR in *r/m16.*

Description

Stores the segment selector from the task register (TR) in the destination operand. The destination operand can be a general-purpose register or a memory location. The segment selector stored with this instruction points to the task state segment (TSS) for the currently running task.

When the destination operand is a 32-bit register, the 16-bit segment selector is copied into the lower 16 bits of the register and the upper 16 bits of the register are cleared. When the destination operand is a memory location, the segment selector is written to memory as a 16-bit quantity, regardless of operand size.

The STR instruction is useful only in operating-system software. It can only be executed in protected mode.

Operation

```
DEST ← TR(SegmentSelector);
```

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If the destination is a memory operand that is located in a non-writable segment or if the effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a null 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.

Real-Address Mode Exceptions

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

Virtual-8086 Mode Exceptions

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

SUB—Subtract

Opcode	Instruction	Description
2C *ib*	SUB AL,*imm8*	Subtract *imm8* from AL.
2D *iw*	SUB AX,*imm16*	Subtract *imm16* from AX.
2D *id*	SUB EAX,*imm32*	Subtract *imm32* from EAX.
80 /5 *ib*	SUB *r/m8,imm8*	Subtract *imm8* from *r/m8.*
81 /5 *iw*	SUB *r/m16,imm16*	Subtract *imm16* from *r/m16.*
81 /5 *id*	SUB *r/m32,imm32*	Subtract *imm32* from *r/m32.*
83 /5 *ib*	SUB *r/m16,imm8*	Subtract sign-extended *imm8* from *r/m16.*
83 /5 *ib*	SUB *r/m32,imm8*	Subtract sign-extended *imm8* from *r/m32.*
28 /*r*	SUB *r/m8,r8*	Subtract *r8* from *r/m8.*
29 /*r*	SUB *r/m16,r16*	Subtract *r16* from *r/m16.*
29 /*r*	SUB *r/m32,r32*	Subtract *r32* from *r/m32.*
2A /*r*	SUB *r8,r/m8*	Subtract *r/m8* from *r8.*
2B /*r*	SUB *r16,r/m16*	Subtract *r/m16* from *r16.*
2B /*r*	SUB *r32,r/m32*	Subtract *r/m32* from *r32.*

Description

Subtracts the second operand (source operand) from the first operand (destination operand) and stores the result in the destination operand. The destination operand can be a register or a memory location; the source operand can be an immediate, register, or memory location. (However, two memory operands cannot be used in one instruction.) When an immediate value is used as an operand, it is sign-extended to the length of the destination operand format.

The SUB instruction performs integer subtraction. It evaluates the result for both signed and unsigned integer operands and sets the OF and CF flags to indicate an overflow in the signed or unsigned result, respectively. The SF flag indicates the sign of the signed result.

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

Operation

```
DEST ← DEST – SRC;
```

Flags Affected

The OF, SF, ZF, AF, PF, and CF flags are set according to the result.

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.

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.

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.

SUBPD—Subtract Packed Double-Precision Floating-Point Values

Opcode	Instruction	Description
66 0F 5C /r	SUBPD *xmm1, xmm2/m128*	Subtract packed double-precision floating-point values in *xmm2/m128* from *xmm1*.

Description

Performs an SIMD subtract of the two packed double-precision floating-point values in the source operand (second operand) from the two packed double-precision floating-point values in the destination operand (first operand), and stores the packed double-precision floating-point results in the destination operand. The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. See Figure 11-3 in the *IA-32 Intel Architecture Software Developer's Manual, Volume 1* for an illustration of an SIMD double-precision floating-point operation.

Operation

```
DEST[63-0] ← DEST[63-0] − SRC[63-0];
DEST[127-64] ← DEST[127-64] − SRC[127-64];
```

Intel C/C++ Compiler Intrinsic Equivalent

```
SUBPD          __m128d _mm_sub_pd (m128d a, m128d b)
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
	If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0.
	If EM in CR0 is set.

	If OSFXSR in CR4 is 0.
	If CPUID feature flag SSE2 is 0.

Real-Address Mode Exceptions

#GP(0)	If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0.
	If EM in CR0 is set.
	If OSFXSR in CR4 is 0.
	If CPUID feature flag SSE2 is 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.

SUBPS—Subtract Packed Single-Precision Floating-Point Values

Opcode	Instruction	Description
0F 5C /r	SUBPS *xmm1 xmm2/m128*	Subtract packed single-precision floating-point values in *xmm2/mem* from *xmm1*.

Description

Performs an SIMD subtract of the four packed single-precision floating-point values in the source operand (second operand) from the four packed single-precision floating-point values in the destination operand (first operand), and stores the packed single-precision floating-point results in the destination operand. The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. See Figure 10-5 in the *IA-32 Intel Architecture Software Developer's Manual, Volume 1* for an illustration of an SIMD double-precision floating-point operation.

Operation

```
DEST[31-0] ← DEST[31-0] – SRC[31-0];
DEST[63-32] ← DEST[63-32] – SRC[63-32];
DEST[95-64] ← DEST[95-64] – SRC[95-64];
DEST[127-96] ← DEST[127-96] – SRC[127-96];
```

Intel C/C++ Compiler Intrinsic Equivalent

```
SUBPS          __m128 _mm_sub_ps(__m128 a, __m128 b)
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
	If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.

#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0.
	If EM in CR0 is set.
	If OSFXSR in CR4 is 0.
	If CPUID feature flag SSE is 0.

Real-Address Mode Exceptions

#GP(0)	If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0.
	If EM in CR0 is set.
	If OSFXSR in CR4 is 0.
	If CPUID feature flag SSE is 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.

SUBSD—Subtract Scalar Double-Precision Floating-Point Values

Opcode	Instruction	Description
F2 0F 5C /r	SUBSD *xmm1*, xmm2/m64	Subtract the low double-precision floating-point value in *xmm2/mem64* from *xmm1*.

Description

Subtracts the low double-precision floating-point value in the source operand (second operand) from the low double-precision floating-point value in the destination operand (first operand), and stores the double-precision floating-point result in the destination operand. The source operand can be an XMM register or a 64-bit memory location. The destination operand is an XMM register. The high quadword of the destination operand remains unchanged. See Figure 11-4 in the *IA-32 Intel Architecture Software Developer's Manual, Volume 1* for an illustration of a scalar double-precision floating-point operation.

Operation

```
DEST[63-0] ← DEST[63-0] − SRC[63-0];
* DEST[127-64] remains unchanged *;
```

Intel C/C++ Compiler Intrinsic Equivalent

SUBSD	__m128d _mm_sub_sd (m128d a, m128d b)

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0.
	If EM in CR0 is set.
	If OSFXSR in CR4 is 0.
	If CPUID feature flag SSE2 is 0.

#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

GP(0)	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0.
	If EM in CR0 is set.
	If OSFXSR in CR4 is 0.
	If CPUID feature flag SSE2 is 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

SUBSS—Subtract Scalar Single-Precision Floating-Point Values

Opcode	Instruction	Description
F3 0F 5C /r	SUBSS *xmm1, xmm2/m32*	Subtract the lower single-precision floating-point values in *xmm2/m32* from *xmm1*.

Description

Subtracts the low single-precision floating-point value in the source operand (second operand) from the low single-precision floating-point value in the destination operand (first operand), and stores the single-precision floating-point result in the destination operand. The source operand can be an XMM register or a 32-bit memory location. The destination operand is an XMM register. The three high-order doublewords of the destination operand remain unchanged. See Figure 10-6 in the *IA-32 Intel Architecture Software Developer's Manual, Volume 1* for an illustration of a scalar single-precision floating-point operation.

Operation

```
DEST[31-0] ← DEST[31-0] - SRC[31-0];
* DEST[127-96] remains unchanged *;
```

Intel C/C++ Compiler Intrinsic Equivalent

```
SUBSS          __m128 _mm_sub_ss(__m128 a, __m128 b)
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0.
	If EM in CR0 is set.
	If OSFXSR in CR4 is 0.
	If CPUID feature flag SSE is 0.

#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

GP(0)	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0.
	If EM in CR0 is set.
	If OSFXSR in CR4 is 0.
	If CPUID feature flag SSE is 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

SYSENTER—Fast System Call

Opcode	Instruction	Description
0F 34	SYSENTER	Fast call to privilege level 0 system procedures.

Description

Executes a fast call to a level 0 system procedure or routine. This instruction is a companion instruction to the SYSEXIT instruction. The SYSENTER instruction is optimized to provide the maximum performance for system calls from user code running at privilege level 3 to operating system or executive procedures running at privilege level 0.

Prior to executing the SYSENTER instruction, software must specify the privilege level 0 code segment and code entry point, and the privilege level 0 stack segment and stack pointer by writing values into the following MSRs:

- SYSENTER_CS_MSR—Contains the 32-bit segment selector for the privilege level 0 code segment. (This value is also used to compute the segment selector of the privilege level 0 stack segment.)
- SYSENTER_EIP_MSR—Contains the 32-bit offset into the privilege level 0 code segment to the first instruction of the selected operating procedure or routine.
- SYSENTER_ESP_MSR—Contains the 32-bit stack pointer for the privilege level 0 stack.

These MSRs can be read from and written to using the RDMSR and WRMSR instructions. The register addresses are listed in Table 4-3. These addresses are defined to remain fixed for future IA-32 processors.

Table 4-3. MSRs Used By the SYSENTER and SYSEXIT Instructions

MSR	Address
SYSENTER_CS_MSR	174H
SYSENTER_ESP_MSR	175H
SYSENTER_EIP_MSR	176H

When the SYSENTER instruction is executed, the processor does the following:

1. Loads the segment selector from the SYSENTER_CS_MSR into the CS register.
2. Loads the instruction pointer from the SYSENTER_EIP_MSR into the EIP register.
3. Adds 8 to the value in SYSENTER_CS_MSR and loads it into the SS register.
4. Loads the stack pointer from the SYSENTER_ESP_MSR into the ESP register.
5. Switches to privilege level 0.
6. Clears the VM flag in the EFLAGS register, if the flag is set.
7. Begins executing the selected system procedure.

The processor does not save a return IP or other state information for the calling procedure.

The SYSENTER instruction always transfers program control to a protected-mode code segment with a DPL of 0. The instruction requires that the following conditions are met by the operating system:

- The segment descriptor for the selected system code segment selects a flat, 32-bit code segment of up to 4 GBytes, with execute, read, accessed, and non-conforming permissions.
- The segment descriptor for selected system stack segment selects a flat 32-bit stack segment of up to 4 GBytes, with read, write, accessed, and expand-up permissions.

The SYSENTER can be invoked from all operating modes except real-address mode.

The SYSENTER and SYSEXIT instructions are companion instructions, but they do not constitute a call/return pair. When executing a SYSENTER instruction, the processor does not save state information for the user code, and neither the SYSENTER nor the SYSEXIT instruction supports passing parameters on the stack.

To use the SYSENTER and SYSEXIT instructions as companion instructions for transitions between privilege level 3 code and privilege level 0 operating system procedures, the following conventions must be followed:

- The segment descriptors for the privilege level 0 code and stack segments and for the privilege level 3 code and stack segments must be contiguous in the global descriptor table. This convention allows the processor to compute the segment selectors from the value entered in the SYSENTER_CS_MSR MSR.
- The fast system call "stub" routines executed by user code (typically in shared libraries or DLLs) must save the required return IP and processor state information if a return to the calling procedure is required. Likewise, the operating system or executive procedures called with SYSENTER instructions must have access to and use this saved return and state information when returning to the user code.

The SYSENTER and SYSEXIT instructions were introduced into the IA-32 architecture in the Pentium II processor. The availability of these instructions on a processor is indicated with the SYSENTER/SYSEXIT present (SEP) feature flag returned to the EDX register by the CPUID instruction. An operating system that qualifies the SEP flag must also qualify the processor family and model to ensure that the SYSENTER/SYSEXIT instructions are actually present. For example:

```
IF (CPUID SEP bit is set)
   THEN IF (Family = 6) AND (Model < 3) AND (Stepping < 3)
        THEN
            SYSENTER/SYSEXIT_Not_Supported
        FI;
   ELSE SYSENTER/SYSEXIT_Supported
FI;
```

When the CPUID instruction is executed on the Pentium Pro processor (model 1), the processor returns a the SEP flag as set, but does not support the SYSENTER/SYSEXIT instructions.

Operation

```
IF CR0.PE = 0 THEN #GP(0); FI;
IF SYSENTER_CS_MSR = 0 THEN #GP(0); FI;

EFLAGS.VM ← 0                        (* Insures protected mode execution *)
EFLAGS.IF ← 0                        (* Mask interrupts *)
EFLAGS.RF ← 0

CS.SEL ← SYSENTER_CS_MSR             (* Operating system provides CS *)
(* Set rest of CS to a fixed value *)
CS.SEL.CPL ← 0
CS.BASE ← 0                          (* Flat segment *)

CS.LIMIT ← FFFFH                     (* 4 GByte limit *)
CS.ARbyte.G ← 1                      (* 4 KByte granularity *)
CS.ARbyte.S ← 1
CS.ARbyte.TYPE ← 1011B               (* Execute + Read, Accessed *)
CS.ARbyte.D ← 1                      (* 32-bit code segment*)
CS.ARbyte.DPL ← 0
CS.ARbyte.RPL ← 0
CS.ARbyte.P ← 1

SS.SEL ← CS.SEL + 8
(* Set rest of SS to a fixed value *)
SS.BASE ← 0                          (* Flat segment *)
SS.LIMIT ← FFFFH                     (* 4 GByte limit *)
SS.ARbyte.G ← 1                      (* 4 KByte granularity *)
SS.ARbyte.S ←
SS.ARbyte.TYPE ← 0011B               (* Read/Write, Accessed *)
SS.ARbyte.D ← 1                      (* 32-bit stack segment*)
SS.ARbyte.DPL ← 0
SS.ARbyte.RPL ← 0
SS.ARbyte.P ← 1

ESP ← SYSENTER_ESP_MSR
EIP ← SYSENTER_EIP_MSR
```

Flags Affected

VM, IF, RF (see Operation above)

Protected Mode Exceptions

#GP(0)	If SYSENTER_CS_MSR contains zero.

Real-Address Mode Exceptions

#GP(0)	If protected mode is not enabled.

Virtual-8086 Mode Exceptions

#GP(0)	If SYSENTER_CS_MSR contains zero.

SYSEXIT—Fast Return from Fast System Call

Opcode	Instruction	Description
0F 35	SYSEXIT	Fast return to privilege level 3 user code.

Description

Executes a fast return to privilege level 3 user code. This instruction is a companion instruction to the SYSENTER instruction. The SYSEXIT instruction is optimized to provide the maximum performance for returns from system procedures executing at protections levels 0 to user procedures executing at protection level 3. This instruction must be executed from code executing at privilege level 0.

Prior to executing the SYSEXIT instruction, software must specify the privilege level 3 code segment and code entry point, and the privilege level 3 stack segment and stack pointer by writing values into the following MSR and general-purpose registers:

- SYSENTER_CS_MSR—Contains the 32-bit segment selector for the privilege level 0 code segment in which the processor is currently executing. (This value is used to compute the segment selectors for the privilege level 3 code and stack segments.)
- EDX—Contains the 32-bit offset into the privilege level 3 code segment to the first instruction to be executed in the user code.
- ECX—Contains the 32-bit stack pointer for the privilege level 3 stack.

The SYSENTER_CS_MSR MSR can be read from and written to using the RDMSR and WRMSR instructions. The register address is listed in Table 4-3. This address is defined to remain fixed for future IA-32 processors.

When the SYSEXIT instruction is executed, the processor does the following:

1. Adds 16 to the value in SYSENTER_CS_MSR and loads the sum into the CS selector register.
2. Loads the instruction pointer from the EDX register into the EIP register.
3. Adds 24 to the value in SYSENTER_CS_MSR and loads the sum into the SS selector register.
4. Loads the stack pointer from the ECX register into the ESP register.
5. Switches to privilege level 3.
6. Begins executing the user code at the EIP address.

See “SYSENTER—Fast System Call” for information about using the SYSENTER and SYSEXIT instructions as companion call and return instructions.

The SYSEXIT instruction always transfers program control to a protected-mode code segment with a DPL of 3. The instruction requires that the following conditions are met by the operating system:

- The segment descriptor for the selected user code segment selects a flat, 32-bit code segment of up to 4 GBytes, with execute, read, accessed, and non-conforming permissions.
- The segment descriptor for selected user stack segment selects a flat, 32-bit stack segment of up to 4 GBytes, with expand-up, read, write, and accessed permissions.

The SYSENTER can be invoked from all operating modes except real-address mode.

The SYSENTER and SYSEXIT instructions were introduced into the IA-32 architecture in the Pentium II processor. The availability of these instructions on a processor is indicated with the SYSENTER/SYSEXIT present (SEP) feature flag returned to the EDX register by the CPUID instruction. An operating system that qualifies the SEP flag must also qualify the processor family and model to ensure that the SYSENTER/SYSEXIT instructions are actually present. For example:

```
IF (CPUID SEP bit is set)
   THEN IF (Family = 6) AND (Model < 3) AND (Stepping < 3)
        THEN
            SYSENTER/SYSEXIT_Not_Supported
        FI;
   ELSE SYSENTER/SYSEXIT_Supported
FI;
```

When the CPUID instruction is executed on the Pentium Pro processor (model 1), the processor returns a the SEP flag as set, but does not support the SYSENTER/SYSEXIT instructions.

Operation

```
IF SYSENTER_CS_MSR = 0 THEN #GP(0); FI;
IF CR0.PE = 0 THEN #GP(0); FI;
IF CPL ≠ 0 THEN #GP(0)

CS.SEL ← (SYSENTER_CS_MSR + 16)    (* Segment selector for return CS *)
(* Set rest of CS to a fixed value *)
CS.BASE ← 0                        (* Flat segment *)
CS.LIMIT ← FFFFH                   (* 4 GByte limit *)
CS.ARbyte.G ← 1                    (* 4 KByte granularity *)
CS.ARbyte.S ← 1
CS.ARbyte.TYPE ← 1011B             (* Execute, Read, Non-Conforming Code *)
CS.ARbyte.D ← 1                    (* 32-bit code segment*)
CS.ARbyte.DPL ← 3
CS.ARbyte.RPL ← 3
CS.ARbyte.P ← 1

SS.SEL ← (SYSENTER_CS_MSR + 24)    (* Segment selector for return SS *)
(* Set rest of SS to a fixed value *)
```

```
SS.BASE ← 0                          (* Flat segment *)
SS.LIMIT ← FFFFH                     (* 4 GByte limit *)
SS.ARbyte.G ← 1                      (* 4 KByte granularity *)
SS.ARbyte.S ←
SS.ARbyte.TYPE ← 0011B               (* Expand Up, Read/Write, Data *)
SS.ARbyte.D ← 1                      (* 32-bit stack segment*)
SS.ARbyte.DPL ← 3
SS.ARbyte.RPL ← 3
SS.ARbyte.P ← 1

ESP    ← ECX
EIP    ← EDX
```

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If SYSENTER_CS_MSR contains zero.

Real-Address Mode Exceptions

#GP(0)	If protected mode is not enabled.

Virtual-8086 Mode Exceptions

#GP(0)	If SYSENTER_CS_MSR contains zero.

TEST—Logical Compare

Opcode	Instruction	Description
A8 *ib*	TEST AL,*imm8*	AND *imm8* with AL; set SF, ZF, PF according to result.
A9 *iw*	TEST AX,*imm16*	AND *imm16* with AX; set SF, ZF, PF according to result.
A9 *id*	TEST EAX,*imm32*	AND *imm32* with EAX; set SF, ZF, PF according to result.
F6 /0 *ib*	TEST *r/m8,imm8*	AND *imm8* with *r/m8*; set SF, ZF, PF according to result.
F7 /0 *iw*	TEST *r/m16,imm16*	AND *imm16* with *r/m16*; set SF, ZF, PF according to result.
F7 /0 *id*	TEST *r/m32,imm32*	AND *imm32* with *r/m32*; set SF, ZF, PF according to result.
84 */r*	TEST *r/m8,r8*	AND *r8* with *r/m8*; set SF, ZF, PF according to result.
85 */r*	TEST *r/m16,r16*	AND *r16* with *r/m16*; set SF, ZF, PF according to result.
85 */r*	TEST *r/m32,r32*	AND *r32* with *r/m32*; set SF, ZF, PF according to result.

Description

Computes the bit-wise logical AND of first operand (source 1 operand) and the second operand (source 2 operand) and sets the SF, ZF, and PF status flags according to the result. The result is then discarded.

Operation

```
TEMP ← SRC1 AND SRC2;
SF ← MSB(TEMP);
IF TEMP = 0
    THEN ZF ← 1;
    ELSE ZF ← 0;
FI:
PF ← BitwiseXNOR(TEMP[0:7]);
CF ← 0;
OF ← 0;
(*AF is Undefined*)
```

Flags Affected

The OF and CF flags are set to 0. The SF, ZF, and PF flags are set according to the result (see the “Operation” section above). The state of the AF flag is undefined.

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register 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.

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.

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.

UCOMISD—Unordered Compare Scalar Double-Precision Floating-Point Values and Set EFLAGS

Opcode	Instruction	Description
66 0F 2E /r	UCOMISD *xmm1*, *xmm2/m64*	Compare (unordered) the low double-precision floating-point values in *xmm1* and *xmm2/m64* and set EFLAGS accordingly.

Description

Performs and unordered compare of the double-precision floating-point values in the low quadwords of source operand 1 (first operand) and source operand 2 (second operand), and sets the ZF, PF, and CF flags in the EFLAGS register according to the result (unordered, greater than, less than, or equal). The OF, SF and AF flags in the EFLAGS register are set to 0. The unordered result is returned if either source operand is a NaN (QNaN or SNaN).

Source operand 1 is an XMM register; source operand 2 can be an XMM register or a 64 bit memory location.

The UCOMISD instruction differs from the COMISD instruction in that it signals an SIMD floating-point invalid operation exception (#I) only when a source operand is an SNaN. The COMISD instruction signals an invalid operation exception if a source operand is either a QNaN or an SNaN.

The EFLAGS register is not updated if an unmasked SIMD floating-point exception is generated.

Operation

```
RESULT ← UnorderedCompare(SRC1[63-0] <> SRC2[63-0]) {
* Set EFLAGS *CASE (RESULT) OF
    UNORDERED:          ZF,PF,CF ← 111;
    GREATER_THAN:       ZF,PF,CF ← 000;
    LESS_THAN:          ZF,PF,CF ← 001;
    EQUAL:              ZF,PF,CF ← 100;
ESAC;
OF,AF,SF ← 0;
```

Intel C/C++ Compiler Intrinsic Equivalent

```
int_mm_ucomieq_sd(__m128d a, __m128d b)
int_mm_ucomilt_sd(__m128d a, __m128d b)
int_mm_ucomile_sd(__m128d a, __m128d b)
int_mm_ucomigt_sd(__m128d a, __m128d b)
int_mm_ucomige_sd(__m128d a, __m128d b)
int_mm_ucomineq_sd(__m128d a, __m128d b)
```

SIMD Floating-Point Exceptions

Invalid (if SNaN operands), Denormal.

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0.
	If EM in CR0 is set.
	If OSFXSR in CR4 is 0.
	If CPUID feature flag SSE2 is 0.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

GP(0)	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0.
	If EM in CR0 is set.
	If OSFXSR in CR4 is 0.
	If CPUID feature flag SSE2 is 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

UCOMISS—Unordered Compare Scalar Single-Precision Floating-Point Values and Set EFLAGS

Opcode	Instruction	Description
0F 2E /r	UCOMISS *xmm1, xmm2/m32*	Compare lower single-precision floating-point value in *xmm1* register with lower single-precision floating-point value in *xmm2/mem* and set the status flags accordingly.

Description

Performs and unordered compare of the single-precision floating-point values in the low doublewords of the source operand 1 (first operand) and the source operand 2 (second operand), and sets the ZF, PF, and CF flags in the EFLAGS register according to the result (unordered, greater than, less than, or equal). In The OF, SF and AF flags in the EFLAGS register are set to 0. The unordered result is returned if either source operand is a NaN (QNaN or SNaN).

Source operand 1 is an XMM register; source operand 2 can be an XMM register or a 32 bit memory location.

The UCOMISS instruction differs from the COMISS instruction in that it signals an SIMD floating-point invalid operation exception (#I) only when a source operand is an SNaN. The COMISS instruction signals an invalid operation exception if a source operand is either a QNaN or an SNaN.

The EFLAGS register is not updated if an unmasked SIMD floating-point exception is generated.

Operation

```
RESULT ← UnorderedCompare(SRC1[63-0] <> SRC2[63-0]) {
* Set EFLAGS *CASE (RESULT) OF
    UNORDERED:          ZF,PF,CF ← 111;
    GREATER_THAN:       ZF,PF,CF ← 000;
    LESS_THAN:          ZF,PF,CF ← 001;
    EQUAL:              ZF,PF,CF ← 100;
ESAC;
OF,AF,SF ← 0;
```

Intel C/C++ Compiler Intrinsic Equivalent

```
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)
```

SIMD Floating-Point Exceptions

Invalid (if SNaN operands), Denormal.

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0.
	If EM in CR0 is set.
	If OSFXSR in CR4 is 0.
	If CPUID feature flag SSE is 0.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

GP(0)	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0.
	If EM in CR0 is set.
	If OSFXSR in CR4 is 0.
	If CPUID feature flag SSE is 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

UD2—Undefined Instruction

Opcode	Instruction	Description
0F 0B	UD2	Raise invalid opcode exception.

Description

Generates an invalid opcode. This instruction is provided for software testing to explicitly generate an invalid opcode. The opcode for this instruction is reserved for this purpose.

Other than raising the invalid opcode exception, this instruction is the same as the NOP instruction.

Operation

```
#UD (* Generates invalid opcode exception *);
```

Flags Affected

None.

Exceptions (All Operating Modes)

#UD	Instruction is guaranteed to raise an invalid opcode exception in all operating modes.

UNPCKHPD—Unpack and Interleave High Packed Double-Precision Floating-Point Values

Opcode	Instruction	Description
66 0F 15 /r	UNPCKHPD *xmm1*, *xmm2/m128*	Unpack and interleave double-precision floating-point values from the high quadwords of *xmm1* and *xmm2/m128*.

Description

Performs an interleaved unpack of the high double-precision floating-point values from the source operand (second operand) and the destination operand (first operand). See Figure 4-14. The source operand can be an XMM register or a 128-bit memory location; the destination operand is an XMM register.

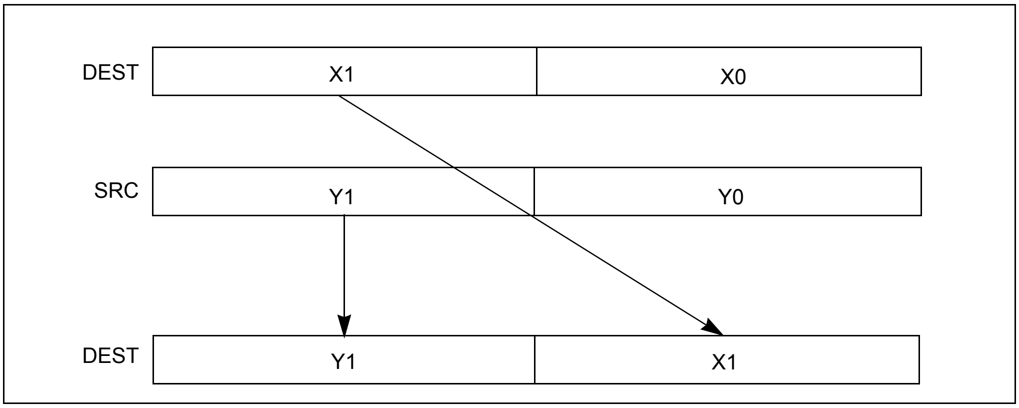


Figure 4-14. UNPCKHPD Instruction High Unpack and Interleave Operation

When unpacking from a memory operand, an implementation may fetch only the appropriate 64 bits; however, alignment to 16-byte boundary and normal segment checking will still be enforced.

Operation

```
DEST[63-0] ← DEST[127-64];
DEST[127-64]  ← SRC[127-64];
```

Intel C/C++ Compiler Intrinsic Equivalent

UNPCKHPD	__m128d _mm_unpackhi_pd(__m128d a, __m128d b)

SIMD Floating-Point Exceptions

None.

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
	If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS in CR0 is set.
#UD	If EM in CR0 is set.
	If OSFXSR in CR4 is 0.
	If CPUID feature flag SSE2 is 0.

Real-Address Mode Exceptions

#GP(0)	If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If TS in CR0 is set.
#UD	If EM in CR0 is set.
	If OSFXSR in CR4 is 0.
	If CPUID feature flag SSE2 is 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.

UNPCKHPS—Unpack and Interleave High Packed Single-Precision Floating-Point Values

Opcode	Instruction	Description
0F 15 /r	UNPCKHPS *xmm1, xmm2/m128*	Unpack and interleave the single-precision floating-point values from high quadwords of *xmm1* and *xmm2/mem* into *xmm1*.

Description

Performs an interleaved unpack of the high-order single-precision floating-point values from the source operand (second operand) and the destination operand (first operand). See Figure 4-15. The source operand can be an XMM register or a 128-bit memory location; the destination operand is an XMM register.

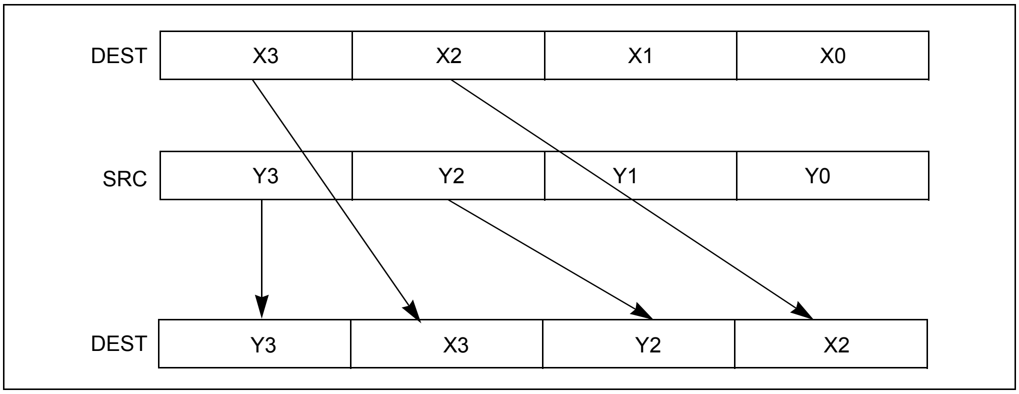


Figure 4-15. UNPCKHPS Instruction High Unpack and Interleave Operation

When unpacking from a memory operand, an implementation may fetch only the appropriate 64 bits; however, alignment to 16-byte boundary and normal segment checking will still be enforced.

Operation

```
DEST[31-0] ← DEST[95-64];
DEST[63-32] ← SRC[95-64];
DEST[95-64] ← DEST[127-96];
DEST[127-96] ← SRC[127-96];
```

Intel C/C++ Compiler Intrinsic Equivalent

```
UNPCKHPS      __m128 _mm_unpackhi_ps(__m128 a, __m128 b)
```

SIMD Floating-Point Exceptions

None.

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
	If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS in CR0 is set.
#UD	If EM in CR0 is set.
	If OSFXSR in CR4 is 0.
	If CPUID feature flag SSE is 0.

Real-Address Mode Exceptions

#GP(0)	If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If TS in CR0 is set.
#UD	If EM in CR0 is set.
	If OSFXSR in CR4 is 0.
	If CPUID feature flag SSE is 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.

UNPCKLPD—Unpack and Interleave Low Packed Double-Precision Floating-Point Values

Opcode	Instruction	Description
66 0F 14 /r	UNPCKLPD *xmm1*, *xmm2/m128*	Unpack and interleave the double-precision floating-point values from low quadwords of *xmm1* and *xmm2/m128*.

Description

Performs an interleaved unpack of the low double-precision floating-point values from the source operand (second operand) and the destination operand (first operand). See Figure 4-16. The source operand can be an XMM register or a 128-bit memory location; the destination operand is an XMM register.

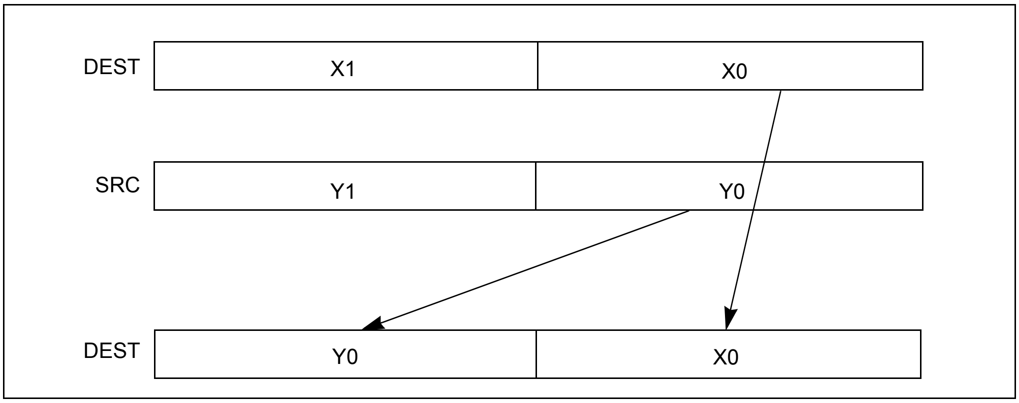


Figure 4-16. UNPCKLPD Instruction Low Unpack and Interleave Operation

When unpacking from a memory operand, an implementation may fetch only the appropriate 64 bits; however, alignment to 16-byte boundary and normal segment checking will still be enforced.

Operation

DEST[63-0] ← DEST[63-0];
DEST[127-64] ← SRC[63-0];

Intel C/C++ Compiler Intrinsic Equivalent

UNPCKHPD __m128d _mm_unpacklo_pd(__m128d a, __m128d b)

SIMD Floating-Point Exceptions

None.

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
	If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS in CR0 is set.
#UD	If EM in CR0 is set.
	If OSFXSR in CR4 is 0.
	If CPUID feature flag SSE2 is 0.

Real-Address Mode Exceptions

#GP(0)	If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If TS in CR0 is set.
#UD	If EM in CR0 is set.
	If OSFXSR in CR4 is 0.
	If CPUID feature flag SSE2 is 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.

UNPCKLPS—Unpack and Interleave Low Packed Single-Precision Floating-Point Values

Opcode	Instruction	Description
0F 14 /r	UNPCKLPS *xmm1, xmm2/m128*	Unpack and interleaves the single-precision floating-point values from low quadwords of *xmm1* and *xmm2/mem* into *xmm1*.

Performs an interleaved unpack of the low-order single-precision floating-point values from the source operand (second operand) and the destination operand (first operand). See Figure 4-17. The source operand can be an XMM register or a 128-bit memory location; the destination operand is an XMM register.

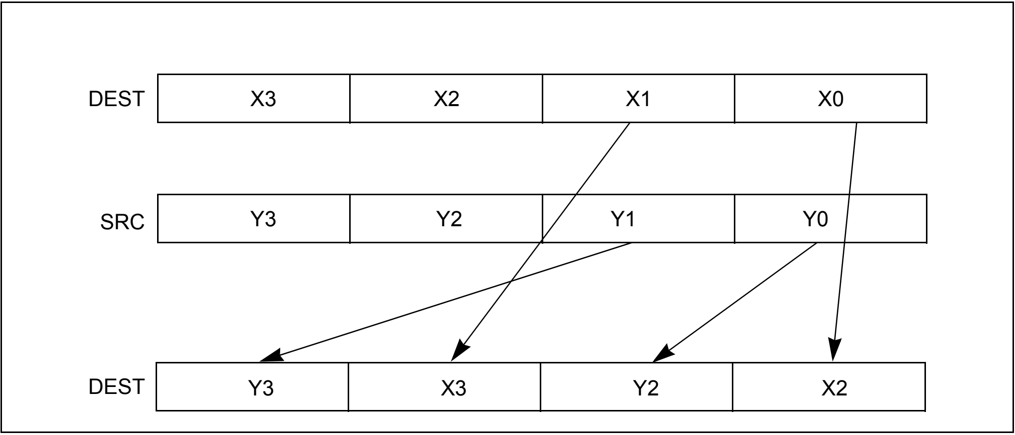


Figure 4-17. UNPCKLPS Instruction Low Unpack and Interleave Operation

When unpacking from a memory operand, an implementation may fetch only the appropriate 64 bits; however, alignment to 16-byte boundary and normal segment checking will still be enforced.

Operation

```
DEST[31-0] ← DEST[31-0];
DEST[63-32] ← SRC[31-0];
DEST[95-64] ← DEST[63-32];
DEST[127-96] ← SRC[63-32];
```

Intel C/C++ Compiler Intrinsic Equivalent

```
UNPCKLPS      __m128 _mm_unpacklo_ps(__m128 a, __m128 b)
```

SIMD Floating-Point Exceptions

None.

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
	If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS in CR0 is set.
#UD	If EM in CR0 is set.
	If OSFXSR in CR4 is 0.
	If CPUID feature flag SSE is 0.

Real-Address Mode Exceptions

#GP(0)	If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If TS in CR0 is set.
#UD	If EM in CR0 is set.
	If OSFXSR in CR4 is 0.
	If CPUID feature flag SSE is 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.

VERR, VERW—Verify a Segment for Reading or Writing

Opcode	Instruction	Description
0F 00 /4	VERR *r/m16*	Set ZF=1 if segment specified with *r/m16* can be read.
0F 00 /5	VERW *r/m16*	Set ZF=1 if segment specified with *r/m16* can be written.

Description

Verifies whether the code or data segment specified with the source operand is readable (VERR) or writable (VERW) from the current privilege level (CPL). The source operand is a 16-bit register or a memory location that contains the segment selector for the segment to be verified. If the segment is accessible and readable (VERR) or writable (VERW), the ZF flag is set; otherwise, the ZF flag is cleared. Code segments are never verified as writable. This check cannot be performed on system segments.

To set the ZF flag, the following conditions must be met:

- The segment selector is not null.
- The selector must denote a descriptor within the bounds of the descriptor table (GDT or LDT).
- The selector must denote the descriptor of a code or data segment (not that of a system segment or gate).
- For the VERR instruction, the segment must be readable.
- For the VERW instruction, the segment must be a writable data segment.
- If the segment is not a conforming code segment, the segment's DPL must be greater than or equal to (have less or the same privilege as) both the CPL and the segment selector's RPL.

The validation performed is the same as is performed when a segment selector is loaded into the DS, ES, FS, or GS register, and the indicated access (read or write) is performed. The segment selector's value cannot result in a protection exception, enabling the software to anticipate possible segment access problems.

Operation

```
IF SRC(Offset) > (GDTR(Limit) OR (LDTR(Limit))
        THEN
            ZF ← 0
Read segment descriptor;
IF SegmentDescriptor(DescriptorType) = 0 (* system segment *)
    OR (SegmentDescriptor(Type) ≠ conforming code segment)
    AND (CPL > DPL) OR (RPL > DPL)
        THEN
            ZF ← 0
        ELSE
            IF ((Instruction = VERR) AND (segment = readable))
                OR ((Instruction = VERW) AND (segment = writable))
                THEN
                    ZF ← 1;
            FI;
FI;
```

Flags Affected

The ZF flag is set to 1 if the segment is accessible and readable (VERR) or writable (VERW); otherwise, it is set to 0.

Protected Mode Exceptions

The only exceptions generated for these instructions are those related to illegal addressing of the source operand.

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a null 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.

Real-Address Mode Exceptions

#UD	The VERR and VERW instructions are not recognized in real-address mode.

Virtual-8086 Mode Exceptions

#UD	The VERR and VERW instructions are not recognized in virtual-8086 mode.

WAIT/FWAIT—Wait

Opcode	Instruction	Description
9B	WAIT	Check pending unmasked floating-point exceptions.
9B	FWAIT	Check pending unmasked floating-point exceptions.

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 insures 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 *IA-32 Intel Architecture Software Developer's Manual, Volume 1*, for more information on using the WAIT/FWAIT instruction.

Operation

```
CheckForPendingUnmaskedFloatingPointExceptions;
```

FPU Flags Affected

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

Floating-Point Exceptions

None.

Protected Mode Exceptions

#NM	MP and TS in CR0 is set.

Real-Address Mode Exceptions

#NM	MP and TS in CR0 is set.

Virtual-8086 Mode Exceptions

#NM	MP and TS in CR0 is set.

WBINVD—Write Back and Invalidate Cache

Opcode	Instruction	Description
0F 09	WBINVD	Write back and flush internal caches; initiate writing-back and flushing of external caches.

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 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 8 of the *IA-32 Intel Architecture Software Developer's Manual, Volume 3*).

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

IA-32 Architecture Compatibility

The WBINVD instruction is implementation dependent, and its function may be implemented differently on future 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);
```

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If the current privilege level is not 0.

Real-Address Mode Exceptions

None.

Virtual-8086 Mode Exceptions

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

WRMSR—Write to Model Specific Register

Opcode	Instruction	Description
0F 30	WRMSR	Write the value in EDX:EAX to MSR specified by ECX.

Description

Writes the contents of registers EDX:EAX into the 64-bit model specific register (MSR) specified in the ECX register. The input value loaded into the ECX register is the address of the MSR to be written to. 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. Undefined or reserved bits in an MSR should be set to the values previously read.

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

When the WRMSR instruction is used to write to an MTRR, the TLBs are invalidated, including the global entries (see "Translation Lookaside Buffers (TLBs)" in Chapter 3 of the *IA-32 Intel Architecture Software Developer's Manual, Volume 3*).

The MSRs control functions for testability, execution tracing, performance-monitoring and machine check errors. Appendix B, *Model-Specific Registers (MSRs)*, in the *IA-32 Intel Architecture Software Developer's Manual, Volume 3*, lists all the MSRs that can be read 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 8 of the *IA-32 Intel Architecture Software Developer's Manual, Volume 3*).

The CPUID instruction should be used to determine whether MSRs are supported (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.

Real-Address Mode Exceptions

#GP	If the value in ECX specifies a reserved or unimplemented MSR address.

Virtual-8086 Mode Exceptions

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

XADD—Exchange and Add

Opcode	Instruction	Description
0F C0 /r	XADD *r/m8, r8*	Exchange *r8* and *r/m8*; load sum into *r/m8*.
0F C1 /r	XADD *r/m16, r16*	Exchange *r16* and *r/m16*; load sum into *r/m16*.
0F C1 /r	XADD *r/m32, r32*	Exchange *r32* and *r/m32*; load sum into *r/m32*.

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.

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.

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.

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.

XCHG—Exchange Register/Memory with Register

Opcode	Instruction	Description
90+*rw*	XCHG AX, *16*	Exchange *r16* with AX.
90+*rw*	XCHG *r16*, X	Exchange AX with *r16.*
90+*rd*	XCHG EAX, *r32*	Exchange *r32* with EAX.
90+*rd*	XCHG *r32*, EAX	Exchange EAX with *r32.*
86 /*r*	XCHG *r/m8, r8*	Exchange *r8* (byte register) with byte from *r/m8.*
86 /*r*	XCHG *r8, r/m8*	Exchange byte from *r/m8* with *r8* (byte register).
87 /*r*	XCHG *r/m16, r16*	Exchange *r16* with word from *r/m16.*
87 /*r*	XCHG *r16, r/m16*	Exchange word from *r/m16* with *r16.*
87 /*r*	XCHG *r/m32, r32*	Exchange *r32* with doubleword from *r/m32.*
87 /*r*	XCHG *r32, r/m32*	Exchange doubleword from *r/m32* with *r32.*

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 7 of the *IA-32 Intel Architecture Software Developer's Manual, Volume 3*, for more information on bus locking.)

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

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.

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.

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.

XLAT/XLATB—Table Look-up Translation

Opcode	Instruction	Description
D7	XLAT *m8*	Set AL to memory byte DS:[(E)BX + unsigned AL].
D7	XLATB	Set AL to memory byte DS:[(E)BX + unsigned AL].

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.

Operation

```
IF AddressSize = 16
    THEN
        AL ← (DS:BX + ZeroExtend(AL))
    ELSE (* AddressSize = 32 *)
        AL ← (DS:EBX + 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.

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.

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.

XOR—Logical Exclusive OR

Opcode	Instruction	Description
34 *ib*	XOR AL,*imm8*	AL XOR *imm8.*
35 *iw*	XOR AX,*imm16*	AX XOR *imm16.*
35 *id*	XOR EAX,*imm32*	EAX XOR *imm32.*
80 /6 *ib*	XOR *r/m8,imm8*	*r/m8* XOR *imm8.*
81 /6 *iw*	XOR *r/m16,imm16*	*r/m16* XOR *imm16.*
81 /6 *id*	XOR *r/m32,imm32*	*r/m32* XOR *imm32.*
83 /6 *ib*	XOR *r/m16,imm8*	*r/m16* XOR *imm8 (sign-extended).*
83 /6 *ib*	XOR *r/m32,imm8*	*r/m32* XOR *imm8 (sign-extended).*
30 */r*	XOR *r/m8,r8*	*r/m8* XOR *r8.*
31 */r*	XOR *r/m16,r16*	*r/m16* XOR *r16.*
31 */r*	XOR *r/m32,r32*	*r/m32* XOR *r32.*
32 */r*	XOR *r8,r/m8*	*r8* XOR *r/m8.*
33 */r*	XOR *r16,r/m16*	*r16 XOR r/m16.*
33 */r*	XOR *r32,r/m32*	*r32 XOR r/m32.*

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.

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.

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.

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.

XORPD—Bitwise Logical XOR for Double-Precision Floating-Point Values

Opcode	Instruction	Description
66 0F 57 /r	XORPD *xmm1*, *xmm2/m128*	Bitwise exclusive-OR of *xmm2/m128* and *xmm1*.

Description

Performs a bitwise logical exclusive-OR of the two packed double-precision floating-point values from the source operand (second operand) and the destination operand (first operand), and stores the result in the destination operand. The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register.

Operation

```
DEST[127-0] ← DEST[127-0] BitwiseXOR SRC[127-0];
```

Intel C/C++ Compiler Intrinsic Equivalent

```
XORPD           __m128d _mm_xor_pd(__m128d a, __m128d b)
```

SIMD Floating-Point Exceptions

None.

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
	If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS in CR0 is set.
#UD	If EM in CR0 is set.
	If OSFXSR in CR4 is 0.
	If CPUID feature flag SSE2 is 0.

Real-Address Mode Exceptions

#GP(0)	If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If TS in CR0 is set.
#UD	If EM in CR0 is set.
	If OSFXSR in CR4 is 0.
	If CPUID feature flag SSE2 is 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.

XORPS—Bitwise Logical XOR for Single-Precision Floating-Point Values

Opcode	Instruction	Description
0F 57 /r	XORPS *xmm1, xmm2/m128*	Bitwise exclusive-OR of *xmm2/m128* and *xmm1*.

Description

Performs a bitwise logical exclusive-OR of the four packed single-precision floating-point values from the source operand (second operand) and the destination operand (first operand), and stores the result in the destination operand. The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register.

Operation

```
DEST[127-0] ← DEST[127-0] BitwiseXOR SRC[127-0];
```

Intel C/C++ Compiler Intrinsic Equivalent

```
XORPS          __m128 _mm_xor_ps(__m128 a, __m128 b)
```

SIMD Floating-Point Exceptions

None.

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
	If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS in CR0 is set.
#UD	If EM in CR0 is set.
	If OSFXSR in CR4 is 0.
	If CPUID feature flag SSE is 0.

Real-Address Mode Exceptions

#GP(0)	If a memory operand is not aligned on a 16-byte boundary, regardless of segment.

	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If TS in CR0 is set.
#UD	If EM in CR0 is set.
	If OSFXSR in CR4 is 0.
	If CPUID feature flag SSE is 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.

A

Opcode Map

APPENDIX A
OPCODE MAP

Opcode tables in this appendix are provided to aid in interpreting IA-32 object code. Instructions are divided into three encoding groups: 1-byte opcode encoding, 2-byte opcode encoding, and escape (floating-point) encoding.

One and 2-byte opcode encoding is used to encode integer, system, MMX technology, and SSE/SSE2/SSE3 instructions. The opcode maps for these instructions are given in Table A-2 and Table A-3. Section A.3.1., "One-Byte Opcode Instructions" through Section A.3.4., "Opcode Extensions For One- And Two-byte Opcodes" give instructions for interpreting 1- and 2-byte opcode maps.

Escape encoding is used to encode floating-point instructions. The opcode maps for these instructions are in Table A-5 through Table A-20. Section A.3.5., "Escape Opcode Instructions" provides instructions for interpreting the escape opcode maps.

A.1. NOTES ON USING OPCODE TABLES

Tables in this appendix define a primary opcode (including instruction prefix where appropriate) and the ModR/M byte. Blank cells in the tables indicate opcodes that are reserved or undefined. Use the four high-order bits of the primary opcode as an index to a row of the opcode table; use the four low-order bits as an index to a column of the table. If the first byte of the primary opcode is 0FH, or 0FH is preceded by either 66H, F2H, F3H; refer to the 2-byte opcode table and use the second byte of the opcode to index the rows and columns of that table.

When the ModR/M byte includes opcode extensions, this indicates that the instructions are an instruction group in Table A-2, Table A-3. More information about opcode extensions in the ModR/M byte are covered in Table A-4.

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

Refer to Chapter 2 in *IA-32 Intel Architecture Software Developer's Manual, Volume 2A* for more information on the ModR/M byte, register values, and addressing forms.

A.2. KEY TO ABBREVIATIONS

Operands are identified by a two-character code of the form Zz. The first character (Z) specifies the addressing method; the second character (z) specifies the type of operand.

A.2.1. Codes for Addressing Method

The following abbreviations are used for 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)).
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, or a displacement.
F	EFLAGS register.
G	The reg field of the ModR/M byte selects a general register (for example, AX (000)).
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).
M	The ModR/M byte may refer only to memory: mod != 11B (BOUND, LEA, LES, LDS, LSS, LFS, LGS, CMPXCHG8B, LDDQU).
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 mod field of the ModR/M byte may refer only to a general register (for example, MOV (0F20-0F24, 0F26)).
S	The reg field of the ModR/M byte selects a segment register (for example, MOV (8C,8E)).
T	The reg field of the ModR/M byte selects a test register (for example, MOV (0F24,0F26)).
V	The reg field of the ModR/M byte selects a 128-bit XMM register.
W	A ModR/M byte follows the opcode and specifies the operand. The operand is either a 128-bit XMM 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
X	Memory addressed by the DS:SI register pair (for example, MOVS, CMPS, OUTS, or LODS).
Y	Memory addressed by the ES:DI register pair (for example, MOVS, CMPS, INS, STOS, or SCAS).

A.2.2. Codes for Operand Type

The following abbreviations are used for 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 or 48-bit pointer, depending on operand-size attribute.
pi	Quadword MMX technology register (for example, mm0)
pd	128-bit packed double-precision floating-point data
ps	128-bit packed single-precision floating-point data.
q	Quadword, regardless of operand-size attribute.
s	6-byte pseudo-descriptor.
sd	Scalar element of a 128-bit packed double-precision floating data.
ss	Scalar element of a 128-bit packed single-precision floating data.
si	Doubleword integer register (e.g., eax)
v	Word or doubleword, depending on operand-size attribute.
w	Word, regardless of operand-size attribute.

A.2.3. Register Codes

When an operand is a specific register encoded in the opcode, the register is identified by its name (for example, AX, CL, or ESI). The name of the register indicates whether the register is 32, 16, or 8 bits wide. A register identifier of the form eXX is used when the width of the register depends on the operand-size attribute. 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.

A.3. OPCODE LOOK-UP EXAMPLES

This section provides several examples to demonstrate how the following opcode maps are used.

A.3.1. One-Byte Opcode Instructions

The opcode maps for 1-byte opcodes are shown in Table A-2. Looking at the 1-byte opcode maps, the instruction mnemonic and its operands can be determined from the hexadecimal value of the 1-byte opcode. 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 mnemonic and operand types using the notations listed in Section A.2.2.
- An opcode used as an instruction prefix

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

- ModR/M byte is required and is interpreted according to the abbreviations listed in Section A.2. and Chapter 2 in *IA-32 Intel Architecture Software Developer's Manual, Volume 2A*. The operand types are listed according to the notations listed in Section A.2.2.
- ModR/M byte is required and includes an opcode extension in the reg field within the ModR/M byte. Use Table A-4 when interpreting the ModR/M byte.
- The use of the ModR/M byte is reserved or undefined. This applies to entries that represents an instruction prefix or an entry for instruction without operands related to ModR/M (for example: 60H, PUSHA; 06H, PUSH ES).

For example to look up the opcode sequence below:

Opcode: 030500000000H

LSB address					MSB address
03	05	00	00	00	00

Opcode 030500000000H for an ADD instruction can be interpreted from the 1-byte opcode map as follows. The first digit (0) of the opcode indicates the row, and the second digit (3) indicates the column in the opcode map tables. 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 that 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, which indicates that a 32-bit displacement follows (00000000H). The reg/opcode portion of the ModR/M byte (bits 3 through 5) is 000, indicating the EAX register. Thus, it can be determined that 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. These group numbers indicate that the instruction uses the reg/opcode bits in the ModR/M byte as an opcode extension (refer to Section A.3.4., "Opcode Extensions For One- And Two-byte Opcodes").

A.3.2. 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 byte is used as indices to 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), the escape opcode, the upper and lower four bits of the third byte is used as indices to a particular row and column in Table A-3. The two-byte escape sequence consists of a mandatory prefix (either 66H, F2H, or F3H), followed by the escape prefix byte 0FH.

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

- ModR/M byte is required and is interpreted according to the abbreviations listed in Section A.2. and Chapter 2 in *IA-32 Intel Architecture Software Developer's Manual, Volume 2A* for more information on the ModR/M byte, register values, and the various addressing forms. The operand types are listed according to the notations listed in Section A.2.2.
- ModR/M byte is required and includes an opcode extension in the reg field within the ModR/M byte. Use Table A-4 when interpreting the ModR/M byte.
- The use of the ModR/M byte is reserved or undefined. This applies to entries that represents an instruction without operands encoded via ModR/M (e.g. 0F77H, EMMS).

For example, the opcode 0FA4050000000003H is located on the two-byte opcode map in row A, column 4. This opcode indicates a SHLD instruction with the operands Ev, Gv, and Ib. These operands are defined 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 indicate that a 32-bit displacement follows, located in the EAX register, and is the source.

The next part of the opcode is the 32-bit displacement for the destination memory operand (00000000H), and finally the immediate byte representing the count of the shift (03H).

By this breakdown, it has been shown that this opcode represents the instruction:

```
SHLD DS:00000000H, EAX, 3
```

The next part of the SHLD opcode is the 32-bit displacement for the destination memory operand (00000000H), which is followed by the immediate byte representing the count of the shift (03H). By this breakdown, it has been shown that the opcode 0FA4050000000003H represents the instruction:

SHLD DS:00000000H, EAX, 3.

Lower case is used in the following tables to highlight the mnemonics added by MMX technology, SSE, and SSE2 instructions.

A.3.3. Opcode Map Notes

Table A-1 contains notes on particular encodings in the opcode map tables. These notes are indicated in the following Opcode Maps (Tables A-2 and A-3) by superscripts.

For the One-byte Opcode Maps (Table A-2) shading indicates instruction groupings.

Table A-1. Notes on Instruction Encoding in Opcode Map Tables

Symbol	Note
1A	Bits 5, 4, and 3 of ModR/M byte used as an opcode extension (refer to Section A.3.4., “Opcode Extensions For One- And Two-byte Opcodes”).
1B	Use the 0F0B opcode (UD2 instruction) or the 0FB9H opcode when deliberately trying to generate an invalid opcode exception (#UD).
1C	Some instructions added in the Pentium III processor may 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 completely decode the instruction, see Table A-4. (These instructions include SFENCE, STMXCSR, LDMXCSR, FXRSTOR, and FXSAVE, as well as PREFETCH and its variations.)
1D	The instruction represented by this opcode expression does not have a ModR/M byte following the primary opcode.
1E	Valid encoding for the r/m field of the ModR/M byte is shown in parenthesis.
1F	The instruction represented by this opcode expression does not support both source and destination operands to be registers.
1G	When the source operand is a register, it must be an XMM register.
1H	The instruction represented by this opcode expression does not support any operand to be a memory location.
1J	The instruction represented by this opcode expression does not support register operand.
1K	Valid encoding for the reg/opcode field of the ModR/M byte is shown in parenthesis.

Table A-2. One-byte Opcode Map† ††

	0	1	2	3	4	5	6	7
0	ADD						PUSH ES^{1D}	POP ES^{1D}
	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib^{1D}	eAX, Iv^{1D}		
1	ADC						PUSH SS^{1D}	POP SS^{1D}
	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib^{1D}	eAX, Iv^{1D}		
2	AND						SEG=ES Prefix	DAA^{1D}
	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib^{1D}	eAX, Iv^{1D}		
3	XOR						SEG=SS Prefix	AAA^{1D}
	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib^{1D}	eAX, Iv^{1D}		
4	INC general register							
	eAX^{1D}	eCX^{1D}	eDX^{1D}	eBX^{1D}	eSP^{1D}	eBP^{1D}	eSI^{1D}	eDI^{1D}
5	PUSH general register^{1D}							
	eAX	eCX	eDX	eBX	eSP	eBP	eSI	eDI
6	PUSHA/ PUSHAD^{1D}	POPA/ POPAD^{1D}	BOUND Gv, Ma	ARPL Ew, Gw	SEG=FS Prefix	SEG=GS Prefix	Opd Size Prefix	Addr Size Prefix
7	Jcc, Jb - Short-displacement jump on condition							
	O^{1D}	NO^{1D}	B/NAE/C^{1D}	NB/AE/NC^{1D}	Z/E^{1D}	NZ/NE^{1D}	BE/NA^{1D}	NBE/A^{1D}
8	Immediate Grp 1^{1A}				TEST		XCHG	
	Eb, Ib	Ev, Iv	Eb, Ib	Ev, Ib	Eb, Gb	Ev, Gv	Eb, Gb	Ev, Gv
9	NOP^{1D}	XCHG word or double-word register with eAX^{1D}						
		eCX	eDX	eBX	eSP	eBP	eSI	eDI
A	MOV^{1D}				MOVS/ MOVSB Yb, Xb^{1D}	MOVS/ MOVSW/ MOVSD Yv, Xv^{1D}	CMPS/ CMPSB Yb, Xb^{1D}	CMPS/ CMPSW/ CMPSD Xv, Yv^{1D}
	AL, Ob	eAX, Ov	Ob, AL	Ov, eAX				
B	MOV immediate byte into byte register^{1D}							
	AL	CL	DL	BL	AH	CH	DH	BH
C	Shift Grp 2^{1A}		RET Iw^{1D}	RET^{1D}	LES Gv, Mp	LDS Gv, Mp	Grp 11^{1A} - MOV	
	Eb, Ib	Ev, Ib					Eb, Ib	Ev, Iv
D	Shift Grp 2^{1A}				AAM Ib^{1D}	AAD Ib^{1D}		XLAT/ XLATB^{1D}
	Eb, 1	Ev, 1	Eb, CL	Ev, CL				
E	LOOPNE/ LOOPNZ Jb^{1D}	LOOPE/ LOOPZ Jb^{1D}	LOOP Jb^{1D}	JCXZ/ JECXZ Jb^{1D}	IN		OUT	
					AL, Ib^{1D}	eAX, Ib^{1D}	Ib, AL^{1D}	Ib, eAX^{1D}
F	LOCK Prefix		REPNE Prefix	REP/ REPE Prefix	HLT^{1D}	CMC^{1D}	Unary Grp 3^{1A}	
							Eb	Ev

NOTES:

† All blanks in the opcode map shown in Table A-2 are reserved and should not be used. Do not depend on the operation of these undefined or reserved opcodes.

†† To use the table, take the opcode's first Hex character from the row designation and the second character from the column designation. For example: 07H for [POP ES].

Table A-2. One-byte Opcode Map (Continued)

	8	9	A	B	C	D	E	F
0	OR						PUSH CS^{1D}	Escape opcode to 2-byte
	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib^{1D}	eAX, Iv^{1D}		
1	SBB						PUSH DS^{1D}	POP DS^{1D}
	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib^{1D}	eAX, Iv^{1D}		
2	SUB						SEG=CS Prefix	DAS^{1D}
	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib^{1D}	eAX, Iv^{1D}		
3	CMP						SEG=DS Prefix	AAS^{1D}
	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib^{1D}	eAX, Iv^{1D}		
4	DEC general register							
	eAX^{1D}	eCX^{1D}	eDX^{1D}	eBX^{1D}	eSP^{1D}	eBP^{1D}	eSI^{1D}	eDI^{1D}
5	POP into general register^{1D}							
	eAX	eCX	eDX	eBX	eSP	eBP	eSI	eDI
6	PUSH Iv^{1D}	IMUL Gv, Ev, Iv	PUSH Ib^{1D}	IMUL Gv, Ev, Ib	INS/ INSB Yb, DX^{1D}	INS/ INSW/ INSD Yv, DX^{1D}	OUTS/ OUTSB DX, Xb^{1D}	OUTS/ OUTSW/ OUTSD DX, Xv^{1D}
7	Jcc, Jb- Short displacement jump on condition							
	S^{1D}	NS^{1D}	P/PE^{1D}	NP/PO^{1D}	L/NGE^{1D}	NL/GE^{1D}	LE/NG^{1D}	NLE/G^{1D}
8	MOV				MOV Ew, Sw	LEA Gv, M	MOV Sw, Ew	POP Ev
	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev				
9	CBW/ CWDE^{1D}	CWD/ CDQ^{1D}	CALLF Ap^{1D}	FWAIT/ WAIT^{1D}	PUSHF/ PUSHFD Fv^{1D}	POPF/ POPFD Fv^{1D}	SAHF^{1D}	LAHF^{1D}
A	TEST^{1D}		STOS/ STOSB Yb, AL ^{1D}	STOS/ STOSW/ STOSD Yv, eAX^{1D}	LODS/ LODSB AL, Xb^{1D}	LODS/ LODSW/ LODSD eAX, Xv^{1D}	SCAS/ SCASB AL, Yb^{1D}	SCAS/ SCASW/ SCASD eAX, Yv^{1D}
	AL, Ib	eAX, Iv						
B	MOV immediate word or double into word or double register^{1D}							
	eAX	eCX	eDX	eBX	eSP	eBP	eSI	eDI
C	ENTER Iw, Ib^{1D}	LEAVE^{1D}	RETF Iw^{1D}	RETF^{1D}	INT 3^{1D}	INT Ib^{1D}	INTO^{1D}	IRET^{1D}
D	ESC (Escape to coprocessor instruction set)							
E	CALL	JMP			IN		OUT	
	Jv^{1D}	near Jv^{1D}	far Ap^{1D}	short Jb^{1D}	AL, DX^{1D}	eAX, DX^{1D}	DX, AL^{1D}	DX, eAX^{1D}
F	CLC^{1D}	STC^{1D}	CLI^{1D}	STI^{1D}	CLD^{1D}	STD^{1D}	INC/DEC Grp 4^{1A}	INC/DEC Grp 5^{1A}

Table A-3. Two-byte Opcode Map (First Byte is 0FH)†††

	0	1	2	3	4	5	6	7
0	Grp 6[1A]	Grp 7[1A]	LAR Gv, Ew	LSL Gv, Ew			CLTS[1D]	
1	MOVUPS Vps, Wps MOVSS (F3) Vss, Wss MOVUPD (66) Vpd, Wpd MOVSD (F2) Vsd, Wsd	MOVUPS Wps, Vps MOVSS (F3) Wss, Vss MOVUPD (66) Wpd, Vpd MOVSD (F2) Wsd, Vsd	MOVLPS Vq, Mq[1F] MOVLPD (66) Vq, Mq[1F] MOVHLPS Vps, Vps MOVDDUP (F2) Vq, Wq[1G] MOVSLDUP (F3) Vps, Wps	MOVLPS Mq, Vq[1F] MOVLPD (66) Mq, Vq[1F]	UNPCKLPS Vps, Wps UNPCKLPD (66) Vpd, Wpd	UNPCKHPS Vps, Wps UNPCKHPD (66) Vpd, Wpd	MOVHPS Vq, Mq[1F] MOVHPD (66) Vq, Mq[1F] MOVLHPS Vps, Vps MOVSHDUP (F3) Vps, Wps	MOVHPS Mq, Vps[1F] MOVHPD (66) Mq, Vpd[1F]
2	MOV Rd, Cd[1H]	MOV Rd, Dd[1H]	MOV Cd, Rd[1H]	MOV Dd, Rd[1H]	MOV Rd, Td†††		MOV Td, Rd†††	
3	WRMSR[1D]	RDTSC[1D]	RDMSR[1D]	RDPMC[1D]	SYSENTER[1D]	SYSEXIT[1D]		
4	CMOVcc, (Gv, Ev) - Conditional Move							
	O	NO	B/C/NAE	AE/NB/NC	E/Z	NE/NZ	BE/NA	A/NBE
5	MOVMSKPS Gd, Vps[1H] MOVMSKPD (66) Gd, Vpd[1H]	SQRTPS Vps, Wps SQRTSS (F3) Vss, Wss SQRTPD (66) Vpd, Wpd SQRTSD (F2) Vsd, Wsd	RSQRTPS Vps, Wps RSQRTSS (F3) Vss, Wss	RCPPS Vps, Wps RCPSS (F3) Vss, Wss	ANDPS Vps, Wps ANDPD (66) Vpd, Wpd	ANDNPS Vps, Wps ANDNPD (66) Vpd, Wpd	ORPS Vps, Wps ORPD (66) Vpd, Wpd	XORPS Vps, Wps XORPD (66) Vpd, Wpd
6	PUNPCKLBW Pq, Qd PUNPCKLBW (66) Vdq, Wdq	PUNPCKLWD Pq, Qd PUNPCKLWD (66) Vdq, Wdq	PUNPCKLDQ Pq, Qd PUNPCKLDQ (66) Vdq, Wdq	PACKSSWB Pq, Qq PACKSSWB (66) Vdq, Wdq	PCMPGTB Pq, Qq PCMPGTB (66) Vdq, Wdq	PCMPGTW Pq, Qq PCMPGTW (66) Vdq, Wdq	PCMPGTD Pq, Qq PCMPGTD (66) Vdq, Wdq	PACKUSWB Pq, Qq PACKUSWB (66) Vdq, Wdq
7	PSHUFW Pq, Qq, Ib PSHUFD (66) Vdq, Wdq, Ib PSHUFHW (F3) Vdq, Wdq, Ib PSHUFLW (F2) Vdq, Wdq, Ib	(Grp 12[1A])	(Grp 13[1A])	(Grp 14[1A])	PCMPEQB Pq, Qq PCMPEQB (66) Vdq, Wdq	PCMPEQW Pq, Qq PCMPEQW (66) Vdq, Wdq	PCMPEQD Pq, Qq PCMPEQD (66) Vdq, Wdq	EMMS[1D]

NOTES:

† All blanks in the opcode map shown in Table A-3 are reserved and should not be used. Do not depend on the operation of these undefined or reserved opcodes.

†† To use the table, use 0FH for the first byte of the opcode. For the second byte, take the first Hex character from the row designation and the second character from the column designation. For example: 0F03H for [LSL GV, EW].

††† Not currently supported after Pentium Pro and Pentium II families. Using this opcode on the current generation of processors will generate a #UD. For future processors, this value is reserved.

Table A-3. Two-byte Opcode Map (Proceeding Byte is 0FH)

	8	9	A	B	C	D	E	F
0	INVD[1D]	WBINVD[1D]		UD2				
1	PREFETCH[1C] (Grp 16[1A])							
2	MOVAPS Vps, Wps MOVAPD (66) Vpd, Wpd	MOVAPS Wps, Vps MOVAPD (66) Wpd, Vpd	CVTPI2PS Vps, Qq CVTSI2SS (F3) Vss, Ed CVTPI2PD (66) Vpd, Qq CVTSI2SD (F2) Vsd, Ed	MOVNTPS Mps, Vps[1F] MOVNTPD (66) Mpd, Vpd[1F]	CVTTPS2PI Pq, Wq CVTTSS2SI (F3) Gd, Wss CVTTPD2PI (66) Pq, Wpd CVTTSD2SI (F2) Gd, Wsd	CVTPS2PI Pq, Wq CVTSS2SI (F3) Gd, Wss CVTPD2PI (66) Pq, Wpd CVTSD2SI (F2) Gd, Wsd	UCOMISS Vss, Wss UCOMISD (66) Vsd, Wsd	COMISS Vps, Wps COMISD (66) Vsd, Wsd
3								
4	CMOVcc(Gv, Ev) - Conditional Move							
	S	NS	P/PE	NP/PO	L/NGE	NL/GE	LE/NG	NLE/G
5	ADDPS Vps, Wps ADDSS (F3) Vss, Wss ADDPD (66) Vpd, Wpd ADDSD (F2) Vsd, Wsd	MULPS Vps, Wps MULSS (F3) Vss, Wss MULPD (66) Vpd, Wpd MULSD (F2) Vsd, Wsd	CVTPS2PD Vpd, Wq CVTSS2SD (F3) Vsd, Wss CVTPD2PS (66) Vps, Wpd CVTSD2SS (F2) Vss, Wsd	CVTDQ2PS Vps, Wdq CVTPS2DQ (66) Vdq, Wps CVTTPS2DQ (F3) Vdq, Wps	SUBPS Vps, Wps SUBSS (F3) Vss, Wss SUBPD (66) Vpd, Wpd SUBSD (F2) Vsd, Wsd	MINPS Vps, Wps MINSS (F3) Vss, Wss MINPD (66) Vpd, Wpd MINSD (F2) Vsd, Wsd	DIVPS Vps, Wps DIVSS (F3) Vss, Wss DIVPD (66) Vpd, Wpd DIVSD (F2) Vsd, Wsd	MAXPS Vps, Wps MAXSS (F3) Vss, Wss MAXPD (66) Vpd, Wpd MAXSD (F2) Vsd, Wsd
6	PUNPCKHBW Pq, Qq PUNPCKHBW (66) Vdq, Qdq	PUNPCKHWD Pq, Qq PUNPCKHWD (66) Vdq, Qdq	PUNPCKHDQ Pq, Qq PUNPCKHDQ (66) Vdq, Qdq	PACKSSDW Pq, Qq PACKSSDW (66) Vdq, Qdq	PUNPCKLQDQ (66) Vdq, Wdq	PUNPCKHQD Q (66) Vdq, Wdq	MOVD Pd, Ed MOVD (66) Vd, Ed	MOVQ Pq, Qq MOVDQA (66) Vdq, Wdq MOVDQU (F3) Vdq, Wdq
7	MMX UD (Reserved for future use)				HADDPD (66) Vpd, Wpd HADDPS (F2) Vps, Wps	HSUBPD (66) Vpd, Wpd HSUBPS (F2) Vps, Wps	MOVD Ed, Pd MOVD (66) Ed, Vd MOVQ (F3) Vq, Wq	MOVQ Qq, Pq MOVDQA (66) Wdq, Vdq MOVDQU (F3) Wdq, Vdq

Table A-3. Two-byte Opcode Map (Proceeding Byte is 0FH)

	0	1	2	3	4	5	6	7
8	Jcc, Jv - Long-displacement jump on condition							
	O[1D]	NO[1D]	B/C/NAE[1D]	AE/NB/NC[1D]	E/Z[1D]	NE/NZ[1D]	BE/NA[1D]	A/NBE[1D]
9	SETcc, Eb - Byte Set on condition (000)[1K]							
	O	NO	B/C/NAE	AE/NB/NC	E/Z	NE/NZ	BE/NA	A/NBE
A	PUSH FS[1D]	POP FS[1D]	CPUID[1D]	BT Ev, Gv	SHLD Ev, Gv, Ib	SHLD Ev, Gv, CL		
B	CMPXCHG		LSS Mp	BTR Ev, Gv	LFS Mp	LGS Mp	MOVZX	
	Eb, Gb	Ev, Gv					Gv, Eb	Gv, Ew
C	XADD Eb, Gb	XADD Ev, Gv	CMPPS Vps, Wps, Ib CMPSS (F3) Vss, Wss, Ib CMPPD (66) Vpd, Wpd, Ib CMPSD (F2) Vsd, Wsd, Ib	MOVNTI Md, Gd[1F]	PINSRW Pw, Ew, Ib PINSRW (66) Vw, Ew, Ib	PEXTRW Gw, Pw, Ib[1H] PEXTRW (66) Gw, Vw, Ib[1H]	SHUFPS Vps, Wps, Ib SHUFPD (66) Vpd, Wpd, Ib	Grp 9[1A]
D	ADDSUBPD (66) Vpd, Wpd ADDSUBPS (F2) Vps, Wps	PSRLW Pq, Qq PSRLW (66) Vdq, Wdq	PSRLD Pq, Qq PSRLD (66) Vdq, Wdq	PSRLQ Pq, Qq PSRLQ (66) Vdq, Wdq	PADDQ Pq, Qq PADDQ (66) Vdq, Wdq	PMULLW Pq, Qq PMULLW (66) Vdq, Wdq	MOVQ (66) Wq, Vq MOVQ2DQ (F3) Vdq, Qq[1H] MOVDQ2Q (F2) Pq, Vq[1H]	PMOVMSKB Gd, Pq[1H] PMOVMSKB (66) Gd, Vdq[1H]
E	PAVGB Pq, Qq PAVGB (66) Vdq, Wdq	PSRAW Pq, Qq PSRAW (66) Vdq, Wdq	PSRAD Pq, Qq PSRAD (66) Vdq, Wdq	PAVGW Pq, Qq PAVGW (66) Vdq, Wdq	PMULHUW Pq, Qq PMULHUW (66) Vdq, Wdq	PMULHW Pq, Qq PMULHW (66) Vdq, Wdq	CVTPD2DQ (F2) Vdq, Wpd CVTTPD2DQ (66) Vdq, Wpd CVTDQ2PD (F3) Vpd, Wq	MOVNTQ Mq, Vq[1F] MOVNTDQ (66) Mdq, Vdq[1F]
F	LDDQU (F2) Vdq, Mdq	PSLLW Pq, Qq PSLLW (66) Vdq, Wdq	PSLLD Pq, Qq PSLLD (66) Vdq, Wdq	PSLLQ Pq, Qq PSLLQ (66) Vdq, Wdq	PMULUDQ Pq, Qq PMULUDQ (66) Vdq, Wdq	PMADDWD Pq, Qq PMADDWD (66) Vdq, Wdq	PSADBW Pq, Qq PSADBW (66) Vdq, Wdq	MASKMOVQ Pq, Pq[1H] MASKMOV- DQU (66) Vdq, Vdq[1H]

Table A-3. Two-byte Opcode Map (Proceeding Byte is 0FH)

	8	9	A	B	C	D	E	F
8	Jcc, Jv - Long-displacement jump on condition							
	S[1D]	NS[1D]	P/PE[1D]	NP/PO[1D]	L/NGE[1D]	NL/GE[1D]	LE/NG[1D]	NLE/G[1D]
9	SETcc, Eb - Byte Set on condition (000)[1K]							
	S	NS	P/PE	NP/PO	L/NGE	NL/GE	LE/NG	NLE/G
A	PUSH GS[1D]	POP GS[1D]	RSM[1D]	BTS Ev, Gv	SHRD Ev, Gv, Ib	SHRD Ev, Gv, CL	(Grp 15[1A])[1C]	IMUL Gv, Ev
B		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
C	BSWAP[1D]							
	EAX	ECX	EDX	EBX	ESP	EBP	ESI	EDI
D	PSUBUSB Pq, Qq PSUBUSB (66) Vdq, Wdq	PSUBUSW Pq, Qq PSUBUSW (66) Vdq, Wdq	PMINUB Pq, Qq PMINUB (66) Vdq, Wdq	PAND Pq, Qq PAND (66) Vdq, Wdq	PADDUSB Pq, Qq PADDUSB (66) Vdq, Wdq	PADDUSW Pq, Qq PADDUSW (66) Vdq, Wdq	PMAXUB Pq, Qq PMAXUB (66) Vdq, Wdq	PANDN Pq, Qq PANDN (66) Vdq, Wdq
E	PSUBSB Pq, Qq PSUBSB (66) Vdq, Wdq	PSUBSW Pq, Qq PSUBSW (66) Vdq, Wdq	PMINSW Pq, Qq PMINSW (66) Vdq, Wdq	POR Pq, Qq POR (66) Vdq, Wdq	PADDSB Pq, Qq PADDSB (66) Vdq, Wdq	PADDSW Pq, Qq PADDSW (66) Vdq, Wdq	PMAXSW Pq, Qq PMAXSW (66) Vdq, Wdq	PXOR Pq, Qq PXOR (66) Vdq, Wdq
F	PSUBB Pq, Qq PSUBB (66) Vdq, Wdq	PSUBW Pq, Qq PSUBW (66) Vdq, Wdq	PSUBD Pq, Qq PSUBD (66) Vdq, Wdq	PSUBQ Pq, Qq PSUBQ (66) Vdq, Wdq	PADDB Pq, Qq PADDB (66) Vdq, Wdq	PADDW Pq, Qq PADDW (66) Vdq, Wdq	PADDD Pq, Qq PADDD (66) Vdq, Wdq	

A.3.4. Opcode Extensions For One- And Two-byte Opcodes

Some of the 1-byte and 2-byte opcodes use bits 5, 4, and 3 of the ModR/M byte (the nnn field in Figure A-1) as an extension of the opcode. The value of bits 5, 4, and 3 of the ModR/M byte also corresponds to “/digit” portion of the opcode notation described in Chapter 3. Those opcodes that have opcode extensions are indicated in Table A-4 with group numbers (Group 1, Group 2, etc.). The group numbers (ranging from 1 to 16) in the second column provide an entry point into Table A-4 where the encoding of the opcode extension field can be found. The valid encoding the r/m field of the ModR/M byte for each instruction can be inferred from the third column.

For example, the ADD instruction with a 1-byte opcode of 80H is a Group 1 instruction. Table A-4 indicates that the opcode extension field that must be encoded in the ModR/M byte for this instruction is 000B. The r/m field for this instruction can be encoded to access a register (11B); or a memory address using addressing modes (for example: mem = 00B, 01B, 10B).

mod	nnn	R/M

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

Table A-4. Opcode Extensions for One- and Two-byte Opcodes by Group Number

Opcode	Group	Mod 7,6	Encoding of Bits 5,4,3 of the ModR/M Byte							
			000	001	010	011	100	101	110	111
80-83	1	mem, 11B	ADD	OR	ADC	SBB	AND	SUB	XOR	CMP
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/Iv		NOT	NEG	MUL AL/eAX	IMUL AL/eAX	DIV AL/eAX	IDIV AL/eAX
FE	4	mem, 11B	INC Eb	DEC Eb						
FF	5	mem, 11B	INC Ev	DEC Ev	CALLN Ev	CALLF Ep 1J	JMPN Ev	JMPF Ep 1J	PUSH Ev	
OF OO	6	mem, 11B	SLDT Ew	STR Ev	LLDT Ew	LTR Ew	VERR Ew	VERW Ew		
OF 01	7	mem	SGDT Ms	SIDT Ms	LGDT Ms	LIDT Ms	SMSW Ew		LMSW Ew	INVLPG Mb
		11B		MONITOR eAX, eCX, eDX (000)1E MWAIT eAX, eCX (001)1E						
OF BA	8	mem, 11B					BT	BTS	BTR	BTC
OF C7	9	mem		CMPXCH8B Mq						
		11B								

Table A-4. Opcode Extensions for One- and Two-byte Opcodes by Group Number (Contd.)

OF B9	10	mem								
		11B								
C6	11	mem, 11B	MOV Eb, Ib							
C7		mem, 11B	MOV Ev, Iv							
OF 71	12	mem								
		11B			PSRLW Pq, Ib PSRLW (66) Pdq, Ib		PSRAW Pq, Ib PSRAW (66) Pdq, Ib		PSLLW Pq, Ib PSLLW (66) Pdq, Ib	
OF 72	13	mem								
		11B			PSRLD Pq, Ib PSRLD (66) Wdq, Ib		PSRAD Pq, Ib PSRAD (66) Wdq, Ib		PSLLD Pq, Ib PSLLD (66) Wdq, Ib	
OF 73	14	mem								
		11B			PSRLQ Pq, Ib PSRLQ (66) Wdq, Ib	PSRLDQ (66) Wdq, Ib			PSLLQ Pq, Ib PSLLQ (66) Wdq, Ib	PSLLDQ (66) Wdq, Ib
OF AE	15	mem	FXSAVE	FXRSTOR	LDMXCSR	STMXCSR				CLFLUSH
		11B						LFENCE (000) 1E	MFENCE (000) 1E	SFENCE (000) 1E
OF 18	16	mem	PREFETCH- NTA	PREFETCH- T0	PREFETCH- T1	PREFETCH- T2				
		11B								

NOTE:

All blanks in the opcode map are reserved and should not be used. Do not depend on the operation of these undefined or reserved opcodes.

A.3.5. Escape Opcode Instructions

The opcode maps for the coprocessor escape instruction opcodes (x87 floating-point instruction opcodes) are given in Table A-5 through Table A-20. These opcode maps are grouped by the first byte of the opcode from D8 through DF. Each of these opcodes has a ModR/M byte. If the ModR/M byte is within the range of 00H through BFH, bits 5, 4, and 3 of the ModR/M byte are used as an opcode extension, similar to the technique used for 1-and 2-byte opcodes (refer to Section A.3.4., "Opcode Extensions For One- And Two-byte Opcodes"). 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.3.5.1. OPCODES WITH MODR/M BYTES IN THE 00H THROUGH BFH RANGE

The opcode DD0504000000H can be interpreted as follows. The instruction encoded with this opcode can be located in Section A.3.5.8., "Escape Opcodes with DD as First Byte". Since the ModR/M byte (05H) is within the 00H through BFH range, bits 3 through 5 (000) of this byte indicate the opcode to be for an FLD double-real instruction (refer to Table A-7). The double-real value to be loaded is at 00000004H, which is the 32-bit displacement that follows and belongs to this opcode.

A.3.5.2. OPCODES WITH MODR/M BYTES OUTSIDE THE 00H THROUGH BFH RANGE

The opcode D8C1H illustrates an opcode with a ModR/M byte outside the range of 00H through BFH. The instruction encoded here, can be located in Section A.3.4., "Opcode Extensions For One- And Two-byte Opcodes". In Table A-6, the ModR/M byte C1H indicates row C, column 1, which is an FADD instruction using ST(0), ST(1) as the operands.

A.3.5.3. ESCAPE OPCODES WITH D8 AS FIRST BYTE

Table A-5 and Table A-6 contain the opcode maps for the escape instruction opcodes that begin with D8H. Table A-5 shows the opcode map if the accompanying ModR/M byte within the range of 00H through BFH. Here, the value of bits 5, 4, and 3 (the nnn field in Figure A-1) selects the instruction.

Table A-5. D8 Opcode Map When ModR/M Byte is Within 00H to BFH[1]

nnn Field of ModR/M Byte (refer to Figure A-1)							
000B	001B	010B	011B	100B	101B	110B	111B

NOTE:

1. All blanks in the opcode map are reserved and should not be used. Do not depend on the operation of these undefined opcodes.

Table A-6 shows the opcode map if the accompanying ModR/M byte is outside the range of 00H to BFH. In this case the first digit of the ModR/M byte selects the row in the table and the second digit selects the column.

Table A-6. D8 Opcode Map When ModR/M Byte is Outside 00H to BFH[1]

	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)

NOTE:

1. All blanks in the opcode map are reserved and should not be used. Do not depend on the operation of these undefined opcodes.

A.3.5.4. ESCAPE OPCODES WITH D9 AS FIRST BYTE

Table A-7 and Table A-8 contain opcode maps for escape instruction opcodes that begin with D9H. Table A-7 shows the opcode map if the accompanying ModR/M byte is within the range of 00H through BFH. Here, the value of bits 5, 4, and 3 (the Figure A-1 nnn field) selects the instruction.

Table A-7. D9 Opcode Map When ModR/M Byte is Within 00H to BFH[1].

nnn Field of ModR/M Byte (refer to Figure A-1)							
000B	001B	010B	011B	100B	101B	110B	111B
FLD single-real		FST single-real	FSTP single-real	FLDENV 14/28 bytes	FLDCW 2 bytes	FNSTENV 14/28 bytes	FNSTCW 2 bytes

NOTE:

1. All blanks in the opcode map are reserved and should not be used. Do not depend on the operation of these undefined opcodes.

Table A-8 shows the opcode map if the accompanying ModR/M byte is outside the range of 00H to BFH. In this case the first digit of the ModR/M byte selects the row in the table and the second digit selects the column.

Table A-8. D9 Opcode Map When ModR/M Byte is Outside 00H to BFH[1]

	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

NOTE:

1. All blanks in the opcode map are reserved and should not be used. Do not depend on the operation of these undefined opcodes.

A.3.5.5. ESCAPE OPCODES WITH DA AS FIRST BYTE

Table A-9 and Table A-10 contain the opcode maps for the escape instruction opcodes that begin with DAH. Table A-9 shows the opcode map if the accompanying ModR/M byte within the range of 00H through BFH. Here, the value of bits 5, 4, and 3 (the nnn field in Figure A-1) selects the instruction.

Table A-9. DA Opcode Map When ModR/M Byte is Within 00H to BFH[1]

nnn Field of ModR/M Byte (refer to Figure A-1)							
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

NOTE:

1. All blanks in the opcode map are reserved and should not be used. Do not depend on the operation of these undefined opcodes.

Table A-10 shows the opcode map if the accompanying ModR/M byte is outside the range of 00H to BFH. In this case the first digit of the ModR/M byte selects the row in the table and the second digit selects the column.

Table A-10. DA Opcode Map When ModR/M Byte is Outside 00H to BFH[1]

	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								

NOTE:

1. All blanks in the opcode map are reserved and should not be used. Do not depend on the operation of these undefined opcodes.

A.3.5.6. ESCAPE OPCODES WITH DB AS FIRST BYTE

Table A-11 and Table A-12 contain the opcode maps for the escape instruction opcodes that begin with DBH. Table A-11 shows the opcode map if the accompanying ModR/M byte within the range of 00H through BFH. Here, the value of bits 5, 4, and 3 (the nnn field in Figure A-1) selects the instruction.

Table A-11. DB Opcode Map When ModR/M Byte is Within 00H to BFH[1]

nnn Field of ModR/M Byte (refer to Figure A-1)							
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

NOTE:

1. All blanks in the opcode map are reserved and should not be used. Do not depend on the operation of these undefined opcodes.

Table A-12 shows the opcode map if the accompanying ModR/M byte is outside the range of 00H to BFH. In this case the first digit of the ModR/M byte selects the row in the table and the second digit selects the column.

Table A-12. DB Opcode Map When ModR/M Byte is Outside 00H to BFH[1]

	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			FNCLEX	FNINIT				
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								

NOTE:

1. All blanks in the opcode map are reserved and should not be used. Do not depend on the operation of these undefined opcodes.

A.3.5.7. ESCAPE OPCODES WITH DC AS FIRST BYTE

Table A-13 and Table A-14 contain the opcode maps for the escape instruction opcodes that begin with DCH. Table A-13 shows the opcode map if the accompanying ModR/M byte within the range of 00H through BFH. Here, the value of bits 5, 4, and 3 (the nnn field in Figure A-1) selects the instruction.

Table A-13. DC Opcode Map When ModR/M Byte is Within 00H to BFH[1]

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

NOTE:

1. All blanks in the opcode map are reserved and should not be used. Do not depend on the operation of these undefined opcodes.

Table A-14 shows the opcode map if the accompanying ModR/M byte is outside the range of 00H to BFH. In this case the first digit of the ModR/M byte selects the row in the table and the second digit selects the column.

Table A-14. DC Opcode Map When ModR/M Byte is Outside 00H to BFH[4]

	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)

NOTE:

1. All blanks in the opcode map are reserved and should not be used. Do not depend on the operation of these undefined opcodes.

A.3.5.8. ESCAPE OPCODES WITH DD AS FIRST BYTE

Table A-15 and Table A-16 contain the opcode maps for the escape instruction opcodes that begin with DDH. Table A-15 shows the opcode map if the accompanying ModR/M byte within the range of 00H through BFH. Here, the value of bits 5, 4, and 3 (the nnn field in Figure A-1) selects the instruction.

Table A-15. DD Opcode Map When ModR/M Byte is Within 00H to BFH[1]

nnn Field of ModR/M Byte (refer to Figure A-1)							
000B	001B	010B	011B	100B	101B	110B	111B

NOTE:

1. All blanks in the opcode map are reserved and should not be used. Do not depend on the operation of these undefined opcodes.

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

Table A-16. DD Opcode Map When ModR/M Byte is Outside 00H to BFH[1]

	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								

NOTE:

1. All blanks in the opcode map are reserved and should not be used. Do not depend on the operation of these undefined opcodes.

A.3.5.9. ESCAPE OPCODES WITH DE AS FIRST BYTE

Table A-17 and Table A-18 contain the opcode maps for the escape instruction opcodes that begin with DEH. Table A-17 shows the opcode map if the accompanying ModR/M byte within the range of 00H through BFH. Here, the value of bits 5, 4, and 3 (the nnn field in Figure A-1) selects the instruction.

Table A-17. DE Opcode Map When ModR/M Byte is Within 00H to BFH[1]

nnn Field of ModR/M Byte (refer to Figure A-1)							
000B	001B	010B	011B	100B	101B	110B	111B

NOTE:

1. All blanks in the opcode map are reserved and should not be used. Do not depend on the operation of these undefined opcodes.

Table A-18 shows the opcode map if the accompanying ModR/M byte is outside the range of 00H to BFH. In this case the first digit of the ModR/M byte selects the row in the table and the second digit selects the column.

Table A-18. DE Opcode Map When ModR/M Byte is Outside 00H to BFH[1]

	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)

NOTE:

1. All blanks in the opcode map are reserved and should not be used. Do not depend on the operation of these undefined opcodes.

A.3.5.10. ESCAPE OPCODES WITH DF AS FIRST BYTE

Table A-19 and Table A-20 contain the opcode maps for the escape instruction opcodes that begin with DFH. Table A-19 shows the opcode map if the accompanying ModR/M byte within the range of 00H through BFH. Here, the value of bits 5, 4, and 3 (the nnn field in Figure A-1) selects the instruction.

Table A-19. DF Opcode Map When ModR/M Byte is Within 00H to BFH[1]

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

NOTE:

1. All blanks in the opcode map are reserved and should not be used. Do not depend on the operation of these undefined opcodes.

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

Table A-20. DF Opcode Map When ModR/M Byte is Outside 00H to BFH[1]

	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								

NOTE:

1. All blanks in the opcode map are reserved and should not be used. Do not depend on the operation of these undefined opcodes.

B

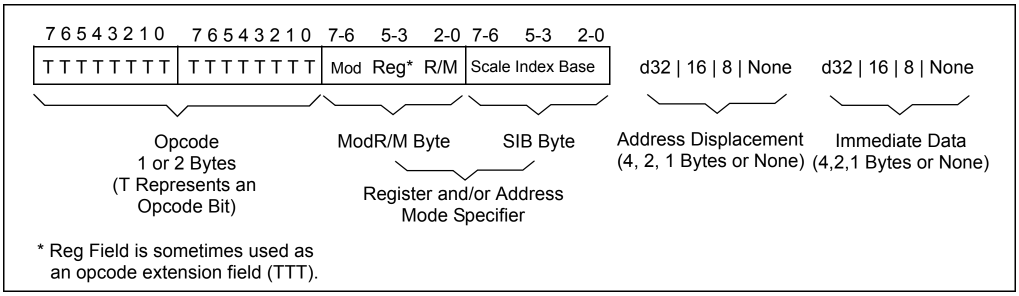
Instruction Formats and Encodings

APPENDIX B INSTRUCTION FORMATS AND ENCODINGS

This appendix shows the machine instruction formats and encodings of the IA-32 architecture instructions. The first section describes in detail the IA-32 architecture's machine instruction format. The following sections show the formats and encoding of general-purpose, MMX, P6 family, SSE/SSE2/SSE3, and x87 FPU instructions.

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 (if required) consisting of the ModR/M byte and sometimes the scale-index-base (SIB) byte, a displacement (if required), and an immediate data field (if required).



7 6 5 4 3 2 1 0
7 6 5 4 3 2 1 0
7-6 5-3 2-0
7-6 5-3 2-0
T T T T T T T T
T T T T T T T T
Mod Reg* R/M
Scale Index Base
d32 | 16 | 8 | None
d32 | 16 | 8 | None
Opcode 1 or 2 Bytes (T Represents an Opcode Bit)
ModR/M Byte
SIB Byte
Register and/or Address Mode Specifier
Address Displacement (4, 2, 1 Bytes or None)
Immediate Data (4,2,1 Bytes or None)
* Reg Field is sometimes used as an opcode extension field (TTT).

Figure B-1. General Machine Instruction Format

The primary opcode for an instruction is encoded in one or two bytes of the instruction. Some instructions also use an opcode extension field encoded in bits 5, 4, and 3 of the ModR/M byte. Within the primary opcode, smaller encoding fields may be defined. These fields vary according to the class of operation being performed. The fields define such information as register encoding, conditional test performed, or sign extension of immediate byte.

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, the reg field, and the R/M field. Certain encodings of the ModR/M byte indicate that a second address mode byte, the SIB byte, must be used.

If the selected addressing mode specifies a displacement, the displacement value is placed immediately following the ModR/M byte or SIB byte. If a displacement is present, the possible sizes are 8, 16, or 32 bits.

If the instruction specifies an immediate operand, the immediate value follows any displacement bytes. An immediate operand, if specified, is always the last field of the instruction.

Table B-1 lists several smaller fields or bits that appear in certain instructions, sometimes within the opcode bytes themselves. The following tables describe these fields and bits and list the allowable values. All of these fields (except the d bit) are shown in the general-purpose instruction formats given in Table B-11.

Table B-1. Special Fields Within Instruction Encodings

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

B.1.1. Reg Field (reg)

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 (see Table B-4). Table B-2 shows the encoding of the reg field when the w bit is not present in an encoding, and Table B-3 shows the encoding of the reg field when the w bit is present.

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

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

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

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

B.1.2. Encoding of Operand Size Bit (w)

The current operand-size attribute determines whether the processor is performing 16-or 32-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 (16 bits or 32 bits). Table B-4 shows the encoding of the w bit depending on the current operand-size attribute.

Table B-4. 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.1.3. Sign Extend (s) Bit

The sign-extend (s) bit occurs primarily in instructions with immediate data fields that are being extended from 8 bits to 16 or 32 bits. Table B-5 shows the encoding of the s bit.

Table B-5. 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. Segment Register Field (sreg)

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-6 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-6. 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*
111	Reserved*

* Do not use reserved encodings.

B.1.5. Special-Purpose Register (eee) Field

When the control or debug registers are referenced in an instruction they are encoded in the eee field, which is located in bits 5, 4, and 3 of the ModR/M byte. Table B-7 shows the encoding of the eee field.

Table B-7. Encoding of Special-Purpose Register (eee) Field

eee	Control Register	Debug Register
000	CR0	DR0
001	Reserved*	DR1
010	CR2	DR2
011	CR3	DR3
100	CR4	Reserved*
101	Reserved*	Reserved*
110	Reserved*	DR6
111	Reserved*	DR7

* Do not use reserved encodings.

B.1.6. Condition Test Field (tttn)

For conditional instructions (such as conditional jumps and set on condition), the condition test field (tttn) is encoded for the condition being tested for. 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-8 shows the encoding of the tttn field.

Table B-8. 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.1.7. Direction (d) Bit

In many two-operand instructions, a direction bit (d) indicates which operand is considered the source and which is the destination. Table B-9 shows the encoding of the d bit. When used for integer instructions, the d bit is located at bit 1 of a 1-byte primary opcode. This bit does not appear as the symbol “d” in Table B-11; instead, 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-9. 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.1.8. Other Notes

Table B-10 contains notes on particular encodings. These notes are indicated in the tables shown in the following sections by superscripts.

Table B-10. Notes on Instruction Encoding

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

B.2. GENERAL-PURPOSE INSTRUCTION FORMATS AND ENCODINGS

Table B-11 shows the machine instruction formats and encodings of the general purpose instructions.

Table B-11. General Purpose Instruction Formats and Encodings

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

Table B-11. General Purpose Instruction Formats and Encodings (Contd.)

Instruction and Format	Encoding
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
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

Table B-11. General Purpose Instruction Formats and Encodings (Contd.)

Instruction and Format	Encoding
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
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

Table B-11. General Purpose Instruction Formats and Encodings (Contd.)

Instruction and Format	Encoding
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
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	1111 011w : 11 111 reg
AL, AX, or EAX by memory	1111 011w : mod 111 r/m

Table B-11. General Purpose Instruction Formats and Encodings (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
IRET/IRETD – Interrupt Return	1100 1111
J*cc* – 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

Table B-11. General Purpose Instruction Formats and Encodings (Contd.)

Instruction and Format	Encoding
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
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 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 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

Table B-11. General Purpose Instruction Formats and Encodings (Contd.)

Instruction and Format	Encoding
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
AL, AX, or EAX to memory	1010 001w : full displacement
MOV – Move to/from Control Registers	
CR0 from register	0000 1111 : 0010 0010 : 11 000 reg
CR2 from register	0000 1111 : 0010 0010 : 11 010reg
CR3 from register	0000 1111 : 0010 0010 : 11 011 reg
CR4 from register	0000 1111 : 0010 0010 : 11 100 reg
register from CR0-CR4	0000 1111 : 0010 0000 : 11 eee reg
MOV – Move to/from Debug Registers	
DR0-DR3 from register	0000 1111 : 0010 0011 : 11 eee reg
DR4-DR5 from register	0000 1111 : 0010 0011 : 11 eee reg
DR6-DR7 from register	0000 1111 : 0010 0011 : 11 eee reg
register from DR6-DR7	0000 1111 : 0010 0001 : 11 eee reg
register from DR4-DR5	0000 1111 : 0010 0001 : 11 eee reg
register from DR0-DR3	0000 1111 : 0010 0001 : 11 eee reg
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

Table B-11. General Purpose Instruction Formats and Encodings (Contd.)

Instruction and Format	Encoding
MOVS/MOVSB/MOVSW/MOVSD – Move Data from String to String	1010 010w
MOVSX – Move with Sign-Extend	
register2 to register1	0000 1111 : 1011 111w : 11 reg1 reg2
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 reg
NEG – Two's Complement Negation	
register	1111 011w : 11 011 reg
memory	1111 011w : mod 011 r/m
NOP – No Operation	1001 0000
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

Table B-11. General Purpose Instruction Formats and Encodings (Contd.)

Instruction and Format	Encoding
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
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

Table B-11. General Purpose Instruction Formats and Encodings (Contd.)

Instruction and Format	Encoding
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
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
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

Table B-11. General Purpose Instruction Formats and Encodings (Contd.)

Instruction and Format	Encoding
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
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

Table B-11. General Purpose Instruction Formats and Encodings (Contd.)

Instruction and Format	Encoding
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-11. General Purpose Instruction Formats and Encodings (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
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

Table B-11. General Purpose Instruction Formats and Encodings (Contd.)

Instruction and Format	Encoding
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
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

B.3. PENTIUM FAMILY INSTRUCTION FORMATS AND ENCODINGS

The following table shows formats and encodings introduced by the Pentium Family.

Table B-12. Pentium Family Instruction Formats and Encodings

Instruction and Format	Encoding
CMPXCHG8B – Compare and Exchange 8 Bytes	
memory, register	0000 1111 : 1100 0111 : mod 001 r/m

B.4. MMX INSTRUCTION FORMATS AND ENCODINGS

All 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.4.1. 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-13 shows the encoding of this gg field.

Table B-13. Encoding of Granularity of Data Field (gg)

gg	Granularity of Data
00	Packed Bytes
01	Packed Words
10	Packed Doublewords
11	Quadword

B.4.2. 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.4.3. MMX Instruction Formats and Encodings Table

Table B-14 shows the formats and encodings of the integer instructions.

Table B-14. MMX Instruction Formats and Encodings

Instruction and Format	Encoding
EMMS - Empty MMX technology state	0000 1111:01110111
MOVD - Move doubleword	
reg to mmreg	0000 1111:01101110: 11 mmxreg reg
reg from mmxreg	0000 1111:01111110: 11 mmxreg reg
mem to mmxreg	0000 1111:01101110: mod mmxreg r/m
mem from mmxreg	0000 1111:01111110: mod mmxreg r/m

Table B-14. MMX Instruction Formats and Encodings (Contd.)

Instruction and Format	Encoding
MOVQ - Move quadword	
mmxreg2 to mmxreg1	0000 1111:01101111: 11 mmxreg1 mmxreg2
mmxreg2 from mmxreg1	0000 1111:01111111: 11 mmxreg1 mmxreg2
mem to mmxreg	0000 1111:01101111: mod mmxreg r/m
mem from mmxreg	0000 1111:01111111: mod mmxreg r/m
PACKSSDW[1] - Pack dword to word data (signed with saturation)	
mmxreg2 to mmxreg1	0000 1111:01101011: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111:01101011: mod mmxreg r/m
PACKSSWB[1] - Pack word to byte data (signed with saturation)	
mmxreg2 to mmxreg1	0000 1111:01100011: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111:01100011: mod mmxreg r/m
PACKUSWB[1] - Pack word to byte data (unsigned with saturation)	
mmxreg2 to mmxreg1	0000 1111:01100111: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111:01100111: mod mmxreg r/m
PADD - Add with wrap-around	
mmxreg2 to mmxreg1	0000 1111: 111111gg: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111: 111111gg: mod mmxreg r/m
PADDS - Add signed with saturation	
mmxreg2 to mmxreg1	0000 1111: 111011gg: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111: 111011gg: mod mmxreg r/m
PADDUS - Add unsigned with saturation	
mmxreg2 to mmxreg1	0000 1111: 110111gg: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111: 110111gg: mod mmxreg r/m
PAND - Bitwise And	
mmxreg2 to mmxreg1	0000 1111:11011011: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111:11011011: mod mmxreg r/m
PANDN - Bitwise AndNot	
mmxreg2 to mmxreg1	0000 1111:11011111: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111:11011111: mod mmxreg r/m

Table B-14. MMX Instruction Formats and Encodings (Contd.)

Instruction and Format	Encoding
PCMPEQ - Packed compare for equality	
mmxreg1 with mmxreg2	0000 1111:011101gg: 11 mmxreg1 mmxreg2
mmxreg with memory	0000 1111:011101gg: mod mmxreg r/m
PCMPGT - Packed compare greater (signed)	
mmxreg1 with mmxreg2	0000 1111:011001gg: 11 mmxreg1 mmxreg2
mmxreg with memory	0000 1111:011001gg: mod mmxreg r/m
PMADDWD - Packed multiply add	
mmxreg2 to mmxreg1	0000 1111:11110101: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111:11110101: 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
PMULHW - Packed multiplication, store high word	
mmxreg2 to mmxreg1	0000 1111:11100101: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111:11100101: mod mmxreg r/m
PMULLW - Packed multiplication, store low word	
mmxreg2 to mmxreg1	0000 1111:11010101: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111:11010101: mod mmxreg r/m
POR - Bitwise Or	
mmxreg2 to mmxreg1	0000 1111:11101011: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111:11101011: mod mmxreg r/m
PSLL^{2} - Packed shift left logical	
mmxreg1 by mmxreg2	0000 1111:111100gg: 11 mmxreg1 mmxreg2
mmxreg by memory	0000 1111:111100gg: mod mmxreg r/m
mmxreg by immediate	0000 1111:011100gg: 11 110 mmxreg: imm8 data
PSRA^{2} - Packed shift right arithmetic	
mmxreg1 by mmxreg2	0000 1111:111000gg: 11 mmxreg1 mmxreg2
mmxreg by memory	0000 1111:111000gg: mod mmxreg r/m
mmxreg by immediate	0000 1111:011100gg: 11 100 mmxreg: imm8 data

Table B-14. MMX Instruction Formats and Encodings (Contd.)

Instruction and Format	Encoding
PSRL[2] - Packed shift right logical	
mmxreg1 by mmxreg2	0000 1111:110100gg: 11 mmxreg1 mmxreg2
mmxreg by memory	0000 1111:110100gg: mod mmxreg r/m
mmxreg by immediate	0000 1111:011100gg: 11 010 mmxreg: imm8 data
PSUB - Subtract with wrap-around	
mmxreg2 from mmxreg1	0000 1111:111110gg: 11 mmxreg1 mmxreg2
memory from mmxreg	0000 1111:111110gg: mod mmxreg r/m
PSUBS - Subtract signed with saturation	
mmxreg2 from mmxreg1	0000 1111:111010gg: 11 mmxreg1 mmxreg2
memory from mmxreg	0000 1111:111010gg: mod mmxreg r/m
PSUBUS - Subtract unsigned with saturation	
mmxreg2 from mmxreg1	0000 1111:110110gg: 11 mmxreg1 mmxreg2
memory from mmxreg	0000 1111:110110gg: mod mmxreg r/m
PUNPCKH - Unpack high data to next larger type	
mmxreg2 to mmxreg1	0000 1111:011010gg: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111:011010gg: mod mmxreg r/m
PUNPCKL - Unpack low data to next larger type	
mmxreg2 to mmxreg1	0000 1111:011000gg: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111:011000gg: mod mmxreg r/m
PXOR - Bitwise Xor	
mmxreg2 to mmxreg1	0000 1111:11101111: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111:11101111: 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.5. P6 FAMILY INSTRUCTION FORMATS AND ENCODINGS

Table B-15 shows the formats and encodings for several instructions that were introduced into the IA-32 architecture in the P6 family processors.

Table B-15. 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)
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	00001111:10101110: mod^A 001 r/m
FXSAVE—Save x87 FPU, MMX, SSE, and SSE2 State	00001111:10101110: mod^A 000 r/m
SYSENTER—Fast System Call	00001111:00110100
SYSEXIT—Fast Return from Fast System Call	00001111:00110101

NOTE:

1 In FXSAVE and FXRSTOR, “mod=11” is reserved.

B.6. 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-16, B-17, and B-18) 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. These mandatory prefixes are included in the tables.

Table B-16. Formats and Encodings of SSE Floating-Point Instructions

Instruction and Format	Encoding
ADDPS—Add Packed Single-Precision Floating-Point Values	
xmmreg to xmmreg	00001111:01011000:11 xmmreg1 xmmreg2
mem to xmmreg	00001111:01011000: mod xmmreg r/m
ADDSS—Add Scalar Single-Precision Floating-Point Values	
xmmreg to xmmreg	11110011:00001111:01011000:11 xmmreg1 xmmreg2
mem to xmmreg	11110011:00001111:01011000: mod xmmreg r/m
ANDNPS—Bitwise Logical AND NOT of Packed Single-Precision Floating-Point Values	
xmmreg to xmmreg	00001111:01010101:11 xmmreg1 xmmreg2
mem to xmmreg	00001111:01010101: mod xmmreg r/m
ANDPS—Bitwise Logical AND of Packed Single-Precision Floating-Point Values	
xmmreg to xmmreg	00001111:01010100:11 xmmreg1 xmmreg2
mem to xmmreg	00001111:01010100: mod xmmreg r/m
CMPPS—Compare Packed Single-Precision Floating-Point Values	
xmmreg to xmmreg, imm8	00001111:11000010:11 xmmreg1 xmmreg2: imm8
mem to xmmreg, imm8	00001111:11000010: mod xmmreg r/m: imm8
CMPSS—Compare Scalar Single-Precision Floating-Point Values	
xmmreg to xmmreg, imm8	11110011:00001111:11000010:11 xmmreg1 xmmreg2: imm8
mem to xmmreg, imm8	11110011:00001111:11000010: mod xmmreg r/m: imm8

Table B-16. Formats and Encodings of SSE Floating-Point Instructions (Contd.)

Instruction and Format	Encoding
COMISS—Compare Scalar Ordered Single-Precision Floating-Point Values and Set EFLAGS	
xmmreg to xmmreg	00001111:00101111:11 xmmreg1 xmmreg2
mem to xmmreg	00001111:00101111: mod xmmreg r/m
CVTPI2PS—Convert Packed Doubleword Integers to Packed Single-Precision Floating-Point Values	
mmreg to xmmreg	00001111:00101010:11 xmmreg1 mmreg1
mem to xmmreg	00001111:00101010: mod xmmreg r/m
CVTPS2PI—Convert Packed Single-Precision Floating-Point Values to Packed Doubleword Integers	
xmmreg to mmreg	00001111:00101101:11 mmreg1 xmmreg1
mem to mmreg	00001111:00101101: mod mmreg r/m
CVTSI2SS—Convert Doubleword Integer to Scalar Single-Precision Floating-Point Value	
r32 to xmmreg1	11110011:00001111:00101010:11 xmmreg r32
mem to xmmreg	11110011:00001111:00101010: mod xmmreg r/m
CVTSS2SI—Convert Scalar Single-Precision Floating-Point Value to Doubleword Integer	
xmmreg to r32	11110011:00001111:00101101:11 r32 xmmreg
mem to r32	11110011:00001111:00101101: mod r32 r/m
CVTTPS2PI—Convert with Truncation Packed Single-Precision Floating-Point Values to Packed Doubleword Integers	
xmmreg to mmreg	00001111:00101100:11 mmreg1 xmmreg1
mem to mmreg	00001111:00101100: mod mmreg r/m
CVTTSS2SI—Convert with Truncation Scalar Single-Precision Floating-Point Value to Doubleword Integer	
xmmreg to r32	11110011:00001111:00101100:11 r32 xmmreg1
mem to r32	11110011:00001111:00101100: mod r32 r/m
DIVPS—Divide Packed Single-Precision Floating-Point Values	
xmmreg to xmmreg	00001111:01011110:11 xmmreg1 xmmreg2
mem to xmmreg	00001111:01011110: mod xmmreg r/m

Table B-16. Formats and Encodings of SSE Floating-Point Instructions (Contd.)

Instruction and Format	Encoding
DIVSS—Divide Scalar Single-Precision Floating-Point Values	
xmmreg to xmmreg	11110011:00001111:01011110:11 xmmreg1 xmmreg2
mem to xmmreg	11110011:00001111:01011110: mod xmmreg r/m
LDMXCSR—Load MXCSR Register State	
m32 to MXCSR	00001111:10101110:mod^{A} 010 mem
MAXPS—Return Maximum Packed Single-Precision Floating-Point Values	
xmmreg to xmmreg	00001111:01011111:11 xmmreg1 xmmreg2
mem to xmmreg	00001111:01011111: mod xmmreg r/m
MAXSS—Return Maximum Scalar Double-Precision Floating-Point Value	
xmmreg to xmmreg	11110011:00001111:01011111:11 xmmreg1 xmmreg2
mem to xmmreg	11110011:00001111:01011111: mod xmmreg r/m
MINPS—Return Minimum Packed Double-Precision Floating-Point Values	
xmmreg to xmmreg	00001111:01011101:11 xmmreg1 xmmreg2
mem to xmmreg	00001111:01011101: mod xmmreg r/m
MINSS—Return Minimum Scalar Double-Precision Floating-Point Value	
xmmreg to xmmreg	11110011:00001111:01011101:11 xmmreg1 xmmreg2
mem to xmmreg	11110011:00001111:01011101: mod xmmreg r/m
MOVAPS—Move Aligned Packed Single-Precision Floating-Point Values	
xmmreg2 to xmmreg1	00001111:00101000:11 xmmreg2 xmmreg1
mem to xmmreg1	00001111:00101000: mod xmmreg r/m
xmmreg1 to xmmreg2	00001111:00101001:11 xmmreg1 xmmreg2
xmmreg1 to mem	00001111:00101001: mod xmmreg r/m
MOVHLPS—Move Packed Single-Precision Floating-Point Values High to Low	
xmmreg to xmmreg	00001111:00010010:11 xmmreg1 xmmreg2
MOVHPS—Move High Packed Single-Precision Floating-Point Values	
mem to xmmreg	00001111:00010110: mod xmmreg r/m
xmmreg to mem	00001111:00010111: mod xmmreg r/m

Table B-16. Formats and Encodings of SSE Floating-Point Instructions (Contd.)

Instruction and Format	Encoding
MOVLHPS—Move Packed Single-Precision Floating-Point Values Low to High	
xmmreg to xmmreg	00001111:00010110:11 xmmreg1 xmmreg2
MOVLPS—Move Low Packed Single-Precision Floating-Point Values	
mem to xmmreg	00001111:00010010: mod xmmreg r/m
xmmreg to mem	00001111:00010011: mod xmmreg r/m
MOVMSKPS—Extract Packed Single-Precision Floating-Point Sign Mask	
xmmreg to r32	00001111:01010000:11 r32 xmmreg
MOVSS—Move Scalar Single-Precision Floating-Point Values	
xmmreg2 to xmmreg1	11110011:00001111:00010000:11 xmmreg2 xmmreg1
mem to xmmreg1	11110011:00001111:00010000: mod xmmreg r/m
xmmreg1 to xmmreg2	11110011:00001111:00010001:11 xmmreg1 xmmreg2
xmmreg1 to mem	11110011:00001111:00010001: mod xmmreg r/m
MOVUPS—Move Unaligned Packed Single-Precision Floating-Point Values	
xmmreg2 to xmmreg1	00001111:00010000:11 xmmreg2 xmmreg1
mem to xmmreg1	00001111:00010000: mod xmmreg r/m
xmmreg1 to xmmreg2	00001111:00010001:11 xmmreg1 xmmreg2
xmmreg1 to mem	00001111:00010001: mod xmmreg r/m
MULPS—Multiply Packed Single-Precision Floating-Point Values	
xmmreg to xmmreg	00001111:01011001:11 xmmreg1 xmmreg2
mem to xmmreg	00001111:01011001: mod xmmreg rm
MULSS—Multiply Scalar Single-Precision Floating-Point Values	
xmmreg to xmmreg	11110011:00001111:01011001:11 xmmreg1 xmmreg2
mem to xmmreg	11110011:00001111:01011001: mod xmmreg r/m
ORPS—Bitwise Logical OR of Single-Precision Floating-Point Values	
xmmreg to xmmreg	00001111:01010110:11 xmmreg1 xmmreg2
mem to xmmreg	00001111:01010110 mod xmmreg r/m

Table B-16. Formats and Encodings of SSE Floating-Point Instructions (Contd.)

Instruction and Format	Encoding
RCPPS—Compute Reciprocals of Packed Single-Precision Floating-Point Values	
xmmreg to xmmreg	00001111:01010011:11 xmmreg1 xmmreg2
mem to xmmreg	00001111:01010011: mod xmmreg r/m
RCPSS—Compute Reciprocals of Scalar Single-Precision Floating-Point Value	
xmmreg to xmmreg	11110011:00001111:01010011:11 xmmreg1 xmmreg2
mem to xmmreg	11110011:00001111:01010011: mod xmmreg r/m
RSQRTPS—Compute Reciprocals of Square Roots of Packed Single-Precision Floating-Point Values	
xmmreg to xmmreg	00001111:01010010:11 xmmreg1 xmmreg2
mem to xmmreg	00001111:01010010 mode xmmreg r/m
RSQRTSS—Compute Reciprocals of Square Roots of Scalar Single-Precision Floating-Point Value	
xmmreg to xmmreg	11110011:00001111:01010010:11 xmmreg1 xmmreg2
mem to xmmreg	11110011:00001111:01010010 mod xmmreg r/m
SHUFPS—Shuffle Packed Single-Precision Floating-Point Values	
xmmreg to xmmreg, imm8	00001111:11000110:11 xmmreg1 xmmreg2: imm8
mem to xmmreg, imm8	00001111:11000110: mod xmmreg r/m: imm8
SQRTPS—Compute Square Roots of Packed Single-Precision Floating-Point Values	
xmmreg to xmmreg	00001111:01010001:11 xmmreg1 xmmreg 2
mem to xmmreg	00001111:01010001 mod xmmreg r/m
SQRTSS—Compute Square Root of Scalar Single-Precision Floating-Point Value	
xmmreg to xmmreg	11110011:00001111:01010001:11 xmmreg1 xmmreg 2
mem to xmmreg	11110011:00001111:01010001:mod xmmreg r/m
STMXCSR—Store MXCSR Register State	
MXCSR to mem	00001111:10101110:mod^A 011 mem
SUBPS—Subtract Packed Single-Precision Floating-Point Values	
xmmreg to xmmreg	00001111:01011100:11 xmmreg1 xmmreg2
mem to xmmreg	00001111:01011100:mod xmmreg r/m

Table B-16. Formats and Encodings of SSE Floating-Point Instructions (Contd.)

Instruction and Format	Encoding
SUBSS—Subtract Scalar Single-Precision Floating-Point Values	
xmmreg to xmmreg	11110011:00001111:01011100:11 xmmreg1 xmmreg2
mem to xmmreg	11110011:00001111:01011100:mod xmmreg r/m
UCOMISS—Unordered Compare Scalar Ordered Single-Precision Floating-Point Values and Set EFLAGS	
xmmreg to xmmreg	00001111:00101110:11 xmmreg1 xmmreg2
mem to xmmreg	00001111:00101110 mod xmmreg r/m
UNPCKHPS—Unpack and Interleave High Packed Single-Precision Floating-Point Values	
xmmreg to xmmreg	00001111:00010101:11 xmmreg1 xmmreg2
mem to xmmreg	00001111:00010101 mod xmmreg r/m
UNPCKLPS—Unpack and Interleave Low Packed Single-Precision Floating-Point Values	
xmmreg to xmmreg	00001111:00010100:11 xmmreg1 xmmreg2
mem to xmmreg	00001111:00010100 mod xmmreg r/m
XORPS—Bitwise Logical XOR of Single-Precision Floating-Point Values	
xmmreg to xmmreg	00001111:01010111:11 xmmreg1 xmmreg2
mem to xmmreg	00001111:01010111 mod xmmreg r/m

Table B-17. Formats and Encodings of SSE Integer Instructions

Instruction and Format	Encoding
PAVGB/PAVGW—Average Packed Integers	
mmreg to mmreg	00001111:11100000:11 mmreg1 mmreg2
	00001111:11100011:11 mmreg1 mmreg2
mem to mmreg	00001111:11100000 mod mmreg r/m
	00001111:11100011 mod mmreg r/m
PEXTRW—Extract Word	
mmreg to reg32, imm8	00001111:11000101:11 r32 mmreg: imm8
PINSRW - Insert Word	
reg32 to mmreg, imm8	00001111:11000100:11 mmreg r32: imm8
m16 to mmreg, imm8	00001111:11000100 mod mmreg r/m: imm8
PMAXSW—Maximum of Packed Signed Word Integers	
mmreg to mmreg	00001111:11101110:11 mmreg1 mmreg2
mem to mmreg	00001111:11101110 mod mmreg r/m
PMAXUB—Maximum of Packed Unsigned Byte Integers	
mmreg to mmreg	00001111:11011110:11 mmreg1 mmreg2
mem to mmreg	00001111:11011110 mod mmreg r/m
PMINSW—Minimum of Packed Signed Word Integers	
mmreg to mmreg	00001111:11101010:11 mmreg1 mmreg2
mem to mmreg	00001111:11101010 mod mmreg r/m
PMINUB—Minimum of Packed Unsigned Byte Integers	
mmreg to mmreg	00001111:11011010:11 mmreg1 mmreg2
mem to mmreg	00001111:11011010 mod mmreg r/m
PMOVMSKB - Move Byte Mask To Integer	
mmreg to reg32	00001111:11010111:11 r32 mmreg
PMULHUW—Multiply Packed Unsigned Integers and Store High Result	
mmreg to mmreg	00001111:11100100:11 mmreg1 mmreg2
mem to mmreg	00001111:11100100 mod mmreg r/m
PSADBW—Compute Sum of Absolute Differences	
mmreg to mmreg	00001111:11110110:11 mmreg1 mmreg2
mem to mmreg	00001111:11110110 mod mmreg r/m

Table B-17. Formats and Encodings of SSE Integer Instructions (Contd.)

Instruction and Format	Encoding
PSHUFW—Shuffle Packed Words	
mmreg to mmreg, imm8	00001111:01110000:11 mmreg1 mmreg2: imm8
mem to mmreg, imm8	00001111:01110000:11 mod mmreg r/m: imm8

Table B-18. Format and Encoding of SSE Cacheability and Memory Ordering Instructions

Instruction and Format	Encoding
MASKMOVQ—Store Selected Bytes of Quadword	
mmreg to mmreg	00001111:11110111:11 mmreg1 mmreg2
MOVNTPS—Store Packed Single-Precision Floating-Point Values Using Non-Temporal Hint	
xmmreg to mem	00001111:00101011: mod xmmreg r/m
MOVNTQ—Store Quadword Using Non-Temporal Hint	
mmreg to mem	00001111:11100111: mod mmreg r/m
PREFETCHT0—Prefetch Temporal to All Cache Levels	00001111:00011000:mod^{A} 001 mem
PREFETCHT1—Prefetch Temporal to First Level Cache	00001111:00011000:mod^{A} 010 mem
PREFETCHT2—Prefetch Temporal to Second Level Cache	00001111:00011000:mod^{A} 011 mem
PREFETCHNTA—Prefetch Non-Temporal to All Cache Levels	00001111:00011000:mod^{A} 000 mem
SFENCE—Store Fence	00001111:10101110:11 111 000

B.7. 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.7.1. 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-19 shows the encoding of this gg field.

Table B-19. Encoding of Granularity of Data Field (gg)

gg	Granularity of Data
00	Packed Bytes
01	Packed Words
10	Packed Doublewords
11	Quadword

Table B-20. Formats and Encodings of SSE2 Floating-Point Instructions

Instruction and Format	Encoding
ADDPD - Add Packed Double-Precision Floating-Point Values	
xmmreg to xmmreg	01100110:00001111:01011000:11 xmmreg1 xmmreg2
mem to xmmreg	01100110:00001111:01011000: mod xmmreg r/m
ADDSD - Add Scalar Double-Precision Floating-Point Values	
xmmreg to xmmreg	11110010:00001111:01011000:11 xmmreg1 xmmreg2
mem to xmmreg	11110010:00001111:01011000: mod xmmreg r/m
ANDNPD—Bitwise Logical AND NOT of Packed Double-Precision Floating-Point Values	
xmmreg to xmmreg	01100110:00001111:01010101:11 xmmreg1 xmmreg2
mem to xmmreg	01100110:00001111:01010101: mod xmmreg r/m

Table B-20. Formats and Encodings of SSE2 Floating-Point Instructions (Contd.)

Instruction and Format	Encoding
ANDPD—Bitwise Logical AND of Packed Double-Precision Floating-Point Values	
xmmreg to xmmreg	01100110:00001111:01010100:11 xmmreg1 xmmreg2
mem to xmmreg	01100110:00001111:01010100: mod xmmreg r/m
CMPPD—Compare Packed Double-Precision Floating-Point Values	
xmmreg to xmmreg, imm8	01100110:00001111:11000010:11 xmmreg1 xmmreg2: imm8
mem to xmmreg, imm8	01100110:00001111:11000010: mod xmmreg r/m: imm8
CMPSD—Compare Scalar Double-Precision Floating-Point Values	
xmmreg to xmmreg, imm8	11110010:00001111:11000010:11 xmmreg1 xmmreg2: imm8
mem to xmmreg, imm8	11110010:00001111:11000010: mod xmmreg r/m: imm8
COMISD—Compare Scalar Ordered Double-Precision Floating-Point Values and Set EFLAGS	
xmmreg to xmmreg	01100110:00001111:00101111:11 xmmreg1 xmmreg2
mem to xmmreg	01100110:00001111:00101111: mod xmmreg r/m
CVTPI2PD—Convert Packed Doubleword Integers to Packed Double-Precision Floating-Point Values	
mmreg to xmmreg	01100110:00001111:00101010:11 xmmreg1 mmreg1
mem to xmmreg	01100110:00001111:00101010: mod xmmreg r/m
CVTPD2PI—Convert Packed Double-Precision Floating-Point Values to Packed Doubleword Integers	
xmmreg to mmreg	01100110:00001111:00101101:11 mmreg1 xmmreg1
mem to mmreg	01100110:00001111:00101101: mod mmreg r/m
CVTSI2SD—Convert Doubleword Integer to Scalar Double-Precision Floating-Point Value	
r32 to xmmreg1	11110010:00001111:00101010:11 xmmreg r32
mem to xmmreg	11110010:00001111:00101010: mod xmmreg r/m

Table B-20. Formats and Encodings of SSE2 Floating-Point Instructions (Contd.)

Instruction and Format	Encoding
CVTSD2SI—Convert Scalar Double-Precision Floating-Point Value to Doubleword Integer	
xmmreg to r32	11110010:00001111:00101101:11 r32 xmmreg
mem to r32	11110010:00001111:00101101: mod r32 r/m
CVTTPD2PI—Convert with Truncation Packed Double-Precision Floating-Point Values to Packed Doubleword Integers	
xmmreg to mmreg	01100110:00001111:00101100:11 mmreg xmmreg
mem to mmreg	01100110:00001111:00101100: mod mmreg r/m
CVTTSD2SI—Convert with Truncation Scalar Double-Precision Floating-Point Value to Doubleword Integer	
xmmreg to r32	11110010:00001111:00101100:11 r32 xmmreg
mem to r32	11110010:00001111:00101100: mod r32 r/m
CVTPD2PS—Covert Packed Double-Precision Floating-Point Values to Packed Single-Precision Floating-Point Values	
xmmreg to xmmreg	01100110:00001111:01011010:11 xmmreg1 xmmreg2
mem to xmmreg	01100110:00001111:01011010: mod xmmreg r/m
CVTPS2PD—Covert Packed Single-Precision Floating-Point Values to Packed Double-Precision Floating-Point Values	
xmmreg to xmmreg	00001111:01011010:11 xmmreg1 xmmreg2
mem to xmmreg	00001111:01011010: mod xmmreg r/m
CVTSD2SS—Covert Scalar Double-Precision Floating-Point Value to Scalar Single-Precision Floating-Point Value	
xmmreg to xmmreg	11110010:00001111:01011010:11 xmmreg1 xmmreg2
mem to xmmreg	11110010:00001111:01011010: mod xmmreg r/m
CVTSS2SD—Covert Scalar Single-Precision Floating-Point Value to Scalar Double-Precision Floating-Point Value	
xmmreg to xmmreg	11110011:00001111:01011010:11 xmmreg1 xmmreg2
mem to xmmreg	11110011:00001111:01011010: mod xmmreg r/m

Table B-20. Formats and Encodings of SSE2 Floating-Point Instructions (Contd.)

Instruction and Format	Encoding
CVTPD2DQ—Convert Packed Double-Precision Floating-Point Values to Packed Doubleword Integers	
xmmreg to xmmreg	11110010:00001111:11100110:11 xmmreg1 xmmreg2
mem to xmmreg	11110010:00001111:11100110: mod xmmreg r/m
CVTTPD2DQ—Convert With Truncation Packed Double-Precision Floating-Point Values to Packed Doubleword Integers	
xmmreg to xmmreg	01100110:00001111:11100110:11 xmmreg1 xmmreg2
mem to xmmreg	01100110:00001111:11100110: mod xmmreg r/m
CVTDQ2PD—Convert Packed Doubleword Integers to Packed Single-Precision Floating-Point Values	
xmmreg to xmmreg	11110011:00001111:11100110:11 xmmreg1 xmmreg2
mem to xmmreg	11110011:00001111:11100110: mod xmmreg r/m
CVTPS2DQ—Convert Packed Single-Precision Floating-Point Values to Packed Doubleword Integers	
xmmreg to xmmreg	01100110:00001111:01011011:11 xmmreg1 xmmreg2
mem to xmmreg	01100110:00001111:01011011: mod xmmreg r/m
CVTTPS2DQ—Convert With Truncation Packed Single-Precision Floating-Point Values to Packed Doubleword Integers	
xmmreg to xmmreg	11110011:00001111:01011011:11 xmmreg1 xmmreg2
mem to xmmreg	11110011:00001111:01011011: mod xmmreg r/m
CVTDQ2PS—Convert Packed Doubleword Integers to Packed Double-Precision Floating-Point Values	
xmmreg to xmmreg	00001111:01011011:11 xmmreg1 xmmreg2
mem to xmmreg	00001111:01011011: mod xmmreg r/m
DIVPD—Divide Packed Double-Precision Floating-Point Values	
xmmreg to xmmreg	01100110:00001111:01011110:11 xmmreg1 xmmreg2
mem to xmmreg	01100110:00001111:01011110: mod xmmreg r/m
DIVSD—Divide Scalar Double-Precision Floating-Point Values	
xmmreg to xmmreg	11110010:00001111:01011110:11 xmmreg1 xmmreg2
mem to xmmreg	11110010:00001111:01011110: mod xmmreg r/m

Table B-20. Formats and Encodings of SSE2 Floating-Point Instructions (Contd.)

Instruction and Format	Encoding
MAXPD—Return Maximum Packed Double-Precision Floating-Point Values	
xmmreg to xmmreg	01100110:00001111:01011111:11 xmmreg1 xmmreg2
mem to xmmreg	01100110:00001111:01011111: mod xmmreg r/m
MAXSD—Return Maximum Scalar Double-Precision Floating-Point Value	
xmmreg to xmmreg	11110010:00001111:01011111:11 xmmreg1 xmmreg2
mem to xmmreg	11110010:00001111:01011111: mod xmmreg r/m
MINPD—Return Minimum Packed Double-Precision Floating-Point Values	
xmmreg to xmmreg	01100110:00001111:01011101:11 xmmreg1 xmmreg2
mem to xmmreg	01100110:00001111:01011101: mod xmmreg r/m
MINSD—Return Minimum Scalar Double-Precision Floating-Point Value	
xmmreg to xmmreg	11110010:00001111:01011101:11 xmmreg1 xmmreg2
mem to xmmreg	11110010:00001111:01011101: mod xmmreg r/m
MOVAPD—Move Aligned Packed Double-Precision Floating-Point Values	
xmmreg2 to xmmreg1	01100110:00001111:00101001:11 xmmreg2 xmmreg1
mem to xmmreg1	01100110:00001111:00101001: mod xmmreg r/m
xmmreg1 to xmmreg2	01100110:00001111:00101000:11 xmmreg1 xmmreg2
xmmreg1 to mem	01100110:00001111:00101000: mod xmmreg r/m
MOVHPD—Move High Packed Double-Precision Floating-Point Values	
mem to xmmreg	01100110:00001111:00010111: mod xmmreg r/m
xmmreg to mem	01100110:00001111:00010110: mod xmmreg r/m
MOVLPD—Move Low Packed Double-Precision Floating-Point Values	
mem to xmmreg	01100110:00001111:00010011: mod xmmreg r/m
xmmreg to mem	01100110:00001111:00010010: mod xmmreg r/m
MOVMSKPD—Extract Packed Double-Precision Floating-Point Sign Mask	
xmmreg to r32	01100110:00001111:01010000:11 r32 xmmreg

Table B-20. Formats and Encodings of SSE2 Floating-Point Instructions (Contd.)

Instruction and Format	Encoding
MOVSD—Move Scalar Double-Precision Floating-Point Values	
xmmreg2 to xmmreg1	11110010:00001111:00010001:11 xmmreg2 xmmreg1
mem to xmmreg1	11110010:00001111:00010001: mod xmmreg r/m
xmmreg1 to xmmreg2	11110010:00001111:00010000:11 xmmreg1 xmmreg2
xmmreg1 to mem	11110010:00001111:00010000: mod xmmreg r/m
MOVUPD—Move Unaligned Packed Double-Precision Floating-Point Values	
xmmreg2 to xmmreg1	01100110:00001111:00010001:11 xmmreg2 xmmreg1
mem to xmmreg1	01100110:00001111:00010001: mod xmmreg r/m
xmmreg1 to xmmreg2	01100110:00001111:00010000:11 xmmreg1 xmmreg2
xmmreg1 to mem	01100110:00001111:00010000: mod xmmreg r/m
MULPD—Multiply Packed Double-Precision Floating-Point Values	
xmmreg to xmmreg	01100110:00001111:01011001:11 xmmreg1 xmmreg2
mem to xmmreg	01100110:00001111:01011001: mod xmmreg rm
MULSD—Multiply Scalar Double-Precision Floating-Point Values	
xmmreg to xmmreg	11110010:00001111:01011001:11 xmmreg1 xmmreg2
mem to xmmreg	11110010:00001111:01011001: mod xmmreg r/m
ORPD—Bitwise Logical OR of Double-Precision Floating-Point Values	
xmmreg to xmmreg	01100110:00001111:01010110:11 xmmreg1 xmmreg2
mem to xmmreg	01100110:00001111:01010110: mod xmmreg r/m
SHUFPD—Shuffle Packed Double-Precision Floating-Point Values	
xmmreg to xmmreg, imm8	01100110:00001111:11000110:11 xmmreg1 xmmreg2: imm8
mem to xmmreg, imm8	01100110:00001111:11000110: mod xmmreg r/m: imm8
SQRTPD—Compute Square Roots of Packed Double-Precision Floating-Point Values	
xmmreg to xmmreg	01100110:00001111:01010001:11 xmmreg1 xmmreg 2
mem to xmmreg	01100110:00001111:01010001: mod xmmreg r/m

Table B-20. Formats and Encodings of SSE2 Floating-Point Instructions (Contd.)

Instruction and Format	Encoding
SQRTSD—Compute Square Root of Scalar Double-Precision Floating-Point Value	
xmmreg to xmmreg	11110010:00001111:01010001:11 xmmreg1 xmmreg 2
mem to xmmreg	11110010:00001111:01010001: mod xmmreg r/m
SUBPD—Subtract Packed Double-Precision Floating-Point Values	
xmmreg to xmmreg	01100110:00001111:01011100:11 xmmreg1 xmmreg2
mem to xmmreg	01100110:00001111:01011100: mod xmmreg r/m
SUBSD—Subtract Scalar Double-Precision Floating-Point Values	
xmmreg to xmmreg	11110010:00001111:01011100:11 xmmreg1 xmmreg2
mem to xmmreg	11110010:00001111:01011100: mod xmmreg r/m
UCOMISD—Unordered Compare Scalar Ordered Double-Precision Floating-Point Values and Set EFLAGS	
xmmreg to xmmreg	01100110:00001111:00101110:11 xmmreg1 xmmreg2
mem to xmmreg	01100110:00001111:00101110: mod xmmreg r/m
UNPCKHPD—Unpack and Interleave High Packed Double-Precision Floating-Point Values	
xmmreg to xmmreg	01100110:00001111:00010101:11 xmmreg1 xmmreg2
mem to xmmreg	01100110:00001111:00010101: mod xmmreg r/m
UNPCKLPD—Unpack and Interleave Low Packed Double-Precision Floating-Point Values	
xmmreg to xmmreg	01100110:00001111:00010100:11 xmmreg1 xmmreg2
mem to xmmreg	01100110:00001111:00010100: mod xmmreg r/m
XORPD—Bitwise Logical OR of Double-Precision Floating-Point Values	
xmmreg to xmmreg	01100110:00001111:01010111:11 xmmreg1 xmmreg2
mem to xmmreg	01100110:00001111:01010111: mod xmmreg r/m

Table B-21. Formats and Encodings of SSE2 Integer Instructions

Instruction and Format	Encoding
MOVD - Move Doubleword	
reg to xmmeg	01100110:0000 1111:01101110: 11 xmmreg reg
reg from xmmreg	01100110:0000 1111:01111110: 11 xmmreg reg
mem to xmmreg	01100110:0000 1111:01101110: mod xmmreg r/m
mem from xmmreg	01100110:0000 1111:01111110: mod xmmreg r/m
MOVDQA—Move Aligned Double Quadword	
xmmreg to xmmreg	01100110:00001111:01101111:11 xmmreg1 xmmreg2
	01100110:00001111:01111111:11 xmmreg1 xmmreg2
mem to xmmreg	01100110:00001111:01101111: mod xmmreg r/m
mem from xmmreg	01100110:00001111:01111111: mod xmmreg r/m
MOVDQU—Move Unaligned Double Quadword	
xmmreg to xmmreg	11110011:00001111:01101111:11 xmmreg1 xmmreg2
	11110011:00001111:01111111:11 xmmreg1 xmmreg2
mem to xmmreg	11110011:00001111:01101111: mod xmmreg r/m
mem from xmmreg	11110011:00001111:01111111: mod xmmreg r/m
MOVQ2DQ—Move Quadword from MMX to XMM Register	
mmreg to xmmreg	11110011:00001111:11010110:11 mmreg1 mmreg2
MOVDQ2Q—Move Quadword from XMM to MMX Register	
xmmreg to mmreg	11110010:00001111:11010110:11 mmreg1 mmreg2
MOVQ - Move Quadword	
xmmreg2 to xmmreg1	11110011:00001111:01111110: 11 xmmreg1 xmmreg2
xmmreg2 from xmmreg1	01100110:00001111:11010110: 11 xmmreg1 xmmreg2
mem to xmmreg	11110011:00001111:01111110: mod xmmreg r/m
mem from xmmreg	01100110:00001111:11010110: mod xmmreg r/m
PACKSSDW[1] **- Pack Dword To Word Data (signed with saturation)**	
xmmreg2 to xmmreg1	01100110:0000 1111:01101011: 11 xmmreg1 xmmreg2
memory to xmmreg	01100110:0000 1111:01101011: mod xmmreg r/m

Table B-21. Formats and Encodings of SSE2 Integer Instructions (Contd.)

Instruction and Format	Encoding
PACKSSWB - Pack Word To Byte Data (signed with saturation)	
xmmreg2 to xmmreg1	01100110:0000 1111:01100011: 11 xmmreg1 xmmreg2
memory to xmmreg	01100110:0000 1111:01100011: mod xmmreg r/m
PACKUSWB - Pack Word To Byte Data (unsigned with saturation)	
xmmreg2 to xmmreg1	01100110:0000 1111:01100111: 11 xmmreg1 xmmreg2
memory to xmmreg	01100110:0000 1111:01100111: mod xmmreg r/m
PADDQ—Add Packed Quadword Integers	
mmreg to mmreg	00001111:11010100:11 mmreg1 mmreg2
mem to mmreg	00001111:11010100: mod mmreg r/m
xmmreg to xmmreg	01100110:00001111:11010100:11 xmmreg1 xmmreg2
mem to xmmreg	01100110:00001111:11010100: mod xmmreg r/m
PADD - Add With Wrap-around	
xmmreg2 to xmmreg1	01100110:0000 1111: 111111gg: 11 xmmreg1 xmmreg2
memory to xmmreg	01100110:0000 1111: 111111gg: mod xmmreg r/m
PADDS - Add Signed With Saturation	
xmmreg2 to xmmreg1	01100110:0000 1111: 111011gg: 11 xmmreg1 xmmreg2
memory to xmmreg	01100110:0000 1111: 111011gg: mod xmmreg r/m
PADDUS - Add Unsigned With Saturation	
xmmreg2 to xmmreg1	01100110:0000 1111: 110111gg: 11 xmmreg1 xmmreg2
memory to xmmreg	01100110:0000 1111: 110111gg: mod xmmreg r/m
PAND - Bitwise And	
xmmreg2 to xmmreg1	01100110:0000 1111:11011011: 11 xmmreg1 xmmreg2
memory to xmmreg	01100110:0000 1111:11011011: mod xmmreg r/m
PANDN - Bitwise AndNot	
xmmreg2 to xmmreg1	01100110:0000 1111:11011111: 11 xmmreg1 xmmreg2
memory to xmmreg	01100110:0000 1111:11011111: mod xmmreg r/m
PAVGB—Average Packed Integers	
xmmreg to xmmreg	01100110:00001111:11100000:11 xmmreg1 xmmreg2
mem to xmmreg	01100110:00001111:11100000 mod xmmreg r/m

Table B-21. Formats and Encodings of SSE2 Integer Instructions (Contd.)

Instruction and Format	Encoding
PAVGW—Average Packed Integers	
xmmreg to xmmreg	01100110:00001111:11100011:11 xmmreg1 xmmreg2
mem to xmmreg	01100110:00001111:11100011 mod xmmreg r/m
PCMPEQ - Packed Compare For Equality	
xmmreg1 with xmmreg2	01100110:0000 1111:011101gg: 11 xmmreg1 xmmreg2
xmmreg with memory	01100110:0000 1111:011101gg: mod xmmreg r/m
PCMPGT - Packed Compare Greater (signed)	
xmmreg1 with xmmreg2	01100110:0000 1111:011001gg: 11 xmmreg1 xmmreg2
xmmreg with memory	01100110:0000 1111:011001gg: mod xmmreg r/m
PEXTRW—Extract Word	
xmmreg to reg32, imm8	01100110:00001111:11000101:11 r32 xmmreg: imm8
PINSRW - Insert Word	
reg32 to xmmreg, imm8	01100110:00001111:11000100:11 xmmreg r32: imm8
m16 to xmmreg, imm8	01100110:00001111:11000100 mod xmmreg r/m: imm8
PMADDWD - Packed Multiply Add	
xmmreg2 to xmmreg1	01100110:0000 1111:11110101: 11 xmmreg1 xmmreg2
memory to xmmreg	01100110:0000 1111:11110101: mod xmmreg r/m
PMAXSW—Maximum of Packed Signed Word Integers	
xmmreg to xmmreg	01100110:00001111:11101110:11 xmmreg1 xmmreg2
mem to xmmreg	01100110:00001111:11101110 mod xmmreg r/m
PMAXUB—Maximum of Packed Unsigned Byte Integers	
xmmreg to xmmreg	01100110:00001111:11011110:11 xmmreg1 xmmreg2
mem to xmmreg	01100110:00001111:11011110 mod xmmreg r/m
PMINSW—Minimum of Packed Signed Word Integers	
xmmreg to xmmreg	01100110:00001111:11101010:11 xmmreg1 xmmreg2
mem to xmmreg	01100110:00001111:11101010 mod xmmreg r/m
PMINUB—Minimum of Packed Unsigned Byte Integers	
xmmreg to xmmreg	01100110:00001111:11011010:11 xmmreg1 xmmreg2
mem to xmmreg	01100110:00001111:11011010 mod xmmreg r/m

Table B-21. Formats and Encodings of SSE2 Integer Instructions (Contd.)

Instruction and Format	Encoding
PMOVMSKB - Move Byte Mask To Integer	
xmmreg to reg32	01100110:00001111:11010111:11 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	01100110:0000 1111:11100101: 11 xmmreg1 xmmreg2
memory to xmmreg	01100110:0000 1111:11100101: mod xmmreg r/m
PMULLW - Packed Multiplication, store low word	
xmmreg2 to xmmreg1	01100110:0000 1111:11010101: 11 xmmreg1 xmmreg2
memory to xmmreg	01100110:0000 1111:11010101: mod xmmreg r/m
PMULUDQ—Multiply Packed Unsigned Doubleword Integers	
mmreg to mmreg	00001111:11110100:11 mmreg1 mmreg2
mem to mmreg	00001111:11110100: mod mmreg r/m
xmmreg to xmmreg	01100110:00001111:11110100:11 xmmreg1 xmmreg2
mem to xmmreg	01100110:00001111:11110100: mod xmmreg r/m
POR - Bitwise Or	
xmmreg2 to xmmreg1	01100110:0000 1111:11101011: 11 xmmreg1 xmmreg2
xmemory to xmmreg	01100110:0000 1111:11101011: mod xmmreg r/m
PSADBW—Compute Sum of Absolute Differences	
xmmreg to xmmreg	01100110:00001111:11110110:11 xmmreg1 xmmreg2
mem to xmmreg	01100110:00001111:11110110: mod xmmreg r/m
PSHUFLW—Shuffle Packed Low Words	
xmmreg to xmmreg, imm8	11110010:00001111:01110000:11 xmmreg1 xmmreg2: imm8
mem to xmmreg, imm8	11110010:00001111:01110000:11 mod xmmreg r/m: imm8
PSHUFHW—Shuffle Packed High Words	
xmmreg to xmmreg, imm8	11110011:00001111:01110000:11 xmmreg1 xmmreg2: imm8
mem to xmmreg, imm8	11110011:00001111:01110000:11 mod xmmreg r/m: imm8

Table B-21. Formats and Encodings of SSE2 Integer Instructions (Contd.)

Instruction and Format	Encoding
PSHUFD—Shuffle Packed Doublewords	
xmmreg to xmmreg, imm8	01100110:00001111:01110000:11 xmmreg1 xmmreg2: imm8
mem to xmmreg, imm8	01100110:00001111:01110000:11 mod xmmreg r/m: imm8
PSLLDQ—Shift Double Quadword Left Logical	
xmmreg, imm8	01100110:00001111:01110011:11 111 xmmreg: imm8
PSLL - Packed Shift Left Logical	
xmmreg1 by xmmreg2	01100110:0000 1111:111100gg: 11 xmmreg1 xmmreg2
xmmreg by memory	01100110:0000 1111:111100gg: mod xmmreg r/m
xmmreg by immediate	01100110:0000 1111:011100gg: 11 110 xmmreg: imm8 data
PSRA - Packed Shift Right Arithmetic	
xmmreg1 by xmmreg2	01100110:0000 1111:111000gg: 11 xmmreg1 xmmreg2
xmmreg by memory	01100110:0000 1111:111000gg: mod xmmreg r/m
xmmreg by immediate	01100110:0000 1111:011100gg: 11 100 xmmreg: imm8 data
PSRLDQ—Shift Double Quadword Right Logical	
xmmreg, imm8	01100110:00001111:01110011:11 011 xmmreg: imm8
PSRL - Packed Shift Right Logical	
xmmxreg1 by xmmxreg2	01100110:0000 1111:110100gg: 11 xmmreg1 xmmreg2
xmmxreg by memory	01100110:0000 1111:110100gg: mod xmmreg r/m
xmmxreg by immediate	01100110:0000 1111:011100gg: 11 010 xmmreg: imm8 data
PSUBQ—Subtract Packed Quadword Integers	
mmreg to mmreg	00001111:11111011:11 mmreg1 mmreg2
mem to mmreg	00001111:11111011: mod mmreg r/m
xmmreg to xmmreg	01100110:00001111:11111011:11 xmmreg1 xmmreg2
mem to xmmreg	01100110:00001111:11111011: mod xmmreg r/m
PSUB - Subtract With Wrap-around	
xmmreg2 from xmmreg1	01100110:0000 1111:111110gg: 11 xmmreg1 xmmreg2
memory from xmmreg	01100110:0000 1111:111110gg: mod xmmreg r/m
PSUBS - Subtract Signed With Saturation	
xmmreg2 from xmmreg1	01100110:0000 1111:111010gg: 11 xmmreg1 xmmreg2
memory from xmmreg	01100110:0000 1111:111010gg: mod xmmreg r/m

Table B-21. Formats and Encodings of SSE2 Integer Instructions (Contd.)

Instruction and Format	Encoding
PSUBUS - Subtract Unsigned With Saturation	
xmmreg2 from xmmreg1	0000 1111:110110gg: 11 xmmreg1 xmmreg2
memory from xmmreg	0000 1111:110110gg: mod xmmreg r/m
PUNPCKH—Unpack High Data To Next Larger Type	
xmmreg to xmmreg	01100110:00001111:011010gg:11 xmmreg1 Xmmreg2
mem to xmmreg	01100110:00001111:011010gg: mod xmmreg r/m
PUNPCKHQDQ—Unpack High Data	
xmmreg to xmmreg	01100110:00001111:01101101:11 xmmreg1 xmmreg2
mem to xmmreg	01100110:00001111:01101101: mod xmmreg r/m
PUNPCKL—Unpack Low Data To Next Larger Type	
xmmreg to xmmreg	01100110:00001111:011000gg:11 xmmreg1 xmmreg2
mem to xmmreg	01100110:00001111:011000gg: mod xmmreg r/m
PUNPCKLQDQ—Unpack Low Data	
xmmreg to xmmreg	01100110:00001111:01101100:11 xmmreg1 xmmreg2
mem to xmmreg	01100110:00001111:01101100: mod xmmreg r/m
PXOR - Bitwise Xor	
xmmreg2 to xmmreg1	01100110:0000 1111:11101111: 11 xmmreg1 xmmreg2
memory to xmmreg	01100110:0000 1111:11101111: mod xmmreg r/m

Table B-22. Format and Encoding of SSE2 Cacheability Instructions

Instruction and Format	Encoding
MASKMOVDQU—Store Selected Bytes of Double Quadword	
xmmreg to xmmreg	01100110:00001111:11110111:11 xmmreg1 xmmreg2
CLFLUSH—Flush Cache Line	
mem	00001111:10101110:mod r/m
MOVNTPD—Store Packed Double-Precision Floating-Point Values Using Non-Temporal Hint	
xmmreg to mem	01100110:00001111:00101011: mod xmmreg r/m
MOVNTDQ—Store Double Quadword Using Non-Temporal Hint	
xmmreg to mem	01100110:00001111:11100111: mod xmmreg r/m

Table B-22. Format and Encoding of SSE2 Cacheability Instructions (Contd.)

Instruction and Format	Encoding
MOVNTI—Store Doubleword Using Non-Temporal Hint	
reg to mem	00001111:11000011: mod reg r/m
PAUSE—Spin Loop Hint	11110011:10010000
LFENCE—Load Fence	00001111:10101110: 11 101 000
MFENCE—Memory Fence	00001111:10101110: 11 110 000

B.7.2. SSE3 Formats and Encodings Table

The tables in this section provide Prescott 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.

Table B-23. 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

Table B-23. Formats and Encodings of SSE3 Floating-Point Instructions (Contd.)

Instruction and Format	Encoding
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-24. 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-25. 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	
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

Table B-25. Formats and Encodings for SSE3 Integer and Move Instructions (Contd.)

Instruction and Format	Encoding
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.8. FLOATING-POINT INSTRUCTION FORMATS AND ENCODINGS

Table B-26 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-26. 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-27 shows the formats and encodings of the floating-point instructions.

Table B-27. 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	
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)

Table B-27. Floating-Point Instruction Formats and Encodings (Contd.)

Instruction and Format	Encoding
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	
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	
ST(0) ← 16-bit memory ÷ ST(0)	11011 110 : mod 111 r/m
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	
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	

Table B-27. Floating-Point Instruction Formats and Encodings (Contd.)

Instruction and Format	Encoding
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	
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	
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

Table B-27. Floating-Point Instruction Formats and Encodings (Contd.)

Instruction and Format	Encoding
FLDZ – Load +0.0 into ST(0)	11011 001 : 1110 1110
FMUL – Multiply	
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

Table B-27. Floating-Point Instruction Formats and Encodings (Contd.)

Instruction and Format	Encoding
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
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) × $\log_2$(ST(0))	11011 001 : 1111 0001
FYL2XP1 – ST(1) × $\log_2$(ST(0) + 1.0)	11011 001 : 1111 1001
FWAIT – Wait until FPU Ready	1001 1011

C

Intel C/C++ Compiler Intrinsics and Functional Equivalents

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, and SSE3 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 the *Intel C/C++ Compiler User's Guide With Support for the Streaming SIMD Extensions 2* (Order Number 718195-2001).

Table C-1 presents simple intrinsics and Table C-2 presents composite intrinsics. Some intrinsics are "composites" because they require more than one instruction to implement them.

Intel C/C++ Compiler intrinsic names reflect the following naming conventions:

_mm_<intrin_op>_<suffix>

where:

<intrin_op>	Indicates the intrinsics basic operation; for example, add for addition and sub for subtraction
<suffix>	Denotes the type of data operated on by the instruction. The first one or two letters of each suffix denotes whether the data is packed (p), extended packed (ep), or scalar (s). The remaining letters denote the type:

s	single-precision floating point
d	double-precision floating point
i128	signed 128-bit integer
i64	signed 64-bit integer
u64	unsigned 64-bit integer
i32	signed 32-bit integer
u32	unsigned 32-bit integer
i16	signed 16-bit integer
u16	unsigned 16-bit integer

i8	signed 8-bit integer
u8	unsigned 8-bit integer

The variable r is generally used for the intrinsic's return value. A number appended to a variable name indicates the element of a packed object. For example, r0 is the lowest word of r.

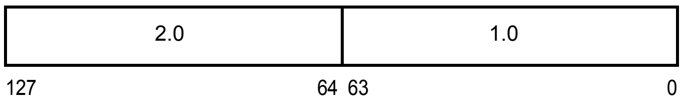
The packed values are represented in right-to-left order, with the lowest value being used for scalar operations. Consider the following example operation:

```
double a[2] = {1.0, 2.0};
__m128d t = _mm_load_pd(a);
```

The result is the same as either of the following:

```
__m128d t = _mm_set_pd(2.0, 1.0);
__m128d t = _mm_setr_pd(1.0, 2.0);
```

In other words, the XMM register that holds the value t will look as follows:



The “scalar” element is 1.0.   Due to the nature of the instruction, some intrinsics require their arguments to be immediates (constant integer literals).

To use an intrinsic in your code, insert a line with the following syntax:

```
data_type intrinsic_name (parameters)
```

Where:

data_type	Is the return data type, which can be either void, int, __m64, __m128, __m128d, or __m128i. Only the _mm_empty intrinsic returns void.
intrinsic_name	Is the name of the intrinsic, which behaves like a function that you can use in your C/C++ code instead of in-lining the actual instruction.
parameters	Represents the parameters required by each intrinsic.

C.1. SIMPLE INTRINSICS

Table C-1. Simple Intrinsics

Mnemonic	Intrinsic	Description
ADDPD	__m128d _mm_add_pd(__m128d a, __m128d b)	Adds the two DP FP (double-precision, floating-point) values of a and b.
ADDPS	__m128 _mm_add_ps(__m128 a, __m128 b)	Adds the four SP FP (single-precision, floating-point) values of a and b.
ADDSD	__m128d _mm_add_sd(__m128d a, __m128d b)	Adds the lower DP FP values of a and b; the upper three DP FP values are passed through from a.
ADDSS	__m128 _mm_add_ss(__m128 a, __m128 b)	Adds the lower SP FP values of a and b; the upper three SP FP values are passed through from a.
ADDSUBPD	__m128d _mm_addsub_pd(__m128d a, __m128d b)	Add/Subtract packed DP FP numbers from XMM2/Mem to XMM1.
ADDSUBPS	__m128 _mm_addsub_ps(__m128 a, __m128 b)	Add/Subtract packed SP FP numbers from XMM2/Mem to XMM1.
ANDNPD	__m128d _mm_andnot_pd(__m128d a, __m128d b)	Computes the bitwise AND-NOT of the two DP FP values of a and b.
ANDNPS	__m128 _mm_andnot_ps(__m128 a, __m128 b)	Computes the bitwise AND-NOT of the four SP FP values of a and b.
ANDPD	__m128d _mm_and_pd(__m128d a, __m128d b)	Computes the bitwise AND of the two DP FP values of a and b.
ANDPS	__m128 _mm_and_ps(__m128 a, __m128 b)	Computes the bitwise AND of the four SP FP values of a and b.
CLFLUSH	void _mm_clflush(void const *p)	Cache line containing p is flushed and invalidated from all caches in the coherency domain.
CMPPD	__m128d _mm_cmpeq_pd(__m128d a, __m128d b)	Compare for equality.
	__m128d _mm_cmplt_pd(__m128d a, __m128d b)	Compare for less-than.
	__m128d _mm_cmple_pd(__m128d a, __m128d b)	Compare for less-than-or-equal.
	__m128d _mm_cmpgt_pd(__m128d a, __m128d b)	Compare for greater-than.

Table C-1. Simple Intrinsics (Contd.)

Mnemonic	Intrinsic	Description
	__m128d _mm_cmpge_pd(__m128d a, __m128d b)	Compare for greater-than-or-equal.
	__m128d _mm_cmpneq_pd(__m128d a, __m128d b)	Compare for inequality.
	__m128d _mm_cmpnlt_pd(__m128d a, __m128d b)	Compare for not-less-than.
	__m128d _mm_cmpngt_pd(__m128d a, __m128d b)	Compare for not-greater-than.
	__m128d _mm_cmpnge_pd(__m128d a, __m128d b)	Compare for not-greater-than-or-equal.
	__m128d _mm_cmpord_pd(__m128d a, __m128d b)	Compare for ordered.
	__m128d _mm_cmpunord_pd(__m128d a, __m128d b)	Compare for unordered.
	__m128d _mm_cmpnle_pd(__m128d a, __m128d b)	Compare for not-less-than-or-equal.
CMPPS	__m128 _mm_cmpeq_ps(__m128 a, __m128 b)	Compare for equality.
	__m128 _mm_cmplt_ps(__m128 a, __m128 b)	Compare for less-than.
	__m128 _mm_cmple_ps(__m128 a, __m128 b)	Compare for less-than-or-equal.
	__m128 _mm_cmpgt_ps(__m128 a, __m128 b)	Compare for greater-than.
	__m128 _mm_cmpge_ps(__m128 a, __m128 b)	Compare for greater-than-or-equal.
	__m128 _mm_cmpneq_ps(__m128 a, __m128 b)	Compare for inequality.
	__m128 _mm_cmpnlt_ps(__m128 a, __m128 b)	Compare for not-less-than.
	__m128 _mm_cmpngt_ps(__m128 a, __m128 b)	Compare for not-greater-than.
	__m128 _mm_cmpnge_ps(__m128 a, __m128 b)	Compare for not-greater-than-or-equal.
	__m128 _mm_cmpord_ps(__m128 a, __m128 b)	Compare for ordered.
	__m128 _mm_cmpunord_ps(__m128 a, __m128 b)	Compare for unordered.
	__m128 _mm_cmpnle_ps(__m128 a, __m128 b)	Compare for not-less-than-or-equal.
CMPSD	__m128d _mm_cmpeq_sd(__m128d a, __m128d b)	Compare for equality.
	__m128d _mm_cmplt_sd(__m128d a, __m128d b)	Compare for less-than.
	__m128d _mm_cmple_sd(__m128d a, __m128d b)	Compare for less-than-or-equal.
	__m128d _mm_cmpgt_sd(__m128d a, __m128d b)	Compare for greater-than.
	__m128d _mm_cmpge_sd(__m128d a, __m128d b)	Compare for greater-than-or-equal.
	__m128 _mm_cmpneq_sd(__m128d a, __m128d b)	Compare for inequality.
	__m128 _mm_cmpnlt_sd(__m128d a, __m128d b)	Compare for not-less-than.

Table C-1. Simple Intrinsics (Contd.)

Mnemonic	Intrinsic	Description
	__m128d _mm_cmpnle_sd(__m128d a, __m128d b)	Compare for not-greater-than.
	__m128d _mm_cmpngt_sd(__m128d a, __m128d b)	Compare for not-greater-than-or-equal.
	__m128d _mm_cmpnge_sd(__m128d a, __m128d b)	Compare for ordered.
	__m128d _mm_cmpord_sd(__m128d a, __m128d b)	Compare for unordered.
	__m128d _mm_cmpunord_sd(__m128d a, __m128d b)	Compare for not-less-than-or-equal.
CMPSS	__m128 _mm_cmpeq_ss(__m128 a, __m128 b)	Compare for equality.
	__m128 _mm_cmplt_ss(__m128 a, __m128 b)	Compare for less-than.
	__m128 _mm_cmple_ss(__m128 a, __m128 b)	Compare for less-than-or-equal.
	__m128 _mm_cmpgt_ss(__m128 a, __m128 b)	Compare for greater-than.
	__m128 _mm_cmpge_ss(__m128 a, __m128 b)	Compare for greater-than-or-equal.
	__m128 _mm_cmpneq_ss(__m128 a, __m128 b)	Compare for inequality.
	__m128 _mm_cmpnlt_ss(__m128 a, __m128 b)	Compare for not-less-than.
	__m128 _mm_cmpnle_ss(__m128 a, __m128 b)	Compare for not-greater-than.
	__m128 _mm_cmpngt_ss(__m128 a, __m128 b)	Compare for not-greater-than-or-equal.
	__m128 _mm_cmpnge_ss(__m128 a, __m128 b)	Compare for ordered.
	__m128 _mm_cmpord_ss(__m128 a, __m128 b)	Compare for unordered.
	__m128 _mm_cmpunord_ss(__m128 a, __m128 b)	Compare for not-less-than-or-equal.
COMISD	int _mm_comieq_sd(__m128d a, __m128d b)	Compares the lower DP FP value of a and b for a equal to b. If a and b are equal, 1 is returned. Otherwise 0 is returned.
	int _mm_comilt_sd(__m128d a, __m128d b)	Compares the lower DP FP value of a and b for a less than b. If a is less than b, 1 is returned. Otherwise 0 is returned.
	int _mm_comile_sd(__m128d a, __m128d b)	Compares the lower DP FP value of a and b for a less than or equal to b. If a is less than or equal to b, 1 is returned. Otherwise 0 is returned.

Table C-1. Simple Intrinsics (Contd.)

Mnemonic	Intrinsic	Description
	int _mm_comigt_sd(__m128d a, __m128d b)	Compares the lower DP FP value of a and b for a greater than b. If a is greater than b are equal, 1 is returned. Otherwise 0 is returned.
	int _mm_comige_sd(__m128d a, __m128d b)	Compares the lower DP FP value of a and b for a greater than or equal to b. If a is greater than or equal to b, 1 is returned. Otherwise 0 is returned.
	int _mm_comineq_sd(__m128d a, __m128d b)	Compares the lower SDP FP value of a and b for a not equal to b. If a and b are not equal, 1 is returned. Otherwise 0 is returned.
COMISS	int _mm_comieq_ss(__m128 a, __m128 b)	Compares the lower SP FP value of a and b for a equal to b. If a and b are equal, 1 is returned. Otherwise 0 is returned.
	int _mm_comilt_ss(__m128 a, __m128 b)	Compares the lower SP FP value of a and b for a less than b. If a is less than b, 1 is returned. Otherwise 0 is returned.
	int _mm_comile_ss(__m128 a, __m128 b)	Compares the lower SP FP value of a and b for a less than or equal to b. If a is less than or equal to b, 1 is returned. Otherwise 0 is returned.
	int _mm_comigt_ss(__m128 a, __m128 b)	Compares the lower SP FP value of a and b for a greater than b. If a is greater than b are equal, 1 is returned. Otherwise 0 is returned.
	int _mm_comige_ss(__m128 a, __m128 b)	Compares the lower SP FP value of a and b for a greater than or equal to b. If a is greater than or equal to b, 1 is returned. Otherwise 0 is returned.
	int _mm_comineq_ss(__m128 a, __m128 b)	Compares the lower SP FP value of a and b for a not equal to b. If a and b are not equal, 1 is returned. Otherwise 0 is returned.

Table C-1. Simple Intrinsics (Contd.)

Mnemonic	Intrinsic	Description
CVTDQ2PD	__m128d _mm_cvtepi32_pd(__m128i a)	Convert the lower two 32-bit signed integer values in packed form in a to two DP FP values.
CVTDQ2PS	__m128 _mm_cvtepi32_ps(__m128i a)	Convert the four 32-bit signed integer values in packed form in a to four SP FP values.
CVTPD2DQ	__m128i _mm_cvtpd_epi32(__m128d a)	Convert the two DP FP values in a to two 32-bit signed integer values.
CVTPD2PI	__m64 _mm_cvtpd_pi32(__m128d a)	Convert the two DP FP values in a to two 32-bit signed integer values.
CVTPD2PS	__m128 _mm_cvtpd_ps(__m128d a)	Convert the two DP FP values in a to two SP FP values.
CVTPI2PD	__m128d _mm_cvtpi32_pd(__m64 a)	Convert the two 32-bit integer values in a to two DP FP values
CVTPI2PS	__m128 _mm_cvt_pi2ps(__m128 a, __m64 b) __m128 _mm_cvtpi32_ps(__m128 a, __m64 b)	Convert the two 32-bit integer values in packed form in b to two SP FP values; the upper two SP FP values are passed through from a.
CVTPS2DQ	__m128i _mm_cvtps_epi32(__m128 a)	Convert four SP FP values in a to four 32-bit signed integers according to the current rounding mode.
CVTPS2PD	__m128d _mm_cvtps_pd(__m128 a)	Convert the lower two SP FP values in a to DP FP values.
CVTPS2PI	__m64 _mm_cvt_ps2pi(__m128 a) __m64 _mm_cvtps_pi32(__m128 a)	Convert the two lower SP FP values of a to two 32-bit integers according to the current rounding mode, returning the integers in packed form.
CVTSD2SI	int _mm_cvtsd_si32(__m128d a)	Convert the lower DP FP value in a to a 32-bit integer value.
CVTSD2SS	__m128 _mm_cvtsd_ss(__m128 a, __m128d b)	Convert the lower DP FP value in b to a SP FP value; the upper three SP FP values of a are passed through.
CVTSI2SD	__m128d _mm_cvtsi32_sd(__m128d a, int b)	Convert the 32-bit integer value b to a DP FP value; the upper DP FP values are passed through from a.

Table C-1. Simple Intrinsics (Contd.)

Mnemonic	Intrinsic	Description
CVTSI2SS	__m128 _mm_cvt_si2ss(__m128 a, int b) __m128 _mm_cvtsi32_ss(__m128a, int b)	Convert the 32-bit integer value b to an SP FP value; the upper three SP FP values are passed through from a.
CVTSS2SD	__m128d _mm_cvtss_sd(__m128d a, __m128 b)	Convert the lower SP FP value of b to DP FP value, the upper DP FP value is passed through from a.
CVTSS2SI	int _mm_cvt_ss2si(__m128 a) int _mm_cvtss_si32(__m128 a)	Convert the lower SP FP value of a to a 32-bit integer.
CVTTPD2DQ	__m128i _mm_cvttpd_epi32(__m128d a)	Convert the two DP FP values of a to two 32-bit signed integer values with truncation, the upper two integer values are 0.
CVTTPD2PI	__m64 _mm_cvttpd_pi32(__m128d a)	Convert the two DP FP values of a to 32-bit signed integer values with truncation.
CVTTPS2DQ	__m128i _mm_cvttps_epi32(__m128 a)	Convert four SP FP values of a to four 32-bit integer with truncation.
CVTTPS2PI	__m64 _mm_cvtt_ps2pi(__m128 a) __m64 _mm_cvttps_pi32(__m128 a)	Convert the two lower SP FP values of a to two 32-bit integer with truncation, returning the integers in packed form.
CVTTSD2SI	int _mm_cvttsd_si32(__m128d a)	Convert the lower DP FP value of a to a 32-bit signed integer using truncation.
CVTTSS2SI	int _mm_cvtt_ss2si(__m128 a) int _mm_cvttss_si32(__m128 a)	Convert the lower SP FP value of a to a 32-bit integer according to the current rounding mode.
	__m64 _mm_cvtsi32_si64(int i)	Convert the integer object i to a 64-bit __m64 object. The integer value is zero extended to 64 bits.
	int _mm_cvtsi64_si32(__m64 m)	Convert the lower 32 bits of the __m64 object m to an integer.
DIVPD	__m128d _mm_div_pd(__m128d a, __m128d b)	Divides the two DP FP values of a and b.
DIVPS	__m128 _mm_div_ps(__m128 a, __m128 b)	Divides the four SP FP values of a and b.

Table C-1. Simple Intrinsics (Contd.)

Mnemonic	Intrinsic	Description
DIVSD	__m128d _mm_div_sd(__m128d a, __m128d b)	Divides the lower DP FP values of a and b; the upper three DP FP values are passed through from a.
DIVSS	__m128 _mm_div_ss(__m128 a, __m128 b)	Divides the lower SP FP values of a and b; the upper three SP FP values are passed through from a.
EMMS	void _mm_empty()	Clears the MMX technology state.
HADDPD	__m128d _mm_hadd_pd(__m128d a, __m128d b)	Add horizontally packed DP FP numbers from XMM2/Mem to XMM1
HADDPS	__m128 _mm_hadd_ps(__m128 a, __m128 b)	Add horizontally packed SP FP numbers from XMM2/Mem to XMM1
HSUBPD	__m128d _mm_hsub_pd(__m128d a, __m128d b)	Subtract horizontally packed DP FP numbers in XMM2/Mem from XMM1.
HSUBPS	__m128 _mm_hsub_ps(__m128 a, __m128 b)	Subtract horizontally packed SP FP numbers in XMM2/Mem from XMM1.
LDDQU	__m128i _mm_lddqu_si128(__m128i const *p)	Load 128 bits from Mem to XMM register.
LDMXCSR	_mm_setcsr(unsigned int i)	Sets the control register to the value specified.
LFENCE	void _mm_lfence(void)	Guaranteed that every load that proceeds, in program order, the load fence instruction is globally visible before any load instruction that follows the fence in program order.
MASKMOVDQU	void _mm_maskmoveu_si128(__m128i d, __m128i n, char *p)	Conditionally store byte elements of d to address p. The high bit of each byte in the selector n determines whether the corresponding byte in d will be stored.
MASKMOVQ	void _mm_maskmove_si64(__m64 d, __m64 n, char *p)	Conditionally store byte elements of d to address p. The high bit of each byte in the selector n determines whether the corresponding byte in d will be stored.

Table C-1. Simple Intrinsics (Contd.)

Mnemonic	Intrinsic	Description
MAXPD	__m128d _mm_max_pd(__m128d a, __m128d b)	Computes the maximums of the two DP FP values of a and b.
MAXPS	__m128 _mm_max_ps(__m128 a, __m128 b)	Computes the maximums of the four SP FP values of a and b.
MAXSD	__m128d _mm_max_sd(__m128d a, __m128d b)	Computes the maximum of the lower DP FP values of a and b; the upper DP FP values are passed through from a.
MAXSS	__m128 _mm_max_ss(__m128 a, __m128 b)	Computes the maximum of the lower SP FP values of a and b; the upper three SP FP values are passed through from a.
MFENCE	void _mm_mfence(void)	Guaranteed that every memory access that proceeds, in program order, the memory fence instruction is globally visible before any memory instruction that follows the fence in program order.
MINPD	__m128d _mm_min_pd(__m128d a, __m128d b)	Computes the minimums of the two DP FP values of a and b.
MINPS	__m128 _mm_min_ps(__m128 a, __m128 b)	Computes the minimums of the four SP FP values of a and b.
MINSD	__m128d _mm_min_sd(__m128d a, __m128d b)	Computes the minimum of the lower DP FP values of a and b; the upper DP FP values are passed through from a.
MINSS	__m128 _mm_min_ss(__m128 a, __m128 b)	Computes the minimum of the lower SP FP values of a and b; the upper three SP FP values are passed through from a.
MONITOR	void _mm_monitor(void const *p, unsigned extensions, unsigned hints)	Sets up a linear address range to be monitored by hardware and activates the monitor. The address range should be of a write-back memory caching type.
MOVAPD	__m128d _mm_load_pd(double * p)	Loads two DP FP values. The address p must be 16-byte-aligned.

Table C-1. Simple Intrinsics (Contd.)

Mnemonic	Intrinsic	Description
	void_mm_store_pd(double *p, __m128d a)	Stores two DP FP values to address p. The address p must be 16-byte-aligned.
MOVAPS	__m128 _mm_load_ps(float * p)	Loads four SP FP values. The address p must be 16-byte-aligned.
	void_mm_store_ps(float *p, __m128 a)	Stores four SP FP values. The address p must be 16-byte-aligned.
MOVD	__m128i _mm_cvtsi32_si128(int a)	Moves 32-bit integer a to the lower 32-bit of the 128-bit destination, while zero-extending he upper bits.
	int _mm_cvtsi128_si32(__m128i a)	Moves lower 32-bit integer of a to a 32-bit signed integer.
	__m64 _mm_cvtsi32_si64(int a)	Moves 32-bit integer a to the lower 32-bit of the 64-bit destination, while zero-extending he upper bits.
	int _mm_cvtsi64_si32(__m64 a)	Moves lower 32-bit integer of a to a 32-bit signed integer.
MOVDDUP	__m128d _mm_movedup_pd(__m128d a) __m128d _mm_loaddup_pd(double const * dp)	Move 64 bits representing the lower DP data element from XMM2/Mem to XMM1 register and duplicate.
MOVDQA	__m128i _mm_load_si128(__m128i * p)	Loads 128-bit values from p. The address p must be 16-byte-aligned.
	void_mm_store_si128(__m128i *p, __m128i a)	Stores 128-bit value in a to address p. The address p must be 16-byte-aligned.
MOVDQU	__m128i _mm_loadu_si128(__m128i * p)	Loads 128-bit values from p. The address p need not be 16-byte-aligned.
	void_mm_storeu_si128(__m128i *p, __m128i a)	Stores 128-bit value in a to address p. The address p need not be 16-byte-aligned.
MOVDQ2Q	__m64 _mm_movepi64_pi64(__m128i a)	Return the lower 64-bits in a as __m64 type.
MOVHLPS	__m128 _mm_movehl_ps(__m128 a, __m128 b)	Moves the upper 2 SP FP values of b to the lower 2 SP FP values of the result. The upper 2 SP FP values of a are passed through to the result.

Table C-1. Simple Intrinsics (Contd.)

Mnemonic	Intrinsic	Description
MOVHPD	__m128d _mm_loadh_pd(__m128d a, double * p)	Load a DP FP value from the address p to the upper 64 bits of destination; the lower 64 bits are passed through from a.
	void _mm_storeh_pd(double * p, __m128d a)	Stores the upper DP FP value of a to the address p.
MOVHPS	__m128 _mm_loadh_pi(__m128 a, __m64 * p)	Sets the upper two SP FP values with 64 bits of data loaded from the address p; the lower two values are passed through from a.
	void _mm_storeh_pi(__m64 * p, __m128 a)	Stores the upper two SP FP values of a to the address p.
MOVLPD	__m128d _mm_loadl_pd(__m128d a, double * p)	Load a DP FP value from the address p to the lower 64 bits of destination; the upper 64 bits are passed through from a.
	void _mm_storel_pd(double * p, __m128d a)	Stores the lower DP FP value of a to the address p.
MOVLPS	__m128 _mm_loadl_pi(__m128 a, __m64 *p)	Sets the lower two SP FP values with 64 bits of data loaded from the address p; the upper two values are passed through from a.
	void_mm_storel_pi(__m64 * p, __m128 a)	Stores the lower two SP FP values of a to the address p.
MOVLHPS	__m128 _mm_movelh_ps(__m128 a, __m128 b)	Moves the lower 2 SP FP values of b to the upper 2 SP FP values of the result. The lower 2 SP FP values of a are passed through to the result.
MOVMSKPD	int _mm_movemask_pd(__m128d a)	Creates a 2-bit mask from the sign bits of the two DP FP values of a.
MOVMSKPS	int _mm_movemask_ps(__m128 a)	Creates a 4-bit mask from the most significant bits of the four SP FP values.
MOVNTDQ	void_mm_stream_si128(__m128i * p, __m128i a)	Stores the data in a to the address p without polluting the caches. If the cache line containing p is already in the cache, the cache will be updated. The address must be 16-byte-aligned.

Table C-1. Simple Intrinsics (Contd.)

Mnemonic	Intrinsic	Description
MOVNTPD	void_mm_stream_pd(double * p, __m128d a)	Stores the data in a to the address p without polluting the caches. The address must be 16-byte-aligned.
MOVNTPS	void_mm_stream_ps(float * p, __m128 a)	Stores the data in a to the address p without polluting the caches. The address must be 16-byte-aligned.
MOVNTI	void_mm_stream_si32(int * p, int a)	Stores the data in a to the address p without polluting the caches.
MOVNTQ	void_mm_stream_pi(__m64 * p, __m64 a)	Stores the data in a to the address p without polluting the caches.
MOVQ	__m128i _mm_loadl_epi64(__m128i * p)	Loads the lower 64 bits from p into the lower 64 bits of destination and zero-extend the upper 64 bits.
	void_mm_storel_epi64(_m128i * p, __m128i a)	Stores the lower 64 bits of a to the lower 64 bits at p.
	__m128i _mm_move_epi64(__m128i a)	Moves the lower 64 bits of a to the lower 64 bits of destination. The upper 64 bits are cleared.
MOVQ2DQ	__m128i _mm_movpi64_epi64(__m64 a)	Move the 64 bits of a into the lower 64-bits, while zero-extending the upper bits.
MOVSD	__m128d _mm_load_sd(double * p)	Loads a DP FP value from p into the lower DP FP value and clears the upper DP FP value. The address P need not be 16-byte aligned.
	void_mm_store_sd(double * p, __m128d a)	Stores the lower DP FP value of a to address p. The address P need not be 16-byte aligned.
	__m128d _mm_move_sd(__m128d a, __m128d b)	Sets the lower DP FP values of b to destination. The upper DP FP value is passed through from a.
MOVSHDUP	__m128 _mm_movehdup_ps(__m128 a)	Move 128 bits representing packed SP data elements from XMM2/Mem to XMM1 register and duplicate high.

Table C-1. Simple Intrinsics (Contd.)

Mnemonic	Intrinsic	Description
MOVSLDUP	__m128 _mm_moveldup_ps(__m128 a)	Move 128 bits representing packed SP data elements from XMM2/Mem to XMM1 register and duplicate low.
MOVSS	__m128 _mm_load_ss(float * p)	Loads an SP FP value into the low word and clears the upper three words.
	void_mm_store_ss(float * p, __m128 a)	Stores the lower SP FP value.
	__m128 _mm_move_ss(__m128 a, __m128 b)	Sets the low word to the SP FP value of b. The upper 3 SP FP values are passed through from a.
MOVUPD	__m128d _mm_loadu_pd(double * p)	Loads two DP FP values from p. The address p need not be 16-byte-aligned.
	void_mm_storeu_pd(double *p, __m128d a)	Stores two DP FP values in a to p. The address p need not be 16-byte-aligned.
MOVUPS	__m128 _mm_loadu_ps(float * p)	Loads four SP FP values. The address need not be 16-byte-aligned.
	void_mm_storeu_ps(float *p, __m128 a)	Stores four SP FP values. The address need not be 16-byte-aligned.
MULPD	__m128d _mm_mul_pd(__m128d a, __m128d b)	Multiplies the two DP FP values of a and b.
MULPS	__m128 _mm_mul_ss(__m128 a, __m128 b)	Multiplies the four SP FP value of a and b.
MULSD	__m128d _mm_mul_sd(__m128d a, __m128d b)	Multiplies the lower DP FP value of a and b; the upper DP FP value are passed through from a.
MULSS	__m128 _mm_mul_ss(__m128 a, __m128 b)	Multiplies the lower SP FP value of a and b; the upper three SP FP values are passed through from a.
MWAIT	void _mm_mwait(unsigned extensions, unsigned hints)	A hint that allows the processor to stop instruction execution and enter an implementation-dependent optimized state until occurrence of a class of events.
ORPD	__m128d _mm_or_pd(__m128d a, __m128d b)	Computes the bitwise OR of the two DP FP values of a and b.

Table C-1. Simple Intrinsics (Contd.)

Mnemonic	Intrinsic	Description
ORPS	__m128 _mm_or_ps(__m128 a, __m128 b)	Computes the bitwise OR of the four SP FP values of a and b.
PACKSSWB	__m128i _mm_packs_epi16(__m128i m1, __m128i m2)	Pack the eight 16-bit values from m1 into the lower eight 8-bit values of the result with signed saturation, and pack the eight 16-bit values from m2 into the upper eight 8-bit values of the result with signed saturation.
PACKSSWB	__m64 _mm_packs_pi16(__m64 m1, __m64 m2)	Pack the four 16-bit values from m1 into the lower four 8-bit values of the result with signed saturation, and pack the four 16-bit values from m2 into the upper four 8-bit values of the result with signed saturation.
PACKSSDW	__m128i _mm_packs_epi32 (__m128i m1, __m128i m2)	Pack the four 32-bit values from m1 into the lower four 16-bit values of the result with signed saturation, and pack the four 32-bit values from m2 into the upper four 16-bit values of the result with signed saturation.
PACKSSDW	__m64 _mm_packs_pi32 (__m64 m1, __m64 m2)	Pack the two 32-bit values from m1 into the lower two 16-bit values of the result with signed saturation, and pack the two 32-bit values from m2 into the upper two 16-bit values of the result with signed saturation.
PACKUSWB	__m128i _mm_packus_epi16(__m128i m1, __m128i m2)	Pack the eight 16-bit values from m1 into the lower eight 8-bit values of the result with unsigned saturation, and pack the eight 16-bit values from m2 into the upper eight 8-bit values of the result with unsigned saturation.

Table C-1. Simple Intrinsics (Contd.)

Mnemonic	Intrinsic	Description
PACKUSWB	__m64 _mm_packs_pu16(__m64 m1, __m64 m2)	Pack the four 16-bit values from m1 into the lower four 8-bit values of the result with unsigned saturation, and pack the four 16-bit values from m2 into the upper four 8-bit values of the result with unsigned saturation.
PADDB	__m128i _mm_add_epi8(__m128i m1, __m128i m2)	Add the 16 8-bit values in m1 to the 16 8-bit values in m2.
PADDB	__m64 _mm_add_pi8(__m64 m1, __m64 m2)	Add the eight 8-bit values in m1 to the eight 8-bit values in m2.
PADDW	__m128i _mm_addw_epi16(__m128i m1, __m128i m2)	Add the 8 16-bit values in m1 to the 8 16-bit values in m2.
PADDW	__m64 _mm_addw_pi16(__m64 m1, __m64 m2)	Add the four 16-bit values in m1 to the four 16-bit values in m2.
PADDD	__m128i _mm_add_epi32(__m128i m1, __m128i m2)	Add the 4 32-bit values in m1 to the 4 32-bit values in m2.
PADDD	__m64 _mm_add_pi32(__m64 m1, __m64 m2)	Add the two 32-bit values in m1 to the two 32-bit values in m2.
PADDQ	__m128i _mm_add_epi64(__m128i m1, __m128i m2)	Add the 2 64-bit values in m1 to the 2 64-bit values in m2.
PADDQ	__m64 _mm_add_si64(__m64 m1, __m64 m2)	Add the 64-bit value in m1 to the 64-bit value in m2.
PADDSB	__m128i _mm_adds_epi8(__m128i m1, __m128i m2)	Add the 16 signed 8-bit values in m1 to the 16 signed 8-bit values in m2 and saturate.
PADDSB	__m64 _mm_adds_pi8(__m64 m1, __m64 m2)	Add the eight signed 8-bit values in m1 to the eight signed 8-bit values in m2 and saturate.
PADDSW	__m128i _mm_adds_epi16(__m128i m1, __m128i m2)	Add the 8 signed 16-bit values in m1 to the 8 signed 16-bit values in m2 and saturate.
PADDSW	__m64 _mm_adds_pi16(__m64 m1, __m64 m2)	Add the four signed 16-bit values in m1 to the four signed 16-bit values in m2 and saturate.

Table C-1. Simple Intrinsics (Contd.)

Mnemonic	Intrinsic	Description
PADDUSB	__m128i _mm_adds_epu8(__m128i m1, __m128i m2)	Add the 16 unsigned 8-bit values in m1 to the 16 unsigned 8-bit values in m2 and saturate.
PADDUSB	__m64 _mm_adds_pu8(__m64 m1, __m64 m2)	Add the eight unsigned 8-bit values in m1 to the eight unsigned 8-bit values in m2 and saturate.
PADDUSW	__m128i _mm_adds_epu16(__m128i m1, __m128i m2)	Add the 8 unsigned 16-bit values in m1 to the 8 unsigned 16-bit values in m2 and saturate.
PADDUSW	__m64 _mm_adds_pu16(__m64 m1, __m64 m2)	Add the four unsigned 16-bit values in m1 to the four unsigned 16-bit values in m2 and saturate.
PAND	__m128i _mm_and_si128(__m128i m1, __m128i m2)	Perform a bitwise AND of the 128-bit value in m1 with the 128-bit value in m2.
PAND	__m64 _mm_and_si64(__m64 m1, __m64 m2)	Perform a bitwise AND of the 64-bit value in m1 with the 64-bit value in m2.
PANDN	__m128i _mm_andnot_si128(__m128i m1, __m128i m2)	Perform a logical NOT on the 128-bit value in m1 and use the result in a bitwise AND with the 128-bit value in m2.
PANDN	__m64 _mm_andnot_si64(__m64 m1, __m64 m2)	Perform a logical NOT on the 64-bit value in m1 and use the result in a bitwise AND with the 64-bit value in m2.
PAUSE	void _mm_pause(void)	The execution of the next instruction is delayed by an implementation-specific amount of time. No architectural state is modified.
PAVGB	__m128i _mm_avg_epu8(__m128i a, __m128i b)	Perform the packed average on the 16 8-bit values of the two operands.
PAVGB	__m64 _mm_avg_pu8(__m64 a, __m64 b)	Perform the packed average on the eight 8-bit values of the two operands.
PAVGW	__m128i _mm_avg_epu16(__m128i a, __m128i b)	Perform the packed average on the 8 16-bit values of the two operands.

Table C-1. Simple Intrinsics (Contd.)

Mnemonic	Intrinsic	Description
PAVGW	__m64 _mm_avg_pu16(__m64 a, __m64 b)	Perform the packed average on the four 16-bit values of the two operands.
PCMPEQB	__m128i _mm_cmpeq_epi8(__m128i m1, __m128i m2)	If the respective 8-bit values in m1 are equal to the respective 8-bit values in m2 set the respective 8-bit resulting values to all ones, otherwise set them to all zeroes.
PCMPEQB	__m64 _mm_cmpeq_pi8(__m64 m1, __m64 m2)	If the respective 8-bit values in m1 are equal to the respective 8-bit values in m2 set the respective 8-bit resulting values to all ones, otherwise set them to all zeroes.
PCMPEQW	__m128i _mm_cmpeq_epi16 (__m128i m1, __m128i m2)	If the respective 16-bit values in m1 are equal to the respective 16-bit values in m2 set the respective 16-bit resulting values to all ones, otherwise set them to all zeroes.
PCMPEQW	__m64 _mm_cmpeq_pi16 (__m64 m1, __m64 m2)	If the respective 16-bit values in m1 are equal to the respective 16-bit values in m2 set the respective 16-bit resulting values to all ones, otherwise set them to all zeroes.
PCMPEQD	__m128i _mm_cmpeq_epi32(__m128i m1, __m128i m2)	If the respective 32-bit values in m1 are equal to the respective 32-bit values in m2 set the respective 32-bit resulting values to all ones, otherwise set them to all zeroes.
PCMPEQD	__m64 _mm_cmpeq_pi32(__m64 m1, __m64 m2)	If the respective 32-bit values in m1 are equal to the respective 32-bit values in m2 set the respective 32-bit resulting values to all ones, otherwise set them to all zeroes.

Table C-1. Simple Intrinsics (Contd.)

Mnemonic	Intrinsic	Description
PCMPGTB	__m128i _mm_cmpgt_epi8 (__m128i m1, __m128i m2)	If the respective 8-bit values in m1 are greater than the respective 8-bit values in m2 set the respective 8-bit resulting values to all ones, otherwise set them to all zeroes.
PCMPGTB	__m64 _mm_cmpgt_pi8 (__m64 m1, __m64 m2)	If the respective 8-bit values in m1 are greater than the respective 8-bit values in m2 set the respective 8-bit resulting values to all ones, otherwise set them to all zeroes.
PCMPGTW	__m128i _mm_cmpgt_epi16(__m128i m1, __m128i m2)	If the respective 16-bit values in m1 are greater than the respective 16-bit values in m2 set the respective 16-bit resulting values to all ones, otherwise set them to all zeroes.
PCMPGTW	__m64 _mm_cmpgt_pi16 (__m64 m1, __m64 m2)	If the respective 16-bit values in m1 are greater than the respective 16-bit values in m2 set the respective 16-bit resulting values to all ones, otherwise set them to all zeroes.
PCMPGTD	__m128i _mm_cmpgt_epi32(__m128i m1, __m128i m2)	If the respective 32-bit values in m1 are greater than the respective 32-bit values in m2 set the respective 32-bit resulting values to all ones, otherwise set them all to zeroes.
PCMPGTD	__m64 _mm_cmpgt_pi32(__m64 m1, __m64 m2)	If the respective 32-bit values in m1 are greater than the respective 32-bit values in m2 set the respective 32-bit resulting values to all ones, otherwise set them all to zeroes.
PEXTRW	int _mm_extract_epi16(__m128i a, int n)	Extracts one of the 8 words of a. The selector n must be an immediate.
PEXTRW	int _mm_extract_pi16(__m64 a, int n)	Extracts one of the four words of a. The selector n must be an immediate.

Table C-1. Simple Intrinsics (Contd.)

Mnemonic	Intrinsic	Description
PINSRW	__m128i _mm_insert_epi16(__m128i a, int d, int n)	Inserts word d into one of 8 words of a. The selector n must be an immediate.
PINSRW	__m64 _mm_insert_pi16(__m64 a, int d, int n)	Inserts word d into one of four words of a. The selector n must be an immediate.
PMADDWD	__m128i _mm_madd_epi16(__m128i m1 __m128i m2)	Multiply 8 16-bit values in m1 by 8 16-bit values in m2 producing 8 32-bit intermediate results, which are then summed by pairs to produce 4 32-bit results.
PMADDWD	__m64 _mm_madd_pi16(__m64 m1, __m64 m2)	Multiply four 16-bit values in m1 by four 16-bit values in m2 producing four 32-bit intermediate results, which are then summed by pairs to produce two 32-bit results.
PMAXSW	__m128i _mm_max_epi16(__m128i a, __m128i b)	Computes the element-wise maximum of the 16-bit integers in a and b.
PMAXSW	__m64 _mm_max_pi16(__m64 a, __m64 b)	Computes the element-wise maximum of the words in a and b.
PMAXUB	__m128i _mm_max_epu8(__m128i a, __m128i b)	Computes the element-wise maximum of the unsigned bytes in a and b.
PMAXUB	__m64 _mm_max_pu8(__m64 a, __m64 b)	Computes the element-wise maximum of the unsigned bytes in a and b.
PMINSW	__m128i _mm_min_epi16(__m128i a, __m128i b)	Computes the element-wise minimum of the 16-bit integers in a and b.
PMINSW	__m64 _mm_min_pi16(__m64 a, __m64 b)	Computes the element-wise minimum of the words in a and b.
PMINUB	__m128i _mm_min_epu8(__m128i a, __m128i b)	Computes the element-wise minimum of the unsigned bytes in a and b.
PMINUB	__m64 _mm_min_pu8(__m64 a, __m64 b)	Computes the element-wise minimum of the unsigned bytes in a and b.
PMOVMSKB	int _mm_movemask_epi8(__m128i a)	Creates an 16-bit mask from the most significant bits of the bytes in a.

Table C-1. Simple Intrinsics (Contd.)

Mnemonic	Intrinsic	Description
PMOVMSKB	int _mm_movemask_pi8(__m64 a)	Creates an 8-bit mask from the most significant bits of the bytes in a.
PMULHUW	__m128i _mm_mulhi_epu16(__m128i a, __m128i b)	Multiplies the 8 unsigned words in a and b, returning the upper 16 bits of the eight 32-bit intermediate results in packed form.
PMULHUW	__m64 _mm_mulhi_pu16(__m64 a, __m64 b)	Multiplies the 4 unsigned words in a and b, returning the upper 16 bits of the four 32-bit intermediate results in packed form.
PMULHW	__m128i _mm_mulhi_epi16(__m128i m1, __m128i m2)	Multiply 8 signed 16-bit values in m1 by 8 signed 16-bit values in m2 and produce the high 16 bits of the 8 results.
PMULHW	__m64 _mm_mulhi_pi16(__m64 m1, __m64 m2)	Multiply four signed 16-bit values in m1 by four signed 16-bit values in m2 and produce the high 16 bits of the four results.
PMULLW	__m128i _mm_mullo_epi16(__m128i m1, __m128i m2)	Multiply 8 16-bit values in m1 by 8 16-bit values in m2 and produce the low 16 bits of the 8 results.
PMULLW	__m64 _mm_mullo_pi16(__m64 m1, __m64 m2)	Multiply four 16-bit values in m1 by four 16-bit values in m2 and produce the low 16 bits of the four results.
PMULUDQ	__m64 _mm_mul_su32(__m64 m1, __m64 m2)	Multiply lower 32-bit unsigned value in m1 by the lower 32-bit unsigned value in m2 and store the 64 bit results.
	__m128i _mm_mul_epu32(__m128i m1, __m128i m2)	Multiply lower two 32-bit unsigned value in m1 by the lower two 32-bit unsigned value in m2 and store the two 64 bit results.
POR	__m64 _mm_or_si64(__m64 m1, __m64 m2)	Perform a bitwise OR of the 64-bit value in m1 with the 64-bit value in m2.
POR	__m128i _mm_or_si128(__m128i m1, __m128i m2)	Perform a bitwise OR of the 128-bit value in m1 with the 128-bit value in m2.

Table C-1. Simple Intrinsics (Contd.)

Mnemonic	Intrinsic	Description
PREFETCHh	void _mm_prefetch(char *a, int sel)	Loads one cache line of data from address p to a location “closer” to the processor. The value sel specifies the type of prefetch operation.
PSADBW	__m128i _mm_sad_epu8(__m128i a, __m128i b)	Compute the absolute differences of the16 unsigned 8-bit values of a and b; sum the upper and lower 8 differences and store the two 16-bit result into the upper and lower 64 bit.
PSADBW	__m64 _mm_sad_pu8(__m64 a, __m64 b)	Compute the absolute differences of the 8 unsigned 8-bit values of a and b; sum the 8 differences and store the 16-bit result, the upper 3 words are cleared.
PSHUFD	__m128i _mm_shuffle_epi32(__m128i a, int n)	Returns a combination of the four doublewords of a. The selector n must be an immediate.
PSHUFHW	__m128i _mm_shufflehi_epi16(__m128i a, int n)	Shuffle the upper four 16-bit words in a as specified by n. The selector n must be an immediate.
PSHUFLW	__m128i _mm_shufflelo_epi16(__m128i a, int n)	Shuffle the lower four 16-bit words in a as specified by n. The selector n must be an immediate.
PSHUFW	__m64 _mm_shuffle_pi16(__m64 a, int n)	Returns a combination of the four words of a. The selector n must be an immediate.
PSLLW	__m128i _mm_sll_epi16(__m128i m, __m128i count)	Shift each of 8 16-bit values in m left the amount specified by count while shifting in zeroes.
PSLLW	__m128i _mm_slli_epi16(__m128i m, int count)	Shift each of 8 16-bit values in m left the amount specified by count while shifting in zeroes.
PSLLW	__m64 _mm_sll_pi16(__m64 m, __m64 count)	Shift four 16-bit values in m left the amount specified by count while shifting in zeroes. For the best performance, count should be a constant.

Table C-1. Simple Intrinsics (Contd.)

Mnemonic	Intrinsic	Description
	__m64 _mm_slli_pi16(__m64 m, int count)	Shift four 16-bit values in m left the amount specified by count while shifting in zeroes. For the best performance, count should be a constant.
PSLLD	__m128i _mm_slli_epi32(__m128i m, int count)	Shift each of 4 32-bit values in m left the amount specified by count while shifting in zeroes.
	__m128i _mm_sll_epi32(__m128i m, __m128i count)	Shift each of 4 32-bit values in m left the amount specified by count while shifting in zeroes. For the best performance, count should be a constant.
PSLLD	__m64 _mm_slli_pi32(__m64 m, int count)	Shift two 32-bit values in m left the amount specified by count while shifting in zeroes.
	__m64 _mm_sll_pi32(__m64 m, __m64 count)	Shift two 32-bit values in m left the amount specified by count while shifting in zeroes. For the best performance, count should be a constant.
PSLLQ	__m64 _mm_sll_si64(__m64 m, __m64 count)	Shift the 64-bit value in m left the amount specified by count while shifting in zeroes.
	__m64 _mm_slli_si64(__m64 m, int count)	Shift the 64-bit value in m left the amount specified by count while shifting in zeroes. For the best performance, count should be a constant.
PSLLQ	__m128i _mm_sll_epi64(__m128i m, __m128i count)	Shift each of two 64-bit values in m left by the amount specified by count while shifting in zeroes.
	__m128i _mm_slli_epi64(__m128i m, int count)	Shift each of two 64-bit values in m left by the amount specified by count while shifting in zeroes. For the best performance, count should be a constant.
PSLLDQ	__m128i _mm_slli_si128(__m128i m, int imm)	Shift 128 bit in m left by imm bytes while shifting in zeroes.
PSRAW	__m128i _mm_sra_epi16(__m128i m, __m128i count)	Shift each of 8 16-bit values in m right the amount specified by count while shifting in the sign bit.

Table C-1. Simple Intrinsics (Contd.)

Mnemonic	Intrinsic	Description
	__m128i _mm_srai_epi16(__m128i m, int count)	Shift each of 8 16-bit values in m right the amount specified by count while shifting in the sign bit. For the best performance, count should be a constant.
PSRAW	__m64 _mm_sra_pi16(__m64 m, __m64 count)	Shift four 16-bit values in m right the amount specified by count while shifting in the sign bit.
	__m64 _mm_srai_pi16(__m64 m, int count)	Shift four 16-bit values in m right the amount specified by count while shifting in the sign bit. For the best performance, count should be a constant.
PSRAD	__m128i _mm_sra_epi32 (__m128i m, __m128i count)	Shift each of 4 32-bit values in m right the amount specified by count while shifting in the sign bit.
	__m128i _mm_srai_epi32 (__m128i m, int count)	Shift each of 4 32-bit values in m right the amount specified by count while shifting in the sign bit. For the best performance, count should be a constant.
PSRAD	__m64 _mm_sra_pi32 (__m64 m, __m64 count)	Shift two 32-bit values in m right the amount specified by count while shifting in the sign bit.
	__m64 _mm_srai_pi32 (__m64 m, int count)	Shift two 32-bit values in m right the amount specified by count while shifting in the sign bit. For the best performance, count should be a constant.
PSRLW	_m128i _mm_srl_epi16 (__m128i m, __m128i count)	Shift each of 8 16-bit values in m right the amount specified by count while shifting in zeroes.
	__m128i _mm_srli_epi16 (__m128i m, int count)	Shift each of 8 16-bit values in m right the amount specified by count while shifting in zeroes.
PSRLW	__m64 _mm_srl_pi16 (__m64 m, __m64 count)	Shift four 16-bit values in m right the amount specified by count while shifting in zeroes.

Table C-1. Simple Intrinsics (Contd.)

Mnemonic	Intrinsic	Description
	__m64 _mm_srli_pi16(__m64 m, int count)	Shift four 16-bit values in m right the amount specified by count while shifting in zeroes. For the best performance, count should be a constant.
PSRLD	__m128i _mm_srl_epi32 (__m128i m, __m128i count)	Shift each of 4 32-bit values in m right the amount specified by count while shifting in zeroes.
	__m128i _mm_srli_epi32 (__m128i m, int count)	Shift each of 4 32-bit values in m right the amount specified by count while shifting in zeroes. For the best performance, count should be a constant.
PSRLD	__m64 _mm_srl_pi32 (__m64 m, __m64 count)	Shift two 32-bit values in m right the amount specified by count while shifting in zeroes.
	__m64 _mm_srli_pi32 (__m64 m, int count)	Shift two 32-bit values in m right the amount specified by count while shifting in zeroes. For the best performance, count should be a constant.
PSRLQ	__m128i _mm_srl_epi64 (__m128i m, __m128i count)	Shift the 2 64-bit value in m right the amount specified by count while shifting in zeroes.
	__m128i _mm_srli_epi64 (__m128i m, int count)	Shift the 2 64-bit value in m right the amount specified by count while shifting in zeroes. For the best performance, count should be a constant.
PSRLQ	__m64 _mm_srl_si64 (__m64 m, __m64 count)	Shift the 64-bit value in m right the amount specified by count while shifting in zeroes.
	__m64 _mm_srli_si64 (__m64 m, int count)	Shift the 64-bit value in m right the amount specified by count while shifting in zeroes. For the best performance, count should be a constant.
PSRLDQ	__m128i _mm_srli_si128(__m128i m, int imm)	Shift 128 bit in m right by imm bytes while shifting in zeroes.
PSUBB	__m128i _mm_sub_epi8(__m128i m1, __m128i m2)	Subtract the 16 8-bit values in m2 from the 16 8-bit values in m1.
PSUBB	__m64 _mm_sub_pi8(__m64 m1, __m64 m2)	Subtract the eight 8-bit values in m2 from the eight 8-bit values in m1.

Table C-1. Simple Intrinsics (Contd.)

Mnemonic	Intrinsic	Description
PSUBW	__m128i _mm_sub_epi16(__m128i m1, __m128i m2)	Subtract the 8 16-bit values in m2 from the 8 16-bit values in m1.
PSUBW	__m64 _mm_sub_pi16(__m64 m1, __m64 m2)	Subtract the four 16-bit values in m2 from the four 16-bit values in m1.
PSUBD	__m128i _mm_sub_epi32(__m128i m1, __m128i m2)	Subtract the 4 32-bit values in m2 from the 4 32-bit values in m1.
PSUBD	__m64 _mm_sub_pi32(__m64 m1, __m64 m2)	Subtract the two 32-bit values in m2 from the two 32-bit values in m1.
PSUBQ	__m128i _mm_sub_epi64(__m128i m1, __m128i m2)	Subtract the 2 64-bit values in m2 from the 2 64-bit values in m1.
PSUBQ	__m64 _mm_sub_si64(__m64 m1, __m64 m2)	Subtract the 64-bit values in m2 from the 64-bit values in m1.
PSUBSB	__m128i _mm_subs_epi8(__m128i m1, __m128i m2)	Subtract the 16 signed 8-bit values in m2 from the 16 signed 8-bit values in m1 and saturate.
PSUBSB	__m64 _mm_subs_pi8(__m64 m1, __m64 m2)	Subtract the eight signed 8-bit values in m2 from the eight signed 8-bit values in m1 and saturate.
PSUBSW	__m128i _mm_subs_epi16(__m128i m1, __m128i m2)	Subtract the 8 signed 16-bit values in m2 from the 8 signed 16-bit values in m1 and saturate.
PSUBSW	__m64 _mm_subs_pi16(__m64 m1, __m64 m2)	Subtract the four signed 16-bit values in m2 from the four signed 16-bit values in m1 and saturate.
PSUBUSB	__m128i _mm_sub_epu8(__m128i m1, __m128i m2)	Subtract the 16 unsigned 8-bit values in m2 from the 16 unsigned 8-bit values in m1 and saturate.
PSUBUSB	__m64 _mm_sub_pu8(__m64 m1, __m64 m2)	Subtract the eight unsigned 8-bit values in m2 from the eight unsigned 8-bit values in m1 and saturate.
PSUBUSW	__m128i _mm_sub_epu16(__m128i m1, __m128i m2)	Subtract the 8 unsigned 16-bit values in m2 from the 8 unsigned 16-bit values in m1 and saturate.

Table C-1. Simple Intrinsics (Contd.)

Mnemonic	Intrinsic	Description
PSUBUSW	__m64 _mm_sub_pu16(__m64 m1, __m64 m2)	Subtract the four unsigned 16-bit values in m2 from the four unsigned 16-bit values in m1 and saturate.
PUNPCKHBW	__m64 _mm_unpackhi_pi8(__m64 m1, __m64 m2)	Interleave the four 8-bit values from the high half of m1 with the four values from the high half of m2 and take the least significant element from m1.
PUNPCKHBW	__m128i _mm_unpackhi_epi8(__m128i m1, __m128i m2)	Interleave the 8 8-bit values from the high half of m1 with the 8 values from the high half of m2.
PUNPCKHWD	__m64 _mm_unpackhi_pi16(__m64 m1,__m64 m2)	Interleave the two 16-bit values from the high half of m1 with the two values from the high half of m2 and take the least significant element from m1.
PUNPCKHWD	__m128i _mm_unpackhi_epi16(__m128i m1, __m128i m2)	Interleave the 4 16-bit values from the high half of m1 with the 4 values from the high half of m2.
PUNPCKHDQ	___m64 _mm_unpackhi_pi32(__m64 m1, __m64 m2)	Interleave the 32-bit value from the high half of m1 with the 32-bit value from the high half of m2 and take the least significant element from m1.
PUNPCKHDQ	__m128i _mm_unpackhi_epi32(__m128i m1, __m128i m2)	Interleave two 32-bit value from the high half of m1 with the two 32-bit value from the high half of m2.
PUNPCKHQDQ	__m128i _mm_unpackhi_epi64(__m128i m1, __m128i m2)	Interleave the 64-bit value from the high half of m1 with the 64-bit value from the high half of m2.
PUNPCKLBW	__m64 _mm_unpacklo_pi8 (__m64 m1, __m64 m2)	Interleave the four 8-bit values from the low half of m1 with the four values from the low half of m2 and take the least significant element from m1.
PUNPCKLBW	__m128i _mm_unpacklo_epi8 (__m128i m1, __m128i m2)	Interleave the 8 8-bit values from the low half of m1 with the 8 values from the low half of m2.

Table C-1. Simple Intrinsics (Contd.)

Mnemonic	Intrinsic	Description
PUNPCKLWD	__m64 _mm_unpacklo_pi16(__m64 m1, __m64 m2)	Interleave the two 16-bit values from the low half of m1 with the two values from the low half of m2 and take the least significant element from m1.
PUNPCKLWD	__m128i _mm_unpacklo_epi16(__m128i m1, __m128i m2)	Interleave the 4 16-bit values from the low half of m1 with the 4 values from the low half of m2.
PUNPCKLDQ	__m64 _mm_unpacklo_pi32(__m64 m1, __m64 m2)	Interleave the 32-bit value from the low half of m1 with the 32-bit value from the low half of m2 and take the least significant element from m1.
PUNPCKLDQ	__m128i _mm_unpacklo_epi32(__m128i m1, __m128i m2)	Interleave two 32-bit value from the low half of m1 with the two 32-bit value from the low half of m2.
PUNPCKLQDQ	__m128i _mm_unpacklo_epi64(__m128i m1, __m128i m2)	Interleave the 64-bit value from the low half of m1 with the 64-bit value from the low half of m2.
PXOR	__m64 _mm_xor_si64(__m64 m1, __m64 m2)	Perform a bitwise XOR of the 64-bit value in m1 with the 64-bit value in m2.
PXOR	__m128i _mm_xor_si128(__m128i m1, __m128i m2)	Perform a bitwise XOR of the 128-bit value in m1 with the 128-bit value in m2.
RCPPS	__m128 _mm_rcp_ps(__m128 a)	Computes the approximations of the reciprocals of the four SP FP values of a.
RCPSS	__m128 _mm_rcp_ss(__m128 a)	Computes the approximation of the reciprocal of the lower SP FP value of a; the upper three SP FP values are passed through.
RSQRTPS	__m128 _mm_rsqrt_ps(__m128 a)	Computes the approximations of the reciprocals of the square roots of the four SP FP values of a.
RSQRTSS	__m128 _mm_rsqrt_ss(__m128 a)	Computes the approximation of the reciprocal of the square root of the lower SP FP value of a; the upper three SP FP values are passed through.

Table C-1. Simple Intrinsics (Contd.)

Mnemonic	Intrinsic	Description
SFENCE	void_mm_sfence(void)	Guarantees that every preceding store is globally visible before any subsequent store.
SHUFPD	__m128d _mm_shuffle_pd(__m128d a, __m128d b, unsigned int imm8)	Selects two specific DP FP values from a and b, based on the mask imm8. The mask must be an immediate.
SHUFPS	__m128 _mm_shuffle_ps(__m128 a, __m128 b, unsigned int imm8)	Selects four specific SP FP values from a and b, based on the mask imm8. The mask must be an immediate.
SQRTPD	__m128d _mm_sqrt_pd(__m128d a)	Computes the square roots of the two DP FP values of a.
SQRTPS	__m128 _mm_sqrt_ps(__m128 a)	Computes the square roots of the four SP FP values of a.
SQRTSD	__m128d _mm_sqrt_sd(__m128d a)	Computes the square root of the lower DP FP value of a; the upper DP FP values are passed through.
SQRTSS	__m128 _mm_sqrt_ss(__m128 a)	Computes the square root of the lower SP FP value of a; the upper three SP FP values are passed through.
STMXCSR	_mm_getcsr(void)	Returns the contents of the control register.
SUBPD	__m128d _mm_sub_pd(__m128d a, __m128d b)	Subtracts the two DP FP values of a and b.
SUBPS	__m128 _mm_sub_ps(__m128 a, __m128 b)	Subtracts the four SP FP values of a and b.
SUBSD	__m128d _mm_sub_sd(__m128d a, __m128d b)	Subtracts the lower DP FP values of a and b. The upper DP FP values are passed through from a.
SUBSS	__m128 _mm_sub_ss(__m128 a, __m128 b)	Subtracts the lower SP FP values of a and b. The upper three SP FP values are passed through from a.
UCOMISD	int _mm_ucomieq_sd(__m128d a, __m128d b)	Compares the lower DP FP value of a and b for a equal to b. If a and b are equal, 1 is returned. Otherwise 0 is returned.

Table C-1. Simple Intrinsics (Contd.)

Mnemonic	Intrinsic	Description
	int _mm_ucomilt_sd(__m128d a, __m128d b)	Compares the lower DP FP value of a and b for a less than b. If a is less than b, 1 is returned. Otherwise 0 is returned.
	int _mm_ucomile_sd(__m128d a, __m128d b)	Compares the lower DP FP value of a and b for a less than or equal to b. If a is less than or equal to b, 1 is returned. Otherwise 0 is returned.
	int _mm_ucomigt_sd(__m128d a, __m128d b)	Compares the lower DP FP value of a and b for a greater than b. If a is greater than b are equal, 1 is returned. Otherwise 0 is returned.
	int _mm_ucomige_sd(__m128d a, __m128d b)	Compares the lower DP FP value of a and b for a greater than or equal to b. If a is greater than or equal to b, 1 is returned. Otherwise 0 is returned.
	int _mm_ucomineq_sd(__m128d a, __m128d b)	Compares the lower DP FP value of a and b for a not equal to b. If a and b are not equal, 1 is returned. Otherwise 0 is returned.
UCOMISS	int _mm_ucomieq_ss(__m128 a, __m128 b)	Compares the lower SP FP value of a and b for a equal to b. If a and b are equal, 1 is returned. Otherwise 0 is returned.
	int _mm_ucomilt_ss(__m128 a, __m128 b)	Compares the lower SP FP value of a and b for a less than b. If a is less than b, 1 is returned. Otherwise 0 is returned.
	int _mm_ucomile_ss(__m128 a, __m128 b)	Compares the lower SP FP value of a and b for a less than or equal to b. If a is less than or equal to b, 1 is returned. Otherwise 0 is returned.
	int _mm_ucomigt_ss(__m128 a, __m128 b)	Compares the lower SP FP value of a and b for a greater than b. If a is greater than b are equal, 1 is returned. Otherwise 0 is returned.

Table C-1. Simple Intrinsics (Contd.)

Mnemonic	Intrinsic	Description
	int _mm_ucomige_ss(__m128 a, __m128 b)	Compares the lower SP FP value of a and b for a greater than or equal to b. If a is greater than or equal to b, 1 is returned. Otherwise 0 is returned.
	int _mm_ucomineq_ss(__m128 a, __m128 b)	Compares the lower SP FP value of a and b for a not equal to b. If a and b are not equal, 1 is returned. Otherwise 0 is returned.
UNPCKHPD	__m128d _mm_unpackhi_pd(__m128d a, __m128d b)	Selects and interleaves the upper DP FP values from a and b.
UNPCKHPS	__m128 _mm_unpackhi_ps(__m128 a, __m128 b)	Selects and interleaves the upper two SP FP values from a and b.
UNPCKLPD	__m128d _mm_unpacklo_pd(__m128d a, __m128d b)	Selects and interleaves the lower DP FP values from a and b.
UNPCKLPS	__m128 _mm_unpacklo_ps(__m128 a, __m128 b)	Selects and interleaves the lower two SP FP values from a and b.
XORPD	__m128d _mm_xor_pd(__m128d a, __m128d b)	Computes bitwise EXOR (exclusive-or) of the two DP FP values of a and b.
XORPS	__m128 _mm_xor_ps(__m128 a, __m128 b)	Computes bitwise EXOR (exclusive-or) of the four SP FP values of a and b.

C.2. COMPOSITE INTRINSICS

Table C-2. Composite Intrinsics

Mnemonic	Intrinsic	Description
(composite)	__m128i _mm_set_epi64(__m64 q1, __m64 q0)	Sets the two 64-bit values to the two inputs.
(composite)	__m128i _mm_set_epi32(int i3, int i2, int i1, int i0)	Sets the 4 32-bit values to the 4 inputs.
(composite)	__m128i _mm_set_epi16(short w7,short w6, short w5, short w4, short w3, short w2, short w1,short w0)	Sets the 8 16-bit values to the 8 inputs.
(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)	Sets the 16 8-bit values to the 16 inputs.
(composite)	__m128i _mm_set1_epi64(__m64 q)	Sets the 2 64-bit values to the input.
(composite)	__m128i _mm_set1_epi32(int a)	Sets the 4 32-bit values to the input.
(composite)	__m128i _mm_set1_epi16(short a)	Sets the 8 16-bit values to the input.
(composite)	__m128i _mm_set1_epi8(char a)	Sets the 16 8-bit values to the input.
(composite)	__m128i _mm_setr_epi64(__m64 q1, __m64 q0)	Sets the two 64-bit values to the two inputs in reverse order.
(composite)	__m128i _mm_setr_epi32(int i3, int i2, int i1, int i0)	Sets the 4 32-bit values to the 4 inputs in reverse order.
(composite)	__m128i _mm_setr_epi16(short w7,short w6, short w5, short w4, short w3, short w2, short w, short w0)	Sets the 8 16-bit values to the 8 inputs in reverse order.
(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)	Sets the 16 8-bit values to the 16 inputs in reverse order.
(composite)	__m128i _mm_setzero_si128()	Sets all bits to 0.
(composite)	__m128 _mm_set_ps1(float w) __m128 _mm_set1_ps(float w)	Sets the four SP FP values to w.
(composite)	__m128cmm_set1_pd(double w)	Sets the two DP FP values to w.
(composite)	__m128d _mm_set_sd(double w)	Sets the lower DP FP values to w.
(composite)	__m128d _mm_set_pd(double z, double y)	Sets the two DP FP values to the two inputs.
(composite)	__m128 _mm_set_ps(float z, float y, float x, float w)	Sets the four SP FP values to the four inputs.
(composite)	__m128d _mm_setr_pd(double z, double y)	Sets the two DP FP values to the two inputs in reverse order.

Table C-2. Composite Intrinsics (Contd.)

Mnemonic	Intrinsic	Description
(composite)	__m128 _mm_setr_ps(float z, float y, float x, float w)	Sets the four SP FP values to the four inputs in reverse order.
(composite)	__m128d _mm_setzero_pd(void)	Clears the two DP FP values.
(composite)	__m128 _mm_setzero_ps(void)	Clears the four SP FP values.
MOVSD + shuffle	__m128d _mm_load_pd(double * p) __m128d _mm_load1_pd(double *p)	Loads a single DP FP value, copying it into both DP FP values.
MOVSS + shuffle	__m128 _mm_load_ps1(float * p) __m128 _mm_load1_ps(float *p)	Loads a single SP FP value, copying it into all four words.
MOVAPD + shuffle	__m128d _mm_loadr_pd(double * p)	Loads two DP FP values in reverse order. The address must be 16-byte-aligned.
MOVAPS + shuffle	__m128 _mm_loadr_ps(float * p)	Loads four SP FP values in reverse order. The address must be 16-byte-aligned.
MOVSD + shuffle	void _mm_store1_pd(double *p, __m128d a)	Stores the lower DP FP value across both DP FP values.
MOVSS + shuffle	void _mm_store_ps1(float * p, __m128 a) void _mm_store1_ps(float *p, __m128 a)	Stores the lower SP FP value across four words.
MOVAPD + shuffle	_mm_storer_pd(double * p, __m128d a)	Stores two DP FP values in reverse order. The address must be 16-byte-aligned.
MOVAPS + shuffle	_mm_storer_ps(float * p, __m128 a)	Stores four SP FP values in reverse order. The address must be 16-byte-aligned.

Index

INDEX FOR VOLUME 2A & 2B

INTEL SALES OFFICES

ASIA PACIFIC
Australia
Intel Corp.
Level 2
448 St Kilda Road
Melbourne VIC
3004
Australia
Fax:613-9862 5599

China
Intel Corp.
Rm 709, Shaanxi
Zhongda Int'l Bldg
No.30 Nandajie Street
Xian AX710002
China
Fax:(86 29) 7203356

Intel Corp.
Rm 2710, Metropolian
Tower
68 Zourong Rd
Chongqing CQ
400015
China

Intel Corp.
C1, 15 Flr, Fujian
Oriental Hotel
No. 96 East Street
Fuzhou FJ
350001
China

Intel Corp.
Rm 5803 CITIC Plaza
233 Tianhe Rd
Guangzhou GD
510613
China

Intel Corp.
Rm 1003, Orient Plaza
No. 235 Huayuan Street
Nangang District
Harbin HL
150001
China

Intel Corp.
Rm 1751 World Trade
Center, No 2
Han Zhong Rd
Nanjing JS
210009
China

Intel Corp.
Hua Xin International
Tower
215 Qing Nian St.
ShenYang LN
110015
China

Intel Corp.
Suite 1128 CITIC Plaza
Jinan
150 Luo Yuan St.
Jinan SN
China

Intel Corp.
Suite 412, Holiday Inn
Crowne Plaza
31, Zong Fu Street
Chengdu SU
610041
China
Fax:86-28-6785965

Intel Corp.
Room 0724, White Rose
Hotel
No 750, MinZhu Road
WuChang District
Wuhan UB
430071
China

India
Intel Corp.
Paharpur Business
Centre
21 Nehru Place
New Delhi DH
110019
India

Intel Corp.
Hotel Rang Sharda, 6th
Floor
Bandra Reclamation
Mumbai MH
400050
India
Fax:91-22-6415578

Intel Corp.
DBS Corporate Club
31A Cathedral Garden
Road
Chennai TD
600034
India

Intel Corp.
DBS Corporate Club
2nd Floor, 8 A.A.C. Bose
Road
Calcutta WB
700017
India

Japan
Intel Corp.
Kokusai Bldg 5F, 3-1-1,
Marunouchi
Chiyoda-Ku, Tokyo
1000005
Japan

Intel Corp.
2-4-1 Terauchi
Toyonaka-Shi
Osaka
5600872
Japan

Malaysia
Intel Corp.
Lot 102 1/F Block A
Wisma Semantan
12 Jalan Gelenggang
Damansara Heights
Kuala Lumpur SL
50490
Malaysia

Thailand
Intel Corp.
87 M. Thai Tower, 9th Fl.
All Seasons Place,
Wireless Road
Lumpini, Patumwan
Bangkok
10330
Thailand

Viet Nam
Intel Corp.
Hanoi Tung Shing
Square, Ste #1106
2 Ngo Quyen St
Hoan Kiem District
Hanoi
Viet Nam

EUROPE & AFRICA
Belgium
Intel Corp.
Woluwelaan 158
Diegem
1831
Belgium

Czech Rep
Intel Corp.
Nahorni 14
Brno
61600
Czech Rep

Denmark
Intel Corp.
Soelodden 13
Maaloev
DK2760
Denmark

Germany
Intel Corp.
Sandstrasse 4
Aichner
86551
Germany

Intel Corp.
Dr Weyerstrasse 2
Juelich
52428
Germany

Intel Corp.
Buchenweg 4
Wildberg
72218
Germany

Intel Corp.
Kemnader Strasse 137
Bochum
44797
Germany

Intel Corp.
Klaus-Schaefer Strasse
16-18
Erfstadt NW
50374
Germany

Intel Corp.
Heldmanskamp 37
Lemgo NW
32657
Germany

Italy
Intel Corp Italia Spa
Milanofiori Palazzo E/4
Assago
Milan
20094
Italy
Fax:39-02-57501221

Netherland
Intel Corp.
Strausslaan 31
Heesch
5384CW
Netherland

Poland
Intel Poland
Developments, Inc
Jerozolimskie Business
Park
Jerozolimskie 146c
Warsaw
2305
Poland
Fax:+48-22-570 81 40

Portugal
Intel Corp.
PO Box 20
Alcabideche
2765
Portugal

Spain
Intel Corp.
Calle Rioja, 9
Bajo F Izquierda
Madrid
28042
Spain

South Africa
Intel SA Corporation
Bldg 14, South Wing,
2nd Floor
Uplands, The Woodlands
Western Services Road
Woodmead
2052
Sth Africa
Fax:+27 11 806 4549

Intel Corp.
19 Summit Place,
Halfway House
Cnr 5th and Harry
Galaun Streets
Midrad
1685
Sth Africa

United Kingdom
Intel Corp.
The Manse
Silver Lane
Needingworth CAMBS
PE274SL
UK

Intel Corp.
2 Cameron Close
Long Melford SUFFK
CO109TS
UK

Israel
Intel Corp.
MTM Industrial Center,
P.O.Box 498
Haifa
31000
Israel
Fax:972-4-8655444

LATIN AMERICA & CANADA
Argentina
Intel Corp.
Dock IV - Bldg 3 - Floor 3
Olga Cossentini 240
Buenos Aires
C1107BVA
Argentina

Brazil
Intel Corp.
Rua Carlos Gomez
111/403
Porto Alegre
90480-003
Brazil

Intel Corp.
Av. Dr. Chucri Zaidan
940 - 10th Floor
San Paulo
04583-904
Brazil

Intel Corp.
Av. Rio Branco,
1 - Sala 1804
Rio de Janeiro
20090-003
Brazil

Columbia
Intel Corp.
Carrera 7 No. 71021
Torre B, Oficina 603
Santefe de Bogota
Columbia

Mexico
Intel Corp.
Av. Mexico No. 2798-9B,
S.H.
Guadalajara
44680
Mexico

Intel Corp.
Torre Esmeralda II,
7th Floor
Blvd. Manuel Avila
Comacho #36
Mexico Cith DF
11000
Mexico

Intel Corp.
Piso 19, Suite 4
Av. Batallon de San
Patricio No 111
Monterrey, Nuevo le
66269
Mexico

Canada
Intel Corp.
168 Bonis Ave, Suite 202
Scarborough
MIT3V6
Canada
Fax:416-335-7695

Intel Corp.
3901 Highway #7,
Suite 403
Vaughan
L4L 8L5
Canada
Fax:905-856-8868

Intel Corp.
999 CANADA PLACE,
Suite 404,#11
Vancouver BC
V6C 3E2
Canada
Fax:604-844-2813

Intel Corp.
2650 Queensview Drive,
Suite 250
Ottawa ON
K2B 8H6
Canada
Fax:613-820-5936

Intel Corp.
190 Attwell Drive,
Suite 500
Rexcdale ON
M9W 6H8
Canada
Fax:416-675-2438

Intel Corp.
171 St. Clair Ave. E,
Suite 6
Toronto ON
Canada

Intel Corp.
1033 Oak Meadow Road
Oakville ON
L6M 1J6
Canada

USA
California
Intel Corp.
551 Lundy Place
Milpitas CA
95035-6833
USA
Fax:408-451-8266

Intel Corp.
1551 N. Tustin Avenue,
Suite 800
Santa Ana CA
92705
USA
Fax:714-541-9157

Intel Corp.
Executive Center del Mar
12230 El Camino Real
Suite 140
San Diego CA
92130
USA
Fax:858-794-5805

Intel Corp.
1960 E. Grand Avenue,
Suite 150
El Segundo CA
90245
USA
Fax:310-640-7133

Intel Corp.
23120 Alicia Parkway,
Suite 215
Mission Viejo CA
92692
USA
Fax:949-586-9499

Intel Corp.
30851 Agoura Road
Suite 202
Agoura Hills CA
91301
USA
Fax:818-874-1166

Intel Corp.
28202 Cabot Road,
Suite #363 & #371
Laguna Niguel CA
92677
USA

Intel Corp.
657 S Cendros Avenue
Solana Beach CA
90075
USA

Intel Corp.
43769 Abeloe Terrace
Fremont CA
94539
USA

Intel Corp.
1721 Warburton, #6
Santa Clara CA
95050
USA

Colorado
Intel Corp.
600 S. Cherry Street,
Suite 700
Denver CO
80222
USA
Fax:303-322-8670

Connecticut
Intel Corp.
Lee Farm Corporate Pk
83 Wooster Heights
Road
Danbury CT
6810
USA
Fax:203-778-2168

Florida
Intel Corp.
7777 Glades Road
Suite 310B
Boca Raton FL
33434
USA
Fax:813-367-5452

Georgia
Intel Corp.
20 Technology Park,
Suite 150
Norcross GA
30092
USA
Fax:770-448-0875

Intel Corp.
Three Northwinds Center
2500 Northwinds
Parkway, 4th Floor
Alpharetta GA
30092
USA
Fax:770-663-6354

Idaho
Intel Corp.
910 W. Main Street, Suite
236
Boise ID
83702
USA
Fax:208-331-2295

Illinois
Intel Corp.
425 N. Martingale Road
Suite 1500
Schaumburg IL
60173
USA
Fax:847-605-9762

Intel Corp.
999 Plaza Drive
Suite 360
Schaumburg IL
60173
USA

Intel Corp.
551 Arlington Lane
South Elgin IL
60177
USA

Indiana
Intel Corp.
9465 Counselors Row,
Suite 200
Indianapolis IN
46240
USA
Fax:317-805-4939

Massachusetts
Intel Corp.
125 Nagog Park
Acton MA
01720
USA
Fax:978-266-3867

Intel Corp.
59 Composit Way
suite 202
Lowell MA
01851
USA

Intel Corp.
800 South Street,
Suite 100
Waltham MA
02154
USA

Maryland
Intel Corp.
131 National Business
Parkway, Suite 200
Annapolis Junction MD
20701
USA
Fax:301-206-3678

Michigan
Intel Corp.
32255 Northwestern
Hwy., Suite 212
Farmington Hills MI
48334
USA
Fax:248-851-8770

MInnesota
Intel Corp.
3600 W 80Th St
Suite 450
Bloomington MN
55431
USA
Fax:952-831-6497

North Carolina
Intel Corp.
2000 CentreGreen Way,
Suite 190
Cary NC
27513
USA
Fax:919-678-2818

New Hampshire
Intel Corp.
7 Suffolk Park
Nashua NH
03063
USA

New Jersey
Intel Corp.
90 Woodbridge Center
Dr, Suite. 240
Woodbridge NJ
07095
USA
Fax:732-602-0096

New York
Intel Corp.
628 Crosskeys Office Pk
Fairport NY
14450
USA
Fax:716-223-2561

Intel Corp.
888 Veterans Memorial
Highway
Suite 530
Hauppauge NY
11788
USA
Fax:516-234-5093

Ohio
Intel Corp.
3401 Park Center Drive
Suite 220
Dayton OH
45414
USA
Fax:937-890-8658

Intel Corp.
56 Milford Drive
Suite 205
Hudson OH
44236
USA
Fax:216-528-1026

Oregon
Intel Corp.
15254 NW Greenbrier
Parkway, Building B
Beaverton OR
97006
USA
Fax:503-645-8181

Pennsylvania
Intel Corp.
925 Harvest Drive
Suite 200
Blue Bell PA
19422
USA
Fax:215-641-0785

Intel Corp.
7500 Brooktree
Suite 213
Wexford PA
15090
USA
Fax:714-541-9157

Texas
Intel Corp.
5000 Quorum Drive,
Suite 750
Dallas TX
75240
USA
Fax:972-233-1325

Intel Corp.
20445 State Highway
249, Suite 300
Houston TX
77070
USA
Fax:281-376-2891

Intel Corp.
8911 Capital of Texas
Hwy, Suite 4230
Austin TX
78759
USA
Fax:512-338-9335

Intel Corp.
7739 La Verdura Drive
Dallas TX
75248
USA

Intel Corp.
77269 La Cabeza Drive
Dallas TX
75249
USA

Intel Corp.
3307 Northland Drive
Austin TX
78731
USA

Intel Corp.
15190 Prestonwood
Blvd. #925
Dallas TX
75248
USA
Intel Corp.

Washington
Intel Corp.
2800 156Th Ave. SE
Suite 105
Bellevue WA
98007
USA
Fax:425-746-4495

Intel Corp.
550 Kirkland Way
Suite 200
Kirkland WA
98033
USA

Wisconsin
Intel Corp.
405 Forest Street
Suites 109/112
Oconomowoc Wi
53066
USA